Transformación y Análisis de Código de Bytes Orientado a Objetos

TESIS DOCTORAL

Memoria presentada para obtener el grado de doctor en Ingeniería Informática por

Miguel Gómez-Zamalloa Gil

Dirigida por la profesora

Elvira Albert Albiol

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Julio de 2009

Transformation and Analysis of Object-Oriented Bytecode

PhD THESIS

Miguel Gómez-Zamalloa Gil
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Advisor: Elvira Albert Albiol

July, 2009

Resumen

Predecir el comportamiento de los programas antes de su ejecución es cada vez más importante, especialmente teniendo en cuenta que éstos son cada vez más complejos y son utilizados frecuentemente en situaciones críticas, como operaciones médicas, control aéreo u operaciones bancarias. El *análisis estático de programas* es el proceso por el cual el comportamiento de los programas es analizado sin llegar a ejecutar su código. Tradicionalmente, la mayoría de análisis han sido formulados al nivel del código fuente. No obstante, puede darse el caso de que el análisis deba tratar con código compilado, o código de bytes. Esta situación se da en particular cuando un consumidor de código está interesado en verificar ciertas propiedades de programas de un tercero, pero no tiene acceso directo al código fuente, como suele pasar con el software comercial y con el código móvil. Un ejemplo particularmente interesante es el *análisis del consumo de memoria*, el cual puede ser muy útil en contextos en los cuales el consumidor de código quiere verificar que el programa recibido puede ejecutarse sin que su consumo de memoria exceda un límite dado.

Desafortunadamente, razonar sobre programas reales de código de bytes (con orientación a objetos) es una tarea complicada y costosa. Además de las características propias de la orientación a objetos como la herencia y las invocaciones virtuales, un analizador de código de bytes tiene que tratar con ciertas complicaciones propias de los lenguajes de bajo nivel como la ausencia de estructura de control, el uso de la pila de operandos, etc. Una práctica habitual consiste en resolver el problema en dos pasos, de forma que en primer lugar se transforma, o *decompila*, el programa de código de bytes a una *representación intermedia* de más alto nivel, para poder así formular el análisis sobre dicha representación. Esto permite abstraer las características particulares del lenguaje y así poder desarrollar las

herramientas de análisis sobre representaciones más sencillas. La mayoría de los enfoques desarrollan decompiladores *ad hoc*, es decir, decompiladores exclusivamente diseñados para llevar a cabo una transformación particular. Existe no obstante una alternativa al desarrollo de decompiladores ad hoc, llamada *decompilación interpretativa* por *evaluación parcial*. Como veremos, ésta permite decompilar programas evaluando parcialmente un intérprete respecto a éstos.

Esta tesis contribuye a mejorar el estado del arte en la transformación y el análisis de lenguajes de código de bytes, en concreto: (1) proponiendo e implementando un esquema formal para la decompilación automática por compilación interpretativa de programas de código de bytes (con orientación a objetos) a representaciones intermedias de más alto nivel, en particular utilizando programación lógica; (2) estudiando las aplicaciones prácticas que se tienen gracias a disponer de dichas representaciones; y (3) diseñando e implementando un análisis de consumo de memoria para lenguajes de código de bytes con *recolección de basura*.

Abstract

Predicting the behavior of programs before their actual execution becomes more and more relevant as programs increase in complexity and become used in critical situations such as medical operations, flight control or banking cards. *Static program analysis* is the process of automatically analyzing the behavior of programs without actually executing the code. Traditionally, most analyses have been formulated at the source code level. However, it can be the case that the analysis must consider the compiled code, or bytecode, instead. This may happen, in particular, when the code consumer is interested in verifying some properties of 3rd party programs, but has no direct access to the source code, as usual for commercial software and in mobile code. A particularly interesting example is *memory consumption analysis*, which can be very useful in contexts where the code consumer wants to verify that the received program can run within the actual memory available.

Unfortunately, reasoning about realistic (object-oriented) bytecode programs is rather complicated and time consuming. In addition to the object-oriented features such as inheritance and virtual method invocations, a bytecode analyzer has to deal with several low-level language features like the unstructured control flow, the usage of the operand stack, etc. A usual practice is to first transform, or *decompile*, the bytecode program into a higher-level *intermediate representation*, and then develop the analysis over such representation. This allows abstracting away the particular bytecode language features and developing the analysis tools on much simpler representations. Most of the approaches develop *ad-hoc* decompilers, i.e., decompilers exclusively designed to carry out the particular transformation. There is however an alternative to the development of dedicated decompilers which is the so called *interpretive decompilation* by *partial evaluation*,

which allows decompiling programs by partially evaluating an interpreter w.r.t. them.

This thesis contributes to improve the state-of-the-art in the transformation and analysis of bytecode languages by: (1) providing and implementing a formal framework for the automatic decompilation of (object-oriented) bytecode programs to higher-level intermediate representations, in particular represented using logic programming, by means of interpretive decompilation; (2) studying the practical applications that having such representations can have; and (3) designing and implementing a live memory consumption analysis for bytecode languages with *garbage collection*.

Agradecimientos

La realización de esta tesis no habría sido posible sin la ayuda y el apoyo de mucha gente, y me gustaría aprovechar esta oportunidad para expresarles mi más profunda gratitud.

En primer lugar, me gustaría dar las gracias a mi directora, Elvira Albert, por introducirme en el mundo de la investigación, y por su ayuda incalculable con la tesis. Agradezco su paciencia, sentido del humor y todo el ánimo que he recibido desde el principio. Quiero agradecer también a Germán Puebla sus valiosos consejos y todas las cosas que me ha enseñado (algunas voluntariamente y otras simplemente gracias a su manera de trabajar día a día). En tercer lugar, quiero dar las gracias a toda la gente del equipo COSTA, por crear ese magnífico ambiente de trabajo, especialmente Samir, por estar siempre ahí, Puri por ese sentido del humor, Damiano, etc.

Aprovecho también para dar las gracias a toda la gente del grupo CLIP, especialmente a Edison y a Claudio, por su ayuda solucionando esos problemas técnicos que solía tener con Linux, con las instalaciones de Ciao, etc, etc. Una mención especial es para el director del grupo CLIP, Manuel Hermenegildo, a quien siempre estaré muy agradecido por habernos traído y enseñado el verdadero mundo de la investigación.

Una parte importante de mi formación durante estos años (como investigador y como persona) ha sido posible gracias a las distintas estancias de investigación y visitas que he podido realizar. Quiero por tanto agradecérselo a toda la gente involucrada, desde Paco López, por haberlo hecho posible, hasta toda la gente que me ha acogido y ayudado en mis diferentes destinos. En particular, John Gallagher por ser siempre tan cercano (nunca olvidaré aquel partido R. Madrid - Sevilla en su casa), Gourinath Banda por su ayuda incalculable en Roskilde (la bici, esas pizzas indias, etc,

etc), Michael Leuschel por aquellos días charlando sobre evaluación parcial, Jorge Pérez por su ayuda en Bolonia, Roberto Bagnara por ese magnífico día en Parma, y Andy King por su impresionante dedicación (aquello fue investigación pura, ya casi había olvidado lo que era).

Quiero también dar las gracias a la gente de mi departamento en la Universidad Complutense por ayudar a crear ese magnífico ambiente de trabajo que tengo la suerte de disfrutar. En particular, a Ana Gil por guiarme, a Teresa Martínez por toda la ayuda con la burocracia, al despacho 220 (el futuro de nuestro departamento!), etc, etc.

Finalmente, esta tesis no habría sido posible sin el apoyo y ayuda de mi familia y amigos, especialmente mi mujer, Ali, y mi madre.

Acknowledgments

The making of this thesis would not have been possible without the help and support of many people and I would like to take this opportunity to express my gratitude to all of them.

First, I would like to thank my advisor, Elvira Albert, for introducing me to the world of research, and for her invaluable help with this thesis. I am deeply grateful for her patience with me, her sense of humor and all the encouragement I received from the beginning. Secondly, I want to thank Germán Puebla for his valuable comments, suggestions, and for all the things he has taught me (some on purpose and some others just because of the way he works day by day). Thirdly, I also thank the people of the COSTA team, for creating such an amazing working environment, especially Samir for being always there, Puri for her sense of humor, Damiano, etc.

I also would like to thank all the people in the CLIP lab, especially Edison and Claudio, for their help in sorting out the technical problems I used to have with Linux, the Ciao installations, etc, etc. A special mention goes to its director, Manuel Hermenegildo, to whom I will be always very grateful for bringing the real research world to us.

An important part of my formation during these years (as a researcher and as a person) has been possible thanks to the several research stays and visits I have been able to do. I therefore want to thank all the people involved, from Paco López, for making them possible, to the people hosting and helping me in my different destinations. In particular, I thank John Gallagher for being always so accessible (I will never forget that R. Madrid vs. Sevilla match at his place), Gourinath Banda for his invaluable help (the bike, those indian pizzas, etc, etc), Michael Leuschel for those days of discussions about PE, Jorge Pérez for his help in Bologna, Roberto Bag-

nara for that wonderful day in Parma, and Andy King for his impressive dedication (that was pure research, I had almost forgot what was that).

I also would like to thank the people in my department at the Complutense University for creating this wonderful working environment, in particular I thank Ana Gil for guiding me, Teresa Martínez for her help with all the bureaucracy, the 220 office (the future of the department!), etc, etc.

Finally, the completion of this thesis would not have been possible without the support of my family and friends, especially my wife Ali and my mom.

Índice general

Parte I

Versión en Castellano (Spanish Version)

Capítulo 1

Introducción: Motivación y Contribuciones

1.1. Lenguajes de Código de Bytes

Los lenguajes de programación pueden categorizarse, en general, de acuerdo al modelo de ejecución en el que sus programas se ejecutan. En este sentido, se clasifican en una de estas dos categorías: *compilados* o *interpretados*. En los lenguajes compilados, el *código fuente* es primer lugar traducido, o compilado, a un conjunto de instrucciones específicas de hardware, normalmente conocido como *código objeto*. El programa es entonces ejecutado corriendo el código objeto en el hardware correspondiente. Por el contrario, en los lenguajes interpretados, el código fuente se ejecuta directamente en un intérprete. Esta distinción aplicada a lenguajes de programación es algo confusa, pues en principio, cualquier lenguaje podría ser compilado o interpretado. La categorización, por tanto, refleja habitualmente el modelo de ejecución más popular del lenguaje en cuestión y no sus propiedades. Cada una de las alternativas tiene sus propias ventajas y desventajas. Por ejemplo, ejecutar un programa objeto en la máquina correspondiente, tiende a ser mucho más rápido que ejecutar el programa fuente usando un intérprete del lenguaje en cuestión (aproximadamente en un ratio de 10:1). Por otro lado, los lenguajes interpretados proporcionan cierta flexibilidad respecto a los lenguajes compilados, por ejemplo, facilidad de implementación, facilidad de depuración, y lo más importante, independencia de plataforma.

```
void foo(int n,int m){
    this.f = n + m;
}
```

```
void foo(int,int)
  0: aload_0        3: iadd
  1: iload_1        4: putfield f
  2: iload_2        7: return
```

Figura 1.1: Programa Java Bytecode de ejemplo

Una combinación de ambos enfoques, conocida como *compilación a código de bytes* o *interpretación de código de bytes*, está siendo ampliamente utilizada. Los modelos de ejecución basados en compilación a *código de bytes*, traducen primeramente el código fuente a una representación intermedia, conocida como *código de bytes* (en inglés "bytecode"). Por brevedad, utilizaremos el término "bytecode" a partir de ahora. El bytecode no es el código máquina para ninguna máquina en particular, y puede ser portable entre diferentes arquitecturas. El bytecode es entonces interpretado, o ejecutado en una *máquina virtual*. El término bytecode proviene de los repertorios de instrucciones en los que éstas incluyen un código de operación de un byte de longitud seguido de una serie de parámetros opcionales. Las instrucciones bytecode son habitualmente similares a las instrucciones hardware tradicionales. Así por ejemplo, los lenguajes bytecode tienen un flujo de control desestructurado con varios tipos de saltos (condicionales e incondicionales) y utilizan habitualmente una pila de operandos para realizar cálculos auxiliares. Sin embargo, el hecho de que las instrucciones bytecode estén en principio pensadas para ser ejecutadas por software, hace que éstas tengan en ocasiones cierta complejidad, especialmente en el caso de lenguajes bytecode orientados a objetos y declarativos. Para hacerse una idea, la Figura 1.1 muestra el código fuente (Java) y el correspondiente bytecode de un método que toma dos números enteros, los suma, y asigna el resultado al atributo `f` del objeto *this*. Nótese que, en el bytecode de Java, el objeto *this* se pasa explícitamente en la variable local 0. Por tanto, la instrucción `aload_0`, apila la referencia al objeto *this*, en la cima de la pila de operandos.

En cuanto a eficiencia, se puede decir que el modelo de compilación a bytecode, se encuentra en algún punto entre el modelo puro basado en

compilación y el basado en interpretación; mientras que mantiene las ventajas de los modelos basados en interpretación, en particular, la independencia de plataforma. Más aún, nada obliga a un lenguaje bytecode a ser exclusivamente interpretado. De hecho, la compilación *just-in-time* (JIT) puede usarse para acelerar la ejecución del bytecode. Los compiladores JIT convierten el código fuente, o bytecode en este caso, a código nativo gradualmente durante la ejecución del programa, obteniéndose así un mejor rendimiento. La compilación a bytecode junto con la compilación JIT puede por tanto combinar la mayoría de las ventajas de los modelos de ejecución basados en compilación y los basados en interpretación. Ésta es la principal razón del éxito de los entornos de programación de *Microsoft .NET* y *Java*, los cuales son, sin lugar a dudas, los entornos de programación más utilizados en la actualidad.

Java Bytecode *Java Bytecode* es el lenguaje que la máquina virtual de Java (JVM) [66] ejecuta. Fue originalmente diseñado por *Sun Microsystems* como un lenguaje intermedio en el entorno de desarrollo de Java. Una instrucción Java Bytecode consiste en un código de operación, el cual especifica la operación a ser ejecutada, seguido de cero o más operandos con los valores que se utilizan en la operación en cuestión. La JVM utiliza, entre otras, las siguientes estructuras de datos, que las instrucciones manipulan durante su ejecución: el *contador de programa*, que contiene el índice de la instrucción actual, la *pila de operandos* y el *array de variables locales*, en los cuales se almacenan los parámetros, variables y resultados intermedios, el *montículo* o "heap" (utilizaremos el término "heap" a partir de ahora), en el cual se almacenan los objetos y arrays, y la habitual *pila de llamadas* o *pila de "frames"* para tratar con las llamadas y vueltas de llamada a métodos. El lenguaje Java Bytecode incluye, por un lado, las instrucciones habituales de bajo nivel para: transferir valores entre la pila de operandos y el array de variables locales (y viceversa), realizar operaciones aritméticas, saltar condicional o incondicionalmente a otras partes del código, llamar y volver de métodos, etc. Por ejemplo, la instrucción "`iload_1`" carga la variable local 1 en la cima de la pila de operandos, y la instrucción "`iadd`" suma (y desapila) los dos valores de la cima de la pila, y apila el resultado. Por otro lado, Java Bytecode, al tener orientación a objetos y concurrencia, incluye también instrucciones para: crear objetos y arrays, escribir y leer

atributos y elementos de arrays, realizar invocaciones virtuales, adquirir y liberar monitores, etc. Por ejemplo, la instrucción "`putfield f`" escribe el valor que hay en la cima de la pila, en el atributo `f` del objeto referenciado por la dirección de memoria almacenada bajo la cima de la pila.

Aunque Java es el lenguaje más común que se compila a Java Bytecode, hay sin embargo muchos compiladores de diferentes lenguajes de alto nivel a Java Bytecode. Algunos de los más conocidos son: *jython* para programas *Python*, *jRuby* para *Ruby* and *jGNAT* para *Ada*.

El Lenguaje Intermedio Común de .NET El *Lenguaje Intermedio Común* ("*Common Intermediate Language*" o CIL) es el lenguaje bytecode intermedio utilizado en el entorno .NET. Así, los diferentes lenguajes fuente utilizados en .NET, se compilan a CIL. Como Java Bytecode, CIL es un lenguaje orientado a objetos y basado en pila. Incluye por tanto la misma clase de instrucciones bytecode. A diferencia de Java Bytecode, CIL no está pensado para ser interpretado. Fue sin embargo, desde sus comienzos, pensado para ser compilado a código máquina utilizando compilación JIT. Incluso, en ocasiones, el bytecode se compila por completo a código máquina antes de ser ejecutado para mejorar el rendimiento. El entorno .NET es una de las piedras angulares de la tecnología moderna de Microsoft, y se utiliza para el desarrollo de la mayoría de aplicaciones creadas para la plataforma Windows.

Existen otros muchos lenguajes bytecode bien conocidos y ampliamente utilizados, tanto imperativos, como el *p-code* utilizado en algunas implementaciones de Pascal; como declarativos, como el bytecode *WAM*, utilizado en la mayoría de implementaciones de Prolog, el bytecode de *Haskell Hugs'98*, o el bytecode *Erlang BEAM*, por nombrar algunos.

Esta tesis está principalmente centrada en lenguajes bytecode imperativos y con orientación a objetos. En particular, como veremos, los diferentes contenidos técnicos de la tesis, así como los distintos prototipos implementados, consideran subconjuntos representativos de Java Bytecode.

1.2. Análisis Estático de Programas

Predecir el comportamiento de los programas antes de su ejecución es cada vez más relevante, especialmente teniendo en cuenta que éstos son cada vez más complejos y son frecuentemente utilizados en situaciones críticas, como operaciones médicas, control aéreo o tarjetas bancarias. Ser capaces de demostrar de forma automática que los programas cumplen con sus especificaciones funcionales, es un factor básico para su éxito. El *análisis estático de programas* es el proceso por el cual el comportamiento de los programas es analizado sin llegar a ejecutar su código. Por el contrario, cuando el análisis se realiza ejecutando el programa, éste se denomina *análisis dinámico*. Los análisis estáticos clásicos tratan de inferir propiedades de los programas como: *ausencia de errores*, *terminación*, *coste* o *consumo de recursos* (tiempo o memoria), *vida de variables*, *forma de punteros*, etc. Habitualmente, los análisis estáticos basan su funcionamiento en métodos formales. Algunos de los más habituales son: la *interpretación abstracta*, el *chequeo de modelos* y los *sistemas de tipos*. Esta tesis está basada principalmente en el análisis estático basado en interpretación abstracta.

Interpretación Abstracta. La técnica de la interpretación abstracta [30] proporciona un marco general para computar aproximaciones seguras (es decir, abstracciones) del comportamiento de los programas. Su principal aplicación práctica es el análisis estático formal. Los analizadores basados en interpretación abstracta, infieren información de los programas interpretándolos ("ejecutándolos"), utilizando valores abstractos en lugar de valores concretos. Estos analizadores son paramétricos respecto al llamado *dominio abstracto*, el cual proporciona una representación finita de un conjunto posiblemente infinito de valores. Dominios diferentes capturan clases distintas de propiedades, con un nivel distinto de precisión, y a un coste computacional diferente.

Los analizadores basados en interpretación abstracta se han estudiado tanto en el contexto de lenguajes declarativos como en el de lenguajes imperativos. A continuación enumeramos algunos sistemas de análisis:

El Analizador ASTRÉE. *ASTRÉE* [31] es un analizador estático de programas, desarrollado en la *École Normale Supérieure* por Cousot *et. al.*, que es capaz de demostrar ausencia de errores en tiempo de ejecución en programas C. ASTRÉE fue capaz por ejemplo de demostrar, de forma totalmente automática, la ausencia de errores en el software primario de control aéreo del Airbus A340, un programa de 132.000 líneas.

El Sistema CiaoPP. CiaoPP [49] es el preprocesador basado en interpretación abstracta del sistema de *Programación Lógica con Restricciones*, "Constraint Logic Programming" (CLP), Ciao-Prolog [25]. Éste incluye un buen número de funcionalidades para realizar depuración, análisis y transformación *fuente a fuente* de programas Ciao-Prolog. Algunas de las propiedades que el sistema es capaz de inferir son: *tipos*, *modos* y otras propiedades de instanciación de variables, *no fallo*, *determinismo*, *cotas* en el *coste de recursos*, cotas del tamaño de los términos del programa, etc. CiaoPP es también capaz de realizar varios tipos de transformaciones fuente a fuente de programas, como *especialización* de programas, *paralelización* de programas (incluyendo *control de granularidad*), etc.

Otros sistemas de análisis estático bien conocidos (no comerciales y comerciales) son: *Lint*, *CCA* y *BOON* para programas C, *CodeSonar* para C++, *Fluid* y *jLint* para Java, y muchos otros. Otros analizadores estáticos no han llegado a ser herramientas autointegradas sino que aparecen integrados en diversos compiladores. Un ejemplo de esto es el verificador de la JVM que integra un analizador del flujo de datos ("data-flow analysis").

Tradicionalmente, la mayoría de análisis han sido formulados al nivel del código fuente. No obstante, puede darse el caso de que el análisis deba tratar con código compilado, o bytecode. Esta situación se da en particular cuando un consumidor de código está interesado en verificar ciertas propiedades de programas de un tercero, pero no tiene acceso directo al código fuente, como suele pasar con el software comercial y con el código móvil. Éste es el marco general en el que nació la idea del *Código con demostración*, "Proof-Carrying Code" [74]: para que un usuario pueda verificar cierto código, éste debe venir acompañado de una *demostración* de que se cumplen ciertas propiedades de seguridad, referidas al código compilado o bytecode (posiblemente inferida por análisis estático), de forma que el usuario pueda

chequear la corrección de la demostración proporcionada y verificar que las propiedades efectivamente se cumplen (por ejemplo, que el código no requiere más de una cierta cantidad de memoria para ser ejecutado, o que ejecuta en menos de una cierta cantidad de tiempo).

Existe por tanto la necesidad de desarrollar herramientas de análisis y verificación que trabajen directamente al nivel de programas bytecode. Desafortunadamente, razonar sobre programas reales bytecode (con orientación a objetos) es una tarea complicada y costosa. Además de las características propias de la orientación a objetos como la herencia y las invocaciones virtuales, un analizador de bytecode tiene que tratar con ciertas complicaciones propias de los lenguajes de bajo nivel como la ausencia de estructura de control, el uso de la pila de operandos, etc.

1.3. Del Bytecode a Representaciones Intermedias

En el contexto del análisis de lenguajes bytecode, una práctica habitual consiste en resolver el problema en dos pasos: (1) transformar el programa bytecode a una *representación intermedia* (RI) de más alto nivel, y (2) formular el análisis sobre dicha RI. Esto permite abstraer las características particulares del lenguaje bytecode y así poder desarrollar las herramientas de análisis sobre representaciones más sencillas de tratar. Otra ventaja importante de este enfoque, es que éste permite la posibilidad de reutilizar la fase de análisis (fase (2)) para poder analizar distintos lenguajes (bytecode y no bytecode), siempre que éstos puedan ser transformados a la misma RI. Utilizaremos a partir de ahora el término *decompilación* para referirnos a la transformación de bytecode a la RI, pues se traduce un lenguaje de bajo a alto nivel.

La mayoría de los enfoques desarrollan decompiladores *dedicados*, o *ad hoc*, es decir, decompiladores exclusivamente diseñados para llevar a cabo una decompilación particular. Existe no obstante una alternativa al desarrollo de decompiladores dedicados, llamada *decompilación interpretativa* por *evaluación parcial*. Como veremos, ésta permite decompilar programas evaluando parcialmente un intérprete respecto a éstos.

Evaluación Parcial. La *Evaluación Parcial* (EP) [56] es una técnica, basada en semántica, de transformación fuente a fuente de programas, que permite especializar programas respecto a parte de sus datos de entrada. Se suele llamar de hecho *especialización de programas*. Consideremos un programa P, y sus datos de entrada I divididos en I_{static} y $I_{dynamic}$. I_{static} son los datos estáticos, es decir, los datos conocidos en tiempo de compilación, y $I_{dynamic}$ son el resto de los datos. Podemos ver la ejecución del programa P como una función de los datos de entrada a los de salida de la siguiente forma:

$$P : I_{static} \times I_{dynamic} \longrightarrow O$$

Un evaluador parcial trasforma el par $\langle P, I_{static} \rangle$ en $P' : I_{dynamic} \longrightarrow O$, realizando las computaciones de P que dependen de I_{static} en tiempo de compilación. Se denomina a P' el *programa residual*, el cual debería ser más eficiente que el programa original P.

El Enfoque Interpretativo de Compilación. Una aplicación particularmente interesante de la EP, primeramente descrita por Yoshihiko Futamura en los años 70 [41], aparece cuando el programa P a ser evaluado parcialmente es un intérprete de un lenguaje de programación. Esto se conoce como el *enfoque interpretativo de compilación* o la *primera proyección de Futamura*. Asumamos un intérprete, escrito en un lenguaje *objetivo* o "target" L_T, que interpreta programas escritos en un lenguaje fuente o "source", L_S. Entonces, si I_{static} es un programa fuente, escrito en L_S, la evaluación parcial del intérprete respecto al programa (datos), producirá P', una versión del intérprete que sólo puede ejecutar ese programa fuente, que está escrita en el lenguaje de implementación del intérprete, L_T, y que no necesita el código fuente para poder interpretarlo. P' puede considerarse como una versión compilada de I_{static} al lenguaje objetivo L_T. La compilación interpretativa permite por tanto compilar programas escritos en L_S a otro lenguaje L_T, evaluando parcialmente un intérprete de L_S escrito en L_T respecto a ellos.

En el caso particular de la decompilación de lenguajes bytecode, el enfoque interpretativo de compilación nos permitiría decompilar un programa bytecode escrito en un lenguaje bytecode BC, a una representación

de más alto nivel, digamos que al lenguaje HL, evaluando parcialmente un intérprete de BC escrito en HL. Este enfoque es, en principio, más genérico, flexible, más seguro y más fácil de mantener que un decompilador dedicado para la misma tarea. Dichas ventajas serán discutidas más adelante en la Sección 2. El enfoque interpretativo, aunque es en principio muy atractivo, no ha sido muy utilizado en la práctica, principalmente debido a la dificultad de encontrar estrategias de EP con las cuales se puedan obtener decompilaciones efectivas, o de calidad, y de forma eficiente.

1.4. Análisis de Consumo del Heap para Bytecode

La investigación sobre el consumo de recursos de los programas comenzó con el trabajo de Wegbreit es 1975 [86], en el cual se propuso un análisis del rendimiento de un programa basado en la derivación de una expresión matemática que representaba su comportamiento en tiempo de ejecución. El enfoque para realizar análisis estático de coste es el siguiente: dado un programa de entrada, (1) en una primera fase, el análisis de coste genera un sistema de ecuaciones de coste, "cost equation system" (CES) a partir del programa, que captura las relaciones entre las diferentes partes del código. Un CES es un conjunto de ecuaciones de recurrencia que expresa el coste del programa en términos de los tamaños de sus argumentos de entrada. (2) En la segunda fase, el CES se trata de resolver o aproximar, típicamente utilizando técnicas algebraicas, obteniéndose una *forma cerrada* (por ejemplo, sin incluir recurrencias) que representa una *cota superior* (o *cota inferior*) del coste.

Los análisis de coste se han estudiado de forma intensiva en el contexto de la programación declarativa, tanto para programación funcional [79, 80, 44, 15], como para programación lógica [34, 35], y también en el contexto de lenguajes imperativos de alto nivel (centrándose principalmente en la estimación de tiempos en el *caso peor* y en el diseño de *modelos de coste* [88]). Tradicionalmente, como pasa con la mayoría de análisis estáticos, los análisis de coste han sido formulados al nivel del código fuente. Sin embargo, como hemos visto, existen situaciones en las que no se tiene acceso a éste, sino que sólamente se tiene acceso al código compilado o al

bytecode. Recientemente, en [2] se ha propuesto un esquema genérico para el análisis de coste de Java Bytecode, el cual constituye la base formal sobre la que se sustenta el sistema COSTA [5].

El Sistema COSTA**.** COSTA [5] es un prototipo de investigación que es capaz de realizar automáticamente **A**nálisis de **COS**te y **T**erminación para Java Bytecode. El sistema recibe como entrada un programa bytecode y un modelo de coste, elegido a partir de una selección de descripciones de recursos, y trata de obtener una cota del consumo de recursos del programa respecto al modelo de coste dado. COSTA sigue el enfoque estándar para realizar el análisis de coste, es decir, primeramente produce un CES, el cual es una forma extendida de relaciones de recurrencia, y después trata de obtener una forma cerrada, que representa una cota superior del coste del programa, utilizando para ello un resolutor de ecuaciones de recurrencia propio [9].

Una de las aplicaciones más interesantes del análisis de coste, la cual presenta importantes retos por resolver, es el análisis de consumo del heap. Éste trata de inferir cotas del espacio de heap consumido por un programa. De nuevo, los análisis de heap se han formulado habitualmente al nivel del código fuente (ver por ejemplo [83, 50, 85, 54] en el contexto de la programación funcional y [52, 23] para lenguajes imperativos de alto nivel). En el contexto de lenguajes bytecode, el análisis de consumo de heap puede tener aplicaciones muy interesantes. Por ejemplo, la *certificación de cotas de recursos*, "resource bound certification" [33, 8, 10, 51, 22], propone utilizar propiedades de seguridad incluyendo requerimientos de coste, es decir, el código recibido ha de adherirse a unos requerimientos específicos respecto a su consumo de memoria. También, las cotas del consumo del heap pueden resultar útiles en sistemas empotrados ("embedded systems"), por ejemplo, en tarjetas inteligentes en las cuales la memoria es limitada y no puede recuperarse de forma sencilla.

Desafortunadamente, la gestión automática de memoria, también llamada *recolección de basura* ("garbage collection" o GC), la cual es utilizada cada vez más habitualmente en lenguajes bytecode como en Java Bytecode y en el .NET CIL, provoca que el problema de predecir la memoria utilizada por un programa sea mucho más difícil. Una primera aproximación al

problema es inferir el *consumo total* de memoria, es decir, la cantidad acumulada de memoria alojada por el programa ignorando el efecto del GC. Si se dispone de dicha cantidad de memoria, está asegurado que el programa puede ejecutar, incluso aún cuando no se aplica el GC. Sin embargo, ésta es una estimación muy pesimista del consumo real del programa. Recientemente, en [83, 18, 24] se han propuesto *análisis de consumo del heap activo*, "live heap space analysis", los cuales tratan de aproximar el tamaño de la memoria *activa* en el heap durante la ejecución del programa, resultando en una aproximación mucho más precisa. Dichos enfoques, están sin embargo restringidos a cotas polinomiales y a métodos no recursivos [18], o a cotas lineales, en este caso, capaz de tratar recursión [24].

1.5. Objetivos y Contribuciones

El principal objetivo de esta tesis es mejorar el estado del arte en la transformación y el análisis de lenguajes bytecode, en concreto: (1) proponiendo e implementando un esquema formal para la decompilación automática por compilación interpretativa de programas bytecode (con orientación a objetos) a representaciones intermedias de más alto nivel, en particular utilizando programación lógica (LP); (2) estudiando las aplicaciones prácticas que se tienen gracias a disponer de dichas RIs basadas en LP; y (3) diseñando e implementando un análisis de consumo de la memoria activa para lenguajes bytecode con recolección de basura. Más detalladamente, las contribuciones de esta tesis son las siguientes:

1. **Decompilación interpretativa de bytecode a LP:** Ha habido en la literatura varias *pruebas de concepto* mostrando que el enfoque interpretativo es factible [63, 48, 76, 64]. Sin embargo, en la práctica, a la hora de decompilar lenguajes y programas complejos, aún quedan varias cuestiones por resolver. Éstas incluyen: *escalabilidad*, que a su vez depende de la *composicionalidad* del enfoque, y *efectividad*, es decir, obtener programas decompilados de calidad. Esta tesis presenta, por lo que conocemos, el primer esquema de decompilación interpretativa que es capaz de decompilar lenguajes bytecode reales a una representación de alto nivel. En particular, decompilamos Java Bytecode a Prolog.

a) **Estrategias de control:** Una de las principales dificultades de la decompilación interpretativa, y de la EP en general, es el ser capaz de tratar adecuadamente con signaturas infinitas. Hemos propuesto técnicas novedosas que nos han permitido definir reglas de control sofisticadas. En particular, hemos introducido la relación de la *subsunción homeomórfica basada en tipos*, una generalización de la relación original de *subsunción homeomórfica*, que proporciona resultados más precisos en presencia de signaturas infinitas. Hemos mostrado como esta técnica, a parte de resultar crucial en la especialización de intérpretes, mejora el estado del arte de las herramientas de especialización "online". Este trabajo fue primeramente propuesto en el Artículo 3 (ver el Apéndice A) y posteriormente extendido y reformulado en el Artículo 4, el cual ha sido publicado en la revista "*Information Processing Letters*".

b) **Controlando la *polivarianza* de la EP:** Incluso una vez después de haber integrado la *subsunción homeomórfica basada en tipos* en la EP, los programas decompilados que se obtienen tienden a tener demasiadas versiones especializadas (redundantes) para algunos predicados. Este aspecto se ha estudiado en el Artículo 2, donde se proponen técnicas avanzadas para controlar la *polivarianza* del proceso de EP, es decir, evitar tener dichas versiones especializadas redundantes.

c) **Cómo escribir el intérprete de bytecode:** Como se ha mostrado en trabajos previos de compilación interpretativa, las características del intérprete en cuestión, resultan cruciales para poder obtener una especialización exitosa. Hemos identificado los aspectos necesarios que el intérprete debe tener para poder obtener un esquema composicional de decompilación.

d) **Decompilación óptima:** Aseguramos la calidad de la decompilación, tanto en términos de efectividad como de eficiencia, proponiendo una serie de *criterios de optimalidad*. Éstos básicamente requieren que: (1) la decompilación no genere código más de una vez para cada punto de programa, y (2) que haya como máximo una regla residual para cada bloque del programa byte-

code. Proponemos un esquema de decompilación que es *óptimo* respecto a estos criterios de optimalidad. Esto asegura tanto la escalabilidad del proceso como la calidad de las decompilaciones. Este trabajo junto con lo descrito en el punto (c) dieron lugar al Artículo 5.

e) **Tratando con la orientación a objetos:** Mostramos como nuestro esquema se puede adaptar fácilmente para tratar las características de la orientación a objetos. En particular, proponemos mecanismos para: tratar con el heap y con sus instrucciones asociadas, representar clases por medio de módulos Prolog, y representar invocaciones virtuales por medio de llamadas Prolog con calificación de módulo.

f) **Implementación y evaluación experimental:** Todas las técnicas mencionadas han sido implementadas e integradas en un decompilador prototipo de Java Bytecode secuencial a Prolog, llamado jbc2prolog. Presentamos resultados experimentales usando dicho prototipo (utilizando, y contrastando con, otros sistemas). En particular, se han estudiado tanto la escalabilidad como la eficiencia de nuestro enfoque, utilizando el conjunto de "benchmarks" JOlden [55]. El trabajo descrito en los puntos (b), (c), (d), (e) y (f), ha dado lugar al Artículo 6, el cual ha sido recientemente publicado por la revista "*Information and Software Technology*". Este Artículo, por tanto, lleva a cabo el objetivo (1) (ver más arriba).

2. **Aplicaciones de la decompilación interpretativa:** Utilizar un lenguaje declarativo para definir las RIs ofrece ventajas importantes. En particular, se pueden reutilizar las potentes y avanzadas herramientas de análisis y transformación de programas existentes en el contexto de la programación declarativa (ya probadas correctas y efectivas) para el análisis y transformación de programas bytecode. Este trabajo se corresponde por tanto con el objetivo (2).

   a) **Reutilizando herramientas de análisis de LP:** Utilizando el sistema `CiaoPP` sobre nuestros programas decompilados, hemos sido capaces de demostrar ciertas propiedades no triviales de programas Java Bytecode como terminación y *ausencia de*

errores, así como, para programas sencillos, inferir cotas de su consumo de recursos. Este trabajo aparece en el Artículo 1.

b) **Generación de datos de prueba:** Uno de los enfoques estándar para generar automáticamente datos de prueba consiste en hacer *ejecución simbólica* de los programas [29, 57]. En ésta, los contenidos de las variables son expresiones en lugar de valores concretos. El hecho de que nuestros programas decompilados sean programas Prolog *ejecutables*, nos permite poder utilizar técnicas inherentes a la CLP (como el *backtracking* y la manipulación de restricciones) para realizar la ejecución simbólica. Hemos desarrollado un esquema novedoso de *generación de casos de prueba* utilizando nuestros programas (C)LP decompilados. Mostramos además como la fase de generación de casos de prueba en CLP, puede verse como otra EP, lo que nos permite obtener no sólo casos de prueba, sino también *generadores de casos de prueba*. Este trabajo ha dado lugar al Artículo 7. Como contribución tangencial, hemos aplicado la misma idea para generar automáticamente casos de prueba para Prolog. Un estudio preliminar en esta dirección aparece en el Artículo 8.

3. **Análisis de consumo del heap:**

a) **Consumo total del heap:** En primer lugar, hemos desarrollado una aplicación novedosa del analizador de coste presentado en [3], para inferir cotas superiores del consumo del heap de programas secuenciales Java Bytecode. Para ello, simplemente hemos propuesto un modelo de coste que define el coste de las instrucciones que alojan memoria, en términos de las unidades de heap (memoria) que consumen. Podemos entonces generar *relaciones de coste* del consumo del heap que pueden ser utilizadas directamente para inferir cotas superiores en el consumo del heap de programas Java Bytecode.

b) **Análisis de consumo del heap *activo* en lenguajes con GC:** En presencia de recolección automática de basura, el enfoque propuesto proporciona estimaciones demasiado pesimistas del consumo real de los programas. Esta tesis presenta un esquema general para inferir el consumo *pico* del heap durante la

ejecución de un programa bytecode, es decir, el máximo uso de memoria *activa* durante su ejecución, no estando restringido a ninguna clase de complejidad (como pasa con enfoques anteriores).

c) **Implementación y evaluación experimental:** Los análisis han sido implementados e integrados en el sistema COSTA. Hemos realizado una evaluación experimental con una serie de aplicaciones Java que hacen un uso intensivo del heap, incluyendo los "benchmarks" JOlden [55]. Los resultados demuestran que nuestro enfoque es capaz de obtener cotas del consumo de heap activo razonablemente precisas de forma totalmente automática. Todo este trabajo sobre el análisis de consumo del heap ha dado lugar a los Artículos 9 y 10, llevándose a cabo por tanto el objetivo (3).

1.6. Organización de la Tesis

Esta tesis sigue el formato "tesis por publicaciones" y consiste por tanto en una introducción describiendo sus principales objetivos, contribuciones y conclusiones, la cual se presenta en los Capítulos 1, 2, 3, 4 y 5, y, el conjunto de publicaciones editadas que han dado lugar a la tesis, presentadas en el formato y longitud en el que aparecen en las correspondientes publicaciones, como un apéndice.

El resto de la tesis se organiza de la siguiente manera: El Capítulo 2 ofrece una visión general de todo el trabajo correspondiente a la contribución (1). En particular, la Sección 2.1 proporciona resumidamente los fundamentos básicos de la EP de programas lógicos, después, se presentan en la Sección 2.2 los retos que aparecen al especializar un intérprete de bytecode, la Sección 2.3 introduce la relación de *subsunción homeomórfica basada en tipos*, las Secciones 2.4 y 2.5 resumen los detalles técnicos de los esquemas de decompilación modular y óptimo, la Sección 2.6 discute brevemente algunos detalles de implementación, así como la evaluación experimental llevada a cabo, y finalmente la Sección 2.7 presenta el trabajo relacionado en decompilación (interpretativa).

El Capítulo 3 proporciona una visión general del trabajo realizado sobre las aplicaciones que surgen al utilizar nuestro enfoque de decompila-

ción interpretativa, bien para analizar programas bytecode (Sección 3.1), o bien para realizar generación automática de datos de entrada para pruebas (Sección 3.2). El Capítulo 4 resume nuestro trabajo sobre el análisis de consumo del heap (Sección 4.1) y su extensión para considerar el efecto de la recolección de basura (Sección 4.2), y discute finalmente sobre el trabajo relacionado en el área (Sección 4.3). Finalmente, el Capítulo 5 presenta las conclusiones de la tesis y discute algunas líneas de trabajo en progreso y futuro.

Todos los detalles técnicos aparecen en los Artículos que han dado lugar a la tesis, los cuales aparecen íntegramente en el Apéndice A.

Capítulo 2

Decompilación Interpretativa de Bytecode a LP

Decompilar lenguajes bytecode a representaciones intermedias es hoy en día una práctica habitual en el desarrollo de analizadores, verificadores, chequeadores de modelos, etc. Por ejemplo, en el contexto de código móvil, al no tenerse acceso al código fuente, la decompilación puede facilitar la reutilización de herramientas de análisis y chequeadores de modelos existentes. En general, las representaciones intermedias de alto nivel permiten abstraer las características particulares de cada lenguaje, permitiendo por tanto que las herramientas puedan trabajar sobre representaciones más sencillas. Así por ejemplo, Java Bytecode ha sido decompilado a una representación basada en reglas en [2], a programas basados en cláusulas en [70], a una representación basada en código de *tres direcciones* en el sistema Soot [84] y al lenguaje procedural tipado BoogiePL en [37]. Así mismo, en [87] el análisis de programas Java es formalizado y llevado a cabo utilizando programas Datalog, y, en [48] programas de código ensamblador PIC son transformados a programas lógicos para su posterior análisis. Estos trabajos han demostrado que, las representaciones basadas en reglas usadas en la programación declarativa en general—y en LP en particular—proporcionan un formalismo adecuado a la hora de definir dichas representaciones intermedias. Por ejemplo, como puede verse en [2, 70, 48], la pila de operandos utilizada en los lenguajes bytecode, puede representarse explícitamente por medio de variables lógicas, y el flujo de control desestructurado puede transformarse en reglas condicionales y recursivas.

Las representaciones intermedias resultantes simplifican en gran medida el desarrollo de las herramientas arriba mencionadas para lenguajes modernos y, además, herramientas existentes desarrolladas para lenguajes declarativos (las cuales han demostrado ser efectivas) pueden ser directamente aplicadas.

La mayoría de los enfoques citados con anterioridad desarrollan decompiladores dedicados o *ad hoc*, para llevar a cabo la decompilación correspondiente. Como se comentó en la Sección 1.3, una alternativa muy prometedora al desarrollo de decompiladores dedicados es la decompilación interpretativa por evaluación parcial. Las ventajas de la (de)compilación interpretativa respecto los decompiladores dedicados son bien conocidas y discutidas en la literatura de la EP. Brevemente, éstas incluyen:

1. *Flexibilidad*: Es muy sencillo modificar el intérprete correspondiente para *afinar* la decompilación (por ejemplo, para observar nuevas propiedades de interés). Así por ejemplo, en el Artículo 1, un intérprete de Java Bytecode es instrumentado con un argumento adicional que almacena la *traza* de instrucciones bytecode para poder así acumular la historia de la ejecución. Los programas decompilados usando este intérprete incluyen por tanto en sus reglas un argumento adicional con la traza de ejecución al nivel de Java Bytecode. Esta traza permite observar distintas propiedades de los programas. Por ejemplo, se puede demostrar la ausencia de errores en tiempo de ejecución a base de demostrar que la traza no contiene instrucciones relacionadas con errores o determinados tipos de excepciones.

2. *Seguridad*: Es en principio mucho más difícil demostrar, o *confiar* en que un decompilador dedicado preserve la semántica del programa original. Por ejemplo, a la hora de definir nuestro intérprete de Java Bytecode, hemos utilizado la especificación formal Bicolano [78], la cual fue escrita, y verificada, utilizando el asistente de demostraciones Coq [11].

3. *Mantenibilidad*: Los cambios introducidos en la semántica del lenguaje pueden ser fácilmente reflejados en el intérprete. Más adelante, veremos como efectivamente definir, así como hacer evolucionar, un intérprete de bytecode en Prolog es una tarea bastante sencilla.

El reto se encuentra ahora en definir un esquema de decompilación interpretativa práctico y escalable, que sea capaz de obtener programas decompilados de calidad. Una vez seamos capaces de definir dicho esquema, será posible entonces hacer valer las ventajas arriba mencionadas.

En la literatura ha habido varias pruebas de concepto demostrando que la decompilación interpretativa es efectivamente aplicable (ver por ejemplo [48, 63]), sin embargo aún quedan muchas cuestiones por resolver si queremos aplicarlo a la decompilación de lenguajes y programas reales. A continuación, la Figura 2.1 enumera dichas cuestiones para facilitar posteriores referencias. Esta tesis responde afirmativamente a dichas cuestiones

a) *¿es el enfoque escalable?*

b) *¿preservan los programas decompilados la estructura de los programas originales?*

c) *¿es comparable la calidad de los programas decompilados con la de los programas obtenidos utilizando decompiladores ad hoc?*

Figura 2.1: Cuestiones por resolver de la decompilación interpretativa

proponiendo un esquema modular de decompilación, que asegura la obtención de decompilaciones de calidad que preservan la estructura de los programas originales.

El resto del capítulo está organizado de la siguiente manera:

- En primer lugar, la Sección 2.1 presenta informalmente los fundamentos de la EP de programas lógicos necesarios para comprender los detalles presentados en el resto del capítulo. Dichos fundamentos se presentan formalmente en la Sección 2 del Artículo 6.

- La Sección 2.2 introduce las dificultades y retos que surgen al especializar un intérprete de bytecode utilizando un ejemplo representativo.

- La Sección 2.3 presenta informalmente la *Subsunción homeomórfica basada en tipos*, una extensión de la relación de *subsunción homeomórfica* original que, teniendo en cuenta información del progra-

ma, obtiene resultados más precisos en presencia de signaturas infinitas. En los Artículos 4 y 3 se presentan los correspondientes detalles formales así como una evaluación experimental detallada.

- Las Secciones 2.4 y 2.5 introducen los ingredientes necesarios para desarrollar un esquema de decompilación *modular* y *óptimo*, capaz de tratar los puntos *a)*, *b)* y *c)* (ver Figura 2.1). Primeramente definimos la noción de *optimalidad* por medio de una serie de *criterios de optimalidad*. Después, presentamos las limitaciones de la decompilación *no-modular* e identificamos los componentes necesarios para posibilitar la definición de un esquema modular. Éstos incluyen, el cómo escribir el intérprete y el cómo controlar un evaluador parcial "online" para poder preservar la estructura del programa original respecto a las invocaciones a métodos. Finalmente, introducimos un esquema de decompilación interpretativa capaz de responder a los puntos *(a)*, *(b)* y *(c)*, produciendo programas decompilados cuya calidad es equivalente a la obtenida utilizando decompiladores ad hoc. Esto requiere una decompilación *a nivel de bloques* que evite las duplicaciones y reevaluaciones de código.

- La Sección 2.6 resume los resultados experimentales obtenidos con dos prototipos de decompilador de Java Bytecode a Prolog que incorporan las técnicas mencionadas. Dichos resultados demuestran empíricamente la escalabilidad y eficiencia del enfoque propuesto sobre una serie de programas reales escritos en Java Bytecode.

- Finalmente, la Sección 2.7 presenta el trabajo relacionado en el campo de la decompilación de lenguajes bytecode.

Para concretar, nuestro enfoque de decompilación ha sido formalizado en el contexto de la EP de programas lógicos. No obstante, las ideas propuestas que han hecho posible su aplicación práctica son de interés para la (de)compilación interpretativa de cualquier par de lenguajes origen y destino.

2.1. Fundamentos Básicos de la Evaluación Parcial de Programas Lógicos

Asumimos que el lector está familiarizado con las nociones básicas de programación lógica [68]. Ejecutar un programa lógico P para un átomo A consiste en construir un árbol SLD para $P \cup \{A\}$ y extraer de él las sustituciones computadas para cada rama no fallida del árbol. La evaluación parcial se basa en dicho enfoque de ejecución, aunque con dos diferencias fundamentales:

- Para garantizar la terminación del proceso de *desplegado*, o "*unfolding*", es en ocasiones necesario no desplegar un objetivo, dejando por tanto una hoja en el árbol con un objetivo no vacío, y posiblemente no fallido. El árbol SLD resultante se dice que es un árbol *parcial*. Nótese que incluso si los árboles SLD para todas las posibles *consultas*, o "*queries*", son finitos, los árboles SLD que se construyen en el proceso de evaluación parcial pueden ser infinitos. Esto ocurre porque, al no disponerse en tiempo de EP de los valores concretos correspondientes a los argumentos dinámicos, el árbol de EP puede tener más ramas, en particular infinitas, que el árbol SLD de ejecución. Qué átomo seleccionar de cada objetivo y cuándo parar el proceso de desplegado queda determinado por la llamada *regla de desplegado*.

- El evaluador parcial, en general, tiene que construir varios árboles SLD para garantizar que todos los átomos que aparecen en las hojas de éstos queden *cubiertos* por la raíz de algún árbol (esto se conoce como la condición de *recubrimiento* de la EP [67]). El llamado *operador de abstracción* realiza *generalizaciones* en los átomos que han de ser evaluados parcialmente para evitar que se computen árboles SLD parciales para un número infinito de átomos. Cuando todos los átomos quedan cubiertos, entonces ya no hay necesidad de construir más árboles y el proceso termina.

La esencia de la mayoría de algoritmos de evaluación parcial de programas lógicos (ver por ejemplo [42]), aparece reflejada en el algoritmo de la Figura 2.2, el cual es paramétrico respecto a la regla de desplegado, unfold, y al operador de abstracción, abstract. La función EP toma un programa

```
1: function EP (P, A, S)
2:     S_0 := S;  i := 0;
3:     repeat
4:         L^pe := unfold(S_i, P, A);
5:         S_{i+1} := abstract(S_i, L^pe, A);
6:         i := i + 1;
7:     until S_i = S_{i-1}      % (modulo renaming)
8:     return codegen(L^pe, unfold);
```

Figura 2.2: Algoritmo genérico de EP de programas lógicos

P, un conjunto (posiblemente vacío) de anotaciones $\mathcal{A}$, y un conjunto inicial de llamadas S. En cada iteración, el llamado *control local* es llevado a cabo por la regla de desplegado unfold (Línea 4), la cual toma el conjunto actual de átomos S_i, el programa y las anotaciones, y construye un árbol parcial SLD para cada llamada en S_i. En el control global, llevado a cabo por el operador de abstracción abstract, cuando hay llamadas en las hojas de los árboles que no están aún *cubiertas*, el operador abstract las añade al nuevo conjunto de átomos a ser evaluado parcialmente. Para garantizar la terminación del proceso de control global, es decir, para asegurar que la condición $S_i = S_{i-1}$ se cumple, estos átomos deben *generalizarse* de forma adecuada.

La evaluación parcial de P con respecto a S es entonces extraída sistemáticamente del conjunto resultante de llamadas L^{pe} de la última iteración, en la llamada fase de generación código, codegen en L8. Se utiliza entonces la noción de *resultante* para generar las reglas de programa asociadas a cada derivación de la raíz a cada hoja del árbol SLD para el conjunto final L^{pe}. Dada una derivación SLD de $P \cup \{A\}$, con $A \in L^{pe}$ terminando en B y siendo θ la composición de los *unificadores más generales posibles* (ver [68]) de los pasos de derivación, la regla "$\theta(A)$:- B" se denomina un *resultante* asociado a dicha derivación. Una EP se define como el conjunto de resultantes (cláusulas) asociados a las derivaciones de los árboles SLD parciales para $P \cup L^{pe}$. El programa resultante se denomina habitualmente *programa especializado* o *programa residual*. La Sección 2 del Artículo 6 presenta más formalmente los fundamentos de evaluación parcial de programas lógicos.

2.1.1. Evaluación Parcial "Online" frente a "Offline"

Es bien sabido que tanto la calidad de los programas especializados como el tiempo que requiere el proceso de EP, dependen en gran medida de las estrategias de control utilizadas. Tradicionalmente se han considerado dos enfoques diferentes de EP, "*online*" y "*offline*". En la EP online, todas las decisiones de control se toman "al vuelo" durante la fase de especialización teniendo en cuenta la historia pasada. En el enfoque offline, todas las decisiones de control son tomadas antes de la propia fase de especialización. Éstas se basan en descripciones abstractas de los datos en lugar de en los datos concretos. Normalmente, la estrategia de control se representa por medio de anotaciones en el programa, las cuales constituyen en realidad el único criterio que controla la EP. Por ejemplo, en el control local, una anotación podría indicar explícitamente que un átomo no debe desplegarse. En el control global, las anotaciones normalmente especifican, para cada llamada, qué argumentos deben generalizarse (en este caso, reemplazarse por variables libres). Dichas anotaciones, en algunos casos son generadas automáticamente por un análisis de "binding-time" [32, 65], y en otros casos son proporcionadas, parcial o totalmente, por el usuario.

De acuerdo a esta clasificación, el algoritmo de EP que proponemos se puede considerar como un híbrido pues el conjunto de anotaciones $\mathcal{A}$ puede proporcionar información a los operadores de control, como pasa en la EP offline, y incluye reglas de control basadas en la historia de especialización, como pasa en la EP online. La principal ventaja del enfoque offline es que, una vez son tomadas todas las decisiones de control, la fase de EP es en principio mucho más simple y eficiente. Por otro lado, la EP online, aunque es en principio más ineficiente, es estrictamente más potente pues las decisiones de control pueden basarse en los valores concretos en lugar de en abstracciones de éstos. Por tanto, aunque los resultados obtenidos por un evaluador parcial offline pueden siempre replicarse usando uno online, en muchos casos, los resultados obtenidos usando EP online no pueden reproducirse usando EP offline.

En este trabajo estamos interesados en investigar hasta donde se puede llegar usando un enfoque online, interesándonos por tanto primeramente en obtener resultados precisos sin importarnos la eficiencia. De esta forma esperamos poder obtener decompilaciones de alta calidad que no podrían obtenerse utilizando EP offline. Más adelante, afrontaremos el reto de la

eficiencia del proceso tratando de no perder calidad. Como veremos, muchas de las lecciones aprendidas en ésta tesis son de interés tanto para el campo de la EP online como offline.

2.2. Retos en la Especialización de Intérpretes de Bytecode

Esta sección ilustra por medio de un ejemplo, los retos a los que hay que enfrentarse a la hora de especializar un intérprete de bytecode. La Figura 2.3 muestra un fragmento de un intérprete de bytecode implementado en Prolog. Se asume que el código de cada método del programa bytecode viene representado como un conjunto de hechos `bytecode/3` tal que, para cada par $pc_i : bc_i$ en el código del método m, se tiene un hecho `bytecode(`m`,`pc_i`,`bc_i`)`. El estado que el intérprete manipula es de la forma `st(Fr,FrStack)`, donde `Fr` representa el "frame" (o contexto) actual y `FrStack` la pila de frames (o pila de llamadas), implementada como una lista Prolog. Los frames son de la forma `fr(M,PC,OStack,LocalV)`, donde `M` representa el método actual, `PC` el contador de programa, `OStack` la pila de operandos y `LocalV` la lista de variables locales. El predicado `main/3`, dado el método a ser interpretado `Method` y sus argumentos de entrada `InArgs`, construye primeramente un estado inicial llamando al predicado `build_s0/3`, y llama después al predicado `execute/2`, devolviéndose el resultado en la variable `Res`, la cual representa la cima de la pila de operandos al final de la ejecución. A su vez, `execute/2` llama al predicado `step/3`, el cual produce `S'`, el estado inmediatamente después de ejecutar el correspondiente bytecode, y llama recursivamente al predicado `execute/2` con `S'` hasta que se encuentre una instrucción `return` con la pila de llamadas vacía en el estado. Por brevedad, sólo mostramos la definición del predicado `step/3` para una selección de instrucciones bytecode, y omitimos el código de algunos predicados auxiliares. En particular, no mostramos el código de `build_s0/3`, el cual fue explicado con anterioridad, `next/3`, que produce el siguiente contador de programa dado el actual, y `split_OS/4`, que divide la pila de operandos actual entre la lista de parámetros del método llamado y el resto.

La figura 2.4 muestra el programa bytecode que utilizaremos como ejem-

```
main(Method,InArgs,Res) :-
   build_s0(Method,InArgs,S0),
   execute(S0,Sf),
   Sf = st(fr(_,_,[Res|_],_),_)).

execute(S,S) :-
   S = st(fr(M,PC,_,_),[]),
   bytecode(M,PC,return).
execute(S,Sf) :-
   S = st(fr(M,PC,_,_),_),
   bytecode(M,PC,Inst),
   step(Inst,S,S'),
   execute(S',Sf).

step(goto(PC),S,S') :-
   S = st(fr(M,_,OS,LV),FrS),
   S' = st(fr(M,PC,OS,LV),FrS).
```

```
step(push(X),S,S') :-
   S = st(fr(M,PC,OS,L),FrS),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[X|OS],L),FrS).
...
step(invoke(M'),S,S') :-
   S = st(fr(M,PC,OS,LV),FrS),
   split_OS(M',OS,Args,OS''),
   build_s0(M',Args,
            st(fr(M',PC',OS',LV'),_)),
   S' = st(fr(M',PC',OS',LV'),
           [fr(M,PC,OS'',LV)|FrS]).
step(return,S,S') :-
   S = st(fr(_,_,[RV|_],_),
          [fr(M,PC,OS,LV)|FrS]),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[RV|OS],LV),FrS).
```

Figura 2.3: Fragmento de un intérprete de bytecode

plo. Arriba, mostramos el código fuente Java y abajo el bytecode correspondiente. Nótese que el código fuente sólo se muestra para facilitar la comprensión del programa, pues el decompilador trabaja directamente sobre el bytecode. El ejemplo consiste en una serie de métodos que realizan diferentes cálculos aritméticos. El método `gcd` calcula el máximo común divisor, `abs` el valor absoluto y `fact` el factorial implementado de forma recursiva. El método `count` no tiene ningún significado especial, simplemente incrementa un contador, inicializado a 0, hasta que su valor llega al valor del argumento de entrada.

Para poder obtener una decompilación *efectiva*, es necesario disponer de estrategias de control (es decir, operadores unfold y abstract) lo suficientemente potentes como para librarse de la llamada *capa de interpretación*. Es por ello, que en nuestros primeros experimentos en el Artículo 1, utilizamos estrategias de control *agresivas* basadas en la *subsunción homeomórfica* [58, 62]. En el control local, entendemos por agresivas a aquellas reglas de desplegado capaces de expandir las derivaciones lo más posible siempre que no haya problemas de terminación. En el control global, nos referi-

```
int count(int n){
   int i = 0;
   while (i < n) i++;
   return i;}

int gcd(int x,int y){
   int res;
   while (y != 0){
      res = x%y; x = y;
      y = res;}
   return abs(x);}
```

```
int abs(int x){
   if (x < 0) return -x;
   else return x;
}

int fact(int x){
   if (x == 0)
      return 1;
   else
      return x*fact(x-1);
}
```

```
Method count
  0:push(0)
  1:store(1)
  2:load(1)
  3:load(0)
  4:ifge(3)
  5:inc(1,1)
  6:goto(2)
  7:load(1)
  8:return
```

```
Method gcd
  0:load(1)
  1:if0eq(11)
  2:load(0)
  3:load(1)
  4:rem
  5:store(2)
  6:load(1)
  7:store(0)
  8:load(2)
  9:store(1)
  10:goto 0
  11:load(0)
  12:invoke(abs)
  13:return
```

```
Method abs
  0:load(0)
  1:if0ge(5)
  2:load(0)
  3:neg
  4:return
  5:load(0)
  6:return
```

```
Method fact
  0:load(0)
  1:if0ne(4)
  2:push(1)
  3:return
  4:load(0)
  5:load(0)
  6:push(1)
  7:sub
  8:invoke(fact)
  9:mul
  10:return
```

Figura 2.4: Código fuente y bytecode del programa de ejemplo

mos a operadores de abstracción que generalizan en el menor número de situaciones posible sin hacer peligrar la terminación.

La Figura 2.5 muestra el programa decompilado obtenido utilizando el evaluador parcial disponible en el sistema `CiaoPP` [49]. Para estos experimentos preliminares, usamos la *subsunción homeomórfica* para controlar tanto el nivel local como el global. Mirando al código obtenido, podemos observar lo siguiente:

1. El evaluador parcial no ha sido capaz de decompilar satisfactoriamen-

```
main(count,[N],A) :-
    % out of memory error

main(gcd,[A,0],A) :- A>=0.
main(gcd,[B,0],A) :-
    B<0, A is -B.
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).
```

```
execute_1(A,0,A) :- A>=0.
execute_1(A,0,C) :- A<0, C is -A.
execute_1(A,B,G) :-
    B\=0, I is A rem B,
    execute_1(B,I,G).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[N],A) :- ...% Full interpreter
```

Figura 2.5: Código decompilado para el ejemplo. Primer intento.

te el método `count`. Realmente se queda sin memoria por problemas de terminación, en este caso, tanto en el control local como en el global. El problema es que la *subsunción homeomórfica* no garantiza la terminación de la EP en programas que pueden generar potencialmente un número infinito de valores, como es el caso de nuestro intérprete de bytecode (en particular, por el uso del predicado `is/3`).

2. La composicionalidad respecto a métodos del programa original se ha perdido en el programa decompilado. Esto puede verse observando el código correspondiente al método `gcd`. Nótese que en la versión decompilada no aparece la llamada a `abs`, sino que el código correspondiente a éste aparece como "inline". Aunque esto puede considerarse como una característica positiva desde el punto de vista de la especialización, las consecuencias pueden degradar considerablemente la eficiencia y la calidad del proceso, y de hecho hace imposible el poder escalar al considerar programas reales que incluyan por ejemplo llamadas a librerías.

3. El código decompilado correspondiente al método `fact` contiene básicamente al intérprete completo y no aparece en la figura por motivos de espacio. Este problema fue primeramente detectado en [43] y aparece al tratar de decompilar por EP un método recursivo. Como veremos, este problema, así como la solución que propondremos más adelante, está muy relacionado con el problema de la composicionalidad explicado en el punto anterior.

4. Si miramos el código de los predicados `main(gcd,...)` y `execute/3`, podemos observar que hay duplicaciones de código. Y lo que es peor, el evaluador parcial produce estas duplicaciones porque parte del programa bytecode se reevalúa en el proceso de EP. En nuestra evaluación experimental demostraremos que el tener o no tener estas duplicaciones (reevaluaciones), hace la diferencia entre que el decompilador sea o no capaz de escalar en la práctica.

Las soluciones a estos problemas se resumen en los siguientes tres retos:

- **Reto I. Tratamiento de signaturas infinitas en la EP:** Estudiaremos en primer lugar las soluciones existentes identificando sus debilidades, y propondremos entonces la *subsunción homeomórfica basada en tipos*. Este asunto será discutido en la Sección 2.3 y elaborado con más detalle en los Artículos 3 y 4.

- **Reto II: Un esquema de decompilación modular.** Incluso después de solucionar el *Reto I*, veremos que es necesario diseñar un esquema de decompilación modular que preserve la composicionalidad respecto a métodos del programa original, y que además resuelva el problema de los programas recursivos. Dicho esquema de decompilación se introduce en la Sección 2.4 y se estudia con más nivel de detalle en el Artículo 6.

- **Reto III: Decompilación óptima.** En nuestros primeros experimentos utilizando el esquema de decompilación modular con programas reales, observaremos que aún no es posible escalar con éxito. Introduciremos entonces un esquema *óptimo* de decompilación que asegura que los tiempos de decompilación y los tamaños de los programas decompilados, crecen linealmente respecto al tamaño de los programas bytecode de entrada. Esto se consigue principalmente evitando las duplicaciones y reevaluaciones de código. Éste asunto se introduce en la Sección 2.5 y se estudia con más detalle en el Artículo 6.

2.3. Reto I: Tratamiento de Signaturas Infinitas en la EP

2.3.1. La Subsunción Homeomórfica

La relación de *subsunción homeomórfica*, "homeomorphic embedding" (HEm) [58, 61, 62], se ha convertido en una técnica muy popular para supervisar la terminación de métodos online de transformación y especialización, resultando esencial para obtener optimizaciones potentes, por ejemplo en el contexto de la EP online. Intuitivamente, el HEm es un orden estructural bajo el cual una expresión t_1 *subsume* a una expresión t_2, escrito como $t_2 \trianglelefteq t_1$, si t_2 puede obtenerse a partir de t_1 borrando algunos operadores. Por ejemplo, $\underline{\mathtt{s}}(\mathtt{s}(\underline{\mathtt{U}} + \mathtt{W}) \underline{\times} (\underline{\mathtt{U}} \underline{+} \mathtt{s}(\underline{\mathtt{V}})))$ subsume a $\mathtt{s}(\mathtt{U} \times (\mathtt{U} + \mathtt{V}))$.

La relación HEm puede usarse para garantizar terminación gracias a la siguiente propiedad: si asumimos que el conjunto de constantes y functores es finito, toda secuencia infinita de expresiones $t_1, t_2, \ldots$, contiene al menos un par de elementos t_i y t_j con $i < j$ tal que $t_i \trianglelefteq t_j$. Por tanto, al computar iterativamente una secuencia $t_1, t_2, \ldots, t_n$, podemos garantizar su finitud utilizando el HEm como un "silbato". Cuando una expresión t_{n+1} va a añadirse a la secuencia, primero se chequea que se cumple $t_i \not\trianglelefteq t_{n+1}$ para todo i tal que $1 \leq i \leq n$. Intuitivamente, la computación puede progresar mientras la nueva expresión obtenida no sea mayor que (no subsuma a) ninguna de las expresiones previamente computadas, pues esto podría significar que estamos ante una secuencia infinita. El éxito del HEm se debe a que las secuencias pueden crecer considerablemente antes de que el silbato se active, en general más que con otras técnicas para garantizar terminación, lo cual suele significar una mayor efectividad de las transformaciones.

Aunque se ha demostrado que el HEm es una técnica potente para computaciones simbólicas, éste aún puede presentar algunas dificultades, en particular en presencia de signaturas infinitas. En el caso de programas lógicos, éstas pueden aparecer al utilizar algunos "builtins" de Prolog como `is/2`, `functor/3` y `name/2`. Se han definido variantes del HEm para tratar con signaturas infinitas (ver por ejemplo [61, 6]), sin embargo éstas tienden a ser demasiado conservadoras en la práctica.

2.3.2. Ejemplo Motivador

Consideremos el método `count` que aparece en la parte izquierda de la Figura 2.4. El método recibe un número entero, inicializa un contador a "0" (ver bytecodes `0` y `1`) y ejecuta un bucle que incrementa el contador en uno en cada iteración (bytecode `5`), hasta que el valor llega al valor del argumento de entrada (la condición se chequea en los bytecodes `2`, `3` y `4`). El método devuelve el valor del contador en los bytecodes `7` y `8`. Para decompilar el método `count`, evaluamos parcialmente el intérprete de la Figura 2.3 respecto al bytecode del método `count` empezando por el átomo `main(count,[N],I)`, donde `N` representa el argumento de entrada y `I` el valor de retorno (es decir, la cima de la pila al final de la computación).

En la Figura 2.6 se muestra (una versión reducida de) uno de los árboles SLD que da lugar a una decompilación efectiva del método `count`, y al que nos referiremos posteriormente. Para simplificar la comprensión, aparte del átomo de entrada `main/3`, solo mostramos los átomos correspondientes al predicado `execute/2`, pues es el único predicado recursivo del programa. Así, cada flecha en el árbol se corresponde realmente con varios pasos de derivación. Nótese que, algunas de las operaciones dentro del cuerpo de cada regla del predicado `step`, pueden quedar residuales al necesitar datos no conocidos en tiempo de EP. La regla de computación utilizada por el operador de desplegado es capaz de residualizar llamadas que no estén suficientemente instanciadas, y seleccionar así átomos del objetivo que no sean necesariamente los de más a la izquierda de forma segura [7], en particular, se seleccionarán llamadas a átomos `execute/2`. Representaremos estas llamadas residuales como etiquetas asociadas a las ramas del árbol.

Utilizando el HEm original

Consideremos primero un evaluador parcial que utiliza el HEm para controlar la terminación tanto en el control local como en el global. Como puede verse en la figura, el valor del `PC` "`2`" se corresponde con la entrada del bucle. Aplicando el HEm, la evaluación contiene una subsecuencia de átomos de esta forma: $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[\texttt{N},0]),[]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[\texttt{N},1]),[]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[\texttt{N},2]),[]),S_f),\ldots$, la cual aparece marcada en la figura con rectángulos de línea discontinua. Dicha secuencia se corresponde con las sucesivas iteraciones con-

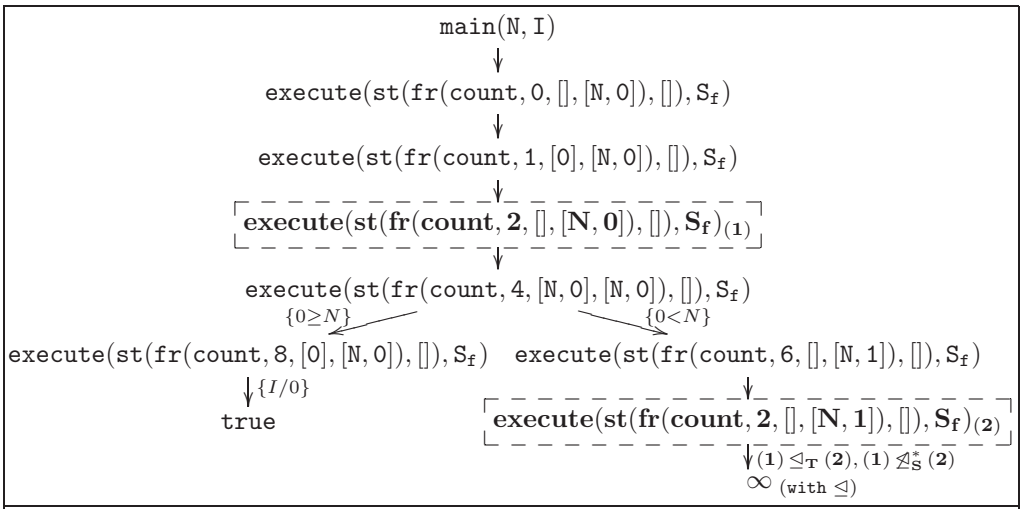


Figura 2.6: Árbol SLD de desplegado y código decompilado del ejemplo

secutivas del bucle, en las cuales el control vuelve a la cabeza de éste (ver el valor 2 en el valor del PC del estado), y el valor del contador (variable de la segunda posición de la lista) se va incrementando de uno en uno. Esta secuencia puede crecer de forma infinita, pues el HEm no la detecta como potencialmente peligrosa (ver "∞ (with $\trianglelefteq$)" en la figura). Esto ocurre debido al uso que hace el intérprete del operador de Prolog is/2, rompiéndose así la propiedad de finitud de signatura que se cumple en los programas lógicos puros.

Para obtener una decompilación de calidad, es necesario que el valor del contador (variable local 1) sea filtrado, pero no así el del PC. Como vemos en la figura, esto requiere parar la derivación cuando aparezca el átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 1]), []), \texttt{S}_\texttt{f})$ (marcado como $(\mathbf{1}) \trianglelefteq_{\mathbf{T}} (\mathbf{2})$), y generalizarlo con respecto al átomo anterior encerrado en el rectángulo con línea discontinua, resultando así el átomo $\texttt{execute}(\texttt{st}(\ \texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, \texttt{X}]), []), \texttt{S}_\texttt{f})$.

Recuperando la terminación: Subsunción con filtrado de números

En programas que contienen predicados aritméticos de Prolog pero que no generan infinitos functores vía `functor/3`, `=../2`, etc., una solución inmediata para recuperar la terminación es utilizar la relación $\trianglelefteq_{num}$. Ésta, es una adaptación del HEm que simplemente filtra los valores numéricos, es decir, cualquier número subsume a otro número. En el ejemplo, el átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[\texttt{N},1]),[]),\ \texttt{S}_\texttt{f})$ subsume a $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[N,0]),[]),S_f)$ bajo $\trianglelefteq_{num}$, evitándose así la *no terminación*. Desafortunadamente, esta modificación del HEm, es demasiado simplista, y da lugar a una perdida excesiva de precisión. Por ejemplo, al especializar $\texttt{main}(\texttt{count},[\texttt{N}],\texttt{I})$, los primeros dos átomos de `execute/2` son $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},0,[],[\texttt{N},0]),[]),\texttt{S}_\texttt{f})$ y $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},1,[0],[\texttt{N},0]),[]),\texttt{S}_\texttt{f})$. Usando $\trianglelefteq_{num}$, el silbato se activa en este punto y el desplegado tiene que parar. Esto provoca que este último átomo sea generalizado en el control global produciéndose $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{X},\texttt{Y},[\texttt{N},0]),[]),\texttt{S}_\texttt{f})$. Esto no es aceptable en el caso de la especialización de nuestro intérprete, pues se pierde la pista de la siguiente instrucción a ser ejecutada—lo que provoca que no se pueda eliminar la capa de interpretación—y de hecho, en la mayoría de casos provoca que el programa residual obtenido contenga prácticamente el intérprete por completo.

Incrementando la Precisión: Símbolos Estáticos del Programa

Una manera sintáctica de mejorar la precisión asegurando al mismo tiempo terminación, propuesta en [61], consiste en considerar dos conjuntos de símbolos: uno con aquellos que aparecen explícitamente en el programa y el objetivo, y otro con el conjunto infinito de símbolos que el programa puede generar potencialmente. Denotaremos esta relación como $\trianglelefteq^*_S$. Al comparar dos términos, nos quedamos con los símbolos que pertenezcan al conjunto finito y filtramos el resto. Bajo esta relación, el átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},1,[0],\ [\texttt{N},0]),[]),\texttt{S}_\texttt{f})$ no subsume al átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},0,[],[\texttt{N},0]),[]),\texttt{S}_\texttt{f})$, pues los números `0` y `1` son símbolos estáticos diferentes del programa. Por tanto, en este caso, el evaluador parcial no se ve obligado a generalizarlos preservándose así el valor del `PC`.

Desafortunadamente, la relación $\trianglelefteq^*_S$ resulta no comportarse tampoco de forma óptima en nuestro caso, pues $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[\texttt{N},1]),[]),\texttt{S}_\texttt{f})$

no subsume a $\texttt{execute}(\ \texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f})$. Esto significa que el proceso de desplegado continua con una segunda iteración del bucle. Aunque está garantizado que el proceso termina, se desplegarán tantas iteraciones del bucle como distintas constantes numéricas consecutivas aparezcan en el programa, en este caso 8. No será posible por tanto obtener la decompilación óptima que aparece en la parte de abajo de la Figura 2.6. Para obtener dicha decompilación, es necesario que el evaluador parcial generalice el contador del bucle lo antes posible, es decir, que el átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 1]), []), \texttt{S}_\texttt{f})$ subsuma a $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f})$.

Intuitivamente, la razón por la que esta relación no se comporta de forma óptima es porque ésta no es capaz de distinguir entre los distintos argumentos y los trata todos por igual. En resumen, este ejemplo sugiere que es necesario tener una relación de subsunción que sea capaz de tener información de contexto en cuenta: en particular, dicha relación dependiente del contexto, debería tratar de forma diferente el valor del PC y el valor de la variable del contador.

2.3.3. Subsunción Homeomórfica basada en Tipos

En presencia de signaturas infinitas, existe un método general para definir relaciones de subsunción homeomórfica; en [61] se define la *subsunción homeomórfica extendida* basada en resultados previos de Kruskal [58] y Dershowitz [38]. Esta solución define una familia de relaciones de subsunción, donde una relación subsidiaria de orden, definida sobre los símbolos de función del programa, juega un papel esencial. No obstante, veremos que ésta no resuelve realmente el problema en la práctica, pues no propone ningún mecanismo automático para encontrar la relación "correcta" entre los símbolos de función.

Esta tesis propone la *subsunción homeomórfica basada en tipos*, "type-based homeomorphic embedding" (TbHEm), una relación que mejora el HEm original haciendo uso de información adicional proporcionada en forma de tipos. Veremos como este enfoque puede verse como una forma de generar instancias concretas de la relación de HEm extendida como definió Leuschel, incluyendo la posibilidad de tener en cuenta la semántica del programa. Los tipos requeridos para guiar al TbHEm pueden darse manual-

mente o, lo que es más interesante, pueden inferirse automáticamente por análisis de programas, como discutimos en el Artículo 3.

La observación principal en al que se basa el TbHEm es que, incluso aunque una expresión este definida sobre una signatura infinita, es posible que sólo tome un conjunto finito de valores sobre el dominio correspondiente para cada computación. Para realizar dicha distinción, nuestra relación se define sobre tipos, los cuales se estructuran en una partición finita (posiblemente vacía) y una partición infinita (también posiblemente vacía). Intuitivamente, el TbHEm permite expandir secuencias mientras, al comparar subtérminos de un tipo infinito, los valores concretos que aparecen en la expresión se mantengan en la partición finita del tipo correspondiente.

Utilizando el TbHEm para controlar la EP del intérprete de bytecode

En el caso de nuestro intérprete de bytecode, el argumento del `PC` se puede definir por un tipo estructurado de forma que el intervalo acotado en el cual éste se mueve constituye su partición finita, y el resto de los números enteros forma su parte de infinita. De esta manera, el TbHEm no generalizará el `PC` mientras su valor permanezca dentro del intervalo acotado.

Para inferir este tipo, utilizaremos técnicas de análisis existentes, en particular, usaremos el análisis de *tipos buenos* ("well-typings") descrito por Bruynooghe *et al.* [20][1]. Éste infiere el siguiente tipo τ_{PC} para el contador de programa del intérprete de la Figura 2.3, teniendo en cuenta el programa bytecode de la Figura 2.4:

τ_{PC} `--> -4; 0; 1; 2; 3; 4; 5; 6; 7; 8; num`

Se puede interpretar que el tipo τ_{PC} consiste en una partición finita (las constantes numéricas) y una partición infinita (el resto de los números distintos de las constantes). Es decir, el tipo se puede interpretar como $\tau_{PC} \rightarrow F; I$ donde la partición F es $\{-4, 0, 1, 2, \ldots, 8\}$ y $I = \texttt{num} \setminus F$. Usando esta regla de tipo, el TbHEm asegura que el contador de programa nunca será abstraído durante la EP, mientras su valor se mantenga en el rango esperado (las constantes numéricas). El átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 1, [0], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f})$ no subsume a

[1]Disponible on-line en `http://saft.ruc.dk/Tattoo/`

$\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 0, [], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f})$ usando esta definición de tipo, por tanto, la derivación puede avanzar. Esto evita la necesidad de generalizar el PC lo que provocaría que no pudiésemos obtener una especialización efectiva. La derivación bien terminará, o bien el valor del PC se repetirá por algún salto hacia atrás en el código (bucle). En este caso, el TbHEm, también escrito $\trianglelefteq_T$, detectará el átomo correspondiente como peligroso, por ejemplo, $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f}) \trianglelefteq_\texttt{T} \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 1]), []), \texttt{S}_\texttt{f})$, como vemos en la Figura 2.6.

El programa decompilado que obtenemos usando los tipos inferidos en combinación con el TbHEm se muestra en la parte baja de la Figura 2.6. Se puede observar que esta decompilación es óptima[2] en el sentido de que la capa de interpretación se ha eliminado totalmente o no aparece código residual superfluo.

Aparte de la inferencia de "well-typings" que hemos visto, en el Artículo 3 se bosqueja como utilizar un análisis de cotas numéricas para inferir información que puede ser útil para el TbHEm. Este tipo de análisis calcula sobreaproximaciones del conjunto de valores que los argumentos del programa pueden tomar. Intuitivamente, si podemos probar que dicho conjunto está acotado, entonces sabemos que la partición infinita del tipo es vacía, y por tanto podemos aplicar se forma segura el HEm tradicional (mejorando así la efectividad de la EP).

Nótese que, determinar el conjunto exacto de símbolos que pueden aparecer en tiempo de ejecución en un punto específico del programa, y en particular determinar si ese conjunto es finito, está estrechamente relacionado con el problema de la terminación, y es por tanto indecidible. Sin embargo, cuanto mejores sean los tipos, más agresiva será la EP sin sacrificar en ningún caso su terminación. Si los tipos derivados tienen particiones finitas demasiado pequeñas, entonces probablemente se producirán demasiadas generalizaciones resultando en una especialización muy pobre; mientras que si éstos son demasiado grandes, entonces la especialización tenderá a ser demasiado agresiva, produciendo posiblemente versiones innecesarias.

[2]Veremos después que ésta puede mejorarse

```
main(count,[N],0) :- 0>=N.
main(count,[N],I) :-
    0<N, execute_2(N,1,I).

execute_2(N,I,I) :- I>=N.
execute_2(N,A,I) :-
    A<N, A' is A+1,
    execute_2(N,A',I).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[N],A) :- ...% full int.
```

```
main(gcd,[A,0],A) :- A>=0.
main(gcd,[B,0],A) :-
    B<0, A is -B.
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).

execute_1(A,0,A) :- A>=0.
execute_1(A,0,C) :- A<0, C is -A.
execute_1(A,B,G) :-
    B\=0, I is A rem B,
    execute_1(B,I,G).
```

Figura 2.7: Código decompilado para el ejemplo después de superar el Reto 1

2.4. Reto II: Decompilación Modular

Una vez se ha superado el problema de las signaturas infinitas, la clase de programas bytecode que podemos decompilar con éxito es considerablemente más amplia. Otro aspecto importante, que no hemos discutido en esta introducción, es que utilizando el operador clásico de abstracción basado simplemente en el HEm original, o incluso mejorándolo con el TbHEm, los programas decompilados que obtenemos tienden a tener demasiadas versiones especializadas (redundantes) para algunos predicados. Este problema se estudia con detalle en el Artículo 2 donde se propone un operador de abstracción avanzado el cual es capaz de controlar la *polivarianza* del proceso de EP, es decir, es capaz de evitar tener dichas versiones especializadas redundantes. Como mostramos en el Artículo 2, esto nos permite obtener mejores decompilaciones, de forma más eficiente, lo que amplia aún más la clase de programas que podemos decompilar con éxito. No obstante, incluso mejorando nuestro evaluador parcial para que incluya tanto el TbHEm como este operador de abstracción mejorado, el esquema de decompilación resultante resulta aún insatisfactorio a la hora de decompilar programas reales, pues entre otras cosas, y como veremos, la composicionalidad del programa original respecto a las llamadas a métodos se pierde en la decompilación.

Consideremos de nuevo nuestro ejemplo de la Figura 2.4. El código decompilado que obtenemos usando el evaluador parcial mejorado se muestra en la Figura 2.7. Se puede observar que éste es básicamente el mismo de la Figura 2.5 excepto para el método `count`, para el que se obtiene ahora el código que mostramos en la parte baja de la Figura 2.6. Es importante hacer notar que este ejemplo no es suficientemente complejo como para poner en evidencia el problema de la polivarianza que el nuevo operador de abstracción del Artículo 2 resuelve. El lector interesado puede consultar el Artículo 2 donde se presenta un ejemplo representativo en este sentido.

A la vista del ejemplo, identificamos las siguientes cuatro limitaciones del esquema actual de decompilación (a partir de ahora llamado decompilación *no-modular*) denotadas como **(L1)**... **(L4)**. Nótese que dichas limitaciones, así como la forma de resolverlas que explicamos más adelante, son también relevantes para el caso de la decompilación por medio de EP puramente offline.

(L1) Los métodos aparecen "inlined" en los diferentes contextos en los que son llamados, lo que hace que se pierda la estructura original del código. Por ejemplo, la invocación a `abs` desde `gcd` (línea 12 de `gcd`) no aparece en el código decompilado. Como resultado, el código decompilado para `gcd` tiene dos casos *base* en los que aparecen "inlined" los correspondientes "builtins" de `abs`, es decir, `A>=0`, `B<0` y `A is -B`. Esto ocurre porque las llamadas a métodos se tratan de forma "small-step" en el intérprete, es decir, el código de los métodos invocados se despliega como se estuviese incluido dentro el método que lo invoca.

(L2) Como consecuencia, el proceso de decompilación es muy ineficiente cuando aparecen muchas llamadas a métodos. Por ejemplo, si se tienen n llamadas a un mismo método, éste será decompilado n veces. Incluso aún peor, si aparece una invocación a método dentro de un bucle, el código será decompilado en el caso mejor 2 veces, al tenerse que realizar la correspondiente generalización en el control global antes de llegar al punto fijo en la EP. Esto podría incluso ser peor en el caso de bucles anidados.

(L3) El esquema no-modular no trabaja de forma incremental, en el sentido de que no permite la decompilación *separada* de métodos sino que redecompila todas las llamadas. Por tanto, decompilar un lenguaje real es totalmente inviable, pues han de considerarse las librerías, cuyo código podría incluso no estar disponible. La limitación L2 junto con la L3

responden negativamente al punto *(a)* de la Figura 2.1.

(L4) El programa decompilado contendrá básicamente el intérprete completo cuando aparezcan métodos recursivos. Esta es la razón por la que en el programa decompilado de la Figura 2.7 no aparece el código correspondiente al método recursivo `fact`. El problema con la recursión es el siguiente. Asumamos que queremos decompilar el método recursivo $m1$ cuyo código es de la siguiente forma $\langle pc_0 : bc_0, \ldots, pc_j : invoke(m1), \ldots, pc_n : return\rangle$. Hay una primera decompilación para $A_k = \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}, \texttt{lv}), []), \texttt{S}_\texttt{f})$ en la que la pila de llamadas es vacía. Al decompilarla, aparece una llamada de la forma $A_l = \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}', \texttt{lv}'), [\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}, \texttt{lv})]), \texttt{S}_\texttt{f})$, con la pila de llamadas conteniendo la anterior llamada, al llegar a la llamada recursiva. En este punto, la derivación debe pararse pues $A_k \trianglelefteq_T A_l$. Para asegurar terminación, el control global generaliza estas llamadas a $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \_, \_), \_), \texttt{S}_\texttt{f})$, donde $\_$ denota una variable libre, siendo por tanto la pila de llamadas desconocida. Como consecuencia, al evaluar la instrucción *return*, la continuación obtenida de la pila de llamadas es desconocida produciéndose la llamada $\texttt{execute}(\texttt{st}(\texttt{fr}(\_, \_, \_, \_), \_), \texttt{S}_\texttt{f})$, que habrá de decompilada. El hecho de que el método y el contador de programa sean desconocidos provoca que sea imposible eliminar la capa de interpretación, y de hecho, el código decompilado contendrá potencialmente el intérprete al completo. Esta situación se da al decompilar el método `fact`. Las limitaciones L1 y L4 responden negativamente al punto *(b)* (ver Figura 2.1).

A continuación identificamos los ingredientes necesarios para definir un esquema de decompilación *modular*. Entendemos por decompilación *modular*, una decompilación en la que la unidad de procesamiento es el método, es decir, se decompila un método cada vez. Mostraremos como dicho esquema resuelve las cuatro limitaciones descritas de la decompilación nomodular y responde afirmativamente a los puntos *(a)* y *(b)* de la Figura 2.1. Básicamente necesitaremos: (i) Dar un tratamiento composicional a las invocaciones a método. Veremos que esto se puede conseguir considerando un intérprete implementado utilizando una semántica "big-step". (ii) Proporcionar un mecanismo para residualizar las llamadas del programa decompilado (es decir, no desplegarlas y añadirlas sin modificaciones al código residual). Generaremos automáticamente anotaciones en la EP

para este propósito. (iii) Estudiaremos las condiciones que aseguran que la decompilación *separada* de métodos es correcta.

2.4.1. Intérprete con Semántica "Big-step" para habilitar la Modularidad

Tradicionalmente, se han considerado dos enfoques distintos a la hora de definir la semántica de un lenguaje, la semántica "big-step" (o *natural*) y la "small-step" (o *operacional-estructural*), ver por ejemplo [59]). Básicamente, en una semántica "big-step" las transiciones relacionan los estas inicial y final para cada instrucción, mientras que en una "small-step" las transiciones definen el siguiente paso de ejecución para cada sentencia. En el contexto de los intérpretes de bytecode, ocurre que la mayoría de las instrucciones se ejecutan en un solo paso, haciendo que ambos enfoques sean prácticamente equivalentes. Este es el caso de nuestro intérprete de bytecode de la Figura 2.3 para todas las instrucciones excepto para *invoke*. La transición para *invoke* en la semántica "small-step" define el siguiente paso de la computación, es decir, el "frame" actual se apila en la pila de llamadas y se inicializa un nuevo "frame" para la ejecución del método invocado. Nótese que después de dar este paso, no es posible distinguir ya entre el código del método anterior y el llamado. Esto provoca que no podamos obtener modularidad en la decompilación.

En el contexto de la decompilación interpretativa de lenguajes imperativos, tradicionalmente se han utilizado intérpretes con semántica "small-step" (ver por ejemplo [77, 48]). En esta tesis sostenemos que el uso de intérprete con una semántica "big-step" es necesario para poder definir un esquema modular y poder así escalar al considerar lenguajes y programas reales. En la Figura 2.8, mostramos la parte relevante de la versión "big-step" del intérprete de bytecode de la Figura 2.3. Podemos observar que ahora, la instrucción *invoke*, una vez extraídos los parámetros de llamada de la pila de operandos, llama recursivamente al predicado `main/3` para ejecutar el método llamado. Al terminar la ejecución del método, el valor de retorno se apila de vuelta en la pila de operandos del nuevo estado y la ejecución procede normalmente. Por otro lado ya no es necesario llevar en el estado explícitamente la pila de llamadas, sino sólo la información de la ejecución actual, es decir, los estado son ahora de la forma

```
execute(S,S) :-
   S = st(M,PC,[_Top|_],_),
   bytecode(M,PC,return).
execute(S,Sf) :-
   S = st(M,PC,_,_),
   bytecode(M,PC,Inst),
   step(Inst,S,S'),
   execute(S',Sf).
```

```
step(invoke(M'),S,S') :-
   S = st(M,PC,OS,LV),
   next(M,PC,PC'),
   split_OS(M',OS,Args,OSRs),
   main(M',Args,RV),
   S' = st(M,PC',[RV|OSRs],LV).
```

Figura 2.8: Fragmento del intérprete de bytecode "big-step"

`st(M,PC,OStack,LocalV)`. La pila de llamadas la mantendría ahora el propio Prolog por medio de las llamadas recursivas al predicado `main/3`.

El tratamiento composicional en cuanto a las llamadas a métodos no sólo es esencial para permitir la decompilación modular (solucionando así L1, L2 y L3) sino que también resuelve el problema con la recursión de una manera simple y elegante. De hecho, la decompilación usando el intérprete "big-step" ya no presenta la limitación L4. Por ejemplo, la decompilación de un método recursivo $m1$ empezaría por la llamada $\texttt{main}(\texttt{m1}, \_, \_)$ y llegaría entonces a $\texttt{main}(\texttt{m1}, \texttt{args}, \_)$ donde `args` representaría a los argumentos de la llamada recursiva. Esta llamada sería detectada como peligrosa por el control local y por tanto se pararía la derivación. A diferencia de lo que pasaba anteriormente, no es necesario una segunda evaluación pues la segunda llamada es necesariamente una instancia de la primera, y por tanto, no habrá ninguna perdida de información asociada con la generalización de la pila de llamadas.

Nótese que la idea de utilizar una semántica "big-step" para definir el intérprete y así conseguir una especialización modular es igual de necesario en el caso de la especialización de intérpretes por medio de EP offline. Más aún, esta idea es, por lo que sabemos, nueva y no había sido propuesta antes en ningún contexto, ni en la especialización online ni offline de intérpretes.

2.4.2. El Esquema de Decompilación Modular

Además de usar un intérprete "big-step", para poder diseñar un esquema de decompilación modular, es necesario: (1) proporcionar un mecanismo

```
main(count,[N],0) :- 0>=N.
main(count,[N],I) :-
    0<N, execute_2(N,1,I).

execute_2(N,I,I) :- I>=N.
execute_2(N,A,I) :-
    A<N, A' is A+1,
    execute_2(N,A',I).

main(gcd,[B,0],A) :-
    main(abs,[B],A).
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).
```

```
execute_1(A,0,C) :- main(abs,[A],C).
execute_1(A,B,F) :- B\=0,
                    H is A rem B,
                    execute_1(B,H,F).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[B],A) :- B\=0,
                    C is B-1,
                    main(fact,[C],D),
                    A is B*D.
main(fact,[0],1).
```

Figura 2.9: Código decompilado usando la decompilación modular

para residualizar llamadas en el programa decompilado (es decir, no desplegarlas y añadirlas sin cambios al código residual), y (2) definir la noción de decompilación *separada* y estudiar las condiciones que aseguran su corrección.

El Artículo 6 estudia con detalle estos aspectos y define un esquema de decompilación modular demostrando formalmente su corrección y completitud. También se demuestra que el esquema propuesto satisface el criterio de la *método-optimalidad*, que asegura que cada método es decompilado una sola vez.

La decompilación modular funciona básicamente de la siguiente manera: cuando se va a decompilar una invocación a método, aparece la llamada `step(invoke(m'),_,_)` durante el proceso de desplegado. Utilizando el intérprete "big-step" de la Figura 2.8, se generará una llamada de la forma `main(m',_,_)`. En este punto, habrá una anotación que indica al evaluador parcial que no debe desplegar la llamada y que debe sin embargo dejarla en el código residual sin modificaciones. Si `m'` es *interno* (es decir, está definido en el programa de entrada) se realizará (o ya se habrá realizado) su correspondiente decompilación, pues el esquema de decompilación asegura que la EP se efectúa para todos los métodos del programa.

La Figura 2.9 muestra el programa decompilado que se obtiene utilizan-

do el esquema de decompilación modular sobre nuestro ejemplo motivador. Se puede observar que la estructura del programa original respecto a las llamadas a métodos si se preserva ahora. Por ejemplo, puede verse como en la definición de `gcd` hay una llamada a `abs` como ocurre en el programa bytecode original. Mas aún, ahora si obtenemos una decompilación efectiva para el método recursivo `fact` donde la capa de interpretación se ha eliminado por completo. Concluimos así que todas las limitaciones expuestas anteriormente en esta sección se han resuelto satisfactoriamente.

2.5. Reto III: Un Esquema de Decompilación Óptima

Como mencionamos en la Sección 2.2, y como podemos ver mirando el código de la Figura 2.9, los programas decompilados obtenidos usando el esquema modular no son aún totalmente óptimos pues contienen duplicaciones de código. Ver por ejemplo el código de la parte derecha de las reglas que definen `main(gcd,...)` y `execute_1/3`. Estas duplicaciones normalmente se producen debido a que parte del código se reevalúa durante la fase de EP. Desafortunadamente, como veremos después estas duplicaciones y reevaluaciones crecen exponencialmente con el número de puntos de *divergencia* y *convergencia* respectivamente, y como veremos en la evaluación experimental, degradan mucho la eficiencia del proceso y la calidad del código decompilado. Un aspecto fundamental de esta tesis es estudiar si se puede obtener, por medio de la decompilación interpretativa, programas decompilados cuya calidad sea equivalente a la calidad de los programas obtenidos utilizando decompiladores dedicados (punto (c) de la Figura 2.1). Para poder obtener resultados comparables, tiene sentido que se usen heurísticas similares. El hecho de que los decompiladores habitualmente construyan siempre un *grafo del flujo de control*, "control-flow-graph" (CFG), hace pensar que aplicar una noción similar para controlar la EP de nuestro intérprete pueda resultar útil.

A continuación explicamos el problema a través de un ejemplo. Consideremos el método m_{bl} de la Figura 2.10. El código fuente a la izquierda, el bytecode relevante en el centro y su CFG a la derecha. Como es habitual, el CFG [1] consiste en bloques básicos que contienen una secuencia de

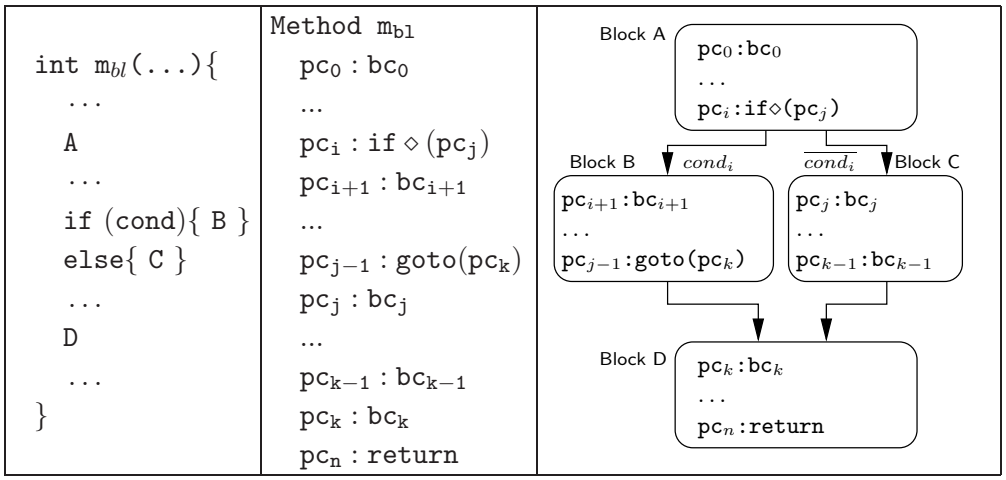


Figura 2.10: Código fuente, bytecode y CFG del método m_{bl}

instrucciones bytecode (sin bifurcaciones), los cuales están conectados por aristas las cuales describen los posibles flujos originados por las distintas instrucciones de bifurcación (como los saltos condicionales, las excepciones, las invocaciones virtuales, etc). En los programas que mostramos, éstas se corresponden simplemente con los saltos condicionales (es decir `if`$\diamond$ y `if0`$\diamond$). Un *punto de divergencia* (punto D) es un punto de programa (índice bytecode) del cual parten más de una rama; de forma similar, un *punto de convergencia* (punto C) es un punto de programa en el cual convergen dos o más ramas. En el CFG de m_{bl}, el único punto de divergencia (resp. convergencia) es pc_i (resp. pc_k).

Utilizando el esquema de decompilación actual se obtendría el árbol SLD de desplegado que aparece en la Figura 2.11, en el que todas las llamadas se han desplegado por completo al no haber ningún riesgo de terminación. El código decompilado correspondiente se muestra en la propia figura bajo el árbol. Usamos $\{\texttt{res}_\texttt{X}\}$ para referirnos al código emitido para el bloque BlockX y $\texttt{cond}_\texttt{i}$ para referirnos a la condición asociada a la instrucción de bifurcación en el índice $\texttt{pc}_\texttt{i}$ ($\overline{\texttt{cond}_\texttt{i}}$ denota su negación). La calidad del código decompilado no es óptima debido a lo siguiente:

D. El código decompilado $\{\texttt{res}_\texttt{A}\}$ para BlockA aparece duplicado en ambas reglas. Durante la EP, este código se ha evaluado sólo una vez pero, debido a la forma en la que se definen los resultantes (ver Sección 2.1), cada regla contiene el código decompilado asociado a la

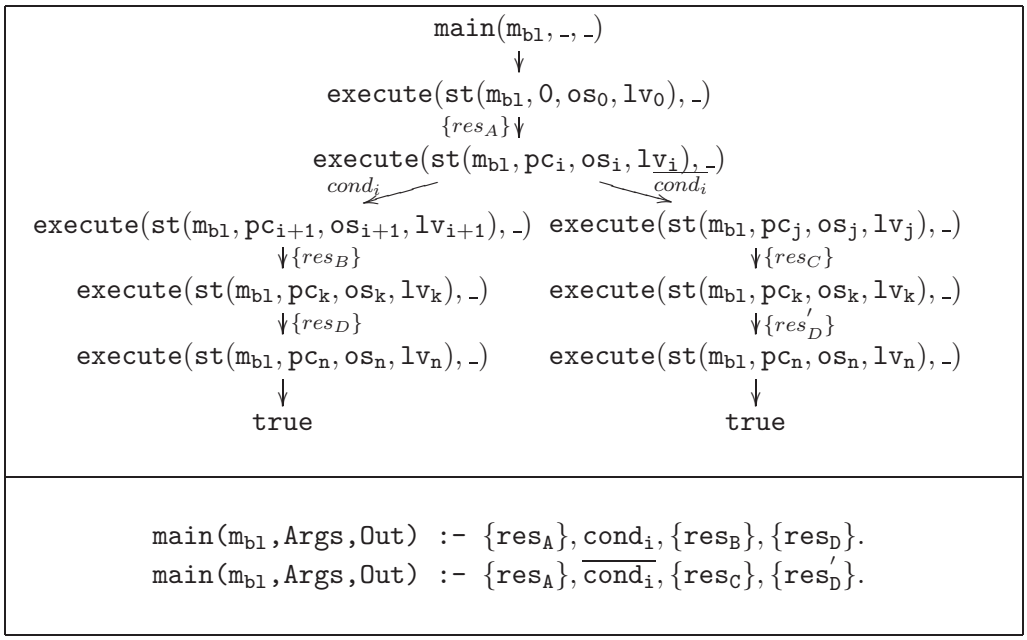


Figura 2.11: Árbol SLD de desplegado y código decompilado para m_{bl}

rama completa correspondiente del árbol. Este tipo de duplicación de código tiene dos consecuencias importantes: aumenta considerablemente el tamaño de los programas decompilados y provoca que su ejecución sea más lenta. Por ejemplo, cuando $\overline{\texttt{cond}_\texttt{i}}$ se cumple, la ejecución habría de pasar necesariamente a través de $\{\texttt{res}_\texttt{A}\}$ de la primera regla, fallaría al evaluar $\texttt{cond}_\texttt{i}$, y probaría después con la segunda regla.

C. El código decompilado para BlockD se emite de nuevo más de una vez. Cada regla del programa contiene ahora una versión (posiblemente diferente), $\{\texttt{res}_\texttt{D}\}$ y $\{\texttt{res}_\texttt{D}'\}$, para el código obtenido al evaluar el bloque BlockD. Ahora, en tiempo de EP, el código de BlockD se evalúa en el contexto de $\{\texttt{cond}_\texttt{i}, \{\texttt{res}_\texttt{B}\}\}$ y se vuelve a evaluar después en el contexto de $\{\overline{\texttt{cond}_\texttt{i}}, \{\texttt{res}_\texttt{C}\}\}$. Por tanto, debido a los puntos de convergencia, tanto la eficiencia del proceso como la calidad de la decompilación se ven seriamente perjudicados.

La cantidad de código residual repetido crece exponencialmente con el número de puntos C y D y la cantidad de código reevaluado crece exponen-

cialmente con el número de puntos C. Tratamos a continuación de definir un esquema de decompilación *óptimo*, y *a nivel de bloques*, que resuelva los problemas D y C. Intuitivamente, una decompilación a nivel de bloques debe producir código residual para cada bloque del CFG. Esto puede conseguirse básicamente haciendo que los árboles SLD de desplegado se correspondan con cada bloque, no expandiéndolos más después de un final de bloque. Nótese que esta idea va en contra de la filosofía habitual de la EP, donde normalmente, para maximizar la cantidad de información estática propagada, se suele tratar de expandir las secuencias lo máximo posible y parar el proceso de desplegado sólo cuando se ponga en peligro la terminación.

Este comportamiento puede conseguirse fácilmente en nuestro esquema simplemente proporcionando anotaciones de forma que se fuerce al proceso de desplegado a parar cuando aparezca en la secuencia un átomo `execute/2` cuyo *PC* se corresponda con un punto D. En el ejemplo, el desplegado debería parar en el punto pc_i. En cuanto al problema C, un requerimiento adicional es que los bloques que comiencen en puntos C deben ser evaluados parcialmente una sola vez. Esto básicamente se puede conseguir de la siguiente manera: (1) parando las derivaciones en las llamadas `execute/2` cuyo *PC* se corresponda con un punto C, y (2) pasando la llamada al control global, y asegurando que ésta se evalúa en un contexto suficientemente generalizado de forma que se cubran todos los posibles contextos en los que se evalúa dicha llamada. El primer punto se asegura proporcionando al evaluador parcial las correspondientes anotaciones, mientras que el segundo se asegura incluyendo en el conjunto inicial de átomos pasado a la EP, una llamada generalizada de la forma $\texttt{execute}(\texttt{st}(\texttt{m}_{\texttt{bl}}, \texttt{pc}_{\texttt{k}}, \_, \_), \_)$ para cada punto C. Obsérvese que, tanto las anotaciones, como el conjunto inicial de llamadas pueden calcularse automáticamente simplemente haciendo dos pasadas sobre el programa bytecode (ver por ejemplo [2, 84]).

El código resultante utilizando el esquema de decompilación óptima para el método m_{bl} se muestra en la Figura 2.12. Ahora, el código residual asociado a cada bloque aparece una sola vez, asegurando que se preserva la forma del CFG, como hacen los decompiladores dedicados. Conseguimos así obtener programas cuya calidad es equivalente a la obtenida usando decompiladores dedicados [2, 70], pero preservando las ventajas de la decompilación interpretativa.

```
main(m_bl,Args,Out) :- {res_A}, execute_1(...).
execute_1(...) :- cond_i, {res_B}, execute_2(...).
execute_1(...) :- ¬cond_i, {res_C}, execute_2(...).
execute_2(...) :- {res_D}.
```

Figura 2.12: Código decompilado óptimo para el método m_{bl}

Estos aspectos se estudian en detalle en el Artículo 6 donde se define formalmente dicho esquema de decompilación. Se demuestra también formalmente que el esquema propuesto satisface el criterio de la *bloque-optimalidad*, que asegura que: (I) el código residual para cada instrucción se emite una sola vez en el código decompilado, (II) cada instrucción bytecode se evalúa como máximo una vez durante la EP, y (III) hay como máximo una regla residual por cada bloque del programa bytecode.

2.5.1. Conclusiones de la Decompilación Óptima

Una vez se tiene en cuenta la observación principal de la Sección 2.4 de que el intérprete se debe escribir usando una semántica "big-step", cada una de las condiciones del criterio de la *bloque-optimalidad* puede resultar más o menos complicada de asegurar dependiendo de la estrategia de control local utilizada. Por ejemplo, si empezamos con un decompilador modular como el expuesto en la Sección 2.4, la condición (III) se cumplirá en general, pero no así las condiciones (I) ni (II) pues la regla de control local tiende a sobreespecializar las llamadas, lo que resulta en duplicaciones y reevaluaciones de código.

Por otro lado, si se utilizara un evaluador parcial offline, la regla de control local natural residualizaría todas las llamadas a `execute`, y filtraría en el control global toda la información del estado excepto la signatura del método y el contador de programa. Esta estrategia de control garantiza trivialmente las condiciones (I) y (II), pues asegura que cada instrucción bytecode se decompila de forma independiente. Sin embargo, tiende a ser demasiado conservadora, y, en particular, no satisface la condición (III), pues tan pronto como se encuentre con un bloque que tenga más de una instrucción bytecode (lo que ocurre casi siempre), el programa especializado

generará una regla para cada instrucción bytecode del bloque. Como resultado, el programa residual obtenido es de *alto nivel* en el sentido de que está escrito en Prolog. No obstante, su estrategia de control está altamente influenciada por el hecho de que estamos decompilando desde un programa bytecode (y no por ejemplo desde un programa fuente Java), y el programa obtenido no se parece para nada al programa Prolog que un programador Prolog podría escribir para realizar la misma tarea. Pues uno de los objetivos importantes de la decompilación es facilitar su comprensión y su análisis, en esta tesis sostenemos que los programas que cumplen el criterio de la *bloque-optimalidad*, y en particular aquellos que cumplen la condición (III), como los que se generan usando nuestro esquema de decompilación óptima, son más fáciles de tratar.

Otra observación importante es que los costosos mecanismos utilizados para controlar la EP, usados anteriormente para obtener los resultados de las Secciones 2.3.3 y 2.4, en particular el TbHEm y el control avanzado de polivarianza del Artículo 2, ya no son necesarios al utilizarse el esquema de decompilación óptima. Se pueden ahora usar los siguiente operadores triviales de control: unfold despliega todas las llamadas excepto aquellas que se correspondan con una anotación, y abstract añade al conjunto S_{i+1} todas las llamadas en L^{pe} que no sean una instancia de ninguna llamada en S_i (ver el algoritmo genérico de la Sección 2.1). Se puede demostrar fácilmente que la terminación está garantizada, tanto a nivel local como global gracias a las anotaciones y al conjunto inicial de átomos proporcionados en la EP.

2.6. Implementación y Resultados Experimentales

El Artículo 6 discute varios detalles de implementación y realiza una evaluación experimental exhaustiva de los diferentes esquemas de decompilación propuestos. Hemos llevado a cabo dos implementaciones distintas de un decompilador de Java Bytecode (secuencial) a Prolog. En la primera, hemos extendido un evaluador parcial online existente, aquel integrado en el sistema `CiaoPP`. Éste es un evaluador parcial muy potente y implementa reglas de desplegado y operadores de abstracción. Esto nos ha permitido comparar los diferentes esquemas de decompilación, y en particular

comparar respecto a los esquemas no óptimos. Sin embargo, la sobrecarga introducida al utilizar una herramienta tan potente y genérica no nos permite competir, respecto a eficiencia, con decompiladores dedicados. Por ello, hemos llevado a cabo una segunda implementación para la cual hemos escrito un evaluador parcial autocontenido que sólo contiene las estrategias de control necesarias para el esquema óptimo. Éste evaluador parcial ha sido integrado en una herramienta de decompilación, llamada jbc2prolog, la cual incluye también un intérprete de Java Bytecode. Esto ha hecho posible obtener decompilaciones óptimas y al mismo tiempo competir en términos de eficiencia respecto a decompiladores dedicados. El Artículo 6 realiza una comparación exhaustiva frente al decompilador del sistema COSTA [5] y al decompilador de Java JDec [14].

Ambas implementaciones consideran el lenguaje Java Bytecode (secuencial) al completo. Las extensiones necesarias para poder tratar los aspectos del lenguaje no tratadas en esta introducción se discuten en el Artículo 6. Éstas incluyen a las excepciones, operaciones del heap, invocaciones virtuales, decompilación al nivel de clases, etc. Todas ellas han sido fácilmente integradas en nuestro esquema de decompilación, en la mayoría de los casos, simplemente las funcionalidades correspondientes en el intérprete de bytecode.

Para la evaluación experimental del Artículo 6, hemos utilizado el conjunto estándar de "benchmarks" JOlden [55]. En particular, nos hemos interesado en: a) demostrar empíricamente la escalabilidad del enfoque, y b) comprobar la eficiencia de la herramienta implementada respecto a otros decompiladores. Concluimos lo siguiente:

- **Escalabilidad:** Mientras que en la decompilación no-óptima los tiempos de decompilación y los tamaños de los programas decompilados crecen de forma muy significativa con el tamaño de los "benchmarks", esto no ocurre en el esquema óptimo. Con la decompilación óptima, estos valores se mantienen totalmente estables. Mostramos que tanto los tiempos de decompilación como los tamaños de los programas decompilados crecen de forma *lineal* con respecto al tamaño de los programas bytecode de entrada, demostrando así la escalabilidad de la decompilación óptima.

- **Eficiencia:** Para demostrar la eficiencia de nuestro esquema, hemos

comparado los tiempos de decompilación obtenidos usando nuestra herramienta jbc2prolog frente a aquellos obtenidos usando el decompilador del sistema COSTA, y , a aquellos obtenidos con el decompilador JDec [14]. Podemos concluir que los resultados son competitivos respecto a aquellos obtenidos con decompiladores dedicados. En particular, observamos que son bastante similares a los obtenidos con COSTA. Mas aún, en la mayoría de ejemplos, podemos observar que jbc2prolog es cerca de diez veces más rápido que JDec. Nuestra conclusión en este sentido es que es muy difícil comparar decompiladores escritos en diferentes lenguajes de programación y más aún que decompilan a diferentes lenguajes.

2.7. Trabajo Relacionado

Los trabajos previos sobre decompilación interpretativa se han centrado básicamente en demostrar que el enfoque es viable para pequeños y medianos intérpretes y lenguajes. Principalmente han tratado de demostrar su *efectividad*, es decir, que la llamada capa de interpretación se puede eliminar de los programas compilados. Para ello se han usado técnicas de EP offline [63], online [48, 77] y híbridas [64]. Esta tesis se ha centrado en, primeramente, demostrar la viabilidad del enfoque para un lenguaje bytecode con orientación a objetos, para después estudiar cuestiones más avanzadas como su escalabilidad y la calidad de las decompilaciones, las cuales no se habían estudiado hasta ahora. Los trabajos sobre decompilación interpretativa ya se han ido comparando en las diferentes secciones del capítulo y en la introducción. Revisamos a continuación el trabajo relacionado en el campo de la decompilación.

Se puede realizar decompilación a diferentes niveles, con sus correspondientes grados de precisión y éxito. El caso más complicado es sin duda la decompilación de *ejecutables binarios*. Hay una serie de complicaciones como por ejemplo la dificultad a la hora de recuperar el flujo de control. Un problema intrínseco es la imposibilidad de distinguir entre el código de los datos de forma estática. Ver por ejemplo [26, 81] y sus referencias donde se discuten los problemas y las técnicas que se aplican en la decompilación de ejecutables binarios. El siguiente nivel es la decompilación de *código ensamblador* [27]. En este contexto, muchas de las complicaciones de la

decompilación de binarios se siguen presentando, aunque al menos se suele poder separar el código de los datos. Un nivel más arriba se encontraría la decompilación de código a ser ejecutado por una máquina virtual, como el bytecode. Esto es en general más sencillo, pues las máquinas virtuales son generalmente más sencillas que las arquitecturas hardware. Además, estos programas suelen cumplir varias restricciones, como que por ejemplo sean *verificables* [60] o que los tipos de las variables estén disponibles. Como resultado, en el caso particular de la decompilación de Java Bytecode, hay un buen número de herramientas de decompilación capaces de tratar una amplia clase de programas bytecode, especialmente aquellos generados por compiladores de Java, por ejemplo `javac`. No obstante, las cosas se pueden complicar bastante cuando el java Bytecode se ha generado con un *obfuscador*, y especialmente cuando se ha utilizado un compilador optimizante o un compilador de un lenguaje distinto de Java como Haskell, ML, Ada, etc. Ver por ejemplo [72] y sus referencias para una discusión más detallada sobre decompiladores de Java Bytecode y las dificultades a las que éstos se enfrentan.

Como hemos mencionado anteriormente, existen varios analizadores de Java Bytecode que transforman el bytecode en algún tipo de representación intermedia de más alto nivel, y por tanto pueden verse como decompiladores dedicados. En particular, los sistemas COSTA [5] y `CiaoPP` [49] convierten bytecode a una representación que es usada después como entrada para la fase de análisis. Aunque en ambos casos la representación utilizada es similar, en el caso de COSTA se formaliza como una representación basada en reglas [2], mientras que en `CiaoPP` se formaliza como cláusulas de Horn, es decir, como un programa lógico [70]. Esto se hace en `CiaoPP` para así poder usar directamente los análisis disponibles para programas CLP de `CiaoPP`.

Hay sin embargo una diferencia crucial entre los programas generados en [70] o [5], y los generados por nuestro decompilador. Mientras que los programas de [70] y [5] están exclusivamente pensados para ser analizados, no siendo por tanto ejecutables, los programas que nosotros generamos pueden tanto ser analizados como ejecutados. La razón de esto es que los primeros, aunque representan el flujo de control de los programas bytecode con reglas, dejan las instrucciones bytecode como "builtins", es decir, predicados predefinidos, que el análisis posteriormente interpreta. Producir programas ejecutables es algo no trivial, pues muchas de las instrucciones

bytecode operan con el heap de una forma u otra. Por tanto, para conseguir una decompilación CLP ejecutable, se debe introducir el heap de la JVM explícitamente en el programa CLP. Esto se consigue de forma automática en nuestro enfoque.

Capítulo 3

Aplicaciones de la Decompilación Interpretativa

Como ya se mencionó en el capítulo anterior, una de las ventajas importantes de la decompilación interpretativa es que los programas que obtenemos son totalmente *ejecutables*, lo que amplia su campo de aplicación. En este capítulo, resumimos dos estudios experimentales llevados a cabo que tratan de aprovechar dicha ventaja de los programas decompilados:

1. Análisis de programas bytecode analizando sus decompilaciones a LP utilizando herramientas de análisis de LP. Este punto se elabora en el Artículo 1.

2. *Generación de datos de prueba* para programas bytecode por medio de evaluación parcial en CLP. Esto se estudia con detalle en el Artículo 7.

3.1. Análisis de Bytecode utilizando Herramientas de Análisis LP

Analizar programas en el paradigma de la programación lógica ofrece una serie de ventajas, quizás la más importante sea la madurez y sofisticación de las herramientas de análisis ya disponibles en dicho contexto. En particular, el sistema `CiaoPP`, aparte de proporcionar el potente evaluador parcial que hemos utilizado para realizar parte de los experimentos en el capítulo anterior, proporciona un motor genérico de análisis con un buen

número de dominios abstractos disponibles. Esto permite inferir una gran cantidad de propiedades de los programas lógicos como *terminación*, cotas en el consumo de recursos, *tipos* y *modos*, *ausencia de errores*, etc.

Uno de los objetivos de esta tesis ha sido investigar la posibilidad de reutilizar herramientas existentes en el paradigma de la CLP, en particular `CiaoPP`, para analizar programas bytecode a base de analizar sus decompilaciones a LP. Esto permitiría diseñar un esquema de análisis y verificación de programas bytecode en el cual la potencia de las herramientas de análisis de CLP se transfiere automáticamente al análisis y verificación de programas bytecode. Esta misma idea había sido aplicada para analizar versiones muy reducidas de lenguajes imperativos de alto nivel [77] y también código ensamblador del procesador PIC [47], un microprocesador de 8 bits. Sin embargo, por lo que conocemos, esta es la primera vez que este enfoque se ha aplicado con éxito para un lenguaje imperativo bytecode, real y de propósito general.

Dicho estudio se presenta en el Artículo 1, donde: (1) proponemos dicho esquema para el análisis y verificación de programas bytecode (en particular para Java Bytecode), y (2) se realizan una serie de experimentos utilizando el sistema `CiaoPP` demostrando así la viabilidad práctica del enfoque propuesto. En resumen, el Artículo 1 muestra como, razonando sobre nuestros programas decompilados, utilizando para ello los análisis disponibles en el sistema `CiaoPP`, podemos demostrar automáticamente propiedades no triviales de los programas bytecode como terminación, ausencia de errores en tiempo de ejecución e inferir cotas en el consumo de recursos. Por ejemplo, para demostrar ausencia de errores en tiempo de ejecución, proponemos instrumentar un intérprete de bytecode, que además de computar el valor de retorno del método llamado, también calcula la *traza de ejecución*, la cual captura la historia de la ejecución. Dichas trazas representan los pasos semánticos dados, y por tanto no sólo representan las instrucciones ejecutadas, sino que representan también cierta información de contexto. Éstas nos permiten distinguir, para una misma instrucción bytecode, si el correspondiente paso lanza una excepción o se ejecuta normalmente. Por ejemplo, `invoke_step_ok` y `invoke_step_NullPointerException` representan respectivamente la llamada a método *normal* y la llamada sobre una referencia *null* que lanza una excepción. Esta flexibilidad adicional de la decompilación interpretativa nos ha permitido demostrar ausencia de

errores en tiempo de ejecución de una forma casi directa, simplemente especificando la propiedad de "error-free" basada en las trazas. Una traza es "error-free" si no contiene ningún paso de excepción, o si no termina lanzando una excepción. Esto, en cualquier caso, dependerá de la *política de seguridad* utilizada para la propiedad "error-freeness". De nuevo, nuestro enfoque demuestra su flexibilidad en este punto, pues se pueden definir fácilmente diferentes políticas, simplemente especificando la propiedad correspondiente en `CiaoPP`.

3.2. Generación de Datos de Prueba por EP en CLP

Una característica especial de nuestros programas decompilados es que éstos representan el estado *completo* del programa, a diferencia de otros enfoques [70, 2, 84] Hasta ahora, la principal motivación de decompilar bytecode a LP había sido el ser capaz de realizar análisis estático sobre los programas decompilados, para poder así obtener propiedades de los programas bytecode. Si la decompilación produce programas LP que son ejecutables, entonces dichos programas decompilados pueden usarse no solo para ser analizados estáticamente, sino también para análisis dinámico y ejecución. Nótese que no siempre ocurre esto, pues en otros enfoques (como [4, 70]) las transformaciones están exclusivamente pensadas para el análisis estático, y por tanto los programas no pueden ejecutarse. Una aplicación novedosa de la decompilación interpretativa que proponemos en esta tesis es la *generación automática de datos de prueba*.

La *generación de datos de prueba*, *test data generation* (TDG), trata de generar automáticamente casos de prueba para un determinado *criterio de recubrimiento*. El criterio de recubrimiento mide lo que se ejercita el programa por el conjunto dado de casos de prueba. Ejemplos de criterios de recubrimiento son: *recubrimiento de instrucciones*, que requiere que cada instrucción del programa se ejecute; *recubrimiento de caminos*, que requiere que cada posible traza del código se ejercite, etc. Existe una gran variedad de enfoques para realizar TDG (ver [90] para un resumen). Nuestro trabajo se centra en TDG de tipo "glass-box", en el que los casos de prueba se generan a partir del programa concreto, en lugar de generarse a partir de

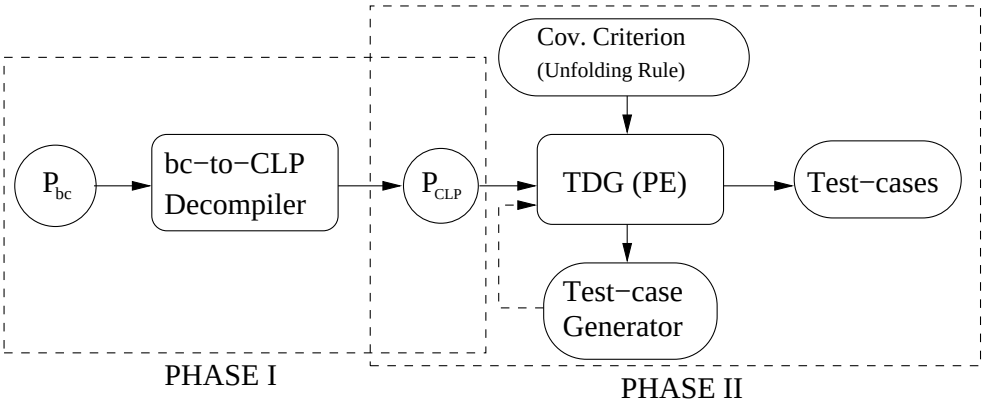


Figura 3.1: Visión general del enfoque de TDG de bytecode por EP en CLP

una especificación de éste. Además, nos centraremos en TDG estático, en el cual no se asume ningún conocimiento de los datos de entrada, a diferencia de los enfoques dinámicos [39, 46], en los cuales el programa es en algún momento ejecutado con valores de entrada concretos.

El enfoque estándar para generar casos de prueba estáticamente consiste en realizar una *ejecución simbólica* del programa (ver por ejemplo [29]), donde los contenidos de las variables son expresiones en lugar de valores concretos. La ejecución simbólica produce finalmente un sistema de *restricciones*, las cuales definen las condiciones bajo las que se ejecutan los distintos caminos. Esto ocurre por ejemplo, en las instrucciones condicionales, como en los "if-then-else", donde se suele requerir generar casos de prueba para las dos alternativas, y por tanto se han de acumular las condiciones de cada camino como restricciones. Para el caso de Java Bytecode, en [73] se diseñó una JVM simbólica (SJVM), la cual integraba varios resolutores de restricciones. La SJVM requiere una serie de extensiones no triviales respecto a la JVM: (1) necesita que el bytecode sea ejecutado simbólicamente como explicamos anteriormente, y (2) debe ser capaz de realizar "backtracking", pues al no conocerse los datos de entrada, el motor de ejecución debe considerar todas las alternativas. El mecanismo de "backtracking" utilizado en [73] es de hecho esencialmente el mismo al utilizado en la programación lógica.

Esta tesis propone un esquema novedoso de TDG de bytecode basado en técnicas de EP en CLP, el cual, a diferencia de trabajos previos, no requiere

el desarrollo de una máquina simbólica virtual. La Figura 3.1 muestra un diagrama con el esquema general. Como podemos ver, el enfoque se basa en dos fases independientes de EP en CLP, que consisten básicamente en lo siguiente:

1. *Decompilación del bytecode en un programa CLP.* Ya hemos explicado en el Capítulo 2 que la decompilación de bytecode a LP se puede obtener automáticamente por medio de la EP de LP, o alternativamente utilizando un decompilador dedicado [70]. Las modificaciones en el esquema para obtener programas CLP en lugar de programas LP son prácticamente triviales. Esto se puede conseguir básicamente transformando los "builtins" aritméticos del intérprete por los correspondientes "builtins" CLP de la librería correspondiente.

2. *Generación de casos de prueba.* Ésta es una aplicación novedosa de la EP que nos permite generar *generadores de casos de prueba* a partir de los programas CLP decompilados. En este caso, utilizaremos un evaluador parcial CLP capaz de propagar restricciones de la misma manera que haría una máquina simbólica. Los operadores de control de la EP juegan un papel esencial: (1) El control local permite capturar fácilmente diferentes criterios de recubrimiento. (2) El control global permite la generación de *generadores de casos de prueba*. Intuitivamente, éstos son programas CLP cuya ejecución en CLP devuelve más casos de prueba bajo demanda, sin la necesidad de empezar el proceso de TDG de nuevo.

Esta tesis sostiene que este enfoque de TDG de bytecode tiene varias ventajas importantes respecto a enfoques previos basados de alguna u otra manera en ejecución simbólica. Éstas incluyen: (i) Es más *genérico*, pues la mismas técnicas se pueden aplicar para otros lenguajes (imperativos) de entrada. En particular, una vez se la realizado la decompilación a CLP, las características del lenguaje las características del lenguaje quedan abstraídas, siendo por tanto la fase de generación de datos de prueba totalmente *independiente del lenguaje*. Esto evita tener que tratar con aspectos como la recursión, las llamadas a procedimientos, la memoria dinámica, etc. (ii) Es más *flexible* pues es muy fácil incorporar diferentes criterios de recubrimiento simplemente proporcionando las correspondientes reglas de control local a la EP. (iii) Es más potente gracias a la característica

de poder generar generadores de casos de prueba. (iv) Es más simple de implementar pues no requiere el desarrollo de ninguna máquina simbólica, asumiendo claro que se dispone de una evaluador parcial.

Como se acaba de mencionar en la ventaja (iv), una de las ventajas de los programas decompilados CLP respecto a sus versiones bytecode es que se puede realizar una ejecución simbólica de éstos sin necesidad de escribir un mecanismo específico de ejecución simbólica. Simplemente podemos ejecutar el programa decompilado usando el mecanismo de ejecución estándar de CLP, poniendo variables en todos los argumentos del correspondiente predicado. Por ejemplo, para nuestro ejemplo motivador de la Figura 2.4, podríamos ejecutar simbólicamente el método `gcd` lanzando el objetivo $\texttt{main}(\texttt{gcd}, [\texttt{X}, \texttt{Y}], \texttt{Z})$ sobre el programa decompilado. Los resultados obtenidos (restricciones sobre las variables) se pueden interpretar como las condiciones que han de cumplir las variables de entrada (en este caso X e Y) para seguir el camino de ejecución correspondiente. La solución de dichas restricciones nos daría por tanto datos de entrada concretos.

Sin embargo, un problema importante de la ejecución simbólica, independientemente de si se realiza en CLP o utilizando una máquina simbólica, es que el árbol de ejecución a ser recorrido, es en la mayoría de los casos infinito, pues los programas suelen contener construcciones iterativas y recursiones las cuales suelen inducir un número infinito de caminos de ejecución al ejecutarse sin valores concretos. Es por tanto esencial establecer un *criterio de terminación*, en este contexto *criterio de recubrimiento*, que garantice que el número de caminos a ser recorrido es finito, y al mismo tiempo que se obtiene un conjunto interesante de casos de valores de entrada.

La mayoría de criterios de recubrimiento están definidos sobre lenguajes de programación estructurados y de alto nivel. Un criterio basado en el flujo de control ampliamente utilizado es el loop-count(k) [53], el cual limita a una cantidad k el número de veces que se puede iterar en los bucles. No obstante, el bytecode tiene un flujo sin estructura, cuyos CFG's pueden variar mucho en forma. Es por ello que en esta tesis hemos introducido el criterio de recubrimiento *block-count*(k) . Éste, en lugar de limitar el número de veces que se itera en bucles, cuenta el número de veces que se visita cada bloque durante cada computación. Básicamente, un conjunto de caminos de computación satisface el criterio *block-count*(k) si éste incluye

todos los caminos de computación terminados en los que el número de veces que se visita cada bloque no excede la k dada.

En el Artículo 7 se discuten los detalles técnicos de dicho enfoque de TDG de bytecode. En particular:

- Se define formalmente el criterio *block-count*(k) .
- Se define una *estrategia de evaluación* que garantiza construir un árbol SLD de forma que se generen suficientes derivaciones para cumplir el criterio block-count(k), asegurando al mismo tiempo la terminación del proceso.
- La fase de TDG se formaliza como una EP en CLP del programa CLP decompilado donde la regla de desplegado juega el papel del criterio de recubrimiento. Definimos además una regla de desplegado que implementa el criterio de recubrimiento block-count(k) y describimos como debe el operador de abstracción tratar con restricciones para poder obtener generadores de casos de prueba efectivos.
- Todos estos aspectos se ilustran a través de un ejemplo que consiste en una serie de métodos que realizan diferentes cálculos aritméticos.

3.2.1. Generando Datos de Prueba para Prolog por EP

Como contribución tangencial de esta tesis, hemos aplicado la idea de utilizar EP para generar automáticamente datos de prueba en el contexto de la LP. Ya mencionamos que nuestro enfoque podría utilizarse en principio para hacer TDG de cualquier lenguaje imperativo. Sin embargo, al tratar de aplicarlo a un lenguaje declarativo como Prolog, encontramos problemas a la hora de generar datos de prueba que cubran ciertos flujos de control de Prolog. Básicamente, el problema es que una característica intrínseca de la EP es que sólo computa derivaciones no fallidas, mientras que en la TDG de Prolog es esencial generar casos de prueba asociados a derivaciones de fallo. En el Artículo 8 hemos realizado un estudio preliminar en este dirección, en el que se propone transformar el programa Prolog original en un programa Prolog equivalente con *fallo explícito*. Esto puede hacerse evaluando parcialmente un intérprete Prolog que captura las derivaciones de fallo del

programa. Otro aspecto importante que se discute en el Artículo es que, mientras que en el caso del lenguaje bytecode considerado anteriormente, el dominio de restricciones sólo necesitaba manipular números enteros, en Prolog éste debe tratar adecuadamente los datos simbólicos que maneja el programa. Nuestros experimentos preliminares sugieren que el enfoque de TDG basado en EP propuesto en la sección puede ser también útil para la generación automática de casos de prueba para Prolog.

3.2.2. Trabajo Relacionado en la Generación de Datos de Prueba

Como hemos mencionado anteriormente, nuestro enfoque se centra en TDG estático, en el que los casos de prueba se generan sin ejecutar realmente el programa con datos particulares. Por el contrario, los enfoques *dinámicos* [39, 46] ejecutan el programa para ciertos valores de entrada concretos hasta conseguir satisfacer el criterio de recubrimiento correspondiente. El enfoque estándar para generar casos de prueba estáticamente es la *ejecución simbólica* [29, 71, 73, 57, 45], Ésta se ha combinado con el uso de *resolutores de restricciones* en [73, 45] para: tratar los sistemas de restricciones resolviendo la viabilidad de los caminos y después instanciar las variables de entrada. Para el caso particular de Java Bytecode, en [73] se propone una JVM simbólica que integra varios resolutores de restricciones.

La TDG para lenguajes declarativos ha recibido comparativamente mucha menos atención que para lenguajes imperativos. La mayoría de las herramientas existentes para lenguajes funcionales son de tipo "black-box", es decir, generan los casos de prueba a partir de la especificación del programa (ver por ejemplo [28]). Una excepción es [40] donde se propone un enfoque de tipo "glass-box" para el lenguaje Curry. En el caso de CLP, se han generado casos de prueba para Prolog en [69, 13, 89]; y más recientemente para Mercury [36]. Básicamente, para obtener los casos de prueba, primeramente calculan las restricciones sobre las variables de entrada asociadas con los diferentes caminos de computo considerados, y después resuelven las restricciones para obtener valores concretos. Por otro lado, se han definido criterios de recubrimiento específicos para lenguajes funcionales [40] en los que se consideran aspectos del lenguaje como la *pereza* ("laziness").

En general, los lenguajes declarativos plantean diferentes problemas re-

lacionados con sus propios modelos de ejecución –como la *pereza* en lenguajes funcionales o el *fallo* en lenguajes lógicos (con restricciones)– los cuales han de ser tratados por los correspondientes criterios de recubrimiento. Una vez dicho esto, pensamos que las ideas relacionadas con el uso de técnicas de EP para generar generadores de casos de prueba, así como el uso de reglas de desplegado para supervisar la evaluación, se pueden adaptar para lenguajes declarativos como hemos mostrado en nuestros resultados preliminares del Artículo 8.

Capítulo 4

Análisis del Consumo del Heap para Bytecode

Predecir la cantidad de memoria que un programa requiere para su ejecución es crucial en muchos contextos, como en aplicaciones *empotradas*, donde suele haber fuertes restricciones de espacio, o en sistemas de tiempo real, que han de responder a eventos tan rápido como sea posible. Se sabe también que la estimación del uso de memoria también es importante para una predicción precisa del tiempo de ejecución, pues los fallos de página y de memoria *cache* contribuyen significativamente al tiempo de ejecución.

El análisis del consumo del heap trata de inferir *cotas* en el consumo del heap de los programas. Como es habitual, éste se ha formulado típicamente al nivel del código fuente (ver por ejemplo[83, 50, 85, 54] en el contexto de la programación funcional y [52, 23] para lenguajes imperativos de alto nivel). Como mencionamos en el Capítulo 1.4, hay sin embargo situaciones en las que no se tiene acceso al código fuente. El análisis del consumo del heap tiene aplicaciones interesantes en este contexto. Por ejemplo, la *certificación de cotas de recursos*, "resource bound certification" [33, 8, 10, 51, 22], propone utilizar propiedades de seguridad incluyendo requerimientos de coste, es decir, el código recibido ha de adherirse a unos requerimientos específicos respecto a su consumo de memoria. También, las cotas en el consumo del heap pueden resultar útiles en sistemas empotrados ("embedded systems"), por ejemplo, en tarjetas inteligentes en las cuales la memoria es limitada y no puede recuperarse de forma sencilla.

Recientemente, en [3] se ha propuesto un análisis de coste para Java

Bytecode secuencial, dando lugar al sistema COSTA [5]. Dicho análisis genera estáticamente *relaciones de coste* (*CRs*) que definen el coste del programa como una función de los tamaños de sus argumentos de entrada. Estas relaciones, se expresan por medio de *ecuaciones recursivas* generadas abstrayendo la estructura recursiva del programa e infiriendo relaciones entre los argumentos. El análisis es paramétrico respecto al *modelo de coste*, el cual define la unidad de coste asociada con cada instrucción bytecode de programa.

Esta tesis desarrolla una aplicación novedosa del análisis de coste propuesto en [3] para inferir cotas en el consumo del heap de programas Java Bytecode:

1. En un primer paso, desarrollamos un modelo de coste que define el coste de cada instrucción de alojamiento de memoria (`new`, `newarray`, etc) en términos del número de unidades del heap que consumen. Por ejemplo, el coste de crear un nuevo objeto será el número de unidades de heap alojadas por el objeto. El resto de las instrucciones bytecode no añaden ningún coste. Con este modelo, generamos *CRs*, y las usamos para inferir cotas del consumo de memoria de los diferentes métodos del programa. Estas cotas, proporcionan información de la cantidad máxima de heap que se requiere para ejecutar cada método.

2. Desafortunadamente, en el caso de lenguajes con *recolección automática de basura*, "garbage collection" (GC), este enfoque, aunque es correcto, produce estimaciones demasiado pesimistas. Es por ello, que en un segundo paso, refinamos el análisis para que se considere el efecto del GC. Proponemos por tanto un *análisis del consumo de la memoria activa*, que aproxima la cantidad máxima de heap usada durante la ejecución de un programa, proporcionando una estimación mucho más precisa en presencia del GC. Para ello, nos basamos en un *análisis de escape* [17] para identificar aquellos objetos creados en un método, que serán recolectados antes de salir de él. Con esta información, inferimos cotas de la *memoria escapada* en la ejecución del método, es decir, la memoria que se aloja durante la ejecución del método, *y que* permanece ocupada tras su finalización. Proponemos entonces una nueva forma de *CRs* del *consumo pico* del heap, que capturan el consumo pico de la ejecución del programa sobre todos

sus posibles estados. Una característica esencial de nuestras *CRs*, es que éstas pueden resolverse utilizando resolutores existentes, en particular el propuesto en [9].

Estos aspectos se introducen respectivamente en las Secciones 4.1 y 4.2, y se estudian en detalle en los Artículos 9 y 10.

Una característica única de los análisis presentados en esta tesis con respecto a trabajos previos (por ejemplo [10, 50, 18, 24]), es que éstos no están limitados a cotas lineales ni polinómicas, pues nuestras *CRs* pueden en principio capturar cualquier clase de complejidad. Más aún, en la mayoría de los casos, utilizando el resolutor de *CRs* del sistema COSTA, nuestras relaciones pueden simplificarse a una forma cerrada, lo que nos da información directa sobre el consumo del código en cuestión.

Una observación importante es que estos análisis se podrían también haber desarrollado de forma similar utilizando nuestros programas decompilados LP. De hecho, COSTA decompila también el bytecode a una representación basada en reglas antes de realizar el análisis propiamente dicho, con el propósito de simplificar el diseño (ver [3] para más detalles). Las representaciones intermedias de COSTA son de hecho muy similares a nuestros programas decompilados, con la diferencia fundamental de que, en COSTA, prácticamente, todas las instrucciones de bytecode quedan residuales en el código como "builtins", es decir predicados predefinidos. Por el contrario, en nuestras decompilaciones, las instrucciones de bytecode se interpretan y evalúan en tiempo de decompilación, y se convierten, en su caso, a instrucciones básicas de Prolog, como unificaciones y operaciones aritméticas. La razón fundamental por la que decidimos no usar nuestras decompilaciones para el análisis de consumo de memoria, es que de esta manera hemos sido capaces de integrar nuestro análisis en el sistema COSTA, aprovechando así toda la maquinaría para el análisis de coste incluida en él (por ejemplo, el *análisis de tamaños* que infiere las relaciones de tamaños entre argumentos, el resolutor de *CRs*, etc).

4.1. Análisis del Consumo Total

Consideremos el programa Java que aparece en la Figura 4.1. Consiste en una serie de clases Java que definen una estructura de datos del tipo

```
abstract class List {
  abstract List copy();
}

class Nil extends List {
  List copy() {
    return new Nil();
  }
}

class Cons extends List {
  int elem;
  List next;

  List copy(){
    Cons aux = new Cons();
    aux.elem = m(this.elem);
    aux.next = this.next.copy();
    return aux;
  }

  static int m(int n) {
    Integer aux = new Integer(n);
    return aux.intValue();
  }
} // class Cons
```

Figura 4.1: Ejemplo de consumo de memoria

lista enlazada, implementada en un estilo fuertemente orientado a objetos. La clase `Cons` se utiliza para los nodos de datos (en este caso números enteros), y la clase `Nil` juega el papel de *null* para indicar el final de la lista. Tanto `Cons` como `Nil` heredan de la clase abstracta `List`. Así, los objetos del tipo `List` pueden ser bien instancias de `Cons` o de `Nil`. Ambas subclases implementan el método `copy`, el cual se utiliza para *clonar* el objeto correspondiente. En el caso de `Nil`, `copy` simplemente devuelve una nueva instancia de sí mismo, pues es el último elemento de la lista. En el caso de `Cons`, se devuelve una instancia clonada donde el dato se clona invocando al método estático `m`, y la continuación se clona llamando recursivamente al método `copy` sobre el objeto `next`.

Nuestro análisis de consumo del heap infiere relaciones de coste (simplificadas) para el método `copy` de la clase `Cons`:

$$C_{copy}(a) = 12, \quad a = 1$$
$$C_{copy}(a) = 12 + C_{copy}(a-1), \quad a > 1$$

las cuales se pueden resolver usando el resolutor de COSTA dando como resultado la siguiente cota en forma cerrada:

$$C_{copy}(a) = 12*nat(a-1) + 12$$

Se puede observar que el consumo de heap es lineal respecto al parámetro de entrada `a`, que se corresponde con el tamaño del objeto *this* del método,

es decir, la longitud de la lista a clonar. Esto ocurre gracias a que la abstracción utilizada por nuestro análisis para referencias a objetos es la *longitud de la cadena de referencias más larga*, que en este caso se corresponde con la longitud de la lista. La constante numérica 12 se obtiene sumando 8 y 4, siendo 8 el número de bytes ocupados por una instancia de la clase `Cons`, y 4 los bytes ocupados por una instancia de `Integer`. Nótese, que estamos aproximando el tamaño de los objetos como la suma de los tamaños de todos sus atributos. En particular, asumimos que, tanto un entero (*integer*) como una referencia ocupan 4 bytes.

El análisis ha sido integrado en el sistema COSTA. En el Artículo 9 se discuten los resultados obtenidos en nuestra evaluación experimental, en la que se estudia el comportamiento del análisis con una serie de aplicaciones, escritas en un estilo fuertemente orientado a objetos, que hacen un uso intensivo del heap, y que ilustran diferentes aspectos relevantes como consumos dependientes de atributos, herencia y polimorfismo, invocaciones virtuales, etc. Estos ejemplos ilustran los aspectos más relevantes de nuestro análisis: inferencia de consumos constantes, consumos proporcionales al tamaño de la entrada, soporte para estructuras de datos como listas, árboles, arrays, etc. Por que sabemos, éste es el primer análisis de consumo capaz de inferir cotas arbitrarias para Java Bytecode.

4.2. Análisis de Consumo del Heap Activo para Lenguajes con GC

Como hemos comentado anteriormente, la recolección de basura (GC) complica mucho el problema de la predicción de memoria. Una primera aproximación es inferir el consumo de memoria *total*, es decir, la cantidad acumulada de memoria alojada por el programa, sin tener en cuenta el GC (como se hicimos en la Sección anterior). Si disponemos de dicha cantidad, está garantizado que el programa podrá ejecutarse, incluso si el GC no se aplica durante la ejecución. No obstante, el consumo inferido es una estimación demasiado pesimista del consumo de memoria real.

Esta tesis presenta un enfoque genérico para inferir el *consumo pico de memoria* durante la ejecución del programa. Nuestro análisis del consumo del heap activo se ha formulado para (una representación intermedia de)

un lenguaje bytecode con orientación a objetos y con gestión automática de memoria (es decir, GC).

El análisis de la memoria activa se diferencia del análisis del consumo total pues requiere considerar el consumo en *todos los estado del programa* durante su ejecución, a diferencia del consumo total, en el que sólo se ha de tener en cuenta el estado *final*. Como consecuencia, el enfoque clásico de análisis de coste estático propuesto por Wegbreit en 1975 [86] sólo se ha aplicado para inferir consumos totales (o acumulativos). La ventaja de este enfoque es que puede obtener información precisa sin estar restringido a ninguna clase de complejidad. Además, en principio, el enfoque es genérico en el sentido de que puede usarse para inferir diferentes nociones de recursos como consumo de memoria, número de instrucciones, número de llamadas a métodos, etc. Desafortunadamente, como discutimos en el Artículo 9, el enfoque no es válido para inferir el consumo pico pues éste no es un recurso acumulativo de la ejecución del programa. Sin embargo, requiere razonar sobre todos los posibles estados para calcular su máximo. Basándonos en distintas técnicas, que no generan *CRs*, el análisis de consumo del heap activo está actualmente limitado a cotas polinómicas y a métodos no recursivos [18] o a cotas lineales con recursión [24].

Inspirándonos en las técnicas básicas usadas en los análisis de coste, en esta tesis, presentamos un enfoque general para inferir cotas precisas en el consumo pico de los programas, mejorando el estado actual del arte al no estar el enfoque restringido a ninguna clase de complejidad, y al ser capaz de tratar la recursión. Para desarrollar nuestro análisis necesitamos primeramente caracterizar el comportamiento del recolector de basura subyacente. Asumiremos un gestor de memoria basado en *entornos* ("scopes"), que reclama memoria sólamente al finalizar los métodos. Nuestras principales contribuciones son:

1. *Análisis de la memoria escapada.* En primer lugar, desarrollamos un análisis para inferir cotas superiores de la *memoria que escapa* de los métodos, es decir, la memoria alojada durante la ejecución del método y que permanece cuando éste finaliza. La idea básica es inferir primero una cota superior del consumo total de memoria del método, como hacemos en la Sección 4.1. Después, dicha cota puede ser manipulada, utilizando la información inferida por un *análisis de escape* [17] para extraer de él una cota superior de la memoria escapada.

2. *Análisis del consumo de memoria activa.* Utilizando las cotas superiores del punto anterior, como nuestra principal contribución, proponemos una nueva forma de *ecuaciones del consumo pico*, que capturan el consumo pico sobre todos los estados del programa para el gestor de memoria considerado. Una característica fundamental de nuestras *CRs* es que aún se pueden resolver usando los resolutores existentes.

3. *Recolector de basura ideal.* Un aspecto muy interesante, y al mismo tiempo novedoso de nuestro enfoque es que podemos refinar fácilmente el análisis para acomodar otros tipos de gestores de memoria, en particular más cercanos al gestor *ideal*, el cual recolectaría los objetos tan pronto como éstos dejan de ser referenciables.

4. *Implementación.* El análisis ha sido implementado e integrado en el sistema COSTA. Hemos realizado además una evaluación experimental usando los "benchmarks" JOlden. Los resultado preliminares demuestran que el sistema obtiene cotas del consumo pico de los programas razonablemente precisas de forma totalmente automática.

Consideremos de nuevo el ejemplo de la sección previa. Nuestro análisis de consumo del heap activo infiere las siguientes *CRs* (simplificadas) para el método `copy` de la clase `Cons`:

$$C_{copy}(a) = 12, \quad a = 1$$
$$C_{copy}(a) = 8 + \max(4, C_{copy}(a-1)), \quad a > 1$$

La intuición de la segunda relación es que el consumo pico del método cuando $a > 1$ es el consumo del método (un objeto `Cons`) más el máximo entre el consumo pico del método `m` y la memoria escapada de `m` más el consumo pico de *copy* con el argumento decrementado. El *CRs* puede de nuevo resolverse utilizando el resolutor de COSTA devolviendo la siguiente cota en forma cerrada:

$$C_{copy}(a) = 8*nat(a-1) + 24$$

Una observación interesante es que el objeto de tipo *Integer* creado dentro del método *m*, no es alcanzable desde fuera y por tanto puede ser recolectado. Nuestro análisis lo tiene en cuenta, y es por ello, que ha borrado el tamaño del objeto *Integer* de la ecuación recursiva, obteniéndose 8 en lugar

de 12 multiplicando a $\texttt{nat}(\texttt{A}-\texttt{1})$. También podemos observar que COSTA no está siendo del todo preciso, pues el consumo pico real del método es $\texttt{8} * \texttt{nat}(\texttt{A} - \texttt{1}) + \texttt{8}$ (el tamaño de la lista clonada). La razón de esta imprecisión es que el resolutor de cotas superiores ha de considerar los casos adicionales introducidos por el análisis del consumo pico de memoria en las expresiones *max* para asegurar su corrección, haciendo que la segunda constante crezca hasta 24.

4.3. Trabajo Relacionado

En la literatura hay una gran cantidad de trabajos sobre el análisis de recursos y de complejidad, aunque la mayoría son sobre análisis de tiempos (ver por ejemplo [**?**]). El análisis del consumo del heap activo es diferente pues requiere que se consideren todos los estados del programa. La mayoría de los trabajos sobre estimación de memoria, se han realizado en el contexto de lenguajes funcionales. El trabajo en [50] infiere estáticamente, por medio de derivaciones de tipos y programación lineal, expresiones lineales que dependen de los parámetros funcionales. Nótese que con nuestro enfoque se pueden calcular cotas no lineales (polinómicas, logarítmicas, exponenciales, etc). Las técnicas propuestas en [83, 82] consisten en, dada una función, construir una nueva función que representa simbólicamente el coste de la primera. Aunque estas funciones recuerdan a nuestras relaciones de coste, éstas deben ejecutarse sobre unos valores concretos de los parámetros para obtener una cota de memoria para una asignación concreta. A diferencia de nuestras cotas en forma cerrada, la evaluación de su función podría no terminar, incluso aunque el programa original si lo hiciese.

Merece la pena mencionar el trabajo en [21], donde se presenta una análisis del consumo de memoria. A diferencia de nuestro enfoque, su propósito es verificar que el consumo de memoria del programa está acotado. Los autores consiguen esto simplemente comprobando que no se creen objetos dentro de bucles, pero en ningún momento infieren cotas simbólicas como hacemos en nuestro análisis. Nótese que realizar este tipo de comprobación resultaría directo usando nuestras ecuaciones de coste. Otro trabajo relacionado es el realizado en el proyecto MRG ("Mobile Resource Guarantees") [10, 16], en el cual se centran en construir una arquitectura de "Proof-Carrying-Code"[74] en la que se asegura que los programas no

violan las restricciones de consumo de recursos impuestas. El análisis se desarrolla para un lenguaje funcional que se compila posteriormente a un subconjunto de Java Bytecode, y está restringido a cotas lineales.

Para lenguajes del estilo de Java, podemos mencionar el trabajo de [52], donde se presenta un sistema de tipos para realizar análisis de heap sin recolección de basura. El análisis está desarrollado a nivel del código fuente y está basado en técnicas de *análisis amortizado*. Es por tanto técnicamente diferente al nuestro, y no llega a proponer un método de inferencia de consumo del heap.

Se han propuesto también recientemente técnicas que tratan de mejorar nuestro primer enfoque, presentado en el Artículo 9. En particular, [24] considera un lenguaje ensamblador, e infiere cotas del consumo de memoria (tanto de la pila como del heap). El enfoque está, no obstante, restringido a cotas lineales, y se basa en la existencia de comandos explícitos de liberación de memoria en lugar de en un gestor automático de memoria. En su sistema, estos comandos de liberación de memoria pueden generarse automáticamente a partir de unas anotaciones de usuario. En [18] se considera un lenguaje del estilo de Java y se infieren cotas del consumo pico basándose en un gestor automático de memoria como hacemos nosotros. Sin embargo no tratan con métodos recursivos y su enfoque está restringido a cotas polinomiales. Además, nuestro enfoque (Artículo 10) es más flexible en cuanto a su posible adaptación a distintos esquemas de recolección de basura. Pensamos que nuestro sistema es el primero capaz de inferir cotas del consumo pico de los programas no restringidas a ninguna clase de complejidad.

Capítulo 5

Conclusiones y Trabajo Futuro

El principal objetivo de esta tesis ha sido mejorar el estado del arte en la transformación y análisis de lenguajes bytecode. Nuestro primer reto fue proponer un esquema formal para la decompilación automática de programas bytecode (con orientación a objetos) a representaciones intermedias de alto nivel usando LP, por medio de decompilación interpretativa. Este enfoque ofrece una serie de ventajas comparado con el desarrollo de decompiladores dedicados como *flexibilidad*, *mantenibilidad*, *seguridad* y *generalidad*. Aunque es muy atractivo, hasta ahora no se había usado realmente en la práctica exceptuando algunas *pruebas de concepto* en las que simplemente se demuestra su viabilidad [63, 48, 76, 64].. Quedaban por tanto una serie de cuestiones abiertas a la hora de tratar de aplicar el enfoque interpretativo en lenguajes y aplicaciones reales, en particular su *escalabilidad* y *efectividad*. Esta tesis propone soluciones novedosas y finalmente responde afirmativamente a estas cuestiones presentando un esquema de decompilación modular y óptimo que: (1) produce programas decompilados cuya calidad es equivalente a la obtenido utilizando decompiladores dedicados, y (2) demuestra teórica y empíricamente escalar en la práctica. Los resultados experimentales obtenidos muestran que nuestro decompilador es competitivo en la práctica, desde el punto de vista de la eficiencia, con decompiladores dedicados. Creemos por tanto que, las técnicas propuestas, junto con su evaluación experimental, proporcionan por primera vez una prueba tangible de que la teoría interpretativa propuesta por Futamura en los años 70, es de hecho una alternativa viable y realista al desarrollo de decompiladores dedicados de lenguajes modernos a representaciones

intermedias.

Para concretar, nuestro esquema de decompilación interpretativa ha sido formalizado en el contexto de la EP de programas lógicos, e implementado para el lenguaje Java Bytecode. Es sin embargo importante notar que las ideas propuestas para posibilitar la puesta en práctica del enfoque, son por supuesto de interés para la decompilación interpretativa de cualquier par de lenguajes *origen* y *destino*.

Por otro lado, el estudio de una aplicación tan compleja de la EP, nos ha llevado a resolver varios problemas no triviales de ésta en general, como por ejemplo el tratamiento de signaturas infinitas. A este respecto, se ha presentado la relación de la *subsunción homeomórfica basada en tipos*, la cual ha demostrado mejorar el estado del arte en las herramientas de especialización online. Hemos visto también como distintos enfoques existentes que extendían la relación original no tipada para tratar con signaturas infinitas, se pueden reconstruir como instancias de nuestra relación TbHEm. Aunque se han esbozado procedimientos para inferir los tipos en el contexto de la LP, nuestra relación basada en tipos no está atada a ningún paradigma de programación. Más aún, se podría usar en un rango amplio de aplicaciones como en distintas áreas de análisis, síntesis, verificación, especialización y transformación automática de programas, y además se beneficiaría directamente de cualquier progreso en inferencia automática de tipos.

Hemos visto como la representación intermedia resultante utilizando LP, puede simplificar en gran medida el desarrollo de herramientas de análisis, verificación y chequeo de modelos para lenguajes modernos, y, en particular, como pueden ser directamente aplicadas herramientas existentes desarrolladas para LP (probadas correctas y efectivas). Hemos realizado dos estudios experimentales en esta dirección. En el primero de ellos, hemos investigado si es viable analizar programas bytecode a base de analizar sus decompilaciones a LP utilizando herramientas de análisis LP existentes. En este sentido, hemos sido capaces de demostrar automáticamente, usando el sistema `CiaoPP`, algunas propiedades no triviales de programas Java Bytecode (de tamaño pequeño) como, ausencia de errores en tiempo de ejecución, terminación e inferencia de cotas en el uso de recursos.

En nuestro segundo estudio experimental, hemos aprovechado el hecho de que nuestros programas decompilados son totalmente *ejecutables*, pues

a diferencia de otros enfoques [70, 2, 84], representan el estado completo de ejecución (es decir, contienen una representación explícita del heap además de la pila de operandos). En este sentido, hemos propuesto un esquema de generación automática de datos de prueba, basado en ejecución simbólica, para lenguajes bytecode por medio de técnicas de EP en CLP. Nuestro enfoque consiste en dos fases diferenciadas: (1) la (de)compilación del bytecode a un programa CLP, y (2) la generación de datos de prueba a partir del programa CLP. una pregunta que surge inevitablemente es si este enfoque puede utilizarse para otros lenguajes imperativos que no sean necesariamente bytecode. Los enfoques basados en ejecución simbólica existentes para Java [73], y para C [45], presentan problemas a la hora de tratar con aspectos como la recursión, las llamadas a métodos, la memoria dinámica, etc. Hemos visto como estos aspectos se tratan de forma uniforme en nuestro enfoque gracias a la transformación a CLP. En particular, todas las clases de bucles en el bytecode se representan de forma uniforme como predicados recursivos en el programa CLP. Hemos visto también como las llamadas a métodos se tratan de la misma manera que las llamadas a bloques, y por tanto no presentan ninguna dificultad adicional.

Pensamos que el estudio experimental realizado es una prueba de concepto muy prometedora, que muestra que la EP en el contexto de CLP es una técnica muy potente, en particular, para realizar TDG de lenguajes bytecode e imperativos en general. Para poner en práctica nuestras ideas hemos considerado un sencillo lenguaje bytecode imperativo dejando aspectos como la orientación a objetos para trabajo futuro. Hemos restringido también el lenguaje a números enteros quedando también como trabajo futuro la extensión para tratar con diferentes tipos de datos. A corto plazo, planeamos realizar una evaluación experimental con programas Java Bytecode de "benchmarks" existentes. Al considerar aspectos de lenguajes bytecode más realistas como el uso de números reales, llamadas virtuales, etc, seguro que tendremos que tratar como numerosas dificultades. Uno de los puntos prácticos fundamentales es la escalabilidad del enfoque, que suele venir ligada al problema de la *no viabilidad de caminos* [90]. Éste suele ocurrir sobretodo en enfoques que no integran la fase de resolución de restricciones con la de generación de caminos, sino que realizan estas fases de forma independiente. En este sentido, no esperamos tener problemas pues nuestro enfoque integra ambas fases y por tanto detecta y descar-

ta los caminos inviables tan rápido como éstos aparecen. Otro problema interesante es el obtener una representación manejable del heap, lo cual será necesario para poder obtener casos de prueba en programas que manipulen objetos y arrays. Para la evaluación experimental también planeamos extender nuestra técnica para incluir criterios de recubrimiento más avanzados. En particular sería interesante considerar otras clases de criterios que, por ejemplo, nos permitan obtener casos de prueba que recubran una determinada instrucción del programa.

Como hemos visto, en principio nuestro enfoque de TDG puede aplicarse a cualquier lenguaje, tanto de alto como de bajo nivel. En este sentido, esta tesis también ha presentado un estudio experimental en el que tratamos de aplicar la segunda fase para generar casos de prueba de programas CLP, no necesariamente obtenidos por decompilación de programas bytecode o imperativos. Esto introduce algunas dificultades como el tratamiento de las derivaciones de fallo y de los datos simbólicos. Esta tesis ha esbozado las soluciones para superar dichas dificultades. En particular, hemos propuesto una transformación de programa, basada en EP, que hace explícito el fallo en los programas lógicos. Para tratar la negación en los programas transformados, hemos esbozado soluciones basadas en técnicas existentes, que hacen posible transformar la información negativa en positiva. Aunque nuestros primeros experimentos en este sentido ya sugieren que el enfoque puede ser muy útil, por ejemplo para generar casos de prueba para programas Prolog, pensamos realizar aún una evaluación experimental en profundidad comparando con técnicas existentes. Esto requeriría cubrir ciertos aspectos del lenguaje Prolog que aún no hemos considerado, como el corte, el sistema de módulos, etc. Queremos también estudiar la integración de otras clases de criterios de recubrimiento como los basados en el *flujo de datos*. Finalmente, nos gustaría estudiar la integración los análisis estáticos en el contexto de TDG. Por ejemplo, utilizar la información inferida por un *análisis de fallo* podría ser muy útil para poder así podar algunas de las ramas que nuestros programas transformados en principio han de considerar.

Otro de los grandes retos de esta tesis ha sido mejorar el estado del arte en el análisis de consumo del heap de lenguajes bytecode. En este sentido, hemos desarrollado una aplicación novedosa del esquema de análisis de coste de [3] para analizar consumo del heap, el cual ha sido también extendido

para considerar el efecto de la recolección de basura. Hemos presentado por tanto un enfoque genérico para el análisis automático y preciso de consumo del heap activo para lenguajes bytecode con recolección automática de basura. Para ello, primero hemos propuesto como obtener cotas precisas de la memoria que escapa de los métodos, combinando el consumo total inferido por el propio método, junto con la información obtenida por un análisis de escape. Después, hemos introducido una forma novedosa de *relaciones del consumo de memoria pico*, que utilizando las cotas de la memoria escapada, capturan el consumo pico de los programas considerando el efecto de la recolección de basura. Estas relaciones de coste se pueden convertir a forma cerrada utilizando resolutores existentes, en particular el del sistema COSTA. Para concretar, nuestro análisis ha sido desarrollado para un lenguaje bytecode con orientación a objetos, aunque pensamos que las mismas técnicas podrían aplicarse a otros lenguajes con recolección de basura. En primer lugar, el análisis considera un gestor de memoria basado en *entornos*, que sólamente reclama la memoria en la finalización de los métodos. La cantidad de memoria requerida para poder ejecutar un método bajo este modelo, puede usarse como una sobreaproximación de la cantidad requerida en el contexto de una recolector de basura ideal (que reclama la memoria de los objetos tan pronto como estos dejan de ser alcanzables). Hemos mostrado también como aproximar el comportamiento de dicho recolector ideal en nuestro análisis.

Finalmente, es importante notar que este enfoque podría también utilizarse para estimar otros recursos no acumulativos, que requieren maximizar el consumo de varios caminos de ejecución diferentes. Por ejemplo, pensamos que podría utilizarse para estimar la profundidad máxima de la pila de llamadas de la siguiente manera:

Dada una regla $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) ::= \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$, donde $b_{i_1} \ldots b_{i_k}$ son las llamadas en r, con $1 \leq i_1 \leq \cdots \leq i_k \leq n$ y $\mathrm{b}_{\mathrm{i_j}} = \mathrm{q}_{\mathrm{i_j}}(\langle \bar{\mathrm{x}}_{\mathrm{i_j}} \rangle, \langle \bar{\mathrm{y}}_{\mathrm{i_j}} \rangle)$, su ecuación correspondiente sería

$$p(\bar{x}) = \text{máx}(1 + q_{i_1}(\bar{x}_{i_j}), \ldots, 1 + q_{i_k}(\bar{x}_{i_k})) \quad \varphi_r$$

que considera la profundidad máxima de todas las cadenas de llamadas. Cada "1" corresponde a cada "frame" creado para la llamada correspondiente. Ésta es por tanto otra línea de posible trabajo futuro.

Parte II

Versión en Inglés (English Version)

Chapter 6

Introduction: Motivation and Contributions

6.1. Bytecode Languages

Programming languages can in general be categorized by the underlying execution model that runs them. Thus, they typically fall into one of two categories: *compiled* or *interpreted*. In compiled languages the *source code* is first translated, or compiled, into a set of hardware-specific instructions, often called *object code*. The program is then run by executing the object code in the corresponding hardware. In contrast, in interpreted languages the source code is executed by an interpreter. The distinction applied to programming languages is somewhat vague as in theory any language may be compiled or interpreted. The categorization usually reflects the most popular or widespread implementations of languages and not their underlying properties. Each of the alternatives has their own advantages and drawbacks. For instance, the execution of an object program in the corresponding machine tends to be much faster than executing the same source program using an interpreter, even a 10:1 ratio is not uncommon. On the other hand, interpreted languages provide certain extra flexibility over compiled programs, namely, ease of implementation, ease of debugging, and the most important, platform-independence.

A combination of both approaches, known as *bytecode-compilation* or *bytecode-interpretation*, is becoming widely used. In a bytecode-compilation-based execution model, source code is translated to some inter-

```
void foo(int n,int m){
    this.f = n + m;
}
```

```
void foo(int,int)
  0: aload_0        3: iadd
  1: iload_1        4: putfield f
  2: iload_2        7: return
```

Figura 6.1: Simple Java Bytecode program

mediate form, known as *bytecode*. The bytecode is not the machine code for any particular computer, and may be portable among computer architectures. The bytecode may be then interpreted by, or run on, a virtual machine. The name bytecode stems from instruction sets which have one-byte *opcodes* followed by optional parameters. Bytecode instructions are often akin to traditional hardware instructions. For instance, bytecode languages often have an unstructured control flow with several sources of branching (e.g., conditional and unconditional jumps) and use an operand stack to perform intermediate computations. Nevertheless, since bytecode instructions are thought to be processed by software, they may be arbitrarily complex, especially in the case of object-oriented and declarative bytecode languages. To get the picture, Figure 6.1 shows the source code and (Java) bytecode of a method that takes two integer numbers, adds them, and assigns the result to the `f` field of the *this* object. Note that, in Java Bytecode, the *this* object is explicitly passed in local variable 0, thus, the `aload_0` instruction, pushes the *this* reference at the top of the stack.

As regards efficiency, the bytecode-compilation model is typically somewhere in between the pure compilation-based and interpretation-based models; while it keeps the advantages of interpretation-based models, in particular, platform independence. Furthermore, there is nothing about a bytecode language that requires it to be exclusively interpreted. Therefore, *just-in-time* compilation (JIT) can be used to speed up the execution of bytecode. A JIT compiler performs the conversion to native machine code gradually during the program's execution thus obtaining a better performance. Bytecode-compilation together with JIT compilation can therefore combine most advantages of interpretation-based and compilation-based execution models. This is the main reason of success of the runtime envi-

ronments of *Microsoft .NET* and the *Java* frameworks, which are probably the most widely used programming environments nowadays.

Java Bytecode Java Bytecode is the language designed to be executed by the *Java Virtual Machine* (JVM) [66]. It was originally designed by *Sun Microsystems* to be an intermediate language in the Java runtime environment. A Java Bytecode instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon. The JVM uses a set of structures that the bytecode instructions manipulate. The main ones are: the *program counter* which contains the index of the current instruction, the *operand stack* and *local variables array* in which parameters, variables and intermediate results are stored, the *heap* from which memory for all class instances and arrays is allocated, and the usual *call stack* or *frame stack* to handle method calls and returns. Java Bytecode comprises, on one hand, the classical low-level instructions to transfer values from the operand stack to the local variables and viceversa, to perform arithmetic operations (most of them operate directly on the operand stack), to jump to other part of the code (conditionally or unconditionally), to call (and return from) methods, etc. For instance, the instruction `iload_1` loads the local variable 1 on the top of the operand stack and instruction `iadd` adds (and pops) the top two values of the stack and pushes the result on it. On the other hand, Java Bytecode has object-oriented and concurrency features. Thus it also includes instructions to create objects and arrays, to get and set object fields and array elements, to perform virtual invocations, to enter and exit monitors, etc. E.g., instruction `putfield f` sets with the value on top of the stack, the `f` field of the object referenced by the memory address stored inmediately under the top of the stack.

Although Java is the most common language targeting Java Bytecode, there are nowadays many compilers from different high-level languages to Java Bytecode. Some of the most well known are: *jython* for *Python* programs, *jRuby* for *Ruby* and *jGNAT* for *Ada*.

.NET Common Intermediate Language The *Common Intermediate Language* (CIL) is the bytecode intermediate language used by the *.NET*

programming framework. Thus, source languages targeting the .NET framework compile to CIL. As Java Bytecode, CIL is an object-oriented and stack-based bytecode language. It therefore includes the same kind of bytecode instructions. Unlike Java Bytecode, CIL is not meant to be interpreted. From the beginning, it was rather thought to be compiled into machine code using JIT compilation. The bytecode is even sometimes entirely converted into machine code before runtime using a native image generator to further improve performance. The .NET framework is a key Microsoft offering and is intended to be used by most new applications created for the Windows platform.

There are other well known bytecode languages both imperative, like the *p-code* used in some Pascal implementations, and declarative, like the *WAM* bytecode, used in most Prolog implementations, the *Haskell Hugs'98* bytecode and the *Erlang BEAM* bytecode, to name some.

This thesis is mainly focused on object-oriented imperative bytecode languages. In particular, as we will see, the technical parts of this thesis, as well as the different prototype implementations we have developed, consider representative subsets of Java Bytecode.

6.2. Static Program Analysis

Predicting the behavior of programs before their actual execution becomes more and more relevant as programs increase in complexity and get used in critical situations such as medical operations, flight control or banking cards. Being able to prove, in an automatic way, that programs do adhere to their functional specifications is a basic factor to their success. *Static program analysis* is the process of automatically analyzing the behavior of programs without actually executing the code. In contrast, analysis performed by executing programs is known as *dynamic analysis*. Classical static analyses aim at inferring properties of programs like: *error-freeness*, *termination*, cost or resource consumption (time or memory), *liveness* of variables, *pointer shape*, etc. The usual way to perform static analysis is to use formal methods. Some of the most common are: *abstract interpretation*, *model checking* and *type systems*. This thesis mainly focuses on static analysis based on abstract interpretation.

Abstract Interpretation. The technique of abstract interpretation [30] provides a general formal framework for computing safe approximations (i.e., abstractions) of programs behavior. Its main practical application is formal static analysis. Analyzers based on abstract interpretation infer information on programs by interpreting ("running") them using abstract values rather than concrete ones. These analyzers are parametric w.r.t. the so-called abstract domain, which provides a finite representation of possibly infinite sets of values. Different domains capture different properties of the program with different levels of precision and at a different computational cost.

Abstract interpretation-based static analysis has been studied in the context of declarative languages and also for high-level imperative languages. In what follows we enumerate some analysis systems:

The ASTRÉE analyzer. *ASTRÉE* [31] is a static program analyzer developed at the *École Normale Supérieure* by Cousot *et. al.* aiming at proving the absence of run-time errors of C programs. ASTRÉE was able for example to prove fully automatically the absence of any run-time error in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132.000 lines.

The CiaoPP system. CiaoPP [49] is the abstract interpretation-based preprocessor of the Ciao-Prolog *Contraint Logic Programming* (CLP) system [25]. It can perform a number of program debugging, analysis and source-to-source transformation tasks on Ciao-Prolog programs. Some of the properties it is able to infer are: *types*, *modes* and other variable instantiation properties, *non-failure*, *determinacy*, bounds on computational cost, bounds on sizes of terms in the program, etc. CiaoPP is also able to perform several kinds of source to source program transformations such as program specialization, program parallelization (including granularity control), etc.

Some other well-known (non-commercial and commercial) static analysis systems are: *Lint*, *CCA* and *BOON* for C programs, *CodeSonar* for C++, *Fluid* and *jLint* for Java, and many others. Other static analyzers

have not become self-contained tools but are rather integrated in most compilers. An example of this is the JVM verifier which integrates a data-flow analyzer.

Traditionally, most analyses have been formulated at the source code level. However, it can be the case that the analysis must consider the compiled code, or bytecode, instead. This may happen, in particular, when the code consumer is interested in verifying some properties of 3rd party programs, but has no direct access to the source code, as usual for commercial software and in mobile code. This is the general picture where the idea of *Proof-Carrying code* [74] was born: in order for the code to be verifiable by the user, security properties (possibly inferred by static analysis) must refer to the compiled code (or bytecode) available to the user, so that it is possible to check the provided proof and verify that the program satisfies the requirements (e.g., that the code does not require more than a certain amount of memory, or that it executes in less than a certain amount of time).

Hence, there is a need to develop analysis and verification tools which work directly on bytecode programs. Unfortunately, reasoning about realistic (object-oriented) bytecode programs is rather complicated and time consuming. In addition to the object-oriented features such as inheritance and virtual method invocations, a bytecode analyzer has to deal with several low-level language features like the unstructured control flow, the usage of the operand stack, etc.

6.3. From Bytecode to Intermediate Representations

In the context of analysis of bytecode languages, a usual practice is to approach the problem into two steps: (1) the bytecode program is transformed into a higher-level *intermediate representation* (IR), and (2) the analysis is developed over such IR. This allows abstracting away the particular bytecode language features and developing the analysis tools on much simpler representations. As another advantage, this approach also enables the possibility of reusing the analysis step (step (2) below) for analyzing different bytecode (and not bytecode) languages, provided they are trans-

formed into the same IR. In the rest of the thesis, we will use the term *decompilation* to refer to the transformation of bytecode to an IR, as it translates a low-level language to a higher-level one.

Most of the approaches develop *ad-hoc*, or dedicated, decompilers, i.e., decompilers exclusively designed to carry out a particular decompilation. There is however an alternative to the development of dedicated decompilers which is the so called *interpretive decompilation* by *partial evaluation*. As we will see, it allows decompiling programs by partially evaluating an interpreter w.r.t. them.

Partial Evaluation. *Partial evaluation* (PE for short) [56] is a semantics-based, source to source, program transformation technique, which allows specializing programs w.r.t. part of their input data. Hence it is often called *program specialization*. Consider a program P, and its input which is split in I_{static} and $I_{dynamic}$. I_{static} is the static data, i.e., the input which is known at compile time, and $I_{dynamic}$ is the rest of the input. We can see the program P as a mapping of its input into its output as follows:

$$P : I_{static} \times I_{dynamic} \longrightarrow O$$

The partial evaluator transforms the pair $\langle P, I_{static} \rangle$ into $P' : I_{dynamic} \longrightarrow O$ by performing the computations of P that depend on I_{static} at compile time. P' is called the *residual program* and should run more efficiently than the original program P.

The Interpretive Approach to Compilation. A particularly interesting application of PE, first described in the 1970s by Yoshihiko Futamura [41], is when the program P to partially evaluated, is an interpreter for a programming language. This is called the *interpretive approach to compilation* or the *first Futamura projection*. Let us assume an interpreter, written in a target language L_T, for programs written in a source language L_S. Then, if I_{static} is a source program, written in L_S, the PE of the interpreter w.r.t. this data/program produces P', a version of the interpreter that only runs that source code, which is written in the implementation language of the interpreter, L_T, and which does not require the source code to be resupplied. P' can be considered as a

compiled version of I_{static} into the target language L_T. The interpretive compilation thus allows compiling programs written in L_S into another language L_T by partially evaluating an interpreter for L_S written in L_T w.r.t. them.

In the particular case of bytecode decompilation, the interpretive approach to compilation allows us to decompile a bytecode program written in some bytecode language BC, into a higher-level representation, written in a high-level language HL, by partially evaluating an interpreter of BC written in HL. This is in principle more generic and flexible, safer and easier to maintain than the development of a dedicated decompiler for the same task. These advantages will be discussed later on in Section 7. The interpretive approach, though very attractive in principle, has not been widely applied in practice mainly because of the difficulty in finding partial evaluation strategies which produce *effective*, i.e., *quality*, and *efficient* decompilations.

6.4. Heap Space Analysis for Bytecode

Research about the resource usage of programs goes back to the seminal work by Wegbreit in 1975 [86], which proposes to analyze the performance of a program by deriving a mathematical expression which represents its runtime behavior. The standard approach to perform static cost analysis is as follows: given an input program, (1) in a first step, the cost analysis generates an associated cost equation system (CES) from the program, which captures the relation between the different parts of the code. CESs are sets of recurrence equations which express the cost of a program in terms of the size of its input arguments. (2) In the second step, CESs can be often solved (or approximated) by typically relying on algebraic techniques, thus obtaining a closed form (e.g., without recurrences) solution or an upper (or lower) bound for it.

Cost analysis has been intensively studied in the context of declarative (see, e.g., [79, 80, 44, 15] for functional programming and [34, 35] for logic programming) and high-level imperative programming languages (mainly focused on the estimation of worst-case execution times and the design of cost models [88]). Traditionally, as most static analyses, cost analysis has

been formulated at the source level. However, as we have seen, there are situations where we do not have access to the source code, but only to the compiled code. Recently, [2] has proposed a cost analysis framework for Java Bytecode which is the formal base of the COSTA system [5].

The COSTA **system.** COSTA [5] is a research prototype which performs automatic **COS**t and **T**ermination **A**nalysis for Java Bytecode programs. The system receives as input a bytecode program and a cost model chosen from a selection of resource descriptions, and tries to bound the resource consumption of the program with respect to the given cost model. COSTA follows the standard approach to perform cost analysis, i.e., it first produces a CES, which is an extended form of recurrence relation. Then, in order to obtain a closed form for such recurrence relations, which represents an upper bound, COSTA includes a dedicated solver [9].

An interesting application of cost analysis which poses new challenges is heap space analysis. It aims at inferring *bounds* on the heap space consumption of programs. Again, heap analysis is more typically formulated at the source level (see, e.g., [83, 50, 85, 54] in the context of functional programming and [52, 23] for high-level imperative programming languages). In the context of bytecode languages, heap space analysis has interesting applications. For instance, *resource bound certification* [33, 8, 10, 51, 22] proposes the use of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on the resource consumption. Also, heap bounds are useful on embedded systems, e.g., smart cards in which memory is limited and cannot easily be recovered.

Unfortunately, automatic memory management, also known as *garbage collection* (GC), which is increasingly used in bytecode languages like Java Bytecode and the .NET CIL, makes the problem of predicting the memory required to run a program very difficult. A first approximation to this problem is to infer the *total memory allocation*, i.e., the *accumulated* amount of memory allocated by a program ignoring GC. If such amount is available it is ensured that the program can be executed without exhausting the memory, even if no GC is performed during its execution. However, this is an overly pessimistic estimation of the actual memory requirement. Recently,

[83, 18, 24] have proposed *live heap space analysis*, which aims at approximating the size of the *live* data on the heap during a program's execution, thus providing a much tighter estimation. These approaches are however currently restricted to polynomial bounds and non-recursive methods [18] or to linear bounds dealing with recursion [24].

6.5. Main Goals and Contributions

The main objective of this thesis is to improve the state-of-the-art in the transformation and analysis of bytecode languages by: (1) providing and implementing a formal framework for the automatic decompilation of (object-oriented) bytecode programs to higher-level intermediate representations, in particular represented using logic programming (LP), by means of the interpretive approach to compilation; (2) study the practical applications that having such IRs based on LP can have; and (3) designing and implementing a live memory consumption analysis for bytecode languages with *garbage collection*. In particular, the main contributions of this thesis are the following:

1. **Interpretive decompilation of bytecode to LP:** There have been several proofs-of-concept showing that the interpretive approach is feasible [63, 48, 76, 64]. However, there remain important open issues when it comes to decompile realistic languages. These include scalability, which in turn depends on compositionality, and effectiveness, i.e., quality of the obtained programs. This thesis presents, to the best of our knowledge, the first scheme to enable interpretive decompilation of a realistic bytecode language to a high-level representation, namely, we decompile Java Bytecode to Prolog.

   a) **Control strategies:** One of the main difficulties of interpretive decompilation and of EP in general, is to properly handle infinite signatures. We have proposed novel techniques which enable the definition of sophisticated control rules. In particular, we have introduced the *Type-based homeomorphic embedding* relation, a generalization of the original *Homeomorphic embedding* relation which provides more precise results in the presence

of infinite signatures. We have shown that this technique, besides being crucial in the specialization of interpreters, improves state-of-the-art (online) specialization tools. This work was first proposed in Paper 3 (see Appendix A) and later extended in Paper 4, which was published in the *Information Processing Letters* journal.

b) **Controlling the polyvariance of EP:** Even after enhancing a partial evaluator with the *Type-based homeomorphic embedding*, the decompiled programs we obtain tend to have too many (redundant) specialized versions of some predicates. This issue is studied in detail in Paper 2, where we propose advanced techniques to control the *polyvariance* of the PE process, i.e., which avoid having such redundant specialized versions.

c) **How to write the bytecode interpreter:** As shown in previous work on interpretive compilation, the characteristics of the interpreter can be crucial for obtaining a successful specialization. We have identified the necessary features in order to obtain a compositional decompilation scheme.

d) **Optimal decompilation:** We ensure the quality of the decompilations, both in terms of effectiveness and efficiency, by providing different *optimality criteria*. They basically require that (1) the decompilation does not generate code more than once for each program point, and (2) there is at most one residual rule for each block in the bytecode. We propose a decompilation scheme which is *optimal* w.r.t. these optimality criteria. This ensures scalability of the process and quality decompilations. This work, together with that described in issue (c), led to Paper 5.

e) **Dealing with object-oriented features:** We show how our scheme can be easily adapted to handle object-oriented features. Namely, we provide the mechanisms to: handle the heap and its associated instructions, represent classes by means of Prolog modules, and represent virtual invocations by means of module-qualified calls.

f) **Implementation and experimental evaluation:** All the techniques above have been implemented and integrated in a pro-

totype decompiler of full sequential Java Bytecode to Prolog, called jbc2prolog. Experimental results are reported using our prototype (and other systems). In particular, both the scalability and efficiency of our approach are assessed using the JOlden suite of benchmarks [55]. The work described in issues (b), (c), (d), (e) and (f), led to Paper 6 which has been recently published by the *Journal of Information and Software Technology*. This paper thus achieves the objective (1) above.

2. **Applications of interpretive decompilation:** Using a declarative language for defining our IR offers important advantages. In particular, existing advanced analysis and transformation tools for declarative languages (already proven correct and effective) could be then re-used for the analysis and transformation of bytecode programs

   a) **Re-using LP analysis tools:** Using the `CiaoPP` system with our decompiled programs we have been able to prove some non-trivial properties of Java Bytecode programs such as *termination* and run-time *error-freeness*, as well as, for some simple programs, to infer bounds on their resource consumption. This work is presented in Paper 1.

   b) **Test data generation:** A standard approach to the automatic generation of test data is to perform *symbolic* execution of the program [29, 57], where the contents of variables are expressions rather than concrete values. The fact that our decompiled programs are executable Prolog programs allows us to directly rely on available techniques for CLP (where backtracking is inherent to the language) to carry out such symbolic execution. We have therefore developed a novel framework for *test-case generation* of bytecode by relying on our decompiled (C)LP programs. Interestingly, we show that the generation of test-cases in CLP, can be seen as another PE, which allows us obtaining not only test-cases but *test-case generators*. This work led to Paper 7. As a tangential contribution, we have applied this idea to automatically generate test-cases for Prolog. A preliminary study in this direction is found in Paper 8.

3. **Heap and Live heap space analysis:**

   a) **Total heap space analysis:** We have first developed a novel application of the cost analysis framework of [3] to infer upper bounds on the heap space consumption of sequential Java Bytecode programs. To do that, we have just provided a cost model that defines the cost of memory allocation instructions in terms of the number of heap (memory) units they consume. We can then generate *heap space cost relations* which are directly used to infer upper bounds on the heap space usage of Java Bytecode programs.

   b) **Live heap space analysis for languages with garbage collection:** In presence of garbage collection, the proposed approach is a too pessimistic estimation of the actual memory requirement. This thesis presents a general approach for inferring the *peak heap consumption* of a bytecode program's execution, i.e., the maximum of the live heap usage along its execution which, unlike previous works, is not restricted to any complexity class.

   c) **Implementation and experimental evaluation:** The analyses have been implemented and integrated in the COSTA system. We experimentally evaluate them with a series of example applications which make intensive use of the heap, including the JOlden benchmark suite [55]. Preliminary results demonstrate that our system obtains reasonably accurate live heap space upper bounds in a fully automatic way. All this work on heap space analysis led to papers 9 and 10, thus achieving objective (3) above.

6.6. Organization of this Thesis

This thesis is a "thesis by articles" and therefore it consists of an introduction describing its main objectives, contributions and conclusions, which is presented in chapters 6, 7, 8, 9 and 10, and, the set of papers which led to the thesis presented as they appear on the corresponding formal proceedings as an appendix.

The rest of the thesis is thus organized as follows: Chapter 7 overviews the work covering the contribution (1) above. In particular, Section 7.1 provides some background on PE of logic programs, then the challenges that specializing a bytecode interpreter are presented in Section 7.2, Section 7.3 introduces the *Type-based homeomorphic embedding* relation, Sections 7.4 and 7.5 summarize the technical details of our modular and optimal decompilation schemes, Section 7.6 summarizes the implementation and experimental evaluation, and finally Section 7.7 overviews related work on (interpretive) decompilation.

Chapter 8 overviews our work on the applications of using our interpretive decompilation technique to analyze bytecode programs (Section 8.1), and to perform test data generation (Section 8.2). Then, Chapter 9 introduces our work on heap space analysis (Section 9) and its extension to consider the effect of garbage collection (Section 9.2), and discusses related work (Section 9.3). Finally, Chapter 10 presents the conclusions of the thesis and discusses ongoing and future work.

The technical details are presented in the papers which led to this thesis, which can be found in Appendix A.

Chapter 7

Interpretive Decompilation of Bytecode to LP

Decompiling bytecode languages to an intermediate representation has become a usual practice nowadays within the development of analyzers, verifiers, model checkers, etc. For instance, in the context of *mobile* code, as the source code is not available, decompilation facilitates the reuse of existing analysis and model checking tools. In general, high-level intermediate representations allow abstracting away the particular language features and developing the tools on simpler representations. In particular, Java Bytecode is decompiled to a rule-based representation in [2], to clause-based programs in [70], to a three-address code representation in Soot [84] and to the typed procedural language BoogiePL in [37]. Also, analysis of Java programs is formalized and performed using Datalog in [87] and in [48] PIC assembly is transformed into logic programs. This shows that the rule-based representations used in declarative programming in general—and in LP in particular—provide a convenient formalism to define such intermediate representations. For instance, as it can be seen in [2, 70, 48], the operand stack used in a bytecode language can be represented by means of explicit logic variables and its unstructured control flow can be transformed into recursion.

The resulting intermediate representation greatly simplifies the development of the above tools for modern languages and, interestingly, existing advanced tools developed for declarative programs (already proven correct and effective) can be directly applied on it.

All above cited approaches (except [48]) develop *ad-hoc*, or dedicated, decompilers to carry out the particular decompilations. As we pointed out in Section 6.3, an appealing alternative to the development of dedicated decompilers is interpretive decompilation by partial evaluation. The advantages of interpretive (de)compilation w.r.t. dedicated (de)compilers are well-known and discussed in the PE literature. Very briefly, they include:

1. *Flexibility*: it is easier to modify the interpreter in order to tune the decompilation (e.g., observe new properties of interest). As an interesting example, in Paper 1, a Java Bytecode interpreter is instrumented with an additional argument which computes the *trace* of bytecode instructions in order to collect the computation history. A program decompiled by using this interpreter contains an additional argument with the execution trace at the level of Java Bytecode. This trace will allow observing a good number of interesting properties about the program, e.g., runtime *error-freeness* can be ensured when the trace does not contain instructions which issue any kind of run-time error.

2. *Easier to trust*: it is more difficult to prove (or trust) that ad-hoc decompilers preserve the program semantics. For example, the formal specification chosen for defining our bytecode interpreter is Bicolano [78], which is written with the Coq Proof Assistant [11]. This allows checking that the specification is consistent and also proving properties on the behavior of some programs.

3. *Easier to maintain*: new changes in the language semantics can be easily reflected in the interpreter. This will become apparent later when we see that defining a bytecode interpreter in Prolog is a rather easy task and, hence, also maintaining it.

The challenge now is in defining a practical, scalable scheme to interpretive decompilation which achieves quality decompiled programs and, provided this is feasible, we will be able to take advantage of the above features.

There have been several proofs-of-concept of interpretive (de)compilation (e.g., [48, 63]), but there remain interesting open issues when it comes to assess its power and/or limitations to decompile a real language. Such issues are enumerated in Figure 7.1 to facilitate further referencing. This thesis answers these questions positively by proposing

a) *does the approach scale?*

b) *do decompiled programs preserve the structure of the original ones?*

c) *is the "quality" of decompiled programs comparable to that obtained by dedicated decompilers?*

Figura 7.1: Open Issues of Interpretive Decompilation

a modular decompilation scheme which can be steered to control the structure of decompiled code and ensure quality decompilations which preserve the original program's structure.

The rest of the chapter is organized as follows:

- We first provide in Section 7.1 an informal background on PE of logic programs in order to enable the reader to understand the details explained throughout the chapter. A more formal background is given in Section 2 of Paper 6.
- Section 7.2 introduces the challenges behind the specialization a bytecode interpreter through a representative example.
- Section 7.3.3 informally introduces the *Type-based homeomorphic embedding*, an extension of the original *homeomorphic embedding* relation which, by taking information about the behavior of the computation into account, provides more precise results in the presence of infinite signatures. The formal details and an experimental evaluation is presented in Papers 4 and 3.
- Sections 7.4 and 7.5 introduce the necessary ingredients to develop a *modular* and *optimal* decompilation scheme addressing issues *a)*, *b)* and *c)*. The notion of *optimality* is first defined by means of a series of *optimality criteria*. Then, the problems of *non-modular* decompilation are presented and the components needed to enable a *modular* scheme are identified. This includes how to write an interpreter and how to control an *online* partial evaluator in order to preserve the structure of the original program w.r.t. method invocations. We finally introduce an interpretive decompilation scheme which answers

issues *(a)*, *(b)* and *(c)* by producing decompiled programs whose *quality* is equivalent to that of dedicated decompilers. This requires a *block-level* decompilation scheme which avoids code duplication and code re-evaluation.

- Section 7.6 summarizes our experimental results on a prototype implementation of a decompiler of Java bytecode to Prolog which incorporates the above techniques, and demonstrates its scalability and efficiency on an set of realistic Java Bytecode programs.

- Finally, Section 7.7 discusses related work on interpretive decompilation of bytecode languages.

For the sake of concreteness, our interpretive decompilation scheme is formalized in the context of PE of logic programs but the ideas we propose for enabling the practicality of the approach are also of interest for the interpretive (de)compilation of any pair of source and target languages.

7.1. Basics of Partial Evaluation of Logic Programs

We assume familiarity with basic notions of logic programming [68]. Executing a logic program P for an atom A consists in building a so-called SLD tree for $P \cup \{A\}$ and then extracting the computed answer substitutions from every non-failing branch of the tree. Partial evaluation builds upon the execution approach of logic programs with two main differences:

- In order to guarantee termination of the *unfolding* process, when building the SLD-trees, it is possible to choose *not* to further unfold a goal, and rather leave a leaf in the tree with a non-empty, possibly non-failing, goal. The resulting SLD tree is called a *partial* SLD tree. Note that even if the SLD trees for all possible queries are finite, the SLD tree to be built during partial evaluation may be infinite. The reason for this is that since dynamic values are not known at specialization time, the specialization SLD tree can have more branches (in particular, infinite branches) than the actual SLD tree at run-time.

```
1: function EP (P, A, S)
2:     S_0 := S;  i := 0;
3:     repeat
4:         L^pe := unfold(S_i, P, A);
5:         S_{i+1} := abstract(S_i, L^pe, A);
6:         i := i + 1;
7:     until S_i = S_{i-1}      % (modulo renaming)
8:     return codegen(L^pe, unfold);
```

Figura 7.2: A generic PE algorithm for logic programs

Which atom to select from each resolvent and when to stop unfolding is determined by the *unfolding rule*.

- The partial evaluator may have to build several SLD-trees to ensure that all atoms left in the leaves are "covered" by the root of some tree (this is known as the *closedness* condition of EP [67]). The so-called *abstraction operator* performs "generalizations" on the atoms that have to be partially evaluated in order to avoid computing partial SLD trees for an infinite number of atoms. When all atoms are covered, then there is no need to build more trees and the process finishes.

The essence of most algorithms for partial evaluation of logic programs (see e.g. [42]) can be viewed in the algorithm shown in Figure 7.2, which is parametric w.r.t. the unfolding rule, unfold, and the abstraction operator, abstract. EP starts from a program P, a (possibly empty) set of annotations $\mathcal{A}$ and an initial set of calls S. At each iteration, the so-called *local control* is performed by the unfolding rule unfold (Line 4), which takes the current set of atoms S_i, the program and the annotations and constructs a *partial* SLD tree for each call in S_i. In the *global control*, which is performed by the abstraction operator abstract, when some calls in the leaves of the trees are not properly *covered*, the operator abstract adds them to the new set of atoms to be partially evaluated in a proper "generalized" form such that termination is ensured (i.e., the condition $S_i = S_{i-1}$ is reached).

A partial evaluation of P w.r.t. S is then systematically extracted from the resulting set of calls L^{pe} in the final phase, codegen in L8. The notion of *resultant* is used to generate a program rule associated to each root-

to-leaf derivation of the SLD-trees for the final set of atoms L^{pe}. Given an SLD derivation of $P \cup \{A\}$ with $A \in L^{pe}$ ending in B and θ being the composition of the mgu's in the derivation steps, the rule $\theta(A) : -B$ is called the *resultant* of the derivation. A PE is defined as the set of resultants (clauses) associated to the derivations of the constructed partial SLD trees for all $P \cup L^{pe}$. The resulting program is often referred to as the *specialized program* or *residual program*. A more formal background is given in Section 2 of Paper 6.

7.1.1. Online vs. Offline Partial Evaluation

It is well-known that both the quality of the specialized programs and the time required for the PE process greatly vary with the control strategies used. Traditionally, two approaches to PE have been considered, *online* and *offline* PE. In online PE, all control decisions are taken on the fly during the specialization phase by keeping track of the specialization history. In the offline approach, all control decisions are taken before the proper specialization phase. These control decisions are based on abstract descriptions of the data instead of the actual data. The control strategy is usually represented as program annotations which are the sole decision criteria for control of the partial evaluator. For instance, in the local control, an annotation can explicitly indicate that an atom should not be unfolded. In the global control, annotations typically specify for each call which arguments have to be generalized away (i.e. replaced by variables). Such annotations are in some partial evaluators automatically generated by a *binding-time analysis* and in other partial evaluators they are manually provided by the user, either in part or in full.

Under this classification, the PE algorithm we propose can be considered a generic or a hybrid approach since the $\mathcal{A}$ annotations can provide information to the control operators, as in offline PE, and the algorithm can include control rules based on the actual specialization history, as in online PE. The advantages of the offline approach are that, once all control annotations are available, PE is quite simple and efficient. On the other hand, online PE, though less efficient, has a strictly more powerful control strategy since control decisions are based on actual data instead of abstract descriptions of data. Therefore, though all offline PEs can be re-

plicated using online techniques, many online PEs cannot be reproduced using offline techniques.

In this work we are interested in investigating how far we can go with the more powerful but less efficient online PE approach. The motivation for this is that this way we may obtain decompilations of higher quality than those achievable using offline PE. Thus, our challenges are both in terms of quality of the decompiled programs and in terms of efficiency of the decompilation process. As we will see later, many of the lessons learned in this thesis are of interest both to the online and offline approaches to the PE of interpreters.

7.2. Challenges in the Specialization of Bytecode Interpreters

This section illustrates the challenges which appear in the specialization of a bytecode interpreter by means of an example. Fig. 7.3 shows a fragment of a bytecode interpreter implemented in Prolog. We assume that the code for every method in the bytecode program is represented as a set of facts `bytecode/3` such that, for every pair $pc_i : bc_i$ in the code for method m, we have a fact `bytecode(`m`,`pc_i`,`bc_i`)`. The state carried around by the interpreter is of the form `st(Fr,FrStack)` where `Fr` represents the current frame (environment) and `FrStack` the stack of frames (call-stack) implemented as a list. Frames are of the form `fr(M,PC,OStack,LocalV)`, where `M` represents the current method, `PC` the program counter, `OStack` the operand stack and `LocalV` the list of local variables. Predicate `main/3`, given the method to be interpreted `Method` and its input method arguments `InArgs`, first builds the initial state by means of predicate `build_s0/3` and then calls predicate `execute/2`, returning `Res`, which is the top of the operand stack at the end of the computation. In turn, `execute/2` calls predicate `step/3`, which produces `S'`, the state after executing the bytecode, and then calls predicate `execute/2` recursively with `S'` until we reach a `return` instruction with the empty stack. For brevity, we only show the definition of `step/3` for a selected set of instructions and omit the code of some auxiliary predicates. Namely `build_s0/3`, which was explained above, `next/3`, which produces the next program counter given the current one, and `split_OS/4`, which

```
main(Method,InArgs,Res) :-
   build_s0(Method,InArgs,S0),
   execute(S0,Sf),
   Sf = st(fr(_,_,[Res|_],_),_)).

execute(S,S) :-
   S = st(fr(M,PC,_,_),[]),
   bytecode(M,PC,return).
execute(S,Sf) :-
   S = st(fr(M,PC,_,_),_),
   bytecode(M,PC,Inst),
   step(Inst,S,S'),
   execute(S',Sf).

step(goto(PC),S,S') :-
   S = st(fr(M,_,OS,LV),FrS),
   S' = st(fr(M,PC,OS,LV),FrS).
```

```
step(push(X),S,S') :-
   S = st(fr(M,PC,OS,L),FrS),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[X|OS],L),FrS).
...
step(invoke(M'),S,S') :-
   S = st(fr(M,PC,OS,LV),FrS),
   split_OS(M',OS,Args,OS''),
   build_s0(M',Args,
            st(fr(M',PC',OS',LV'),_)),
   S' = st(fr(M',PC',OS',LV'),
           [fr(M,PC,OS'',LV)|FrS]).
step(return,S,S') :-
   S = st(fr(_,_,[RV|_],_),
          [fr(M,PC,OS,LV)|FrS]),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[RV|OS],LV),FrS).
```

Figura 7.3: Fragment of a bytecode interpreter

splits the current operand stack into the parameters list to be used in the called method and the rest.

Figure 7.4 depicts the bytecode program that we will use as our working example. On the top of the figure we depict the Java source code for clarity. Note that the decompiler works directly on the bytecode which is shown at the bottom. Our working example consists of a set of methods that carry out different arithmetic operations. Method `gcd` computes the greatest-common divisor, `abs` the absolute value and `fact` the factorial recursively. Method `count` has no particular meaning, it just increments a counter initialized to 0 until its value reaches the value of the given argument.

In order to achieve an *effective* decompilation, one of the crucial requirements is to have available control strategies (i.e., unfold and abstract operators) which are powerful enough to remove the interpreter overhead. For this reason, our first experiments in Paper 1 were performed using "aggressive" control strategies based on *homeomorphic embedding* [58, 62]. In local control, by aggressiveness we mean unfolding rules which compute derivations as long as possible provided there are no termination problems.

```
int count(int n){
   int i = 0;
   while (i < n) i++;
   return i;}

int gcd(int x,int y){
   int res;
   while (y != 0){
      res = x%y; x = y;
      y = res;}
   return abs(x);}
```

```
int abs(int x){
   if (x < 0) return -x;
   else return x;
}

int fact(int x){
   if (x == 0)
      return 1;
   else
      return x*fact(x-1);
}
```

```
Method count
  0:push(0)
  1:store(1)
  2:load(1)
  3:load(0)
  4:ifge(3)
  5:inc(1,1)
  6:goto(2)
  7:load(1)
  8:return
```

```
Method gcd
  0:load(1)
  1:if0eq(11)
  2:load(0)
  3:load(1)
  4:rem
  5:store(2)
  6:load(1)
  7:store(0)
  8:load(2)
  9:store(1)
  10:goto 0
  11:load(0)
  12:invoke(abs)
  13:return
```

```
Method abs
  0:load(0)
  1:if0ge(5)
  2:load(0)
  3:neg
  4:return
  5:load(0)
  6:return
```

```
Method fact
  0:load(0)
  1:if0ne(4)
  2:push(1)
  3:return
  4:load(0)
  5:load(0)
  6:push(1)
  7:sub
  8:invoke(fact)
  9:mul
  10:return
```

Figura 7.4: Source code and bytecode for working example

In global control, it denotes abstraction operators which generalize in as few situations as possible without endangering termination.

Figure 7.5 depicts the decompiled program we obtain using the state-of-the-art partial evaluator available in the CiaoPP system [49]. For this preliminary experiments the *homeomorphic embedding* was used both for the local and global control levels. By looking at the code we observe the following:

1. The partial evaluator has not been able to successfully decompile the

```
main(count,[N],A) :-
    % out of memory error

main(gcd,[A,0],A) :- A>=0.
main(gcd,[B,0],A) :-
    B<0, A is -B.
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).
```

```
execute_1(A,0,A) :- A>=0.
execute_1(A,0,C) :- A<0, C is -A.
execute_1(A,B,G) :-
    B\=0, I is A rem B,
    execute_1(B,I,G).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[N],A) :- ...% Full interpreter
```

Figura 7.5: Decompiled code for working example. First attempt.

`count` method. It runs out of memory because there are termination problems both at the local and at the global control. The reason for this is that the control rules based on the *homeomorphic embedding* do not ensure termination of PE in programs which can potentially generate infinite values, as it is the case of the bytecode interpreter (in particular because of the `is/3` predicate).

2. The method compositionality of the original program has been lost in the decompiled program. This can be seen by looking at the decompiled code of the `gcd` method. Note that it does not call `abs` but instead it has inlined its code. Though this might be seen as positive from the point of view of the specialization, the consequences can highly degrade the efficiency and quality of the process and makes impossible to scale up when considering realistic bytecode programs with calls to libraries.

3. The decompiled code we obtain for the `fact` method basically contains the full interpreter and is not shown in the figure due to space limitations. The problem was first detected in [43] and arises when the method to be decompiled is recursive. As we will see, this problem (and also its solution) is very related to the composisionality problem explained in the previous item.

4. If we look at the code of the `main(gcd,...)` and `execute/3` predicates, we can see that there are code duplications. Even more, the

partial evaluator has produced such duplications because some part of the bytecode program has been re-evaluated during EP. Our experimental evaluation demonstrates that having or not these duplications (re-evaluations) makes the difference between being or not being able to scale up in practice.

The solutions to the above problems are summarized in the following three challenges:

- **Challenge I. Handling infinite signatures in PE:** We first review existing solutions identifying their flaws and then introduce the *type-based homeomorphic embedding*. This issue is further discussed in Section 7.3 and elaborated in more detail in Papers 3 and 4.
- **Challenge II: A modular decompilation scheme.** Even after achieving *Challenge I*, we will see that it is a necessity to design a modular decompilation scheme which preserves the method compositionality of the original programs and, besides, solves the problem with recursive programs. Such a decompilation scheme is introduced in Section 7.4 and further studied in detail in Paper 6.
- **Challenge III: Optimal decompilation.** Preliminary experiments performed using the modular decompilation scheme with realistic programs show that it is yet not possible to successfully scale up. We therefore introduce an *optimal* decompilation scheme which ensures that the decompilation times and decompiled program sizes grow linearly w.r.t. to the size of the input programs by avoiding code duplications and re-evaluations. This issue is introduced in Section 7.5 and further studied in depth in Paper 6.

7.3. Challenge I: Handling Infinite Signatures

7.3.1. The Homemorphic Embedding

The *homeomorphic embedding* (HEm) relation [58, 61, 62] has become very popular to ensure online termination of *symbolic* transformation and

specialization methods and it is essential to obtain powerful optimizations, for instance, in the context of online PE. Intuitively, HEm is a structural ordering under which an expression t_1 *embeds* expression t_2, written as $t_2 \trianglelefteq t_1$, if t_2 can be obtained from t_1 by deleting some operators, e.g., $\underline{\mathtt{s}}(\mathtt{s}(\underline{\mathtt{U}}+\mathtt{W})\underline{\times}(\underline{\mathtt{U}}+\underline{\mathtt{s}}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U} \times (\mathtt{U} + \mathtt{V}))$.

The HEm relation can be used to guarantee termination because, assuming that the set of constants and functors is finite, every infinite sequence of expressions $t_1, t_2, \ldots$, contains at least a pair of elements t_i and t_j with $i < j$ s.t. $t_i \trianglelefteq t_j$. Therefore, when iteratively computing a sequence $t_1, t_2, \ldots, t_n$, finiteness of the sequence can be guaranteed by using HEm as a "whistle". Whenever a new expression t_{n+1} is to be added to the sequence, we first check whether $t_i \not\trianglelefteq t_{n+1}$ for all i s.t. $1 \leq i \leq n$. If that is the case, finiteness is guaranteed and computation can proceed. Otherwise, HEm is not capable of guaranteeing finiteness and the computation has to be stopped. The intuition is that computation can proceed as long as the new expression is not larger than any of the previously computed ones since that is a sign of potential non-termination. The success of HEm is due to the fact that sequences can usually grow considerably large before the whistle blows, when compared to other online approaches for guaranteeing termination.

While HEm has been proved very powerful for symbolic computations, some difficulties remain in the presence of infinite signatures such as the numbers. In the case of logic programs, infinite signatures appear as soon as certain Prolog built-ins such `is/2`, `functor/3` and `name/2` are used. HEm relations over infinite signatures have been defined (e.g. [61, 6]), but they tend to be too conservative in practice ("whistling" too early).

7.3.2. A Challenging Example

Consider the `count` method which appears in the left hand side of Figure 7.4. It can be seen that `count` receives an integer and executes a loop where a counter initialized to "0" (in bytecodes `0` and `1`) is incremented by one at each iteration (bytecode `5`) until the counter reaches the value of the input parameter (checking the condition comprises bytecodes `2`, `3` and `4`). The method returns the value of the counter in bytecodes `7` and `8`. In order to decompile the `count` method, we partially evaluate the interpreter

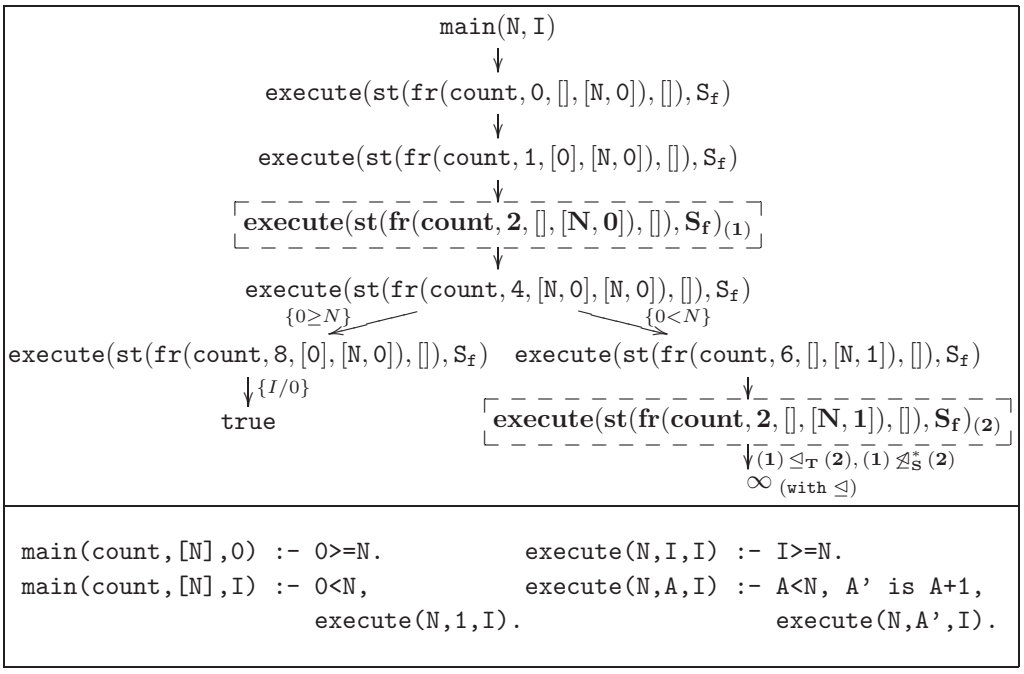


Figura 7.6: Partial unfolding SLD tree and residual code of working example

in Figure 7.3 w.r.t. the `count` bytecode method by specializing the atom `main(count,[N],I)`, where `N` is the input parameter and `I` represents the returned value (i.e., the top of the stack at the end of the computation).

In Figure 7.6, we depict (a reduced version of) one of the SLD trees that leads to an effective decompilation of the `count` method and that we will refer to below. For simplicity, apart from the entry atom `main/3`, we only show atoms for `execute/2`, as it is the only recursive predicate in the program. Thus, each arrow in the tree involves the application of several unfolding steps. Note that some of the statements within the body of each `step` operation can remain residual when they involve data which is not known at specialization time. The computation rule used in the unfolding operator is able to residualize calls which are not sufficiently instantiated and select non-leftmost atoms in a safe way [7], in particular, further calls to `execute` can be selected. We represent such residual calls as labels in the arrows of the tree.

Using the Original HE

Let us first consider an online partial evaluator which uses HEm to control termination both at the local and global control levels. As it can be seen in the figure, the PC value "2" corresponds to the loop entry. By applying HEm, the evaluation contains a subsequence of atoms of the form: execute(st(fr(count, 2, [], [N, 0]), []), S_f), execute(st(fr(count, 2, [], [N, 1]), []), S_f), execute(st(fr(count, 2, [], [N, 2]), []), S_f), ... marked within dashed frames in the figure, which correspond to consecutive iterations of the loop in which the control returns to the loop head (PC value 2 in the first position of the state) with a value for the loop counter (local variable at the second position in the resulting state) increased by one. This sequence can grow infinitely, as the HEm does not flag it as potentially dangerous, which is marked by "∞ (with $\trianglelefteq$)" in the figure. This is because the interpreter uses Prolog's arithmetic (i.e., the is/2 predicate), which breaks the finite signature property featured by pure logic programs.

In order to get a quality decompilation, we need to filter out the value of the counter (local variable 1) but not that of the PC. As shown in the figure, this requires stopping the derivation when we hit the atom execute(st(fr(count, 2, [], [N, 1]), []), S_f) (marked as $(\mathbf{1}) \trianglelefteq_{\mathbf{T}} (\mathbf{2})$) and generalize it w.r.t. the above atom within a dashed frame, resulting in execute(st(fr(count, 2, [], [N, X]), []), S_f).

Recovering Termination: Embedding with Number Filtering

In programs which contain Prolog arithmetic but do not generate an infinite number of functors via functor/3, =../2, etc., a relatively straightforward solution in order to recover termination is to use the $\trianglelefteq_{num}$ relation, which is an adaptation of HEm which filters out numeric values, i.e., any number embeds any other number. The atom execute(st(fr(count, 2, [], [N, 1]), []), S_f) embeds execute(st(fr(count, 2, [], [N, 0]), []), S_f) under $\trianglelefteq_{num}$ and therefore we avoid non-termination. Unfortunately, this modification to HEm, is far too conservative, and leads to excessive precision loss. For instance, in the specialization of main(count, [N], I), the first two atoms for execute/2 are execute(st(fr(count, 0, [], [N, 0]), []), S_f) and execute(st(fr(count, 1, [0], [N, 0]), []), S_f). By using $\trianglelefteq_{num}$, the whistle

blows at this point and unfolding has to stop. Furthermore, the latter atom is generalized at the global control level into $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{X},\texttt{Y},[\texttt{N},\texttt{0}]),[]),\texttt{S}_\texttt{f})$ before proceeding with the specialization. This turns out not to be acceptable for the specialization of our interpreter, since we lose track of which the next instruction to execute is—which prevents us from eliminating the interpretation layer—and in many cases the residual program ends up containing the whole original interpreter.

Increasing Accuracy: Static Symbols in the Program

A simple syntactic way of increasing the accuracy while preserving termination, as proposed in [61], consists in considering two sets of symbols: those which appear explicitly in the program and goal, which is obviously finite, and another infinite set which contains all other symbols. In the following, this relation is denoted as $\trianglelefteq^*_S$. When comparing two terms we keep those symbols which belong to the finite set and filter out all other ones. Under this relation, the atom $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{1},[\texttt{0}],[\texttt{N},\texttt{0}]),[]),\texttt{S}_\texttt{f})$ does not embed the atom $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{0},[],[\texttt{N},\texttt{0}]),[]),\texttt{S}_\texttt{f})$ in the figure, as the numbers `0` and `1` are different static symbols which occur in the program. Hence, we are not forced to generalize them and we can keep the `PC` value.

Unfortunately, the $\trianglelefteq^*_S$ relation turns out not to be optimal in our case either since $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{2},[],[\texttt{N},\texttt{1}]),[]),\texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{2},[],[\texttt{N},\texttt{0}]),[]),\texttt{S}_\texttt{f})$. This means that unfolding proceeds with a second iteration of the loop. The process is guaranteed to terminate, we will unfold at most as many iterations of the loop as distinct numbers appear in the program. However, we are not able to achieve the quality decompilation which appears at the bottom of Figure 7.6. For obtaining such good decompilation, we need to generalize the loop counter, i.e., the atom $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{2},[],[\texttt{N},\texttt{1}]),[]),\texttt{S}_\texttt{f})$ has to embed $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{2},[],[\texttt{N},\texttt{0}]),[]),\texttt{S}_\texttt{f})$. Intuitively, the reason why this relation does not behave optimally is because many symbols which appear explicitly in the program for one argument (in our case the PC counter) should not affect the set of symbols which we should consider as static for other arguments (the list of local variables).

In conclusion, this example suggests that embeddings that take context

information into account are needed: a context-sensitive embedding should handle in a different way the PC values and the numeric values in program variables such as the loop counter.

7.3.3. Type-based Homeomorphic Embedding

In the presence of infinite signatures, a general method of defining homeomorphic embedding relations exists; an *extended homeomorphic embedding relation* is defined in [61] based on previous results by Kruskal [58] and by Dershowitz [38]. This solution defines a family of embedding relations, where a subsidiary ordering on function symbols plays an essential role. However, we argue that this does not really solve the practical problem of finding an effective embedding relation, since there is no automated mechanism for finding the "right" ordering relation on the function symbols in the signature.

In this thesis, we propose the *type-based homeomorphic embedding* (TbHEm for short), a relation which improves HEm by making use of additional information provided in the form of types. We outline how this approach can be seen as a way of generating instances of extended HEm as defined by Leuschel, including the possibility of taking into account the program semantics. The types required for guiding TbHEm can be provided manually or, interestingly, be automatically inferred by program analysis, as we discuss in Paper 3.

A starting point of TbHEm is the observation that, even if an expression is defined over an infinite signature, it might only take a finite set of values over such domain for each computation. To perform such a distinction our typed relation is defined on types which are structured into a (possibly empty) finite part and a (possibly empty) infinite partition. Intuitively, TbHEm allows expanding sequences as long as, whenever we compare sub-terms from an infinite type, the concrete values which appear in the expression remain within the finite part of the type.

Using the TbHEm to control the PE of the bytecode interpreter

In the case of our bytecode interpreter, the PC argument can be defined by a structured type such that the bounded interval in which it ranges constitutes its finite partition and the remaining integers form its infinite

part. This way, the TbHEm will not generalize the `PC` as long as its value remains within the bounded interval.

In order to infer such type, let us rely on existing analysis techniques, namely on the inference of well-typings described by Bruynooghe *et al.* [20][1]. The following type τ_{PC} for the program counter argument is inferred for the interpreter of Figure 7.3, together with the particular bytecode program of Figure 7.4:

```
τPC --> -4; 0; 1; 2; 3; 4; 5; 6; 7; 8; num
```

Type τ_{PC} can be naturally interpreted as consisting of a finite part (the named constants) and an infinite part (the numbers other than the named constants). In other words, the partition F of the rule is $\{-4, 0, 1, 2, \ldots, 8\}$ and $I = \texttt{num} \setminus F$. Using the rule structured in this way, TbHEm ensures that the program counter is never abstracted away during partial evaluation, so long as its value remains in the expected range (the named constants). The atom $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 1, [0], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 0, [], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f})$ by using the type definition above, thus, the derivation can proceed. This avoids the need for generalizing the `PC` what would prevent us from having a quality specialization (decompilation) as explained above. The derivation will either eventually end or the `PC` value will be repeated due to a backwards jump in the code (loops). In this case, the TbHEm, also written $\trianglelefteq_T$, will flag the relevant atom as dangerous, e.g., $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f}) \trianglelefteq_\texttt{T} \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 1]), []), \texttt{S}_\texttt{f})$, as can be seen in Figure 7.6.

The decompiled program that we obtain using the inferred typings and combined with TbHEm is shown at the bottom of Figure 7.6. We can observe that the decompilation is optimal[2] in the sense that the interpretation layer has been completely removed and there is no superfluous residual code.

Besides the inference of well-typings we saw above, Paper 3 also outlines how analysis of numeric bounds can be used to infer useful information for TbHEm. Such analysis makes over-approximations of the set of values that

[1] Available on-line at `http://saft.ruc.dk/Tattoo/`

[2] We will see later that this can be further improved

```
main(count,[N],0) :- 0>=N.
main(count,[N],I) :-
    0<N, execute_2(N,1,I).

execute_2(N,I,I) :- I>=N.
execute_2(N,A,I) :-
    A<N, A' is A+1,
    execute_2(N,A',I).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[N],A) :- ...% full int.
```

```
main(gcd,[A,0],A) :- A>=0.
main(gcd,[B,0],A) :-
    B<0, A is -B.
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).

execute_1(A,0,A) :- A>=0.
execute_1(A,0,C) :- A<0, C is -A.
execute_1(A,B,G) :-
    B\=0, I is A rem B,
    execute_1(B,I,G).
```

Figura 7.7: Decompiled code for working example after overcoming Chall. I

the program arguments can have. Intuitively, when we can prove that such set of values is bounded, then we know that the infinite partition of the type is empty and, hence, we can safely apply traditional HEm (and improve the effectiveness of PE).

Note that, determining the exact set of symbols that can appear at runtime at a specific program point, and in particular determining whether the set is finite, is closely related to termination detection and is thus undecidable. However, the better the derived types are, the more aggressive partial evaluation can be without risking non-termination. If the derived types have finite components that are too small, then over-generalization is likely to result; if they are too large, then specialization might be over-aggressive, producing unnecessary versions.

7.4. Challenge II: Modular Decompilation

Once we have overcome the problem of handling infinite signatures, the class of bytecode programs which can be successfully decompiled is significantly wider. Another issue, which is not further discussed in this introduction, is that using the classical online abstraction operator which is simply based on the original HEm, or even enhancing it by using the TbHEm, the decompiled programs we obtain tend to have too many (re-

dundant) specialized versions of some predicates. This is studied in detail in Paper 2 where we propose an advanced abstraction operator which is able to control the *polyvariance* of the PE process, i.e., which is able to avoid having such redundant specialized versions. As it is shown in Paper 2, this allows obtaining better decompiled programs, in a more efficient way, which also widens a bit more the class of programs which we can successfully decompile. Nevertheless, even after enhancing the partial evaluator so that it integrates both the TbHEm and such advanced abstraction operator, the current scheme can still be rather unsatisfactory with realistic programs, since the compositionality of the original programs as regards method calls is lost.

Let us consider again our bytecode example in Figure 7.4. The decompiled code we obtain using the enhanced partial evaluator is shown in Figure 7.7. It can be noted that it is the same as the one in Figure 7.5 except for the method `count` for which the code at the bottom of Figure 7.6 is now obtained. Note that, this example is not complex enough to expose the problem of polyvariance that the advanced abstraction operator solves. We refer the reader again to Paper 2 where a representative example is presented.

We now identify four limitations, which we name as **(L1)**... **(L4)**, of the current decompilation (from now on *non-modular* decompilation). It is important to note that such limitations, and the way to avoid them which we introduce later, are also relevant to the case of offline PE.

(L1) Calls to methods are *inlined* within their calling contexts and, as a consequence, the structure of the original code is lost. For example, the method invocation from `gcd` to `abs` (index 12 of `gcd`) does not appear in the decompiled code. As a result, the decompiled code for `gcd` has two base cases in which the builtins of `abs` are inlined, namely, `A>=0`, `B<0` and `A is -B`. This happens because calls to methods are dealt with in a *small-step* fashion within the interpreter, i.e., the code of invoked methods is unfolded as if it was inlined inside the "caller" method.

(L2) As a consequence, decompilation becomes very inefficient. E.g., if n calls to the same method appear within a code, such method will be decompiled n times. Even worse, if there is a method invocation inside a loop, its code will be evaluated twice in the best case, as we have to perform the corresponding generalizations in the global control before reaching a

fixpoint. This can be even worse in the case of nested loops.

(L3) The non-modular approach does not work incrementally, in the sense that it does not support *separate* decompilation of methods but rather has to (re)decompile all method calls. Thus, decompiling a real language becomes unfeasible, as one needs to consider system libraries, whose code might not be available. Limitation L2 together with L3 answer issue *(a)* of Figure 7.1 negatively.

(L4) The decompiled code contains basically the whole interpreter when there are recursive methods. This is why the decompiled program in Figure 7.7 does not contain the code corresponding to the recursive `fact` method. The problem with recursion is as follows. Assume we want to decompile method $m1$ whose code is $\langle pc_0 : bc_0, \ldots, pc_j : invoke(m1), \ldots, pc_n : return\rangle$. There is an initial decompilation for $A_k = \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}, \texttt{lv}), []), \texttt{S}_\texttt{f})$ in which the call-stack is empty. During its decompilation, a call of the form $A_l = \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}', \texttt{lv}'), [\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}, \texttt{lv})]), \texttt{S}_\texttt{f})$ with the call-stack containing the previous frame appears when we arrive to the recursive call. At this point, the derivation must be stopped as $A_k \trianglelefteq_T A_l$. In order to ensure termination, global control generalizes the above calls into $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \_, \_), \_), \texttt{S}_\texttt{f})$, where $\_$ denotes a fresh variable and thus the call-stack has become unknown. As a consequence, after evaluating the *return* statement, the continuation obtained from the call-stack is unknown and we produce the call $\texttt{execute}(\texttt{st}(\texttt{fr}(\_, \_, \_, \_), \_), \texttt{S}_\texttt{f})$ to be decompiled. Here, the fact that the method and the program counter are unknown prevents us from any chance of removing the interpretation layer, i.e., the decompiled code will potentially contain the whole interpreter. This indeed happens during the decompilation of `fact`. Limitations L1 and L4 answer issue *(b)* (see Figure 7.1) negatively.

We now identify the ingredients which are necessary in order to achieve a *modular* decompilation scheme. By *modular* decompilation, we refer to a decompilation technique whose decompilation unit is the method, i.e., we decompile a method at a time. We show that this approach overcomes the four limitations of non-modular decompilation described above and answers issues *(a)* and *(b)* of Figure 7.1 positively. In essence, we need to: (i) Give a compositional treatment to method invocations. We show that this can be achieved by considering an interpreter implemented using a *big-step* semantic. (ii) Provide a mechanism to residualize calls in the decompiled

program (i.e., do not unfold them and add them without modifications to the residual code). We automatically generate program annotations for this purpose. (iii) Study the conditions which ensure that *separate* decompilation of methods is sound.

7.4.1. Big-step Semantics Interpreter to Enable Modularity

Traditionally, two different approaches have been considered to define language semantics, *big-step* (or *natural*) semantics and *small-step* (or *structural operational*) semantics (see, e.g., [59]). Essentially, in big-step semantics, transitions relate the initial and final states for each statement, while in small-step semantics transitions define the *next* step of the execution for each statement. In the context of bytecode interpreters, it turns out that most of the statements execute in a single step, hence making both approaches equivalent for such statements. This is the case for our bytecode interpreter in Figure 7.3 for all statements except for *invoke*. The transition for *invoke* in small-step defines the next step of the computation, i.e., the current frame is pushed on the call-stack and a new environment is initialized for the execution of the invoked method. Note that, after performing this step, we do not distinguish anymore between the code of the caller method and that of the callee. This prevents us from having modularity in decompilation.

In the context of interpretive (de)compilation of imperative languages, small-step interpreters are commonly used (see e.g. [77, 48]). We argue that the use of a big-step interpreter is a necessity to enable modular decompilation which scales to realistic languages. In Fig. 7.8, we depict the relevant part of a big-step version for our bytecode interpreter. We can see that the *invoke* statement, after extracting the method parameters from the operand stack, calls recursively predicate `main/3` in order to execute the callee. Upon return from the method execution, the return value is pushed on the operand stack of the new state and execution proceeds normally. Also, we do not need to carry the call-stack explicitly within the state, but only the information for the current environment, i.e., states are of the form `st(M,PC,OStack,LocalV)`. This is because the call-stack is already available by means of the calls for predicate `main/3`.

The compositional treatment of methods within the interpreter is not only essential to enable modular decompilation (overcome L1, L2 and L3) but also to solve the recursion problem in a simple and elegant way. Indeed, the decompilation based on the big-step interpreter does not present L4. E.g., the decompilation of a recursive method $m1$ starts from the call $\texttt{main}(\texttt{m1}, \_, \_)$ and then reaches a call $\texttt{main}(\texttt{m1}, \texttt{args}, \_)$ where `args` represents the particular arguments in the recursive call. This call is flagged as dangerous by local control and the derivation is stopped. The important points are that, unlike before, no re-computation is needed as the second call is necessarily an instance of the first one and, besides, there is no information loss associated to the generalization of the call-stack, as there is no stack.

Partial solutions to the recursion problem exist and are discussed in the following. The problem was first detected in [43] and a solution based on computing regular approximations during PE was proposed. Although feasible in theory, such technique might be too inefficient in practice and problematic to scale it up to realistic applications due to the overhead introduced by the underlying analysis. Another solution is proposed in [48], where a simpler control-flow analysis is performed before PE in order to collect all possible instructions which might follow the *return*. Such information may then be used to recover information lost by the generalization. This solution turns out to be also impractical for our purposes when considering realistic programs that make intensive use of library code (e.g. Java Bytecode) as many continuations can follow. Our solution does not require the use of static analysis and, as our experiments show, does not pose scalability problems.

It is important to note that the idea of using a big-step semantics for describing the interpreter in order to achieve modular (de)compilation is equally useful in the offline approach to interpretive decompilation. Furthermore, to the best of our knowledge, our idea is novel and has not been proposed before, neither in online nor in offline PE of interpreters.

7.4.2. The Modular Decompilation Scheme

In addition to use a big-step interpreter, it is necessary in order to design a modular decompilation scheme to: 1) provide a mechanism to

```
execute(S,S) :-
   S = st(M,PC,[_Top|_],_),
   bytecode(M,PC,return).
execute(S,Sf) :-
   S = st(M,PC,_,_),
   bytecode(M,PC,Inst),
   step(Inst,S,S'),
   execute(S',Sf).
```

```
step(invoke(M'),S,S') :-
   S = st(M,PC,OS,LV),
   next(M,PC,PC'),
   split_OS(M',OS,Args,OSRs),
   main(M',Args,RV),
   S' = st(M,PC',[RV|OSRs],LV).
```

Figura 7.8: Fragment of big-step bytecode interpreter

residualize calls in the decompiled program (i.e., do not unfold them and add them without modifications to the residual code), and 2) define the notion of *separate* decompilation and study the conditions which ensure its soundness.

Paper 6 studies in detail these issues and defines a modular decompilation scheme whose correctness and completeness is formally proven. It is also proven that the proposed scheme satisfies the *method-optimality* criterion, which ensures that each method is decompiled only once.

Modular decompilation basically works as follows: when a method invocation is to be decompiled, the call `step(invoke(m'),_,_)` occurs during unfolding. We can see that, by using the big-step interpreter in Fig. 7.8, a subsequent call `main(m',_,_)` will be generated. At this point, there will be an annotation indicating to the partial evaluator to not to unfold this call and rather add it without modifications to the residual code. If `m'` is internal (i.e., it is defined in the input program), a corresponding decompilation from the call `main(m',_,_)` will be, or has already been, performed since modular decompilation ensures that the PE is executed for every method in the bytecode program.

Figure 7.9 shows the decompiled program we obtain using the modular decompilation scheme with our working example. It can be observed that the structure of the original program w.r.t. method calls is now preserved, as the residual predicate for `gcd` contains an invocation to the definition of `abs`, as it happens in the original bytecode. Moreover, we now obtain an effective decompilation for the recursive method `fact` where the interpretive layer is completely removed. Thus, L1 and L4 have been successfully

```
main(count,[N],0) :- 0>=N.
main(count,[N],I) :-
    0<N, execute_2(N,1,I).

execute_2(N,I,I) :- I>=N.
execute_2(N,A,I) :-
    A<N, A' is A+1,
    execute_2(N,A',I).

main(gcd,[B,0],A) :-
    main(abs,[B],A).
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).
```

```
execute_1(A,0,C) :- main(abs,[A],C).
execute_1(A,B,F) :- B\=0,
                    H is A rem B,
                    execute_1(B,H,F).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[B],A) :- B\=0,
                    C is B-1,
                    main(fact,[C],D),
                    A is B*D.
main(fact,[0],1).
```

Figura 7.9: Decompiled code obtained using modular decompilation

solved.

Note that modular decompilation gives a monovariant treatment to methods in the sense that it does not allow creating specialized versions of method definitions. This is against the usual spirit in PE, where polyvariance is a main goal to achieve further specialization. However, in the context of decompilation, we have shown that a monovariant treatment is necessary to enable scalability and to preserve program structure. It naturally raises the question whether a polyvariant treatment could achieve, even if at the cost of efficiency and loss of structure, a better quality decompilation. Note that enabling polyvariant specialization in the modular setting can be trivially done by not introducing the corresponding annotations for certain selected methods which should be treated in a polyvariant manner. Our experience indicates that there is often a small quality gain at the price of a highly inefficient decompilation.

7.5. Challenge III: An Optimal Decompilation Scheme

As we already mentioned in Section 7.2, and as we can see by looking at Figure 7.9, the decompiled programs we obtain using the modular scheme

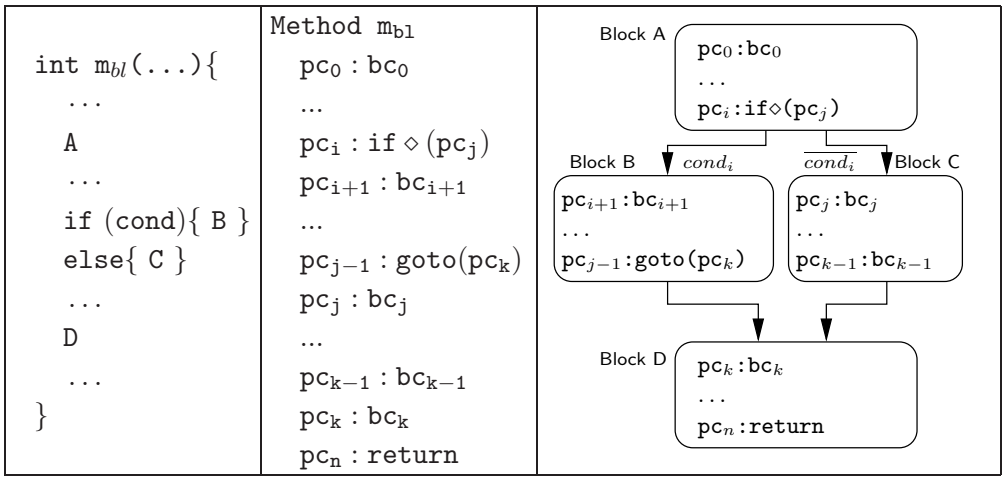


Figura 7.10: Source code, bytecode and CFG of m_{bl} method

are still not optimal as they can contain code duplications. See for example the code on the right-hand side of the rules defining `main(gcd,...)` and `execute_1/3`. Duplications are (very often) produced because part of the code is re-evaluated during PE. Unfortunately, as we will see later, such duplications and re-evaluations grow exponentially with the number of branching and merging points respectively, and as our experiments show, highly degrade the effiency of the process and the quality of the decompiled code. The main issue is whether it is possible to obtain, by means of interpretive decompilation, programs whose quality is equivalent to that obtained by dedicated decompilers; issue *(c)* in Figure 7.1. In order to obtain comparable results, it makes sense to use similar heuristics. Since decompilers first build a *control flow graph* (CFG) for the method, which guides the decompilation process, we now study how a similar notion can be used for controlling PE of the interpreter.

Let us explain the problem by means of an example. Consider the method m_{bl} in Fig. 7.10. The source code is shown to the left, the relevant bytecode in the center and its CFG to the right. As customary, the CFG [1] consists of basic blocks which contain a sequence of non-branching bytecode instructions and which are connected by edges which describe the possible flows originated from the branching instructions (like conditional jumps, exceptions, virtual method invocation, etc.). In our small bytecode programs, they correspond with conditional jumps (i.e. `if`$\diamond$ and `if0`$\diamond$). A

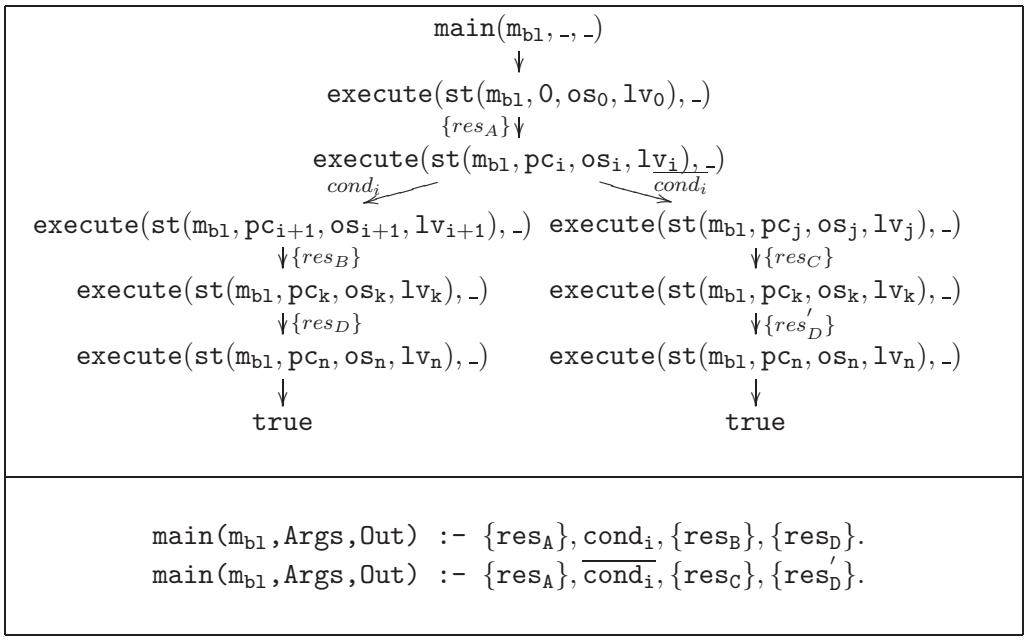


Figura 7.11: Unfolding SLD-tree and decompiled code of m_{bl} method

divergence point (D point) is a program point (bytecode index) from which more than one branch originates; likewise, a *convergence point* (C point) is a program point where two or more branches merge. In the CFG of m_{bl}, the only divergence (resp. convergence) point is pc_i (resp. pc_k).

By using the decompilation scheme presented so far, we obtain the SLD-tree shown in Fig. 7.11, in which all calls are completely unfolded as there is no termination risk. The decompiled code is shown under the tree. We use $\{\mathtt{res_X}\}$ to refer to the residual code emitted for BlockX and $\mathtt{cond_i}$ to refer to the condition associated to the branching instruction at $\mathtt{pc_i}$ ($\overline{\mathtt{cond_i}}$ denotes its negation). The quality of the decompiled code is not optimal due to:

D. Decompiled code $\{\mathtt{res_A}\}$ for BlockA is duplicated in both rules. During PE, this code is evaluated once but, due to the way resultants are defined (see Section 7.1), each rule contains the decompiled code associated to the whole branch of the tree. This code duplication brings in two problems: it increases considerably the size of decompiled programs and also makes their execution slower. For instance, when $\overline{\mathtt{cond_i}}$ holds, the execution goes unnecessarily through $\{\mathtt{res_A}\}$

in the first rule, fails to prove $\texttt{cond}_\texttt{i}$ and, then, attempts the second rule.

C. Decompiled code of BlockD is again emitted more than once. Each rule for the decompiled code contains a (possibly different) version, $\{\texttt{res}_\texttt{D}\}$ and $\{\texttt{res}'_\texttt{D}\}$, of the code of BlockD. Unlike above, at PE time, the code of BlockD is actually evaluated in the context of $\{\texttt{cond}_\texttt{i}, \{\texttt{res}_\texttt{B}\}\}$ and then re-evaluated in the context of $\{\overline{\texttt{cond}_\texttt{i}}, \{\texttt{res}_\texttt{C}\}\}$. Convergence points thus might degrade both efficiency (and endanger scalability) and quality of decompilation (due to larger residual code).

The amount of repeated residual code grows exponentially with the number of C and D points and the amount of re-evaluated code grows exponentially with the number of C points. Thus, we now aim at designing an *optimal*, *block-level* decompilation that helps overcome problems D and C above. Intuitively, a block-level decompilation must produce a residual rule for each block in the CFG. This can be achieved by building SLD-trees which correspond to each single block, rather than expanding them further. Note that this idea is against the typical spirit of PE which, in order to maximize the propagation of static information, tries to build SLD-trees as large as possible and only stops unfolding when there is termination risk.

This can be easily done in our setting by providing annotations that force the unfolding process to stop when an `execute/2` atom whose *PC* corresponds to a D point appears in the sequence. In the example, unfolding should stop at pc_i. Regarding C points, an additional requirement is to partially evaluate the code on blocks starting at these points at most once. The problem is similar to the polyvariant vs. monovariant treatment in the decompilation of methods in the previous section, by viewing entries to blocks as method calls. Not surprisingly, the solution can be achieved similarly in our setting by: (1) stopping the derivation at `execute/2` calls whose *PC* corresponds to C points and (2) passing the call to the global control, and ensuring that it is evaluated in a sufficiently generalized context which covers all incoming contexts. The former point is ensured by the use of the corresponding annotations and the latter by including in the initial set of atoms a generalized call of the form $\texttt{execute}(\texttt{st}(\texttt{m}_\texttt{bl}, \texttt{pc}_\texttt{k}, \_, \_), \_)$ for all C points, which forces such generalization.

```
main(m_bl,Args,Out) :- {res_A}, execute_1(...).
execute_1(...) :- cond_i, {res_B}, execute_2(...).
execute_1(...) :- ¬cond_i, {res_C}, execute_2(...).
execute_2(...) :- {res_D}.
```

Figura 7.12: Optimal decompiled code for m_{bl} method

An important point is that, unlike annotations used in offline PE [63] which are generated by only taking the interpreter into account, our annotations for the optimal decompilation are generated by taking into account the particular program to be decompiled. Importantly, both the annotations and the initial set of calls can be computed automatically by performing two passes on the bytecode (see, e.g., [2, 84]).

The result of performing an optimal decompilation on m_{bl} is shown in Figure 7.12. Now, the residual code associated to each block appears once in the code. This ensures that the optimal decompilation preserves the CFG shape as dedicated decompilers do. Thus, the quality of our decompiled code is as good as that obtained by state-of-the-art decompilers [2, 70] but with the advantages of interpretive decompilation.

Paper 6 studies in detail these issues and defines a block-level, optimal decompilation scheme overcoming the problems above. It is also formally proved that the proposed scheme satisfies the *block-optimality* criterion, which ensures that: (I) residual code for each bytecode instruction in the program is emitted once in the decompiled program, (II) each bytecode instruction is evaluated at most once during PE, and (III) there is at most one residual rule for each block in the bytecode program.

7.5.1. Conclusions of Optimal Decompilation

After taking into account the central observation from Section 7.4 that the interpreter should be written in big-step semantics, each condition of the block-optimality criterion above is simpler or more complicated to achieve depending on the local control strategy we use. For example, if we start from a modular decompiler as discussed in Section 7.4 above, condition (III) will in general be satisfied, but not condition (I) nor (II) since the

local control rule tends to over-specialize calls which results in re-evaluating expressions and emitting code multiple times.

Conversely, if we use an offline partial evaluator, the natural local control is to residualize all calls to `execute` and, then, filter out all information other than the method signature and program counter when transferring the atom to the global control. This control strategy trivially guarantees conditions (I) and (II) of the block-optimality criterion since it guarantees that each bytecode instruction is decompiled independently of the others. However, it tends to under-specialize and namely it does not satisfy the condition (III): as soon as there is a block with more than one bytecode instruction, which is almost always the case, the specialized program will contain a separate rule for each and every bytecode instruction in the block. As a result, the residual program thus obtained is high-level in the sense that it is written in Prolog. However, its control strategy is heavily influenced by the fact that we decompile bytecode (instead of converting, e.g. from Java source) and the decompiled program is not at all similar to the Prolog program which a Prolog programmer would write for performing the same task. Since an important objective of decompilation is to enable program understanding and analysis, we argue that programs which satisfy this optimality criterion, in particular meeting condition (III), like the ones we generate, are easier to reason about.

Another important observation is that the costly mechanisms, namely the TbHEm and the advanced polyvariance control from Paper 2, used for controlling the PE that were used earlier to achieve the results in Sections 7.3.3 and 7.4, are not needed anymore using the optimal decompilation scheme. Instead, the following trivial control operators can be used: unfold unfolds all calls except those matching an annotation, and abstract adds to the set S_{i+1} every call in L^{pe} which is not an instance of any call in S_i (see the generic algorithm in Section 7.1). It can be easily proved that termination is ensured both at the local and at the global control level thanks to the annotations and the initial set of atoms provided to the PE.

7.6. Implementation and Experimental Results

Paper 6 discusses several implementation details and performs a thorough experimental evaluation of the different decompilation schemes proposed in this chapter. We have reported on two different implementations of a decompiler for full (sequential) Java Bytecode into Prolog. For the first one we have extended an already existing powerful online PE, the one integrated in the CiaoPP system. This partial evaluator implements several unfolding rules and abstraction operators. This has allowed us to compare the different decompilation schemes, in particular, to compare against the non-optimal ones. However, the overhead introduced by using such generic and powerful tool prevents us from competing with ad-hoc decompilers as regards efficiency (decompilation times). For this reason, we have carried out a second implementation for which we have written a stand-alone PE which only contains the local and global strategies required by an optimal decompilation. This partial evaluator is integrated into a decompilation tool called jbc2prolog which also includes a Java Bytecode interpreter. This makes it possible to both obtain optimal decompilations and be competitive in terms of efficiency with ad-hoc decompilers. Paper 6 performs a thorough comparison against the decompiler in the COSTA [5] system and against the JDec [14] decompiler.

Both implementations consider full sequential Java Bytecode. The extensions needed to handle the features not considered in this introduction are further discussed in Paper 6. These include exceptions, heap operations, virtual invocations, decompilation at the level of classes, etc. It is important to note that all of them have been easily accommodated in our decompilation scheme, most of the times, simply by providing the corresponding support within the bytecode interpreter.

For the experimental evaluation in Paper 6, we have used the standardized set of benchmarks in the JOlden suite [55]. In particular, we are interested in: a) empirically demonstrating the scalability of the approach, and b) assessing the efficiency of the implemented tool by comparing it against other decompilers. We conclude the following:

- **Scalability:** While in the non-optimal decompilation both the de-

compilation times and the decompiled program sizes greatly increase with the size of the benchmarks, this does not happen in the optimal scheme. In the optimal decompilation, these figures are totally stable. We show that both the decompilation times and the decompiled program sizes are *linear* with the size of the input bytecode program, thus demonstrating the scalability of our optimal decompilation.

- **Efficiency:** To assess the efficiency of our approach we have compared the decompilation times we get using our tool jbc2prolog w.r.t. those obtained using the decompiler in the COSTA system and those obtained using the well-known Java decompiler JDec [14]. It can be concluded that our results our competitive with those of an ad-hoc decompiler. In particular, we see that they are similar to those obtained in COSTA. Furthermore, in most examples, jbc2prolog is slightly more efficient. On the other hand we can see that jbc2prolog is about ten times faster than JDec. Our conclusion in this regard is that it is very difficult to compare with decompilers written in other programming languages, since the performance of the implementation language heavily influences the decompilation time.

7.7. Related Work on Interpretive Decompilation

Previous work in *interpretative* (de)compilation has mainly focused on proving that the approach is feasible for small interpreters and medium-sized programs. The focus has been on demonstrating its *effectiveness*, i.e., that the so-called interpretation layer can be removed from the compiled programs. To achieve effectiveness, offline [63], online [48, 77] and hybrid [64] PE techniques have been assessed. This thesis has firstly focused on demonstrating that interpretive decompilation is feasible (as shown in previous work) and has studied further issues which had not been explored yet. Let us review now related work both in the field of decompilation of low-level code. Related work on on the PE of interpreters has been already compared in the introduction of this chapter and in several places throughout the paper.

The work by Breuer and Bowen [19] is only tangentially related to

ours. They propose a general method for compiling decompilers from the specifications of (non-optimizing) compilers. The main idea is that a data type specification for a programming-language grammar can be remolded into a functional program that enumerates all of the abstract syntax trees of the grammar. It is showed that by relying on this technique a decompiler can be generated from a simple Occam-like compiler specification. The only similarity with our work is that decompiled programs are somehow obtained from specifications (in our case of the interpreter and in their case of the compiler). However, the underlying methods are technically different and also they do not provide a practical solution for ensuring applicable conditions for their technique.

As regards (direct) decompilation of low-level back to source code, it has been the subject of a good amount of research. Decompilation can be attempted at different levels, with different levels of success. The most complicated case is when decompiling binary executables. There are a good number of associated complications, such as recovering the control flow. One intrinsic problem in this approach is that it is not possible in general to distinguish code from data statically. See e.g. [26, 81] and their references for a discussion on the problems and techniques for binary decompilation. The next level is decompilation of assembly, see e.g. [27]. This shares many of the complications associated to the decompilation of binaries, since current hardware architectures are rather complex, but at least it is possible to separate code from data. The following level is decompilation of code to be run on a virtual machine. This is in general easier to perform since virtual machines are usually simpler than current hardware architectures and because often the code for this virtual machines (bytecode) must satisfy certain behavior restrictions (must be *verifiable* [60]) and types of variables are available. As a result, in the particular case of decompilation of Java Bytecode back to Java source, a number of successful commercial and free software decompilers exist which are able to handle a large class of bytecode programs, especially those generated by common Java compilers, i.e., `javac`. Nevertheless, things become more complicated when the Java Bytecode has been generated by an obfuscator, and especially when an optimizing compiler, or a compiler from other programming languages such as Haskell, Eiffel, ML, Ada, and Fortran is used. See e.g. [72] and its references for a good account on the existing Java Bytecode decompilers

and the difficulties associated to its decompilation.

As already mentioned, there exist several analyzers for Java Bytecode which use a higher-level intermediate representation and which can be seen as ad-hoc decompilers. In particular, both the COSTA [5] and `CiaoPP` [49] systems have a front-end which converts bytecode into an intermediate representation which is then the input to the subsequent analysis. Though in both cases the intermediate representation is similar, in the case of COSTA it is formalized as a rule-based representation [2], whereas in `CiaoPP` it is formalized as Horn clauses, i.e., a logic program [70]. The reason for doing that in `CiaoPP` is that, at least in principle, that allows using the analysis which are already available in `CiaoPP`. However, there is a crucial difference between the logic programs generated in [70] and those generated by our decompiler. Whereas the programs generated by [70] are only meant to be the subject of static analysis and are not executable, the programs we generate can both be subject to analysis or be executed. The reason why the programs in [70] nor those in [2] are executable is because they basically capture the control-flow of the bytecode program, but the basic bytecode instructions themselves remain as *builtins*, i.e., predefined predicates, to the analysis. Analysis results are correct as long as the behavior of such bytecode instructions is safely approximated by the analysis. Producing fully executable logic programs as the result of decompilation is not trivial since many of the bytecode instructions operate on the heap in a way or another. Thus, in order to make an executable decompiled program we need to introduce the JVM heap explicitly in the logic program. All this is done automatically in our approach.

Chapter 8

Applications of Interpretive Decompilation

As already mentioned in the previous chapter, an important advantage is that the decompiled programs obtained by interpretive decompilation are fully *executable*, which in turn broadens their application field. In this chapter, we summarize two different experimentations we have performed which take advantage of such feature of our decompiled programs:

1. Analysis of bytecode programs by analyzing its decompilations to LP using LP analysis tools. This is further elaborated in Paper 1.

2. *Test data generation* of bytecode programs by CLP partial evaluation. This issue is studied in detail in Paper 7.

8.1. Analysis of Bytecode using LP Analysis Tools

Analyzing programs in the CLP paradigm offers a good number of advantages, an important one being the maturity and sophistication of the analysis tools available for it. In particular, the `CiaoPP` system, besides providing a very powerful partial evaluator which we have used to perform part of our experimentation in the previous chapter, also provides a generic analysis engine with a good number of abstract domains available. This

allows inferring a good number of properties of logic programs like *termination*, bounds on resource consumption, *types* and *modes*, *error-freeness*, etc.

One of the objectives of this thesis has been to investigate whether it is feasible to reuse existing analysis tools already available in the CLP paradigm, in particular `CiaoPP`, to analyze bytecode programs by analyzing their decompilations to LP. This allows devising a generic framework for the analysis and verification of bytecode programs in which the power of the analysis tools for CLP is automatically transferred to the analysis and verification of bytecode programs. The same idea had been applied to analyze rather restricted versions of high-level imperative languages [77] and also assembly code for PIC [47], an 8-bit microprocessor. However, to the best of our knowledge, this is the first time this approach has been successfully applied to a general purpose, realistic, imperative programming language.

These issues are further elaborated in Paper 1, where: 1) we propose such a framework for the analysis and verification of bytecode programs (in particular for Java Bytecode), and 2) we perform a series of experiments using the `CiaoPP` system demonstrating the feasibility of the proposed approach. In summary, Paper 1 shows how, by reasoning on our decompiled programs, we can automatically prove, by relying on the analyses available in the `CiaoPP` system, some non-trivial properties of bytecode programs such as termination, run-time *error-freeness* and infer bounds on its resource consumption. For instance, in order to prove run-time *error-freeness*, we propose an enhanced bytecode interpreter which computes, in addition to the return value of the method called, also the trace which captures the computation history. Such traces represent the semantic steps used, and therefore do not only represent instructions, as the context has also some importance. They have allowed us to distinguish, for example, for a same instruction, the step that throws an exception from the normal behavior. E.g., `invoke_step_ok` and `invoke_step_NullPointerException` represent, respectively, a normal method call and a method call on a null reference that throws an exception. Such additional flexibility of interpretive decompilation has allowed to prove run-time *error-freeness* in a straightforward way by simply specifying the property of being *error-free* as verifying that the corresponding trace in the decompiled program does not contain an exceptional step, or that it does not end raising an exception, depending on

the particular policy for the *error-freeness* property. Again, our approach demonstrates its flexibility here, as different policies can be easily defined simply by specifying the corresponding property in `CiaoPP`.

8.2. Test Data Generation by CLP PE

A unique feature of our decompiled programs is that they represent the whole program state, in contrast to [70, 2, 84]. In particular, they contain a representation of the heap explicitly in addition to the operand stack. Up to now, the main motivation for decompiling bytecode to LP had been to be able to perform static analysis on the decompiled programs in order to infer properties about the original bytecode. If the decompilation approach produces LP programs which are executable, then such decompiled programs can be used not only for static analysis, but also for dynamic analysis and execution. Note that this is not always the case, since there are approaches (like [4, 70]) which are aimed at producing static analysis targets only and their decompiled programs cannot be executed. A novel interesting application of interpretive decompilation which we propose in this thesis is the automatic generation of *test data*.

Test data generation (TDG) aims at automatically generating test-cases for interesting test *coverage criteria*. The coverage criteria measure how well the program is exercised by a test suite. Examples of coverage criteria are: *statement coverage* which requires that each line of the code is executed; *path coverage* which requires that every possible trace through a given part of the code is executed; etc. There are a wide variety of approaches to TDG (see [90] for a survey). Our work focuses on *glass-box* testing, where test-cases are obtained from the concrete program in contrast to *black-box* testing, where they are deduced from a specification of the program. Also, our focus is on *static* testing, where we assume no knowledge about the input data, in contrast to *dynamic* approaches [39, 46] which execute the program to be tested for concrete input values.

The standard approach to generating test-cases statically is to perform a *symbolic* execution of the program (see e.g. [29]), where the contents of variables are expressions rather than concrete values. The symbolic execution produces a system of *constraints* consisting of the conditions to execute the different paths. This happens, for instance, in branching ins-

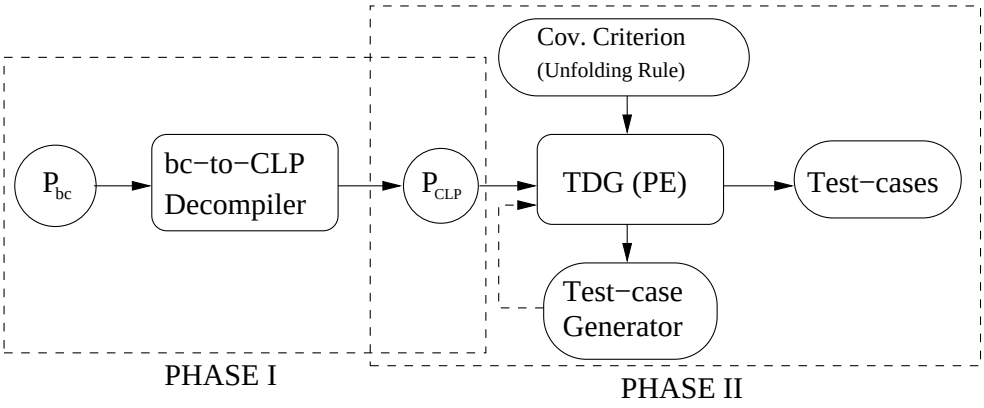


Figura 8.1: Overview of our approach for TDG of bytecode by CLP PE

tructions, like if-then-else, where we might want to generate test-cases for the two alternative branches and hence accumulate the conditions for each path as constraints. For the particular case of Java Bytecode, a symbolic JVM (SJVM) which integrates several constraint solvers has been designed in [73]. A SJVM requires non-trivial extensions w.r.t. a JVM: (1) it needs to execute the bytecode symbolically as explained above, (2) it must be able to backtrack, as without knowledge about the input data, the execution engine might need to execute more than one path. The backtracking mechanism used in [73] is essentially the same as in logic programming.

In this thesis we propose a novel approach to TDG of bytecode which is based on PE techniques developed for CLP and which, in contrast to previous work, does not require devising a dedicated symbolic virtual machine. Figure 8.1 depicts a diagram with an overview of the framework. As can be seen, it comprises two independent, CLP PE phases, which basically consist in the following:

1. *The decompilation of bytecode into a CLP program.* We already discussed in Chapter 7 that the decompilation of bytecode to LP can be achieved automatically by means of partial evaluation of LP, or alternatively by means of an ad-hoc decompiler [70]. The modification to obtain CLP instead of LP programs is straightforward, e.g. by means of a trivial transformation of the arithmetic builtins into their CLP counterparts.

2. *The generation of test-cases.* This is a novel application of PE which

allows generating test-case generators from the CLP decompiled bytecode. In this case, we rely on a CLP partial evaluator which is able to solve the constraint system, in much the same way as a symbolic abstract machine would do. The two control operators of a CLP partial evaluator play an essential role: (1) The local control applied to the decompiled code will allow capturing interesting coverage criteria for TDG of the bytecode. (2) The global control will enable the generation of *test-case generators*. Intuitively, the test-case generators we produce are CLP programs whose execution in CLP returns further test-cases on demand without the need to start the TDG process from scratch.

We argue that our CLP PE based approach to TDG of bytecode has several advantages w.r.t. existing approaches based on symbolic execution: (i) It is more *generic*, as the same techniques can be applied to other both low and high-level imperative languages. In particular, once the CLP decompilation is done, the language features are abstracted away and, the whole part related to the generation of test data is totally *language independent*. This avoids the difficulties of dealing with recursion, procedure calls, dynamic memory, etc. that symbolic abstract machines typically face. (ii) It is more *flexible*, as different coverage criteria can be easily incorporated to our framework just by adding the appropriate local control to the partial evaluator. (iii) It is more *powerful* as we can generate test-case generators. (iv) It is *simpler* to implement compared to the development of a dedicated symbolic virtual machine, as long as a CLP partial evaluator is available.

As noted in point (iv) above, an important advantage of CLP decompiled programs w.r.t. their bytecode counterparts is that symbolic execution does not require, at least in principle, to build a dedicated symbolic execution mechanism. Instead, we can simply run the decompiled program by using the standard CLP execution mechanism with all arguments being distinct free variables. E.g., for our working example of Figure 7.4, we could perform symbolic execution of the `gcd` method by running the query `main(gcd,[X,Y],Z)` on the decompiled program. Note that as we are not providing input values, each successful execution corresponds to a different computation path in the bytecode. Furthermore, along the execution, a constraint store on the program's variables is obtained which can be used for inferring the conditions that the input values (in our case `X` and `Y`) must

satisfy for the execution to follow the corresponding computation path.

However, an important problem with symbolic execution, regardless of whether it is performed using CLP or a dedicated execution engine, is that the execution tree to be traversed is in most cases infinite, since programs usually contain iterative constructs such as loops and recursion which induce an infinite number of execution paths when executed without input values. Therefore, it is essential to establish a *termination criterion* (in this context coverage criterion) which guarantees that the number of paths traversed remains finite, while at the same time an interesting set of test data is generated.

Another issue is that depending on the particular type of decompilation –and even on the options used within a particular method– we can obtain different correct decompilations which are valid for the purpose of execution. However, for the purpose of generating useful test-cases, additional requirements are needed: we must be able to define coverage criteria on the CLP decompilation which produce test-cases which cover the *equivalent* coverage criteria for the bytecode. Fortunately, our notion of *block-level* decompilation, introduced in 7.5, provides a sufficient condition for ensuring that equivalent coverage criteria can be defined. According to this definition, there is a one to one correspondence between blocks in the CFG of the bytecode program and rules in the decompiled one.

Most existing coverage criteria are defined on high-level, structured programming languages. A widely used control-flow based coverage criterion is loop-count(k), which dates back to 1977 [53], and limits the number of times we iterate on loops to a threshold k. However, bytecode has an unstructured control flow: CFGs can contain multiple different shapes, some of which do not correspond to any of the loops available in high-level, structured programming languages.

In this thesis, we introduce the *block-count*(k) coverage criterion which is not explicitly based on limiting the number of times we iterate on loops, but rather on counting how many times we visit each block in the CFG within each computation. Basically, a set of computation paths satisfies the *block-count*(k) *criterion* if the set includes all finished computation paths which can be built such that the number of times each block is visited within each computation does not exceed a given k.

Paper 7 discusses the technical details of such an approach to TDG of

bytecode. In particular:

- The *block-count*(k) coverage criterion is formally defined.
- We define an *evaluation strategy* which guarantees constructing an SLD tree so that we generate sufficiently many derivations so as to satisfy the block-count(k) criterion while, at the same time, guaranteeing termination.
- The TDG phase is formalized as a CLP PE of the CLP decompiled program where the unfolding rule plays the role of the coverage criterion. We hence provide an unfolding rule which implements the block-count(k) coverage criterion and outline how the abstraction operator must deal with constraints so that we get effective test-case generators.
- All such issues are illustrated through a working example which comprises a set of methods performing different arithmetic computations.

8.2.1. On the Generation of Test Data for Prolog by EP

As a tangential contribution of the thesis, we have applied the idea of using PE to automatically generate test data in the context of LP. We argue that our approach to TDG can in principle be directly applied to any imperative language. However, when one tries to apply it to a declarative language like Prolog, we have found as a main difficulty the generation of test-cases which cover the more complex control flow of Prolog. Essentially, the problem is that an intrinsic feature of PE is that it only computes non-failing derivations while in TDG for Prolog it is essential to generate test-cases associated to failing computations. Paper 8 performs a preliminary study in this direction. Basically, it proposes to transform the original Prolog program into an equivalent Prolog program with *explicit failure* by partially evaluating a Prolog interpreter which captures failing derivations w.r.t. the input program. Another issue that we have discussed in the paper is that, while in the case of bytecode the underlying constraint domain only manipulates integers, in Prolog it should properly handle the symbolic data

manipulated by the program. Our preliminary experiments already suggest that the approach can be very useful to generate test-cases for Prolog.

8.2.2. Related work on Test Data Generation

As mentioned before, our approach is focused on static TDG, in which test-cases are obtained without running the program with particular input values. In contrast, *dynamic* approaches [39, 46] execute the program to be tested for concrete input values until achieving the particular coverage of the program. The standard approach to generating test-cases statically is to perform a *symbolic* execution of the program [29, 71, 73, 57, 45]. The symbolic execution approach has been combined with the use of *constraint solvers* [73, 45] in order to: handle the constraint systems by solving the feasibility of paths and, afterwards, to instantiate the input variables. For the particular case of Java Bytecode, a symbolic JVM machine (SJVM) which integrates several constraint solvers has been designed in [73].

TDG for declarative languages has received comparatively less attention than for imperative languages. The majority of existing tools for functional programs are based on black-box testing (see e.g. [28]). An exception is [40] where a glass-box testing approach is proposed to generate test-cases for Curry. In the case of CLP, test-cases are obtained for Prolog in [69, 13, 89]; and very recently for Mercury in [36]. Basically the test-cases are obtained by first computing constraints on the input arguments that correspond to execution paths of logic programs and then solving these constraints to obtain test inputs for such paths. For functional logic languages, specific coverage criteria are defined in [40] which capture the control flow of these languages as well as new language features are considered, namely laziness.

In general, declarative languages pose different problems to testing related to their own execution models –like laziness in functional languages and failing derivations in (C)LP– which need to be captured by appropriate coverage criteria. Having said this, we believe our ideas related to the use of PE techniques to generate test data generators and the use of unfolding rules to supervise the evaluation could be adapted to declarative programs as our preliminary experiments in Paper 8 show.

Chapter 9

Heap Space Analysis of Bytecode Programs

Predicting the memory required to run a program is crucial in many contexts like in embedded applications with stringent space requirements or in real-time systems which must respond to events or signals as fast as possible. It is widely recognized also that memory usage estimation is important for an accurate prediction of running time, as cache misses and page faults contribute directly to the runtime.

Heap space analysis aims at inferring *bounds* on the heap space consumption of programs. Heap analysis is more typically formulated at the source level (see, e.g., [83, 50, 85, 54] in the context of functional programming and [52, 23] for high-level imperative programming languages). As mentioned in Chapter 6.4, there are however situations where one has only access to the compiled code and not to the source code. Automatic heap space analysis has interesting applications in this context. For instance, *resource bound certification* [33, 10, 51, 22] proposes the use of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on the resource consumption. Also, heap bounds are useful on embedded systems, e.g., smart cards in which memory is limited and cannot easily be recovered.

A general framework for the cost analysis of sequential Java Bytecode has been proposed in [3] which led to the COSTA system [5]. Such analysis statically generates *cost relations* (*CRs*) which define the cost of a program as a function of its input data size. The *CRs* are expressed by means of

recursive equations generated by abstracting the recursive structure of the program and by inferring size relations between arguments. The analysis is parametric w.r.t. a *cost model* which defines the cost unit associated to each bytecode.

This thesis develops a novel application of the cost analysis framework of [3] to infer bounds on the heap space consumption of sequential Java Bytecode programs:

1. In a first step, we develop a cost model that defines the cost of memory allocation instructions (e.g., `new` and `newarray`) in terms of the number of heap (memory) units they consume. E.g., the cost of creating a new object is the number of heap units allocated to that object. The remaining bytecode instructions do not add any cost. With this cost model, we generate heap space *CRs* which are then used to infer upper bounds on the heap space usage of the different methods. These upper bounds provide information on the maximal heap space required for executing each method in the program.

2. Unfortunately, in the case of languages with automatic memory management (*garbage collection*), the above approach, though still correct, can produce too pessimistic estimations. Therefore, in a second step, we refine the analysis to consider the effect of garbage collection. We propose a *live heap space analysis*, which aims at approximating the maximum of the live heap usage along the execution of a program, thus providing a much tighter estimation in presence of garbage collection. This is done by relying on escape analysis [17] to identify those memory allocation instructions which create objects that will be garbage collected upon exit from the corresponding method. With this information available, we can infer upper bounds on the *escaped memory* of method's execution, i.e., the memory that is allocated during the execution of the method *and* which remains upon exit. We then propose a novel form of *peak consumption CRs* which capture the peak memory consumption over all program states along its execution. An essential feature of our *CRs* is that they can be solved by using existing tools for solving standard *CRs*.

These issues are respectively introduced and summarized in Sections 9.1 and 9.2 and studied in detail in Papers 9 and 10.

A distinguishing feature of the analyses presented in this chapter w.r.t. previous approaches (e.g., [10, 50, 18, 24]) is that they are not restricted to linear bounds since the generated *CRs* can in principle capture any complexity class. Moreover, in many cases, using the upper bound solver of the COSTA system, the relations can be simplified to a *closed form* solution from which one can glean immediate information about the expected consumption of the code to be run.

It is important to note that the analysis could have been developed on the decompiled LP programs in a similar way. In fact, COSTA performs a decompilation of the bytecode into a rule-based representation before the actual analysis phase with the aim of making the analysis design simpler (see [3] for details). The IRs of COSTA are actually very similar to our LP decompiled programs with the main difference that in the COSTA IRs, all bytecode instructions remain residual and have to be taken as builtins, i.e., predefined procedures. In contrast, in our decompilations, bytecode instructions are interpreted at decompilation time and converted into basic Prolog instructions such as unifications and arithmetic operations. The reason why we have not used our interpretive decompilations for the analysis is that this way we have been able to integrate our analysis in COSTA and thus take advantage of all the machinery for the cost analysis which is included in it, like e.g., the *size analysis* for inferring the size relations among arguments, the upper bound solver, etc.

9.1. Total Heap Space Analysis of Bytecode

Let us consider the Java program depicted in Figure 9.1. It consists of a set of Java classes which define a linked-list data structure in an object-oriented style. The class `Cons` is used for data nodes (in this case integer numbers) and the class `Nil` plays the role of *null* to indicate the end of a list. Both `Cons` and `Nil` extend the abstract class `List`. Thus, a `List` object can be either a `Cons` or a `Nil` instance. Both subclasses implement a `copy` method which is used to clone the corresponding object. In the case of `Nil`, `copy` just returns a new instance of itself since it is the last element of the list. In the case of `Cons`, it returns a cloned instance where the data is cloned by calling the static method `m`, and the continuation is cloned by calling recursively the `copy` method on `next`.

```
abstract class List {
  abstract List copy();
}

class Nil extends List {
  List copy() {
    return new Nil();
  }
}

class Cons extends List {
  int elem;
  List next;

  List copy(){
    Cons aux = new Cons();
    aux.elem = m(this.elem);
    aux.next = this.next.copy();
    return aux;
  }

  static int m(int n) {
    Integer aux = new Integer(n);
    return aux.intValue();
  }
} // class Cons
```

Figura 9.1: Example for memory consumption

Our heap space analysis infers the following simplified *CRs* for the `copy` method of class `Cons`:

$$C_{copy}(a) = 12, \quad a = 1$$
$$C_{copy}(a) = 12 + C_{copy}(a-1), \quad a > 1$$

which can then be solved using the upper-bound solver of COSTA yielding the following upper-bound in closed-form:

$$C_{copy}(a) = 12*nat(a-1) + 12$$

It can be observed that the heap consumption is linear w.r.t. the input parameter `a`, which corresponds to the size of the *this* object of the method, i.e., the length of the list which is being cloned. This is because the abstraction being used by our analysis for object references is the *length of the longest reference chain*, which in this case corresponds to the length of the list. The numeric constant 12 is obtained by adding 8 and 4, being 8 the bytes taken by an instance of class `Cons`, and 4 the bytes taken by an `Integer` instance. Note that we are approximating the size of an object by the sum of the sizes of all of its fields. In particular, both an integer and a reference are assumed to consume 4 bytes.

The analysis has been integrated in the COSTA system. Paper 9 performs an experimental evaluation by means of a series of example applications written in an object-oriented style which make intensive use of the

heap and which present novel features like heap consumption that depends on the class fields, multiple inheritance, virtual invocation, etc. These examples allow us to illustrate the most salient features of our analysis: inference of constant heap usage, heap usage proportional to input size, support of standard data-structures like lists, trees, arrays, etc. To the best of our knowledge, this is the first analysis able to infer arbitrary heap usage bounds for Java Bytecode.

9.2. Live Heap Space Analysis for Languages with GC

As mentioned earlier, garbage collection (GC) makes the problem of predicting the memory required to run a program difficult. A first approximation is to infer the total memory allocation, i.e., the *accumulated* amount of memory allocated by a program ignoring GC, as done in the previous section. If such amount is available it is ensured that the program can be executed without exhausting the memory, even if no GC is performed during its execution. However, it is an overly pessimistic estimation of the actual memory requirement.

This thesis presents a general approach for inferring the *peak heap consumption* of a program's execution, i.e., the maximum of the live heap usage along its execution. Our live heap space analysis is developed for (an intermediate representation of) an object-oriented *bytecode* language with automatic memory management.

Analysis of live heap usage is different from total memory allocation because it involves reasoning on the memory consumed at *all program states* along an execution, while total allocation needs to observe the consumption at the *final* state only. As a consequence, the classical approach to static cost analysis proposed by Wegbreit in 1975 [86] has been applied only to infer total allocation. Intuitively, given a program, this approach produces a *CR* system which is a set of recursive equations that capture the cost *accumulated* along the program's execution. Symbolic closed-form solutions (i.e., without recursion) are found then from the *CR*. This approach leads to very accurate cost bounds as it is not limited to any complexity class (infers polynomial, logarithmic, exponential consumption, etc.) and, besides, it can

be used to infer different notions of resources (total memory allocation, number of executed instructions, number of calls to specific methods, etc.). Unfortunately, as argued in Paper 9, it is not suitable to infer peak heap consumption because it is not an accumulative resource of a program's execution as *CR* capture. Instead, it requires to reason on all possible states to obtain their maximum. By relying on different techniques which do not generate *CR*, live heap space analysis is currently restricted to polynomial bounds and non-recursive methods [18] or to linear bounds dealing with recursion [24].

Inspired by the basic techniques used in cost analysis, in this thesis, we present a general framework to infer accurate bounds on the peak heap consumption of programs which improves the state-of-the-art in that it is not restricted to any complexity class and deals with all bytecode language features including recursion. To pursue our analysis, we need to characterize the behavior of the underlying garbage collector. We assume a standard *scoped-memory* manager that reclaims memory when methods return. In this setting, our main contributions are:

1. *Escaped Memory Analysis.* We first develop an analysis to infer upper bounds on the *escaped memory* of method's execution, i.e., the memory that it is allocated during the execution of the method *and* which remains upon exit. The key idea is to infer first an upper bound for the total memory allocation of the method, as done in Section 9.1. Then, such bound can be manipulated, by relying on information computed by *escape analysis* [17], to extract from it an upper bound on its escaped memory.

2. *Live Heap Space Analysis.* By relying on the upper bounds on the escaped memory, as our main contribution, we propose a novel form of *peak consumption CR* which captures the peak memory consumption over all program states along the execution for the considered scoped-memory manager. An essential feature of our *CRs* is that they can be solved by using existing tools for solving standard *CRs*.

3. *Ideal Garbage Collection.* An interesting, novel feature of our approach is that we can refine the analysis to accommodate other kinds of scope-based managers which are closer to an *ideal* garbage collector which collects objects as soon as they become unreachable.

4. *Implementation.* We report on a prototype implementation which is integrated in COSTA and experimentally evaluate it on the JOlden benchmark suite. Preliminary results demonstrate that our system obtains reasonably accurate live heap space upper bounds in a fully automatic way.

Let us consider again the example of the previous section. Our live heap analysis now infers the following simplified *CRs* for the `copy` method of class `Cons`:

$$C_{copy}(a) = 12, \quad a = 1$$
$$C_{copy}(a) = 8 + \max(4, C_{copy}(a-1)), \quad a > 1$$

The intuition of the second *CR* is that the peak consumption of the method when $a > 1$ is the consumption of the method (a `Cons` object) plus the maximum between the peak consumption of method `m` and the escaped memory from `m` plus the peak consumption of *copy* with the decremented argument. The *CRs* can again be solved using the upper-bound solver of COSTA yielding the following upper-bound in closed-form:

$$C_{copy}(a) = 8*nat(a-1) + 24$$

An interesting observation is that the *Integer* object which is created inside the *m* method is not reachable from outside and thus can be garbage collected. The peak heap analyzer accounts for this and therefore has deleted the size of the *Integer* object from the recursive equation, thus obtaining 8 instead of 12 multiplying $\mathtt{nat}(\mathtt{A}-1)$. It can also be observed that COSTA is not being fully precise, as the actual peak consumption of this method is $8*\mathtt{nat}(\mathtt{A}-1)+8$ (i.e. the size of the cloned list). The reason for this is that the upper bound solver has to consider the additional cases introduced by the peak heap analysis in the *max* expressions to ensure soundness, hence making the second constant increase to 24.

9.3. Related Work on Heap Space Analysis

There has been much work on analyzing program cost or resource complexities, but the majority of it is on time analysis (see, e.g., [**?**]). Analysis of live heap space is different because it involves explicit analysis of all

program states. Most of the work on memory estimation has been studied for functional languages. The work in [50] statically infers, by typing derivations and linear programming, linear expressions that depend on functional parameters while we are able to compute non-linear bounds (exponential, logarithmic, polynomial). The technique is developed for functional programs with an explicit deallocation mechanism while our technique is meant for imperative bytecode programs which are better suited for an automatic memory manager. The techniques proposed in [83, 82] consist in, given a function, constructing a new function that symbolically mimics the memory consumption of the former. Although these functions resemble our cost equations, their computed function has to be executed over a concrete valuation of parameters to obtain a memory bound for that assignment. Unlike our closed-form upper bounds, the evaluation of that function might not terminate, even if the original program does. Other differences with the work by Unnikrishnan et al. are that their analysis is developed for a functional language by relying on *reference counts* for the functional data constructed, which basically count the number of pointers to data and that they focus on particular aspects of functional languages such as tail call optimizations.

It is worth mentioning also the work in [21], where a memory consumption analysis is presented. In contrast to ours, their aim is to verify that the program executes in bounded memory by simply checking that the program does not create new objects inside loops, but they do not infer bounds as our analysis does. Moreover, it is straightforward to check that new objects are not created inside loops from our cost relations. Another related work includes research in the MRG project [10, 16], which focuses on building a proof-carrying code [75] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds. In contrast to ours, the analysis is developed for a functional language which then compiles to a (subset of) Java Bytecode and it is restricted to linear bounds. In [12] the Bytecode Specification Language is used to annotate Java Bytecode programs with memory consumption behavior and policies, and then verification tools can be used to verify those policies.

For Java-like languages, the work of [52] presents a type system for heap analysis without garbage collection, it is developed at the level of the source code and based on amortized analysis (hence it is technically quite different

to our work) and, unlike us, they do not present an inference method for heap consumption.

Related techniques have been also recently proposed to improve our first proposal of Paper 9. In particular, for an assembly language, [24] infers memory resource bounds (both stack usage and heap usage) for low-level programs (assembly). The approach is limited to linear bounds, they rely on explicit disposal commands rather than on automatic memory management. In their system, dispose commands can be automatically generated only if alias annotations are provided. For a Java-like language, the approach of [18] infers upper bounds of the peak consumption by relying on an automatic memory manager as we do. They do not deal with recursive methods and are restricted to polynomial bounds. Besides, our approach in Paper 10 is more flexible as regards its adaptation to other GC schemes. We believe that our system is the first one to infer upper bounds on the live heap consumption which are not restricted to simple complexity classes.

Chapter 10

Conclusions and Future Work

The main objective of this thesis has been to improve the state-of-the-art in the transformation and analysis of bytecode languages. Our first challenge was to provide a formal framework for the automatic decompilation of (object-oriented) bytecode programs into higher level intermediate representations using LP by means of interpretive decompilation. Compared to the development of a dedicated decompiler, interpretive decompilation has important advantages like *flexibility*, *maintainability*, *security* and *genericity*. Though very attractive, up to now it had not been widely applied in practice except for some proofs-of-concept showing the feasibility of the approach [63, 48, 76, 64]. Hence, there remained important open issues when it came to decompile realistic languages, namely, *scalability* and *effectiveness*. The thesis proposed novel solutions and finally answered them positively by presenting a modular, optimal decompilation scheme which: 1) produces decompiled programs whose quality is equivalent to that of dedicated decompilers, and 2) is demonstrated (theoretically and empirically) to scale up in practice. Our experimental results show that our decompiler is competitive, from the point of view of efficiency, with dedicated decompilers. We thus believe that the proposed techniques, together with their experimental evaluation, provide for the first time actual evidence that the interpretive theory proposed by Futamura in the 70s is indeed an appealing and feasible alternative to the development of dedicated decompilers from modern languages to intermediate representations.

For the sake of concreteness, our interpretive decompilation scheme has been formalized in the context of PE of logic programs, and implemented

for the Java Bytecode language. It is however important to note that the ideas we propose for enabling the practicality of the approach are also of interest for the interpretive (de)compilation of any pair of source and target languages.

On the other hand, the study of such a complex application of EP has led us to solve some non-trivial problems of EP in general, like the handling of infinite signatures. In this regard, we have come out with the *type-based homeomorphic embedding* relation, which has been demonstrated to improve the state-of-the-art of (online) specialization tools. We have shown that existing approaches which extend the untyped embedding relation to handle infinite signatures can be reconstructed as instances of our TbHEm relation. Though we have outlined procedures to infer the types in the context of LP, our type-based relation is not tied to any programming paradigm. Moreover, it can be used for a wide range of applications, namely in all areas of automatic program analysis, synthesis, verification, specialization and transformation; and will directly benefit from any progress on automatic type inference.

We have seen that the resulting intermediate representation, using LP, can greatly simplify the development of analysis, verification and model-checking tools for modern languages and, more interestingly, existing advanced tools developed for declarative languages (already proven correct and effective) can be directly applied on it. We have performed two experimentations in this direction. In the first one we have investigated whether it is feasible to analyze bytecode programs by analyzing its decompilations to LP using existing LP analysis tools. In this sense, we have been able to automatically prove in the `CiaoPP` system some non-trivial properties of Java Bytecode programs such as termination, run-time error freeness and infer bounds on its resource consumption for some simple programs.

For our second experimentation we have taken advantage of the fact that our decompiled programs are fully *executable* since, in contrast to other approaches [70, 2, 84], they represent the whole program state (i.e. they contain a representation of the heap in addition to the operand stack). We have therefore proposed a methodology for test data generation of bytecode by means of existing EP techniques developed for CLP. Our approach consists of two separate phases: (1) the compilation of the bytecode to a CLP program, and (2) the generation of test-cases from the CLP program.

It naturally raises the question whether this approach can be applied to other imperative languages in addition to bytecode. This is interesting as existing approaches for Java [73], and for C [45], struggle for dealing with features like recursion, method calls, dynamic memory, etc. during symbolic execution. We have shown that these features can be uniformly handled in our approach after the transformation to CLP. In particular, all kinds of loops in the bytecode become uniformly represented by recursive predicates in the CLP program. Also, we have seen that method calls are treated in the same way as calls to blocks.

We believe this experimentation is a, very promising, proof-of-concept that partial evaluation of CLP is a powerful technique for carrying out TDG in bytecode languages. To develop our ideas, we have considered a simple imperative bytecode language and left out object-oriented features which require a further study. Also, our language is restricted to integer numbers and the extension to deal with real numbers is subject of future work. We plan to carry out an experimental evaluation by transforming Java Bytecode programs from existing test suites to CLP programs and then trying to obtain useful test-cases. When considering realistic programs with object-oriented features and real numbers, we will surely face additional difficulties. One of the main practical issues is related to the scalability of our approach. An important threaten to scalability in TDG is the so-called infeasibility problem [90]. It happens in approaches that do not handle constraints along the construction of execution paths but rather perform two independent phases (1) path selection and (2) constraint solving our approach integrates both parts in a single phase, we do not expect scalability limitations in this regard. Also, a challenging problem is to obtain a decompilation which achieves a manageable representation of the heap. This will be necessary to obtain test-cases which involve data for objects stored in the heap. For the practical assessment, we also plan to extend our technique to include further coverage criteria. We want to consider other classes of coverage criteria which, for instance, generate test-cases which cover a certain statement in the program.

In principle, such an approach to TDG can be applied to any language, both to high-level and low-level. In this direction, this thesis has performed a preliminary experimentation in which we study whether the second phase can be useful for test-case generation of CLP programs, which are

not necessarily obtained from a decompilation of an imperative code. This introduces some difficulties like the handling of failing derivations and of symbolic data. In this thesis, we have sketched solutions to overcome such difficulties. In particular, we have proposed a program transformation, based on PE, to make failure explicit in the Prolog programs. To handle Prolog's negation in the transformed programs, we have outlined existing solutions that make it possible to turn the negative information into positive information. Though our preliminary experiments already suggest that the approach can be very useful to generate test-cases for Prolog, we plan to carry out a thorough practical assessment. This requires to cover additional Prolog features which have not been handled yet, and, also to compare the results with other TDG systems. We also want to study the integration of other kinds of coverage criteria like *data-flow* based criteria. Finally, we would like to explore the use of static analyses in the context of TDG. For instance, the information inferred by a *failure analysis* can be very useful to prune some of the branches that our transformed programs have to consider.

Another big challenge of this thesis has been to improve the state-of-the-art of heap space analysis of bytecode languages. In this regard, we have developed a novel application of the cost analysis framework of [3] which has been further extended to consider the effect of garbage collection. We have therefore presented a general approach to the automatic and accurate live heap space analysis for bytecode languages with garbage collection. First, we have proposed how to obtain accurate bounds on the memory *escaped* from a method's execution by combining the total allocation performed by the method together with information obtained by means of escape analysis. Then, we have introduced a novel form of *peak consumption cost relation* which uses the computed escaped memory bounds and precisely captures the actual heap consumption of programséxecution for garbage-collected languages. Such cost relations can be converted into closed-form upper bounds by relying on standard upper bound solvers, in particular the one in COSTA. For the sake of concreteness, our analysis has been developed for object-oriented bytecode, though the same techniques can be applied to other languages with garbage collection. We have first developed our analysis under a scoped-memory management which reclaims memory on method's return. The amount of memory required to run a method under

such model can be used as an over-approximation of the amount required to run it in the context of an ideal garbage collection which frees objects as soon as they become dead. We have also shown how to approximate such ideal behavior with our analysis.

Finally, it is important to note that this approach could be used to estimate other (non accumulative) resources which require to consider the maximal consumption of several execution paths. For example, it can be used to estimate the maximal height of the frames stack as follows. Given a rule $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) ::= \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$, where $b_{i_1} \ldots b_{i_k}$ are the calls in r, with $1 \leq i_1 \leq \cdots \leq i_k \leq n$ and $\mathrm{b}_{\mathrm{i_j}} = \mathrm{q}_{\mathrm{i_j}}(\langle \bar{\mathrm{x}}_{\mathrm{i_j}} \rangle, \langle \bar{\mathrm{y}}_{\mathrm{i_j}} \rangle)$, its corresponding equation would be

$$p(\bar{x}) = \mathrm{máx}(1 + q_{i_1}(\bar{x}_{i_j}), \ldots, 1 + q_{i_k}(\bar{x}_{i_k})) \quad \varphi_r$$

which takes the maximal height from all possible call chains. Each "1" corresponds to a single frame created for the corresponding call. Note that in this setting, tail call optimization can be also supported, by using an analysis that detects calls in tail position, and then replace their corresponding 1's by 0's. This is also a subject for future work.

Bibliografía

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *16th European Symposium on Programming, ESOP'07*, Lecture Notes in Computer Science. Springer, March 2007. Available online http://www.clip.dia.fi.upm.es/papers/jvm-cost-esop.pdf.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.

[5] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Post-proceedings of Formal Methods for Components and Objects (FMCO'07)*, number 5382 in LNCS, pages 113–133. Springer-Verlag, October 2008.

[6] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[7] E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In

15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05), number 3901 in LNCS, pages 115–132. Springer-Verlag, April 2006.

[8] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

[9] Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 15-17, 2008, Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer-Verlag, July 2008.

[10] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, LNCS 3362, pages 1–27. Springer-Verlag, 2005.

[11] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, and B. Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Technical Report RT-0203, 1997. citeseer.ist.psu.edu/barras97coq.html.

[12] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In *SEFM*, pages 86–95, 2005.

[13] F. Belli and O. Jack. Implementation-based analysis and testing of prolog programs. In *ISSTA*, pages 70–80, 1993.

[14] S. Belur and K. Bettadapura. Jdec: Java Decompiler. http://jdec.sourceforge.net/.

[15] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.

[16] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *Proc. of LPAR'04*, LNCS 3452, pages 347–362. Springer, 2004.

[17] Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java(TM). In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34. ACM, November 1999.

[18] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *ISMM*. ACM Press, 2008.

[19] Peter T. Breuer and Jonathan P. Bowen. Decompilation: The enumeration of types and grammars. *ACM Trans. Program. Lang. Syst.*, 16(5):1613–1647, 1994.

[20] M. Bruynooghe, J. Gallagher, and W. Humbeeck. Inference of Well-typings for Logic Programs with Application to Termination Analysis. In *12th International Static Analysis Symposium (SAS'05)*, volume 3672 of *LNCS*, pages 35–51. Springer-Verlag, 2005.

[21] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *FM'05*, number 3582 in LNCS. Springer, 2005.

[22] A. Chander, D. Espinosa, N. Islam, P. Lee, and G.Ñecula. Enforcing resource bounds via static verification of dynamic checks. In *ESOP'05*, volume 3444 of *LNCS*. Springer, 2005.

[23] W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard. Memory Usage Verification for OO Programs. In *Proc. of SAS'05*, volume 3672 of *LNCS*, pages 70–86, 2005.

[24] W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *ISMM*. ACM Press, 2008.

[25] The Ciao Development Team. The Ciao Multiparadigm Language and Program Development Environment, November 2006. The ALP Newsletter 19(3). The Association for Logic Programming.

[26] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Softw., Pract. Exper.*, 25(7):811–829, 1995.

[27] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *ICSM*, pages 228–237, 1998.

[28] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[29] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[30] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, 1977.

[31] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *The European Symposium on Programming (ESOP 2005)*, number 3444, pages 21–30. Springer-Verlag, 2005.

[32] Stephen-John Craig, John P. Gallagher, Michael Leuschel, and Kim S. Henriksen. Fully automatic binding-time analysis for prolog. In *LOPSTR*, pages 53–68, 2004.

[33] K. Crary and S. Weirich. Resource Bound Certification. In *POPL'00*, pages 184–198. ACM, 2000.

[34] S. K. Debray and N.-W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

[35] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[36] F. Degrave, T. Schrijvers, and W. Vanhoof. Automatic generation of test inputs for mercury. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, LNCS. Springer-Verlag, 2009.

[37] R. DeLine and K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[38] N. Dershowitz and J. P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, 1990.

[39] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[40] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *PPDP*, pages 63–74, 2007.

[41] Y. Futamura. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[42] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[43] J.P. Gallagher and J.C. Peralta. Using regular approximations for generalisation during partial evaluation. In *Proc. of the SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 44–51. ACM Press, 2000.

[44] G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 2002.

[45] A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Computational Logic*, pages 399–413, 2000.

[46] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Automated Software Engineering*, pages 219–228, 2000.

[47] Kim S. Henriksen and John P. Gallagher. Analysis and specialisation of a pic processor. In *SMC (2)*, pages 1131–1135. IEEE, 2004.

[48] Kim S. Henriksen and John P. Gallagher. Abstract interpretation of pic programs through logic programming. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 184–196. IEEE Computer Society, 2006.

[49] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[50] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[51] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, 2003.

[52] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[53] W.E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[54] J. Hughes and L. Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *Proc. of ICFP'99*, pages 70–81. ACM Press, 1999.

[55] JOlden Suite Collection. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.

[56] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.

[57] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[58] J.B. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[59] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *POPL*, pages 144–154, 1993.

[60] Xavier Leroy. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[61] M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 379–403. Springer, 2002.

[62] M. Leuschel and M. Bruynooghe. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[63] M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 340–375. Springer, 2004.

[64] M. Leuschel, S. Craig, and D. Elphick. Supervising offline partial evaluation of logic programs using online techniques. In *LOPSTR*, volume 4407 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2006.

[65] Michael Leuschel and Germán Vidal. Fast offline partial evaluation of large logic programs. In *LOPSTR*, pages 119–134, 2008.

[66] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[67] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[68] J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd Ext. Ed., 1987.

[69] G. Luo, G. Bochmann, B. Sarikaya, and M. Boyer. Control-flow based testing of prolog programs. In *In Proc. of the 3rd International Symposium on Software Reliability Engineering*, pages 104–113, 1992.

[70] M. Méndez-Lojo, J.Ñavas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915, pages 154–168. LNCS, August 2007.

[71] C. Meudec. Atgen: Automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.

[72] Jerome Miecznikowski and Laurie J. Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In R.Ñigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2002.

[73] R. A. Müller, C. Lembeck, and H. Kuchen. A symbolic java virtual machine for test case generation. In *IASTED Conf. on Software Engineering*, pages 365–371, 2004.

[74] G. Necula. Proof-Carrying Code. In *ACM Symposium on Principles of programming languages (POPL 1997)*, pages 106–119. ACM Press, 1997.

[75] G. Necula. Proof-Carrying Code. In *POPL'97*, pages 106–119. ACM Press, 1997.

[76] J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, *Static Analysis. 5th International Symposium, SAS'98, Pisa*, volume 1503 of *LNCS*, pages 246–261, 1998.

[77] J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In *Proc. of SAS'98*, volume 1503 of *LNCS*, pages 246–261, 1998.

[78] D. Pichardie. Bicolano (Byte Code Language in cOq). http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html.

[79] F. A. Rabhi and G. A. Manson. Using Complexity Functions to Control Parallelism in Functional Programs. Res. Rep. CS-90-1, Dept. of Computer Science, Univ. of Sheffield, England, January 1990.

[80] D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.

[81] Benjamin Schwarz, Saumya K. Debray, and Gregory R. Andrews. Disassembly of executable code revisited. In Arie van Deursen and Elizabeth Burd, editors, *WCRE*, pages 45–54. IEEE Computer Society, 2002.

[82] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic Accurate Live Memory Analysis for Garbage-Collected Languages. In *Proc. of LCTES/OM*, pages 102–111. ACM, 2001.

[83] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, volume 2575 of *LNCS*, pages 70–85, 2003.

[84] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.

[85] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.

[86] B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.

[87] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[88] Reinhard Wilhelm. Timing Analysis and Timing Predictability. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium (FMCO)*, volume 3657 of *LNCS, Revised Lectures*, pages 317–323. Springer, 2004.

[89] L. Zhao, T. Gu, J. Qian, and G. Cai. A novel test case generation method for prolog programs based on call patterns semantics. In *APLAS*, pages 105–121, 2007.

[90] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Apéndice A

Artículos de la Tesis (Papers of the Thesis)

Lista de artículos (List of papers):

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 124–139. Springer-Verlag, January 2007.

2. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Improving the Decompilation of Java Bytecode to Prolog by Partial Evaluation. In M. Huisman and F. Spoto, editors, *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, volume 190, Issue 1 of *Electronic Notes in Theoretical Computer Science*, pages 85–101. Elsevier - North Holland, July 2007.

3. E. Albert, J. Gallagher, M. Gómez-Zamalloa, and G. Puebla. Type-based Homeomorphic Embedding and its Applications to Online Partial Evaluation. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, volume 4915 of *LNCS*, pages 23–42. Springer-Verlag, February 2008.

4. E. Albert, J. Gallagher, M. Gómez-Zamalloa, and G. Puebla. Type-based Homeomorphic Embedding for Online Termination. *Information Processing Letters*, 2009.

5. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Modular Decompilation of Low-Level Code by Partial Evaluation. In *8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 239–248. IEEE Computer Society, September 2008.

6. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Journal of Information and Software Technology*, 2009. To appear.

7. E. Albert, M. Gómez-Zamalloa, and G. Puebla. Test Data Generation of Bytecode by clp Partial Evaluation. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, number 5438 in LNCS, pages 4–23. Springer-Verlag, March 2009.

8. M. Gómez-Zamalloa, E. Albert, and G. Puebla. On the Generation of Test Data for Prolog by Partial Evaluation. In *Workshop on Logic-based methods in Programming Environments (WLPE'08)*, volume WLPE/2008/06, pages 26–43, 2008.

9. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.

10. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *ISMM'09: Proceedings of the 8th international symposium on Memory management*, New York, NY, USA, June 2009. ACM Press.

Verification of Java Bytecode Using Analysis and Transformation of Logic Programs

E. Albert[1], M. Gómez-Zamalloa[1], L. Hubert[2], and G. Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain
{elvira,mzamalloa,laurent,german}@clip.dia.fi.upm.es

Abstract. State of the art analyzers in the Logic Programming (LP) paradigm are nowadays mature and sophisticated. They allow inferring a wide variety of global properties including termination, bounds on resource consumption, etc. The aim of this work is to automatically transfer the power of such analysis tools for LP to the analysis and verification of Java bytecode (JVML). In order to achieve our goal, we rely on well-known techniques for meta-programming and program specialization. More precisely, we propose to partially evaluate a JVML interpreter implemented in LP together with (an LP representation of) a JVML program and then analyze the residual program. Interestingly, at least for the examples we have studied, our approach produces very simple LP representations of the original JVML programs. This can be seen as a decompilation from JVML to high-level LP source. By reasoning about such residual programs, we can automatically prove in the CiaoPP system some non-trivial properties of JVML programs such as termination, run-time error freeness and infer bounds on its resource consumption. We are not aware of any other system which is able to verify such advanced properties of Java bytecode.

1 Introduction

Verifying programs in the (Constraint) Logic Programming paradigm —(C)LP— offers a good number of advantages, an important one being the maturity and sophistication of the analysis tools available for it. The work presented in this paper is motivated by the existence of *abstract interpretation*-based analyzers [3] which infer information on programs by interpreting ("running") them using abstract values rather than concrete ones, thus, obtaining safe approximations of programs behavior. These analyzers are parametric w.r.t. the so-called abstract domain, which provides a finite representation of possibly infinite sets of values. Different domains capture different properties of the program with different levels of precision and at different computational costs. This includes error freeness, data structure shape (like pointer sharing), bounds on data structure sizes, and other operational variable instantiation properties, as well as procedure-level properties such as determinacy, termination, non-failure, and bounds on resource consumption (time or space cost), etc. CiaoPP [9] is the *abstract interpretation*-based preprocessor of the Ciao (C)LP system, where analysis results have been applied to perform high- and low-level optimizations and *program verification*.

M. Hanus (Ed.): PADL 2007, LNCS 4354, pp. 124–139, 2007.

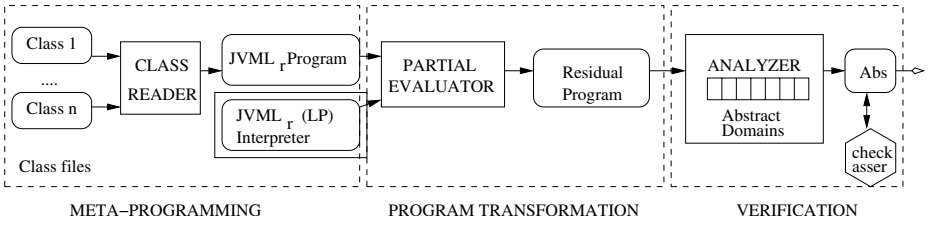


Fig. 1. Verification of Java Bytecode using Logic Programming Tools

A principal advantage of verifying programs on the (LP) *source* code level is that we can infer complex global properties (like the aforementioned ones) for them. However, in certain applications like within the context of mobile code, one may only have the *object* code available. In general, analysis tools for such low-level languages are unavoidably more complicated than for high-level languages because they have to cope with complicated and unstructured control flow. Furthermore, as the JVML (Java Virtual Machine Language, i.e., Java bytecode) is a stack-based language, stacks cells are used to store intermediate values, and therefore their type can change from one assignment to another, and they can also be used to store 32 bits of a 64 bit value, which make the inference of stack information much more difficult. Besides, it is a non trivial task to specify/infer global properties for the bytecode by using pre- and post-conditions (as it is usually done in existing tools for high-level languages).

The aim of this work is to provide a practical framework for the verification of JVML which exploits the expressiveness, automation and genericity of the advanced analysis tools for LP source. In order to achieve this goal, we will focus on the techniques of meta-programming, program specialization and static analysis that together support the use of LP tools to analyze JVML programs. Interpretative approaches which rely on CLP tools have been applied to analyze rather restricted versions of high-level imperative languages [13] and also assembly code for PIC [8], an 8-bit microprocessor. However, to the best of our knowledge, this is the first time the interpretative approach has been successfully applied to a general purpose, realistic, imperative programming language.

Overview. Fig. 1 presents a general overview of our approach. We depict an element within a straight box to denote its use as a program and a rounded box for data. The whole verification process is split in three main parts:

1. *Meta-programming.* We use LP as a language for representing and manipulating JVML programs. We have implemented an automatic translator, called CLASS_READER, which given a set of `.class` files {`Class 1`,..., `Class n`} returns P, an LP representation of them in JVML$_r$ (a representative subset of JVML presented in Sect. 2). Furthermore, we also describe in Sect. 3 an interpreter in LP, called JVML$_r$_INT, which captures the JVM semantics. The interpreter has been extended in order to compute *execution traces*, which will be very useful for reasoning about certain properties.

2. *Partial evaluation.* The development of partial evaluation techniques [10] has allowed the so-called "interpretative approach" to compilation which consists in specializing an interpreter w.r.t. a fixed object code. We have used an existing PARTIAL_EVALUATOR for LP in order to specialize the JVML$_r$_INT w.r.t. P. As a result, we obtain I_P, an LP residual program which can be seen as a decompiled and translated version of P into LP (see Sect. 4).
3. *Verification of Java bytecode.* The final goal is that the JVML program can be verified by analyzing the residual program I_P obtained in Step 2) above by using state-of-the-art ANALYZERs developed for LP, as we will see in Sect. 5.

The resulting scheme has been implemented and incorporated in the `CiaoPP` preprocessor. Our preliminary experiments show that it is possible to infer global properties of the computation of the residual LP programs. We believe our proposed approach is very promising in order to bring the analysis power of declarative languages to low-level, imperative code such as Java bytecode.

2 The Class Reader (JVML to JVML$_r$ in LP)

As notation, we use *Prog* to denote LP programs and *Class* to denote `.class` files (i.e., JVML classes). The input of our verification process is a set of `.class` files, denoted as $C_1 \ldots C_n \in Class$, as specified by the Java Virtual Machine Specification [12]. Then, the CLASS_READER takes $C_1 \ldots C_n$ and returns an LP file which contains all the information in $C_1 \ldots C_n$ represented in our JVML$_r$ language. JVML$_r$ is a representative subset of the JVML language which is able to handle: classes, interfaces, arrays, objects, constructors, exceptions, method call to class and instance methods, etc. For simplicity, some other features such as packages, concurrency and types as float, double, long and string are left out of the chosen subset. For conciseness, we use JVML$_r$_*Prog* to make it explicit that an LP program contains a JVML$_r$ representation. The differences between JVML and JVML$_r$ are essentially the following:

1. *Bytecode factorization.* Some instructions in JVML have a similar behavior and have been factorized in JVML$_r$ in order to have fewer instructions[1]. This makes the JVML$_r$ code easier to read (as well as the traces which will be discussed in Sect. 3) and the JVML$_r$_INT easier to program and maintain.
2. *References resolution.* The original JVML instructions contain indexes onto the *constant-pool* table [12], a structure present in the `.class` file which stores different kinds of data (constants, field and method names, descriptors, class names, etc.) and which is used in order to make bytecode programs as compact as possible. The CLASS_READER removes all references to the constant-pool table in the bytecode instructions by replacing them with the complete information to facilitate the task of the tools which need to handle the bytecode later.

[1] This allows covering over 200 instructions of JVML in 54 instructions in JVML$_r$.

```
1 class(
2   className(packageName(''),shortClassName('Rational')),final(false),public(true),
3   abstract(false),className(packageName('java/lang/'),shortClassName('Object')),[],
4   [field(
5      fieldSignature(
6        fieldName(
7          className(packageName(''),shortClassName('Rational')),shortFieldName(num)),
8        primitiveType(int)),
9      final(false),static(false),public,initialValue(undef)),
10    field(
11     fieldSignature(
12       fieldName(
13         className(packageName(''),shortClassName('Rational')),shortFieldName(den)),
14         primitiveType(int)),
15     final(false),static(false),public,initialValue(undef))],
16  [method(
17     methodSignature(
18       methodName(
19         className(packageName(''),shortClassName('Rational')),shortMethodName('<init>')),
20       [primitiveType(int),primitiveType(int)],none),
21     bytecodeMethod(3,2,0,methodId('Rational_class',1),[]),
22     final(false),static(false),public),
23    method(
24     methodSignature(
25       methodName(
26         className(packageName(''),shortClassName('Rational')),shortMethodName(exp)),
27       [primitiveType(int)],
28       refType(classType(className(packageName(''),shortClassName('Rational'))))),
29      bytecodeMethod(4,4,0,methodId('Rational_class',2),[]),
30      final(false),static(false),public),
31    method(
32     methodSignature(
33       methodName(
34         className(packageName(''),shortClassName('Rational')),shortMethodName(expMain)),
35       [primitiveType(int),primitiveType(int),primitiveType(int)],
36       refType(classType(className(packageName(''),shortClassName('Rational'))))),
37     bytecodeMethod(3,4,0,methodId('Rational_class',3),[]),
38     final(false),static(true),public)]).
```

Fig. 2. Extract of the Program Fact Describing the Rational Class of Running Example

The `Ciao` file generated by the CLASS_READER contains the bytecode instructions for all methods in $C_1 \ldots C_n$, represented as a set of facts; and also, a single fact obtained by putting together all the other information available in the `.class` files (class name, methods and fields signatures, etc.).

Example 1 (running example). Our running example considers a main Java class named `Rational` which represents rational numbers using two attributes: `num` and `den`. The class has a constructor, an instance method `exp` for computing the exponential of rational numbers w.r.t. a given exponent (the result is returned on a new rational object), and a static method `expMain` which given three integers, creates a new rational object using the first two ones as numerator and denominator, respectively, and invokes its `exp` method using the third argument as parameter. Finally, it returns the corresponding rational object. This example features arithmetic operations, object creation, field access, and invocation of both class and instance methods. It also shows that our approach is not restricted to intra-procedural analysis.

In Fig. 2, we show the extract of the program fact corresponding to class `Rational`. Line numbers are provided for convenience but they are not part of the

```
1  class(
2    className(packageName(''),shortClassName('Rational')),final(false),public(true),
3    abstract(false),className(packageName('java/lang/'),shortClassName('Object')),[],
4    [field(
5       fieldSignature(
6         fieldName(
7           className(packageName(''),shortClassName('Rational')),shortFieldName(num)),
8         primitiveType(int)),
9       final(false),static(false),public,initialValue(undef)),
10    field(
11      fieldSignature(
12        fieldName(
13          className(packageName(''),shortClassName('Rational')),shortFieldName(den)),
14          primitiveType(int)),
15      final(false),static(false),public,initialValue(undef))],
16   [method(
17      methodSignature(
18        methodName(
19          className(packageName(''),shortClassName('Rational')),shortMethodName('<init>')),
20        [primitiveType(int),primitiveType(int)],none),
21      bytecodeMethod(3,2,0,methodId('Rational_class',1),[]),
22      final(false),static(false),public),
23    method(
24      methodSignature(
25        methodName(
26          className(packageName(''),shortClassName('Rational')),shortMethodName(exp)),
27        [primitiveType(int)],
28        refType(classType(className(packageName(''),shortClassName('Rational'))))),
29       bytecodeMethod(4,4,0,methodId('Rational_class',2),[]),
30       final(false),static(false),public),
31    method(
32      methodSignature(
33        methodName(
34          className(packageName(''),shortClassName('Rational')),shortMethodName(expMain)),
35        [primitiveType(int),primitiveType(int),primitiveType(int)],
36        refType(classType(className(packageName(''),shortClassName('Rational'))))),
37      bytecodeMethod(3,4,0,methodId('Rational_class',3),[]),
38      final(false),static(true),public)]).
```

Fig. 3. Extract of the Bytecode facts of our Running Example

code. The description of the field `num` appears in Lines 4-9, `den` in L.10-15 and the methods in L.16-38. For conciseness, only methods actually used are shown. The first method (L.16-22) is a constructor that takes two integers (L.20) as arguments. The second method (L.23-30) is named `exp` (L.26), it is an instance method (cf. `static(false)` L.30)) and takes an integer (L.27) as a parameter and returns an instance of `Rational` (L.28). Finally, the last method (L.31-38), `expMain`, is a class method (cf. `static(true)` L.38), that takes as parameters three integers (L.35) and returns an instance of `Rational` (L.36).

Fig. 3 presents the bytecode facts corresponding to the methods `exp` and `expMain`. Each fact is of the form `bytecode(PC,MethodID,Class,Inst,Size)`, where `Class` and `MethodID`, respectively, identify the class and the method to which the instruction `Inst` belongs. `PC` corresponds to the program counter and `Size` to the number of bytes of the instruction in order to be able to compute the next value of the program counter. The class method number 3 (i.e., `expMain`) creates first an instance of `Rational` (Instructions 0-6) and then invokes the instance method `exp` (I.9-10). The bytecode of the method number 2 (i.e., `exp`), can be divided in 3 parts. First, the initialization (I.0-3) of two local variables,

say x_2 and x_3, to 1. Then, the loop body (I.4-25) first compares the exponent to 0 and, if it is less or equal to 0, exits the loop by jumping 23 bytes ahead (I.4-5). Then, the current value of x_2 (`iload`) and the denominator (`aload` and `getfield`) are retrieved (I.8-10), multiplied and stored in x_2 (I.13-14). The same is done for x_3 with the numerator in I.15-21. Finally, the value of the exponent is decreased by one (I.22) and PC is decreased by 21 (I.25) i.e., we jump back to the beginning of the loop. After the loop, the method creates an instance of `Rational`, stores the result (I.28-34), and returns this object (I.37).

3 Specification of the Dynamic Semantics

(C)LP programs have been used traditionally for expressing the semantics of both high- and low-level languages [13,17]. In our approach, we express the JVML semantics in `Ciao`. The formal JVML specification chosen for our work is Bicolano [14], which is written with the Coq Proof Assistant [1]. This allows checking that the specification is consistent and also proving properties on the behavior of some programs.

In the specification, a state is modeled by a 3-tuple[2] $\langle\ Heap, Frame, StackFrame\ \rangle$ which represents the machine's state where $Heap$ represents the contents of the heap, $Frame$ represents the execution state of the current *Method* and, $StackFrame$ is a list of frames corresponding to the call stack. Each frame is of the form $\langle\ Method, PC, OperandStack, LocalVar\ \rangle$ and contains the stack of operands $OperandStack$ and the values of the local variables $LocalVar$ at the program point PC of the method $Method$. The definition of the dynamic semantics is based on the notion of *step*.

Definition 1 ($step \xrightarrow{L}_P$)**.** *The dynamic semantics of each instruction is specified as a partial function* $step : \text{JVML}_r\_Prog \times State_{JVM} \rightarrow State_{JVM} \times Step\_Name$ *that, given a program* $P \in \text{JVML}_r\_Prog$ *and a state* $S \in State_{JVM}$*, computes the next state* $S' \in State_{JVM}$ *and returns the name of the step* $L \in Step\_Name$*. For convenience, we write* $S \xrightarrow{L}_P S'$ *to denote* $step(P, S) = (S', L)$*.*

In order to formally define our interpreter, we need to define the following function which iterates over the steps of the program until obtaining a final state.

Definition 2 ($\xrightarrow{T}{}^*_P$)**.** *Let* $\xrightarrow{T}{}^*_P$ *be a relation on* $State_{JVM}$ *with* $S \xrightarrow{T}{}^*_P S'$ *iff:*

- *there exists a sequence of steps* L_1 *to* L_n *such that* $S \xrightarrow{L_1}_P \ldots \xrightarrow{L_n}_P S'$*,*
- *there is no state* $S'' \in State_{JVM}$ *such that* $S' \xrightarrow{L}_P S''$*, and*
- $T \in Traces$ *such that* $T = [L_1, \ldots, L_n]$ *is the list of the names of the steps.*

We can now define a general interpreter which takes as parameters a program and a *method invocation specification* (MIS in the following) that indicates: 1)

[2] Both in Bicolano and in our implementation there is another kind of state for exceptions, but we have omitted it from this formalization for the sake of simplicity.

the method the execution should start from, 2) the corresponding effective parameters of the method which will often contain logical variables or partially instantiated terms (and should be interpreted as the set of all their instances) and 3) an initial heap. The interpreter relies on an EXECUTE function that takes as parameters a program $P \in \text{JVML}_r\text{\_}Prog$ and a state $S \in State_{JVM}$ and returns (S', T) where $S \xrightarrow{T}{}^{*}_{P} S'$.

The following definition of JVML_r_INT computes, in addition to the return value of the method called, also the trace which captures the computation history. Traces represent the semantic steps used and therefore do not only represent instructions, as the context has also some importance. They allow us to distinguish, for example, for a same instruction, the step that throws an exception from the normal behavior. E.g., `invokevirtual_step_ok` and `invokevirtual_step_NullPointerException` represent, respectively, a normal method call and a method call on a null reference that throws an exception.

Definition 3 (JVML_r_INT)**.** *Let M be a* MIS *that contains a method signature, the parameters for the method and a heap, written as $M \in$ MIS. We define a general interpreter* $\text{JVML}_r\text{\_INT}(P, M) = (R, T)$ *with*

- $S = initialState(P, M)$*, where function initialState builds, from the program P and the* MIS M*, a state $S \in State_{JVM}$,*
- $\text{EXECUTE}(P, S) = (S', T)$ *and*
- $R = result\_of(S')$ *is the result of the execution of the method specified by M (the value on top of the stack of the current frame of S').*

This definition of JVML_r_INT returns the trace and the result of the method but it is straightforward to modify the definitions of JVML_r_INT and EXECUTE to return less information or to add more. This gives more flexibility to our interpretative approach when compared to direct compilation: for example, if needed, we can return in an additional argument a list containing the information about each state which we would like to *observe* in order to prove properties which may require a deeper inspection of execution states.

4 Automatic Generation of Residual Programs

Partial evaluation (PE) [10] is a semantics-based program optimization technique which has been deeply investigated within different programming paradigms. The main purpose of PE is to specialize a given program w.r.t. the *static data*, i.e., the part of its input data which is known—hence it is also known as *program specialization*. The partially evaluated (or residual) program will be (hopefully) executed more efficiently since those computations that depend only on the static data are performed once and for all at PE time. We use the partial evaluator for LP programs of [15] which is part of `CiaoPP`. Here, we represent it as a function PARTIAL_EVALUATOR: $Prog \times Data \rightarrow Prog$ which, for a given program $P \in Prog$ and static data $S \in Data$, returns a residual program $P_S \in Prog$ which is a *specialization* [10] of P w.r.t. S.

The development of PE, program specialization and related techniques [6,10,7] has led to an alternative approach to compilation (known as the first Futamura projection) based on specializing an interpreter with respect to a fixed object program. The success of the application of the technique involves eliminating the overhead of parsing the program, fetching instructions, etc., and leading to a residual program whose operations mimic those of the object program. This can also be seen as a translation of the object program into another programming language, in our case `Ciao`. The *residual* program is ready now to be, for instance, efficiently executed in such language or, as in our case, accurately analyzed by tools for the language in which it has been translated. The application of this interpretative approach to compilation within our framework consists in partially evaluating the JVML$_r$_INT w.r.t. $P =$ CLASS_READER$(C_1, \ldots, C_n)$ and a MIS.

Definition 4 (LP residual program). *Let* JVML$_r$_INT $\in$ *Prog be a* JVML$_r$ *interpreter,* $M \in$ MIS *and* $C_1, \ldots, C_n \in$ *Class be a set of classes. The* LP residual program*,* I_P*, for* JVML$_r$_INT *w.r.t.* $C_1, \ldots, C_n$ *and* M *is defined as* $I_P =$PARTIAL_EVALUATOR$($JVML$_r$_INT, (CLASS_READER$(C_1, \ldots, C_n), M))$.

Note that, instead of using the interpretative approach, we could have implemented a compiler from Java bytecode to LP. However, we believe that the interpretative approach has at least the following advantages: 1) more flexible, in the sense that it is easy to modify the interpreter in order to observe new properties of interest, see Sect. 3, 2) easier to trust, in the sense that it is rather difficult to prove (or trust) that the compiler preserves the program semantics and, it is also complicated to explicitly specify what the semantics used is, 3) easier to maintain, new changes in the JVM semantics can be easily reflected in the interpreter by modifying (or adding) a proper "step" definition, and 4) easier to implement, provided a powerful partial evaluator for LP is available.

Example 2 (residual programs). We now want to partially evaluate our implementation of the interpreter which does not output the trace (see Sect. 3) w.r.t. the bytecode method `expMain` in Ex. 1, an empty heap and three free variables as parameters. The size of the program to be partially evaluated (i.e., interpreter) is 86,326 bytes (2,240 lines) while the size of the data (i.e., bytecode representation) is 16,677 bytes (101 lines) of JVML$_r$. The partial evaluator has different options for tuning the level of specialization. For this example, we have used local and global control strategies based on *homeomorphic embedding* (see [11]).

We show in Fig. 4 the residual program resulting of such automatic PE. The parameters `A`, `B` and `C` of `expMain/5` represent the numerator, denominator and exponent, respectively. The fourth and fifth parameters represent, respectively, the top of the stack and the heap where the method result (i.e., an object of type `Rational` in the bytecode) will be returned. In particular, the result corresponds to the second element, `ref(loc(2))`, in the heap. Note that this object is represented in our LP program as a list of two atoms, the first one corresponds to attribute `num` and the second one to `den`. The first two rules for `expMain/5` are the base cases for exponents `C = 0` and `C = 1`, respectively. The third rule, for `C > 1`, uses an auxiliary recursive predicate `execute/6` which computes $\texttt{A}^{\texttt{C}+1}$

```
expMain(A,B,C,ref(loc(2)),heap([[num(int(A)),num(int(B))],
            [num(int(1)),num(int(1))]])) :- C=<0 .
expMain(A,B,C,ref(loc(2)),heap([[num(int(A)),num(int(B))],
            [num(int(A)),num(int(B))]])) :- C>0, F is C-1, F=<0 .
expMain(A,B,C,D,E) :- C>0, H is C-1, H>0, I is A*A,
            J is B*B, K is H-1, execute(A,B,K,I,J,E,D) .

execute(A,B,C,D,E,heap([[num(int(A)),num(int(B))],
            [num(int(D)),num(int(E))]]),ref(loc(2))) :- C=<0 .
execute(A,B,C,D,E,G,L) :- C>0, N is D*A, O is E*B, P is C-1,
            execute(A,B,P,N,O,G,L) .
```

Fig. 4. Residual Exponential Program without Trace

and B^{C+1} and returns the result in the second element of the heap. It should be noted that our PE tool has done a very good job by transforming a rather large interpreter into a small residual program (where all the interpretation overhead has been removed). The most relevant point to notice about the residual program is that we have converted low level jumps into a recursive behavior and achieved a very satisfactory translation from the Java bytecode method `expMain`. Indeed, it is not very different from the `Ciao` version one could have written by hand, provided that we need to store the result in the fifth argument of predicate `expMain/5` as an object in the heap, using the corresponding syntax.

While the above LP program can be of a lot of interest when reasoning about functional properties of the code, it is also of great importance to augment the interpreter with an additional argument which computes a trace (see Def. 3) in order to capture the computation history. The residual program which computes execution traces is `expMain/4`, which on success contains in the fourth argument the execution trace at the level of Java bytecode (rather than the top of the stack and the heap). Below, we show the recursive rule of predicate `execute/8` whose last argument represents the trace (and corresponds to the second rule of `execute/7` without trace in Fig. 4):

```
execute(B,C,D,E,F,G,I,[goto_step_ok,iload_step,if0_step_continue,
         iload_step,aload_step_ok,getfield_step_ok,ibinop_step_ok,
         istore_step_ok,iload_step,aload_step_ok,getfield_step_ok,
         ibinop_step_ok,istore_step_ok,iinc_step|H]) :-
     D>0, I is E*B, J is F*C, K is D-1, execute(B,C,K,I,J,G,I,H) .
```

As we will see in the next section, this trace will allow observing a good number of interesting properties about the program.

5 Verification of Java Bytecode Using LP Analysis Tools

Having obtained an LP representation of a Java bytecode program, the next task is to use existing analysis tools for LP in order to infer and verify properties about the original bytecode program. We now recall some basic notions

on *abstract interpretation* [3]. *Abstract interpretation* provides a general formal framework for computing safe approximations of program behaviour. In this framework, programs are interpreted using *abstract values* instead of *concrete values*. An abstract value is a finite representation of a, possibly infinite, set of concrete values in the concrete domain D. The set of all possible abstract values constitutes the *abstract domain*, denoted D_α, which is usually a complete lattice or cpo which is ascending chain finite. Abstract values and sets of concrete values are related by an *abstraction* function $\alpha : 2^D \rightarrow D_\alpha$, and a *concretization* function $\gamma : D_\alpha \rightarrow 2^D$. The concrete and abstract domains must be related in such a way that the following condition holds [3]: $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. In general, the comparison in D_α, written $\sqsubseteq$, is induced by $\subseteq$ and α.

We rely on a generic analysis algorithm (in the style of [9]) defined as a function ANALYZER: $Prog \times AAtom \times ADom \rightarrow AApprox$ which takes a program $P \in Prog$, an abstract domain $D_\alpha \in ADom$ and a set of abstract atoms $S_\alpha \in AAtom$ which are descriptions of the entries (or calling modes) into the program and returns $Approx_\alpha \in AApprox$. Correctness of analysis ensures that $Approx_\alpha$ safely approximates the semantics of P. We denote that S_α and $Approx_\alpha$ are abstract semantic values in D_α by using the same subscript α.

In order to verify the program, the user has to provide the intended semantics $Assert_\alpha$ (or specification) as a semantic value in D_α in terms of *assertions* (these are linguistic constructions which allow expressing properties of programs) [16]. This intended semantics embodies the requirements as an expression of the user's expectations. The *verifier* has to compare the (actual) inferred semantics $Approx_\alpha$ w.r.t. $Assert_\alpha$. We use the *abstract interpretation*-based verifier integrated in `CiaoPP`. It is dealt here as a function AI_VERIFIER: $Prog \times AAtom \times ADom \times AAssert \rightarrow boolean$ which for a given program $P \in Prog$, a set of abstract atoms $S_\alpha \in AAtom$, an abstract domain $D_\alpha \in ADom$ and an intended semantics $Assert_\alpha$ in D_α succeeds if the approximation computed by ANALYZER(P, S_α, D_α)=$Approx_\alpha$ entails that P satisfies $Assert_\alpha$, i.e., $Approx_\alpha \sqsubseteq Assert_\alpha$.

Definition 5 (verified bytecode). *Let $I_P \in Prog$ be an LP residual program for* JVML$_r$_INT *w.r.t. $C_1, \ldots, C_n \in Class$ and $M \in$* MIS *(see Def. 3). Let $D_\alpha \in ADom$ be an abstract domain, $S_\alpha \in AAtom$ be a set of abstract atoms and $Assert_\alpha \in D_\alpha$ be the abstract intended semantics of I_P. We say that $(C_1, \ldots, C_n, M)$ is verified w.r.t. $Assert_\alpha$ in $ADom$ if* AI_VERIFIER$(I_P, S_\alpha, D_\alpha, Assert_\alpha)$ *succeeds.*

In principle, any of the considerable number of abstract domains developed for *abstract interpretation* of logic programs can be applied to residual programs, as well as to any other program. In addition, arguably, analysis of logic programs is inherently simpler than that of Java bytecode since the bytecode programs decompiled into logic programs no longer contain an operand stack for arithmetic and execution flow is transformed from jumps (since loops in the Java program are compiled into conditional and unconditional jumps) into recursion.

5.1 Run-Time Error Freeness Analysis

The use of objects in Ex. 1 could in principle issue exceptions of type `NullPointerException`. Clearly, the execution of the `expMain` method will not produce any exception, as the unique object used is created within the method. However, the JVM is unaware of this and has to perform the corresponding run-time test. We illustrate that by using our approach we can statically verify that the previous code cannot issue such an exception (nor any other kind of run-time error).

First, we proceed to specify in `Ciao` the property "`goodtrace`" which encodes the fact that a bytecode program is run-time error free in the sense that its execution does not issue `NullPointerException` nor any other kind of run-time error (e.g., `ArrayIndexOutOfBoundsException`, etc). As this property is not predefined in `Ciao`, we declare it as a regular type using the `regtype` declarations in `CiaoPP`. Formally, we define this property as a *regular unary logic* program, see [5]. The following regular type `goodtrace` defines this notion of safety for our example (for conciseness, we omit the bytecode instructions which do not appear in our program):

```
:- regtype goodtrace/1.
goodtrace(T) :- list(T,goodstep).

:- regtype goodstep/1.
goodstep(iinc_step).          goodstep(aload_step_ok).        goodstep(invokevirtual_step_ok).
goodstep(iload_step).         goodstep(if0_step_jump).        goodstep(invokestatic_step_ok).
goodstep(normal_end).         goodstep(const_step_ok).        goodstep(if0_step_continue).
goodstep(new_step_ok).        goodstep(return_step_ok).       goodstep(if_icmp_step_jump).
goodstep(pop_step_ok).        goodstep(astore_step_ok).       goodstep(putfield_step_ok).
goodstep(dup_step_ok).        goodstep(istore_step_ok).       goodstep(getfield_step_ok).
goodstep(goto_step_ok).       goodstep(ibinop_step_ok).       goodstep(if_icmp_step_continue).
goodstep(areturn_step_ok).    goodstep(invokespecial_step_here_ok).
```

Next, the version with traces of the residual program in Fig. 4 is extended with the following assertions:

```
:- entry expMain(Num,Den,Exp,Trace):(num(Num),num(Den),num(Exp),var(Trace)).
:- check success expMain(Num,Den,Exp,Trace) => goodtrace(Trace).
```

The entry assertion describes the valid external queries to predicate `expMain/4`, where the first three parameters are of type `num` and the fourth one is a variable. We use the "`success`" assertion as a way to provide a partial specification of the program. It should be interpreted as: for all calls to `expMain(Num,Den,Exp, Trace)`, if the call succeeds, then `Trace` must be a `goodtrace`.

Finally, we use `CiaoPP` to perform regular type analysis using the *eterms* domain [18]. This allows computing safe approximations of the success states of all predicates. After this, `CiaoPP` performs compile-time checking of the `success` assertion above, comparing it with the assertions inferred by the analysis, and produces as output the following assertion:

```
:- checked success expMain(Num,Den,Exp,Trace) => goodtrace(Trace).
```

Thus, the provided assertion has been *validated* (marked as `checked`).

5.2 Cost Analysis and Termination

As mentioned before, *abstract interpretation*-based program analysis techniques allow inferring very rich information including also resource-related issues. For example, `CiaoPP` can compute upper and lower bounds on the number of execution steps required by the computation [9,4]. Such bounds are expressed as functions on the sizes of the input arguments. Various metrics are used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. Types, modes, and size measures are first automatically inferred by the analyzers and then used in the size and cost analysis.

Let us illustrate the cost analysis in `CiaoPP` on our running example. We consider a slightly modified version of the residual program in Fig. 4 in which we have eliminated the accumulating parameter due to a current limitation of the cost analysis in `CiaoPP`. The cost analysis can then infer the following property of the recursive predicate `execute/5` (and a similar one of `expMain/4`) using the same entry assertion as in Sect. 5.1:

```
:- true pred execute(A,B,C,D,E): (num(A),num(B),num(C),var(D),var(E))
        => ( num(A), num(B), num(C), num(D), num(E),
             size_ub(A,int(A)), size_ub(B,int(B)), size_ub(C,int(C)),
             size_ub(D,expMain(int(A),int(C)+1)+int(A)),
             size_ub(E,expMain(int(B),int(C)+1)+int(B)) )
         + steps_ub(int(C)+1).
```

which states that `execute/5` is called in this program with the first three parameters being of type `num` (i.e., bound to numbers) and two variables. The part of the assertion after the `=>` symbol indicates that on success of the predicate all five parameters are bound to numbers. This is used by the cost analysis in order to set the integer-value as size-metric for all five arguments. The first three arguments are input to the procedure and thus their size (value) is fixed. The last two arguments are output and their size (value) is a function on the value of (some of) the first three arguments. The upper bound computed by the analysis for D (i.e., the fourth argument) is $A^{C+1} + A$. Note that this is a correct upper bound, though the most accurate one is indeed A^{C+1}. A similar situation occurs with the upper bound for the fifth argument (E). Finally, the part of the assertion after the `+` symbol indicates that an upper bound on the number of execution steps is $C + 1$, which corresponds to a linear algorithmic complexity. This is indeed the most accurate upper bound possible, since predicate `execute/5` is called $C + 1$ times until C becomes zero. Note that, in this case, we do not mean the number of JVM steps in Def. 1, but the number of computational steps.

`CiaoPP`'s termination analysis relies on the cost analysis described in the previous section. In particular, it is able to prove termination of a program provided it obtains a non-infinite upper bound of its cost. Following the example of Sect. 5.2, `CiaoPP` is able to turn into `checked` status the following assertion (and the similar one for `expMain/4`): "`:- check comp execute(A,B,C,D,E) + terminates`". which ensures that the execution of the recursive predicate always terminates w.r.t. the previous entry.

6 Experiments and Discussion

We have implemented and performed a preliminary experimental evaluation of our framework within the `CiaoPP` preprocessor [9], where we have available a partial evaluator and a generic analysis engine with a good number of abstract domains, including the ones illustrated in the previous section. Our interpretative approach has required the implementation in `Ciao` of two new packages: the CLASS_READER (1141 lines of code) which parses the `.class` files into `Ciao` and the JVML$_r$_INT interpreter for the JVML$_r$ (3216 lines). These tools, together with a collection of examples, are available at: `http://cliplab.org/Systems/jvm-by-pe`.

Table 1 studies two crucial points for the practicality of our proposal: the size of the residual program and the relative efficiency of the full transformation+analysis process. As mentioned before, the algorithms are parametric w.r.t. the abstract domain. In our experiments we use *eterms*, an abstract domain based on regular types, that is very useful for reasoning about functional properties of the code, run-time errors, etc., which are crucial aspects for the safety of the Java bytecode. The system is implemented in Ciao 1.13 [2] with compilation to WAM bytecode. The experiments have been performed on an Intel P4 Xeon 2 GHz with 4 GB of RAM, running GNU Linux FC-2, 2.6.9.

The input "program" to be partially evaluated is the JVML$_r$_INT interpreter in all the examples. Then, the first group of columns **Bytecode** shows information about the input "data" to the partial evaluator, i.e., about the `.class` files. The columns **Class** and **Size** show the names of the classes used for the experiments and their sizes in bytes, respectively. The second column **Method** refers to the name of the method within each class which is going to form the MIS, i.e., to be the starting point for PE and context-sensitive program analysis. We use a set of classical algorithms as benchmarks. The first 9 methods belong to programs with iterations and static methods but without object-oriented features, where **mod**, **fact**, **gcd** and **lcm**, compute respectively the modulo, factorial, greatest-common-divisor and least-common-multiple (two versions); the **Combinatory** class has different methods for computing the number of selections of subsets given a set of elements for every ordering/repetition combination. The next two benchmarks, **LinearSearch** and **BinarySearch**, deal with arrays and correspond to the classic linear and binary search algorithms. Finally, the last four benchmarks correspond to programs which make extensive use of object-oriented features such as instance method invocation, field accessing and setting, object creation and initialization, etc.

The information about the "output" of the PE process appears in the second group of columns, **Residual**. The columns **Size** and **NUnfs** show the size in bytes of each residual program and the number of unfolding steps performed by the partial evaluator to generate it, respectively. We can observe that the partial evaluator has done a good job in all examples by transforming a rather large interpreter (86,326 bytes) in relatively small residual programs. The sizes range from 317 bytes for **m2** (99.4% reduction) to 4.911 for **Lcm2** (83.6 %). The number of required unfolding steps explains the high PE times, as we discuss

Table 1. Sizes of residual programs and transformation and analysis times

Bytecode			Residual		Times (ms)			
Class	**Size**	**Method**	**Size**	**NUnfs**	**Trans**	**PE**	**Ana**	**Total**
Mod	314	mod	956	1645	18	1244	59	1322
Fact	324	fact	1007	1537	19	1432	74	1525
Gcd	265	gcd	940	1273	18	1160	125	1303
Lcm	299	lcm	2260	4025	21	5832	817	6670
Lcm2	547	lcm2	4911	3724	26	3963	1185	5174
Combinatory	703	varNoRep	1314	1503	32	1837	87	1955
Combinatory	703	combNoRep	2177	2491	34	3676	150	3860
Combinatory	703	combRep	2151	3033	29	5331	950	6310
Combinatory	703	perm	1022	1256	29	1234	65	1328
LinearSearch	318	search	3114	8832	22	45228	296	45546
BinarySearch	412	search	3670	14117	23	72945	313	73282
Np	387	m2	317	527	20	502	12	534
ExpFact	890	main	2266	8353	35	23773	95	23903
Rational	559	expMain	3131	6613	31	13692	16	13739
Date	602	forward	11046	26982	36	80960	218	81213

below. A relevant point to note is that, for most programs, the size of the LP translation is larger than the original bytecode. This can be justified by the fact that the resulting program does not only represent the bytecode program but it also makes explicit some internal machinery of the JVM. This is the case, for instance, of the exception handling. As there are no `Ciao` exceptions in the residual program, the implicit exceptions in JVML have been made explicit in LP. Furthermore, the Java bytecode has been designed to be really compact, while the LP version has been designed to be easier to read by human beings and contains type information that must be inferred on the JVML. It should not be difficult to reduce the size of the residual bytecode if so required by, for example, simply using short identifiers.

The final part of the table provides the times for performing the transformations and the analysis process. Execution times are given in milliseconds and measure *runtime*. They are computed as the arithmetic mean of five runs. For each benchmark, **Trans**, **PE** and **Ana** are the times for executing the CLASS_READER, the partial evaluator and the analyzer, respectively. The column **Total** accumulates all the previous times. We can observe that most of the time is due to the partial evaluation phase (and this time is directly related to the number of unfolding steps performed). This is to be expected because the specialization of a large program (i.e., the interpreter) requires to perform many unfolding steps in all the examples (ranging from 14.117 steps for **search** in **BinarySearch** to 527 for **m2**), plus many additional generalization steps which are not shown in the table. The analysis time is then relatively low, as the residual programs to be analyzed are significantly smaller than the program to be partially evaluated.

As for future work, we plan to obtain accurate bounds on resource consumption by considering the traces that the residual program contains and the concrete cost of each bytecode instruction. Also, we are in the process of studying the scalability of our approach to the verification of larger Java bytecode programs. We also plan to exploit the advanced features of the partial evaluator which integrates abstract interpretation [15] in order to handle recursion.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry (TIN-2005-09207 *MERIT*), and the Madrid Regional Government (S-0505/TIC/0407 *PROMESAS*). The authors would like to thank David Pichardie and Samir Genaim for useful discussions on the Bicolano JVM specification and on termination analysis, respectively.

References

1. B. Barras et al. The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, 1997. citeseer.ist.psu.edu/barras97coq.html.
2. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla (Eds.). The Ciao System. (v1.13). At `http://clip.dia.fi.upm.es/Software/Ciao/`.
3. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
4. S. Debray, P. López, M. Hermenegildo, and N. Lin. Estimating the Computational Cost of Logic Programs. *Proc. of SAS'94*, LNCS 864, pp. 255–265. Springer.
5. T. Früwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. LICS'91*, pages 300–309, 1991.
6. Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
7. J. Gallagher. Transforming logic programs by specializing interpreters. In *Proc. of the 7th. European Conference on Artificial Intelligence*, 1986.
8. Kim S. Henriksen and John P. Gallagher. Analysis and specialisation of a pic processor. In *SMC (2)*, pages 1131–1135. IEEE, 2004.
9. M. Hermenegildo, G. Puebla, F. Bueno, and P. López. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation. *Science of Computer Programming*, 58(1–2):115–140, October 2005.
10. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, New York, 1993.
11. M. Leuschel. On the power of homeomorphic embedding for online termination. *Proc. of SAS'98*, pages 230–245, 1998. Springer-Verlag.
12. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* A-W, 1996.
13. J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of CLP. In *Proc. of SAS'98*, LNCS 1503, pp. 246–261, 1998.
14. D. Pichardie. Bicolano (Byte Code Language in cOq). http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html.
15. G. Puebla, E. Albert, and M. Hermenegildo. Abstract Interpretation with Specialized Definitions. In *Proc. of SAS'06*, LNCS. Springer, 2006. To appear.

16. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for CLP. In *Analysis and Visualization Tools for CP*, pages 23–61. Springer LNCS 1870, 2000.
17. Brian J. Ross. The partial evaluation of imperative programs using prolog. In *META*, pages 341–363, 1988.
18. C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *Proc. of SAS'02*, pages 102–116. Springer LNCS 2477, 2002.

Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 190 (2007) 85–101
www.elsevier.com/locate/entcs

Improving the Decompilation of Java Bytecode to Prolog by Partial Evaluation

Miguel Gómez-Zamalloa[1] Elvira Albert[1] Germán Puebla[2]

[1] *DSIC, Complutense University of Madrid,* {mzamalloa,elvira}@fdi.ucm.es

[2] *Technical University of Madrid,* german@fi.upm.es

Abstract

The *interpretative approach* to compilation allows compiling programs by partially evaluating an interpreter w.r.t. a source program. This approach, though very attractive in principle, has not been widely applied in practice mainly because of the difficulty in finding a partial evaluation strategy which always obtain "quality" compiled programs. In spite of this, in recent work we have performed a proof of concept of that, at least for some examples, this approach can be applied to *decompile* Java bytecode into Prolog. This allows applying existing advanced tools for analysis of logic programs in order to verify Java bytecode. However, successful partial evaluation of an interpreter for (a realistic subset of) Java bytecode is a rather challenging problem. The aim of this work is to improve the performance of the decompilation process above in two respects. First, we would like to obtain quality decompiled programs, i.e., simple and small. We refer to this as the *effectiveness* of the decompilation. Second, we would like the decompilation process to be as efficient as possible, both in terms of time and memory usage, in order to scale up in practice. We refer to this as the *efficiency* of the decompilation. With this aim, we propose several techniques for improving the partial evaluation strategy. We argue that our experimental results show that we are able to improve significantly the efficiency and effectiveness of the decompilation process.

Keywords: Java bytecode, decompilation, partial evaluation

1 Introduction

Partial evaluation [12] is a semantics-based program transformation technique whose main purpose is to optimize programs by specializing them w.r.t. part of their input (the *static* data)—hence it is also known as *program specialization*. Essentially, given a program P and a static data s, a partial evaluator returns a residual program P_s which is a specialized version of P w.r.t. the static data s such that $P(s,d) = P_s(d)$ for all *dynamic* (i.e., not static) data d. The development of partial evaluation techniques [12] has led to the so-called "interpretative approach" to compilation, also known as first Futamura projection [5]. In this approach, compilation of a source program P from a source language $\mathcal{L}_S$ to a target language $\mathcal{L}_O$ can in principle be performed by specializing an interpreter *Int* for $\mathcal{L}_S$ written in $\mathcal{L}_O$ w.r.t. P. The

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2007.02.062

program Int_P thus obtained can be akin to the result $Comp_S^O\ (P)$ of direct compilation of P using a compiler $Comp_S^O$ from $\mathcal{L}_S$ to $\mathcal{L}_O$. When $\mathcal{L}_S$ is Java bytecode and $\mathcal{L}_O$ is Prolog, we theoretically obtain a "decompilation" from (low-level) Java bytecode to (high-level) Prolog programs [1]. The motivation for obtaining a high level logic representation of the Java bytecode is clear: we can apply advanced tools developed for high level languages to the resulting programs without having to deal with the complicated unstructured control flow of the bytecode, the use of the stack, the exception handling, its object-oriented features, etc. In particular, for logic programming, we have available generic analysis tools which are incremental [10] and modular [4] that we will be able to directly use [1]. The motivations for using the interpretative approach to decompilation rather than implementing a compiler from Java bytecode to LP include: 1) flexibility, in the sense that it is easy to modify the interpreter in order to observe new properties of interest, 2) easy of trust, in the sense that it is rather difficult to prove (or trust) that the compiler preserves the program semantics and, it is also complicated to explicitly specify what the semantics used is, 3) easier to maintain, new changes in the JVM semantics can be easily reflected in the interpreter, and 4) easier to implement, provided a powerful partial evaluator for LP is available.

The success of the interpretative approach highly depends on eliminating the overhead of parsing the program, fetching instructions, etc., thus obtaining programs which are akin to those obtained by a traditional compiler. When both the $\mathcal{L}_S$ and $\mathcal{L}_O$ languages are the same, fully getting rid of the layer of interpretation is known as "Jones optimality" [11,12] and intuitively means that the result of specializing an interpreter Int w.r.t a program P should be basically the same as P, i.e., $Int_P \approx P$. Specializing interpreters has been a subject of research for many years, especially in the logic programming community (see, e.g., [22,23,15] and their references). However, despite these efforts, achieving Jones optimality in a systematic way is not straightforward since, given a program P, there are an infinite number of residual programs Int_P which can be obtained, and only a small fraction of them are akin to the results of direct compilation. As a result, only partial success has been achieved to date, such as in the specialization of a simple Vanilla interpreter, of the same interpreter extended with a debugger, and of a lambda interpreter [15].

The first requirement for achieving effective decompilation is to have a partial evaluator which is powerful (or "aggressive" in partial evaluation terminology) enough so as to remove the overhead of the interpretation level from the residual program. In a sense, the work in [1] shows that our partial evaluator [20,2] is aggressive enough for being used in the interpretative approach. The next two questions we need to answer, and which are addressed in this work are: is the control strategy used too aggressive in some cases? If so, it is possible to fix this problem? Note that the consequences of the strategy being too aggressive can be rather negative: it can introduce non-termination in the decompilation process and, even if the process terminates, it can result in inefficient decompilation (both in terms of time and memory) and in unnecessarily large residual programs. It should be noted

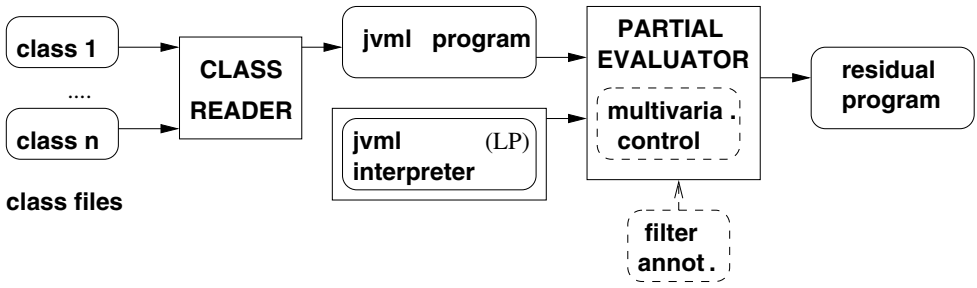


Fig. 1. Decompilation of Java Bytecode into Prolog by online PE w/ offline annotations

that memory efficiency of the decompilation process is quite important since it can happen that the decompiler fails to generate a residual program because the partial evaluator runs out of memory.

2 An Overview of the Decompilation Process

Figure 1 shows an overview of the interpretative decompilation process originally proposed in [1] and followed in this paper. Initially, given a set of `.class` files {`class 1`,..., `class n`}, a program called *class reader*, returns a representation of them in `Ciao` Prolog [3]. We use a slightly modified JVM language where some bytecode instructions are factorized and which contains some other minor simplifications (see [1]). Then, we have a JVML interpreter written in `Ciao` which captures the JVM semantics. The decompilation process consists in specializing the JVML interpreter w.r.t. the LP representation of the classes. In this work, we will improve the decompilation by introducing two new elements (which appear within a dashed box in the figure): an improved *multi-variance* control within the partial evaluator and *filter annotations* to refine the control of the *partial evaluator*.

2.1 The LP Representation of the Bytecode

The LP (`Ciao`) program generated by the *class reader* contains the bytecode instructions for all methods in {`class 1`,..., `class n`}. They are represented as a set of facts `bytecode`; and also, a single fact `class` obtained by putting together all the other information available in the `.class` files (class name, methods and fields signatures, etc.). Each `bytecode` fact is of the form `bytecode`(`PC`, `MethodID`, `Class`, `Inst`, `Size`), where `Class` and `MethodID`, respectively, identify the class and the method to which the instruction `Inst` belongs. `PC` corresponds to the program counter and `Size` to the number of bytes of the instruction in order to be able to compute the next value of the program counter. The form of the fact `class` is not relevant to this work but it can be observed in [1].

Example 2.1 [LP representation] Our running example consists of the single Java class `LinearSearch`, which appears in Fig 2. To the right, we show the `bytecode` facts corresponding to the method `search` identified with number "0" (second ar-

```
class LinearSearch{
 static int search(int[] xs,int x){
    int size = xs.length;
    boolean found = false;
    int i = 0;
    while ((i<size)&&(!found)){
      if (xs[i] == x) found = true;
      else i++;
    }
    return i;
 }
```

```
bytecode(0,'0',1,aload(0),1).
bytecode(1,'0',1,arraylength,1).
bytecode(2,'0',1,istore(2),1).
bytecode(3,'0',1,const(
          primitiveType(int),0),1).
bytecode(4,'0',1,istore(3),1).
bytecode(5,'0',1,const(
          primitiveType(int),0),1).
bytecode(6,'0',1,istore(4),2).
bytecode(8,'0',1,iload(4),2).
bytecode(10,'0',1,iload(2),1).
bytecode(11,'0',1,if_icmp(geInt,26),3).
bytecode(14,'0',1,iload(3),1).
bytecode(15,'0',1,if0(neInt,22),3).
bytecode(18,'0',1,aload(0),1).
bytecode(19,'0',1,iload(4),2).
bytecode(21,'0',1,iaload,1).
bytecode(22,'0',1,iload(1),1).
bytecode(23,'0',1,if_icmp(neInt,8),3).
bytecode(26,'0',1,const(
          primitiveType(int),1),1).
bytecode(27,'0',1,istore(3),1).
bytecode(28,'0',1,goto(-20),3).
bytecode(31,'0',1,iinc(4,1),3).
bytecode(34,'0',1,goto(-26),3).
bytecode(37,'0',1,iload(4),2).
bytecode(39,'0',1,ireturn,1).
```

Fig. 2. Java code and *LP* representation of Running Example

gument) of class number "1" (third argument). Bytecodes labeled from 0 to 6 (first argument) correspond to the first three initialization instructions in the Java program. Then, if the first conjunct in the `while` condition does not hold (bytecodes 8-11), the PC moves 26 positions downwards (i.e., to bytecode 37). Otherwise, the second conjunct is checked and similarly the PC can be increased in 22 positions (i.e., to bytecode 37). The condition in the `if` instruction corresponds to bytecodes 18-23, the `then` branch to 26-28 and the `else` branch to 31-34. Finally, bytecodes 37-39 represent the return.

2.2 The JVML Interpreter

The JVML interpreter expresses the JVM semantics in `Ciao` following the formal specification in Bicolano [19]. In our specification, a *state* is modeled by a term of the form $st(Heap, Frame, StackFrame)$ which represents the machine's state where: $Heap$ represents the contents of the heap, $Frame$ represents the execution state of the current *Method* and $StackFrame$ is a list of frames corresponding to the call stack. Each frame is of the form $fr(Method, PC, OperandStack, LocalVar)$ and contains the stack of operands *OperandStack* and the values of the local variables $LocalVar$ at the program point PC of the method $Method$. Note that, whenever we are at an exception state, the state and the frames will be represented accordingly as stE and frE terms resp., with the same arguments as their homologous st and fr, except for the *OperandStack* which will be a location number (instead of a list) referencing the corresponding exception object in the heap.

Fig. 3 shows a fragment of the `Ciao` JVML interpreter. Given the program and the current state, its main predicate `execute` first calls predicate `step`, which produces the *state* after executing the corresponding bytecode. The process iterates

```
execute(Program,State,FinalState) :-
        step(_,Program,State,NextState),
        execute(Program,NextState,FinalState).
execute(_P,State,State) :-
        check_return(State).
execute(Program,State,NextState) :-
        State=stE(Heap,frE(Method,PC,Loc,_),[]),
        NextState=st(Heap,fr(Method,PC,[ref(Loc)],_),[]),
        not_handled_exception(Program,State).

check_return(st(_H,fr(Method,PC,_Stack,_L),[])) :-
        instructionAt(Method,PC,return).
check_return(st(_H,fr(Method,PC,[num(int(_I))|_Stack],_L),[])) :-
        instructionAt(Method,PC,ireturn).
check_return(st(_H,fr(Method,PC,[ref(loc(_I))|_Stack],_L),[])) :-
        instructionAt(Method,PC,areturn).

step(goto_step_ok,_P,st(H,fr(M,PC,S,L),SF),st(H,fr(M,PCb,S,L),SF)) :-
        instructionAt(M,PC,goto(O)),
        PCb is PC+O.
...
```

Fig. 3. Fragment of the JVML interpreter

with a recursive call to predicate `execute` with the new state until one of the following conditions holds: 1) we reach a return instruction (i.e. `return`, `ireturn` or `areturn`), with the JVM call stack being empty, 2) we are in an exception state for which no suitable exception handler has been found, with the JVM call stack being empty, 3) there is no bytecode instruction at the current `PC`. The latter should never occur for a valid bytecode program. The whole interpreter, together with a collection of examples, are available at: `http://cliplab.org/Systems/jvm-by-pe`.

3 Basics of Online Partial Evaluation of Logic Programs

We assume familiarity with basic notions of logic programming [18]. Executing a logic program P for an atom A consists in building a so-called SLD tree for $P \cup \{A\}$ and then extracting the computed answer substitutions from every non-failing branch of the tree. Online partial evaluation builds upon the execution approach of logic programs with two main differences:

- In order to guarantee termination of the *unfolding* process, when building the SLD-trees, it is possible to choose *not* to further unfold a goal, and rather leave a leaf in the tree with a non-empty, possibly non-failing, goal. The resulting SLD is called a *partial* SLD tree. Note that even if the SLD trees for all possible queries are finite, the SLD to be built during partial evaluation may be infinite. The reason for this is that since dynamic values are not known at specialization time, the specialization SLD tree can have more branches (in particular, infinite branches) than the actual SLD tree at run-time. Which atom to select from each resolvent and when to stop unfolding is determined by the *unfolding rule*.
- The partial evaluator may have to build several SLD-trees to ensure that all atoms left in the leaves are "covered" by the root of some tree (this is known as the closeness condition of partial evaluation [17]). The so-called *abstraction operator* performs "generalizations" on the atoms that have to be partially evaluated in order to avoid computing partial SLD trees for an infinite number of atoms. When

Input: a program P and a set of atoms S
Output: a set of atoms T
Initialization: $i := 0$; $S_0 := S$
Repeat
1. $\mathcal{T} := \mathsf{unfold}(S_i, P)$;
2. $S_{i+1} := \mathsf{abstract}(S_i, \mathcal{T}_{calls})$;
3. $i := i + 1$;

Until $S_i = S_{i-1}$ (modulo renaming)
Return $T := S_i$

Fig. 4. Partial Evaluation Algorithm

all atoms are covered, then there is no need to build more trees and the process finishes. Details on abstraction operators appear in Section 4.

The essence of most algorithms for on-line partial evaluation of logic programs (see e.g. [8]) can be viewed in the algorithm shown in Figure 4, which is parametric w.r.t. the unfolding rule, unfold, and the abstraction operator, abstract. It starts from a program P and an initial set of atoms S. At each iteration, the *local control* is performed by the unfold rule which takes the current set of atoms S_i and the program and constructs partial SLD trees for the atoms in S_i. In the *global control*, when some calls in the leaves of the trees (named $\mathcal{T}_{calls}$ in the algorithm) are not properly *covered*, the operator abstract adds them to the new set of atoms to be partially evaluated in a proper "generalized" form such that termination is ensured (i.e., the condition $S_i = S_{i-1}$ is reached). Thus, basically, the algorithm iteratively constructs partial SLD trees until all their leaves are covered by the root nodes.

A partial evaluation of P w.r.t. S can then be systematically extracted from the resulting set of atoms T. The notion of *resultant* is used to generate a program rule associated to each root-to-leaf derivation of the SLD-trees for the final set of atoms T. In particular, given an SLD derivation of $P \cup \{A\}$ with $A \in T$ ending in B and θ the composition of the mgu's in the derivation step, then the rule $\theta(A) : -B$ is called the *resultant* of the derivation. A *partial evaluation* is then defined as the sequence of resultants associated to the derivations of the constructed partial SLD trees for all $P \cup \{A\}$ with $A \in T$.

4 Challenges in Specialization of JVM Interpreter

In order to achieve an *effective* decompilation, one of the crucial requirements is to have available control strategies (i.e., unfold and abstract operators) which are powerful enough to remove the interpreter overhead. For this reason, the experiments in [1] have been performed by using "aggressive" control strategies based on *homeomorphic embedding* [13,14]. In local control, by aggressiveness we mean unfolding rules which compute derivations as long as possible provided there are no termination problems. In global control, it denotes abstraction operators which generalize in as few situations as possible without endangering termination.

4.1 A Challenging Example

The example in Fig. 5 is instrumental to show the challenges which appear in the specialization of the JVM interpreter in Section 2.2. The specialization process starts by running the PE algorithm of Section 3 for the initial program P being the JVM interpreter and the following initial atom:

```
execute(Prog,st(heap([array(locationArray(_,primitiveType(int)),_)]),
                fr(method('int LinearSearch.search(int[],int)'),
                   0,[],[ref(1),_,0,0,0]),
                []),_)
```

where "`Prog`" would be instantiated to the constant term representing the corresponding JVM program of Sect. 2.1, and a "`_`" represents a logical variable. Let us note that this initial state has been built from a "method invocation specification" (MIS), i.e., a high level description specifying the method we want to decompile and its arguments values. In our case, we want to decompile a method for computing a linear search for any array of integers and any value as argument. Thus, we use "`int LinearSearch.search(int[] _,int _)`"as MIS.

In the figure, we depict (a reduced version of) one of the SLD trees that lead to an effective decompilation of our running example. In order to focus the attention to the relevant arguments only, each atom of the form `execute`(`Program`, `st`(`Heap`, `fr`(`Method`, `PC`, `Stack`, `LocalVar`), `CallStack`), `FinalState`) is represented in the figure as `execute`(`PC`, `LocalVar`) to show only its two key arguments. Indeed, the argument `Program` and `Method` are always constants, the `Stack` is not relevant and the `Heap` is not used in this example. The `CallStack` is always the empty list since the considered method does not invoke any other method (nor itself) and `FinalState` is always a fresh variable. Another simplification in the figure is that each arrow involves the application of several unfolding steps. In particular, the execution of the `step` predicate can be considered as a black box during unfolding, in the sense that it performs all the operations (i.e., a number of unfolding steps) and returns the corresponding state. Therefore, we can ignore the intermediate steps produced in order to unfold the calls to `step` and view each of the derivations as a sequence of the form `execute, step, execute, step,` ... (in the figure actually we only show one `step`). Some of the statements within the body of each `step` operation can stay as residual when they involve data which is not known at specialization time. The computation rule during unfolding is able to residualize calls which are not-sufficiently instantiated and select non-leftmost atoms in a safe way [2], in particular, further calls to `execute`.

4.2 Control Strategies based on Embedding

The interested reader is referred to Leuschel's work [16] where a detailed description of the embedding relation can be found. Informally, atom t_1 *embeds* atom t_2, written $t_2 \trianglelefteq t_1$, if t_2 can be obtained from t_1 by deleting some operators, e.g.,

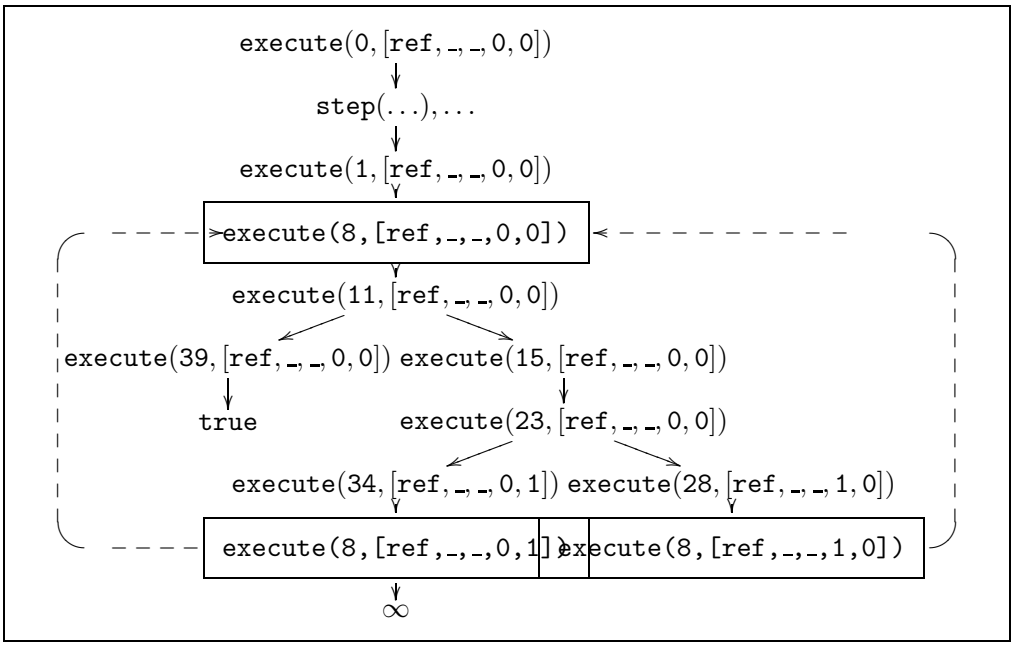


Fig. 5. Partial SLD Tree of Specialization of JVM Interpreter

$\mathtt{s}(\underline{\mathtt{s}}(\underline{\mathtt{U}}+\mathtt{W})\underline{\times}(\underline{\mathtt{U}}+\underline{\mathtt{s}}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U}\times(\mathtt{U}+\mathtt{V}))$. By relying on the embedding relation, the following strategies can be defined (they correspond to those used in [1]):

4.2.1 Local Control

Unfolding operators based on the homeomorphic embedding $\trianglelefteq$, denoted $\mathsf{unfold}_{\trianglelefteq}$, allow the expansion of derivations until reaching an atom which *embeds* some of the previous atoms in its sequence of covering ancestors (see e.g., [20]). The intuition is that reaching larger (or equal) atoms in the same derivation can endanger termination and hence the computation has to be stopped. Furthermore, in order to achieve the required level of aggressiveness it is also required to be able to accurately handle builtin predicates and to safely perform non-leftmost unfolding [2]. However, in the presence of an infinite signature (e.g., integers) as we have in the JVM interpreter, this unfolding rule can lead to non-terminating computations. Consider, for example, a sequence of atoms of the form: $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,0])$, $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,1])$, $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,2])\ldots$, which can grow infinitely and which the homeomorphic embedding does not flag as potentially dangerous. As a result, by considering the usual homeomorphic embedding relation, the second branch of the partial SLD in Figure 5 is not flagged as dangerous and unfolding does not terminate. This is indicated in the figure by the ∞ symbol as continuation of the second branch. A possible relatively straightforward solution for avoiding this nonterminating behavior of unfolding is to use a slight adaptation of the original homeomorphic relation in which any number embeds any other number, denoted $\trianglelefteq_{num}$. Under this relation the atom $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,1])$ embeds $\mathtt{execute}(8,[\mathtt{ref},\_,\_,0,0])$ (and vice-versa). Unfortunately, this modification to the

homeomorphic embedding relation, although it guarantees termination of the partial evaluation process is a too coarse approximation and leads to excessive precision loss. It turns out not to be an acceptable alternative for specialization of our interpreter since in virtually all cases the residual program contains the full interpreter, i.e., we have not been able to eliminate the interpretation layer.

4.2.2 Global Control

The homeomorphic embedding ordering $\trianglelefteq$ can also be used at the global control level within the abstract operator $\texttt{abstract}_{\trianglelefteq}$ in order to decide when to generalize (i.e., to apply the *most specific generalization*) before proceeding to build (possibly partial) SLD trees. Basically, for each new atom A, it checks whether it is larger than (i.e., it embeds) any of the atoms in the set S_i (which contains the atoms in the roots of the partial trees which have already been built). If A does not embed any atom in S_i, it is added to the set; otherwise, the two atoms are generalized by using the *msg* operator. For instance, if we have $\texttt{execute}(8, [\texttt{ref}, \_, \_, 0, 0])$ in S_i and we want to add the atom $\texttt{execute}(8, [\texttt{ref}, \_, \_, 1, 0])$, by using the original homeomorphic embedding relation, no danger is flagged. Thus, in order to guarantee termination at the global control level we also need to modify the relation to be used when infinite signatures (numbers) are considered. By using the modified embedding relation with numbers $\trianglelefteq_{num}$, the latter atom is generalized into $\texttt{execute}(8, [\texttt{ref}, \_, \_, \texttt{X}, 0])$ before being introduced in S_i.

Regarding the *efficiency* of the PE process, it should be noted that the use of control strategies based on embedding introduces a significant overhead, as we need to keep track of the ancestors (see, e.g., [20]) and to perform expensive embedding checks for each of the atom arguments.

5 Partial Evaluation Types for Decompilation

As we have seen in the previous section, in the presence of an infinite signature, like the integers, neither $\trianglelefteq$ nor $\trianglelefteq_{num}$ alone can achieve effective and efficient decompilations. In particular, the use of "$\trianglelefteq$" can be too aggressive in the sense that it leads to too long derivations (even endangering termination), which prevents from a quality decompilation. In contrast, the use of "$\trianglelefteq_{num}$" is definitely too conservative in the sense that stops derivations too early, which causes the loss of essential information to get a quality decompiled program. In this section, we propose to use the *partial evaluation types* of [9] in order to provide additional information to the PE process and improve the results achieved by using previous techniques based on the above embedding orderings. Such additional information is program-dependent and thus, it makes sense to compute it when we are interested in repeatedly partially evaluating a program. This is obviously the case in our approach to decompilation, since we are repeatedly specializing the interpreter w.r.t. different bytecode programs. This information is provided by means of optional partial evaluation types as defined in [9]. They will allow us to give a selective, context-dependent treatment to arguments at PE time. In particular, the following *basic types* are distinguished:

- dyn: which stands for *dynamic*. This type is used to avoid too aggressive strategies. It denotes that the user thinks it is a good idea to *lose* the information stored in the corresponding argument as soon as a discrepancy is found w.r.t. another "similar" atom. Note that unless such information is lost, increased polyvariance is required in order to maintain separate call patterns with the corresponding values of the discrepant information. This results in higher specialization cost and in a larger residual program. An example of an argument which can be marked as *dynamic* is `Loc` (local variables) in our running example.
- f_sig: which stands for *finite signature*. Literally, this means that the number of functors and constant names which may appear is finite. Thus, for arguments of this type, $\trianglelefteq$ guarantees termination. The motivation for considering this type is that it avoids the need for using $\trianglelefteq_{num}$ for arguments which may contain numbers. The user can use this type for those arguments which are guaranteed to contain a *finite* set of numbers only. This is the case, for instance, of the argument `PC` in our example. Though it is natural to use numbers to represent program counters, given a fixed program, the set of instructions is fixed and finite. This is a key observation which is required to obtain the results presented in this paper.
- const: which stands for *constant*. The motivation for introducing this type is just efficiency of the specialization process. Of course, it should only be applied to arguments which we know will always be instantiated to the same value during specialization time. Its usage does not affect the control strategy at all, but it allows avoiding testing the embedding relation over and over again on arguments which never change. This is the case, for instance, of the argument `Program` which remains constant all over the decompilation process.
- term: which stands for *term*. This the the most general type which includes all possible terms, including partially instantiated terms. This is the default type which is assumed unless the user explicitly provides a more precise pe_type. For programs containing arithmetic (such as our JVM interpreter), the default embedding relation we use is $\trianglelefteq_{num}$ since otherwise termination is not guaranteed.

In order to allow the use of the above basic types at any depth within arguments and, also, allow the possibility of having *disjunctive* types with distinctive functors for which we can declare different types, the notion of *partial evaluation types*, pe_type, is defined [9] as a *regular type* [6] combined with the above basic types.

Let us explain the intuition behind the above pe_type's. The first argument of `execute` is `Program`, which is clearly constant because during each partial evaluation there is exactly one fixed program and there is no need to ever generalize this argument. The third argument is the final (output) `State` which is always a variable before the call and thus it can be given the type term. The type of the current `State` in the second argument is disjunctive and we declare it by means of two rules, one for each functor. The first one corresponds to a normal state `st` and the second one to an exception state `stE`. The most relevant points to note are: 1) The types of the heap and the call stack are declared as dyn as we do not mind "losing"

all information about them during partial evaluation when decompiling a method if needed. Intuitively, this is to say that we do not want to generate multiple decompiled versions of a method depending on the state of the heap or the call stack. Instead, as it happens in standard compilation, the decompilation of the method should be independent from the context from which it is called (and hence this information should be ignored). 2) Again, we distinguish two types of `Frame`s for normal (`fr`) and exception behavior (`frE`). The important point here is that both the `PC` and `Method` can be instantiated only to a finite number of values, since given a fixed program, the number of methods and the number of different program counters is finite. Therefore, they can be safely declared as `f_sig`, which prevents from important information loss. Finally, we declare the set of local variables `Loc` and stack positions `Stack` as `dyn` as they threaten termination as we have seen in the example. Note that termination of the partial evaluation requires that the `pe_type`'s provided are safe. For this it is required that any sub(argument) marked as `f_sig` actually has a finite signature.

The importance of `pe_type` declarations is that they can be used at PE time to disregard, to filter or to keep the information available in each argument, as explained above. The embedding relation which makes use of `pe_type` declarations is called *embedding relation with* `pe_type`*'s* and written as $\trianglelefteq_{pt}$ [9]. As the traditional embedding relation, it is used to steer the PE process both at the local and global control by means of the corresponding $\mathsf{unfold}_{\trianglelefteq_{pt}}$ and $\mathsf{abstract}_{\trianglelefteq_{pt}}$ operators, respectively.

6 Reducing Polyvariance in Global Control

In the previous section we have seen how the use of suitable partial evaluation types allows keeping the termination guarantees of $\trianglelefteq_{num}$, both at the local and global control levels, while at the same time being aggressive enough so as to get rid of the interpretation layer.

However, though the decompiled programs thus obtained are acceptable, careful inspection of such residual programs shows that relatively often, *useless specialization* has been performed. At the local control level, performing more unfolding than necessary often results in residual predicates defined by many clauses. At the global control level, trying to be too precise results in producing too many predicates in the residual program.

The question is whether there is any way to take the previous *generalization history* into account when abstracting an atom at the global control. The intuition is to keep track of the information which we have been forced to *forget* during the partial evaluation process and proceed to forget it straight away for all new atoms which are *similar* to the previously handled ones under some criteria. The motivation for doing so is that since it seems likely that we will end up being forced to forgetting such info, the sooner we forget such info, the better, both in terms of specialization times and size of the residual program. We now propose

a technique based on the ideas above, which can be included inside the standard partial evaluation algorithm, by means of an improved abstraction operator. In order to do that, first, we need to give some preliminary definitions.

A term T is a *generalization* of S (or S is an *instance* of T), denoted by $T \leq S$, if $\exists\sigma.\ T\sigma = S$. Two terms T and T' are *variants*, denoted $T \equiv T'$, if both $T \leq T'$ and $T' \leq T$. If T and T' are variants then there exists a renaming ρ such that $T\rho = T'$. A *generalization* of a set of terms $\{T_1, \ldots, T_n\}$ is another term T such that $\exists\sigma_1, \ldots, \sigma_n$ with $T_i = T\sigma_i$, $i = 1, \ldots, n$. A generalization T is the *most specific generalization* (msg) of $\{T_1, \ldots, T_n\}$ if for every other term T' s.t. T' is a generalization of $\{T_1, \ldots, T_n\}$, $T' \leq T$. We also say that two atoms are *homologous*, written as $A \approx B$, if $\mathsf{filter}(A, \mathsf{pe\_type}_A) \equiv \mathsf{filter}(B, \mathsf{pe\_type}_B)$.

Definition 6.1 [HintsTable] We define a HintsTable as a set of pairs of atoms $\langle A, G\rangle$, s.t. $G \leq A$ (i.e., G is a generalization of A).

We refer to these pairs of atoms as *hints* because they provide suggestions on how to *forget* useless information during the abstraction performed at the global control level. Next, we need to define a set of operations over the HintsTable, which will be used later throughout the partial evaluation algorithm both to add and to recover information from the table.

- $\mathsf{addHint}$: HintsTable $\times \langle Atom, Atom\rangle \rightarrow$ HintsTable
$$\mathsf{addHint}(HT, \langle A, G\rangle) = HT \cup \langle A, G\rangle$$
- $\mathsf{applyHint}_{\equiv}$: HintsTable $\times\ Atom \rightarrow Atom$
$$\mathsf{applyHint}_{\equiv}(HT, A) = \mathsf{msg}(Gs \cup A)$$
$$where\ Gs = \{G \mid \langle B, G\rangle \in \text{HintsTable}, A \equiv B\}$$
- $\mathsf{applyHint}_{\approx}$: HintsTable $\times\ Atom \rightarrow Atom$
$$\mathsf{applyHint}_{\approx}(HT, A) = \mathsf{msg}(Gs \cup A)$$
$$where\ Gs = \{G \mid \langle B, G\rangle \in \text{HintsTable}, A \approx B\}$$

Now, we can define the $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}$ operator by relying on the definitions and operators given above.

Definition 6.2 [$\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}$] The abstraction operator $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}$ is defined in terms of the $\mathsf{abstract}_{\trianglelefteq_{pt}}$ operator as follows:

$$\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}(S_i, \mathcal{T}_{calls}, HT) = \mathsf{abstract}_{\trianglelefteq_{pt}}(S_i, \mathcal{AT}_{calls})$$
$$where\ \mathcal{AT}_{calls} = \{H \mid H = \mathsf{applyHint}_{\circledcirc}(HT, A),\ \forall A \in \mathcal{T}_{calls},\ \circledcirc \in \{\equiv, \approx\}\}$$

Let us note that the $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}$ operator definition is parametric w.r.t. "$\circledcirc$", and it represents two different abstraction operators, namely $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\equiv}}$ and $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\approx}}$, depending on which $\mathsf{applyHint}$ operator to use.

After discussing how hints-tables can be exploited during global control, the main question is how exactly we populate such table with the required entries. We propose to simply instrument the $\trianglelefteq_{pt}$ test during partial evaluation in such a way that whenever it flags possible problems between two atoms A and B, i.e., if the

```
main([[ref(loc(1)),num(int(_))],heap([array(B,A)])],[num(int(0))]) :- 0>=B.
main([[ref(loc(1)),num(int(A))],heap([array(B,[num(int(D))|C])])],[E]) :-
        0<B, D\=A, execute([num(int(D))|C],B,A,0,1,F,E).
main([[ref(loc(1)),num(int(A))],heap([array(B,[num(int(A))|C])])],[D]) :-
        0<B, execute([num(int(A))|C],B,A,1,0,E,D).
main([[null,num(int(_))],heap([])],[ref(loc(1))]).

execute(A,B,C,D,E,heap([array(B,A)]),num(int(E))) :- E>=B.
execute(A,B,C,D,E,heap([array(B,A)]),num(int(E))) :- E<B, D\=0.
execute(A,B,C,0,D,E,J) :-
        D<B, 0=<D, L is D+1, nth(L,A,num(int(M))), M\=C,
        N is D+1, execute(A,B,C,0,N,E,J).
execute(A,B,C,0,D,E,J) :-
        D<B, 0=<D, L is D+1, nth(L,A,num(int(C))),
        execute(A,B,C,1,D,E,J).
```

Fig. 6. Decompiled version of the linear search program

relation $A \trianglelefteq_{pt} B$ holds, in addition to returning the value *true*, it also stores the pair $\langle A, msg(A,B)\rangle$ into the hints-table.

Example 6.3 Now, let us consider again the SLD tree in Fig. 5. We start with an empty table of hints $HT = \{\}$. First, in the middle branch, once we reach the embedded atom $\texttt{execute}(8, [\texttt{ref}, \_, \_, 0, 1])$, a new hint will be added to the table by making a call to the addHint operator. Similarly, another hint will be added in the right branch. Thus, after building the first unfolding tree, the table has the following two entries:

$$HT = \left\{ \begin{array}{l} \langle execute(8,[ref,\_,\_,0,1]), execute(8,[ref,\_,\_,0,Y])\rangle, \\ \langle execute(8,[ref,\_,\_,1,0]), execute(8,[ref,\_,\_,X,0])\rangle \end{array} \right\}$$

Once the unfolding process has finished (see the partial evaluation algorithm in section 3) the following call to the abstract operator will be made:

$$\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}(\{\}, \{execute(8,[ref,\_,\_,0,1]), execute(8,[ref,\_,\_,1,0])\}, HT)$$

Now, let us explain the effects of the application of each of the different abstract operators:

- Using $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\equiv}}$. The $\mathsf{applyHint}_{\equiv}$ operator simply returns the corresponding generalized version for each of the atoms. Thus, the standard abstract operator will be called with $\mathsf{abstract}_{\trianglelefteq_{pt}}(\{\}, \{execute(8,[ref,\_,\_,0,Y]), execute(8,[ref,\_,\_,X,0])\})$. Note that, although we keep the same number of different atoms, polyvariance has been potentially reduced as we have generalized a numeric argument, avoiding the possibility of appearing new different versions of the same atom with different numeric values in the corresponding argument.
- Using $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\approx}}$. In this case, polyvariance will be immediately reduced since, as we will see, both atoms will collapse into the same generalized version. This is due to the generalizations between homologous atoms performed inside the $\mathsf{applyHint}_{\approx}$ operator, which will give rise to the following call to the standard abstract operator $\mathsf{abstract}_{\trianglelefteq_{pt}}(\{\}, \{execute(8,[ref,\_,\_,X,Y]), execute(8,[ref,\_,\_,X,Y])\})$

In Fig. 6 we can see the residual code we have obtained taking advantage of the newly introduced techniques, by partial evaluating the JVML interpreter w.r.t. the

Benchmark		$\trianglelefteq$				$\trianglelefteq_{pt}$				Gains	
Name	Size	Tm	Mem	Unf/Eval	Size	Tm	Mem	Unf/Eval	Size	Tm	Size
exp	0.33	1.56	712	1393/227	0.96	0.63	547	1092/187	0.78	2.49	1.23
gcd	0.27	1.19	566	1118/144	0.79	0.48	329	837/110	0.62	2.48	1.26
lcm	0.61	4.15	969	3211/367	2.50	1.39	471	2509/297	2.28	2.98	1.09
combNR	0.33	3.34	1332	2179/287	2.00	0.92	729	1623/216	1.47	3.64	1.36
combR	0.39	5.82	1733	2750/285	2.45	1.47	1203	2131/227	1.78	3.95	1.38
perm	0.28	1.52	562	1099/148	0.85	0.60	321	818/114	0.68	2.53	1.25
add	0.80	29.75	5980	9083/1115	23.15	7.03	3823	6757/830	18.18	4.23	1.27
exp	0.41	8.44	2027	3570/559	4.57	1.22	1079	2444/382	3.16	6.92	1.45
simplify	0.70	14.60	3076	6205/897	8.70	2.87	1917	4774/697	7.26	5.08	1.20
binarySrch	0.42	38.80	9867	10740/1571	29.91	6.00	3361	4837/727	11.53	6.46	2.59
forward	0.60	62.87	4106	14714/2256	16.30	9.20	4108	14714/2256	16.30	6.83	1.00
fib	0.28	—	—	–/–	—	0.64	338	1421/191	1.10	∞	∞
linearSrch	0.32	—	—	–/–	—	1.80	478	2610/394	16.09	∞	∞
signs	0.33	—	—	–/–	—	3.98	1052	4401/702	11.40	∞	∞

Table 1
Measuring the effects of the pe_types

bytecode program of our running example (see Fig. 2). Thus, we have used the $\trianglelefteq_{pt}$ as embedding relation (instrumented to add hints when embedding is flagged) and the $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\approx}}$ operator. Note that the entry call is `main(In,Out)`, where `In` will be instantiated to the list of argument values specified for the method, together with the input heap, and `Out` will be instantiated to the top of the stack at the end of the execution. This `main` predicate is responsible for first obtaining the initial state and the JVML program and then calling for the first time to the `execute` predicate of the interpreter (represented in the SLD tree in Fig. 5).

In the residual code, we see four rules for predicate `main`, three of them correspond to the three branches represented in the SLD tree, and the fourth one represents the trivial case where the input array is `null` (which, for simplicity, is not represented in the SLD tree). As it can be seen, we have successfully got rid of the interpretation layer as we only have calls to: 1) arithmetic builtins, 2) list builtins (`nth` in this case for accessing the contents of the array) and 3) recursive calls to the `execute` predicate, which represents, in essence, recursive calls to the basic blocks in the control flow graph of the bytecode program.

7 Experimental Results

Table 1 shows the benefits that we can obtain by using pe_type's. We use a set of classical algorithms as benchmarks. We have benchmarks belonging to iterative programs without object-oriented features, thus, **exp**, **gcd**, **lcm** and **fib** compute respectively the exponential, greatest-common-divisor, least-common-multiple and Fibonacci; while **combNoRep**, **CombRep** and **perm** are methods for computing different combinatorial functions. Also, we have some benchmarks using integer arrays, such as **linearSearch** and **binarySearch** which implement the classic linear and binary search over an array; and **Signs** which given an integer array, computes

Benchmark	$abstract_{\trianglelefteq_{pt}+gen_{\equiv}}$				$abstract_{\trianglelefteq_{pt}+gen_{\approx}}$			
Name	**Tm**	**Mem**	**Unf/Eval**	**Size**	**Tm**	**Mem**	**Unf/Eval**	**Size**
lcm	1.38	1.00	1.46/1.43	1.79	1.41	1.00	1.46/1.43	1.79
add	1.50	1.00	1.56/1.56	1.42	1.25	1.00	1.56/1.56	1.42
simplify	1.37	1.00	1.47/1.46	1.44	1.35	1.00	1.47/1.46	1.44
binarySearch	0.99	1.00	1.00/1.00	1.00	1.10	1.00	1.26/1.22	1.24
linearSearch	1.25	0.98	1.28/1.28	1.11	1.49	0.98	1.80/1.81	4.58
signs	1.24	1.00	1.30/1.30	1.28	1.86	1.00	1.88/1.90	2.15

Table 2
Measuring the effects of the $\mathsf{abstract}_{\trianglelefteq_{pt}+gen}$

the number of pairs of numbers with different sign. Finally, we have used four benchmarks which make extensive use of object-oriented features such as instance method invocation, field accessing and setting, object creation and initialization, etc. Thus, **add**, **exp** and **simp** compute different operations over rational numbers (represented as objects), while **forward** is invoked over an object representing a date and forwards one day.

For each benchmark, the column **Name** shows the name of the method which is the starting point for the decompilation, and the column **Size** shows its size. All sizes are in KBytes and execution times in seconds. The next four columns, labeled $\trianglelefteq$, provide information about specialization using the original homeomorphic embedding. The first three of them show some data about the specialization process, whereas the fourth one shows the **Size** of the residual program. The aspects which have been measured for the specialization process are **Tm**, which is the time required by partial evaluation, **Mem** which is its memory consumption, and **Unf/Eval** which shows the number of derivation steps together with the number of evaluations steps (i.e., where an `eval` assertion has been applied, see[20]) performed during the partial evaluation process. Similarly, the next four columns provide information about specialization using our proposed combination of embedding with `pe_type`'s. Finally, the last two columns show the gains (in terms of time and size) we obtain with the new embedding definition $\trianglelefteq_{pt}$ based on `pe_type`'s and it is computed as *Old-Cost*/*New-Cost*. The last three benchmarks do not present data for the $\trianglelefteq$ columns because the partial evaluation process does not terminate for them. As it can be seen in the table, our proposed $\trianglelefteq_{pt}$ specialization is able to handle them. It can also be seen that for all other programs, the use of $\trianglelefteq_{pt}$ results in important gains both in terms of time and size. In terms of time, they range from 2.49 in `exp` to 6.83 times faster in the case of `forward`. The gains in terms of size range from obtaining a similar sized program in `forward` to a program 2.59 times smaller in the case of `binarySearch`.

The goal of Table 2 is to study the practical benefits that can be obtained by using the new abstraction operator $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\circledcirc}}$ proposed in Section 6. As in Table 1, for each specialization approach we show four columns, with the same meaning as before. However, in this case, rather than the absolute data we show just the gains obtained w.r.t. the behavior of $\trianglelefteq_{pt}$, which is shown in absolute

terms in Table 1. We have two groups of columns, labeled as $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\equiv}}$ and $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\approx}}$, each of them shows the gains of using respectively such abstraction operator when compared to using $\trianglelefteq_{pt}$.

As it can be seen, the new global control never introduces relevant overhead. Furthermore, in most cases it introduces relevant speedups, which go as high as 1.5 for the case of $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\equiv}}$ and 1.86 in the case of $\mathsf{abstract}_{\trianglelefteq_{pt}+gen_{\approx}}$.

8 Conclusions

In this paper we have studied new mechanisms for achieving "quality" decompilation from Java Bytecode to Prolog while at the same time ensuring termination of the partial evaluation process by using a state-of-the-art *online* partial evaluator. In addition to improving the quality of the residual programs, the techniques we propose provide important efficiency gains during partial evaluation. In particular, we use *partial evaluation types* to provide safe approximations of the values which the arguments of predicates can take during partial evaluation time. Such partial evaluation types are then used by the partial evaluator in order to steer the specialization process, both at the local and global control levels. Besides, we present novel techniques to control the *polyvariance* of the PE process, i.e., to avoid having too many (redundant) specialized versions of some predicates. As we have showed in our experiments, both proposals improve not only the effectiveness but also the efficiency of the decompilation process which, at the same time, widens the class of programs that can be handled by using our interpretative approach. It remains as future work to improve the precision of our techniques to achieve effective decompilation of recursive procedures [7]. To do this, we plan to use more advanced PE techniques [21] which integrate abstract interpretation.

Acknowledgement

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

[1] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Proc. PADL*, LNCS. Springer-Verlag, 2007. To appear.

[2] E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *Proc. of LOPSTR'05*. Springer LNCS 3901, April 2006.

[3] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.13). Technical report, School of Computer Science (UPM), 2006. Available at http://www.ciaohome.org.

[4] G. Puebla et al. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, 3049 in LNCS, pages 234–261. August 2004.

[5] Yoshihiko Futamura. Program evaluation and generalized partial computation. In *International Conference on Fifth Generation Computer Systems - Proceedings*, pages 1–8, Tokyo, Japan, 1988.

[6] J. Gallagher and D. de Waal. Fast and Precise Regular Approximations of Logic Programs. In *Proc. of ICLP'94*, pages 599–613. MIT Press, 1994.

[7] J. P. Gallagher and J. C. Peralta. Regular tree languages as an abstract domain in program specialisation. *HOSC*, 14(2,3):143–172, 2001.

[8] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98. ACM Press, 1993.

[9] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Partial Evaluation Types for Improving the Decompilation of Java Bytecode to Prolog. Technical Report CLIP1/2007.0, School of Computer Science, UPM, February 2007.

[10] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.

[11] N. D. Jones. Partial evaluation, self-application and types. In *Proc. of ICALP'90*, volume 443 of *LNCS*, pages 639–659. Springer, 1990.

[12] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.

[13] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[14] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[15] M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 340–375. Springer, 2004.

[16] Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[17] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[18] J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd Ext. Ed., 1987.

[19] D. Pichardie. Bicolano (Byte Code Language in cOq). http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html.

[20] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *Proc. of LOPSTR'04*, pages 149–165. Springer LNCS 3573, 2005.

[21] G. Puebla, E. Albert, and M. Hermenegildo. Abstract Interpretation with Specialized Definitions. In *Proc. of SAS'06*, number 4134 in LNCS. Springer, 2006.

[22] A. Takeuchi and K. Furukawa. Partial evaluation of prolog programs and its application to meta programming. In *Proc. IFIP '86*, pages 415–420. North-Holland, 1986.

[23] W. Vanhoof, M. Bruynooghe, and M. Leuschel. Binding-time analysis for mercury. In *Program Development in Computational Logic*, volume 3049 of *LNCS*, pages 189–232. Springer, 2004.

Type-Based Homeomorphic Embedding and Its Applications to Online Partial Evaluation

Elvira Albert[1], John Gallagher[2], Miguel Gómez-Zamalloa[1], and Germán Puebla[3]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CBIT, Roskilde University, DK-4000 Roskilde, Denmark
[3] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. *Homeomorphic Embedding* (HEm) has proven to be very powerful for supervising termination of computations, provided that such computations are performed over a *finite signature*, i.e., the number of constants and function symbols involved is finite. However, there are situations, for example numeric computations, which involve an infinite (or too large) signature, in which HEm does not guarantee termination. Some extensions to HEm for the case of infinite signatures have been proposed which guarantee termination, but they either do not provide systematic means for generating such extensions or the extensions are too simplistic and do not produce the expected results in practice. We introduce *Type-based Homeomorphic Embedding* (TbHEm) as an extension of the standard, untyped HEm to deal with infinite signatures. In the paper, we show how TbHEm can be used to improve the accuracy of *online partial evaluation*. For this purpose, we propose an approach to constructing suitable types for partial evaluation automatically based on existing analysis tools for constraint logic programs. We also present useful properties of types which allow us to take full advantage of TbHEm in practice. Experimental results are reported which show that our work improves the state of the practice of online partial evaluation.

1 Introduction

The *homeomorphic embedding* (HEm) relation [10,11,12] has become very popular to ensure online termination of *symbolic* transformation and specialization methods and it is essential to obtain powerful optimizations, for instance, in the context of online Partial Evaluation (PE) [9]. Intuitively, HEm is a structural ordering under which an expression t_1 is greater than, i.e., it *embeds*, another expression t_2, written as $t_2 \trianglelefteq t_1$, if t_2 can be obtained from t_1 by deleting some parts, e.g., $\underline{\mathtt{s}}(\mathtt{s}(\underline{\mathtt{U}} + \mathtt{W}) \underline{\times} (\underline{\mathtt{U}} \underline{+} \mathtt{s}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U} \times (\mathtt{U} + \mathtt{V}))$. The HEm relation can be used to guarantee termination because, provided the set of constants and functors is finite, every infinite sequence of expressions $t_1, t_2, \ldots$, contains at least a pair of elements t_i and t_j with $i < j$ s.t. $t_i \trianglelefteq t_j$. Therefore, when iteratively computing a sequence $t_1, t_2, \ldots, t_n$, finiteness of the sequence can be guaranteed by using HEm as a *whistle*. Whenever a new expression t_{n+1} is to

King, A. (Ed.): LOPSTR 2007, LNCS 4915, pp. 23–42, 2008.

be added to a finite sequence $t_1, \ldots, t_n$, we first check whether t_{n+1} embeds any of the expressions already in the sequence. If that is the case, we say that HEm whistles, i.e., it has detected (potential) non-termination and the computation has to be stopped. Otherwise, t_{n+1} can be safely added to the sequence and the computation can proceed.

Two key features for the success of HEm as an approach for guaranteeing online termination are i) in the case of finite sequences, it often allows sequences to grow considerably large before the whistle blows, to the point that in a good number of cases the full sequence can be computed without the whistle blowing at all; ii) in the case of infinite sequences, it often identifies (potential) non-termination quickly, and the whistle blows without unnecessarily further expanding the sequence.

While HEm has been proved very powerful for symbolic computations, some difficulties remain in the presence of infinite signatures, such as the numbers. In the case of logic programs, infinite signatures appear as soon as certain Prolog built-ins such `is/2`, `functor/3 name/2`, `=../2`, `atom_codes/2`, etc. are used. Some extensions to HEm over infinite signatures have been defined and used in practice (e.g. [11,2]), but they are often too ad-hoc, i.e., they only allow constants which appear explicitly in the program, regardless of which part of the program (predicate, argument position) they appear. As the approach is purely *syntactic*, it sometimes turns out to be too conservative ("whistling" too early) in practice, breaking feature i) above; while it can also be too aggressive, thus also sometimes breaking feature ii) above.

In this paper, we introduce the *type-based homeomorphic embedding* (TbHEm) relation which by taking information about the behavior of the program into account, provides more precise results in the presence of infinite signatures. In a sense, whereas [11,2] take a simple *syntactic* approach to extending the HEm relation, we propose a *semantic* approach for such extension. To achieve this, our typed relation is defined on types structured in two parts: a finite component and an infinite component. Intuitively, TbHEm allows expanding sequences as long as, whenever we compare two terms of a given type, the actual symbols which appear in such terms belong to the finite component of the type.

We illustrate the benefits of TbHEm in the context of online Partial Evaluation (PE) [9]. In particular, we use a simplified interpreter for an imperative, stack-based bytecode language written in Prolog whose specialization (if successful) allows decompiling bytecode programs to Prolog. We show how to automatically construct typings by relying on existing analysis techniques for the inference of well-typings [5]. Moreover, we present the property of a type being of *finite signature* (resp. *infinite signature*) which guarantees that all terms in the type are built out of a finite (resp. infinite) number of constant and functor symbols. We also outline how analysis of numeric bounds can be used to infer finite signature properties of types. In the case of finite signature, we can safely apply traditional HEm. We report on experimental results which compare TbHEm with previous proposals and show the benefits of our approach for the specialization of logic programs with infinite signatures.

The rest of the paper is organized as follows. Sect. 2 recalls some basic notions of PE, with special emphasis on the role of embedding. In Sect. 3, we review existing proposals in specialization of interpreters. In Sect. 4, we introduce TbHEm and prove its correctness. Sect. 5 proposes the use of well-typings as suitable types for the application of TbHEm in online PE and reports some experiments. Sect. 6 presents interesting properties of types to use TbHEm in practice, together with some experimental results. Finally, Sect. 7 discusses related work and concludes.

2 Basics on Embedding in Partial Evaluation

We assume familiarity with the basic concepts of logic programming and partial evaluation, as they are presented in e.g. [16,9]. We start by recalling the definition of HEm, which can be found for instance in Leuschel's work [14].

Definition 1 ($\trianglelefteq$). *Given two atoms $A = p(t_1, \ldots, t_n)$ and $B = p(s_1, \ldots, s_n)$, we say that B* embeds *A, written $A \trianglelefteq B$, if $t_i \trianglelefteq s_i$ for all i s.t. $1 \leq i \leq n$. The embedding relation over terms, also written $\trianglelefteq$, is defined by the following rules:*

1. *$Y \trianglelefteq X$ for all variables X, Y.*
2. *$s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i.*
3. *$f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $s_i \trianglelefteq t_i$ for all i, $1 \leq i \leq n$.*

We now explain the role that HEm plays in online PE (see e.g. [9,12,14]), which is a semantics-based program transformation technique which specializes a program w.r.t. given input data, hence, it is often called program specialization. Essentially, partial evaluators are non-standard interpreters which evaluate goals as long as termination is guaranteed and specialization is considered profitable. Given a program P and an atom S, partial evaluation produces a new program P_S which is a specialization of P for S. In logic programming, the underlying technique is to construct (possibly) *incomplete* SLD trees for the set of atoms to be specialized. In an incomplete tree, it is possible to choose *not* to further unfold a goal. Therefore, the tree may contain three kinds of leaves: failure nodes, success nodes (which contain the empty goal), and non-empty goals which are not further unfolded. The latter are required in order to guarantee termination of the partial evaluation process, since the SLD being built may be infinite. Even if the SLD trees for fully instantiated initial atoms (as regards the *input* arguments) are finite, the SLD trees produced for partially instantiated initial atoms may be infinite. This is because the SLD for partially instantiated atoms can have (infinitely many) more branches than the actual SLD tree at run-time.

HEm *in local control.* The role of local control is to determine how to construct the (incomplete) SLD trees. In particular, the *unfolding rule* decides, for each resolvent, whether to stop unfolding or to continue unfolding it and, if so, which atom to select from the resolvent. Unfolding is continued only if termination is not endangered and specialization is considered profitable. Therefore, it is

desirable to have a mechanism for guaranteeing termination which *whistles* as late as possible. State of the art local control rules based on HEm do not check for embedding against all previously selected atoms but rather only against those in its sequence of *covering ancestors* (see e.g., [18]). This increases both the efficiency of the checking and whistling later.

HEm *in global control.* Partial evaluators need to compute SLD-trees for a number of atoms in order to ensure that all atoms which appear in non-failing leaves of incomplete SLD trees are "covered" by the root of some tree (this is known as the closedness condition of partial evaluation [15]). The role of the *global control* is to ensure that we do not try to compute SLD trees for an infinite number of atoms. The usual way of achieving this is by applying an *abstraction operator* which performs "generalizations" on the atoms for which SLD trees are to be built. HEm can also be used at the global control level in order to decide when to generalize (i.e., to apply the *most specific generalization*) before proceeding to build SLD trees. Basically, for each new atom A, global control checks whether A is larger than (i.e., it embeds) any of the atoms in the set T_i (which contains the atoms in the roots of the partial trees which have already been built). If A does not embed any atom in T_i, it is added to the set; otherwise, A is generalized into $msg(A, A')$, where $A' \in T_i$ and $A' \trianglelefteq A$. At the global control level, HEm can be combined with other techniques such as *global trees*, *characteristic trees*, *trace terms*, etc. See e.g. [12] and its references.

Partial evaluation and Code Generation. As discussed above, the global control returns a set of atoms T. Finally, a partial evaluation of P w.r.t. S can then be systematically extracted from the set T. As notation, we refer to each root-to-leaf path in an SLD tree as *derivation*. The notion of *resultant* is used to generate a program rule associated with each non-failing derivation in an SLD tree. In particular, given a derivation for $P \cup \{A\}$ with $A \in T$ ending in B and θ the composition of the *mgu*s in the derivation steps, then the rule $A\theta \leftarrow B$ is called the *resultant* of the derivation. A *partial evaluation* is then defined as the union of the sets of resultants associated to the SLD trees for all atoms in T.

3 Embedding with Infinite Signatures: Motivating Example

In Fig. 1 we show a fragment of a simplified imperative bytecode interpreter implemented in Prolog. If the partial evaluator is powerful enough, given a bytecode program we can obtain a decompiled version of it in Prolog (see e.g. [1] for an object-oriented stack-based interpreter). For brevity, we omit the code of some predicates like `build_init_state/2` (whose purpose is explained below) and `localVar_update/4` which simply updates the value of a local variable. We only show the definition of `step/3` for a reduced set of instructions. The bytecode to be decompiled is represented as a set of facts `bytecode(PC,Inst)` where `PC` contains a program counter and `Inst` the corresponding bytecode instruction. A state is of the form `st(PC,OStack,LocalV)` where `OStack` represents

```
main(InArgs,Top) :-
    build_init_state(InArgs,S0),
    execute(S0,st(_,[Top|_],_)).
execute(S,S):-
    S = st(PC,_,_),
    bytecode(PC,return).
execute(S1,Sf) :-
    S1 = st(PC,_,_),
    bytecode(PC,Inst),
    step(Inst,S1,S2),
    execute(S2,Sf).
```

```
step(const(_T,Z),st(PC,S,L),S2) :-
    PCp is PC + 1,
    S2 = st(PCp,[Z|S],L).
step(istore(X),st(PC,[I|S],L),S2) :-
    PCp is PC + 1,
    localVar_update(L,X,I,Lb),
    S2 = st(PCp,S,Lb).
step(goto(O),st(PC,S,L),S2) :-
    PCp is PC+O,
    S2 = st(PCp,S,L).
....
```

Fig. 1. Fragment of simplified bytecode interpreter

the operand stack and `LocalV` the list of local variables. The predicate `main/2`, given the input method arguments `InArgs`, first builds the initial state by means of predicate `build_init_state/2` and then calls predicate `execute/2`. In turn, `execute/2` first calls predicate `step/3`, which produces `S2`, the state after executing the corresponding bytecode, and then calls predicate `execute/2` recursively with `S2` until we reach a `return` instruction.

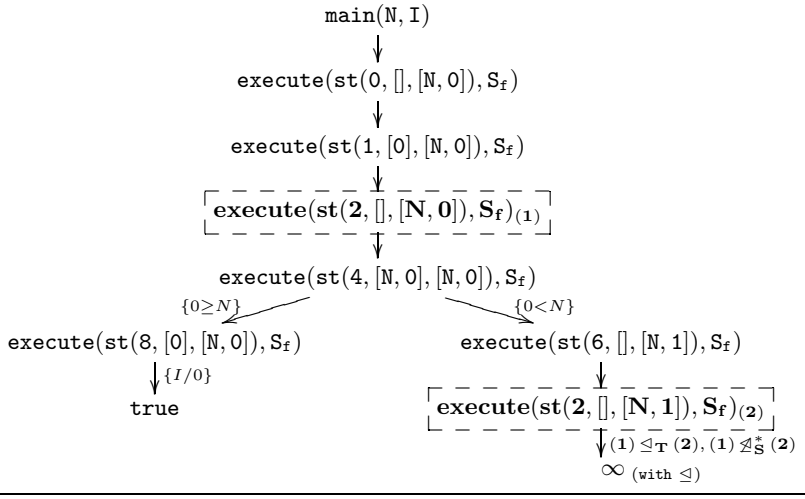
Consider the `count` method which appears in the left hand side of Fig. 2, represented as a set of facts. For clarity of the presentation, on the right hand side of Fig. 2 we show a Java source program which can be compiled into the corresponding bytecode. However, it is important to note that the decompilation is performed directly from the bytecode and that the decompiler does not have access to the source. It can be seen that `count` receives an integer and executes a loop where a counter initialized to "0" (in bytecodes `0` and `1`) is incremented by one at each iteration (bytecode `5`) until the counter reaches the value of the input parameter (checking the condition comprises bytecodes `2`, `3` and `4`). The method returns the value of the counter in bytecodes `7` and `8`. For decompiling the `count` method, we partially evaluate the interpreter w.r.t. the bytecode facts which appear to the left of the figure by specializing the atom: `main(N,I)`, where `N` is the input parameter and `I` represents the return value (i.e., the top of the stack at the end of the computation).

In Figure 3, we depict (a reduced version of) one of the SLD trees that leads to an effective decompilation of our running example and that we will refer to in the next sections. For simplicity, apart from the entry atom `main/2`, we only show atoms for `execute/2`, as it is the only recursive predicate in the program. Thus, each arrow in the tree involves the application of several unfolding steps. Note that some of the statements within the body of each `step` operation can remain residual when they involve data which is not known at specialization time. The computation rule used in the unfolding operator is able to residualize calls which are not sufficiently instantiated and select non-leftmost atoms in a safe way [3], in particular, further calls to `execute` can be selected. We represent such residual calls as labels in the arrows of the tree.

```
bytecode(0,const(int,0)).
bytecode(1,istore(1)).
bytecode(2,iload(1)).
bytecode(3,iload(0)).
bytecode(4,if_icmp(geInt,3)).
bytecode(5,iinc(1,1)).
bytecode(6,goto(-4)).
bytecode(7,iload(1)).
bytecode(8,return).
```

```
static int count(int n){
  int i = 0;
  while (i < n)
    i++;
  return i;
}
```

Fig. 2. Object program for working example




```
main(N,0) :- 0>=N.
main(N,I) :- 0<N,
             sp_execute(N,1,I).
```

```
sp_execute(N,I,I) :- I>=N.
sp_execute(N,A,I) :- A<N, A' is A+1,
                     sp_execute(N,A',I).
```

Fig. 3. Partial unfolding SLD tree and residual code of working example

3.1 Using the Original Homeomorphic Embedding

Let us first consider an online partial evaluator (which is able to accurately handle built-in predicates and to safely perform non-leftmost and) which uses HEm to control termination both at the local and global control levels. As it can be seen in the figure, the PC value "2" corresponds to the loop entry. By applying HEm, the evaluation contains a subsequence of atoms of the form: $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},2]),\texttt{S}_\texttt{f}),\ldots$ marked within dashed frames in the figure, which correspond to consecutive iterations of the loop in which the control returns to the loop head (PC value 2 in the first position of the state) with a value for the loop counter

(local variable at the second position in the resulting state) increased by one. This sequence can grow infinitely, as the HEm does not flag it as potentially dangerous, which is marked by ∞ (with $\trianglelefteq$) in the figure. This is because the interpreter uses Prolog's arithmetic (i.e., the `is/2` predicate), which breaks the finite signature property featured by pure logic programs.

In order to get a quality decompilation, we need to filter out the value of the counter (local variable 1) but not that of the PC. As shown in the figure, this requires stopping the derivation when we hit the atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ (marked as $(\mathbf{1}) \trianglelefteq_{\mathbf{T}} (\mathbf{2})$) and generalize it w.r.t. the above atom within a dashed frame, resulting in $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},\texttt{X}]),\texttt{S}_\texttt{f})$.

3.2 Recovering Termination: Embedding with Number Filtering

In programs which contain Prolog arithmetic but do not generate an infinite number of functors via `functor/3`, `=../2`, etc., a relatively straightforward solution in order to recover termination is to use the $\trianglelefteq_{num}$ relation, which is an adaptation of HEm which filters out numeric values, i.e., any number embeds any other number. The atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ embeds $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$ under $\trianglelefteq_{num}$ and therefore we avoid non-termination. Unfortunately, this modification to HEm, is far too conservative, and leads to excessive precision loss. For instance, in the specialization of $\texttt{main}(\texttt{N},\texttt{I})$, the first two atoms for `execute/2` are $\texttt{execute}(\texttt{st}(0,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$ and $\texttt{execute}(\texttt{st}(1,[0],[\texttt{N},0]),\texttt{S}_\texttt{f})$. By using $\trianglelefteq_{num}$, the whistle blows at this point and unfolding has to stop. Furthermore, the latter atom is generalized at the global control level into $\texttt{execute}(\texttt{st}(\texttt{X},\texttt{Y},[\texttt{N},0]),\texttt{S}_\texttt{f})$ before proceeding with the specialization. This turns out not to be acceptable for the specialization of our interpreter, since we lose track of which the next instruction to execute is—which prevents us from eliminating the interpretation layer—and in many cases the residual program ends up containing the whole original interpreter.

3.3 Increasing Accuracy: Static Symbols in the Program

A simple syntactic way of increasing the accuracy while preserving termination, as proposed in [11], consists in considering two sets of symbols: those which appear explicitly in the program and goal, which is obviously finite, and another infinite set which contains all other symbols. In the following, this relation is denoted as $\trianglelefteq^*_S$. When comparing two terms we keep those symbols which belong to the finite set and filter out all other ones. Under this relation, the atom $\texttt{execute}(\texttt{st}(1,[0],[\texttt{N},0]),\texttt{S}_\texttt{f})$ does not embed the atom $\texttt{execute}(\texttt{st}(0,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$ in the figure, as the numbers `0` and `1` are different static symbols which occur in the program. Hence, we are not forced to generalize them and we can keep the `PC` value.

Unfortunately, the $\trianglelefteq^*_S$ relation turns out not to be optimal in our case either since $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$. This means that unfolding proceeds with a second iteration of the loop. The process is guaranteed to terminate, we will unfold at most as many iterations of the loop as distinct numbers appear in the program. However, we are not able to

achieve the quality decompilation which appears at the bottom of Figure 3. For obtaining such good decompilation, we need to generalize the loop counter, i.e., the atom $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},1]),\texttt{S}_\texttt{f})$ has to embed $\texttt{execute}(\texttt{st}(2,[],[\texttt{N},0]),\texttt{S}_\texttt{f})$. Intuitively, the reason why this relation does not behave optimally is because the fact that many symbols appear explicitly in the program for one argument (in our case the PC counter) should somehow not affect the set of symbols which we should consider as static for other arguments (the list of local variables).

Note that the use of characteristic trees [13] to control the degree of polyvariance does not lead to an optimal decompilation in this example either. The reason is that characteristic trees concern only global and not local control. Therefore, as already mentioned above, they do not stop the local derivation which may perform as many unrollings of the loop as different values for the loop counter there are in the program. Once the local control stops this unfolding process, the value of the counter will be generalized by the global control. However, the characteristic tree of this generalized term is clearly not equivalent to the one of the previous unrolling for the different values in the counter. Therefore, the decompilation of the loop body for the static values remains residual in the specialized code as well.

4 Type-Based Homeomorphic Embedding

In the presence of infinite signatures, a general method of defining homeomorphic embedding relations exists; an *extended homeomorphic embedding relation* is defined in [11] based on previous results by Kruskal [10] and by Dershowitz [6]. This solution defines a family of embedding relations, where a subsidiary ordering on function symbols plays an essential role. However, we argue that this does not really solve the practical problem of finding an effective embedding relation, since there is no automated mechanism for finding the "right" ordering relation on the function symbols in the signature.

In this section, we propose *typed-based homeomorphic embedding* (TbHEm for short), a relation which improves HEm by making use of additional information provided in the form of types. We outline how this approach can be seen as a way of generating instances of extended HEm as defined by Leuschel, including the possibility of taking into account the program semantics. The types required for guiding TbHEm can be provided manually or, interestingly, be automatically inferred by program analysis, as we will see in Section 5.

4.1 Types: Preliminaries and Notation

In the following, let P be a program and Σ_P be a (possibly infinite) signature including the functions and constants appearing in P and goals for P as well as in computations of P. We adopt the syntax of Mercury [20] for type definitions. *Type expressions (types)*, elements of $\mathcal{T}$, are constructed from an infinite set of type variables (parameters) $\mathcal{V}_\mathcal{T}$ and an alphabet of ranked type symbols $\Sigma_\mathcal{T}$; these are disjoint from the set of variables V and the alphabet of functors Σ_P of a given program P respectively.

Definition 2 (type definition). *A* type rule *for a type symbol $h/n \in \Sigma_{\mathcal{T}}$ is of the form $h(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots ; f_k(\bar{\tau}_k); \ldots$ $(k \geq 1)$ where $\bar{T}$ is a n-tuple of distinct type variables, $f_1, \ldots, f_k, \ldots$ are distinct function symbols from Σ_P, $\bar{\tau}_i$ $(i \geq 1)$ are tuples of corresponding arity from $\mathcal{T}$, and type variables in the right hand side, if any, are from $\bar{T}$ (a condition known as* transparency *[17,8]). A* type definition *is a finite set of type rules where no two rules contain the same type symbol on the left hand side, and there is a rule for each type symbol occurring in the type rules.*

We write $t : \tau$ to mean that term t is of type τ. As in Mercury [20], a function symbol can occur in several type rules. In the definition above we allow type rules containing an infinite number of cases. Thus, standard infinite types such as *integer* are permitted, defined by a rule with an infinite number of cases containing the numeric constants. In order to define TbHEm we introduce some extra annotation into type rules. We consider the right hand side of each type rule to consist of two disjoint components, each possibly empty. More precisely, we will structure a type rule as $h(\bar{T}) \longrightarrow F; I$, where the union $F \cup I$ are the cases in the type rule, $F \cup I$ is non-empty, F is either empty or finite and I is either empty or infinite. We say that a type $\tau \in \mathcal{T}$ is of *infinite component* if I is non-empty in the rule defining τ. Otherwise it is said to be of *finite component*. Note that for types of infinite component there are infinitely many ways of splitting them into type rules; for example $nat \longrightarrow F; I$ where $F = \emptyset$ and $I = \mathbb{N}$, or $F = \{0, 1, 2\}$ and $I = \mathbb{N} \setminus \{0, 1, 2\}$, etc.

A *predicate signature* for an n-ary predicate p is of the form $p(\bar{\tau})$ and declares a type $\tau_i \in \mathcal{T}$ for each argument of the predicate p/n. Programs are assumed to be *well-typed* in the usual sense, namely that every atom and term in a clause can be assigned types consistent with the type declarations such that the type assigned to each head atom is a variant of the signature for its predicate, the types of the body atoms are instances of the corresponding signatures, and multiple occurrences of the same variable in the clause are assigned the same type. Furthermore, we disallow *polymorphic recursion*; body atoms for recursive predicates are assigned a type that is a variant of the signature. The relevant consequences of well-typing for our purpose are firstly that a well-typed program and goal generate only well-typed atoms in computations and secondly that only a finite number of types arise during a computation. An infinite set of different types such as $h(T), h(h(T)), h(h(h(T))), \ldots$ cannot arise in a computation, due to the absence of polymorphic recursion.

4.2 Type-Based Homeomorphic Embedding

We now define TbHEm ($\trianglelefteq_T$). It follows closely the definition of the extended HEm relation defined in [11] on untyped terms; here we define a relation on typed terms. As in the definition in [11], two subsidiary relations $\preceq_F$ and $\preceq_S$ are needed. The first, $\preceq_F$, is a relation on function symbols paired with their associated types, and it refers to the infinite component of type rules described above.

Definition 3. *Let $\preceq_F$ be the following relation on the set of pairs $\Sigma_P \times \mathcal{T}$. $(f_1, \tau_1) \preceq_F (f_2, \tau_2)$ iff (1) the rules defining τ_i are of form $h_i(\bar{V}_i) \longrightarrow F_i; I_i$, for $i = 1, 2$ and (2) either $f_1 = f_2 \wedge \tau_1 = \tau_2$ or f_2 is in the infinite component I_2 of the rule for τ_2.*

For instance, given $\tau \longrightarrow F; I$ with $F = \{1, 2\}$ and $I = \mathbb{N} \setminus \{1, 2\}$ then $(1, \tau) \not\preceq_F (2, \tau)$ and $(1, \tau) \preceq_F (5, \tau)$. The other relation, $\preceq_S$, is a relation on sequences of typed terms, and for our purposes here we can take it to be true for all pairs of sequences of typed terms. In general this relation can be defined to allow more refined treatment of associative operators, among other things; as noted in [11], whether $\wedge(a, b, c)$ is embedded in $\wedge(a, b, c, d)$ depends on the nested structure of the expressions, if $\wedge$ is taken as a binary functor. Though we do not use it here, we include the relation $\preceq_S$ in the following definition for uniformity with [11], so that our notion of typed embedding becomes an instance of the extended homeomorphic embedded defined there.

Definition 4 ($\trianglelefteq_T$). *Given two typed atoms $A = p(t_1, \ldots, t_n)$ and $B = p(s_1, \ldots, s_n)$, with predicate signature $p(\tau_1, \ldots, \tau_n)$, we say that B* embeds *A, written $A \trianglelefteq_T B$, if $t_i : \tau_i \trianglelefteq_T s_i : \tau_i$ for all i s.t. $1 \leq i \leq n$. The embedding relation over typed terms, also written $\trianglelefteq_T$, is defined by the following rules:*

1. *$Y : \tau_Y \trianglelefteq_T X : \tau_X$ for all variables X, Y.*
2. *$s : \tau \trianglelefteq_T f(t_1, \ldots, t_n) : \tau'$ if $s : \tau \trianglelefteq_T t_i : \tau'_i$ for some i, where $\tau'_1, \ldots, \tau'_n$ are the respective types of $t_1, \ldots, t_n$.*
3. *$f(s_1, \ldots, s_n) : \tau \trianglelefteq_T g(t_1, \ldots, t_m) : \tau'$ if*
   (a) $(f, \tau) \preceq_F (g, \tau')$,
   (b) $(s_1 : \tau_1, \ldots, s_n : \tau_n) \preceq_S (t_1 : \tau'_1, \ldots, t_m : \tau'_m)$, and
   (c) $\exists i_1, \ldots, i_n$ such that $1 \leq i_1 < \cdots < i_n \leq m$ and $\forall j \in \{1, \ldots, n\}$, $s_j : \tau_j \trianglelefteq_T t_{i_j} : \tau'_{i_j}$,
   where $\tau_1, \ldots, \tau_n, \tau'_1, \ldots, \tau'_m$ are the respective types of $s_1, \ldots s_n, t_1, \ldots, t_m$.

Rule 3 of the definition specifies that embedding can occur between terms with different function symbols, where the function symbol of the "larger" term using the $\preceq_F$ relation is from the I component of its type. However, as long as we compare distinct terms from an infinite type and remain within the finite component F of the type, no embedding (using rule 3) occurs since the condition $(f, \tau_1) \preceq_F (g, \tau_2)$ does not hold. For instance, consider the following predicate signature and type definition, $p(\tau)$ and $\tau \longrightarrow F; I$. We have that $p(1) \trianglelefteq_T p(2)$ if $F = \emptyset$ and $I = \mathbb{N}$. However, $p(1) \not\trianglelefteq_T p(2)$ if $F = \{0, 1, 2\}$ and $I = \mathbb{N} \setminus \{0, 1, 2\}$.

Proposition 1. *Given a program P that is well-typed with respect to a type definition and set of signatures, there is no infinite sequence of well-typed atoms $A_1, A_2, \ldots$ in a computation for P such that for all i, j where $i < j$, $A_i \not\trianglelefteq_T A_j$.*

Proof. First note that, by the assumption that polymorphic recursion is disallowed, only a finite number of types (up to renaming of type variables) arises in a computation. The proposition follows from the fact that is a $\trianglelefteq_T$ *well quasi order (wqo)* on typed atoms over a finite set of types. A binary relation $\leq : D \times D$ is a

wqo if (i) it is reflexive and transitive, and (ii) for all infinite sequences $d_0, d_1, \ldots$ of elements of D, $\exists i < j$ such that $d_i \leq d_j$. By Theorem 4 from [11], this in turn follows if both $\preceq_F$ and $\preceq_s$ are wqos on their respective domains, which we now prove.

The proof that $\preceq_S$ is a wqo is trivial. For $\preceq_F$, it can easily be verified that the relation is reflexive and transitive. To prove the wqo property (ii) assume that there is an infinite sequence of pairs from $\Sigma_P \times \mathcal{T}$, $(f_0, \tau_0), (f_1, \tau_1), \ldots$. First assume there is only a finite number of function symbols occurring in the sequence; in this case, since there is also a finite number of types, there must exist i and j, $i < j$, such that $f_i = f_j \wedge \tau_i = \tau_j$ and hence $(f_i, \tau_i) \preceq_F (f_j, \tau_j)$. Secondly, assume that there is an infinite set of function symbols occurring in the sequence; since the number of types is finite there must exist some $j > 0$, such that f_j is in the infinite component of the type rule for τ_j, in which case $(f_i, \tau_i) \preceq_F (f_j, \tau_j)$ for all $i < j$. Hence, $\preceq_F$ is a wqo.

Proposition 1 ensures that partial evaluation using TbHEm terminates. The idea of using a typed homeomorphic embedding generalises an idea sketched in [11] to build an extended homeomorphic embedding based on a distinction between the finite number of symbols actually occurring in the program and goal (the *static* symbols), and the rest (the *dynamic* symbols). This could be reconstructed as a TbHEm using a single type rule $term \longrightarrow F; I$ where F contains cases of the form $f(term, ..., term)$ where f is a static symbol, and I contains the infinite number of cases where f is not static.The predicate signatures would allocate the type $term$ to all arguments. As discussed in Section 3.3, that approach lacks control over the different contexts in which static symbols occur in the program. Sometimes a static symbol should block embedding but other times it should not.

5 Automatic Inference of Well-Typings

In this section, we outline and experimentally evaluate an approach which, given an untyped program and a goal or set of goals, automatically infers suitable types to be used in online partial evaluation in combination with TbHEm. The approach is based on existing analysis tools for constraint logic programs.

We note first that the problem does not allow a precise, computable solution. Determining the exact set of symbols that can appear at run-time at a specific program point, and in particular determining whether the set is finite, is closely related to termination detection and is thus undecidable. However, the better the derived types are, the more aggressive partial evaluation can be without risking non-termination. If the derived types have finite components that are too small, then over-generalization is likely to result; if they are too large, then specialization might be over-aggressive, producing unnecessary versions.

A procedure for constructing a monomorphic well-typing of an arbitrary logic program was described by Bruynooghe *et al.* [5][1]. The procedure scales well

[1] Available on-line at `http://saft.ruc.dk/Tattoo/`

(roughly linear in program size) and is robust, in that every program has a well-typing, and the procedure works with partial programs (modules). We first apply this procedure to illustrate the use of well-typings in the context of our running example and, then, we perform an experimental evaluation to assess the gains that we achieve in the specialization of interpreters by using well-typings in combination with TbHEm.

5.1 Well-Typings for Working Example

In the original type inference procedure, an externally defined predicate such as `is/2` is treated as if defined by a clause `X is Y :- true` and is thus implicitly assumed not to generate any symbols not occurring elsewhere in the program. In deriving types for partial evaluation, we provide a type for such built-ins in the form of a dummy additional "fact" for `is/2`, namely `num is num :- true`. The constant `num` (assumed not to occur elsewhere in the program) will thus propagate during type inference into those types that unify with the types of the `is` predicate arguments. In the resulting inferred types, we interpret occurrences of the constant `num` as being an abbreviation for an infinite set of cases.

Example 1. A type is inferred for the interpreter sketched in Figure 1, together with the particular bytecode program of Fig. 2. Note that the program counter is sometimes computed in the interpreter using the predicate `is/2` as an offset from the current program counter value and hence its type is in principle any number. When the extra fact `num is num :- true` is added to the program, the inferred type τ_{PC} for the program counter argument `PC` is as follows.

τ_{PC} `--> -4; 0; 1; 2; 3; 4; 5; 6; 7; 8; num`

Type τ_{PC} can be naturally interpreted as consisting of a finite part (the named constants) and an infinite part (the numbers other than the named constants). In other words, the partition F of the rule is $\{-4, 0, 1, 2, \ldots, 8\}$ and $I = \texttt{num} \setminus F$. Using the rule structured in this way, TbHEm ensures that the program counter is never abstracted away during partial evaluation, so long as its value remains in the expected range (the named constants). The atom $\texttt{execute}(\texttt{st}(1, [0], [\texttt{N}, 0]), \texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(0, [], [\texttt{N}, 0]), \texttt{S}_\texttt{f})$ by using the type definition above, thus, the derivation can proceed. This avoids the need for generalizing the `PC` what would prevent us from having a quality specialization (decompilation) as explained in Sect. 2. The derivation will either eventually end or the `PC` value will be repeated due to a backwards jump in the code (loops). In this case, $\trianglelefteq_T$ will flag the relevant atom as dangerous, e.g., $\texttt{execute}(\texttt{st}(2, [], [\texttt{N}, 0]), \texttt{S}_\texttt{f}) \trianglelefteq_\texttt{T} \texttt{execute}(\texttt{st}(2, [], [\texttt{N}, 1]), \texttt{S}_\texttt{f})$, as can be seen in Fig. 3. If, however, a different value arose, perhaps due to an addressing error, the infinite part of the type rule `num` is encountered and embedding (followed by generalization of the program counter argument) would take place.

The decompiled program that we obtain using the inferred well-typings and combined with TbHEm is shown at the bottom of Fig. 3. We can observe that the decompilation is optimal in the sense that the interpretation layer has been completely removed and there is no superfluous residual code. Note that a more

sophisticated analysis could infer that τ_{PC} becomes of finite component, i.e., $I = \emptyset$ by taking $F = \{-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. This can be done by computing all combinations of bytecode indeces and offsets present in the program. In fact, $F = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ is also a correct finite component. Though this information indicates that τ_{PC} is of *finite signature* (see Section 6 below), the quality of the decompiled program does not require this extra accuracy.

5.2 Experimental Results

We have implemented the proposed TbHEm embedding relation within the partial evaluator available in CiaoPP [19] and combined it with the results obtained from the well-typing analyzer in [5]. Table 1 shows the practical benefits that we can obtain in the context of the specialization of interpreters. Each row in the table corresponds to the specialization of a bytecode interpreter w.r.t. different bytecode programs. **Counter** corresponds to the program presented in Fig. 2. We use a set of classical iterative algorithms as additional benchmarks: **Exp**, **Gcd** and **Fib** compute respectively the exponential, greatest-common-divisor and Fibonacci, and **ExpAlt** corresponds to a different implementation of the exponential. The last two benchmarks, **LinSearch** and **BinSearch**, compute respectively the classical linear and binary searches over integer arrays. Therefore, to handle them, we use an extended version of our bytecode interpreter which handles integer array manipulation. Thus, it includes a heap in the state as well as the bytecode instructions required to manipulate arrays. We have experimented as well extending the interpreter with more advanced features such as exception handling, object orientation, etc. We believe that the results obtained are generalizable to interpreters which manipulates numbers in general, and in particular to low-level language interpreters.

For each benchmark, we study the behavior of $\trianglelefteq_T$ w.r.t. $\trianglelefteq$, $\trianglelefteq_{num}$ and $\trianglelefteq^*_S$ by measuring two aspects which are crucial in the specialization of interpreters, the specialization time and the residual program size. Both aspects are directly related to the quality of the decompilation. Then, from left to right, the first two columns, **Name** and **Size**, show the name of the benchmark and the size (in KBytes) of the Prolog representation of the bytecode program. The following 9 columns show specialization times (in seconds) and residual program sizes (in KBytes) for the different strategies $\trianglelefteq$, $\trianglelefteq_{num}$, $\trianglelefteq^*_S$ and $\trianglelefteq_T$. We write "-" when the specialization does not terminate. Note that, in the group of columns corresponding to $\trianglelefteq_T$, we have an additional column T_{wt} which shows the time taken by the well-typing analysis which should be added to the specialization time in order to obtain a proper evaluation of $\trianglelefteq_T$. It should be noted also that the usage of $\trianglelefteq^*_S$ would require a preprocessing time currently not being taken into account which should be no more than the times in T_{wt}. Since we do not have an implementation of $\trianglelefteq^*_S$ the results obtained for it have been obtained using the TbHEm writing by hand the corresponding types. Finally, the last two columns show the gains (in terms of time and size) of the embedding relation $\trianglelefteq_T$ w.r.t. $\trianglelefteq_{num}$ (in column $\mathbf{T/S}_{(\trianglelefteq_{num})}$) and $\trianglelefteq^*_S$ (in column $\mathbf{T/S}_{(\trianglelefteq^*_S)}$). The gain is computed as *Old-Cost*/*New-Cost*. As we can observe in the table, $\trianglelefteq_T$

Table 1. Measuring the effects of $\trianglelefteq_T$ with the bytecode interpreter

Benchmark		$\trianglelefteq$		$\trianglelefteq_{num}$		$\trianglelefteq^*_S$		$\trianglelefteq_T$			Gains	
Name	Size	Tm	Size	Tm	Size	Tm	Size	T_{wt}	Tm	Size	$T/S_{(\trianglelefteq_{num})}$	$T/S_{(\trianglelefteq^*_S)}$
Counter	0.27	-	-	0.12	1.79	0.60	1.26	0.03	0.09	0.28	1.4/6.3	6.7/4.4
Exp	0.39	0.14	0.50	0.24	5.51	0.14	0.50	0.03	0.14	0.50	1.7/11.0	1.0/1.0
Gcd	0.35	0.13	0.38	0.23	4.80	0.14	0.38	0.03	0.11	0.29	2.2/16.3	1.4/1.3
ExpAlt	0.44	-	-	0.26	6.13	3.75	4.50	0.03	0.13	0.34	2.0/17.8	29.0/13.1
Fib	0.52	-	-	0.49	10.72	0.99	1.41	0.03	0.15	0.51	3.2/21.2	6.6/2.8
LinSearch	0.70	-	-	0.54	13.69	3.99	9.04	0.04	0.25	1.70	2.1/8.1	15.7/5.3
BinSearch	2.00	3.14	9.26	5.05	112.50	3.20	9.26	0.04	1.59	5.51	3.2/20.4	2.0/1.7

guarantees termination and behaves significantly better than $\trianglelefteq_{num}$ and $\trianglelefteq^*_S$ both in time and size. Furthermore, $\trianglelefteq_T$ behaves as well as $\trianglelefteq$ in the examples in which $\trianglelefteq$ terminates, even after adding the additional cost taken by the well-typing analysis. An important observation as regards the gains w.r.t. $\trianglelefteq^*_S$ is that for some benchmarks such gains are large while for others they are almost insignificant. The reason for this lack of improvement is that in the corresponding atoms, the local variables within the state are not instantiated to concrete values almost from the beginning. Therefore, the over-specialization problem of $\trianglelefteq^*_S$ pointed in Sect. 3.3 is not exposed. In fact, note that these cases correspond precisely to the cases where $\trianglelefteq$ terminates (due to the same reason).

6 Type-Based Homeomorphic Embedding in Practice

An important observation is that, in order to take full advantage of TbHEm in practice, it is not always necessary to know the actual type definitions, but only sufficient information for the relations $\preceq_F$ and $\preceq_S$ proposed in Sect. 4.2 to be well defined. In particular it suffices to know whether the infinite component of type rules is (transitively) empty or not. Moreover, it would be desirable to define a condition on types specifying that a type and all the types on which it depends are defined over a finite signature. In this case, we can safely revert to the simpler HEm applied directly to terms of such types. In the following we define such a condition.

Definition 5 (finite signature). *Given a type τ defined by a type rule $\tau \longrightarrow F;\emptyset$ we say that τ is of* finite signature*, denoted* $\mathsf{f\_sig}(\tau)$*, iff $F = \{f_1(\tau_{11}, \ldots, \tau_{1k_1}), \ldots, f_n(\tau_{n1}, \ldots, \tau_{nk_n})\}$ and all types $\tau_{11}, \ldots, \tau_{nk_n}$ are of finite signature.*

Hence, if a type τ is of *finite signature* the (possibly infinite) set of terms of type τ contains only a finite set of functors. As the following Proposition implies, we can then use $\trianglelefteq$ instead of $\trianglelefteq_T$ when comparing terms in the context of finite signatures.

Proposition 2. *Given two typed terms $t_1 : \tau_1$ and $t_2 : \tau_2$, if* $\mathsf{f\_sig}(\tau_2)$ *holds then $t_1 : \tau_1 \trianglelefteq_T t_2 : \tau_2 \Leftrightarrow t_1 \trianglelefteq t_2$.*

In the following, for every type τ for which $\mathsf{f\_sig}(\tau)$ holds, we simply write f_sig instead of the particular type. We now propose an extension to the definition of $\trianglelefteq_T$ to consider f_sig types. This is done simply by adding the following rule to Def. 4: *4.* $s{:}\tau_1 \trianglelefteq_T t{:}\mathsf{f\_sig}$ if $s \trianglelefteq t$.

In order to put these ideas into practice it is convenient to also have the type i_sig which is assigned to an argument when we cannot guarantee it is of finite signature and we do not have further information available about its type. Note that we are assuming a scenario where infinite signatures can include functors as well as numbers.

Definition 6 (i_sig). *The type* i_sig *is defined by the following type rule:* $\mathsf{i\_sig} \longrightarrow \emptyset; I$ *where* $I = \{f_1(\tau_{11}, \ldots, \tau_{1k_1}), \ldots, f_n(\tau_{n1}, \ldots, \tau_{nk_n}), \ldots\}$ *and* f_i *are all possible functors and all types* $\tau_{11}, \ldots, \tau_{nk_n}$ *are* i_sig.

Note that since every case of the type rule belongs to the infinite component then $s{:}\tau \trianglelefteq_T t{:}\mathsf{i\_sig}$ will always hold (as $\preceq_F$ holds for every s, τ and t). Hence, termination is trivially guaranteed for terms of type i_sig. In practice, in programs with infinite signatures, unless the user (or an automatic analysis) explicitly writes more concrete type declarations, a default *typing* will be assumed such that all predicates p/n of a program have the *predicate signature* $p(\tau_1, \ldots, \tau_n)$ with $\tau_i = \mathsf{i\_sig}$, $(0 \leq i \leq n)$. Then, more concrete declarations are allowed both by declaring particular types and signatures (always preserving the well-typing assumption, see Sect. 4) or by using the special type f_sig.

Example 2. Consider again the interpreter in our motivating example. Though it is natural to use integer numbers to represent program counters, the set of instructions is finite in any bytecode program. Therefore the PC can be safely declared as f_sig. Thus we may write the following predicate signature and type definition:

$$\mathtt{execute}(\tau_{st}, \tau_{st}).$$
$$\tau_{st} \longrightarrow \{\mathtt{st}(\mathsf{f\_sig}, \mathsf{i\_sig}, \mathsf{i\_sig})\}; \emptyset.$$

With this type declaration we are able to obtain the same results as in Sect 5.1 in a more efficient way, as we can get rid of the overhead produced by the comparisons checking that the current PC belongs to the finite part of the corresponding type. In addition, the type declaration holds for all input programs, whereas before a separate type inference was needed for each input object program.

Another interesting observation is that the relation $\trianglelefteq_S^*$ may be defined as a particular case of TbHEm by simply declaring the following particular type and assuming that every argument of every predicate is of this type: $\mathsf{s\_symb} \longrightarrow F; I$ where $F = \{f_1(\tau_{11}, \ldots, \tau_{1k_1}), \ldots, f_n(\tau_{n1}, \ldots, \tau_{nk_n})\}$ with $f_1, \ldots, f_k$ being all the functor symbols which explicitly occur in the program text plus initial goal(s) and the types $\tau_{11}, \ldots, \tau_{nk_n}, \ldots$ are s_symb. I contains the infinite set of all other possible functors, with auxiliary types i_sig in all cases.

6.1 Automatic Inference of Finite Signature

If, in a program with builtins, we can use some static analysis which allows us to determine that the type of an argument has a finite signature, we can provide

this information to the partial evaluator as an f_sig declaration, without having to specify the exact type. E.g., given a logic program processing numeric values, analyses exist that make over-approximations of the set of values that the program arguments can have. Polyhedral analyses are perhaps the most widely known of these and they have successfully been applied to constraint logic programs [4]. Let us assume for the sake of this discussion that a polyhedral analysis can return, for a given program and goal, an approximation to the set of calls to each n-ary predicate p, in the form: $p(X_1, \ldots, X_n) \leftarrow c(X_1, \ldots, X_n)$, where the expression $c(X_1, \ldots, X_n)$ is a set of linear constraints (describing a possibly not closed polyhedron). From this information it can be determined whether each argument X_i is bounded or not by projecting $c(X_1, \ldots, X_n)$ onto X_i. If it is bounded (from above and below), and it is known that the ith argument takes on integral values, then it can take only a finite set of values and thus can be declared as f_sig.

Example 3. Consider the following clauses defining a procedure for computing an exponential.

```
exp(Base,Exp,Res)      :- exp_(Base,Exp,1,Res).
exp_(_,0,Ac,Ac).
exp_(Base,Exp,Ac,Res) :- Exp > 0, Exp' is Exp-1, Ac' is Ac*Base,
                         exp_(Base,Exp',Ac',Res)
```

Type inference yields the following signature for the predicate `exp_/4`: `exp_(t24,t24, t24,t24)` with the type `t24 --> 0; 1; num`. A polyhedral analysis of the same program with respect to the goal `exp(Base,10,Res)` yields the following approximation to the queries to `exp_/4`: `exp_(Base,Exp,Ac,Res) :- Exp > -1, Exp =< 10`. Combining this with the inferred type, and assuming that the second argument can take only integer values. the second argument (`Exp`) can be declared as f_sig, and hence we can revert to HEm and do not abstract away the value of the second argument of `exp_/4`. This allows maximum specialization to be achieved.

6.2 Experimental Results

We have incorporated the proposed predefined types f_sig and i_sig within our partial evaluator and instrumented TbHEm to properly handle them as proposed above. Table 2 shows the practical benefits that we obtain on a set of numeric programs which we make extensive use of the arithmetic builtin `is/2`. **exp** and **fib** correspond to the iterative implementations (using accumulators) of the exponential and Fibonacci functions respectively. **vnr** computes a combinatorial function, in this case without accumulators. **list_exp** takes a list of numbers and an exponent and computes a list in which every element is powered to the corresponding exponent (using the predicate `exp/3` defined in **exp**) and also computes the length of the list by using an accumulator. Finally, **dfs** performs a depth-first search avoiding state repetitions in a two dimensional space. Predicate `path/4` computes the path and its cost (using an accumulator) given the initial and final states.

Table 2. Measuring the effects of $\trianglelefteq_T$ with numeric programs

Bench	Entry	$\mathbf{T}_{orig}$	$\mathbf{T}_{res\trianglelefteq}$	$\mathbf{T}_{res\trianglelefteq_{num}}$	PE-type	$\mathbf{T}_{res\trianglelefteq_T}$
exp	exp(11,1000,_)	19.60	14.60	19.20	exp_(i_sig,f_sig,i_sig,i_sig)	14.20
	exp(11,_,_)	19.20	-	19.20		19.00
fib	fib(1000,_)	17.20	14.20	16.00	fib_(f_sig,i_sig,i_sig,i_sig)	14.00
	fib(_,_)	16.80	-	16.00		15.60
vnr	vnr(10000,1000,_)	31.80	14.20	32.40	vnr(i_sig,f_sig,i_sig)	14.00
	vnr(10000,_,_)	30.00	-	30.00		32.20
dfs	path((1,1),(4,4),_,_)	49.79	15.60	43.39	path_(f_sig,f_sig,i_sig,i_sig,...)	15.80
	path(_,_,_,_)	43.39	-	39.79		42.19
list_exp	lel([1,...,40|_],200,_,_)	32.40	-	32.40	lel_(i_sig,i_sig,i_sig,i_sig)	14.40
	lel(_,200,_,_)	31.80	-	31.60		26.80

In this case, in order to measure the quality of the specialization we compare the execution times of the specialized programs ($\mathbf{T}_{res}$) with the execution times of the original programs ($\mathbf{T}_{orig}$) for sufficiently large inputs. From left to right, the first two columns, **Bench** and **Entry**, show respectively the name of the benchmark and the entry for which the program will be specialized. Then, for each pair benchmark-entry, we show the execution times (in seconds) of the original programs in $\mathbf{T}_{orig}$ and of the corresponding residual programs, by using the three relations $\mathbf{T}_{res\trianglelefteq}$, $\mathbf{T}_{res\trianglelefteq_{num}}$ and $\mathbf{T}_{res\trianglelefteq_T}$. We also show the particular type definition which has been used to guide $\trianglelefteq_T$. Note that in this case we do not consider $\trianglelefteq_S^*$ since it does not produce any significant improvement w.r.t. $\trianglelefteq_{num}$ (constants do not play any role in the involved terms). All times have been computed as the arithmetic means of five runs. For each run, in order to accurately compare the involved programs we run five consecutive times the call `findall(_,Goal,_)`. The particular goals used for measuring the execution times have been chosen to match the entries proposed for each benchmark. As it can be seen, $\trianglelefteq_T$ guarantees termination and outperforms significantly $\trianglelefteq_{num}$. As expected, $\trianglelefteq$ exposes termination problems for some entries as showed in column $\mathbf{T}_{res\trianglelefteq}$. In the examples in which $\trianglelefteq$ terminates, $\trianglelefteq_T$ behaves as well as $\trianglelefteq$. In some examples, no improvements are obtained in the residual programs. This is explained by the fact that the corresponding entries do not provide static information to be used in the specialization. In these examples, it is usual to observe the (unnecessary) over-aggressive nature of $\trianglelefteq$ (even endangering termination in presence of infinite signatures) while, we can see, that the particular type declarations can prevent such undesired behavior in $\trianglelefteq_T$. An interesting observation is that, although many of the examples in this table may be handled in offline PE (by providing the corresponding annotations), there are cases, as **dfs**, where it is not possible to obtain a ranking function for the key arguments. Luckily, we may infer boundedness which is a sufficient condition to effectively use our TbHEm.

7 Discussion and Related Work

Guaranteeing termination is essential in a number of tasks which have to deal with possibly infinite computations. These tasks include PE, abstract model checking, rewriting, etc. Broadly speaking, guaranteeing termination can be tackled in an *offline* or an *online* fashion. The main difference between these two perspectives is that in offline termination we aim at statically determining termination. This means that we do not have the concrete values of arguments at each point of the computation but rather just *abstractions* of them. Traditionally, these abstractions refer to the *size* of values under some measure such as list length, term size, numeric value for natural numbers, etc. In contrast, in online termination, we aim at dynamically guaranteeing termination by supervising the computation in such a way that it is not allowed to proceed as soon as we can no longer guarantee termination. The main advantage of the offline approach is that if we can prove termination statically, there is no longer any need to supervise the computation for termination, which results in important performance gains. However, the online approach is potentially more precise, since we have the concrete values at hand. In offline PE, the problem of termination of local unfolding has been tackled by annotating arguments as "bounded static". The work of Glenstrup and Jones [7] is the main reference, though the idea of bounded static variation goes back a long way. To detect bounded static arguments it is necessary to prove some decrease in well-founded ordering (e.g. using size-change techniques). Quasi-termination is weaker than standard termination but still quite hard to prove. Recent work on this has been done by Vidal [21] and by Glenstrup and Jones [7]. On the other hand, ensuring termination in online PE is easier because we can use "dynamic" termination detection based on supervisors of the computations such as for example embeddings. This means that we do not need any well-founded orderings but only well-quasi-orderings. In effect, in our technique it is only necessary to show boundedness of an argument's values instead of decrease.

In the context of online PE, we have compared TbHEm with the extension of the embedding relation to deal with infinite signatures explained in [11], known as *extended embedding* with static symbols in Sect. 3.3, which is based on a distinction between the different static symbols which occur in the program. As we have shown in the paper, the main advantage of TbHEm is that it achieves a more refined treatment, as it allows treating different arguments in a different way depending on their particular types, which can be automatically inferred by semantic-based analysis, while previous proposals are purely syntactic. Additionally, we have shown that TbHEm can be applied to the specialization of numeric programs, by means of finite signature annotations, in which static constants do not play any role.

Acknowledgments. The authors would like to thank the anonymous referees for their useful comments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Danish Natural

Science Research Council under the FNU-272-06-0574 *SAFT* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

1. Albert, E., Gómez-Zamalloa, M., Hubert, L., Puebla, G.: Verification of Java Bytecode Using Analysis and Transformation of Logic Programs. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 124–139. Springer, Heidelberg (2007)
2. Albert, E., Hanus, M., Vidal, G.: A practical partial evaluation scheme for multi-paradigm declarative languages. Journal of Functional and Logic Programming 2002(1) (2002)
3. Albert, E., Puebla, G., Gallagher, J.: Non-leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In: Hill, P.M. (ed.) LOPSTR 2005. LNCS, vol. 3901, pp. 115–132. Springer, Heidelberg (2006)
4. Benoy, F., King, A.: Inferring argument size relationships with CLP(R). In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1996)
5. Bruynooghe, M., Gallagher, J.P., Van Humbeeck, W.: Inference of Well-Typings for Logic Programs with Application to Termination Analysis. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 35–51. Springer, Heidelberg (2005)
6. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Vol. B, pp. 243–320. Elsevier, Amsterdam (1990)
7. Glenstrup, A.J., Jones, N.D.: Termination analysis and specialization-point insertion in offline partial evaluation. ACM Trans. Program. Lang. Syst. 27(6), 1147–1215 (2005)
8. Hill, P.M., Topor, R.W.: A semantics for typed logic programs. In: Pfenning, F. (ed.) Types in Logic Programming, pp. 1–62. MIT Press, Cambridge (1992)
9. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)
10. Kruskal, J.B.: Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. Transactions of the American Mathematical Society 95, 210–225 (1960)
11. Leuschel, M.A.: Homeomorphic Embedding for Online Termination of Symbolic Methods. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 379–403. Springer, Heidelberg (2002)
12. Leuschel, M., Bruynooghe, M.: Logic program specialisation through partial deduction: Control issues. Theory and Practice of Logic Programming 2(4&5), 461–515 (2002)
13. Leuschel, M., Martens, B., De Schreye, D.: Controlling Generalisation and Polyvariance in Partial Deduction of Normal Logic Programs. ACM Transactions on Programming Languages and Systems 20(1), 208–258 (1998)
14. Leuschel, M.: On the power of homeomorphic embedding for online termination. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 230–245. Springer, Heidelberg (1998)
15. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. The Journal of Logic Programming 11, 217–242 (1991)

16. Lloyd, J.W.: Foundations of Logic Programming. Springer, Heidelberg (1987) (second, extended edition)
17. Mycroft, A., O'Keefe, R.A.: A polymorphic type system for Prolog. Artif. Intell. 23(3), 295–307 (1984)
18. Puebla, G., Albert, E., Hermenegildo, M.: Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 149–165. Springer, Heidelberg (2005)
19. Puebla, G., Albert, E., Hermenegildo, M.: Abstract Interpretation with Specialized Definitions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 107–126. Springer, Heidelberg (2006)
20. Somogyi, Z., Henderson, F., Conway, T.: The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. JLP 3 (October 1996)
21. Vidal, G.: Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In: Proc. of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM 2007), pp. 51–60. ACM Press, New York (2007)

Information Processing Letters 109 (2009) 879–886

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Type-based homeomorphic embedding for online termination

Elvira Albert [a], John Gallagher [b,d], Miguel Gómez-Zamalloa [a,*], Germán Puebla [c]

[a] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[b] CBIT, Roskilde University, DK-4000 Roskilde, Denmark
[c] CLIP, DLSIIS, Technical University of Madrid, E-28660 Boadilla del Monte, Spain
[d] IMDEA Software, E-28660 Boadilla del Monte, Spain

ARTICLE INFO

Article history:
Received 7 August 2008
Available online 24 April 2009
Communicated by D. Basin

Keywords:
Analysis of algorithms
Formal methods
Termination
Well-quasi orders
Homeomorphic embedding
Program transformation
Partial evaluation

ABSTRACT

Online termination techniques dynamically guarantee termination of computations by *supervising* them in such a way that computations whose termination can no longer be guaranteed are stopped. *Homeomorphic Embedding* (HEm) has proven to be very useful for online termination provided that the computations supervised are performed over a *finite signature*, i.e., the number of constants and function symbols involved is finite. However, there are many situations, for example numeric computations, which involve an infinite signature and thus HEm does not guarantee termination. Some extensions to HEm for the case of infinite signatures have been proposed which guarantee termination. However, the existing techniques either do not provide systematic means for generating such extensions or the extensions are too simplistic and do not produce the expected results in practice. We propose *Type-based Homeomorphic Embedding* (TbHEm) as an extension of the standard, untyped, HEm. By taking static information about the behavior of the computation into account, expressed as types, TbHEm allows obtaining more precise results than those of the previous extensions to HEm for the case of infinite signatures. We show that the existing extensions to HEm which are currently used in state-of-the-art specialization tools can be reconstructed as instances of TbHEm. We illustrate the applicability of our proposal in a realistic case study: partial evaluation of an interpreter. We argue that the results obtained provide empirical evidence of the interest of our proposal.

1. Introduction

Guaranteeing termination is a key aspect of areas of computer science which have to deal with possibly infinite computations, namely in all areas of automatic program analysis, synthesis, verification, specialization and transformation. Broadly speaking, guaranteeing termination can be tackled in an *offline* or an *online* fashion. The main difference between these is that in offline termination we aim at statically determining termination. This means that we do not have the concrete values of arguments at each point of the computation but rather just *abstractions* of them. Usually these abstractions refer to the *size* of values under some measure, such as list length, term size, numeric value for natural numbers, etc. In contrast, in online termination, we guarantee termination by *supervising* the computation and stopping it as soon as we can no longer guarantee termination.

The main advantage of the offline approach is that if we can prove termination statically, there is no longer any need to supervise the computation for termination, which results in performance gains. In the offline setting, powerful semi-automated termination proof techniques have been developed in the context of term rewrite systems (TRS), the most popular one being the *recursive path ordering* [6].

On the other hand, the online approach is more precise, since we have the concrete values at hand and thus we can compare actual values instead of abstractions of values. Thus, the online approach is of interest in applica-

* Corresponding author.
E-mail address: mzamalloa@clip.dia.fi.upm.es (M. Gómez-Zamalloa).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2009.04.016

tions where precision is of great importance and where the offline approach tends to behave too conservatively in order to guarantee termination. Another advantage of online techniques is that they are usually simpler to implement than offline techniques, which are based on sophisticated static analyses. As a result, which of the two approaches to take greatly depends on the application area. For example, in the context of online supervision of symbolic computations, *well-founded orders* (wfos) [18] and especially *well-quasi orders* (wqos) [4,21] have become widely used.

In order theory, a wqo is a quasi-order with an additional restriction on sequences that ensures that for any infinite sequence $x_1, x_2, \ldots$, there exists $i < j$ with $x_i \leqslant x_j$. In this article, we focus on the *homeomorphic embedding* (HEm) relation [11,13,14], a wqo used in state-of-the-art online specialization tools. Intuitively, HEm is a structural ordering under which an expression e_2 is greater than or equal to another expression e_1, written as $e_1 \trianglelefteq e_2$, if e_1 can be obtained from e_2 by deleting some parts of e_2. Under these circumstances we say that e_2 *embeds* e_1. E.g., $\underline{\mathtt{s}}(\mathtt{s}(\underline{\mathtt{U}} + \mathtt{W}) \underline{\times} (\underline{\mathtt{U}} \underline{+} \mathtt{s}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U} \times (\mathtt{U} + \mathtt{V}))$.

The HEm relation was first defined over strings by Higman [9] and later extended by Kruskal [11] to ordered trees (and thus symbolic expressions). Since then, HEm has been used for many applications. Arguably, the heaviest use of HEm within computer science was made in the context of TRS [7], to automatically derive well-founded orders for static termination analysis. The usefulness of HEm in the context of online partial evaluation was first discovered and advocated by Marlet [17]. It was later, independently, rediscovered and adapted for supercompilation by Sørensen and Glück [23]. Later on, Leuschel and Martens [15,16] demonstrated that HEm provides a mathematically simpler and still more powerful way of ensuring termination of partial deduction than existing wfos and wqos. The latter was then witnessed by Leuschel [12]. A survey on the theory and practice of HEm can be found in [13].

The HEm relation can be used to guarantee termination when computing a sequence $e_1, e_2, \ldots$, by using HEm as a *whistle*. Whenever a new expression e_{n+1} is to be added to a finite sequence $e_1, \ldots, e_n$, we first check whether e_{n+1} embeds any of the expressions already in the sequence. If that is the case, we say that HEm whistles, i.e., it has detected (potential) non-termination and the computation has to be stopped. If HEm does not whistle e_{n+1} can be safely added to the sequence and the computation can proceed without endangering termination.

The reason for the success of HEm as an approach for guaranteeing online termination is twofold. (i) It often allows sequences to grow quite large before the whistle blows, to the point that in a good number of finite sequences the full sequence can be computed without the whistle blowing at all. This is essential for instance, in program specialization, as allowing a larger sequence implies further propagation of information and hence, as we will see in the paper, a better specialization can often be obtained. (ii) It often identifies redundant computations quickly, and the whistle blows without unnecessarily further expanding the sequence, thus avoiding irrelevant computations. This is also essential in program specialization both for efficiency of the specialization process and for quality of the resulting program.

While HEm has proven to be very useful for symbolic computations (as required by program specialization and analysis techniques, see [12]), some difficulties remain in the presence of infinite signatures, such as the numbers. For instance, the `is/2` Prolog built-in is used to evaluate arithmetic expressions; given two numbers, it can produce as output a number which does not appear in the program text. If this can be infinitely repeated, we need to handle an infinite signature. As further examples, in the case of logic programs, infinite signatures appear as soon as certain built-ins such as `functor/3`, `name/2`, `=../2`, `atom_codes/2`, etc. are used, since they allow creating fresh constants and function symbols. Some extensions to HEm over infinite signatures have been defined and used in practice (e.g. [2,13]), but they are often too *ad hoc*; for instance, they only handle constants which appear explicitly in the program, regardless of which part of the program (function, argument position) they appear. As such approaches are purely *syntactic*, in practice they sometimes turn out to be too conservative ("whistling" too early) or else too aggressive, and thus do not have either of the features (i) or (ii) above.

In essence, while other works [2,13] take a simple *syntactic* approach to extending the HEm relation, we propose a *semantic* approach for such extension. In particular, we introduce the *type-based homeomorphic embedding* (TbHEm) relation which, by taking information about the behavior of the computation into account, provides more precise results in the presence of infinite signatures. For this, our typed relation is defined on types structured into a (possibly empty) *finite part* and a (possibly empty) *infinite part*. TbHEm allows expanding sequences as long as the concrete values which appear in the expression remain within the finite part of the type. Note that in computations with a finite signature it is always possible to obtain types whose infinite part is empty, which produces the same effect as the traditional HEm. Intuitively, this allows us to achieve (i) and (ii) simultaneously as: (i) finite sequences are expected to have an empty infinite partition, or non-empty but with a large finite part, and hence they can grow considerably before the whistle blows; (ii) infinite sequences will have a non-empty partition which forces the whistle to blow.

HEm has been extensively used for supervising partial evaluation of logic programs. However, it is important to stress that both the HEm and TbHEm relations are of interest for supervising any computation which manipulates acyclic data structures, such as lists (or equivalently, acyclic linked lists), trees, etc. which can grow indefinitely large. This is so regardless of the programming language in which such computation is implemented.

The rest of the article is organized as follows. Section 2 recalls some basic notions and introduces notation. In Section 3, we introduce TbHEm, as a novel extension to untyped HEm, and prove its soundness. Section 5 shows how TbHEm generalizes existing relations used in current systems. In Section 6 we present some experimental results. Finally, Section 7 discusses the practicality of TbHEm in the context of online termination approaches and concludes.

2. Preliminaries and notation

We recall some preliminary concepts, in particular on the HEm relation, and introduce some notation.

2.1. Symbolic expressions

For the sake of generality we consider the language of *symbolic expressions* (first-order terms). Its *alphabet* consists of the following classes of symbols: (1) *variables* ($\mathcal{V}$) and (2) *function symbols* (Σ). Function symbols have an associated *arity*. *Constants* are function symbols with arity 0. We refer to the set of functions in an alphabet as its *signature*.

Definition 1 *(Symbolic expressions).* The set of symbolic expressions $\mathcal{E}$ over some given alphabet, $\Sigma \cup \mathcal{V}$, is inductively defined as follows:

(i) a variable $v \in \mathcal{V}$ is an expression,
(ii) a function symbol $f \in \Sigma$ of arity $n \geqslant 0$ applied to a sequence $e_1, \ldots, e_n$ of expressions, denoted $f(e_1, \ldots, e_n)$, is also an expression.

We will adhere to the following syntactical conventions: Variables are denoted by upper-case letters like $X, Y, Z, \ldots$, constants by lower-case letters like $a, b, c, \ldots$, and non-constant function symbols by lower-case letters like $f, g, h, \ldots$.

2.2. Homeomorphic embedding

We now introduce some auxiliary definitions on orders which are required to define the HEm relation.

Definition 2 *(Quasi-order).* A *quasi-order* is a reflexive and transitive binary relation on $\mathcal{E}$.

A *well-quasi order* is a *well-binary relation* which is also a quasi-order, as stated below.

Definition 3 *(*wbr, wqo*).* Let $\leqslant$ be a binary relation on $\mathcal{E}$. We say that $\leqslant$ is a *well-binary* relation (wbr) iff for any infinite sequence $e_1, e_2, \ldots$ of expressions, $\exists i, j$: $i < j \wedge e_i \leqslant e_j$. If $\leqslant$ is also a quasi-order then $\leqslant$ is called a *well-quasi order* (wqo).

The next definition recalls the HEm relation on expressions, as presented by Leuschel [12].

Definition 4 *(*HEm, $\trianglelefteq$*).* The homeomorphic embedding relation over expressions, written $\trianglelefteq$, is defined by the following rules:

(i) $Y \trianglelefteq X$ for all variables X, Y.
(ii) $s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i.
(iii) $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $s_i \trianglelefteq t_i$ for all i, $1 \leqslant i \leqslant n$.

As already discussed, $e_1 \trianglelefteq e_2$ iff e_1 can be obtained from e_2 by removing some symbols. Hence, the structure of e_1, split in parts, reappears within e_2. For finite signatures, HEm is a wqo (see, e.g., [12]).

2.3. Types

We adopt the syntax of Mercury [22] for type definitions. The set of *type expressions* (*types*), denoted $\mathcal{T}$, is constructed from an infinite set $\mathcal{V}_\mathcal{T}$ of type variables (parameters) and a set $\Sigma_\mathcal{T}$ of type symbols with their associated arities; these are disjoint from the set of variables $\mathcal{V}$ and the signature Σ. Types and symbolic expressions are related by means of *type definitions*.

Definition 5 *(Type definition).* A *type rule* for a type symbol h with arity n in $\Sigma_\mathcal{T}$ is of one of these two forms:

$$h(\bar{T}) \to f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k) \quad (k \geqslant 1), \quad \text{or}$$

$$h(\bar{T}) \to f_1(\bar{\tau}_1); \ldots \quad \text{(infinite sequence)},$$

where the following conditions hold:

(i) $\bar{T}$ is an n-tuple of distinct type variables,
(ii) $f_1, \ldots, f_k, \ldots$ are distinct function symbols from Σ,
(iii) each $\bar{\tau}_i$ ($i \geqslant 1$) is an m-tuple from $\mathcal{T}$, where m is the arity of the corresponding f_i,
(iv) type variables in the right-hand side, if any, are from $\bar{T}$.

We say that $f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k)$ or $f_1(\bar{\tau}_1); \ldots$ are the, possibly infinite, set of cases of the type.

A *type definition* is a finite set of type rules where no two rules contain the same type symbol on the left-hand side, and there is a rule for each type symbol occurring in the type rules.

Example 6. The following type *natlist* characterizes lists of natural numbers:

$$natlist \to nil; cons(nat, natlist)$$

$$nat \to 0; 1; 2; \ldots$$

A *variable typing* is a mapping $\sigma : \mathcal{V} \to \mathcal{T}$. An expression $t \in \mathcal{E}$ is of type $\tau \in \mathcal{T}$ with respect to a given type definition and a variable typing, written $t : \tau$, if (i) $t \in \mathcal{V}$ and $\sigma(t) = \tau$, or (ii) $t = f(t_1, \ldots, t_n)$, and there is an instance of a type rule, $\tau \to \ldots; f(\tau_1, \ldots, \tau_n); \ldots$, and $t_i : \tau_i$, $1 \leqslant i \leqslant n$.

Definition 5 permits *overloading* – a function symbol can occur in several type rules. Thus, a given expression may be of more than one type. We also allow type rules containing an infinite number of distinct function symbols on the right-hand side. Thus, standard infinite types such as *integer* are permitted, defined by a rule with an infinite number of cases containing the numeric constants.

In order to define TbHEm we need to handle types with an infinite number of cases. This can be done simply by using some sort of intensional notation for them (e.g., $\mathbb{N}$ for natural numbers). However, at the same time we need to distinguish some finite number of elements of the type. Hence, we introduce the following extra annotation into

type rules. The right-hand side of each type rule consists of two disjoint components, each possibly empty. More precisely, type rules are of the form $h(\bar{T}) \to F; I$, where the union $F \cup I$ are the cases in the type rule, $F \cup I$ is non-empty, F is either empty or finite and I is either empty or infinite. We say that a type $\tau \in \mathcal{T}$ is of *infinite component* if I is non-empty in the rule defining τ. Otherwise it is said to be of *finite component*. Thus, for types of infinite component there are infinitely many ways of splitting them into type rules; for example $nat \to F; I$ where $F = \emptyset$ and $I = \mathbb{N}$, or $F = \{0, 1, 2\}$ and $I = \mathbb{N} \setminus \{0, 1, 2\}$, etc. The way in which infinite components are split affects the behavior of TbHEm.

3. Type-based homeomorphic embedding

HEm turns out to be unsatisfactory, due to the restriction to finite signatures. Most real-life programs involve infinite signatures. These, for example, appear quite easily if the program performs arithmetic operations. Indeed, the fully general definition of HEm which dates back to the 1960s [11,7], referred to as the *extended homeomorphic embedding* ($\trianglelefteq^*$), allows infinite signatures. It is based on two generic relations, $\preceq_\Sigma$ and $\preceq_S$ on function symbols and sequences of expressions respectively. It can be shown that if these relations are wbrs (resp. wqos) then $\trianglelefteq^*$ is a wbr (resp. wqo). The next definition is adapted from Leuschel [13], but we use the symbol $\preceq_\Sigma$ instead of $\preceq_F$.

Definition 7 *(Extended* HEm, $\trianglelefteq^*$*).* Given a wqo $\preceq_\Sigma$ on the function symbols and a wqo $\preceq_S$ on sequences of expressions, the *extended homeomorphic embedding* on expressions is defined by the following rules:

(i) $X \trianglelefteq^* Y$ if X and Y are variables,
(ii) $s \trianglelefteq^* f(t_1, \ldots, t_n)$ if $s \trianglelefteq^* t_i$ for some i,
(iii) $f(s_1, \ldots, s_n) \trianglelefteq^* g(t_1, \ldots, t_m)$ if
  (a) $f \preceq_\Sigma g$,
  (b) $\langle s_1, \ldots, s_n \rangle \preceq_S \langle t_1, \ldots, t_m \rangle$, and
  (c) $\exists i_1, \ldots, i_n$ such that $1 \leqslant i_1 < \cdots < i_n \leqslant m$ and $\forall j \in \{1, \ldots, n\}$: $s_j \trianglelefteq^* t_{i_j}$.

The most important point in the above definition is that, in contrast to Definition 4, the left- and right-hand expressions in rule (iii) do not have to have the same function symbol. The two function symbols are instead compared by using the relation $\preceq_\Sigma$. Furthermore, the expressions do not have to be of the same arity; the left-hand side expression can have fewer arguments than the right-hand expression. In this case, $m - n$ arguments from the right-hand expression are ignored.

We do not use the full generality of this definition here; in particular the relation $\preceq_S$ will be taken as the relation that is always true, in which case condition (iii.b) is trivially satisfied. As noted by Leuschel [13], the relation $\preceq_S$ could be used to give a more refined treatment of variables as well as a more refined treatment of associative operators. For example, it would be natural to have $\wedge(a, b, c)$ embedded in $\wedge(a, b, c, d)$, but if $\wedge$ is taken as a binary functor then the embedding relation depends on the nested structure of the expressions.

The above definition establishes a family of embedding relations but leaves open the practical problem of finding an effective embedding relation, since there is no automated mechanism for finding a "good" ordering relation $\preceq_\Sigma$ on the function symbols in the signature to ensure effective termination control. This is the problem we address in the next section, where we propose using types in order to define the $\preceq_\Sigma$ relation.

3.1. The type-based relation

We now introduce *type-based homeomorphic embedding* (TbHEm). We outline how TbHEm can provide a way of generating instances of extended HEm based on given type definitions. Note that this allows taking into account program-specific factors by using the type definition associated to a given program. The types required for guiding TbHEm can be automatically inferred by program analysis, as discussed in Section 7, or be provided manually.

Intuitively, termination control using TbHEm is based on the following idea: embedding occurs between typed expressions with different function symbols, if the function symbol of the "larger" expression is from the infinite component of its type. However, as long as we compare distinct expressions from an infinite type whose function symbols are from the finite component of the type, we can safely use essentially the standard embedding relation. This motivates the definition of the relation $\preceq_{\Sigma,D}$, which plays the role of $\preceq_\Sigma$ in Definition 7.

Definition 8 *($\preceq_{\Sigma,D}$).* Given a type definition D, let $\preceq_{\Sigma,D}$ be the following relation on the set of pairs $\Sigma \times \mathcal{T}$. $(f_1, \tau_1) \preceq_{\Sigma,D} (f_2, \tau_2)$ iff

- $f_1 = f_2 \wedge \tau_1 \trianglelefteq \tau_2$, or
- f_2 appears in the infinite component of some type rule in D.

Here the relation $\trianglelefteq$ is the embedding relation (Definition 4) applied to types. For example, $list(A) \trianglelefteq list(list(A))$ where $list/1$ is a type symbol. As another example, given D containing $\tau \to F; I$ with $F = \{1, 2\}$ and $I = \mathbb{N} \setminus \{1, 2\}$ then $(1, \tau) \not\preceq_{\Sigma,D} (2, \tau)$ and $(1, \tau) \preceq_{\Sigma,D} (5, \tau)$.

Lemma 1. *Let D be a type definition, Σ a set of function symbols and $\Sigma_\mathcal{T}$ a finite set of type symbols. Assume that every function symbol in Σ appears in some type rule in D, and that every type symbol in $\Sigma_\mathcal{T}$ appears on the left of some rule in D. Then $\preceq_{\Sigma,D}$ is a* wqo *on the set $\Sigma \times \mathcal{T}$.*

Proof. It can easily be verified that $\preceq_{\Sigma,D}$ is reflexive and transitive, as required by Definition 2. Now, we prove the wbr property (Definition 3). The proof is by contradiction. Assume that there is an infinite sequence of pairs from $\Sigma \times \mathcal{T}$ of the form $(f_0, \tau_0), (f_1, \tau_1), \ldots$, and for all i, j, $i < j \to (f_i, \tau_i) \not\preceq_{\Sigma,D} (f_j, \tau_j)$. We distinguish two cases:

(i) First, assume that there is a finite number of function symbols from Σ occurring in the sequence. Then there must exist some f occurring infinitely often in the sequence, say $(f, \tau_{k_1}), (f, \tau_{k_2}), \ldots$. The relation $\trianglelefteq$ is a

wqo on $\mathcal{T}$ since $\Sigma_{\mathcal{T}}$ is finite. Hence there must exist i, j such that $i < j$ and $\tau_{k_i} \trianglelefteq \tau_{k_j}$. Hence $(f, \tau_{k_i}) \preceq_{\Sigma,D} (f, \tau_{k_j})$ which contradicts the assumption.

(ii) Second, assume that there is an infinite set of function symbols from Σ occurring in the sequence. Then there must exist some $j > 0$, such that f_j is in the infinite component of some type rule in D, in which case $(f_i, \tau_i) \preceq_{\Sigma} (f_j, \tau_j)$ for all $i < j$ which contradicts the assumption.

Hence, there are no such infinite sequences and together with reflexivity and transitivity this establishes that $\preceq_{\Sigma,D}$ is a wqo. □

The next definition presents our notion of type-based homeomorphic embedding, $\trianglelefteq_T$, based on the above relation $\preceq_{\Sigma,D}$. Since we assume that the relation $\preceq_S$ is true for all arguments, we omit it together with its associated condition (which is trivially true) in the definition below.

Definition 9 *(*TbHEm*, $\trianglelefteq_T$).* Given a type definition D and the relation $\preceq_{\Sigma,D}$, the embedding relation over typed expressions, written $\trianglelefteq_T$, is defined by the following rules:

(i) $Y : \tau_Y \trianglelefteq_T X : \tau_X$ for all variables X, Y;

(ii) $s : \tau \trianglelefteq_T f(t_1, \ldots, t_n) : \tau'$ if $s : \tau \trianglelefteq_T t_i : \tau'_i$ for some i, where $\tau' \to \ldots; f(\tau'_1, \ldots, \tau'_n); \ldots$ is an instance of a type rule in D;

(iii) $f(s_1, \ldots, s_n) : \tau \trianglelefteq_T g(t_1, \ldots, t_m) : \tau'$ if
  (a) $(f, \tau) \preceq_{\Sigma,D} (g, \tau')$ and
  (b) $\exists i_1, \ldots, i_n$ such that $1 \leqslant i_1 < \cdots < i_n \leqslant m$ and $\forall j \in \{1, \ldots, n\}$, $s_j : \tau_j \trianglelefteq_T t_{i_j} : \tau'_{i_j}$, where $\tau_1, \ldots, \tau_n$ (resp. $\tau'_{i_1}, \ldots, \tau'_{i_n}$) are the types of $s_1, \ldots, s_n$ (resp. $t_{i_1}, \ldots, t_{i_n}$).

The following theorem states the *soundness* of TbHEm. Informally, it states that infinite sequences cannot be built when the TbHEm relation is used to blow the whistle. This guarantees that TbHEm can be safely used in online tools (see Section 1).

Theorem 10 *(Soundness). For all infinite sequences $e_1 : \tau_1$, $e_2 : \tau_2, \ldots$ of typed expressions with respect to a type definition D there exists $i < j$ such that $e_i : \tau_i \trianglelefteq_T e_j : \tau_j$.*

Proof. The proof amounts to demonstrating that $\trianglelefteq_T$ is a wqo (see Definition 3) on typed expressions. By Theorem 4 from [13], this in turn follows if $\preceq_{\Sigma,D}$ is a wqo. As the sequence consists of typed terms with respect to some type definition D, all the types occurring in the sequence are constructed from the finite set of type symbols occurring in D and hence $\preceq_{\Sigma,D}$ is a wqo by Lemma 1. Hence the main result follows. □

4. A case-study in online partial evaluation

Partial evaluation (PE) [10] is a semantics-based program transformation technique whose purpose is to specialize a program with respect to the part of its input data which is known at specialization time. Essentially, a partial evaluator dynamically expands program states to propagate the known input data, giving rise to a (possibly infinite) sequence of expressions which represent states. HEm has proven to be very effective in practice to control PE: not only does it ensure termination, it often allows sequences to grow sufficiently large to obtain accurate results while at the same time stopping the computation as soon as potential redundancy is detected.

This section presents as case-study a classical and non-trivial application of online PE: the specialization of interpreters. In particular, we consider an interpreter (implemented in Prolog) for a simple, imperative, bytecode language. In theory [8], the specialization of such an interpreter with respect to a particular bytecode program allows transforming the bytecode program into a semantically equivalent version written in Prolog. In practice, the quality and usefulness of the transformation depends on the particular techniques used to control the process. We have implemented the proposed TbHEm relation within a partial evaluator of logic programs [19], together with the procedure for constructing a monomorphic well-typing devised by Bruynooghe et al. [5] to automatically infer the types.

In PE of interpreters, termination problems occur as soon as the bytecode program with respect to which we are specializing the interpreter has a loop or a recursion whose termination condition is undecidable at specialization time. Let us consider the bytecode program fragment below, which corresponds to the simple loop "*for(i = 0; i < n; i++){}*":

```
0:push(0); 1:store(i); 2:load(i);
3:load(n); 4:ifge(7); 5:inc(i);
6:goto(2); 7:...
```

The two instructions at program counters 0 and 1 initialize `i` to 0. Note that the bytecode language is stack-based, e.g., to perform the operation $i < n$ variables `i` and `n` are first pushed on the stack $(2, 3)$ so that the conditional branching `ifge` (i.e. if greater or equal) uses them. To understand the problem, it is enough to know that the interpreter manipulates an environment of the form $s(PC, LV)$ where PC is the program counter and LV is the list of local variables, in this case $[N, I]$. In the following we ignore variable N and the list constructor for simplicity, thus we write $s(PC, I)$. Given a program, the PC can only take a finite number of values, while the local variables can, in general, change infinitely. The well-typing analysis of [5] allocates the type τ to every $s(PC, I)$ expression such that:

$$\tau \to s(\tau_1, \tau_2),$$

$$\tau_1 \to F; I \quad \text{with } F = \{0, 1, \ldots, 7\} \text{ and } I = \mathbb{N} \setminus F,$$

$$\tau_2 \to \emptyset; \mathbb{N}.$$

During the specialization of the interpreter with respect to this particular program, the partial evaluator expands the interpreter states to propagate the information known from the bytecode program. The following (infinite) sequence of expressions arises $\ldots, s(2, 0), s(3, 0), s(4, 0), s(5, 0), s(6, 0), \underline{s(2, 1)}, s(3, 1), \ldots, s(6, 1), s(2, 2), s(3, 2), \ldots$.

The program counter loops in the interval [2..6], while variable I is infinitely incremented by one after each loop iteration. An optimal strategy should only expand the above sequence until the underlined expression $s(2,1)$ appears, which actually corresponds to a loop in the program. This allows transforming the loop into the Prolog code:

```
p(I,N) :- I >= N.
p(I,N) :- I < N, I1 is I+1, p(I1,N).
```

Such an optimal behavior is achievable by using TbHEm in combination with the above (automatically inferred) types. Note that $s(2,0):\tau \not\trianglelefteq_T s(3,0):\tau$ as $2:\tau_1 \not\trianglelefteq_T 3:\tau_1$ while $s(2,0):\tau \trianglelefteq_T s(2,1):\tau$ as $0:\tau_2 \trianglelefteq_T 1:\tau_2$.

However, stopping the derivation later causes unnecessary unrollings of the loop, thus producing an over-specialized program. E.g., this program is obtained when the sequence is stopped at $s(2,2)$ (two loop unrollings):

```
p(I,N) :- I >= N.
p(I,N) :- I < N, I1 is I+1, I1 >= N.
p(I,N) :- I < N, I1 is I+1, I1 < N,
          I2 is I1+1, p(I2,N).
```

This often highly degrades both the efficiency of the specialization process and the quality of the specialized program. Experimental evidence for this is shown in Section 6 below.

On the other hand, stopping the derivation earlier, e.g., at $s(3,0)$, as other techniques would do, results in a very poor specialization. Note that, in the limit, we could perform a single unfolding step per atom. This basically results in obtaining exactly the same interpreter we started from. Therefore, no gains have been achieved at all and PE does not remove the interpretation layer. Also, if we stop too early, many atoms will be filtered out in order to guarantee termination. This may result in an important information loss. Due to space limitations, we do not present a full algorithm for PE of logic programs here (see, e.g., [14] for more details). Note that as soon as we filter away the value of the *PC*, execution could proceed by any of the instructions in the bytecode program, which results in large specialization times and in residual programs with plenty of useless code.

5. Instances of type-based embedding

This section shows that existing relations based on embedding, which are currently used in state-of-the-art specialization tools (e.g., [2,14,13]) can be reconstructed as instances of TbHEm just by providing a particular type. Let us make a distinction between the *static* symbols occurring in the program and the goal, and the remaining ones, called the *dynamic* symbols. We use S_τ to denote the set of all $f(\tau,\ldots,\tau)$ where f is a static symbol.

5.1. Embedding with number filtering

In programs which contain arithmetic as the only way of generating an infinite number of symbols, a relatively straightforward solution in order to recover termination is to use the $\trianglelefteq_{num}$ relation. It is an adaptation of HEm which filters out numeric values, i.e., any number embeds any other number. The $\trianglelefteq_{num}$ relation could be reconstructed as a TbHEm assuming that every argument in every predicate is of type τ_{num} which is defined as:

$$\tau_{num} \to S_{\tau_{num}} \setminus num; num,$$

where *num* is the infinite set of all numbers.

Example 11. Let us re-consider the example in Section 4. The behavior of $\trianglelefteq_{num}$ is obtained using $\trianglelefteq_T$ by allocating the type τ to every expression of the form $s(PC,I)$, where:

$$\tau \to s(\tau_{num},\tau_{num}),$$

$$\tau_{num} \to \emptyset; \mathbb{N}.$$

Unfortunately, this modification to HEm is far too conservative and leads to excessive precision loss.

Example 12. The sequence in Section 4 is stopped too early by $\trianglelefteq_{num}$ as $s(2,0) \trianglelefteq_{num} s(3,0)$, thus breaking feature (i) in Section 1.

5.2. Static vs. dynamic symbols

TbHEm generalizes an idea sketched by Leuschel [13] to build an extended homeomorphic embedding based on a distinction between the static and the dynamic symbols. This relation is denoted in the following as $\trianglelefteq_S^*$. We introduce the following extra notation. We use *Dyn* to denote the infinite number of cases of the form $f(\tau_d,\ldots,\tau_d)$ where f is a dynamic symbol and $\tau_d \to \emptyset; Dyn$.

Then, $\trianglelefteq_S^*$ can be reconstructed as a TbHEm assuming that every argument in every predicate is of type τ_S which is defined as:

$$\tau_S \to S_{\tau_S}; Dyn.$$

Example 13. For our working example, every expression $s(PC,I)$ would be allocated the type τ:

$$\tau \to s(\tau_S,\tau_S),$$

$$\tau_S \to F; I \quad \text{with } F=\{0,1,\ldots,7\} \text{ and } I=\mathbb{N}\setminus F.$$

As discussed in Section 1, this relation lacks control over the different contexts in which static symbols occur in the program. For example, this relation makes no distinction between the set of values which an argument can take with respect to the set of values for other arguments: all static symbols in the program are put in the same set, regardless of where they come from or where they are used. Furthermore, the result of specializing a piece of code depends on whether in the same compilation unit there is a lot of (dead) code or not, since all static symbols, even if they appear in the context of completely unrelated subprograms will be considered as part of the finite component.

Example 14. Let us re-consider the situation in Section 4. Unlike $\trianglelefteq_{num}$, in this case we have that $s(2,0) \not\trianglelefteq_S^* s(3,0)$.

However, as both argument positions are given the same type, as opposed to the type given in Section 4, the loop will not be detected until an atom containing a number not occurring in the program arises in the sequence, namely $s(2,8)$, since $s(2,0) \trianglelefteq_S^* s(2,8)$. As explained in Section 4, this over expansion can highly degrade the efficiency of the specialization and the quality of the specialized programs.

6. Experimental evaluation

In order to measure the performance of TbHEm, we experimentally evaluated our case-study using TbHEm and compared the results against those obtained using $\trianglelefteq$, $\trianglelefteq_{num}$ and $\trianglelefteq_S^*$. We measured two aspects which are crucial in the specialization of interpreters, the specialization time and the residual program size. Both aspects are directly related to the quality of the decompilation.

From the experiments we conclude that $\trianglelefteq_T$ always guarantees termination (unlike $\trianglelefteq$) and behaves significantly better than $\trianglelefteq_{num}$ and $\trianglelefteq_S^*$. We compute the gain as *Old-Cost*/*New-Cost* and obtain an average gain of 2.3 in time and 14.4 in size with respect to $\trianglelefteq_{num}$, and 8.9 in time and 4.23 in size with respect to $\trianglelefteq_S^*$. Furthermore, $\trianglelefteq_T$ behaves at least as well as $\trianglelefteq$ in the examples in which $\trianglelefteq$ terminates, even after adding the additional cost taken by the well-typing analysis. We have observed that the largest gains are obtained when the sets of numbers in the different contexts do not intersect. In these cases, our method benefits from the context-sensitivity of TbHEm which directly contributes to obtaining smaller decompiled programs and times. As an example, for a *linear search* algorithm, we produce a 1.7 KB Prolog program in about 300 ms using $\trianglelefteq_T$, while we obtain a 9 KB Prolog program in 4 s using $\trianglelefteq_S^*$. For this one, $\trianglelefteq_{num}$ gets a 13.7 KB Prolog program in about 540 ms while $\trianglelefteq$ does not terminate.

7. Discussion

This note presents a novel, type-based, homeomorphic embedding relation (TbHEm) and proves its soundness. We show that existing approaches which extend the untyped embedding relation to handle infinite signatures can be reconstructed as instances of our TbHEm relation.

The practicality of our approach heavily depends on automatically being able to infer suitable types to be used in combination with TbHEm. We note first that the problem does not allow a precise, computable solution. Determining the exact set of symbols that can appear at run-time at a specific program point, and in particular determining whether the set is finite, is closely related to termination detection. Let us briefly describe existing methods developed in the context of logic programming to infer the required types. As pointed out in Section 6, a well-typing analysis for logic programs was described by Bruynooghe et al. [5]. The procedure scales well (roughly linear in program size) and is robust, in that every program has a well-typing. We have seen in Section 6 that one can first apply this analysis to infer well-typings and then achieve good specializations by using the well-typings in combination with TbHEm.

An important observation is that, in order to take full advantage of TbHEm in practice, it is not always necessary to know the actual type definitions. In particular it suffices to know whether the infinite component of type rules is (transitively) empty or not. As another way to infer the types, in a program with built-ins, we can use existing static analyses which allow determining that the type of an argument has a finite signature. We can provide this information without having to specify the exact type. A similar idea has been recently outlined by Ruggieri and Mesnard [20], where a type system for linear constraints and its use in mode analysis of CLP programs is presented.

Analyses exist that make over-approximations of the set of values that a program's numeric arguments can have. Polyhedral analyses and interval analyses are perhaps the most widely known of these and they have successfully been applied to constraint logic programs [3]. Such analyses can determine upper and lower bounds for arguments. If an argument is bounded from above and below, and it is known that such argument takes on integral values, then it can only take a finite set of values. In general, the better the derived types are, the further the sequences can be extended without risking non-termination. If the derived types have finite components that are too small, then it is more likely that sequences will be stopped too early; if they are too large, then sequences could be expanded too far, producing an unnecessary expansion.

Though we have outlined procedures to infer types in the context of logic programming, our type-based relation is not tied to any programming paradigm. Moreover, it can be used for a wide range of applications (as those mentioned in Section 1).

Acknowledgements

We gratefully thank the anonymous referees for many useful comments and suggestions. Preliminary material of this note appeared in the Proc. of LOPSTR 2007 [1]. This work was funded in part by the IST program of the European Commission, Future and Emerging Technologies under the IST-231620 *HATS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* and TIN-2008-05624 *DOVES* projects, the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project and the Danish Natural Science Research Council under grant FNU 272-06-0574 (SAFT).

References

[1] E. Albert, J. Gallagher, M. Gómez-Zamalloa, G. Puebla, Type-based homeomorphic embedding and its applications to online partial evaluation, in: 17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07), in: LNCS, vol. 4915, Springer-Verlag, 2008, pp. 23–42.

[2] E. Albert, M. Hanus, G. Vidal, A practical partial evaluation scheme for multi-paradigm declarative languages, Journal of Functional and Logic Programming 2002 (1) (2002).

[3] F. Benoy, A. King, Inferring argument size relationships with CLP(R), in: P.J. Gallagher (Ed.), Logic-Based Program Synthesis and Transformation (LOPSTR'96), in: LNCS, vol. 1207, Springer-Verlag, 1996, pp. 204–223.
[4] R. Bol, Loop checking in partial deduction, Journal of Logic Programming 16 (1–2) (1993) 25–46.
[5] M. Bruynooghe, J. Gallagher, W. Van Humbeeck, Inference of well-typings for logic programs with application to termination analysis, in: 12th International Static Analysis Symposium (SAS'05), in: LNCS, vol. 3672, Springer-Verlag, 2005, pp. 35–51.
[6] N. Dershowitz, Orderings for term-rewriting systems, Theoretical Computer Science 17 (1982) 279–301.
[7] N. Dershowitz, J.P. Jouannaud, Rewrite systems, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier, 1990, pp. 243–320.
[8] Y. Futamura, Partial evaluation of computation process – An approach to a compiler–compiler, Systems, Computers, Controls 2 (5) (1971) 45–50.
[9] G. Higman, Ordering by divisibility in abstract algebras, Proc. London Math. Soc. 2 (1952) 326–336.
[10] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, New York, 1993.
[11] J.B. Kruskal, Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture, Transactions of the American Mathematical Society 95 (1960) 210–225.
[12] M. Leuschel, On the power of homeomorphic embedding for online termination, in: G. Levi (Ed.), Proceedings of SAS'98, in: LNCS, vol. 1503, Springer-Verlag, 1998, pp. 230–245.
[13] M. Leuschel, Homeomorphic embedding for online termination of symbolic methods, in: The Essence of Computation, in: LNCS, vol. 2566, Springer, 2002, pp. 379–403.
[14] M. Leuschel, M. Bruynooghe, Logic program specialisation through partial deduction: Control issues, Theory and Practice of Logic Programming 2 (4–5) (2002) 461–515.
[15] M. Leuschel, B. Martens, Global control for partial deduction through characteristic atoms and global trees, in: 1996 Dagstuhl Seminar on Partial Evaluation, in: LNCS, vol. 1110, Schloß Dagstuhl, 1996, pp. 263–283.
[16] M. Leuschel, B. Martens, D. De Schreye, Controlling generalisation and polyvariance in partial deduction of normal logic programs, ACM Transactions on Programming Languages and Systems 20 (1) (1998) 208–258.
[17] R. Marlet, Vers une formalisation de l'èvaluation partielle. PhD thesis, Universitè de Nice - Sophia Antipolis, December 1994.
[18] B. Martens, D. De Schreye, Automatic finite unfolding using well-founded measures, Journal of Logic Programming 28 (2) (1996) 89–146.
[19] G. Puebla, E. Albert, M. Hermenegildo, Efficient local unfolding with ancestor stacks for full Prolog, in: Proc. of LOPSTR'04, in: LNCS, vol. 3573, Springer, 2005, pp. 149–165.
[20] S. Ruggieri, F. Mesnard, Typing linear constraints for moding CLP(r) programs, in: 14th International Symposium on Static Analysis, in: LNCS, vol. 5079, Springer, 2008, pp. 128–143.
[21] D. Sahlin, Mixtus: An automatic partial evaluator for full Prolog, New Generation Computing 12 (1) (1993) 7–51.
[22] Z. Somogyi, F. Henderson, T. Conway, The execution algorithm of Mercury: An efficient purely declarative logic programming language, Journal of Logic Programming 29 (1–3) (1996) 17–64.
[23] M.H. Sørensen, R. Glück, An algorithm of generalization in positive supercompilation, in: Proc. of ILPS'95, The MIT Press, 1995, pp. 465–479.

Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

Modular Decompilation of Low-Level Code by Partial Evaluation*

Miguel Gómez-Zamalloa
DSIC, Complutense University of Madrid
mzamalloa@fdi.ucm.es

Elvira Albert
DSIC, Complutense University of Madrid
elvira@fdi.ucm.es

Germán Puebla
CLIP, Technical University of Madrid
german@fi.upm.es

Abstract

Decompiling low-level code to a high-level intermediate representation facilitates the development of analyzers, model checkers, etc. which reason about properties of the low-level code (e.g., bytecode, .NET). Interpretive decompilation consists in partially evaluating an interpreter for the low-level language (written in the high-level language) w.r.t. the code to be decompiled. There have been proofs-of-concept that interpretive decompilation is feasible, but there remain important open issues when it comes to decompile a real language: does the approach scale up? is the quality of decompiled programs comparable to that obtained by ad-hoc decompilers? do decompiled programs preserve the structure of the original programs? This paper addresses these issues by presenting, to the best of our knowledge, the first modular scheme to enable interpretive decompilation of low-level code to a high-level representation, namely, we decompile bytecode into Prolog. We introduce two notions of optimality. The first one requires that each method/block is decompiled just once. The second one requires that each program point is traversed at most once during decompilation. We demonstrate the impact of our modular approach and optimality issues on a series of realistic benchmarks. Decompilation times and decompiled program sizes are linear with the size of the input bytecode program. This demostrates empirically the scalability of modular decompilation of low-level code by partial evaluation.

1. Introduction

Decompilation of low-level code (e.g., bytecode) to an intermediate representation has become a usual practice nowadays within the development of analyzers, verifiers, model checkers, etc. For instance, in the context of *mobile* code, as the source code is not available, decompilation facilitates the reuse of existing analysis and model checking tools. In general, high-level intermediate representations allow abstracting away the particular language features and developing the tools on simpler representations. As a representative example, Java bytecode is decompiled to a rule-based representation in [1], to clause-based programs in [18], to a three-address code view of bytecodes in Soot [20] and to the typed procedural language BoogiePL in [5]. Also, PIC programs are transformed to logic programs in [10]. Rule-based representations used in declarative programming in general—and in Prolog in particular—provide a convenient formalism to define such intermediate representations. E.g., as it can be seen in [1, 18, 20, 10], the operand stack used in a low-level language can be represented by means of explicit logic variables and that its unstructured control flow can be transformed into recursion.

All above cited approaches (except [10]) develop *ad-hoc* decompilers to carry out the particular decompilations. An appealing alternative to the development of dedicated decompilers is the so-called *interpretive* decompilation by *partial evaluation* (PE) [11]. PE is an automatic program transformation technique which specializes a program w.r.t. part of its known input data. Interpretive compilation was proposed in Futamura's seminal work [6], whereby compilation of a program P written in a (*source*) programming language L_S into another (*object*) programming language L_O is achieved by specializing an interpreter for L_S written in L_O w.r.t. P. The advantages of interpretive (de-)compilation w.r.t. dedicated (de-)compilers are well-known and discussed in the PE literature (see, e.g., [3]). Very briefly, they include: *flexibility*, it is easier to modify the interpreter in order to tune the decompilation (e.g., observe new properties of interest); *easier to trust*, it is more difficult to prove that ad-hoc decompilers preserve the program semantics; *easier to maintain*, new changes in the language semantics can be easily reflected in the interpreter.

There have been several proofs-of-concept of interpretive (de-)compilation (e.g., [3, 10, 13]), but there remain interesting open issues when it comes to assess its power and/or limitations to decompile a real language: *(a) does the*

*This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.35

approach scale? (b) do (de-)compiled programs preserve the structure of the original ones? (c) is the "quality" of decompiled programs comparable to that obtained by dedicated decompilers? This paper answers these questions positively by proposing a modular decompilation scheme which can be steered to control the structure of decompiled code and ensure quality decompilations which preserve the original program's structure. Our main contributions are summarized as follows:

1. We present the problems of *non-modular* decompilation and identify the components needed to enable a modular scheme. This includes how to write an interpreter and how to control an *online* partial evaluator in order to preserve the structure of the original program w.r.t. method invocations.
2. We present a modular decompilation scheme which is correct and complete for the proposed big-step interpreter. The *modular-optimality* of the scheme allows addressing issue *(a)* by avoiding decompiling the same method more than once, and *(b)* by ensuring that the structure of the original program can be preserved.
3. We introduce an interpretive decompilation scheme for low-level languages which answers issue *(c)* by producing decompiled programs whose *quality* is similar to that of dedicated decompilers. This requires a *block-level* decompilation scheme which avoids code duplication and code re-evaluation.
4. We report on a prototype implementation which incorporates the above techniques and demonstrate it on an set of realistic Java bytecode programs.

For the sake of concreteness, our decompilation scheme is formalized in the context of logic programming but the techniques to enable modularity can also be applied to compilation for any instantiation of languages (not necessarily low-level languages).

2 Basics of Partial Deduction

We assume familiarity with basic notions of logic programming [16]. Executing a program P for a call A consists in building an *SLD tree* for $P \cup \{A\}$ and then extracting the *computed answers* from every non-failing branch of the tree. PE in logic programming (see e.g. [7]) builds upon the SLD trees mentioned above. We now introduce a generic function PE, which is parametric w.r.t. the *unfolding rule*, unfold, and the *abstraction operator*, abstract and captures the essence of most algorithms for PE of logic programs:

1: **function** PE $(P, \mathcal{A}, S_0)$
2: **repeat**
3: $T^{pe} := \mathsf{unfold}(S_i, P, \mathcal{A})$;
4: $S_{i+1} := \mathsf{abstract}(S_i, \mathit{leaves}(T^{pe}), \mathcal{A})$;
5: $i := i + 1$;
6: **until** $S_i = S_{i-1}$ % (modulo renaming)
7: **return** $\mathsf{codegen}(T^{pe}, \mathsf{unfold})$;

Function PE differs from standard ones in the use of the set of annotations $\mathcal{A}$, whose role is described below. PE starts from a program P, a (possibly empty) set of annotations $\mathcal{A}$ and an initial set of calls S_0. At each iteration, the so-called *local control* is performed by the unfolding rule unfold (L3), which takes the current set of terms S_i, the program and the annotations and constructs a *partial* SLD tree for each call in S_i. Trees are partial in the sense that, in order to guarantee termination of the unfolding process, it must be possible to choose *not* to further unfold a goal, and rather allow leaves in the tree with a non-empty, possibly non-failing, goal. The particular unfold operator determines which call to select from each goal and when to stop unfolding. The partial evaluator may have to build several SLD-trees to ensure that all calls left in the leaves are "covered" by the root of some tree. This is known as the *closedness* condition of PE [17]. In the *global control*, those calls in the leaves which are not covered are added to the new set of terms to be partially evaluated, by the operator abstract (L4). At the next iteration, an SLD-tree is built for such call. Thus, basically, the algorithm iteratively (L2-6) constructs partial SLD trees until all their leaves are covered by the root nodes. An essential point of the operator abstract is that it has to perform "generalizations" on the calls that have to be partially evaluated in order to avoid computing partial SLD trees for an infinite number of calls. A partial evaluation of P w.r.t. S is then systematically extracted from the resulting set of calls T^{pe} in the final phase, codegen in L7. The notion of *resultant* is used to generate a program rule associated to each root-to-leaf derivation of the SLD-trees for the final set of terms T^{pe}. Given an SLD derivation of $P \cup \{A\}$ with $A \in T^{pe}$ ending in B and θ be the composition of the mgu's in the derivation steps, the rule $\theta(A) : -B$ is called the *resultant* of the derivation. A PE is defined as the set of resultants associated to the derivations of the constructed partial SLD trees for all $P \cup T^{pe}$.

The notions of *completeness* and *correctness* of PE [7] ensure that the specialized program produces no less resp. no more answers than the original program. A sufficient condition to ensure completeness is that the specialized program is *closed* by the resulting set of terms T^{pe}. Intuitively, the closedness condition ensures that all calls which may arise during the computation of $P \cup S$ are instances of T^{pe} and hence there is a matching resultant for them (solutions are not lost). The abstraction operator is encharged of ensuring that the closedness condition is met by means of a proper generalization of calls. Correctness is achieved when the resulting set T^{pe} is independent, i.e., there are no two calls in T^{pe} which unify. Independence can be easily recovered by a post-processing of renaming, which often does argument filtering [7].

Finally, the role of the annotations $\mathcal{A}$ (often manually provided) in *offline* PE is to give information to the control operators to decide when to stop derivations in the local control and how to perform generalizations in the global control to ensure termination. In *online* PE, all control decisions are taken during the specialization phase, without the use of annotations. We trivially turn function PE into online by just ignoring the annotations.

3 Non-Modular Interpretive Decompilation

This section describes the state of the art in interpretive decompilation of low-level languages to Prolog, including recent work in [10, 2, 9, 3]. We do so by formulating non-modular decompilation in a generic way and identifying its limitations. The low-level language we consider, denoted as $\mathcal{L}_{bc}$, is a simple imperative bytecode language in the spirit of Java bytecode but, to simplify the presentation, without object-oriented features (our implementation supports full Java bytecode). It uses an operand stack to perform computations. It has an unstructured control flow with explicit conditional and unconditional `goto` instructions and manipulates only integer numbers. A bytecode program P_{bc} is organized in a set of methods which are the basic (de-)compilation units of $\mathcal{L}_{bc}$. The code of a method m, denoted $code(m)$, consists of a sequence of bytecode instructions $BC_m = < pc_0 : bc_0, \ldots, pc_{n_m} : bc_{n_m} >$ with $pc_0, \ldots, pc_{n_m}$ being consecutive natural numbers. The $\mathcal{L}_{bc}$ instruction set is:

BcInst ::= `push(x)` | `load(v)` | `store(v)` | `add` | `sub` | `mul` | `div` | `rem` | | `neg` | `if` ⋄ `(pc)` | `if0` ⋄ `(pc)` | `goto(pc)` | `return` | `call(mn)`

where ⋄ is a comparison operator (`eq`, `le`, `gt`, etc.), v a local variable, x an integer, pc an instruction index and mn a method name. Instructions `push`, `load` and `store` transfer values or constants from the local variables to the stack (and viceversa); `add`, `sub`, `mul`, `div`, `rem` and `neg` perform the usual arithmetic operations, being `rem` the division remainder and `neg` the arithmetic negation; `if` and `if0` are conditional branching instructions (with the special case of comparisons with 0); `goto` is an unconditional branching; `return` marks the end of methods and `call` invokes a method. A method m is uniquely determined by its name. We write $calls(m)$ to denote the set of all method names invoked within the code of m. We use $defs(P_{bc})$ to denote the set of *internal* method names defined in P_{bc}. The remaining methods are *external*. We say that P_{bc} is *self-contained* if $\forall m \in P_{bc}, calls(m) \subseteq defs(P_{bc})$, i.e., P_{bc} does not include calls to external methods.

3.1 Non-modular, Online Decompilation

We rely on the so-called "interpretive approach" to compilation by PE described in Sect. 1, also known as first Futamura projection [6]. In particular, the decompilation of a $\mathcal{L}_{bc}$-bytecode program P_{bc} to LP (for short LP-decompilation) might be obtained by specializing (with an LP partial evaluator) a $\mathcal{L}_{bc}$-interpreter written in LP w.r.t. P_{bc}. In Fig. 1 we show a fragment of a (small-step) $\mathcal{L}_{bc}$ interpreter implemented in Prolog, named $Int_{\mathcal{L}_{bc}}$. We assume that the code for every method in the bytecode program P_{bc} is represented as a set of facts `bytecode/3` such that, for every pair $pc_i : bc_i$ in the code for method m, we have a fact `bytecode(`m`,`pc_i`,`bc_i`)`. The state carried around by the interpreter is of the form `st(Fr,FrStack)` where `Fr` represents the current frame (environment) and `FrStack` the stack of frames (call stack) implemented as a list. Frames are of the form `fr(M,PC,OStack,LocalV)`, where `M` represents the current method, `PC` the program counter, `OStack` the operand stack and `LocalV` the list of local variables. Predicate `main/3`, given the method to be interpreted `Method` and its input method arguments `InArgs`, first builds the initial state by means of predicate `build_s0/3` and then calls predicate `execute/2`. In turn, `execute/2` calls predicate `step/3`, which produces `S2`, the state after executing the bytecode, and then calls predicate `execute/2` recursively with `S2` until we reach a `return` instruction with the empty stack. For brevity, we only show the definition of `step/3` for a selected set of instructions and omit the code of `build_s0/3` and `localVar_update/4`. The latter simply updates the value of a local variable. By using this interpreter, in a purely online setting, we define a *non-modular* decompilation scheme in terms of the generic function PE as follows.

```
main(Method,InArgs,Top) :-
   build_s0(Method,InArgs,S0), execute(S0,Sf),
   Sf = st(fr(_,_,[Top|_],_),_)).
execute(S,S) :-
   S = st(fr(M,PC,[_Top|_],_),[]),
   bytecode(M,PC,return,_).
execute(S1,Sf) :-
   S1 = st(fr(M,PC,_,_),_), bytecode(M,PC,Inst,_),
   step(Inst,S1,S2), execute(S2,Sf).

step(goto(PC),S1,S2) :-
   S1 = st(fr(M,_,S,LV),FrS),
   S2 = st(fr(M,PC,S,LV),FrS).
step(push(X),S1,S2) :-
   S1 = st(fr(M,PC,S,L),FrS), next(M,PC,PC2),
   S2 = st(fr(M,PC2,[X|S],L),FrS).
...
step(call(M2),S1,S2) :-
   S1 = st(fr(M,PC,OS,LV),FrS), split_OS(M2,OS,Args,OS3),
   build_s0(M2,Args,st(fr(M2,PC2,OS2,LV2),_)),
   S2 = st(fr(M2,PC2,OS2,LV2),[fr(M,PC,OS3,LV)|FrS]).
step(return,S1,S2) :-
   S1 = st(fr(_,_,[RV|_],_),[fr(M,PC,OS,LV)|FrS]),
   next(M,PC,PC2), S2 = st(fr(M,PC2,[RV|OS],LV),FrS).
```

Figure 1. Fragment of (small-step) $\mathcal{L}_{bc}$ interpreter

Definition [$\text{DECOMP}_{\mathcal{L}_{bc}}$] Given a self-contained $\mathcal{L}_{bc}$-bytecode program P_{bc}, the (non-modular) LP-decompilation of P_{bc} can be obtained as:

$$\text{DECOMP}_{\mathcal{L}_{bc}}(P_{bc}) = PE(Int_{\mathcal{L}_{bc}} \cup P_{bc}, \emptyset, S)$$

where S is the set of calls $\{main(m, \_, \_) \mid m \in defs(P_{bc})\}$.

```
int gcd(int x,int y){
  int res;
  while (y != 0){
    res = x%y; x = y;
    y = res;}
  return abs(x);}

int abs(int x){
  if (x < 0) return -x;
  else return x; }
```

```
int lcm(int x,int y){
  int gcd = gcd(x,y);
  if (gcd == 0) return 0;
  else return x*y/gcd;}

int fact(int x){
  if (x == 0)
    return 1;
  else
    return x*fact(x-1);}
```

```
Method gcd/2
 0:load(1)
 1:if0eq(11)
 2:load(0)
 3:load(1)
 4:rem
 5:store(2)
 6:load(1)
 7:store(0)
 8:load(2)
 9:store(1)
 10:goto 0
 11:load(0)
 12:call(abs)
 13:return
```

```
Method abs/1
 0:load(0)
 1:if0ge(5)
 2:load(0)
 3:neg
 4:return
 5:load(0)
 6:return
```

```
Method lcm/2
 0:load(0)
 1:load(1)
 2:call(gcd)
 3:store(2)
 4:load(2)
 5:if0ne 8
 6:push(0)
 7:return
 8:load(0)
 9:load(1)
 10:mul
 11:load(2)
 12:div
 13:return
```

```
Method fact/1
 0:load(0)
 1:if0ne(4)
 2:push(1)
 3:return
 4:load(0)
 5:load(0)
 6:push(1)
 7:sub
 8:call(fact)
 9:mul
 10:return
```

Figure 2. Source code and $\mathcal{L}_{bc}$-bytecode for working example

Recent work in interpretive, online decompilation has focused on ensuring that the layer of interpretation is completely removed from decompiled programs, i.e., *effective* decompilations are obtained. This requires the use of advanced control strategies as explained in [2] and [9]. Our starting point is thus a state-of-the-art partial evaluator which incorporates such advanced techniques and which is able to remove the layer of interpretation.

3.2 Limitations

This section illustrates by means of the bytecode example in Fig. 2 that non-modular decompilation does not ensure a satisfactory handling of issues *(a)* and *(b)*. In the examples, we often depict the Java source code for clarity, but the partial evaluator works directly on the bytecode. The program consists of a set of methods that carry out arithmetic operations. Method `gcd` computes the greatest-common divisor, `abs` the absolute value, `lcm` the least-common multiple and `fact` the factorial recursively. The LP-decompilation obtained by applying Def. 3.1 is shown in Fig. 3. We identify the following limitations of non-modular decompilation:

(L1) Method invocations from `lcm` to `gcd` (index 2) and from `gcd` to `abs` (index 12) do not appear in the decompiled code. Instead, such calls have been *inlined* within their calling contexts and, as a consequence, the structure of the original code has been lost. For instance, the last two rules in the decompilation for `lcm`, `execute_1`, correspond to the `while` loop of `gcd`.

(L2) As a consequence, decompilation might become very inefficient. E.g., if n calls to the same method appear within a code, such method will be decompiled n times. This might be even worse in teh case of loops.

```
main(lcm,[B,0],A) :-
   B>0, C is B*0, A is C//B.
main(lcm,[0,0],0).
main(lcm,[B,0],A) :-
   B<0, D is B*0,
   C is -B, A is D//C.
main(lcm,[B,C],A) :-
   C\=0, D is B rem C,
   execute_1(C,D,B,C,A).

execute_1(A,0,B,C,D) :-
   A>0, E is B*C, D is E//A.
execute_1(0,0,_,_,0).
execute_1(A,0,B,C,D) :-
   A<0, E is-A,
   F is B*C, D is F//E.
execute_1(A,B,C,D,I) :-
   B\=0, K is A rem B,
   execute_1(B,K,C,D,I).
```

```
main(gcd,[A,0],A) :-A>=0.
main(gcd,[B,0],A) :-
   B<0, A is-B.
main(gcd,[B,C],A) :-
   C\=0, D is B rem C,
   execute_2(C,D,A) .

execute_2(A,0,A) :-
   A>=0.
execute_2(A,0,C) :-
   A<0, C is-A.
execute_2(A,B,G) :-
   B\=0,
   I is A rem B,
   execute_2(B,I,G).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :-
   B<0, A is-B.
```

Figure 3. Decompiled (non-modular) code for working example

(L3) The non-modular approach does not work incrementally, in the sense that it does not support *separate* decompilation of methods but rather has to (re)decompile all method calls. Thus, decompiling a real language becomes unfeasible, as one needs to consider system libraries. Limitation L2 together with L3 answer issue *(a)* negatively.

(L4) The decompiled program does not contain the code corresponding to recursive `fact` due to space limitations, as the decompiled code contains basically the whole interpreter. The problem with recursion is: assume we want to decompile method $m1$ whose code is $< pc_0 : bc_0, \ldots, pc_j : call(m1), \ldots, pc_n : return >$. There is an initial decompilation for $A_k = \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}, \texttt{lv}), []), \texttt{S}_\texttt{f})$ in which the call stack is empty. During its decompilation, a call of the form $A_l = \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}', \texttt{lv}'), [\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}, \texttt{lv})]), \texttt{S}_\texttt{f})$ with the call stack containing the previous frame appears when we get to the recursive call. At this point, the derivation must be stopped as $A_k \trianglelefteq_T A_l$. In order to ensure termination, the global control generalizes the above calls into $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \_, \_), \_), \texttt{S}_\texttt{f})$, where _ denotes a fresh variable and thus the call-stack has become unknown. As a consequence, after evaluating the $return$ statement, the continuation obtained from the call-stack is unknown and we produce the call $\texttt{execute}(\texttt{st}(\texttt{fr}(\_, \_, \_, \_), \_), \texttt{S}_\texttt{f})$ to be decompiled. Here, the fact that the method and the program counter are unknown prevents us from any chance of removing the interpretation layer, i.e., the decompiled code will potentially contain the whole interpreter. This indeed happens during the decompilation of `fact`. Partial solutions to the recursion problem exist and will be discussed later. Limitations L1 and L4 answer issue *(b)* negatively.

4 A Modular Decompilation Scheme

By *modular* decompilation, we refer to a decompilation technique whose decompilation unit is the method, i.e., we

decompile a method at a time. We show that this approach overcomes the four limitations of non-modular decompilation described in Sect. 3.2 and answers issues *(a)* and *(b)* positively. In essence, we need to: (i) give a compositional treatment to method invocations, we show that this can be achieved by considering a *big-step* interpreter; (ii) provide a mechanism to residualize calls in the decompiled program, we automatically generate program annotations for this purpose; (iii) study the conditions which ensure that *separate* decompilation of methods is sound.

4.1 Big-step Semantics Interpreter

Traditionally, two different approaches have been considered to define language semantics, *big-step* (or *natural*) semantics and *small-step* semantics (see, e.g., [12]). Essentially, in big-step semantics, transitions relate the initial and final states for each statement, while in small-step semantics transitions define the *next* step of the execution for each statement. In the context of bytecode interpreters, it turns out that most of the statements execute in a single step, hence making both approaches equivalent for such statements. This is the case for our $\mathcal{L}_{bc}$-bytecode interpreter for all statements except for *call*. The transition for *call* in small-step defines the next step of the computation, i.e., the current frame is pushed on the call-stack and a new environment is initialized for the execution of the invoked method. Note that, after performing this step, we do not distinguish anymore between the code of the caller method and that of the callee. This avoids modularity of decompilation.

In the context of interpretive (de-)compilation of imperative languages, small-step interpreters are commonly used (see e.g. [19, 10, 3]). We argue that the use of a big-step interpreter is a necessity to enable modular decompilation which scales to realistic languages. In Fig. 4, we depict the relevant part of the big-step interpreter for $\mathcal{L}_{bc}$-bytecode, named $Int^{bs}_{\mathcal{L}_{bc}}$. We can see that the *call* statement, after extracting the method parameters from the operand stack, calls recursively predicate `main/3` in order to execute the callee. Upon return from the method execution, the return value is pushed on the operand stack of the new state and execution proceeds normally. Also, we do not need to carry the call-stack explicitly within the state, but only the information for the current environment. I.e., states are of the form `st(M,PC,OStack,LocalV)`. This is because the call-stack is already available by means of the calls for predicate `main/3`.

The compositional treatment of methods in $Int^{bs}_{\mathcal{L}_{bc}}$ is not only essential to enable modular decompilation (overcome L1, L2 and L3) but also to solve the recursion problem in a simple and elegant way. Indeed, the decompilation based on the big-step interpreter $Int^{bs}_{\mathcal{L}_{bc}}$ does not present L4. E.g., the decompilation of a recursive method $m1$ starts from the

```
execute(S,S) :-
  S = st(M,PC,[_Top|_],_),
  bytecode(M,PC,return,_).
execute(S1,Sf) :-
  S1 = st(M,PC,_,_),
  bytecode(M,PC,Inst,_),
  step(Inst,S1,S2),
  execute(S2,Sf).
```

```
step(call(M2),S1,S2) :-
  S1 = st(M,PC,OS,LV),
  next(M,PC,PC2),
  split_OS(M2,OS,Args,OSRs),
  main(M2,Args,RV),
  S2 = st(M,PC2,[RV|OSRs],LV).
```

Figure 4. Fragment of big-step $\mathcal{L}_{bc}$ interpreter $Int^{bs}_{\mathcal{L}_{bc}}$

call $\texttt{main}(\texttt{m1}, \_, \_)$ and then reaches a call $\texttt{main}(\texttt{m1}, \texttt{args}, \_)$ where `args` represents the particular arguments in the recursive call. This call is flagged as dangerous by local control and the derivation is stopped. The important points are that, unlike before, no recomputation is needed as the second call is necessarily an instance of the first one and, besides, there is no information loss associated to the generalization of the call-stack, as there is no stack. The recursion problem was first detected in [8] and a solution based on computing regular approximations during PE was proposed. Although feasible in theory, such technique might be too inefficient in practice and problematic to scale it up to realistic applications due to the overhead introduced by the underlying analysis. Another solution is proposed in [10], where a simpler control-flow analysis is performed before PE in order to collect all possible instructions which might follow the $return$. Such information may then be used to recover information lost by the generalization. This solution turns out to be also impractical for our purposes when considering realistic programs that make intensive use of library code (e.g. Java bytecode) as many continuations can follow. Our solution does not require the use of static analysis and, as our experiments show, does not pose scalability problems.

4.2 Guiding Online PE with Annotations

We now present the annotations we use to provide additional control information to PE. They are instrumental for obtaining the quality decompilation we aim at. We use the annotation schema: "$[Precond] \Rightarrow Ann\ Pred$" where $Precond$ is an optional precondition defined as a logic formula, Ann is the kind of annotation ($Ann \in \{\mathbf{memo}, \mathbf{rescall}\}$) and $Pred$ is a predicate descriptor, i.e., a predicate function and distinct free variables. Such annotations are used by local control when a call for $Pred$ is found as follows:

- **memo**: The current call is not further unfolded. Therefore, the call is later transferred to the global control to carry out its specialization separately.
- **rescall**: The current call is not further unfolded. Unlike calls marked **memo**, the current call is not transferred to the global control.

In the following, we denote by $\mathsf{unfold}^{\mathcal{A}}_{\trianglelefteq_T}$ the unfolding operator of Sect. 2 enhanced to use the above annotations. We adopt the same names for the annotations as in offline PE

[15]. However, in offline PE they are the *only* means to control terminationand **rescall** annotations are in principle only used for builtins.

4.3 Modular Decompilation

In order to achieve modular decompilation, it is instrumental to allow performing *separate* decompilation. In the interpretive approach this requires being able to perform separate PE, i.e., to be able to specialize parts of the program independently and then join the specializations together to form the residual program. For instance, consider a self-contained logic program P partitioned in a set $\{P_1, \ldots, P_n\}$ of mutually disjoint subprograms which preserve predicate boundaries, i.e., for any predicate *pred* in P we have that all rules for *pred* are in the same partition P_j, for some $j \in \{1, \ldots, n\}$. Consider also the sets of terms $S_1, \ldots, S_n$ such that all calls in S_i correspond to predicates defined in P_i, $i = 1, \ldots, n$. We can now define $S = S_1 \cup \cdots \cup S_n$ and the usual notions of closedness and independence are applicable. A *separate* partial evaluation for P and S is obtained as the union of the individual specializations w.r.t. each corresponding set of calls, i.e., $\bigcup_{P_i \in P} PE(P_i, \emptyset, S_i)$. One additional difficulty for separate PE is related to the use of renaming for guaranteeing independence, since renaming requires a global table which is not available when generating code for the individual subprograms. A simple strategy which we will use in our modular decompilation is to allow polyvariant specialization for calls to predicates locally defined in the subprogram P_i being partially evaluated but to resort to monovariant specialization for predicates used across subprogram boundaries. Then, the renaming can use a local renaming table, which must guarantee that there will be no name clash with renamed calls from other subprograms.

We present now a modular decompilation scheme which, by combining the big-step interpreter with the use of **rescall** annotations, enables separate decompilation and ensures *soundness* (i.e., it is correct and complete w.r.t. internal methods).

Definition [MOD-DECOMP$_{\mathcal{L}_{bc}}$] Given a $\mathcal{L}_{bc}$-bytecode program P_{bc}, a modular LP-decompilation of P_{bc} can be obtained as:

$$\text{MOD-DECOMP}_{\mathcal{L}_{bc}}(P_{bc}) = \bigcup_{\forall m \in \mathit{defs}(P_{bc})} PE(\mathit{Int}^{bs}_{\mathcal{L}_{bc}} \cup \mathit{code}(m), \mathcal{A}_{mod}, S_m)$$

where the set of annotations $\mathcal{A}_{mod} = \{(m \in \mathit{calls}(P_{bc})) \Rightarrow \textbf{rescall}\ main(m, \_, \_)\}$ and the initial sets of calls $S_m = \{main(m, \_, \_)\}$ for each $m \in \mathit{defs}(P_{bc})$.

Let us briefly explain the above definition. Now the function PE is executed once per method defined in P_{bc}, starting each time from a set of calls, S_m, which contains a call of the form `main(m,_,_)` for method `m`. The set $\mathcal{A}_{mod}$ contains a **rescall** annotation which affects all methods invoked (but not necessarily internal) inside P_{bc}. When a method invocation is to be decompiled, the call `step(call(m'),_,_)` occurs during unfolding. We can see that, by using the big-step interpreter in Fig. 4, a subsequent call `main(m',_,_)` will be generated. As there is a **rescall** annotation which affects all methods invoked in the program, such call is not unfolded but rather remains residual. If m' is internal, a corresponding decompilation from the call `main(m',_,_)` will be, or has already been, performed since function PE is executed for every method in P_{bc}. Thus, completeness is ensured for internal predicates.

Example 1 *By applying function* MOD-DECOMP$_{\mathcal{L}_{bc}}$ *to the $\mathcal{L}_{bc}$-bytecode program in Fig. 2 we execute PE once for each of the four methods in the program. In each execution we specialize the interpreter w.r.t. the calls* `main(fact,_,_)`, `main(gcd,_,_)`, `main(lcm,_,_)`, *and* `main(abs,_,_)`. *We obtain the following LP-decompilation:*

```
main(lcm,[B,C],A) :-
   main(gcd,[B,C],D),
   D\=0, E is B*C,
   A is E//D.
main(lcm,[A,B],0) :-
   main(gcd,[A,B],0).

main(gcd,[B,0],A) :-
   main(abs,[B],A).
main(gcd,[B,C],A) :-
   C\=0, D is B rem C,
   exec_1(C,D,A).
```

```
exec_1(A,0,C) :-
   main(abs,[A],C).
exec_1(A,B,F) :- B\=0,
   H is A rem B, exec_1(B,H,F).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is-B.

main(fact,[B],A) :-
   B\=0, C is B-1,
   main(fact,[C],D), A is B*D.
main(fact,[0],1).
```

The structure of the original program w.r.t. method calls is preserved, as the residual predicate for `lcm` *contains an invocation to the definition of* `gcd`*, which in turn invokes* `abs`*, as it happens in the original bytecode. Moreover, we now obtain an effective decompilation for the recursive method* `fact` *where the interpretive layer is completely removed without the need of any analysis. Thus, L1 and L4 have been successfully solved.*

The following theorem ensures the soundness of modular decompilation for the big-step bytecode interpreter. Completeness can be ensured by excluding calls to external methods not defined in the bytecode. It is independent of the way the interpreter is defined, as the closedness condition for the internal methods is enforced by our definitions of $\mathcal{A}_{mod}$ and S_m. Correctness holds in the case of our interpreter, because the only calls which are transferred to the global control are instances of `main/3` and `execute/2` and their first argument is the method's name, which makes them mutually exclusive. A post-processing of renaming is thus optional, but it can be necessary to ensure that the independence condition is met for other interpreters.

Theorem 1 (soundness) *Consider a $\mathcal{L}_{bc}$-bytecode program P_{bc} and a concrete input I. Let P'_{bc} be the result of* MOD-DECOMP$_{\mathcal{L}_{bc}}(P_{bc})$ *and I' the LP representation of I. Then, A' is an answer for $P'_{bc} \cup \{I'\}$ iff A is the result of executing P_{bc} for the input I, where A' is the LP representation of A.*

We now characterize the notion of *modular-optimality* in decompilation which ensures that (1) only the code associated to internal methods is decompiled, thus, we can have external calls (e.g., to libraries) which are not decompiled and overcome L3; (2) and each method is decompiled only once and thus we overcome L2.

Proposition 1 (modular-optimality) *Given a $\mathcal{L}_{bc}$-bytecode program P_{bc}, function* MOD-DECOMP$_{\mathcal{L}_{bc}}$ *only decompiles the code corresponding to internal methods defined in P_{bc}, and the code of each method is decompiled once.*

Note that modular decompilation gives a monovariant treatment to methods in the sense that it does not allow creating specialized versions of method definitions. This is against the usual spirit in PE, where polyvariance is a main goal to achieve further specialization. However, in the context of decompilation, we have shown that a monovariant treatment is necessary to enable scalability and to preserve program structure. It naturally raises the question whether a polyvariant treatment could achieve, even if at the cost of efficiency and loss of structure, a better quality decompilation. Note that enabling polyvariant specialization in the modular setting can be trivially done by not introducing **rescall** annotations for certain selected methods which should be treated in a polyvariant manner. Our experience indicates that there is often a small quality gain at the price of a highly inefficient decompilation.

5 Decompilation of Low-Level Languages

Applying the interpretive approach on a low-level language introduces new challenges. The main issue is whether it is possible to obtain, by means of interpretive decompilation, programs whose *quality* is equivalent to that obtained by dedicated decompilers, issue *(c)* in Sect. 1. We will show now that, using the most effective unfolding strategies of PE, code for the same program point can be emitted (i.e. it can be decompiled) several times, which degrades both efficiency and quality of decompilation. In order to obtain results which are comparable to that of dedicated decompilers, it makes sense to use similar heuristics. Since decompilers first build a *control flow graph* (CFG) for the method, which guides the decompilation process, we now study how a similar notion can be used for controlling PE of the interpreter.

Let us explain *block-level* decompilation by means of an example. Consider the method m_{bl} to the left of Fig. 5, where we only show the relevant bytecode instructions, and its CFG in the center. A *divergence point* (D point) is a program point (bytecode index) from which more than one branch originates; likewise, a *convergence point* (C point) is a program point where two or more branches merge. In the CFG of m_{bl}, the only divergence (resp. convergence) point is pc_i (resp. pc_k).

By using the decompilation scheme presented so far, we obtain the SLD-tree shown to the right of Fig. 5, in which all calls are completely unfolded as there is no termination risk (nor **rescall** annotation). The decompiled code is shown under the tree. We use $\{\texttt{res}_\texttt{X}\}$ to refer to the residual code emitted for BlockX and $\texttt{cond}_\texttt{i}$ to refer to the condition associated to the branching instruction at $\texttt{pc}_\texttt{i}$ ($\overline{\texttt{cond}_\texttt{i}}$ denotes its negation). The quality of the decompiled code is not optimal due to:

D. Decompiled code $\{\texttt{res}_\texttt{A}\}$ for BlockA is duplicated in both rules. During PE, this code is evaluated once but, due to the way resultants are defined (see codegen in Sect. 2), each rule contains the decompiled code associated to the whole branch of the tree. This code duplication brings in two problems: it increases considerably the size of decompiled programs and also makes their execution slower. For instance, when $\overline{\texttt{cond}_\texttt{i}}$ holds, the execution goes unnecessarily through $\{\texttt{res}_\texttt{A}\}$ in the first rule, fails to prove $\texttt{cond}_\texttt{i}$ and, then, attempts the second rule.

C. Decompiled code of BlockD is again emitted more than once. Each rule for the decompiled code contains a (possibly different) version, $\{\texttt{res}_\texttt{D}\}$ and $\{\texttt{res}'_\texttt{D}\}$, of the code of BlockD. Unlike above, at PE time, the code of BlockD is actually evaluated in the context of $\{\texttt{cond}_\texttt{i}, \{\texttt{res}_\texttt{B}\}\}$ and then re-evaluated in the context of $\{\overline{\texttt{cond}_\texttt{i}}, \{\texttt{res}_\texttt{C}\}\}$. Convergence points thus might degrade both efficiency (and endanger scalability) and quality of decompilation (due to larger residual code).

The amount of repeated residual code grows exponentially with the number of C and D points and the amount of re-evaluated code grows exponentially with the number of C points. Thus, we now aim for a *block-level* decompilation that helps overcome problems D and C above. Intuitively, a block-level decompilation must produce a residual rule for each block in the CFG. This can be achieved by building SLD-trees which correspond to each single block, rather than expanding them further.

The **memo** annotations presented in Sect. 4.2 facilitate the design of the block-level interpretive decompilation scheme. In particular, we can easily force the unfolding process to stop at D points by including a **memo** annotation for $\texttt{execute}/2$ calls whose PC corresponds to a D point. In

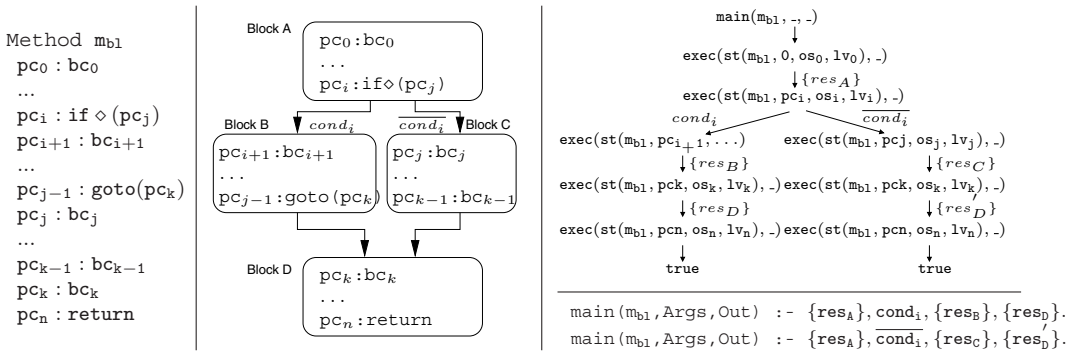


Figure 5. $\mathcal{L}_{bc}$-bytecode, CFG, unfolding tree and decompiled code of m_{bl} method

the example, unfolding stops at pc_i as desired. Regarding C points, an additional requirement is to partially evaluate the code on blocks starting at these points at most once. The problem is similar to the polyvariant vs monovariant treatment in the decompilation of methods in Sect. 4.3, by viewing entries to blocks as method calls. Not surprisingly, the solution can be achieved similarly in our setting by: (1) stopping the derivation at `execute`/2 calls whose PC corresponds to C points and (2) passing the call to the global control, and ensuring that it is evaluated in a sufficiently generalized context which covers all incoming contexts. The former point is ensured by the use of **memo** annotations and the latter by including in the initial set of terms a generalized call of the form $\texttt{execute}(\texttt{st}(\texttt{m}_{\texttt{bl}}, \texttt{pc}_{\texttt{k}}, \_, \_), \_)$ for all C points, which forces such generalization. The next definition presents the *block-level* decompilation scheme where div_points(m) and conv_points(m) denote, resp., the set of D points and C points of a method m.

Definition [BLOCK-MOD-DECOMP$_{\mathcal{L}_{bc}}$] Given a $\mathcal{L}_{bc}$-bytecode program P_{bc}, a block-level, modular LP-decompilation of P_{bc} can be obtained as:

$$\text{BLOCK-MOD-DECOMP}_{\mathcal{L}_{bc}}(P_{bc}) = \bigcup_{\forall m \in \mathit{defs}(P_{bc})} PE(\mathit{Int}^{bs}_{\mathcal{L}_{bc}} \cup \mathit{code}(m), \mathcal{A}_m, S_m)$$

$$\begin{aligned}
\mathcal{A}_{blocks} &= \{pc \in \text{div_points}(m) \cup \text{conv_points}(m) \Rightarrow \\
&\quad \textbf{memo}\ \mathit{execute}(\mathit{st}(m, pc, _, _), _)\} \\
S_m &= \{\mathit{main}(m, _, _)\} \cup \\
&\quad \{\mathit{execute}(\mathit{st}(m, pc, _, _), _) \mid pc \in \text{conv_points}(m)\} \\
\mathcal{A}_m &= \mathcal{A}_{mod} \cup \mathcal{A}_{blocks}, \text{ for each } m \in \mathit{defs}(P_{bc}).
\end{aligned}$$

An important point is that, unlike annotations used in offline PE [13] which are generated by only taking the interpreter into account, our annotations for block-level decompilation are generated by taking into account the particular program to be decompiled. Importantly, both the annotations and the initial set of calls can be computed automatically by performing two passes on the bytecode (see, e.g., [1, 20]). The result of performing block-level decompilation on m_{bl} is:

```
main(mbl,Args,Out) :- {resA}, execute1(...).
execute1(...) :- condi, {resB}, execute2(...).
execute1(...) :- ¬condi, {resC}, execute2(...).
execute2(...) :- {resD}.
```

Now, the residual code associated to each block appears once in the code. This ensures that block-level decompilation preserves the CFG shape as dedicated decompilers do. Thus, the quality of our decompiled code is as good as that obtained by state-of-the-art decompilers [1, 18] but with the advantages of interpretive decompilation (see Sect. 1). We formalize the quality of block-level decompilation.

Proposition 2 (block-optimality) *Given a bytecode program P_{bc}, the block-level decompilation function* BLOCK-MOD-DECOMP$_{\mathcal{L}_{bc}}$ *ensures that: (I) residual code for each bytecode instruction in P_{bc} is emitted once in the decompiled program, and (II) each bytecode instruction in P_{bc} is evaluated at most once during PE.*

6 Experimental Evaluation

We report on our implementation of a decompiler for full (sequential) Java Bytecode into Prolog. The extensions needed to handle the features of Java Bytecode not considered in $\mathcal{L}_{bc}$ (exception handling, dynamic memory allocation, object orientation, etc) are basically carried out by making the decompiler produce the corresponding builtins in the residual code. E.g. the bytecode instruction *putfield* will make the decompiler produce the predicate `set_field/5` in the decompiled code. This naive solution might be considerably improved to increase the precision and quality of the decompilation. However this is out of the scope of this paper. For the experimental evaluation we have used the set of benchmarks in the JOlden suite [4]. Most programs make an extensive use of library methods. Hence, non-modular decompilation cannot be assessed as we run into memory problems when trying to decompile the code of library calls. The experiments have been performed on an Intel Core 2 Duo 1.86GHz with 2GB of RAM, running Linux. Figure 6 depicts four charts measuring different aspects of the decompilation. We assess the differences between the *modular* and the *modular+block-level* (just *block-level* for short) approaches; as well as how the size of the programs affects the decompilation. We measure two as-

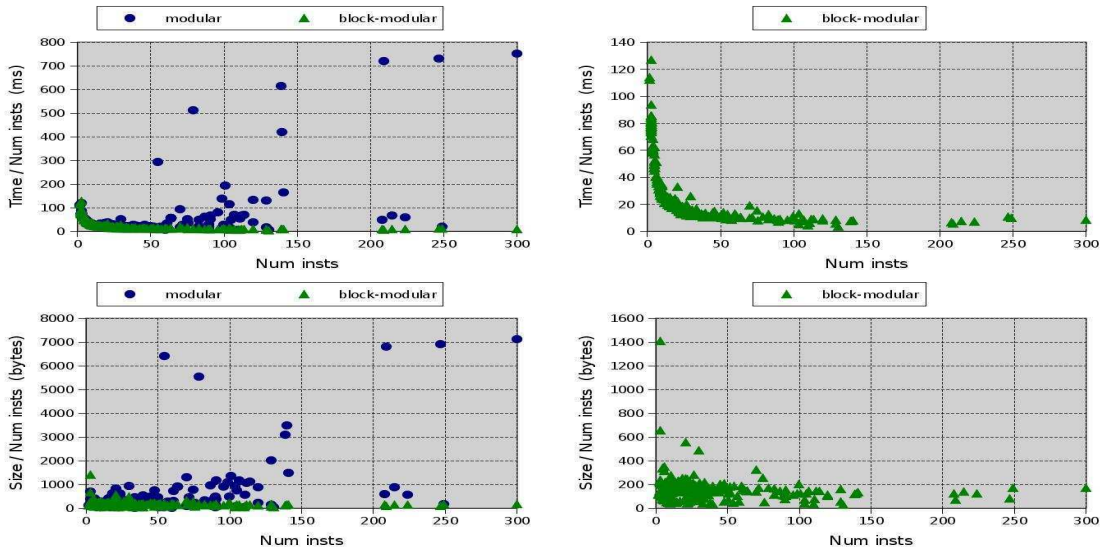


Figure 6. Evaluating *modular* decompilation vs. *modular+block-level* deompilation with the JOlden Suite

pects of the decompilation: the decompilation time (in milliseconds) per instruction and the decompiled program size (in bytes) per instruction. The decompilation time indicates the efficiency of the process and the size of decompiled programs is directly related to the decompilation quality. Each point $[X, Y]$ in the charts corresponds to the decompilation of a single method in the JOlden suite, where X represents the number of instructions of the method and Y the measured data (time or decompiled program size). The tables in the left-hand side show the data obtained (times in the top chart and sizes in the bottom one) for both the modular and the block-level decompilation. The variations in the block-level decompilation cannot be appreciated when combined with modular. Thus, we include in the tables on the right-hand side the figures for the block-level decompilation in isolation such that we adjust the scale on the Y-axis to the domain of the data.

From the charts, we conclude: (1) Times per instruction are notably larger for the smallest methods, as can be seen by looking at the initial curve in the charts. This is because the overhead introduced for starting a new decompilation is more noticeable when the time for decompilation itself is small, while it becomes negligible for larger methods. The same happens for the size of the decompiled programs. (2) Block-level decompilation achieves important speedups in general (for all methods with more than 40 instructions). Besides, it obtains significantly smaller decompiled programs. The speedups per package range from 3.36 in **power** to 31.4 in **bisort** for the decompilation times; and from 2.5 times smaller in **power** to 9 times smaller in **bisort** for the decompiled program sizes. Note that there is a clear correspondence between both measures, since C points introduce both inefficiency and size increase in decompilation, as explained in Sect. 5. Moreover, modular decompilation runs out of memory for some of the largest methods. This is again related to code duplication (C and D points) and (re-)evaluation (C points), which grow exponentially. (3) The most important conclusion is that, while in modular decompilation both the times and the sizes per instruction greatly increase with the size of the benchmarks, this does not happen in the block-level scheme. In block-level decompilation, these figures are totally stable (mostly constant) for all methods with more than 40 instructions. This demonstrates that both the decompilation times and the decompiled program sizes are *linear* with the size of the input bytecode program, thus demonstrating the scalability of the block-level decompilation. One might wonder why there are still small variations in the ratio. In our experience, the following points also matter: 1) the complexity of the control flow of the methods, 2) the relative complexity of the bytecode instructions used, e.g., instructions which operate in the heap tend to produce more residual code, 3) the structure w.r.t. methods of the program, e.g., classes with methods of medium size tend to result in better decompilations than those with few large methods or many small ones.

7 Conclusions and Related Work

We argue that *declarative languages* and the technique of *partial evaluation* have nowadays a large application field within the development of analysis, verification, and model checking tools for modern programming languages. On one hand, declarative languages provide a convenient intermediate representation which allows (1) representing all iterative constructs (loops) as recursion, independently of whether they originate from iterative loops (conditional and unconditional jumps) or recursive calls, and (2) all variables in the local scope of the methods (formal parameters, local variables, fields, and stack values in low-level languages) can be represented uniformly as explicit arguments of a declar-

ative program. On the other hand, the technique of partial evaluation enables the automatic (de-)compilation of a (complicated) modern program to a simple declarative representation by just writing an interpreter for the modern language in the corresponding declarative language and using an existing partial evaluator. The resulting intermediate representation greatly simplifies the development of the above tools for modern languages and, more interestingly, existing advanced tools developed for declarative programs (already proven correct and effective) can be directly applied on it.

Previous work in *interpretative* (de-)compilation has mainly focused on proving that the approach is feasible for small interpreters and medium-sized programs. The focus has been on demonstrating its *effectiveness*, i.e., that the so-called interpretation layer can be removed from the compiled programs. To achieve effectiveness, offline [13], online [3, 10, 19] and hybrid [14] PE techniques have been assessed and novel control strategies have been proposed and proved effective [9, 2]. Our work starts off from the premise that interpretive decompilation is feasible and effective as proved by previous work and studies further issues which have not been explored before. A main objective of our work is to investigate, and provide the necessary techniques, to make interpretive decompilation scale in practice. A further goal is to ensure, and provide the techniques, that decompiled programs preserve the structure of the original programs and that its quality is comparable to that obtained by dedicated decompilers. We believe that the techniques proposed in this paper, together with their experimental evaluation, provide for the first time actual evidence that the interpretive theory proposed by Futamura in the 70s is indeed an appealing and feasible alternative to the development of ad-hoc decompilers from modern languages to intermediate representations.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In R. D. Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[2] E. Albert, J. Gallagher, M. Gómez-Zamalloa, and G. Puebla. Type-based Homeomorphic Embedding and its Applications to Online Partial Evaluation. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, number 4915 in LNCS, pages 23–42. Springer-Verlag, 2008.

[3] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *9th Int. Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 124–139. Springer-Verlag, January 2007.

[4] J. S. Collection. `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`.

[5] R. DeLine and K. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[6] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[7] J. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98. ACM Press, 1993.

[8] J. Gallagher and J. Peralta. Using regular approximations for generalisation during partial evaluation. In *Proc. of the SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 44–51. ACM Press, 2000.

[9] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Improving the Decompilation of Java Bytecode to Prolog by Partial Evaluation. In *ETAPS Ws on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, volume 190 of *ENTCS*, pages 85–101, 2007.

[10] K. S. Henriksen and J. P. Gallagher. Abstract interpretation of pic programs through logic programming. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 184–196. IEEE Computer Society, 2006.

[11] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.

[12] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *POPL*, pages 144–154, 1993.

[13] M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 340–375. Springer, 2004.

[14] M. Leuschel, S. Craig, and D. Elphick. Supervising offline partial evaluation of logic programs using online techniques. In *LOPSTR*, volume 4407 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2006.

[15] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in prolog using a hand-written compiler generator. *TPLP*, 4(1–2):139 – 191, 2004.

[16] J. Lloyd. *Foundations of Logic Programming*. Springer, 2nd Ext. Ed., 1987.

[17] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[18] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.

[19] J. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In *Proc. of SAS'98*, volume 1503 of *LNCS*, pages 246–261, 1998.

[20] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.

Information and Software Technology 51 (2009) 1409–1427

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Decompilation of Java bytecode to Prolog by partial evaluation

Miguel Gómez-Zamalloa [a,*], Elvira Albert [a], Germán Puebla [b]

[a] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[b] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

ARTICLE INFO

Article history:
Available online 5 May 2009

Keywords:
Program transformation
Partial evaluation
Decompilation
Interpreters
Java bytecode
Logic programming

ABSTRACT

Reasoning about Java bytecode (JBC) is complicated due to its unstructured control-flow, the use of three-address code combined with the use of an operand stack, etc. Therefore, many static analyzers and model checkers for JBC first convert the code into a higher-level representation. In contrast to traditional decompilation, such representation is often not Java source, but rather some intermediate language which is a good input for the subsequent phases of the tool. *Interpretive decompilation* consists in partially evaluating an interpreter for the compiled language (in this case JBC) written in a high-level language with respect to the code to be decompiled. There have been proofs-of-concept that interpretive decompilation is feasible, but there remain important open issues when it comes to decompile a real language such as JBC. This paper presents, to the best of our knowledge, the first modular scheme to enable interpretive decompilation of a realistic programming language to a high-level representation, namely of JBC to Prolog. We introduce two notions of optimality which together require that decompilation does not generate code more than once for each program point. We demonstrate the impact of our modular approach and optimality issues on a series of realistic benchmarks. Decompilation times and decompiled program sizes are linear with the size of the input bytecode program. This demonstrates empirically the scalability of modular decompilation of JBC by partial evaluation.

1. Introduction

Decompilation of Java bytecode (JBC for short) to an intermediate representation has become a usual practice nowadays within the development of analyzers, verifiers, model checkers, etc. For instance, in the context of *mobile* code, as the source code is not available, decompilation facilitates the reuse of the existing analysis and model checking tools. In general, high-level intermediate representations allow abstracting away the particular language features and developing the tools on simpler representations. In particular, JBC is decompiled to a rule-based representation in [2], to clause-based programs in [35], to a three-address code representation in Soot [43] and to the typed procedural language BoogiePL in [13]. Also, the analysis of Java programs is formalized and performed using Datalog in [44] and in [20] PIC assembly is transformed into logic programs. This shows that the rule-based representations used in declarative programming in general – and in Prolog in particular – provide a convenient formalism to define such intermediate representations. For instance, as it can be seen in [2,35,20], the operand stack used in a bytecode language can be represented by means of explicit logic variables and its unstructured control flow can be transformed into recursion.

All above cited approaches (except [20]) develop ad hoc, or dedicated, decompilers to carry out the particular decompilations. An appealing alternative to the development of dedicated decompilers is the so-called *interpretive* decompilation by *partial evaluation* (PE for short) [23]. PE is an automatic program transformation technique which specializes programs w.r.t. part of their input data. Interpretive compilation was proposed in Futamura's seminal work [14], whereby compilation of a program P written in a (*source*) programming language L_S into another (*target*) programming language L_T is achieved by specializing an interpreter for L_S written in L_T w.r.t. P. The advantages of interpretive (de-)compilation w.r.t. dedicated (de-)compilers are well known and discussed in the PE literature (see, e.g., [5]). Very briefly, they include:

1. *Flexibility*: it is easier to modify the interpreter in order to tune the decompilation (e.g., observe new properties of interest). As an interesting example, in [5], a Java bytecode interpreter is instrumented with an additional argument which computes the *trace* of bytecode instructions in order to collect the computation history. A program decompiled by using this interpreter contains an additional argument with the execution trace at

* Corresponding author.
E-mail addresses: mzamalloa@fdi.ucm.es, mzamalloa@clip.dia.fi.upm.es (M. Gómez-Zamalloa), elvira@fdi.ucm.es (E. Albert), german@fi.upm.es (G. Puebla).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.04.010

the level of Java bytecode. This trace will allow to observe a good number of interesting properties about the program, e.g., run-time error freeness can be ensured when the trace does not contain instructions which issue any kind of run-time error.
2. *Easier to trust*: it is more difficult to prove that ad hoc decompilers preserve the program semantics. For example, the formal specification chosen for defining our bytecode interpreter is Bicolano [40], which is written with the Coq Proof Assistant [7]. This allows checking that the specification is consistent and also proving properties on the behavior of some programs.
3. *Easier to maintain*: new changes in the language semantics can be easily reflected in the interpreter. This will become apparent later when we see that defining a bytecode interpreter in Prolog is a rather easy task and, hence, also maintaining it.

The challenge now is in defining a practical, scalable scheme to interpretive decompilation which achieves quality decompiled programs and, provided this is feasible, we will be able to take advantage of the above-mentioned features.

1.1. Summary of contributions

There have been several proofs-of-concept of interpretive (de-)compilation (e.g., [5,20,29]), but there remain interesting open issues when it comes to assess its power and/or limitations to decompile a real language:

(a) *does the approach scale?*
(b) *do decompiled programs preserve the structure of the original ones?*
(c) *is the "quality" of decompiled programs comparable to that obtained by dedicated decompilers?*

This article answers these questions positively by proposing a modular decompilation scheme which can be steered to control the structure of decompiled code and ensure quality decompilations which preserve the original program's structure. Our main contributions are summarized as follows:

1. We present the problems of *non-modular* decompilation and identify the components needed to enable a modular scheme. This includes how to write an interpreter and how to control an *online* partial evaluator in order to preserve the structure of the original program w.r.t. method invocations.
2. We present a modular decompilation scheme which is correct and complete for the proposed big-step interpreter. The *modular-optimality* of the scheme allows addressing issue *(a)* by avoiding decompiling the same method more than once, and *(b)* by ensuring that the structure of the original program can be preserved.
3. We introduce an interpretive decompilation scheme which answers issue *(c)* by producing decompiled programs whose *quality* is similar to that of dedicated decompilers. This requires a *block-level* decompilation scheme which avoids code duplication and code re-evaluation.
4. We report on experimental results on an set of realistic JBC programs which demonstrate the scalability and the efficiency of our proposal.

For the sake of concreteness, our interpretive decompilation scheme is formalized in the context of PE of logic programs but the ideas we propose for enabling the practicality of the approach are also of interest for the interpretive (de-)compilation of any pair of source and target languages.

1.2. Outline of the article

The article is organized as follows. The next section recalls some preliminary definitions and presents the background on PE of logic programs. We recall the correctness issues that a partial evaluator must guarantee. We also sketch the differences between online and offline partial evaluators. Section 3 briefly describes the interpretive approach to (de-)compilation. We present the first Futamura projection in generic terms and then instantiate it to the particular decompilation we want to carry out: decompile JBC to Prolog. Section 4 presents the subset of JBC we consider to define our decompilation scheme. It also describes non-modular decompilation (originally presented in [5]) and explains its limitations for the decompilation of real applications. These limitations are not tied to the decompilation of bytecode. They also occur in any application of interpretive decompilation.

Our first contribution is a modular decompilation scheme which is introduced in Section 5. We start by presenting a big-step interpreter and explain why it is necessary to enable a modular decompilation scheme. Then, we define the annotations that must be generated to obtain such modular decompilation. An important property of the resulting method is that it is ensured that each method is decompiled once.

Our second important contribution is the refinement of the modular decompilation scheme in Section 6 to ensure the scalability of our approach. This requires, among other things, that the decompiler does not emit code more than once for each bytecode instruction. This leads to what we call *block-optimality* in decompilation.

In Section 7 we extend the subset of JBC considered in previous sections in order to support a realistic language with object-oriented features. We show how our scheme can be easily adapted to handle the new features: the decompilation of the heap and associated instructions, the representation of classes by means of Prolog modules and virtual invocations by module-qualified calls. Our experimental results are reported in Section 8, where both the scalability and efficiency of our approach are assessed using the JOlden suite of benchmarks [22]. Finally, Section 9 reviews related work and Section 10 concludes.

2. Background on partial evaluation of logic programs

This section presents some preliminary notions and the background on PE of logic programs (often called *partial deduction*) required to formalize our decompilation scheme. We assume some basic knowledge on the terminology of logic programming and refer to [34] for details.

2.1. Logic programming

Very briefly, an *atom* (or call) A is a syntactic construction of the form $p(t_1,\ldots,t_n)$, with $n \geqslant 0$, where p/n is a predicate signature and $t_1,\ldots,t_n$ are terms. A *clause* is of the form $H:-B_1,\ldots,B_m$, with $m \geqslant 0$, where its head H is an atom and its body $B_1,\ldots,B_m$ is a conjunction of m atoms. Note that in this context commas denote conjunctions. When $m=0$ the clause is called a *fact* and is written "H". A *program* is a finite set of clauses. A *goal* is a conjunction of atoms. We denote by $\{X_1 \mapsto t_1,\ldots,X_n \mapsto t_n\}$ the *substitution* σ with $\sigma(X_i)=t_i$ for $i=1,\ldots,n$ (with $X_i \neq X_j$ if $i \neq j$), and $\sigma(X)=X$ for all other variables X. Given an atom A, $\theta(A)$ denotes the application of substitution θ to A. Given two substitutions θ_1 and θ_2, we denote by $\theta_1\theta_2$ their composition. The identity substitution is denoted by *id*. An atom A' is an *instance* of A if there is a substitution σ with $A'=\sigma(A)$.

The operational semantics of logic programs is based on derivations.

Definition 1 (*derivation step*). Let G be $A_1,\ldots,A_R,\ldots,A_k$ and $C = H : -B_1,\ldots,B_m$ be a renamed apart clause in P (i.e., it has no common variables with G). Let A_R be the selected atom for its evaluation. Then G' is *derived* from G if the following conditions hold:

$\theta = mgu(A_R, H)$
G' is the goal $\theta(A_1,\ldots,A_{R-1},B_1,\ldots,B_m,A_{R+1},\ldots,A_k)$

As customary, given a program P and a goal G, a *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P (i.e. C_i has no common variables with any G_j nor C_j with $j < i$), and a sequence of *computed answer substitutions* $\theta_1, \theta_2, \ldots$ (or *most-general unifiers*, mgus for short) such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. Finally, we say that the SLD derivation is composed of the *subsequent* goals $G_0, G_1, G_2, \ldots$.

A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees*.

A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2\ldots\theta_n$ is called the computed answer for goal G. Such a derivation is called *failing* if it is not possible to perform a derivation step with G_n.

Executing a program P for a call A consists in building a *SLD tree* for $P \cup \{A\}$ and then extracting the *computed answers* from every non-failing branch of the tree.

2.2. Partial deduction

Partial evaluation in logic programming (see, e.g. [16]) builds upon the SLD trees mentioned above. We now introduce a generic function *PE*, which is parametric w.r.t. the *unfolding rule*, unfold, and the *abstraction operator*, abstract, and captures the essence of most algorithms for PE of logic programs:

```
1: function PE P, 𝒜, S
2:    S_0 := S;  i := 0;
3:    repeat
4:       L^pe := unfold(S_i, P, 𝒜);
5:       S_{i+1} := abstract(S_i, L^pe, 𝒜);
6:       i := i + 1;
7:    Until S_i = S_{i-1}      % (modulo renaming)
8:    return codegen(L^pe, unfold);
```

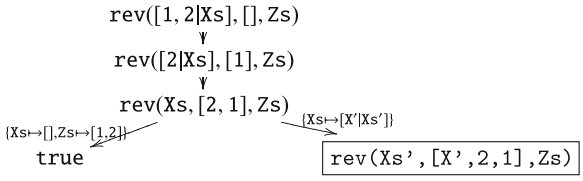
Function PE differs from standard ones in the use of the set of annotations $\mathscr{A}$, whose role is described below. PE starts from a program P, a (possibly empty) set of annotations $\mathscr{A}$ and an initial set of calls S. At each iteration, the so-called *local control* is performed by the unfolding rule unfold (Line 4), which takes the current set of atoms S_i, the program and the annotations and constructs a *partial* SLD tree for each call in S_i. Trees are partial in the sense that, in order to guarantee termination of the unfolding process, it must be possible to choose *not* to further unfold a goal, and rather allow leaves in the tree with a non-empty, possibly non-failing, goal (these goals appear in the next examples within a frame). The atoms corresponding to such goals are returned by unfold and stored in L^{pe} (Line 4). Then, in the *global control*, which is performed by the abstraction operator abstract, when some calls in the leaves of the trees are not properly *covered*, the operator abstract adds them to the new set of atoms to be partially evaluated in a proper "generalized" form such that termination is ensured (i.e., the condition $S_i = S_{i-1}$ is reached).

Let us consider the PE of the following program to reverse a list using an accumulator (predicate `rev/3`) w.r.t. the initial set $S = \{\texttt{rev([1,2|Xs],[],Zs)}\}$ and $\mathscr{A} = \emptyset$:

```
rev([],L,L).
rev([X|Xs],Ys,Zs) :- rev(Xs,[X|Ys],Zs).
```

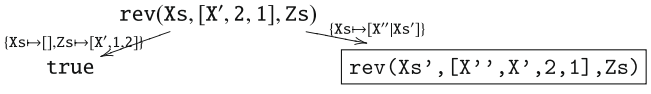
Prolog lists use the notation $[X|L]$ to denote the list with X as head and L as continuation and [] to denote the empty list. The particular unfold operator determines which atom to select from each goal and when to stop unfolding. Let us consider an unfolding rule based on the *homeomorphic embedding* [28] relation, a *well-quasi order* used in state-of-the-art specialization tools. Intuitively, the homeomorphic embedding is a structural ordering under which an expression e_1 is greater than (i.e., it *embeds*), another expression e_2 if e_2 can be obtained from e_1 by deleting some parts, e.g., $\underline{s}(s(\underline{U}+W)\times(\underline{U}+s(\underline{V})))$ embeds $s(U\times(U+V))$.

Such unfolding rule always selects the leftmost atom and stops the derivation when the selected call *embeds* a previous call and thus threatens termination. We start by constructing the following SLD tree:

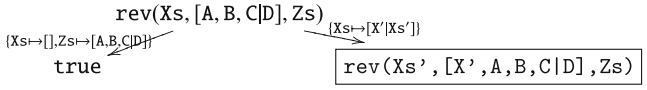


It can be observed that the call in the frame $\texttt{rev(Xs}', [\texttt{X}', 2, 1], \texttt{Zs})$ embeds the previous call $\texttt{rev(Xs},[2,1],\texttt{Zs})$, hence the derivation is stopped. Such call is said to be transferred to the global control in the sense that it is returned by unfold as an element of L^{pe} and hence it is passed away as an argument to abstract.

The partial evaluator may have to build several SLD trees to ensure that all calls left in the leaves (L^{pe} in Line 4) are "covered" by the root of some tree. This is known as the *closedness* condition of PE [33]. For example, after having built the first SLD tree for the call $\texttt{rev([1,2|Xs],[],Zs)}$, the call $\texttt{rev(Xs}', [\texttt{X}', 2, 1], \texttt{Zs})$ is not covered by $\texttt{rev([1,2|Xs],[],Zs)}$ because it is not an instance of it. At this point the abstract operator adds the framed call to the new set of atoms to be partially evaluated. At the next iteration, the following SLD tree is built for such call:



Thus, basically, the algorithm iteratively (Lines 3–7) constructs partial SLD trees until all their leaves are covered by the root nodes. An essential point of the operator abstract is that it has to perform "generalizations" on the calls that have to be partially evaluated in order to avoid computing partial SLD trees for an infinite number of calls. The homeomorphic embedding can be again used here to ensure termination and detect which calls have to be generalized. A classical way of performing generalizations is to use the *most-specific generalizer* operator (*msg* for short) in the following way. Suppose that a call A is to be added to the set S_k, and that there is a call B in S_k s.t. A embeds B, then the *msg* of A and B is added to the set S_{k+1} (and usually B is removed). In the example, the framed call $\texttt{rev(Xs},[\texttt{X}', \texttt{X}'', 2, 1], \texttt{Zs})$ embeds $\texttt{rev(Xs},[\texttt{X}, 2, 1], \texttt{Zs})$ (also framed), therefore both are generalized using the *msg* resulting in $\texttt{rev(Xs},[\texttt{A},\texttt{B},\texttt{C}|\texttt{D}], \texttt{Zs})$. The generalized call is added to the set S_{i+1} and $\texttt{rev(Xs},[\texttt{X}, 2, 1], \texttt{Zs})$ removed. At the next iteration, the following SLD tree is built for the generalized atom:



Without such generalization, the algorithm would keep on adding calls $\texttt{rev}(\texttt{Xs}, [\texttt{X}, \texttt{X}', \texttt{X}'', 2, 1], \texttt{Zs})$, $\texttt{rev}(\texttt{Xs}, [\texttt{X}, \texttt{X}', \texttt{X}'', \texttt{X}''', 2, 1], \texttt{Zs})$,... infinitely.

A partial evaluation of P w.r.t. S is then systematically extracted from the resulting set of calls L^{pe} in the final phase, codegen in Line 8. The notion of *resultant* is used to generate a program rule associated to each root-to-leaf derivation of the SLD trees for the final set of atoms L^{pe}. Given a SLD derivation of $P \cup \{A\}$ with $A \in L^{pe}$ ending in B and θ being the composition of the mgus in the derivation steps, the rule $\theta(A) : -B$ is called the *resultant* of the derivation.

A PE is defined as the set of resultants (clauses) associated to the derivations of the constructed partial SLD trees for all $P \cup L^{pe}$. The resulting program is often referred to as the *specialized program* or *residual program*. In the example, the final set L^{pe} contains the calls $\texttt{rev}([1, 2|\texttt{Xs}], [], \texttt{Zs})$ and $\texttt{rev}(\texttt{Xs}, [\texttt{A}, \texttt{B}, \texttt{C}|\texttt{D}], \texttt{Zs})$ from which the following PE (residual program) is generated:

$\texttt{rev}([1, 2], [], [2, 1]).$
$\texttt{rev}([1, 2, \texttt{A}|\texttt{B}], [], \texttt{C}) : -\texttt{rev\_1}(\texttt{B}, [\texttt{A}, 2, 1], \texttt{C}).$
$\texttt{rev\_1}([], [\texttt{A}, \texttt{B}, \texttt{C}|\texttt{D}], [\texttt{A}, \texttt{B}, \texttt{C}|\texttt{D}]).$
$\texttt{rev\_1}([\texttt{A}|\texttt{B}], [\texttt{C}, \texttt{D}, \texttt{E}|\texttt{F}], \texttt{G}) : -\texttt{rev\_1}(\texttt{B}, [\texttt{A}, \texttt{C}, \texttt{D}, \texttt{E}|\texttt{F}], \texttt{G}).$

The first two resultants are obtained from each derivation (branch) of the first tree above and the last two ones from the last tree above.

It can be also observed that a post-processing of renaming has been performed by codegen as explained below. Such a post-processing use to perform in addition some form of *argument filtering* [32]. This is because automatically generated programs, and in particular those generated by PE, very often contain redundant arguments which do not affect the correctness of the program. Throughout the rest of the paper we will consider a codegen function which is able to remove arguments which are actually not used in any computation but rather just passed around.

2.3. Correctness of partial deduction

Intuitively, the notions of, respectively, *completeness* and *correctness* of PE [16] ensure that the specialized program produces no less, respectively, no more answers than the original program. A sufficient condition to ensure completeness is that the specialized program is *closed* by the resulting set of atoms L^{pe}. As informally explained in Section 2.2, the closedness condition ensures that all calls which may arise during the computation of $P \cup S$ are instances of L^{pe} and hence there is a matching resultant for them (solutions are not lost).

Definition 2 (*closedness*). Let T and S be two sets of atoms. Then, S is T-closed iff each atom in S is an instance of an atom in T. Given a program P and a set of atoms T, we say that $P \cup T$ is S-closed if the set of atoms which occur in the computation of $P \cup T$ are S-closed.

The abstraction operator ensures that the closedness condition is met by means of a proper generalization of calls. For instance, as the set of atoms $\{\texttt{rev}(\texttt{Xs}, [\texttt{X}', \texttt{X}'', 2, 1], \texttt{Zs})\}$ is not closed w.r.t. this set $\{\texttt{rev}(\texttt{Xs}, [\texttt{X}, 2, 1], \texttt{Zs})\}$, the abstraction operator has generalized both terms to the term $\texttt{rev}(\texttt{Xs}, [\texttt{A}, \texttt{B}, \texttt{C}|\texttt{D}], \texttt{Zs})$ which covers both terms.

Let us see an example where the closedness condition does not hold and hence we lose completeness. Consider a program defined by these two clauses:

$\texttt{p}(\texttt{X}) : -\texttt{q}(\texttt{X}).$
$\texttt{q}(\texttt{X}).$

The following partially evaluated program has been obtained by specializing the above-mentioned program w.r.t. the set of atoms $S = \{\texttt{q}(\texttt{a})\}$:

$\texttt{p}(\texttt{X}) : -\texttt{q}(\texttt{X}).$
$\texttt{q}(\texttt{a}).$

The closedness condition w.r.t. the set S does not hold because the atom in the left-hand side of the first rule is not an instance of any atom in S. It can be seen that the partially evaluated program is not complete since the goal $\texttt{p}(\texttt{b})$ succeeds in the original program while it fails in the residual one.

Correctness is achieved when the resulting set L^{pe} is independent, i.e., there are no two calls in L^{pe} which unify.

Definition 3 (*independence*). Let S be a set of atoms. Then, S is *independent* if no pair of atoms in S have a common instance.

Let us see an example where the independence condition does not hold and hence we lose correctness. Consider again the above-mentioned program and the set of atoms $S = \{\texttt{q}(\texttt{X}), \texttt{q}(\texttt{a})\}$ which is not independent. The following program is a partial evaluation w.r.t. the set S:

$\texttt{p}(\texttt{X}) : -\texttt{q}(\texttt{X}).$
$\texttt{q}(\texttt{X}).$
$\texttt{q}(\texttt{a}).$

It can be seen that the residual program produces more answers than the original one. In particular, for the goal $\texttt{q}(\texttt{Y})$ it returns two answers $\{\texttt{Y}\mapsto\texttt{X}\}$ and $\{\texttt{Y}\mapsto\texttt{a}\}$ while the original program generates only the first one.

Independence can be recovered by a post-processing of renaming [16]. In the previous program, the two atoms in S could be renamed as $\texttt{q}_1(\texttt{X})$ and $\texttt{q}_2(\texttt{a})$ and the residual program would contain one clause defining $\texttt{q}_1$ and another one for $\texttt{q}_2$. In addition, renaming has benefits for performance because it reduces the number of rules per predicate. Thus, though the calls in L^{pe} for our running example are independent, we rename the second call for predicate rev to $\texttt{rev\_1}$.

Theorem 1 (correctness). *Let P be a program, L^{pe} be a finite, independent set of atoms and P' be a partial evaluation of L^{pe} in P. For every goal G such that $P' \cup \{G\}$ is L^{pe}-closed, the following conditions hold:*

- *Soundness: $P' \cup \{G\}$ has a successful derivation with an answer θ only if $P \cup \{G\}$ does.*
- *Completeness: $P \cup \{G\}$ has a successful derivation with an answer θ only if $P' \cup \{G\}$ does.*

The above-mentioned theorem is proven in early work on PE of logic programs [33,25].

2.4. Online vs. offline partial deduction

It is well known that both the quality of the specialized programs and the time required for the PE process greatly vary with the control strategies used. Traditionally, two approaches to PE have been considered, *online* and *offline* PE. In online PE, all control decisions are taken on the fly during the specialization phase by keeping track of the specialization history. This is the case of the control rules used in the example of Section 2.2. In the offline approach, all control decisions are taken before the proper specialization phase. These control decisions are based on abstract

descriptions of the data instead of the actual data. The control strategy is usually represented as program annotations which are the sole decision criteria for control of the partial evaluator. For instance, in the local control, an annotation can explicitly indicate that an atom should not be unfolded. In the global control, annotations typically specify for each call which arguments have to be generalized away (i.e. replaced by variables). Such annotations are in some partial evaluators automatically generated by a *binding-time analysis* and in other partial evaluators they are manually provided by the user, either in part or in full.

Under this classification, the PE algorithm we propose can be considered a hybrid approach since the $\mathscr{A}$ annotations provide information to the control operators, as in offline PE, and the algorithm includes control rules based on the actual specialization history, as in online PE. The advantages of the offline approach are that, once all control annotations are available, PE is quite simple and efficient. On the other hand, online PE, though less efficient, has a strictly more powerful control strategy since control decisions are based on actual data instead of abstract descriptions of data. Therefore, though all offline PEs can be replicated using online techniques, many online PEs cannot be reproduced using offline techniques.

In this work we are interested in investigating how far we can go with the more powerful but less efficient online PE approach. The motivation for this is that this way we may obtain decompilations of higher quality than those achievable using offline PE. Thus, our challenges are both in terms of quality of the decompiled programs and in terms of efficiency of the decompilation process. As we will see later in the article many of the lessons learned in this work are of interest both to the online and offline approaches to the PE of interpreters.

3. The interpretive approach to compilation

The development of PE, program specialization and related techniques [14,23,15] has led to an alternative approach to compilation (known as the first Futamura projection) based on specializing an interpreter with respect to a fixed object program. Let us explain intuitively the interpretive approach. We denote by $\texttt{mix}$ a generic partial evaluator, by $\texttt{p}$ a program and by $\texttt{inl}$ (respectively, $\texttt{in2}$) the static (respectively, dynamic) input data. Given a program $\texttt{P}$, we write $\texttt{P}_\texttt{L}$ to denote that $\texttt{P}$ is written in language $\texttt{L}$. When the program is a meta-program, we write as a super-index the language the meta-program manipulates. For instance, $\texttt{mix}^{\texttt{L}}_{\texttt{S}}$ denotes a partial evaluator, written in $\texttt{S}$, which manipulates programs written in $\texttt{L}$. We omit the languages (both sub- and super-indexes) when they are not relevant. Finally, we use the notation $[[\texttt{P}]][\texttt{d}]$ to denote the execution of $\texttt{P}$ with input data $\texttt{d}$. A partial evaluator can be defined as a program which behaves as follows:

$$\texttt{p\_inl} = [[\texttt{mix}]][\texttt{p}, \texttt{inl}]$$
$$\texttt{output} = [[\texttt{p\_inl}]][\texttt{in2}] = [[\texttt{p}]][\texttt{inl}, \texttt{in2}]$$

Essentially, the execution of $\texttt{mix}$ for $\texttt{p}$ and $\texttt{inl}$ returns a specialized program $\texttt{p\_inl}$ whose execution for the dynamic data $\texttt{in2}$ must be the same as executing the original program $\texttt{p}$ w.r.t. all dynamic plus static data $[\texttt{inl}, \texttt{in2}]$. This implies the following:

$$[[\texttt{p}]][\texttt{inl}, \texttt{in2}] = [[[[\texttt{mix}]][\texttt{p}, \texttt{inl}]]][\texttt{in2}] \quad (1)$$

This means that the result of PE is a program which is semantically equivalent w.r.t. the original for the static data.

We now define by means of equations the behavior of an interpreter $\texttt{int}^{\texttt{S}}_{\texttt{L}}$, which interprets programs written in $\texttt{S}$, and is written in (a possibly different) language $\texttt{L}$:

$$\texttt{output} = [[\texttt{source}_\texttt{S}]][\texttt{d}] = [[\texttt{int}^{\texttt{S}}_{\texttt{L}}]][\texttt{source}_\texttt{S}, \texttt{d}] \quad (2)$$

This captures the idea that executing a source program $\texttt{source}$ for some input data $\texttt{d}$ in the interpreter gives the same output as the execution of the program yields. Similarly, we can define a compiler $\texttt{comp}^{\texttt{S}\to\texttt{T}}_{\texttt{L}}$ from $\texttt{S}$ to $\texttt{T}$ written in (a possibly different) language $\texttt{L}$ as follows:

$$[[\texttt{source}_\texttt{S}]][\texttt{d}] = [[[[\texttt{comp}^{\texttt{S}\to\texttt{T}}_{\texttt{L}}]][\texttt{source}_\texttt{S}]]][\texttt{d}] \quad (3)$$
$$[[\texttt{comp}^{\texttt{S}\to\texttt{T}}_{\texttt{L}}]][\texttt{source}_\texttt{S}] = \texttt{target}_\texttt{T}$$

Consider now a partial evaluator $\texttt{mix}^{\texttt{T}}_{\texttt{L}}$ (written in $\texttt{L}$) for programs written in $\texttt{T}$, and an interpreter $\texttt{int}^{\texttt{S}}_{\texttt{T}}$ (written in $\texttt{T}$) for programs written in $\texttt{S}$. Now, the compilation of a program $\texttt{source}_\texttt{S}$ to a program $\texttt{target}_\texttt{T}$ by using a partial evaluator $\texttt{mix}^{\texttt{T}}_{\texttt{L}}$ can be performed as follows:

$$\texttt{target}_\texttt{T} = [[\texttt{mix}^{\texttt{T}}_{\texttt{L}}]][\texttt{int}^{\texttt{S}}_{\texttt{T}}, \texttt{source}_\texttt{S}]$$

which is justified by combining Eqs. (1) and (2) in this way:

$$[[\texttt{p}_\texttt{S}]]\texttt{d} = [[\texttt{int}^{\texttt{S}}_{\texttt{T}}]][\texttt{p}, \texttt{d}] = [[[[\texttt{mix}^{\texttt{T}}_{\texttt{L}}]][\texttt{int}^{\texttt{S}}_{\texttt{T}}, \texttt{p}]]]\texttt{d}$$

Now, by comparing the above-mentioned equation with Eq. (3), it can be observed the essence of compilation by means of PE of interpreters: we obtain the compilation of the program $\texttt{p}$ written in $\texttt{S}$ into another language $\texttt{T}$. The application of this interpretative approach to compilation within our framework consists in specializing a bytecode ($\texttt{BC}$) interpreter $\texttt{int}^{\texttt{BC}}_{\texttt{LP}}$ written in logic programming $\texttt{LP}$ where the static data are the actual bytecode program $\texttt{p}_\texttt{BC}$ to be decompiled:

$$[[\texttt{p}_\texttt{BC}]]\texttt{d} = [[\texttt{int}^{\texttt{BC}}_{\texttt{LP}}]][\texttt{p}_\texttt{BC}, \texttt{d}] = [[[[\texttt{mix}^{\texttt{LP}}]][\texttt{int}^{\texttt{BC}}_{\texttt{LP}}, \texttt{p}_\texttt{BC}]]]\texttt{d} = [[\texttt{p}_\texttt{LP}]]\texttt{d}$$

It can be observed that the result is a decompiled program $\texttt{p}_\texttt{LP}$ in $\texttt{LP}$ which, given the actual input data $\texttt{d}$ produces the same result as the original program $\texttt{p}_\texttt{BC}$.

4. Non-modular interpretive decompilation

This section describes the state of the art in interpretive decompilation of low-level languages to Prolog, including recent work in [20,4,18,5]. We do so by formulating non-modular decompilation in a generic way and by identifying its limitations.

4.1. The bytecode language

The bytecode language we consider, denoted as $\mathscr{L}_{bc}$, is a simple imperative bytecode language in the spirit of Java bytecode. To simplify the presentation, it does not include advanced features of Java bytecode such as exceptions, arrays, object-oriented features, and access control (e.g. public, protected, and private) and it manipulates only integer numbers. The extensions to consider such advanced features will be discussed later in Sections 7 and 8. As in Java bytecode, $\mathscr{L}_{bc}$ uses an operand stack to perform intermediate computations and an array of variables to store the formal parameters of the method and the actual method variables. Also, the global *heap* is not yet considered. Support for object-oriented features will be provided later. Finally, $\mathscr{L}_{bc}$, has an unstructured control flow, i.e., there are no explicit block markers, hence it includes explicit conditional and unconditional $\texttt{goto}$ instructions.

A bytecode program P_{bc} consists of a set of methods which are the basic (de-)compilation units of $\mathscr{L}_{bc}$. The code of a method m, denoted $code(m)$, consists of a sequence of indexed bytecode instructions $\langle pc_0 : bc_0, \ldots, pc_{n_m} : bc_{n_m}\rangle$ with $pc_0, \ldots, pc_{n_m}$ being consecutive natural numbers. The $\mathscr{L}_{bc}$ instruction set is:

$$\begin{aligned} \mathit{Inst}_{\mathscr{L}_{bc}} ::= {} & \texttt{push(x)}|\texttt{load(v)}|\texttt{store(v)}|\texttt{add}|\texttt{sub}|\texttt{mul}|\texttt{div}|\texttt{rem}| \\ & \texttt{neg}|\texttt{if}\diamondsuit\ \texttt{(pc)}|\texttt{if0}\ \diamondsuit\ \texttt{(pc)}|\texttt{goto(pc)}|\texttt{return}|\texttt{invoke(mn)} \end{aligned}$$

where $\diamondsuit$ is a comparison operator (eq, le, gt, etc.), v a local variable, x an integer, pc an instruction index and mn a method name. Instruc-

tions `push`, `load` and `store` transfer values or constants from the local variables to the stack (and vice versa); `add`, `sub`, `mul`, `div`, `rem` and `neg` perform the usual arithmetic operations, `rem` being the division remainder and `neg` being the arithmetic negation; `if` and `if0` are conditional branching instructions (with the special case of comparisons with 0); `goto` is an unconditional branching; `return` marks the end of methods and `invoke` invokes a method. For simplicity, all methods are supposed to return an integer value. A method m is uniquely determined by its name. We write $calls(m)$ to denote the set of all method names invoked within the code of m. We use $defs(P_{bc})$ to denote the set of *internal* method names defined in P_{bc}. The remaining methods are *external*. We say that P_{bc} is *self-contained* if $\forall m \in P_{bc}, calls(m) \subseteq defs(P_{bc})$, i.e., P_{bc} does not include calls to external methods.

Though very simple, $\mathscr{L}_{bc}$ will be enough for our purposes when presenting the main ideas of the different decompilation schemes. Nevertheless it will be gradually extended as needed when we present more advanced features until the point of covering the full Java bytecode language in the experimental evaluation in Section 8.

4.2. Non-modular, online decompilation

We rely on the interpretive approach to compilation by PE described in Section 3. As it has been already explained, the decompilation of a $\mathscr{L}_{bc}$-bytecode program P_{bc} to LP (for short LP-decompilation) might be obtained by specializing (with an LP partial evaluator) a $\mathscr{L}_{bc}$-interpreter written in LP w.r.t. P_{bc}. In Fig. 1 we show a fragment of a (small-step) $\mathscr{L}_{bc}$ interpreter implemented in Prolog, named $Int_{\mathscr{L}_{bc}}$. We assume that the code for every method in the bytecode program P_{bc} is represented as a set of facts `bytecode/3` such that, for every pair $pc_i : bc_i$ in the code for method m, we have a fact `bytecode(`m`,`pc_i`,`bc_i`)`. The state carried around by the interpreter is of the form `st(Fr,FrStack)` where `Fr` represents the current frame (environment) and `FrStack`, the stack of frames (call stack) implemented as a list. Frames are of the form `fr(M,PC,OStack,LocalV)`, where `M` represents the current method, `PC`, the program counter; `OStack`, the operand stack and `LocalV`, the list of local variables. Predicate `main/3`, given the method to be interpreted `Method` and a concrete input (method arguments) `InArgs`, first builds the initial state by means of predicate `build_s0/3` and then calls predicate `execute/2`. In turn, `execute/2` calls predicate `step/3`, which produces `S'`, the state after executing the bytecode, and then calls predicate `execute/2` recursively with `S'` until we reach a `return` instruction with the empty stack. For brevity, we only show the definition of `step/3` for a selected set of instructions and omit the code of some auxiliary predicates. Namely `build_s0/3`, which was explained below, `next/3`, which produces the next program counter given the current one, `nth/3` and `replace_nth/4`, which, respectively, get and set the ith element of a list, and `split_OS/4`, which splits the current operand stack between the parameters list to be used in the called method and the rest. By using this interpreter, we define a *non-modular* decompilation scheme in terms of the generic function *PE* as follows:

Definition 4 ($DECOMP_{\mathscr{L}_{bc}}$). Given a self-contained $\mathscr{L}_{bc}$-bytecode program P_{bc}, the (non-modular) LP-decompilation of P_{bc} can be obtained as:

$$DECOMP_{\mathscr{L}_{bc}}(P_{bc}) = PE(Int_{\mathscr{L}_{bc}} \cup P_{bc}, \emptyset, S)$$

where S is the set of calls $\{main(m,\_,\_) | m \in defs(P_{bc})\}$.

Observe in the above-mentioned definition that the set of annotations is empty. Following the PE terminology, the above-mentioned definition corresponds to *online* PE as we have explained in Section 2.4.

Recent work in interpretive, online decompilation has focused on ensuring that the layer of interpretation is completely removed from decompiled programs, i.e., *effective* decompilations are obtained. This requires the use of the following advanced control techniques. Type-based homeomorphic embedding ($\trianglelefteq_T$) [4] has been used at both the local and global control to decide when to stop derivations and when to generalize calls so that effectiveness of the decompilation can be obtained in the presence of integers without compromising termination. The unfolding operator must also be able to accurately handle built-in predicates and to safely perform non-leftmost unfolding steps as in [6]. The operator abstract must incorporate a polyvariance control mechanism [18] which avoids performing useless specializations that can introduce superfluous decompiled code and thus degrade the decompilation effectiveness. Our starting point is thus a state-of-the-art online partial evaluator based on an unfolding operator $\text{unfold}_{\trianglelefteq_T}$ and abstraction operator $\text{abstract}_{\trianglelefteq_T}$ which incorporate such advanced techniques and is able to remove the layer of interpretation. Such advanced partial evaluator is used in the following for both running examples and experiments.

```
main(Method,InArgs,Top) :-
   build_s0(Method,InArgs,S0),
   execute(S0,Sf),
   Sf = st(fr(_,_,[Top|_],_),_)).

   S = st(fr(M,PC,[_Top|_],_),[]),
   bytecode(M,PC,return,_).
execute(S,Sf) :-
   S = st(fr(M,PC,_,_),_),
   bytecode(M,PC,Inst,_),
   step(Inst,S,S'),
   execute(S',Sf).

step(push(X),S,S') :-
   S = st(fr(M,PC,OS,L),FrS),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[X|OS],L),FrS).
step(add,S,S') :-
   S = st(fr(M,PC,[X,Y|OS],L),FrS),
   next(M,PC,PC'), Z is X + Y,
   S' = st(fr(M,PC',[Z|OS],L),FrS).

step(goto(PC),S,S') :-
   S = st(fr(M,_,OS,LV),FrS),
   S' = st(fr(M,PC,OS,LV),FrS).
step(load(I),S,S') :-
   S = st(fr(M,PC,OS,L),FrS),
   next(M,PC,PC'), nth(L,I,V),
   S' = st(fr(M,PC',[V|OS],L),FrS).
step(store(I),S,S') :-
   S = st(fr(M,PC,[V|OS],L),FrS),
   next(M,PC,PC'), replace_nth(L,I,V,L'),
   S' = st(fr(M,PC',OS,L'),FrS).
...
step(invoke(M'),S,S') :-
   S = st(fr(M,PC,OS,LV),FrS),
   split_OS(M',OS,Args,OS''),
   build_s0(M',Args,st(fr(M',PC',OS',LV'),_)),
   S' = st(fr(M',PC',OS',LV'),
           [fr(M,PC,OS'',LV)|FrS]).
step(return,S,S') :-
   S = st(fr(_,_,[RV|_],_),[fr(M,PC,OS,LV)|FrS]),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[RV|OS],LV),FrS).
```

Fig. 1. Fragment of (small-step) $\mathscr{L}_{bc}$ interpreter.

4.3. Limitations

This section illustrates by means of the bytecode example in Fig. 2 that non-modular decompilation does not ensure a satisfactory handling of issues *(a)* and *(b)* in Section 1. In the examples, we often depict the Java source code for clarity, but the decompiler works directly on the bytecode. The program consists of a set of methods that carry out arithmetic operations. Method `gcd` computes the greatest-common divisor, `abs` the absolute value, `lcm` the least-common multiple and `fact` the factorial recursively. The LP-decompilationobtained by applying Definition 4 is shown in Fig. 3. The partial evaluator performs a post-processing of renaming and argument filtering [16] for all calls except for calls to the `main` predicate (as they represent calls to methods whose name we want to preserve). We identify below four limitations, which we identify as (**L1**)...(**L4**), of non-modular decompilation. It is important to note that such limitations, and the way to avoid them which we propose in Section 5 below, are also relevant to the case of offline PE.

(**L1**) Calls to methods are *inlined* within their calling contexts and, as a consequence, the structure of the original code is lost. For example, method invocations from `lcm` to `gcd` (index 2) and from `gcd` to `abs` (index 12) do not appear in the decompiled code. As a result, the last two rules in the decompilation for `lcm`, `execute_1`, correspond to the `while` loop of `gcd`. This happens because calls to methods are dealt with in a *small-step* fashion within the interpreter, i.e., the code of invoked methods is unfolded as if it was inlined inside the "caller" method.

(**L2**) As a consequence, decompilation becomes very inefficient. For example, if n calls to the same method appear within a code, such method will be decompiled n times. Even worse, if there is a method invocation inside a loop, its code will be evaluated twice in the best case, as we have to perform the corresponding generalizations in the global control before reaching a fixpoint, as in the example of Section 2.2. This is worse in the case of nested loops.

(**L3**) The non-modular approach does not work incrementally, in the sense that it does not support *separate* decompilation of methods but rather has to (re)decompile all method calls. Thus, decompiling a real language becomes unfeasible, as one needs to consider system libraries, whose code might not be available. Limitation L2 together with L3 answers issue *(a)* negatively.

(**L4**) The decompiled code contains basically the whole interpreter when there are recursive methods. This is why the decompiled program in Fig. 3 does not contain the code corresponding to the recursive `fact` method. The problem with recursion is as follows. Assume we want to decompile method $m1$ whose code is $\langle pc_0 : bc_0, \ldots, pc_j : invoke(m1), \ldots, pc_n : return\rangle$. There is an initial decompilation for $A_k = \mathtt{execute}(\mathtt{st}(\mathtt{fr}(\mathtt{ml}, \mathtt{pc_j}, \mathtt{os}, \mathtt{lv}), []), \mathtt{S_f})$ in which the call stack is empty. During its decompilation, a call of the form $A_l = \mathtt{execute}(\mathtt{st}(\mathtt{fr}(\mathtt{ml}, \mathtt{pc_j}, \mathtt{os}', \mathtt{lv}'), [\mathtt{fr}(\mathtt{ml}, \mathtt{pc_j}, \mathtt{os}, \mathtt{lv})]), \mathtt{S_f})$ with the call stack containing the previous frame appears when we arrive to the recursive call. At this point, the derivation must be stopped as $A_k \trianglelefteq_T A_l$. In order to ensure termination, global control generalizes the above-mentioned calls into $\mathtt{execute}(\mathtt{st}(\mathtt{fr}(\mathtt{ml}, \mathtt{pc_j}, \_, \_), \_), \_), S_f)$, where $\_$ denotes a fresh variable and thus the call stack has become unknown. As a consequence, after evaluating the *return* statement, the continuation obtained from the call stack is unknown and we produce the call $\mathtt{execute}(\mathtt{st}(\mathtt{fr}(\_, \_, \_, \_), \_), \mathtt{S_f})$ to be decompiled. Here, the fact that the method and the program counter are unknown

```
int gcd(int x,int y){
   int res;
   while (y != 0){
      res = x%y; x = y;
      y = res;}
   return abs(x);}

int abs(int x){
   if (x < 0) return -x;
   else return x; }
```

```
int lcm(int x,int y){
   int gcd = gcd(x,y);
   if (gcd == 0) return 0;
   else return x*y/gcd;}

int fact(int x){
   if (x == 0)
      return 1;
   else
      return x*fact(x-1);}
```

```
Method gcd
 0:load(1)
 1:if0eq(11)
 2:load(0)
 3:load(1)
 4:rem
 5:store(2)
 6:load(1)
 7:store(0)
 8:load(2)
 9:store(1)
 10:goto 0
 11:load(0)
 12:invoke(abs)
 13:return
```

```
Method abs
 0:load(0)
 1:if0ge(5)
 2:load(0)
 3:neg
 4:return
 5:load(0)
 6:return
```

```
Method lcm
 0:load(0)
 1:load(1)
 2:invoke(gcd)
 3:store(2)
 4:load(2)
 5:if0ne 8
 6:push(0)
 7:return
 8:load(0)
 9:load(1)
 10:mul
 11:load(2)
 12:div
 13:return
```

```
Method fact
 0:load(0)
 1:if0ne(4)
 2:push(1)
 3:return
 4:load(0)
 5:load(0)
 6:push(1)
 7:sub
 8:invoke(fact)
 9:mul
 10:return
```

Fig. 2. Source code and $\mathscr{L}_{bc}$-bytecode for working example.

```
main(lcm,[B,0],A) :- B>0, C is B*0,
                     A is C//B.
main(lcm,[0,0],0).
main(lcm,[B,0],A) :- B<0, D is B*0,
                     C is -B, A is D//C.
main(lcm,[B,C],A) :- C\=0, D is B rem C,
                     execute_1(C,D,B,C,A).

execute_1(A,0,B,C,D) :- A>0, E is B*C, D is E//A.
execute_1(0,0,_,_,0).
execute_1(A,0,B,C,D) :- A<0, E is -A,
                        F is B*C, D is F//E.
execute_1(A,B,C,D,I) :- B\=0, K is A rem B,
                        execute_1(B,K,C,D,I).
```

```
main(gcd,[A,0],A) :- A>=0.
main(gcd,[B,0],A) :- B<0, A is -B.
main(gcd,[B,C],A) :- C\=0,
                     D is B rem C,
                     execute_2(C,D,A).

execute_2(A,0,A) :- A>=0.
execute_2(A,0,C) :- A<0, C is -A.
execute_2(A,B,G) :- B\=0,
                    I is A rem B,
                    execute_2(B,I,G).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.
```

Fig. 3. Decompiled (non-modular) code for working example.

prevents us from any chance of removing the interpretation layer, i.e., the decompiled code will potentially contain the whole interpreter. This indeed happens during the decompilation of `fact`. Partial solutions to the recursion problem exist and will be discussed later. Limitations L1 and L4 answer issue (*b*) negatively.

5. A modular decompilation scheme

By *modular* decompilation, we refer to a decompilation technique whose decompilation unit is the method, i.e., we decompile a method at a time. We show that this approach overcomes the four limitations of non-modular decompilation described in Section 4.3 and answers issues (*a*) and (*b*) positively. In essence, we need to: (i) Give a compositional treatment to method invocations. We show that this can be achieved by considering a *big-step* interpreter. (ii) Provide a mechanism to residualize calls in the decompiled program (i.e. do not unfold them and add them without modifications to the residual code). We automatically generate program annotations for this purpose. (iii) Study the conditions which ensure that *separate* decompilation of methods is sound.

5.1. Big-step semantics interpreter to enable modularity

Traditionally, two different approaches have been considered to define language semantics, *big-step* (or *natural*) semantics and *small-step* (or *structural operational*) semantics (see, e.g., [26]). Essentially, in big-step semantics, transitions relate the initial and final states for each statement, while in small-step semantics transitions define the *next* step of the execution for each statement. In the context of bytecode interpreters, it turns out that most of the statements execute in a single step, hence making both approaches equivalent for such statements. This is the case for our $\mathscr{L}_{bc}$-bytecode interpreter for all statements except for *call*. The transition for *call* in small-step defines the next step of the computation, i.e., the current frame is pushed on the call stack and a new environment is initialized for the execution of the invoked method. Note that, after performing this step, we do not distinguish anymore between the code of the caller method and that of the callee. This prevents us from having modularity in decompilation.

In the context of interpretive (de-)compilation of imperative languages, small-step interpreters are commonly used (see, e.g. [39,20,5]). We argue that the use of a big-step interpreter is a necessity to enable modular decompilation which scales to realistic languages. In Fig. 4, we depict the relevant part of the big-step interpreter for $\mathscr{L}_{bc}$-bytecode, named $Int^{bs}_{\mathscr{L}_{bc}}$. We can see that the *call* statement, after extracting the method parameters from the operand stack, calls recursively predicate `main/3` in order to execute the callee. Upon return from the method execution, the return value is pushed on the operand stack of the new state and execution proceeds normally. Also, we do not need to carry the call stack explicitly within the state, but only the information for the current environment, i.e., states are of the form `st(M,PC,OStack,LocalV)`. This is because the call stack is already available by means of the calls for predicate `main/3` .

The compositional treatment of methods in $Int^{bs}_{\mathscr{L}_{bc}}$ is not only essential to enable modular decompilation (overcome L1, L2 and L3) but also to solve the recursion problem in a simple and elegant way. Indeed, the decompilation based on the big-step interpreter $Int^{bs}_{\mathscr{L}_{bc}}$ does not present L4. For example, the decompilation of a recursive method $m1$ starts from the call `main(ml,_,_)` and then reaches a call `main(ml,args,_)` where `args` represents the particular arguments in the recursive call. This call is flagged as dangerous by local control and the derivation is stopped. The important points are that, unlike before, no re-computation is needed as the second call is necessarily an instance of the first one and, besides, there is no information loss associated to the generalization of the call stack, as there is no stack. The recursion problem was first detected in [17] and a solution based on computing regular approximations during PE was proposed. Although feasible in theory, such technique might be too inefficient in practice and problematic to scale it up to realistic applications due to the overhead introduced by the underlying analysis. Another solution is proposed in [20], where a simpler control-flow analysis is performed before PE in order to collect all possible instructions which might follow the *return*. Such information may then be used to recover information lost by the generalization. This solution turns out to be also impractical for our purposes when considering realistic programs that make intensive use of library code (e.g. Java bytecode) as many continuations can follow. Our solution does not require the use of static analysis and, as our experiments show, does not pose scalability problems.

It is important to note that the idea of using a big-step semantics for describing the interpreter in order to achieve modular (de-)compilation is equally useful in the offline approach to interpretive decompilation. Furthermore, to the best of our knowledge, our idea is novel and has not been proposed before, neither in online nor in offline PE of interpreters.

```
execute(S,S) :-
   S = st(M,PC,[_Top|_],_),
   bytecode(M,PC,return,_).
execute(S,Sf) :-
   S = st(M,PC,_,_),
   bytecode(M,PC,Inst,_),
   step(Inst,S,S'),
   execute(S',Sf).
```

```
step(invoke(M'),S,S') :-
   S = st(M,PC,OS,LV),
   next(M,PC,PC'),
   split_OS(M',OS,Args,OSRs),
   main(M',Args,RV),
   S' = st(M,PC',[RV|OSRs],LV).
```

Fig. 4. Fragment of big-step $\mathscr{L}_{bc}$ interpreter $Int^{bs}_{\mathscr{L}_{bc}}$.

5.2. Guiding online PE with annotations

We now present the annotations we use to provide additional control information to PE. They are instrumental for obtaining the quality decompilation we aim at. We use the annotation schema: "[*Precond*] ⇒ *AnnPred*" where *Precond* is an optional precondition defined as a logic formula, *Ann* is the kind of annotation (*Ann* ∈ {memo, rescall}) and *Pred* is a predicate descriptor, i.e., a predicate name and distinct free variables. Such annotations are used by local control when a call for *Pred* is found as follows:

- **memo**: The current call is not further unfolded and is later transferred to the global control to carry out its specialization separately. It is then replaced by a call to the specialized version.
- **rescall**: The current call is not further unfolded. Unlike calls marked **memo**, the current call is not transferred to the global control. Therefore the call is added to the residual code without modification.

In the following, we denote by $\mathsf{unfold}^{\mathscr{A}}_{\trianglelefteq_T}$ the unfolding operator of Section 2.2 enhanced to use the above-mentioned annotations. We adopt the same names for the annotations as in offline PE [31] (**rescall** stands for residual call while **memo** stands for *memoise*, i.e., pass the call to the *memo* table[1]). However, in offline PE they are the *only* means to control termination while in our method they are only used to improve the accuracy in the local control. As another difference, in offline PE, **rescall** annotations are used only for built-ins. In principle, their use for internal predicates could threaten PE-completeness if a call is residualized but it is not an instance of some call in the final set L^{pe} (i.e., it is not closed by L^{pe}). In the next section, we illustrate the importance of **rescall** annotations also for internal predicates to enable *separate* PE. The role of **memo** becomes important to control the structure of the decompiled programs as we will see in Section 6.

5.3. Modular decompilation

In order to achieve modular decompilation, it is instrumental to allow performing *separate* decompilation. In the interpretive approach this requires being able to perform separate PE, i.e., to be able to specialize parts of the program independently and then join the specializations together to form the residual program. For instance, consider a self-contained logic program P partitioned in a set $\{P_1,\ldots,P_n\}$ of mutually disjoint subprograms which preserve predicate boundaries, i.e., for any predicate *pred* in P we have that all rules for *pred* are in the same partition P_j, for some $j \in \{1,\ldots,n\}$. Consider also the sets of terms $S_1,\ldots,S_n$ such that all calls in S_i correspond to predicates defined in P_i, $i = 1,\ldots,n$. We can now define $S = S_1 \cup \cdots \cup S_n$ and the usual notions of closedness and independence are applicable. A *separate* partial evaluation for P and S is obtained as the union of the individual specializations w.r.t. each corresponding set of calls, i.e., $\bigcup_{P_i \in P} PE(P_i, \emptyset, S_i)$. One additional difficulty for separate PE is related to the use of renaming for guaranteeing independence (see Definition 3), since renaming requires a global table which is not available when generating code for the individual subprograms. A simple strategy which we will use in our modular decompilation is to allow polyvariant specialization (i.e. multiple specialized versions per predicate) for calls to predicates locally defined in the subprogram P_i being partially evaluated, but to resort to monovariant specialization (i.e. only one specialized version per predicate) for predicates used across subprogram boundaries. Then, the renaming can use a local renaming table, which must guarantee that there will be no name clash with renamed calls from other subprograms.

We present now a modular decompilation scheme which, by combining the big-step interpreter with the use of **rescall** annotations, enables separate decompilation and ensures *correctness* (i.e., it is sound and complete w.r.t. internal methods).

Definition 5 ($MOD\text{-}DECOMP_{\mathscr{L}_{bc}}$). Given a $\mathscr{L}_{bc}$-bytecode program P_{bc}, a modular LP-decompilation of P_{bc} can be obtained as:

$$MOD-DECOMP_{\mathscr{L}_{bc}}(P_{bc}) = \bigcup_{\forall m \in defs(P_{bc})} PE(Int^{bs}_{\mathscr{L}_{bc}} \cup code(m), \mathscr{A}_{mod}(m), S(m))$$

where the set of annotations $\mathscr{A}_{mod}(m) = \{(m \in calls(m)) \Rightarrow \texttt{rescall}\ main(m,\_,\_)\}$ and the initial set of calls $S(m) = \{main(m,\_,\_)\}$.

Let us briefly explain the above-mentioned definition. Now the function PE is executed once per method defined in P_{bc}, starting each time from a set of calls, S_m, which contains a call of the form main(m,_,_) for method m. The set $\mathscr{A}_{mod}$ contains a **rescall** annotation which affects all methods invoked (but not necessarily internal) inside P_{bc}. When a method invocation is to be decompiled, the call step(invoke(m′),_,_) occurs during unfolding. We can see that, by using the big-step interpreter in Fig. 4, a subsequent call main(m′,_,_) will be generated. As there is a **rescall** annotation which affects all methods invoked in the program, such call is not unfolded but rather remains residual. If m′ is internal, a corresponding decompilation from the call main(m′,_,_) will be, or has already been, performed since function PE is executed for every method in P_{bc}. Thus, completeness is ensured for internal predicates.

Example 1. By applying function $MOD\text{-}DECOMP_{\mathscr{L}_{bc}}$ to the $\mathscr{L}_{bc}$-bytecode program in Fig. 2 we execute PE once for each of the four methods in the program. In each execution we specialize the interpreter w.r.t. the calls main(fact,_,_), main(gcd,_,_), main(lcm,_,_), and main(abs,_,_). We obtain the following LP-decompilation:

```
main(lcm,[B,C],A) :- main(gcd,[B,C],D),
                     D\=0,
                     E is B*C,
                     A is E//D.
main(lcm,[A,B],0) :- main(gcd,[A,B],0).

main(gcd,[B,0],A) :- main(abs,[B],A).
main(gcd,[B,C],A) :- C\=0,
                     D is B rem C,
                     execute_1(C,D,A).

execute_1(A,0,C) :- main(abs,[A],C).
execute_1(A,B,F) :- B\=0, H is A rem B,
                    execute_1(B,H,F).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[B],A) :- B\=0, C is B-1,
                    main(fact,[C],D), A is B*D.
main(fact,[0],1).
```

[1] This is how the list of atoms to be partially evaluated, named L^{pe} in Section 2.2, is usually denoted in offline PE.

The structure of the original program w.r.t. method calls is preserved, as the residual predicate for lcm contains an invocation to the definition of gcd, which in turn invokes abs, as it happens in

the original bytecode. Moreover, we now obtain an effective decompilation for the recursive method `fact` where the interpretive layer is completely removed without the need of any analysis. Thus, L1 and L4 have been successfully solved.

The following theorem ensures the correctness of modular decompilation for the big-step bytecode interpreter. Completeness can be ensured by excluding calls to external methods not defined in the bytecode. It is independent of the way the interpreter is defined, as the closedness condition for the internal methods is enforced by our definitions of $\mathscr{A}_{mod}$ and S_m. Soundness holds in the case of our interpreter, because the only calls which are transferred to the global control are instances of `main/3` and `execute/2` and their first argument is the method's name, which makes them mutually exclusive. A post-processing of renaming is thus optional, but it can be necessary to ensure that the independence condition is met for other interpreters.

Theorem 2 (correctness). *Consider a $\mathscr{L}_{bc}$-bytecode program P_{bc}, a concrete input I and the $\mathscr{L}_{bc}$-bytecode interpreter $Int^{bs}_{\mathscr{L}_{bc}}$. Let P'_{bc} be the result of MOD-DECOMP$_{\mathscr{L}_{bc}}(P_{bc})$. Then, A is an answer for $P'_{bc} \cup \{I\}$ iff A is an answer obtained running P_{bc} on $Int^{bs}_{\mathscr{L}_{bc}}$ with input I.*

Proof. Let us first prove the completeness of modular decompilation. This requires to prove the closedness condition as stated in Definition 2. We first have to exclude calls to *external* predicates not defined in the bytecode for which we do not obtain an answer in P'_{bc}. Thus, we need to ensure closedness for the calls which have `rescall` annotations and are internal. For the remaining internal calls, closedness is already ensured by traditional PE (Theorem 1). We reason by contradiction. Consider a method invocation to m' which has a **rescall** annotation $true \Rightarrow$ `rescall`$main(m', \_, \_)$ but it is not covered by L^{pe}. This leads to a contradiction because, function PEis executed $\forall m \in defs(P_{bc})$, including m'. Thus, there is an initial call `main(`m'`, _, _)` in $S_{m'}$ and hence it is covered by the final set L^{pe}.

In order to prove the correctness of our modular decompilation scheme, the full code of the interpreter must be studied. Here we focus on proving independence as stated in Definition 3. In the case of $Int^{bs}_{\mathscr{L}_{bc}}$, it is implied by the facts that: (1) the only recursive definitions are `main/3` and `execute/2` and the remaining predicates are always evaluable (in the sense of [41]), (2) thus every call manipulated by the global control is an instance of `main/3` or `execute/2` and (3) all such instances include the method name in some of their (sub-)arguments, which makes them mutually exclusive and hence independent. Since we have proved independence and closedness of the resulting terms, by Theorem 1, we have the correctness of modular decompilation. □

We now characterize the notion of *modular-optimality* in decompilation which ensures that (1) only the code associated to internal methods is decompiled, thus, we can have external calls (e.g., to libraries) which are not decompiled and overcome L3; (2) and each method is decompiled only once and thus we overcome L2.

Proposition 1 (modular-optimality). *Given a $\mathscr{L}_{bc}$-bytecode program P_{bc}, function MOD-DECOMP$_{\mathscr{L}_{bc}}$ only decompiles the code corresponding to internal methods defined in P_{bc}, and the code of each method is decompiled once.*

Proof. Only internal methods of P_{bc} are decompiled because all calls are annotated as **rescall** and hence they are not transferred to the global control. Then, we must prove that each method is decompiled once. The proof follows by contradiction. Assume that a method m is decompiled $n > 1$ times. This means that during the PE process, there have been n calls of the form $main(m, \_, \_)$ that have been unfolded. This leads to a contradiction as there is a **rescall** annotation which affects every method which is called in the program $main(m, \_, \_)$. This prevents from unfolding $main(m, \_, \_)$ and the result follows. □

Note that modular decompilation gives a monovariant treatment to methods in the sense that it does not allow creating specialized versions of method definitions. This is against the usual spirit in PE, where polyvariance is a main goal to achieve further specialization. However, in the context of decompilation, we have shown that a monovariant treatment is necessary to enable scalability and to preserve program structure. It naturally raises the question whether a polyvariant treatment could achieve, even if at the cost of efficiency and loss of structure, a better quality decompilation. Note that enabling polyvariant specialization in the modular setting can be trivially done by not introducing **rescall** annotations for certain selected methods which should be treated in a polyvariant manner. Our experience indicates that there is often a small quality gain at the price of a highly inefficient decompilation.

6. An optimal decompilation scheme

The main issue is whether it is possible to obtain, by means of interpretive decompilation, programs whose *quality* is equivalent to that obtained by dedicated decompilers; issue *(c)* in Section 1. We will show now that, using the most effective unfolding strategies of PE, code for the same program point can be emitted (i.e. it can be decompiled) several times, which degrades both the efficiency and the quality of decompilation. In order to obtain results which are comparable to those of dedicated decompilers, it makes sense to use similar heuristics. Since decompilers first build a *control flow graph* (CFG) for the method, which guides the decompilation process, we now study how a similar notion can be used for controlling PE of the interpreter.

Let us explain *block-level* decompilation by means of an example. Consider the method m_{bl} in Fig. 5. The source code is shown to the left, the relevant bytecode in the center and its CFG to the right. As customary, the CFG [1] consists of basic blocks which contain a sequence of non-branching bytecode instructions and which are connected by edges which describe the possible flows originated from the branching instructions (such as conditional jumps, exceptions, and virtual method invocation). In our small language $\mathscr{L}_{bc}$, conditional jumps (i.e., `if`$\diamondsuit$ and `if0` $\diamondsuit$) are the only branching instructions. A *divergence point* (D point) is a program point (bytecode index) from which more than one branch originates; likewise, a *convergence point* (C point) is a program point where two or more branches merge. In the CFG of m_{bl}, the only divergence (respectively, convergence) point is pc_i (respectively, pc_k).

By using the decompilation scheme presented so far, we obtain the SLD tree shown in Fig. 6, in which all calls are completely unfolded as there is no termination risk (nor **rescall** annotation). The decompiled code is shown under the tree. We use $\{\texttt{res}_X\}$ to refer to the residual code emitted for BlockX and $\texttt{cond}_i$ to refer to the condition associated to the branching instruction at $\texttt{pc}_i$ ($\overline{\texttt{cond}_i}$ denotes its negation). The quality of the decompiled code is not optimal due to:

D. Decompiled code $\{\texttt{res}_A\}$ for BlockA is duplicated in both rules. During PE, this code is evaluated once but, due to the way resultants are defined (see codegen in Section 2.2), each rule contains the decompiled code associated to the whole branch of the tree. This code duplication brings in two problems: it increases considerably the size of decompiled programs and also makes their execution slower. For

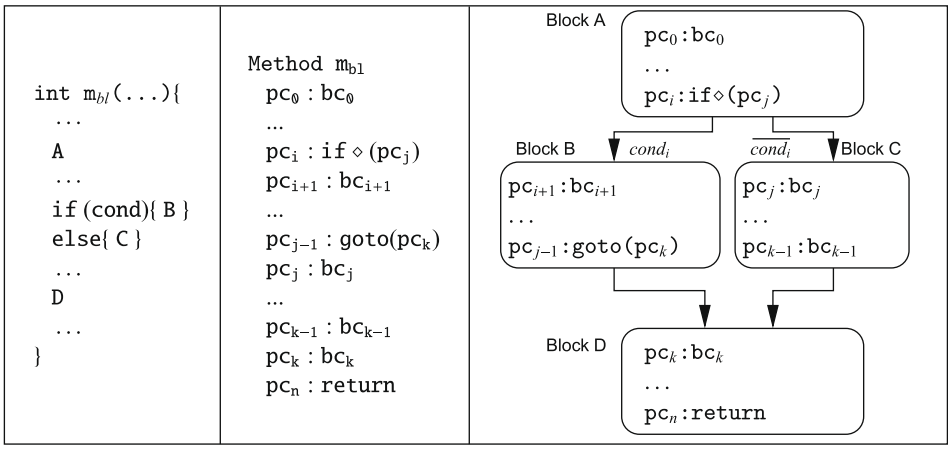


Fig. 5. Source code, $\mathscr{L}_{bc}$-bytecode and CFG of m_{bl} method.

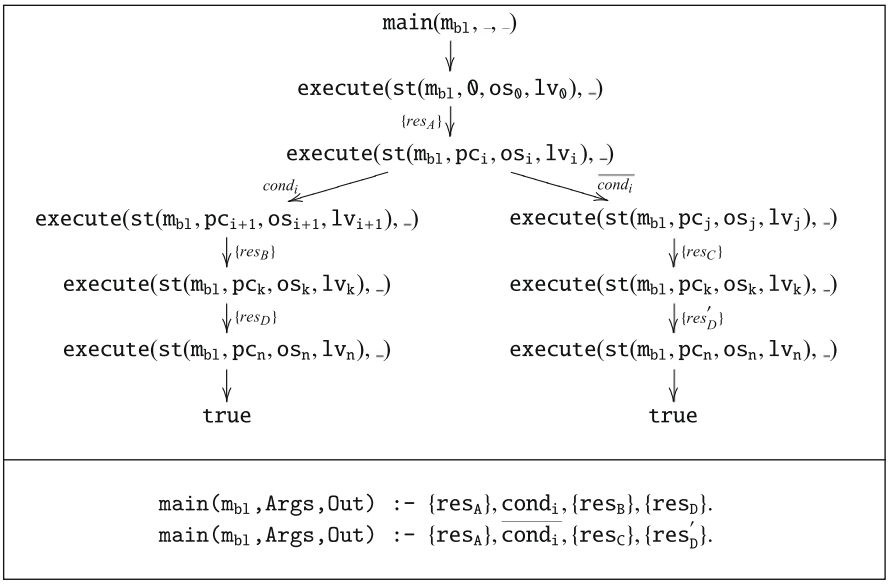


Fig. 6. Unfolding SLD tree and decompiled code of m_{bl} method.

instance, when $\overline{\texttt{cond}_i}$ holds, the execution goes unnecessarily through $\{\texttt{res}_A\}$ in the first rule, fails to prove $\texttt{cond}_i$ and, then, attempts the second rule.

C. Decompiled code of BlockD is again emitted more than once. Each rule for the decompiled code contains a (possibly different) version, $\{\texttt{res}_D\}$ and $\{\texttt{res}'_D\}$, of the code of BlockD. Unlike above, at PE time, the code of BlockD is actually evaluated in the context of $\{\texttt{cond}_i, \{\texttt{res}_B\}\}$ and then re-evaluated in the context of $\{\overline{\texttt{cond}_i}, \{\texttt{res}_C\}\}$. Convergence points thus might degrade both efficiency (and endanger scalability) and quality of decompilation (due to larger residual code).

The amount of repeated residual code grows exponentially with the number of C and D points and the amount of re-evaluated code grows exponentially with the number of C points. Thus, we now aim for an *optimal*, *block-level* decompilation that helps overcome problems D and C above. Intuitively, a block-level decompilation must produce a residual rule for each block in the CFG. This can be achieved by building SLD trees which correspond to each single block, rather than expanding them further. Note that this idea is against the typical spirit of PE which, in order to maximize the propagation of static information, tries to build SLD trees as large as possible and only stops unfolding when there is termination risk.

The **memo** annotations presented in Section 5.2 facilitate the design of the optimal interpretive decompilation scheme. In particular, we can easily force the unfolding process to stop at D points by including a **memo** annotation for `execute/2` calls whose *PC* corresponds to a D point. In the example, unfolding stops at pc_i as desired. Regarding C points, an additional requirement is to partially evaluate the code on blocks starting at these points at most once. The problem is similar to the polyvariant vs monovariant treatment in the decompilation of methods in Section 5.3, by viewing entries to blocks as method calls. Not surprisingly, the solution can be achieved similarly in our setting by: (1) stopping the derivation at `execute/2` calls whose *PC* corresponds to C points and (2) passing the call to the global control, and ensuring that it is evaluated in a sufficiently generalized context which covers all incoming contexts. The former point is ensured by the use of **memo** annotations and the latter by including in the initial set of atoms a generalized call of the form $\texttt{execute}(\texttt{st}(\texttt{m}_{\texttt{bl}}, \texttt{pc}_k, \_, \_), \_)$ for all C points, which forces such generalization. The next definition presents the optimal decompilation scheme where div_points(m) and conv_points(m) denote, respectively., the set of D points and C points of a method m.

Definition 6 ($\mathit{OPTIMAL\text{-}DECOMP}_{\mathscr{L}_{bc}}$). Given a $\mathscr{L}_{bc}$-bytecode program P_{bc}, an optimal, modular LP-decompilationof P_{bc} can be obtained as:

$$\mathit{OPTIMAL-DECOMP}_{\mathscr{L}_{bc}}(P_{bc}) = \bigcup_{\forall m \in defs(P_{bc})} PE(Int^{bs}_{\mathscr{L}_{bc}} \cup code(m), \mathscr{A}_{opt}(m), S(m))$$

$$\mathscr{A}_{blocks}(m) = \{pc \in \mathsf{div\_points}(m) \cup \mathsf{conv\_points}(m) \Rightarrow \mathtt{memo}\ execute(st(m, pc, \_, \_), \_)\}$$
$$S(m) = \{main(m,\_,\_)\} \cup \{execute(st(m,pc,\_,\_),\_) | pc \in \mathsf{conv\_points}(m)\}$$
$$\mathscr{A}_{opt}(m) = \mathscr{A}_{mod}(m) \cup \mathscr{A}_{blocks}(m)$$

An important point is that, unlike annotations used in offline PE [29] which are generated by only taking the interpreter into account, our annotations for the optimal decompilation are generated by taking into account the particular program to be decompiled. Importantly, both the annotations and the initial set of calls can be computed automatically by performing two passes on the bytecode (see, e.g., [2,43]).

The result of performing an optimal decompilation on m_{bl} is:

$$\mathtt{main}(\mathtt{m_{b1}}, \mathtt{Args}, \mathtt{Out}) : -\{\mathtt{res_A}\}, \mathtt{execute_1}(\ldots).$$
$$\mathtt{execute_1}(\ldots) : -\mathtt{cond_i}, \{\mathtt{res_B}\}, \mathtt{execute_2}(\ldots).$$
$$\mathtt{execute_1}(\ldots) : -\overline{\mathtt{cond_i}}, \{\mathtt{res_C}\}, \mathtt{execute_2}(\ldots).$$
$$\mathtt{execute_2}(\ldots) : -\{\mathtt{res_D}\}.$$

Now, the residual code associated to each block appears once in the code. This ensures that the optimal decompilation preserves the CFG shape as dedicated decompilers do. Thus, the quality of our decompiled code is as good as that obtained by state-of-the-art decompilers [2,35] but with the advantages of interpretive decompilation (see Section 1). We formally prove the quality of our proposed decompilation scheme in the next proposition.

Proposition 2 (block-optimality). *Given a bytecode program P_{bc}, the optimal decompilation function $\mathit{OPTIMAL\text{-}DECOMP}_{\mathscr{L}_{bc}}$ ensures that: (I) residual code for each bytecode instruction in P_{bc} is emitted once in the decompiled program, (II) each bytecode instruction in P_{bc} is evaluated at most once during PE and (III) there is at most one residual rule for each block in the bytecode.*

Proof. The proof follows easily by contradiction.

In order to prove (I), consider that two resultants contain residual code for the same bytecode instruction. This can be due to two reasons. (a) There is in the SLD tree a D point which leads to two derivations. This is not possible because D points are annotated as **memo** and hence the derivation must have been stopped. (b) There are two separate trees which contain derivations for instructions of the same block. Then, this block must be a C block. Hence, it is not possible because C points are annotated as **memo** and hence the derivation must have stopped before.

We focus now on D blocks to prove (II). Consider that there have been two evaluations of an instruction pc_x within a D block B starting at $pc_1 \in \mathsf{conv\_points}(M)$. Then, there must have been two different instances $\mathtt{execute(st(M, pc_1, A, B), C))}$ and, later, $\mathtt{execute(st(M, pc_1, D, E), F))}$. This is not possible because there exists the initial call $\mathtt{execute(st(M, pc_1, \_, \_), \_))}$ in S_m which does not allow the evaluation of $\mathtt{execute(st(M, pc_1, D, E), F))}$.

For (III) to be false there must exist a block in the CFG which includes a sequence of bytecode instructions $\langle pc_1 : b_1, \ldots, pc_i : b_i, \ldots, pc_n : b_n \rangle$, with $i \geqslant 2$ and $n \geqslant i$ such that the residual program contains a rule for the subsequence of bytecode instructions $\langle pc_i : b_i, \ldots, pc_j : b_j \rangle$ with $i \in \{2, \ldots, n\}$ and $j \in \{i, \ldots, n\}$. This requires that the local control stops unfolding for a call of the form $\mathtt{execute(st(M, pc_i, A, B), C))}$. According to our optimal local control strategy, execution of a bytecode instruction is only left residual if the instruction at position pc_i in method M is a C point or a D point, which contradicts the assumption that the sequence of instructions $\langle pc_1 : b_1, \ldots, pc_i : b_i, \ldots, pc_n : b_n \rangle$ belongs to the same block in the CFG. □

After taking into account the central observation from Section 5 that the interpreter should be written in big-step semantics, each of the optimality criteria above is simpler or more complicated to achieve depending on the local control strategy we use. For example, if we start from a modular decompiler as discussed in Section 5 above, optimality criterion (III) will in general be satisfied, but not criteria (I) nor (II) since the local control rule tends to over-specialize calls which results in re-evaluating expressions and emitting code multiple times.

Conversely, if we use an offline partial evaluator, the natural local control rule to use is to residualize all calls to `execute` and then filter out all information other than the method signature and program counter when transferring the atom to the global control method. This control strategy trivially guarantees optimality criteria (I) and (II) since it guarantees that each bytecode instruction is decompiled independently of the others. However, it tends to under specialize and namely it does not satisfy the optimality criterion (III): as soon as there is a block with more than one bytecode instruction, which is almost always the case, the specialized program will contain a separate rule for each and every bytecode instruction in the block. As a result, the residual program thus obtained is high-level in the sense that it is written in Prolog. However, its control strategy is heavily influenced by the fact that we decompile JBC (instead of converting, e.g. from Java source) and the decompiled program is not at all similar to the Prolog program which a Prolog programmer would write for performing the same task. Since an important objective of decompilation is to enable program understanding and analysis, we argue that programs which satisfy this optimality criterion (III), like the ones we generate, are easier to reason about.

Another important observation is that the costly mechanisms, namely the type-based homeomorphic embedding [4] and the polyvariance control from [18], used for controlling the PE that were used earlier to achieve the results in Sections 4.2 and 5.3 are not needed anymore using the optimal decompilation scheme. Instead, the following trivial control operators can be used: unfold unfolds all calls except those matching a **memo** or **rescall** annotation, and abstract adds to the set S_{i+1} every call in L^{pe} which is not an instance of any call in S_i. It can be easily proved that termination is ensured both at the local and at the global control level thanks to the annotations and the initial set of atoms provided to the PE in Definition 6. Intuitively, in the local control, the only source of potential non-termination is a loop in the bytecode program, and there is always a convergence point associated with it, therefore termination is guaranteed as the corresponding **memo** annotation associated with the divergence point will force unfolding to stop. In the global control, we have to ensure that the set of atoms to be specialized does not grow infinitely. The only atoms which can potentially occur in the set are those of the form $\mathtt{execute(st(m, pc, \_, \_), \_))}$ with $pc \in \mathsf{div\_points}(m) \cup \mathsf{conv\_points}(m)$. Those with $pc \in \mathsf{conv\_points}(m)$ are always an instance of an atom already present in the set thus they are never added. As regards those with $pc \in \mathsf{div\_points}(m)$, it can be derived from Proposition 2 that only one single version of the atom can be added to the set, otherwise the corresponding bytecode will be traversed more

than once. The complete proof of termination will require a complete formalization of the control rules and a complete definition of the bytecode interpreter used, and is not given in this paper.

7. Decompiling object-oriented bytecode

In this section we present the main extensions that are needed to apply interpretive decompilation to a bytecode language with object-oriented features. Such features include: dynamic memory allocation, arrays, classes and objects, inheritance and polymorphism. We first present an extension of $\mathscr{L}_{bc}$, denoted as $\mathscr{L}^{O}_{bc}$, which includes all these features in the spirit of Java bytecode. An $\mathscr{L}^{O}_{bc}$-bytecode program P_{bc} consists of a set of classes $classes(P_{bc}) = \mathscr{C}$, partially ordered w.r.t the subclass relation. The class is the basic (de-)compilation unit of $\mathscr{L}^{O}_{bc}$. Each class $c \in \mathscr{C}$ contains information about the class it extends[2] and the fields and methods it declares. A method (field) is uniquevocally identified by its method (field) signature which is of the form $c : mn\ (c : fn)$, where c is the class in which it is declared and mn (fn) the method (field) name. The name `init` is reserved for the class initialization methods (constructors). We write *defs(c)* to denote the set of *internal* method signatures defined in the class c. Some other features of Java bytecode such as interfaces, static methods and static fields, exceptions, access control and types besides integers and references are not yet considered to simplify the presentation. We show later in Section 8 that they do not add any complication to the decompilation process. As in $\mathscr{L}_{bc}$, the code associated to a method m, denoted $code(m)$, consists of a sequence of indexed bytecode instructions. The $\mathscr{L}^{O}_{bc}$ instruction set is:

$$
\begin{aligned}
Inst_{\mathscr{L}^{O}_{bc}} ::= {} & \texttt{push(x)} \mid \texttt{load(v)} \mid \texttt{store(v)} \mid \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div} \mid \texttt{rem} \mid \texttt{neg} \mid \\
& \texttt{if}\diamondsuit\texttt{(pc)} \mid \texttt{if0}\ \diamondsuit\texttt{(pc)} \mid \texttt{goto(pc)} \mid \texttt{return} \mid \texttt{invoke(ms)} \\
& \texttt{newarray}(\tau) \mid \texttt{arrload} \mid \texttt{arrstore} \mid \texttt{arraylength} \\
& \texttt{new(c)} \mid \texttt{getfield(fs)} \mid \\
& \texttt{putfield(fs)} \mid \texttt{dup} \mid \texttt{ifnull} \mid \texttt{ifnonnull}
\end{aligned}
$$

where τ is a type signature, $\tau \in C \cup \{int\}$, `c` is a class, $\texttt{c} \in C$, `ms` a method signature and `fs` a field signature. The first two rows correspond to the instructions in $\mathscr{L}_{bc}$, which are already described in Section 4.1. The third row comprises the instructions to manipulate arrays: creation (`newarray(`τ`)`), loading and storing an element (respectively, `arrload` and `arrstore`), and consulting the array length (`arraylength`). The last row contains instructions to manipulate objects: object creation (`new`), accessing and modifying fields (respectively, `getfield` and `sputfield`), the `dup` instruction duplicates the reference stored on top of the operand stack and new conditional branching instructions for references `ifnull` and `ifnonnull`. As we are omitting static methods, the `invoke` instruction always corresponds to virtual invocations. For simplicity, all methods are supposed to return a value (except for constructors).

7.1. Handling the heap during decompilation

An $\mathscr{L}^{O}_{bc}$-bytecode program manipulates both integers and references to objects and arrays[3]. Therefore, besides using an operand stack and an array of local variables, it uses a *heap* where objects and arrays are allocated. Thus, the first design decision which we have to take is how to represent the $\mathscr{L}^{O}_{bc}$ heap in Prolog. A first alternative would be to represent objects as Prolog terms. Each object could have as main functor an identifier for its class and as many arguments as fields there are in the corresponding class. The problem with this approach is that, though apparently simple, logic programs do not allow destructive updates, i.e., once an argument (variable) gets associated (unified) to a functor, it cannot be associated to a different functor, as the subsequent unification would fail. A possible way out would be the use of the non-pure Prolog predicate `setarg`, which allows overwriting values. However, the programs thus obtained are not very amenable to static analysis since the use of `setarg` breaks the declarative nature of logic programs and introduces all difficulties associated to the analysis of shared mutable data structures, which is well known to pose major difficulties to static analysis. Since one of our main motivations is to analyze the programs obtained by our decompilation, we opt for another alternative which produces declarative programs. In this other alternative, the heap is passed as an explicit argument which is not overwritten, but rather modified as needed. We now describe how the $Int^{bs}_{\mathscr{L}_{bc}}$ interpreter is extended to handle the heap, denoted $Int_{\mathscr{L}^{O}_{bc}}$. The extensions include:

- The main predicate of the interpreter is now of the form `main(M,InArgs,Hin,Top,Hout)`, where the new additional parameters `Hin` and `Hout` stand, respectively, for the input and the output heap of the method. Note that now `M` is not just a method name but a method signature.
- The state carried out by the interpreter has to include an extra argument for the heap. Thus, it is of the form `st(M,PC,OS,LV,H)`, where `H` is the current heap. Again, `M` is a method signature.
- The corresponding rules for the `step/3` associated with the new added bytecode instructions have to be provided. As an example, consider the implementation in our Prolog interpreter of the `getfield(f)` operation:

```
step(getfield(F),S,S') :-
    S = st(M,PC,[ref(R)|S],L,H),
    S' = st(M,PC',[V|S],L,H),
    next(M,PC,PC'),
    getfield(H,R,F,V).
```

There is an important difference between the heap and the operand stack which affects the decompilation process. While the operand stack is a local data structure of each method execution, the heap is a global entity which stores objects that can be created at any point during a program's execution and besides objects can be aliased. Basically, the consequence is that the heap becomes unknown at PE time (typically it is a logic variable) and, hence, most operations involving the heap cannot be fully evaluated and have to appear residual in the decompiled code. Essentially, we treat the heap during decompilation as an abstract data type with a set of operations which manipulate it. For instance, this is the case of the atom `getfield(H,R,F,V)` in the code above. In Fig. 7 we list all the predicates used in the interpreter, which use the heap, together with a description of their functionality. Output arguments are underlined (the rest are input). Note that these are exactly the set of predicates that can appear residual in our decompiled programs besides arithmetic operations, calls to `main/5` (bytecode methods) and calls to `execute_i/n` (bytecode blocks). Fig. 8 depicts to the right side an example of a decompiled program containing heap operations.

7.2. Decompilation with classes

Object-oriented programs, both high-level and bytecode, are structured as a set of classes. It makes sense then to devise a decompilation scheme where the decompilation is done at the level of classes: we decompile one class at a time, and within each class, we decompile each declared method at a time. Clearly, it is

[2] If a class does not explicitly extend any class, it implicitly extends class `Object`.

[3] We use the special functor `ref/1` to distinguish references in the Prolog representation.

Creation operations:	
`new(H,C,R,H')`	H' is the heap obtained from the heap H by creating a new object of type C. The new object is stored at location R.
`newarray(H,T,N,R,H')`	H' is the heap obtained from the heap H by creating a new array with N elements of type T. The new array is stored in location R.
Accessing operations:	
`getfield(H,R,F,V)`	Field F of object at location R in the heap H has the value V.
`arrload(H,R,I,V)`	The element at index I of the array at location R in the heap H has the value V.
`arraylength(H,R,N)`	The length of the array at location R in the heap H is N.
Setting operations:	
`putfield(H,R,F,V,H')`	The heap H' results from setting the field F of object at location R in the heap H with the value V.
`arrstore(H,R,I,V,H')`	The heap H' results from setting the element at index I of the array at location R in the heap H with the value V.

Fig. 7. Residual heap operations.

```
class A {
public int n;
public A(){this.n=1;}
[ 0:load(0)
  1:invoke(Object:init)
  2:load(0)
  3:push(1)
  4:putfield(A:n)
  5:return ]
int m(){return n+1;}
[ 0:load(0)
  1:getfield(A:n)
  2:push(1)
  3:add
  4:return ]
}
```

```
:- module('A',[init/4,m/4]).

:- use_module('Object',[init/4]).

init([ref(T)],H0,_Out,H2) :-
   'Object':init([ref(T)],H0,_,H1),
   putfield(H1,T,'A':n,1,H2).

m(ref(T),H0,N',H0) :-
   getfield(H0,T,'A':n,N),
   N' is N+1.
```

```
class B extends A {
[ 0:load(0)
  1:invoke(A:init)
  2:return ]
int m(){return n+2;}
[ 0:load(0)
  1:getfield(A:n)
  2:push(1)
  3:add
  4:return ]
}
```

```
:- module('B',[init/4,m/4]).

:- use_module('A',[init/4]).

init([ref(T)],H0,_,H1) :-
   'A':init([ref(T)],H0,_,H1).

m(ref(T),H0,N',H0) :-
   getfield(H0,T,'A':n,N),
   N' is N+2.
```

```
class Foo {
int foo(A a){return a.m();}
[ 0:load(1)
  1:invoke(A:m)
  2:return ]
}
```

```
:- module('Foo',[foo/4]).
:- use_module('A',[m/4]).
:- use_module('B',[m/4]).

foo([_,Ref],H0,Out,H1) :-
   resolve(Ref,H0,C),
   C:m([Ref],H0,Out,H1).
```

Fig. 8. Example of decompilation with classes.

convenient to structure decompiled programs at a similar level. A natural choice in a module-structured language like Prolog is to make use of modules such that each class is decompiled in a corresponding module. A Prolog module consists of a module name, a list of exported predicates, a list of imported modules (optionally together with the list of predicates imported from each module)

and the code (set of clauses) of the exported and auxiliary predicates. We propose a decompilation scheme with the following characteristics:

1. There is a Prolog module per class in the bytecode program, with the same name.
2. A Prolog predicate is associated with each declared method, with the same name. As we will see later this is done via a simple post-renaming of the `main/5` atoms.
3. Each Prolog module: (1) exports all predicates corresponding to the methods declared in the corresponding class and, (2) imports all needed external predicates from the corresponding modules.

The decompilation scheme with classes is formalized as follows:

Definition 7 ($CLASS\text{-}DECOMP_{\mathscr{L}^O_{bc}}$).
Given a class of an $\mathscr{L}^O_{bc}$-bytecode program, an optimal, LP-decompilation of c is defined as:

$$CLASS-DECOMP_{\mathscr{L}^O_{bc}}(c) = \phi\left(\bigcup_{\forall m \in defs(c)} PE(Int_{\mathscr{L}^O_{bc}} \cup code(m), A_{class}(m), S(m))\right)$$

where $A_{class}(m) = A_{opt}(m) \cup A_{heap}$, being $A_{opt}(m)$ and $S(m)$ the sets in Definition 6 adapted for the new interpreter $Int_{\mathscr{L}^O_{bc}}$. The set A_{heap} denotes the set of **rescall** annotations to residualize the heap operations in Fig. 7.

The function ϕ denotes a simple post-processing which is applied over the set of predicates resulting from the successive PEs, producing a Prolog module with the characteristics enumerated above. Basically, (1) it produces the corresponding module header, with the lists of imported and exported predicates, and (2) it renames all atoms of the form `main(c:mn,Args,Hin,Out,Hout)` as `c:Mn(Args,Hin,Out,Hout)`. This is interpreted in Prolog as a module-qualified call, i.e., a call to predicate `mn` of module `c`.

We can now define an object-oriented decompilation of an $\mathscr{L}^O_{bc}$-bytecode program as follows.

$$OO-DECOMP_{\mathscr{L}^O_{bc}}(P_{bc}) = \bigcup_{\forall c \in classes(P_{bc})} CLASS-DECOMP_{\mathscr{L}^O_{bc}}(c).$$

Observe that $OO\text{-}DECOMP_{\mathscr{L}^O_{bc}}$ takes a set of $\mathscr{L}^O_{bc}$ classes and produces a set of Prolog modules. An example of the application of $OO\text{-}DECOMP_{\mathscr{L}^O_{bc}}$ is shown in Fig. 8. On the left side, we show the Java-like source code of our example program, together with the $\mathscr{L}^O_{bc}$-bytecode which is shown within parentheses. Again, we show the source code for clarity but the decompilation works on the bytecode. It has three classes, `A`, `B` and `Foo`. `B` extends `A` inheriting field `n` and re-defining method `m`. Method `foo` of `Foo` invokes method `m` on an object declared of type `A`. The $\mathscr{L}^O_{bc}$-bytecode of each declared method is shown within parentheses. On the right side, we show the Prolog decompiled program. It has three modules `A`, `B` and `Foo`. Note that, in Prolog, strings starting with an uppercase letter are interpreted as variables and the rest as functors or constants. Thus, if one wants to use the special constant *A*, the notation `'A'` has to be used. The `Foo` module will be explain in the next section. The corresponding predicates are exported/imported. See for example how module `A` exports predicates `init/4` and `m/4`, and imports predicate `init/4` from module `Object`[4]. Note that all predicates have four arguments as they come from the corresponding instance of `main/5`. There are several calls in the decompiled program which are module-qualified call, e.g., the call to `init/4` inside module `B`.

7.3. Virtual invocations

An important feature of object-oriented languages is *polymorphism* in the presence of virtual invocations. In a virtual invocation, the method to be executed is determined at run-time depending on the actual type of the corresponding object. As it happens with heap operations, the operation to *resolve* the method to be called cannot be performed at PE time, and then has to be residualized. In fact, the information needed (object type) for the resolution is in the heap which is in general unknown at PE time as we saw in Section 7.1. In the following we show the code corresponding to the `invoke` operation for virtual invocations in our $Int_{\mathscr{L}^O_{bc}}$ interpreter:

```
step(invoke(C:M'),S,S'):-
    S = st(M,PC,OS,LV,H),next(M,PC,PC'),
    split_OS(M',OS,[Ref|Args],OSRs),
    resolve(Ref,H,C'),main(C':M',[Ref|Args],H,RV,H'),
    S' = st(M,PC',[RV|OSRs],LV,H').
```

Predicate `resolve/3` is encharged of performing the above-mentioned method resolution. Given the call `resolve(Ref,H,C')` it proceeds as follows: (1) the class of the current object at location `Ref` in the heap `H` is obtained, and (2) due to inheritance, it can happen that the method is not declared in such class, then it has to go up in the classes hierarchy until reaching a class in which the method is declared. This class is finally returned in `C'`. Then, the call to `main/4` is done with the method signature `C':M'`. As with heap operations, the corresponding **rescall** annotation has to be provided to make the corresponding `resolve/3` atom appear in the decompiled code. It always appears immediately before calls corresponding to method invocations (except calls to constructors[5]).

Example 2. As an example, consider method `foo` of class `Foo` in Fig. 8. The method `m` is invoked on an object declared of type `A`. However, variable `a` can actually store at run-time a reference to an object of class `A` or of any class extending `A`, in this case `B`. Whether to execute method `m` of class `A` or of class `B` is thus determined at run-time. In the Prolog code, we can observe the call to `resolve/3` immediately before the call to predicate `m/4`, which is module-qualified with the obtained module.

8. Experimental evaluation

We report on two different implementations of a decompiler for full (sequential) Java Bytecode into Prolog. For the first one we extend an already existing powerful online PE, the one integrated in the `CiaoPP` analysis and specialization system [21]. This partial evaluator implements several unfolding rules and abstraction operators. This allows us to compare the different decompilation schemes explained in the paper, in particular, to compare to the non-optimal ones. Such comparison is presented in Section 8.1. However, the overhead introduced by using such generic and powerful tool prevents us from competing with ad hoc decompilers as regards efficiency (decompilation times). For this reason, we have carried out a second implementation for which we have written a stand-alone PE which only contains the local and global strategies required by an optimal decompilation. This partial evaluator is integrated into a decompilation tool called jbc2prologwhich also includes a Java bytecode interpreter. This makes it possible to both obtain optimal decompilation and be competitive in terms of efficiency with ad hoc decompilers. A thorough comparison against

[4] Constructor methods first call the constructor of its super-class, in this case `Object`.

[5] Invocations to constructors are never virtual. They can be statically resolved.

the decompiler in the COSTA[3] system and against the JDec[8] decompiler is presented in Section 8.2.

Both implementations consider full sequential Java bytecode. The extensions needed to handle the features not considered in $\mathscr{L}^{O}_{bc}$ (exceptions, static fields and methods, access control, etc.) do not add any special complication to the decompilation scheme. For instance, exception handling is simply dealt with as another source of branching. This certainly makes the size of our decompiled programs grow considerably, although this is something every decompiler of a real-life language has to deal with. Solutions based on static analysis exist which allow avoiding some exception branches. For example, *nullity* analysis can be used to avoid considering branches corresponding to null-pointer exceptions which are proved to be non-null, which reduces the size of the code considerably. These analyses can be easily incorporated in our decompilation tool and it is indeed a subject of future work.

8.1. Assessing the scalability of decompilation

For the experimental evaluation, we have used the standardized set of benchmarks in the JOlden suite [22]. In particular, our first goal is to compare the scalability of the *optimal* decompilation scheme (see Definition 7) against that of the *modular* (non-optimal) one (see Definition 5). Here it comes the need to use the partial evaluator of CiaoPP, as it combines the power of online control operators like type-based homeomorphic embedding [4], with the ability of adding conditional annotations as described in Definition 6. As most programs in the JOlden suite make an extensive use of library methods, non-modular decompilation cannot be assessed as we run into memory problems when trying to decompile the code of library calls. Fig. 9 depicts four charts measuring different aspects of the decompilation. In order to reason about scalability, we assess the differences between the non-optimal and the optimal approaches, as well as how the size of the programs affects the decompilation. The times are computed as the arithmetic mean of five runs on an Intel Core 2 Duo at 1.86 GHz with 2GB of RAM, running Linux 2.6.24-21. We measure two aspects of the decompilation: the decompilation time (in milliseconds) and the decompiled program size (in bytes). It should be noticed that absolute data are not required to assess scalability issues. We rather need relative data per instruction in order to prove that it does not increase with the size of the programs. The relative decompilation time indicates the efficiency of the process and the size of decompiled programs are directly related to the decompilation quality. Each point $[X, Y]$ in the charts corresponds to the decompilation of a single method in the JOlden suite, where X represents the number of instructions of the method and Y the measured data (time per instruction or decompiled program size per instruction). The charts in the left-hand side show the data obtained (times in the top chart and sizes in the bottom one) for both the non-optimal and the optimal decompilation. The variations in the optimal decompilation cannot be appreciated when combined with the non-optimal. Thus, we include in the charts on the right-hand side the figures for the optimal decompilation in isolation such that we adjust the scale on the Y-axis to the domain of the data.

From the charts, we conclude: (1) Times per instruction are notably larger for the smallest methods, as can be seen by looking at the initial curve in the charts. This is because the overhead introduced for starting a new decompilation is more noticeable when the time for decompilation itself is small, while it becomes negligible for larger methods. The same happens for the size of the decompiled programs. (2) The optimal decompilation achieves important speedups in general (for all methods with more than 40 instructions). Besides, it obtains significantly smaller decompiled programs. The speedups per package range from 3.36 in **power** to 31.4 in **bisort** for the decompilation times; and from 2.5 times smaller in **power** to 9 times smaller in **bisort** for the decompiled program sizes. Note that there is a clear correspondence between both measures, since C points introduce both inefficiency and size increase in decompilation, as explained in Section 6. Moreover, modular decompilation runs out of memory for some of the largest methods. This is again related to code dupli-

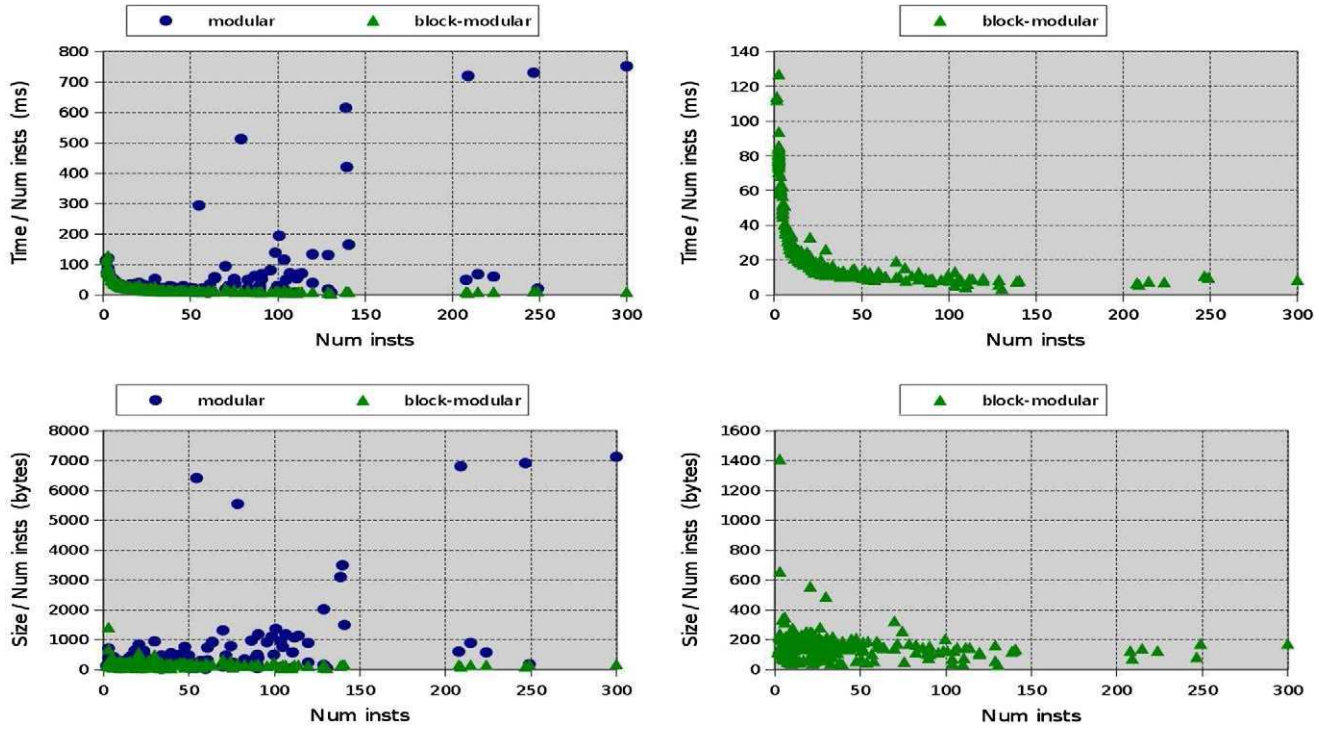


Fig. 9. Evaluating the scalability of *optimal* decompilation with the JOlden Suite.

Table 1
Efficiency of jbc2prolog.

Benchmark				jbc2prolog						COSTA	JDec
Pack	N_{cls}	N_{mths}	N_{ins}	T_{bl}	T_{sps}	T_{ge}	T_{pe}	T_{cg}	T_{j2p}	T_{costa}	T_{jdec}
bisort	2	15	554	10	10	0	147	10	177	170	1802
bh	9	73	2012	57	28	0	652	70	807	860	7394
em3d	4	22	713	27	7	0	184	26	243	347	3386
health	6	27	973	37	13	0	224	26	300	420	4822
mst	5	31	703	14	4	0	173	20	210	317	3958
perimeter	9	46	838	37	9	0	134	13	193	363	6564
power	6	32	1927	43	24	4	566	64	701	693	5330
treeadd	2	12	308	6	3	0	67	14	90	143	1600
tsp	2	16	946	17	13	4	367	26	427	380	1948
voronoi	6	73	1781	50	19	7	673	62	810	1023	5270
Overall	51	347	10755	297	131	14	3186	330	3958	4717	42074

cation (C and D points) and (re-)evaluation (C points), which grow exponentially. (3) The most important conclusion is that, while in the non-optimal decompilation both the times and the sizes per instruction greatly increase with the size of the benchmarks, this does not happen in the optimal scheme. In the optimal decompilation, these figures are totally stable (mostly constant) for all methods with more than 40 instructions. This shows that both the decompilation times and the decompiled program sizes are *linear* with the size of the input bytecode program, thus demonstrating the scalability of our optimal decompilation. One might wonder why there are still small variations in the ratio. In our experience, the following points also matter: (1) the complexity of the control flow of the methods, (2) the relative complexity of the bytecode instructions used, e.g., instructions which operate in the heap tend to produce more residual code, (3) the structure w.r.t. methods of the program, e.g., classes with methods of medium size tend to result in better decompilations than those with few large methods or many small ones.

8.2. Efficiency: comparing against other decompilers

To assess the efficiency of our approach we compare the decompilation times we get using our tool jbc2prologw.r.t. those obtained using the decompiler in the COSTAsystem [3] and those of the well-known Java decompiler JDec[8]. COSTAis a COST and TERMINATION analysis tool for Java bytecode. It performs a decompilation of the bytecode into a rule-based representation before the actual analysis phase with the aim of making the analysis design simpler. This decompilation basically consists of two parts. First, the CFG for each method is built and then, for each block in the CFG, an associated rule is produced. We have chosen the COSTAdecompiler to compare the efficiency of interpretive decompilation because COSTAis also implemented in Prolog and hence the underlying implementation language performance is identical. The resulting decompiled program is a set of rules which resemble our Prolog clauses in several aspects: recursion is the only form of iteration and conditional instructions are captured by guarded rules. However, there are still some differences w.r.t. our decompiled programs: a) in COSTAthe operand stack is explicitly flattened and represented by means of local variables whereas in jbc2prologPE together with argument filtering automatically achieve this effect, and b) we represent the heap explicitly in the residual programs as explained in Section 7.1. These two features together are important since in the programs decompiled using COSTA(or CiaoPP) all bytecode instructions remain residual and have to be taken as builtins, i.e., predefined procedures by analysis. In contrast, in jbc2prologbytecode instructions are interpreted at decompilation time and converted into basic Prolog instructions such as unifications and arithmetic or into the ADT operations in Fig. 7 for those instructions involving the heap. As a result, extending an existing Prolog analyzer to analyze JBC decompiled programs is simpler using our decompiler than using those in COSTA[2] or CiaoPP[35], since the decompiled programs are executable and the analysis does not need to be extended with any further builtins.

Again we use the set of benchmarks in the JOlden suite [22]. Table 1 shows the times taken (in milliseconds) by each of the different phases of jbc2prologtogether with the total time used by the COSTA and JDecdecompiler for each package of the JOlden suite. All times are computed as the arithmetic mean of five runs, in this case on a Intel Core 2 Quad Q9300 at 2.5 GHz with 1.95GB of RAM, running Linux 2.6.27-9. In particular, for each JOlden package we measure: the total number of classes, methods and instructions in the package (columns N_{cls}, N_{mths} and N_{ins}), the time taken by the different phases of jbc2prolog, namely, the parsing and loading time of the `.class`file (column T_{bl}), the pre-processing time to infer the divergence and convergence points of the bytecode program (column T_{sps}), the generation of the entries to the PE (column T_{ge}), the actual specialization time (column T_{pe}) and the time taken by the code generation phase (column T_{cg}). Finally, last three columns show respectively the total times taken by jbc2prolog(column T_{j2p}), the COSTAdecompiler (column T_{costa}) and the JDecdecompiler (column T_{jdec}). The last row shows the overalls of all measurements. We can see that the whole JOlden suite is decompiled by jbc2prologin less than 4 s versus the 4.7 s in COSTAand the 42 s in JDec. It can be concluded that our results our competitive with those of an ad hoc decompiler. In particular, we see that they are similar to those obtained in COSTA. Furthermore, in most examples, jbc2prolog is more efficient than COSTA, especially in **voronoi**, **perimeter** and **treeAdd**. On the other hand we can see that jbc2prologis about ten times faster than JDec. Our conclusion in this regard is that it is very difficult to compare with decompilers written in other programming languages, since the performance of the implementation language heavily influences the decompilation time.

9. Related work

Previous work in *interpretative* (de-)compilation has mainly focused on proving that the approach is feasible for small interpreters and medium-sized programs. The focus has been on demonstrating its *effectiveness*, i.e., that the so-called interpretation layer can be removed from the compiled programs. To achieve effectiveness, offline [29], online [5,20,39] and hybrid [30] PE techniques have been assessed and novel control strategies have been proposed and proven effective [18,4].

Our work starts off from the premise that interpretive decompilation is feasible and effective as proved by previous work and studies further issues which have not been explored yet. Let us re-

view now related work in the field of decompilation of low-level code. Related work on the PE of interpreters has been already compared in Section 1 and in several places throughout the paper. The work by Breuer and Bowen [9] is only tangentially related to ours. They propose a general method for compiling decompilers from the specifications of (non-optimizing) compilers. The main idea is that a data type specification for a programming-language grammar can be remolded into a functional program that enumerates all the abstract syntax trees of the grammar. It is showed that by relying on this technique a decompiler can be generated from a simple Occam-like compiler specification. The only similarity with our work is that decompiled programs are somehow obtained from specifications (in our case of the interpreter and in their case of the compiler). However, the underlying methods are technically different and also they do not provide a practical solution for ensuring applicable conditions for their technique.

As regards (direct) decompilation of low-level back to source code, it has been the subject of a good amount of research. Decompilation can be attempted at different levels, with different levels of success. The most complicated case is when decompiling binary executables. There are a good number of associated complications, such as recovering the control flow. One intrinsic problem in this approach is that it is not possible in general to distinguish code from data statically. See, e.g. [10,42] and their references for a discussion on the problems and techniques for binary decompilation. The next level is decompilation of assembly, see, e.g. [11]. This shares many of the complications associated to the decompilation of binaries, since current hardware architectures are rather complex, but at least it is possible to separate code from data. The following level is decompilation of code to be run on a virtual machine. This is in general easier to perform since virtual machines are usually simpler than the current hardware architectures and because often the code for this virtual machines (bytecode) must satisfy certain behavior restrictions (must be *verifiable* [27]) and types of variables are available. As a result, in the particular case of decompilation of Java bytecode back to Java source, a number of successful commercial and free software decompilers exist which are able to handle a large class of bytecode programs, especially those generated by common Java compilers, i.e., `javac`. Nevertheless, things become more complicated when the Java bytecode has been generated by an obfuscator, and especially when an optimizing compiler, or a compiler from other programming languages such as Haskell, Eiffel, ML, Ada, and Fortran is used. See, e.g. [37] and its references for a good account on the existing Java bytecode decompilers and the difficulties associated to its decompilation.

As already mentioned, there exist several analyzers for Java bytecode which use a higher-level intermediate representation and which can be seen as ad hoc decompilers. In particular, both the COSTA [3] and `CiaoPP` [21] systems have a front-end which converts bytecode into an intermediate representation which is then the input to the subsequent analysis. Though in both cases the intermediate representation is similar, in the case of COSTAit is formalized as a rule-based representation [2], whereas in `CiaoPP` it is formalized as Horn clauses, i.e., a logic program [35]. The reason for doing that in `CiaoPP` is that, at least in principle, that allows using the analysis which are already available in `CiaoPP`. However, there is a crucial difference between the logic programs generated in [35] and those generated by our decompiler. Whereas the programs generated by [35] are only meant to be the subject of static analysis and are not executable, the programs we generate can both be subject to analysis or be executed. The reason why the programs in [35] nor those in [2] are executable is because they basically capture the control-flow of the bytecode program, but the basic bytecode instructions themselves remain as *builtins*, i.e., predefined predicates, to the analysis. Analysis results are correct as long as the behavior of such bytecode instructions is safely approximated by the analysis. Producing fully executable logic programs as the result of decompilation is not trivial since many of the bytecode instructions operate on the heap in a way or another. Thus, in order to make an executable decompiled program we need to introduce the JVM heap explicitly in the logic program. All this is done automatically in our approach.

10. Conclusions

We argue that *declarative languages* and the technique of *partial evaluation* have nowadays a large application field within the development of analysis, verification, and model checking tools for modern programming languages. On the one hand, declarative languages provide a convenient intermediate representation which allows (1) representing all iterative constructs (loops) as recursion, independently of whether they originate from iterative loops (conditional and unconditional jumps) or recursive calls, and (2) all variables in the local scope of the methods (formal parameters, local variables, fields, and stack values in low-level languages) can be represented uniformly as explicit arguments of a declarative program. On the other hand, the technique of PE enables the automatic (de-)compilation of a (complicated) modern program to a simple declarative representation by just writing an interpreter for the modern language in the corresponding declarative language and using an existing partial evaluator.

The resulting intermediate representation greatly simplifies the development of the above-mentioned tools for modern languages and, more interestingly, existing advanced tools developed for declarative programs (already proven correct and effective) can be directly applied on it. In previous work [5], by reasoning on our decompiled residual programs, we have automatically proved in the `CiaoPP`system some non-trivial properties of Java bytecode programs such as termination, run-time error freeness and infer bounds on its resource consumption. In order to prove run-time error freeness, we have proposed an enhanced bytecode interpreter which computes, in addition to the return value of the method called, also the trace which captures the computation history. Such traces represent the semantic steps used and therefore do not only represent instructions, as the context has also some importance. They have allowed us to distinguish, for example, for a same instruction, the step that throws an exception from the normal behavior. For example, `invokevirtual_step_ok` and `invokevirtual_step_NullPointerException` represent, respectively, a normal method call and a method call on a null reference that throws an exception. Such additional flexibility of interpretive decompilation has allowed to prove run-time error freeness in a straightforward way by simply specifying the property of being error-free as verifying that the corresponding trace in the decompiled program does not contain an exceptional step.

A unique feature of our decompiled programs is that they represent the whole program state, i.e., in contrast to [35,2,43], our decompiled programs contain a representation of the heap in addition to the operand stack. The advantage is decompiled programs are fully *executable* which in turn broadens their application field. As an example, recently we have developed a novel framework for *test case generation* [45] of bytecode by relying on our decompiled Prolog programs. Basically, the standard approach to generating test-cases statically is to perform a *symbolic* execution of the program [12,36,38,24,19], where the contents of variables are expressions rather than concrete values. The symbolic execution produces a system of *constraints* consisting of the conditions to execute the different paths. This happens, for instance, in branching instructions, like if-then-else, where we might have to generate test-cases for the two alternative branches and hence accumulate

the conditions for each path as constraints. The fact that our decompiled programs are executable Prolog programs allows us to directly rely on the available techniques for *constraint* logic programs (where backtracking is inherent to the language) to carry out such symbolic execution.

Finally, a main objective of our work has been to investigate, and provide the necessary techniques, to make interpretive decompilation scale in practice. A further goal has been to ensure, and provide the techniques, that decompiled programs preserve the structure of the original programs and that their quality is comparable to that obtained by dedicated decompilers. We believe that the techniques proposed in this paper, together with their experimental evaluation, provide for the first time actual evidence that the interpretive theory proposed by Futamura in the 70 s is indeed an appealing and feasible alternative to the development of ad hoc decompilers from modern languages to intermediate representations.

Acknowledgement

We gratefully acknowledge the anonymous referees for many useful comments and suggestions. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* and TIN-2008-05624 *DOVES* projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

[1] A.V. Aho, R. Sethi, J.D. Ullman, Compilers – Principles, Techniques and Tools, Addison-Wesley, 1986.
[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of Java bytecode, in: Rocco De Nicola (Ed.), 16th European Symposium on Programming, ESOP'07, Lecture Notes in Computer Science, vol. 4421, Springer, 2007, pp. 157–172.
[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, COSTA: design and implementation of a cost and termination analyzer for Java bytecode, in: Post-proceedings of Formal Methods for Components and Objects (FMCO'07), LNCS, vol. 5382, Springer-Verlag, 2008, pp. 113–133.
[4] E. Albert, J. Gallagher, M. Gómez-Zamalloa, G. Puebla, Type-based homeomorphic embedding and its applications to online partial evaluation, in: 17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07), LNCS, vol. 4915, Springer-Verlag, 2008, pp. 23–42.
[5] E. Albert, M. Gómez-Zamalloa, L. Hubert, G. Puebla, Verification of Java bytecode using analysis and transformation of logic programs, in: Nineth International Symposium on Practical Aspects of Declarative Languages, LNCS, vol. 4354, Springer-Verlag, 2007, pp. 124–139.
[6] E. Albert, G. Puebla, J. Gallagher, Non-leftmost unfolding in partial evaluation of logic programs with impure predicates, in: 15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05), LNCS, vol. 3901, Springer-Verlag, 2006, pp. 115–132.
[7] B. Barras et al., The Coq proof assistant reference manual: Version 6.1, Technical Report RT-0203, 1997. <citeseer.ist.psu.edu/barras97coq.html>.
[8] Swaroop Belur, Kartik Bettadapura, Jdec: Java Decompiler, <http://jdec.sourceforge.net/>.
[9] Peter T. Breuer, Jonathan P. Bowen, Decompilation: The enumeration of types and grammars, ACM Trans. Program. Lang. Syst. 16 (5) (1994) 1613–1647.
[10] Cristina Cifuentes, K. John Gough, Decompilation of binary programs, Softw. Pract. Exper. 25 (7) (1995) 811–829.
[11] Cristina Cifuentes, Doug Simon, Antoine Fraboulet, Assembly to high-level language translation, in: ICSM, 1998, pp. 228–237.
[12] L.A. Clarke, A system to generate test data and symbolically execute programs, IEEE Trans. Softw. Eng. 2 (3) (1976) 215–222.
[13] R. DeLine, K.R.M. Leino, BoogiePL: a typed procedural language for checking object-oriented programs, Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
[14] Y. Futamura, Partial evaluation of computation process – an approach to a compiler–compiler, Syst. Comput. Controls 2 (5) (1971) 45–50.
[15] J. Gallagher, Transforming logic programs by specializing interpreters, in: Proceedings of the 7th European Conference on Artificial Intelligence, 1986.
[16] J.P. Gallagher, Tutorial on specialisation of logic programs, in: Proceedings of the PEPM'93, ACM Press, 1993, pp. 88–98.
[17] J.P. Gallagher, J.C. Peralta, Using regular approximations for generalisation during partial evaluation, in: Proceedings of the SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, ACM Press, 2000, pp. 44–51.
[18] M. Gómez-Zamalloa, E. Albert, G. Puebla, Improving the decompilation of Java bytecode to prolog by partial evaluation, in: ETAPS Ws on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07), ENTCS, vol. 190, 2007, pp. 85–101.
[19] A. Gotlieb, B. Botella, M. Rueher, A clp framework for computing structural test data, in: Computational Logic, 2000, pp. 399–413.
[20] Kim S. Henriksen, John P. Gallagher, Abstract interpretation of pic programs through logic programming, in: SCAM'06 Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society, 2006, pp. 184–196.
[21] M. Hermenegildo, G. Puebla, F. Bueno, P. López-García, Integrated program debugging, verification, and optimization using abstract interpretation (and The Ciao System Preprocessor), Sci. Comput.Programming 58 (1–2) (2005) 115–140.
[22] JOlden Suite Collection, <http://www-ali.cs.umass.edu/DaCapo/benchmarks.html>.
[23] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall, New York, 1993.
[24] J.C. King, Symbolic execution and program testing, Commun. ACM 19 (7) (1976) 385–394.
[25] J. Komorovski, An introduction to partial deduction, in: A. Pettorossi (Ed.), Meta Programming in Logic, Proceedings of META'92, LNCS, vol. 649, Springer-Verlag, 1992, pp. 49–69.
[26] J. Launchbury, A natural semantics for lazy evaluation, in: POPL, 1993, pp. 144–154.
[27] Xavier Leroy, Java bytecode verification: algorithms and formalizations, J. Automated Reasoning 30 (3–4) (2003) 235–269.
[28] M. Leuschel, Homeomorphic embedding for online termination of symbolic methods, in: The Essence of Computation, LNCS, vol. 2566, Springer, 2002, pp. 379–403.
[29] M. Leuschel, S. Craig, M. Bruynooghe, W. Vanhoof, Specialising interpreters using offline partial deduction, in: Program Development in Computational Logic, Lecture Notes in Computer Science, vol. 3049, Springer, 2004, pp. 340–375.
[30] M. Leuschel, S. Craig, D. Elphick, Supervising offline partial evaluation of logic programs using online techniques, in: LOPSTR, Lecture Notes in Computer Science, vol. 4407, Springer, 2006, pp. 43–59.
[31] M. Leuschel, J. Jørgensen, W. Vanhoof, M. Bruynooghe, Offline specialisation in prolog using a hand-written compiler generator, TPLP 4 (1–2) (2004) 139–191.
[32] Michael Leuschel, Morten Heine Sørensen, Redundant argument filtering of logic programs, in: LOPSTR, 1996, pp. 83–103.
[33] J.W. Lloyd, J.C. Shepherdson, Partial evaluation in logic programming, The Journal of Logic Programming 11 (1991) 217–242.
[34] J.W. Lloyd, Foundations of Logic Programming, second extended ed., Springer, 1987.
[35] M. Méndez-Lojo, J. Navas, M. Hermenegildo, A flexible (C)LP-based approach to the analysis of object-oriented programs, in: 17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07), 2007.
[36] C. Meudec, Atgen: automatic test data generation using constraint logic programming and symbolic execution, Softw. Test. Verif. Reliab. 11 (2) (2001) 81–96.
[37] Jerome Miecznikowski, Laurie J. Hendren, Decompiling Java bytecode: problems traps and pitfalls, in: R. Nigel Horspool (Ed.), CC, Lecture Notes in Computer Science, vol. 2304, Springer, 2002, pp. 111–127.
[38] R.A. Müller, C. Lembeck, H. Kuchen, A symbolic java virtual machine for test case generation, in: IASTED Conf. on Software Engineering, 2004, pp. 365–371.
[39] J.C. Peralta, J. Gallagher, H. Sağlam, Analysis of imperative programs through analysis of constraint logic programs, in: Proceedings of the SAS'98, LNCS, vol. 1503, 1998, pp. 246–261.
[40] D. Pichardie, Bicolano (Byte Code Language in cOq), <http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html>.
[41] G. Puebla, E. Albert, M. Hermenegildo, Efficient local unfolding with ancestor stacks for full prolog, in: Proceedings of the LOPSTR'04, LNCS, vol. 3573, Springer, 2005, pp. 149–165.
[42] Benjamin Schwarz, Saumya K. Debray, Gregory R. Andrews, Disassembly of executable code revisited, in: Arie van Deursen, Elizabeth Burd (Eds.), WCRE, IEEE Computer Society, 2002, pp. 45–54.
[43] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, P. Co, Soot – a Java optimization framework, in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), 1999, pp. 125–135.
[44] John Whaley, Dzintars Avots, Michael Carbin, Monica S. Lam, Using datalog with binary decision diagrams for program analysis, in: Kwangkeun Yi (Ed.), APLAS, Lecture Notes in Computer Science, vol. 3780, Springer, 2005, pp. 97–118.
[45] Hong Zhu, Patrick A.V. Hall, John H.R. May, Software unit test coverage and adequacy, ACM Comput. Surv. 29 (4) (1997) 366–427.

Test Data Generation of Bytecode by CLP Partial Evaluation

Elvira Albert[1], Miguel Gómez-Zamalloa[1], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. We employ existing *partial evaluation* (PE) techniques developed for Constraint Logic Programming (CLP) in order to automatically generate *test-case generators* for glass-box testing of bytecode. Our approach consists of two independent CLP PE phases. (1) First, the bytecode is transformed into an equivalent (decompiled) CLP program. This is already a well studied transformation which can be done either by using an ad-hoc decompiler or by specialising a bytecode interpreter by means of existing PE techniques. (2) A second PE is performed in order to supervise the generation of test-cases by execution of the CLP decompiled program. Interestingly, we employ control strategies previously defined in the context of CLP PE in order to capture *coverage criteria* for glass-box testing of bytecode. A unique feature of our approach is that, this second PE phase allows generating not only test-cases but also test-case *generators*. To the best of our knowledge, this is the first time that (CLP) PE techniques are applied for test-case generation as well as to generate test-case generators.

1 Introduction

Bytecode (e.g., Java bytecode [19] or .Net) is becoming widely used, especially, in the context of mobile applications for which the source code is not available and, hence, there is a need to develop verification and validation tools which work directly on bytecode programs. Reasoning about complex bytecode programs is rather difficult and time consuming. In addition to object-oriented features such as objects, virtual method invocation, etc., bytecode has several low-level language features: it has an unstructured control flow with several sources of branching (e.g., conditional and unconditional jumps) and uses an operand stack to perform intermediate computations.

Test data generation (TDG) aims at automatically generating test-cases for interesting test *coverage criteria*. The coverage criteria measure how well the program is exercised by a test suite. Examples of coverage criteria are: *statement coverage* which requires that each line of the code is executed; *path coverage* which requires that every possible trace through a given part of the code is executed; etc. There are a wide variety of approaches to TDG (see [27] for a survey).Our work focuses on *glass-box* testing, where test-cases are obtained from the concrete program in contrast to *black-box* testing, where they are deduced from a specification of the program. Also, our focus is on *static* testing,

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 4–23, 2009.

where we assume no knowledge about the input data, in contrast to *dynamic* approaches [9,14] which execute the program to be tested for concrete input values.

The standard approach to generating test-cases statically is to perform a *symbolic* execution of the program [7,22,23,17,13], where the contents of variables are expressions rather than concrete values. The symbolic execution produces a system of *constraints* consisting of the conditions to execute the different paths. This happens, for instance, in branching instructions, like if-then-else, where we might want to generate test-cases for the two alternative branches and hence accumulate the conditions for each path as constraints. The symbolic execution approach has been combined with the use of *constraint solvers* [23,13] in order to: handle the constraints systems by solving the feasibility of paths and, afterwards, to instantiate the input variables. For the particular case of Java bytecode, a symbolic JVM machine (SJVM) which integrates several constraints solvers has been designed in [23]. A SJVM requires non-trivial extensions w.r.t. a JVM: (1) it needs to execute the bytecode symbolically as explained above, (2) it must be able to backtrack, as without knowledge about the input data, the execution engine might need to execute more than one path. The backtracking mechanism used in [23] is essentially the same as in logic programming.

We propose a novel approach to TDG of bytecode which is based on PE techniques developed for CLP and which, in contrast to previous work, does not require the devising a dedicated symbolic virtual machine. Our method comprises two CLP PE phases which are independent. In fact, they rely on different execution and control strategies:

1. *The decompilation of bytecode into a CLP program.* This has been the subject of previous work [15,3,12] and can be achieved automatically by relying on the first Futamura projection by means of partial evaluation for logic programs, or alternatively by means of an adhoc decompiler [21].
2. *The generation of test-cases.* This is a novel application of PE which allows generating test-case generators from CLP decompiled bytecode. In this case, we rely on a CLP partial evaluator which is able to solve the constraint system, in much the same way as a symbolic abstract machine would do. The two control operators of a CLP partial evaluator play an essential role: (1) The local control applied to the decompiled code will allow capturing interesting coverage criteria for TDG of the bytecode. (2) The global control will enable the generation of *test-case generators*. Intuitively, the TDG generators we produce are CLP programs whose execution in CLP returns further test-cases on demand without the need to start the TDG process from scratch.

We argue that our CLP PE based approach to TDG of bytecode has several advantages w.r.t. existing approaches based on symbolic execution: (i) It is more *generic*, as the same techniques can be applied to other both low and high-level imperative languages. In particular, once the CLP decompilation is done, the language features are abstracted away and, the whole part related to TDG generation is totally *language independent*. This avoids the difficulties of dealing

with recursion, procedure calls, dynamic memory, etc. that symbolic abstract machines typically face. (ii) It is more *flexible*, as different coverage criteria can be easily incorporated to our framework just by adding the appropriate local control to the partial evaluator. (iii) It is more *powerful* as we can generate test-case generators. (iv) It is *simpler* to implement compared to the development of a dedicated SJVM, as long as a CLP partial evaluator is available.

The rest of the paper is organized as follows. The next section recalls some preliminary notions. Sec. 3 describes the notion of CLP *block-level* decompilation which corresponds to the first phase above. The second phase is explained in the remainder of the paper. Sec. 4 presents a naïve approach to TDG using CLP decompiled programs. In Sec. 5, we introduce the block count-k coverage criterion and outline an evaluation strategy for it. In Sec. 6, we present our approach to TDG by partial evaluation of CLP. Sec. 7 discusses related work and concludes.

2 Preliminaries and Notation in Constraint Logic Programs

We now introduce some basic notions about *Constraint Logic Programming* (CLP). See e.g. [20] for more details. A *constraint store*, or *store* for short, is a conjunction of expressions built from predefined predicates (such as term equations and equalities or inequalities over the integers) whose arguments are constructed using predefined functions (such as addition, multiplication, etc.). We let $\overline{\exists}_L\theta$ be the constraint store θ restricted to the variables of the syntactic object L. An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. A *literal* L is either an atom or a constraint. A *goal* $L_1, \ldots, L_n$ is a possibly empty finite conjunction of literals. A *rule* is of the form H :- B where H, the *head*, is an atom and B, the *body*, is a goal. A *constraint logic program*, or *program*, is a finite set of rules. We use *mgu* to denote a most general unifier for two unifiable terms.

The operational semantics of a program P is in terms of its *derivations* which are sequences of reductions between *states*. A *state* $\langle G \mid \theta \rangle$ consists of a goal G and a constraint store θ. A state $\langle L, G \mid \theta \rangle$ where L is a literal can be *reduced* as follows:

1. If L is a constraint and $\theta \wedge L$ is satisfiable, it is reduced to $\langle G \mid \theta \wedge L \rangle$.
2. If L is an atom, it is reduced to $\langle B, G \mid \theta \wedge \theta' \rangle$ for some renamed apart rule (L' :- B) in P such that L and L' unify with mgu θ'.

A *derivation* from state S for program P is a sequence of states $S_0 \rightarrow_P S_1 \rightarrow_P ... \rightarrow_P S_n$ where S_0 is S and there is a reduction from each S_i to S_{i+1}. Given a non-empty derivation D, we denote by *curr_state(D)* and *curr_store*(D) the last state in the derivation, and the store in this last state, respectively. E.g., if D is the derivation $S_0 \rightarrow_P^* S_n$, where $\rightarrow^*$ denotes a sequence of steps, with $S_n = \langle G \mid \theta \rangle$ then $curr\_state(D) = S_n$ and $curr\_store(D) = \theta$. A query is a pair (L, θ) where L is a literal and θ a store for which the CLP system starts a computation from $\langle L \mid \theta \rangle$.

The observational behavior of a program is given by its "answers" to queries. A finite derivation D from a query $Q = (L, \theta)$ for program P is *finished* if $curr\_state(D)$ cannot be reduced. A finished derivation D from a query $Q = (L, \theta)$ is *successful* if $curr\_state(D) = \langle \epsilon \parallel \theta' \rangle$, where ϵ denotes the empty conjunction. The constraint $\exists_L \theta'$ is an *answer* to Q. A finished derivation is *failed* if the last state is not of the form $\langle \epsilon \parallel \theta \rangle$. Since evaluation trees may be infinite, we allow *unfinished* derivations, where we decide not to further perform reductions. Derivations can be organized in execution trees: a state S has several children when its leftmost atom unifies with several program clauses.

3 Decompilation of Bytecode to CLP

Let us first briefly describe the bytecode language we consider. It is a very simple imperative low-level language in the spirit of Java bytecode, without object-oriented features and restricted to manipulate only integer numbers. It uses an operand stack to perform computations and has an unstructured control flow with explicit conditional and unconditional `goto` instructions. A bytecode program is organized in a set of methods which are the basic (de)compilation units of the bytecode. The code of a method m consists of a sequence of bytecode instructions $BC_m = < pc_0 : bc_0, \ldots, pc_{n_m} : bc_{n_m} >$ with $pc_0, \ldots, pc_{n_m}$ being consecutive natural numbers. The instruction set is:

$$\begin{aligned} BcInst ::= {} & \mathsf{push(x)} \mid \mathsf{load(v)} \mid \mathsf{store(v)} \mid \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{rem} \mid \mathsf{neg} \mid \\ & \mathsf{if} \diamond \mathsf{(pc)} \mid \mathsf{if0} \diamond \mathsf{(pc)} \mid \mathsf{goto(pc)} \mid \mathsf{return} \mid \mathsf{call(mn)} \end{aligned}$$

where $\diamond$ is a comparison operator (eq, le, gt, etc.), v a local variable, x an integer, pc an instruction index and mn a method name. The instructions push, load and store transfer values or constants from a local variable to the stack (and vice versa); add, sub, mul, div, rem and neg perform arithmetic operations, rem is the division remainder and neg the negation; if and if0 are conditional branching instructions (with the special case of comparisons with 0); goto is an unconditional branching; return marks the end of methods returning an integer and call invokes a method.

Figure 1 depicts the control flow graphs (CFGs) [1] and, within them, the bytecode instructions associated to the methods `lcm` (on the left), `gcd` (on the right) and `abs` (at the bottom). A Java-like source code for them is shown to the left of the figure. It is important to note that we show source code only for clarity, as our approach works directly on the bytecode. The use of the operand stack can be observed in the example: the bytecode instructions at *pc* `0` and `1` in `lcm` load the values of parameters `x` and `y` (resp.) to the stack before invoking the method `gcd`. Method parameters and local variables in the program are referenced by consecutive natural numbers starting from `0` in the bytecode. The result of executing the method `gcd` has been stored on the top of the stack. At *pc* `3`, this value is popped and assigned to variable `2` (called `gcd` in the Java program). The branching at the end of $\texttt{Block}_1$ is due to the fact that the division bytecode instruction `div` can throw an exception if the divisor is zero (control

```
int lcm(int x,int y){
  int gcd = gcd(x,y);
  return abs(x*y/gcd);
}

int gcd(int x,int y){
  int res;
  while (y != 0){
    res = x%y;
    x = y; y = res;
  }
  return abs(x);
}

int abs(int x){
  if (x >= 0)
    return x;
  else return -x;
}
```

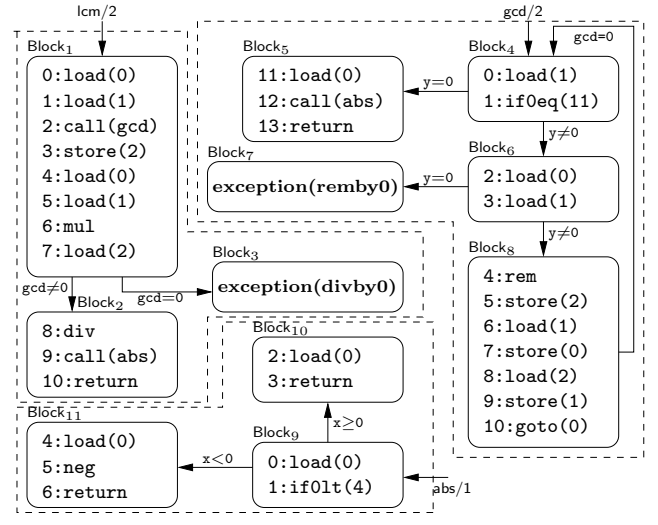


Fig. 1. Working example. Source code and CFGs for the bytecode.

goes to $\texttt{Block}_3$). In the bytecode for `gcd`, we find: conditional jumps, like `if0eq` at *pc* 1, which corresponds to the loop guard, and unconditional jumps, like `goto` in *pc* `10`, where the control returns to the loop entry. Note that the bytecode instruction `rem` can throw an exception as before.

3.1 Decompilation by PE and Block-Level Decompilation

The decompilation of low-level code to CLP has been the subject of previous research, see [15,3,21] and their references. In principle, it can be done by defining an adhoc decompiler (like [2,21]) or by relying on the technique of PE (like [15,3]). The decompilation of low-level code to CLP by means of PE consists in specializing a bytecode interpreter implemented in CLP together with (a CLP representation of) a bytecode program. As the first Futamura projection [11] predicts, we obtain a CLP residual program which can be seen as a decompiled and translated version of the bytecode into high-level CLP source. The approach to TDG that will be presented in the remaining of this paper is independent of the technique used to generate the CLP decompilation. Thus, we will not explain the decompilation process (see [15,3,21]) but rather only state the decompilation requirements our method imposes.

The *correctness* of decompilation must ensure that there is a one to one correspondence between execution paths in the bytecode and derivations in the CLP decompiled program. In principle, depending on the particular type of decompilation –and even on the options used within a particular method– we can obtain different correct decompilations which are valid for the purpose of execution. However, for the purpose of generating useful test-cases, additional requirements

```
lcm([X,Y],Z) :- gcd([X,Y],GCD),P #= X*Y,
                lcm1c([GCD,P],Z).
lcm1c([GCD,P],Z) :- GCD #\= 0,D #= P/GCD,
                    abs([D],Z).
lcm1c([0,_],divby0).

abs([X],Z) :- abs9c(X,Z).
abs9c(X,X) :- X #>= 0.
abs9c(X,Z) :- X #< 0, Z #= -X.

gcd([X,Y],Z) :- gcd4(X,Y,Z).

gcd4(X,Y,Z) :- gcd4c(X,Y,Z).

gcd4c(X,0,Z) :- abs([X],Z).
gcd4c(X,Y,Z) :- Y #\= 0,
                gcd6c(X,Y,Z).

gcd6c(X,Y,Z) :- Y #\= 0,
                R #= X mod Y,
                 gcd4(Y,R,Z).
gcd6c(_,0,remby0).
```

Fig. 2. Block-level decompilation to CLP for working example

are needed: we must be able to define coverage criteria on the CLP decompilation which produce test-cases which cover the *equivalent* coverage criteria for the bytecode. The following notion of *block-level* decompilation, introduced in [12], provides a sufficient condition for ensuring that equivalent coverage criteria can be defined.

Definition 1 (block-level decompilation). *Given a bytecode program BC and its CLP-decompilation P, a block-level decompilation ensures that, for each block in the CFGs of BC, there exists a* single *corresponding rule in P which contains all bytecode instructions within the block.*

The above notion was introduced in [12] to ensure optimality in decompilation, in the sense that each program point in the bytecode is traversed, and decompiled code is generated for it, at most once. According to the above definition there is a one to one correspondence between blocks in the CFG and rules in P, as the following example illustrates. The block-level requirement is usually an implicit feature of adhoc decompilers (e.g., [2,21]) and can be also enforced in decompilation by PE (e.g., [12]).

Example 1. Figure 2 shows the code of the block-level decompilation to CLP of our running example which has been obtained using the decompiler in [12] and uses CLP(FD) built-in operations (in particular those in the `clpfd` library of `Sicstus` Prolog). The input parameters to methods are passed in a list (first argument) and the second argument is the output value. We can observe that each block in the CFG of the bytecode of Fig. 1 is represented by a corresponding clause in the above CLP program. For instance, the rules for `lcm` and `lcm1c` correspond to the three blocks in the CFG for method `lcm`. The more interesting case is for method `gcd`, where the `while` loop has been converted into a cycle in the decompiled program formed by the predicates `gcd4`, `gcd4c`, and `gcd6c`. In this case, since `gcd4` is the head of a loop, there is one more rule (`gcd`) than blocks in the CFG. This additional rule corresponds to the method *entry*. Bytecode instructions are decompiled and translated to their corresponding operations in CLP; conditional statements are captured by the continuation rules.

For instance, in `gcd4`, the bytecode instruction at *pc* 0 is executed to unify a stack position with the local variable `y`. The conditional `if0eq` at *pc* 1 leads to two continuations, i.e. two rules for predicate `gcd4c`: one for the case when `y=0` and another one for `y`$\neq$`0`. Note that we have explicit rules to capture the exceptional executions (which will allow generating test-cases which correspond to exceptional executions). Note also that in the decompiled program there is no difference between calls to blocks and method calls. E.g., the first rule for `lcm` includes in its body a method call to `gcd` and a block call `lcm1c`.

4 Test Data Generation Using CLP Decompiled Programs

Up to now, the main motivation for CLP decompilation has been to be able to perform static analysis on a decompiled program in order to infer properties about the original bytecode. If the decompilation approach produces CLP programs which are executable, then such decompiled programs can be used not only for static analysis, but also for dynamic analysis and execution. Note that this is not always the case, since there are approaches (like [2,21]) which are aimed at producing static analysis targets only and their decompiled programs cannot be executed.

4.1 Symbolic Execution for Glass-Box Testing

A novel interesting application of CLP decompilation which we propose in this work is the automatic generation of glass-box test data. We will aim at generating test-cases which traverse as many different execution paths as possible. From this perspective, different test data should correspond to different execution paths. With this aim, rather than executing the program starting from different input values, a well-known approach consists in performing *symbolic execution* such that a single symbolic run captures the behaviour of (infinitely) many input values. The central idea in symbolic execution is to use constraint variables instead of actual input values and to capture the effects of computation using constraints (see Sec. 1).

Several symbolic execution engines exist for languages such as Java [4] and Java bytecode [23,22]. An important advantage of CLP decompiled programs w.r.t. their bytecode counterparts is that symbolic execution does not require, at least in principle, to build a dedicated symbolic execution mechanism. Instead, we can simply run the decompiled program by using the standard CLP execution mechanism with all arguments being distinct free variables. E.g., in our case we can execute the query `lcm`([`X`, `Y`], `Z`). By running the program without input values on a block level decompiled program, each successful execution corresponds to a different computation path in the bytecode. Furthermore, along the execution, a constraint store on the program's variables is obtained which

can be used for inferring the conditions that the input values (in our case `X` and `Y`) must satisfy for the execution to follow the corresponding computation path.

4.2 From Constraint Stores to Test Data

An inherent assumption in the symbolic execution approach, regardless of whether a dedicated symbolic execution engine is built or the default CLP execution is used, is that all valuations of constraint variables which satisfy the constraints in the store (if any) result in input data whose computation traverses the same execution path. Therefore, it is irrelevant, from the point of view of the execution path, which actual values are chosen as representatives of a given store. In any case, it is often required to find a valuation which satisfies the store. Note that this is a strict requirement if we plan to use the bytecode program for testing, though it is not strictly required if we plan to use the decompiled program for testing, since we could save the final store and directly use it as input test data. Then, execution for the test data should load the store first and then proceed with execution. In what follows, we will concentrate on the first alternative, i.e., we generate actual values as test data.

This postprocessing phase is straightforward to implement if we use CLP(FD) as the underlying constraint domain, since it is possible to enumerate values for variables until a solution which is consistent with the set of constraints is found (i.e., we perform *labeling*). Note, however, that it may happen that some of the computed stores are indeed inconsistent and that we cannot find any valuation of the constraint variables which simultaneously satisfies all constraints in the store. This may happen for unfeasible paths, i.e., those which do not correspond to any actual execution. Given a decompiled method `M`, an integer subdomain `[RMin,RMax]`, the predicate `generate_test_data/4` below produces, on backtracking, a (possibly infinite) set of values for the variables in `Args` and the result value in `Z`.

```
generate_test_data(M,Args,[RMin,RMax],Z) :-
    domain(Args,RMin,RMax), Goal =..[M,Args,Z],
    call(Goal), once(labeling([ff],Args)).
```

Note that the generator first imposes an integer domain for the program variables by means of the call to `domain/3`; then builds the `Goal` and executes it by means of `call(Goal)` to generate the constraints; and finally invokes the enumeration predicate `labeling/2` to produce actual values compatible with the constraints[1]. The test data obtained are in principle specific to some integer subdomain; indeed our bytecode language only handles integers. This is not necessarily a limitation, as the subdomain can be adjusted to the underlying bytecode machine limitations, e.g., $[-2^{31}, 2^{31} - 1]$ in the Java virtual machine. Note that if the variables take floating point values, then other constraint domains such as CLP(R) or CLP(Q) should be used and then, other mechanisms for generating actual values should be used.

[1] We are using the `clpfd` library of `Sicstus` Prolog. See [26] for details on predicates `domain/3`, `labeling/2`, etc.

5 An Evaluation Strategy for *Block-Count*(k) Coverage

As we have seen in the previous section, an advantage of using CLP decompiled programs for test data generation is that there is no need to build a symbolic execution engine. However, an important problem with symbolic execution, regardless of whether it is performed using CLP or a dedicated execution engine, is that the execution tree to be traversed is in most cases infinite, since programs usually contain iterative constructs such as loops and recursion which induce an infinite number of execution paths when executed without input values.

Example 2. Consider the evaluation of the call `lcm([X,Y],Z)`, depicted in Fig. 3. There is an infinite derivation (see the rightmost derivation in the tree) where the cycle {`gcd4,gcd4c,gcd6c`} is traversed forever. This happens because the value in the second argument position of `gcd4c` is not ground during symbolic computation.

Therefore, it is essential to establish a *termination criterion* which guarantees that the number of paths traversed remains finite, while at the same time an interesting set of test data is generated.

5.1 *Block-count*(k): A Coverage Criteria for Bytecode

In order to reason about how interesting a set of test data is, a large series of *coverage criteria* have been developed over the years which aim at guaranteeing that the program is exercised on interesting control and/or data flows. In this section we present a coverage criterion of interest to bytecode programs. Most existing coverage criteria are defined on high-level, structured programming languages. A widely used control-flow based coverage criterion is loop-count(k), which dates back to 1977 [16], and limits the number of times we iterate on loops to a threshold k. However, bytecode has an unstructured control flow: CFGs can contain multiple different shapes, some of which do not correspond to any of the loops available in high-level, structured programming languages. Therefore, we introduce the block-count(k) coverage criterion which is not explicitly based on limiting the number of times we iterate on loops, but rather on counting how many times we visit each block in the CFG within each computation. Note that the execution of each method call is considered as an independent computation.

Definition 2 (block-count(k)). *Given a natural number k, a set of computation paths satisfies the* block-count(k) criterion *if the set includes all finished computation paths which can be built such that the number of times each block is visited within each computation does not exceed the given k.*

Therefore, if we take $k = 1$, this criterion requires that all non-cyclic paths be covered. Note that $k = 1$ will in general not visit all blocks in the CFG, since traversing the loop body of a `while` loop requires $k \geq 2$ in order to obtain a finished path. For the case of structured CFGs, block-count(k) is actually equivalent to loop-count(k'), by simply taking k' to be k-1. We prefer to formulate things in terms of block-count(k) since, formulating loop-count(k) on unstructured CFGs is awkward.

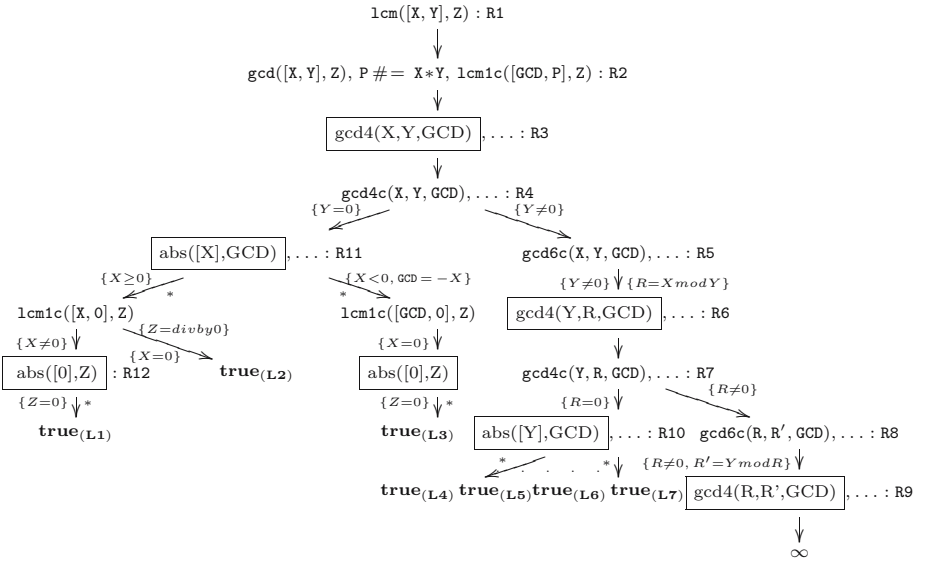


Fig. 3. An evaluation tree for `lcm([X,Y],Z)`

5.2 An Intra-procedural Evaluation Strategy for Block-Count(k)

Fig. 3 depicts (part of) an evaluation tree for `lcm([X,Y],Z)`. Each node in the tree represents a state, which as introduced in Sec. 2, consists of a goal and a store. In order not to clutter the figure, for each state we only show the relevant part of the goal, but not the store. Also, an arc in the tree may involve several reduction steps. In particular, the constraints which precede the leftmost atom (if any) are always processed. Likewise, at least one reduction step is performed on the leftmost atom w.r.t. the program rule whose head unifies with the atom. When more than one step is performed, the arc is labelled with "$*$". Arcs are annotated with the constraints processed at each step. Each branch in the tree represents a *derivation*.

Our aim is to supervise the generation of the evaluation tree so that we generate sufficiently many derivations so as to satisfy the block-count(k) criterion while, at the same time, guaranteeing termination.

Definition 3 (intra-procedural evaluation strategy). *The following two conditions provide an evaluation strategy which ensures block-count(k) in intra-procedural bytecode (i.e., we consider a single CFG for one method):*

(i) annotate every state in the evaluation tree with a multiset, which we refer to as visited*, and which contains the predicates which have been already reduced during the derivation;*

(ii) atoms can only be reduced if there are at most $k-1$ occurrences of the corresponding predicate in visited*.*

It is easy to see that this evaluation strategy is guaranteed to always produce a finite evaluation tree since there is a finite number of rules which can unify with any given atom and therefore non-termination can only be introduced by cycles which are traversed an unbounded number of times. This is clearly avoided by limiting the number of times which resolution can be performed w.r.t. the same predicate.

Example 3. Let us consider the rightmost derivation in Fig. 3, formed by goals R1 to R9. Observe the framed atoms for `gcd4`, the goals R3, R6 and R9 contain an atom for `gcd4` as the leftmost literal. If we take $k = 1$ then resolvent R6 cannot be further reduced since the termination criterion forbids it, as `gcd4` is already once in the multiset of visited predicates. If we take $k = 2$ then R6 can be reduced and the termination criterion is fired at R9, which cannot be further reduced.

5.3 An Inter-procedural Evaluation Strategy Based on Ancestors

The strategy of limiting the number of reductions w.r.t. the same predicate guarantees termination. Furthermore, it also guarantees that the block-count(k) criterion is achieved, but only if the program consists of a single CFG, i.e., at most one method. If the program contains more than one method, as in our example, this evaluation strategy may force termination too early, without achieving block-count(k) coverage.

Example 4. Consider the predicate `abs`. Any successful derivation which does not correspond to exceptions in the bytecode program has to execute this predicate twice, once from the body of method `lcm` and another one from the body of method `gcd`. Therefore, if we take $k = 1$, the leftmost derivation of the tree in Fig. 3 will be stopped at R12, since the atom to be reduced is considered to be a repeated call to predicate `abs`. Thus, the test-case for the successful derivation L1 is not obtained. As a result, our evaluation strategy would not achieve the block-count(k) criterion.

The underlying problem is that we are in an inter-procedural setting, i.e., bytecode programs contain method calls. In this case –meanwhile decompiled versions of bytecode programs without method calls always consist of binary rules– decompiled programs may have rules with several atoms in their body. This is indeed the case for the rule for `lcm` in Ex. 1, which contains an atom for predicate `gcd` and another one for predicate `lcm1c`. Since under the standard left-to-right computation rule, the execution of `gcd` is finished by the time execution reaches `lcm1c` there is no need to take the computation history of `gcd` into account when supervising the execution of `lcm1c`. In our example, the execution of `gcd` often involves an execution of `abs` which is finished by the time the call to `abs` is performed within the execution of `lcm1c`. This phenomenon is well known problem in the context of partial evaluation. There, the notion of *ancestor* has been introduced [5] to allow supervising the execution of conjuncts independently by

only considering visited predicates which are actually ancestors of the current goal. This allows improving accuracy in the specialization.

Given a reduction step where the leftmost atom A is substituted by $B_1, \ldots, B_m$, we say that A is the *parent* of the instance of B_i for $i = 1, \ldots, m$ in the new goal and in each subsequent goal where the instance originating from B_i appears. The *ancestor* relation is the transitive closure of the parent relation. The multiset of ancestors of the atom for `abs` in goal R12 in the SLD tree is {`lcm1c,lcm`}, as `lcm1c` is its parent and `lcm` the parent of its parent. Importantly, `abs` is not in such multiset. Therefore, the leftmost computation in Fig. 3 will proceed upon R12 thus producing the corresponding test-case for every $k \geq 1$. The evaluation strategy proposed below relies on the notion of ancestor sequence.

Definition 4 (inter-procedural evaluation strategy). *The following two conditions provide an evaluation strategy which ensures block-count(k) in inter-procedural bytecode (i.e., we consider several CFGs and methods):*

(i) annotate every atom in the evaluation tree with a multiset which contains its ancestor sequence which we refer to as ancestors*;*
(ii) atoms can only be reduced if there are at most $k - 1$ occurrences of the corresponding predicate in its ancestors*.*

The next section provides practical means to implement this strategy.

6 Test Data Generation by Partial Evaluation

We have seen in Sec. 5 that a central issue when performing symbolic execution for TDG consists in building a finite (possibly unfinished) evaluation tree by using a non-standard execution strategy which ensures both a certain coverage criterion and termination. An important observation is that this is exactly the problem that *unfolding rules*, used in partial evaluators of (C)LP, solve. In essence, partial evaluators are non-standard interpreters which receive a set of partially instantiated atoms and evaluate them as determined by the so-called unfolding rule. Thus, the role of the unfolding rule is to supervise the process of building finite (possibly unfinished) SLD trees for the atoms. This view of TDG as a PE problem has important advantages. First, as we show in Sec. 6.1, we can directly apply existing, powerful, unfolding rules developed in the context of PE. Second, in Sec. 6.2, we show that it is possible to explore additional abilities of partial evaluators in the context of TDG. Interestingly, the generation of a residual program from the evaluation tree returns a program which can be used as a *test-case generator* for obtaining further test-cases.

6.1 Using an Unfolding Rule for Implementing Block-Count(k)

Sophisticated unfolding rules exist which incorporate non-trivial mechanisms to stop the construction of SLD trees. For instance, unfolding rules based on comparable atoms allow expanding derivations as long as no previous *comparable* atom (same predicate symbol) has been already visited. As already discussed, the

use of ancestors [5] can reduce the number of atoms for which the comparability test has to be performed.

In PE terminology, the evaluation strategy outlined in Sec. 5 corresponds to an unfolding rule which allows k comparable atoms in every ancestor sequence. Below, we provide an implementation, predicate `unfold/3`, of such an unfolding rule. The CLP decompiled program is stored as `clause/2` facts. Predicate `unfold/3` receives as input parameters an atom as the initial goal to evaluate, and the value of constant k. The third parameter is used to return the resolvent associated with the corresponding derivation.

```
unfold(A,K,[load_st(St)|Res]) :-
    unf([A],K,[],Res),
    collect_vars([A|Res],Vars),
    save_st(Vars,St).

unf([],_K,_AS,[]).
unf([A|R],K,AncS,Res) :-
    constraint(A),!, call(A),
    unf(R,K,AncS,Res).
unf(['$pop$'|R],K,[_|AncS],Res) :-
    !, unf(R,K,AncS,Res).
unf([A|R],K,AncS,Res) :-
    clause(A,B), functor(A,F,Ar),
    (check(AncS,F,Ar,K) ->
       append(B,['$pop$'|R],NewGoal),
       unf(NewGoal,K,[F/Ar|AncS],Res)
    ;  Res = [A|R]).

check([],_,_,K) :- K > 0.
check([F/Ar|As],F,Ar,K) :- !, K > 1,
    K1 is K - 1, check(As,F,Ar,K1).
check([_|As],F,Ar,K) :- check(As,F,Ar,K).
```

Predicate `unfold/3` first calls `unf/4` to perform the actual unfolding and then, after collecting the variables from the resolvent and the initial atom by means of predicate `collect_vars/2`, it saves the store of constraints in variable `St` so that it is included inside the call `load_st(St)` in the returned resolvent. The reason why we do this will become clear in Sect. 6.2. Let us now explain intuitively the four rules which define predicate `unf/4`. The first one corresponds to having an empty goal, i.e., the end of a successful derivation. The second rule corresponds to the first case in the operational semantics presented in Sec. 2, i.e., when the leftmost literal is a constraint. Note that in CLP there is no need to add an argument for explicitly passing around the store, which is implicitly maintained by the execution engine by simply executing constraints by means of predicate `call/1`. The second case of the operational semantics in Sec. 2, i.e., when the leftmost literal is an atom, corresponds to the fourth rule. Here, on backtracking we look for all rules asserted as `clause/2` facts whose head unifies with the leftmost atom. Note that depending on whether the number of occurrences of comparable atoms in the ancestors sequence is smaller than the given k or not, the derivation continues or it is stopped. The termination check is performed by predicate `check/4`.

In order to keep track of ancestor sequences for every atom, we have adopted the efficient implementation technique, proposed in [25], based on the use of a global *ancestor stack*. Essentially, each time an atom `A` is unfolded using a rule $\mathtt{H} : -\mathtt{B_1}, \ldots, \mathtt{B_n}$, the predicate name of `A`, $pred(A)$, is pushed on the ancestor stack (see third argument in the recursive call). Additionally, a `$pop$` mark is added to the new goal after $\mathtt{B_1}, \ldots, \mathtt{B_n}$ (call to `append/3`) to delimit the scope of the predecessors of `A` such that, once those atoms are evaluated, we find the mark `$pop$` and can remove $pred(A)$ from the ancestor stacks. This way, the

ancestor stack, at each stage of the computation, contains the ancestors of the next atom which will be selected for resolution. If predicate `check/4` detects that the number of occurrences of $pred(A)$ is greater than k, the derivation is stopped and the current goal is returned in `Res`.The third rule of `unf/4` corresponds to the case where the leftmost atom is a pop literal. This indicates that the the execution of the atom which is on top of the ancestor stack has been completed. Hence, this atom is popped from the stack and the pop literal is removed from the goal.

Example 5. The execution of `unfold(lcm([X,Y],Z),2,[_])` builds a finite (and hence unfinished) version of the evaluation tree in Fig. 3. For $k = 2$, the infinite branch is stopped at goal R9, since the ancestor stack at this point is `[gcd6c, gcd4c,gcd4,gcd6c,gcd4c,gcd4,lcm]` and hence it already contains `gcd4` twice. This will make the `check/4` predicate fail and therefore the derivation is stopped. More interestingly, we can generate test-cases, if we consider the following call:

```
findall(([X,Y],Z),unfold([gen_test_data(lcm,[X,Y],[-1000,1000],Z)],2,[_]),TCases).
```

where `generate_test_data` is defined as in Sec. 4. Now, we get on backtracking, concrete values for variables `X`, `Y` and `Z` associated to each finished derivation of the tree.[2] They correspond to test data for the block-count(2) coverage criteria of the bytecode. In particular, we get the following set of test-cases: TCases = [([1,0],0), ([0,0],divby0), ([-1000,0],0), ([0,1],0), ([-1000,1],1000), ([-1000,-1000], 1000),([1,-1],1)] which correspond, respectively, to the leaves labeled as **(L1)**,...,**(L7)** in the evaluation tree of Fig. 3. Essentially, they constitute a particular set of concrete values that traverses all possible paths in the bytecode, including exceptional behaviours, and where the loop body is executed at most once.

The soundness of our approach to TDG amounts to saying that the above implementation, executed on the CLP decompiled program, ensures termination and block-count(k) coverage on the original bytecode.

Proposition 1 (soundness). *Let m be a method with n arguments and BC_m its bytecode instructions. Let* $\mathtt{m}([\mathtt{X_1},\ldots,\mathtt{X_n}],\mathtt{Y})$ *be the corresponding decompiled method and let the CLP block-level decompilation of BC_m be asserted as a set of* `clause/2` *facts. For every positive number k, the set of successful derivations computed by* $\mathtt{unf}(\mathtt{m}([\mathtt{X_1},\ldots,\mathtt{X_n}],\mathtt{Y}),\mathtt{k},[],[],\_)$ *ensures block-count(k) coverage of BC_m.*

Intuitively, the above result follows from the facts that: (1) the decompilation is correct and block-level, hence all traces in the bytecode are derivations in the decompiled program as well as loops in bytecode are cycles in CLP; (2) the unfolding rule computes all feasible paths and traverses cycles at most k times.

[2] We force to consider just finished derivations by providing `[_]` as the obtained resultant.

6.2 Generating Test Data Generators

The final objective of a partial evaluator is to generate optimized *residual* code. In this section, we explore the applications of the code generation phase of partial evaluators in TDG. Let us first intuitively explain how code is generated. Essentially, the residual code is made up by a set of *resultants* or residual rules (i.e., a program), associated to the root-to-leaf derivations of the computed evaluation trees. For instance, consider the rightmost derivation of the tree in Fig. 3, the associated resultant is a rule whose head is the original atom (applying the mgu's to it) and the body is made up by the atoms in the leaf of the derivation. If we ignore the constraints gathered along the derivation (which are encoded in `load_st(S)` as we explain below), we obtain the following resultant:

```
lcm([X,Y],Z) :- load_st(S), gcd4(R,R',GCD), P #= X*Y, lcm1c([GCD,P],Z).
```

The residual program will be (hopefully) executed more efficiently than the original one since those computations that depend only on the static data are performed once and for all at specialization time. Due to the existence of incomplete derivations in evaluation trees, the residual program might not be complete (i.e., it can miss answers w.r.t. the original program). The partial evaluator includes an *abstraction* operator which is encharged of ensuring that the atoms in the leaves of incomplete derivations are "covered" by some previous (partially evaluated) atom and, otherwise, adds the uncovered atoms to the set of atoms to be partially evaluated. For instance, the atoms `gcd4(R,R',GCD)` and `lcm1c([GCD,P],Z)` above are not covered by the single previously evaluated atom `lcm([X,Y],Z)` as they are not instances of it. Therefore, a new unfolding process must be started for each of the two atoms. Hence the process of building evaluation trees by the unfolding operator is iteratively repeated while new atoms are uncovered. Once the final set of trees is obtained, the resultants are generated from their derivations as described above.

Now, we want to explore the issues behind the application of a full partial evaluator, with its code generation phase, for the purpose of TDG. Novel interesting questions arise: (*i*) *what kind of partial evaluator do we need to specialize decompiled CLP programs?;* (*ii*) *what do we get as residual code?;* (*iii*) *what are the applications of such residual code?* Below we try to answer these questions.

As regards question (*i*), we need to extend the mechanisms used in standard PE of logic programming to support constraints. The problem has been already tackled, e.g., by [8] to which we refer for more details. Basically, we need to take care of constraints at three different points: first, during the execution, as already done by `call` within our unfolding rule `unfold/3`; second, during the abstraction process, we can either define an accurate abstraction operator which handles constraints or, as we do below, we can take a simpler approach which safely ignores them; third, during code generation, we aim at generating *constrained* rules which integrate the *store* of constraints associated to their corresponding derivations. To handle the last point, we enhance our schema with the next two basic operations on constraints which are used by `unfold/3` and were left

unexplained in Sec. 6.1. The store is saved and projected by means of predicate `save_st/2`, which given a set of variables in its first argument, saves the current store of the CLP execution, projects it to the given variables and returns the result in its second argument. The store is loaded by means of `load_st/1` which given an explicit store in its argument adds the constraints to the current store. Let us illustrate this process by means of an example.

Example 6. Consider a partial evaluator of CLP which uses as control strategies: predicate `unfold/3` as unfolding rule and a simple abstraction operator based on the combination of the *most specific generalization* and a check of comparable terms (as the unfolding does) to ensure termination. Note that the abstraction operator ignores the constraint store. Given the entry, `gen_test_data(lcm,[X,Y],[-1000,1000],Z)`, we would obtain the following residual code for $k = 2$:

```
gen_test_data(lcm,[1,0],[-1000,1000],0).
gen_test_data(lcm,[0,0],[-1000,1000],
              divby0).
...
gen_test_data(lcm,[X,Y],[-1000,1000],Z) :-
    load_st(S1), gcd4(R,R',GCD),
    P #= X*Y, lcm1c([GCD,P],Z),
    once(labeling([ff],[X,Y])).
```

```
gcd4(R,0,R) :- load_st(S2).
gcd4(R,0,GCD) :- load_st(S3).
gcd4(R,R',GCD) :- load_st(S4),
    gcd4(R',R'',GCD).

lcm1c([GCD,P],Z) :- load_st(S5).
lcm1c([GCD,P],Z) :- load_st(S6).
lcm1c([0,_P],divby0).
```

The residual code for `gen_test_data/4` contains eight rules. The first seven ones are facts corresponding to the seven successful branches (see Fig. 3). Due to space limitations here we only show two of them. Altogether they represent the set of test-cases for the block-count(2) coverage criteria (those in Ex. 6.1). It can be seen that all rules (except the facts[3]) are constrained as they include a residual call to `load_st/1`. The argument of `load_st/1` contains a syntactic representation of the store at the last step of the corresponding derivation. Again, due to space limitations we do not show the stores. As an example, `S1` contains the store associated to the rightmost derivation in the tree of Fig. 3, namely {`X in -1000..1000, Y in (-1000..-1)∨(1..1000), R in (-999.. -1) ∨ (1..999), R' in -998..998, R = X mod Y, R' = Y mod R`}. This store acts as a guard which comprises the constraints which avoid the execution of the paths previously computed to obtain the seven test-cases above.

We can now answer issue (*ii*): it becomes apparent from the example above that we have obtained a program which is a *generator* of test-cases for larger values of k. The execution of the generator will return by backtracking the (infinite) set of values exercising all possible execution paths which traverse blocks more than twice. In essence, our test-case generators are CLP programs whose execution in CLP returns further test-cases on demand for the bytecode under test and without the need of starting the TDG process from scratch.

[3] For the facts, there is no need to consider the store, because a call to `labeling` has removed all variables.

Here, it comes issue (*iii*): Are the above generators useful? How should we use them? In addition to execution (see inherent problems in Sec. 4), we might further partially evaluate them. For instance, we might partially evaluate the above specialized version of `gen_test_data/4` (with the same entry) in order to incrementally generate test-cases for larger values of k. It is interesting to observe that by using $k = 1$ for all atoms different from the initial one, this further specialization will just increment the number of `gen_test_data/4` facts (producing more concrete test-cases) but the rest of the residual program will not change, in fact, there is no need to re-evaluate it later.

7 Conclusions and Related Work

We have proposed a methodology for test data generation of imperative, low-level code by means of existing partial evaluation techniques developed for constraint logic programs. Our approach consist of two separate phases: (1) the compilation of the imperative bytecode to a CLP program and (2) the generation of test-cases from the CLP program. It naturally raises the question whether our approach can be applied to other imperative languages in addition to bytecode. This is interesting as existing approaches for Java [23], and for C [13], struggle for dealing with features like recursion, method calls, dynamic memory, etc. during symbolic execution. We have shown in the paper that these features can be uniformly handled in our approach after the transformation to CLP. In particular, all kinds of loops in the bytecode become uniformly represented by recursive predicates in the CLP program. Also, we have seen that method calls are treated in the same way as calls to blocks. In principle, this transformation can be applied to any language, both to high-level and to low-level bytecode, the latter as we have seen in the paper. In every case, our second phase can be applied to the transformed CLP program.

Another issue is whether the second phase can be useful for test-case generation of CLP programs, which are not necessarily obtained from a decompilation of an imperative code. Let us review existing work for declarative programs. Test data generation has received comparatively less attention than for imperative languages. The majority of existing tools for functional programs are based on black-box testing [6,18]. Test cases for logic programs are obtained in [24] by first computing constraints on the input arguments that correspond to execution paths of logic programs and then solving these constraints to obtain test inputs for the corresponding paths. This corresponds essentially to the naive approach discussed in Sec. 4, which is not sufficient for our purposes as we have seen in the paper. However, in the case of the generation of test data for regular CLP programs, we are interested not only in successful derivations (execution paths), but also in the failing ones. It should be noted that the execution of CLP decompiled programs, in contrast to regular CLP programs, for any actual input values is guaranteed to produce exactly one solution because the operational semantics of bytecode is deterministic. For functional logic languages, specific coverage criteria are defined in [10] which capture the control flow of these languages as well

as new language features are considered, namely laziness. In general, declarative languages pose different problems to testing related to their own execution models –like laziness in functional languages and failing derivations in (C)LP– which need to be captured by appropriate coverage criteria. Having said this, we believe our ideas related to the use of PE techniques to generate test data generators and the use of unfolding rules to supervise the evaluation could be adapted to declarative programs and remains as future work.

Our work is a proof-of-concept that partial evaluation of CLP is a powerful technique for carrying out TDG in imperative low-level languages. To develop our ideas, we have considered a simple imperative bytecode language and left out object-oriented features which require a further study. Also, our language is restricted to integer numbers and the extension to deal with real numbers is subject of future work. We also plan to carry out an experimental evaluation by transforming Java bytecode programs from existing test suites to CLP programs and then trying to obtain useful test-cases. When considering realistic programs with object-oriented features and real numbers, we will surely face additional difficulties. One of the main practical issues is related to the scalability of our approach. An important threaten to scalability in TDG is the so-called infeasibility problem [27]. It happens in approaches that do not handle constraints along the construction of execution paths but rather perform two independent phases (1) path selection and 2) constraint solving). As our approach integrates both parts in a single phase, we do not expect scalability limitations in this regard. Also, a challenging problem is to obtain a decompilation which achieves a manageable representation of the heap. This will be necessary to obtain test-cases which involve data for objects stored in the heap. For the practical assessment, we also plan to extend our technique to include further coverage criteria. We want to consider other classes of coverage criteria which, for instance, generate test-cases which cover a certain statement in the program.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers - Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
3. Albert, E., Gómez-Zamalloa, M., Hubert, L., Puebla, G.: Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 124–139. Springer, Heidelberg (2006)

4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS, vol. 4334. Springer, Heidelberg (2007)
5. Bruynooghe, M., De Schreye, D., Martens, B.: A General Criterion for Avoiding Infinite Unfolding during Partial Deduction. New Generation Computing 1(11), 47–79 (1992)
6. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: ICFP, pp. 268–279 (2000)
7. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Software Eng. 2(3), 215–222 (1976)
8. Craig, S.-J., Leuschel, M.: A compiler generator for constraint logic programs. In: Ershov Memorial Conference, pp. 148–161 (2003)
9. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Trans. Softw. Eng. Methodol. 5(1), 63–86 (1996)
10. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: PPDP, pp. 63–74 (2007)
11. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Systems, Computers, Controls 2(5), 45–50 (1971)
12. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Modular Decompilation of Low-Level Code by Partial Evaluation. In: 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), pp. 239–248. IEEE Computer Society, Los Alamitos (2008)
13. Gotlieb, A., Botella, B., Rueher, M.: A clp framework for computing structural test data. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS, vol. 1861, pp. 399–413. Springer, Heidelberg (2000)
14. Gupta, N., Mathur, A.P., Soffa, M.L.: Generating test data for branch coverage. In: Automated Software Engineering, pp. 219–228 (2000)
15. Henriksen, K.S., Gallagher, J.P.: Abstract interpretation of pic programs through logic programming. In: SCAM 2006: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, pp. 184–196. IEEE Computer Society, Los Alamitos (2006)
16. Howden, W.E.: Symbolic testing and the dissect symbolic evaluation system. IEEE Transactions on Software Engineering 3(4), 266–278 (1977)
17. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
18. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: Generic automated software testing. In: IFL, pp. 84–100 (2002)
19. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading (1996)
20. Marriot, K., Stuckey, P.: Programming with Constraints: An Introduction. MIT Press, Cambridge (1998)
21. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915. Springer, Heidelberg (2008)
22. Meudec, C.: Atgen: Automatic test data generation using constraint logic programming and symbolic execution. Softw. Test., Verif. Reliab. 11(2), 81–96 (2001)
23. Müller, R.A., Lembeck, C., Kuchen, H.: A symbolic java virtual machine for test case generation. In: IASTED Conf. on Software Engineering, pp. 365–371 (2004)
24. Mweze, N., Vanhoof, W.: Automatic generation of test inputs for mercury programs. In: Pre-proceedings of LOPSTR 2006 (July 2006) (extended abstract)

25. Puebla, G., Albert, E., Hermenegildo, M.: Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 149–165. Springer, Heidelberg (2005)
26. Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. SICStus Prolog 3.8 User's Manual, 3.8 edition (October 1999), `http://www.sics.se/sicstus/`
27. Zhu, H., Patrick, A., Hall, V., John, H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. 29(4), 366–427 (1997)

On the Generation of Test Data for Prolog by Partial Evaluation

Miguel Gómez-Zamalloa[1], Elvira Albert[1], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. In recent work, we have proposed an approach to Test Data Generation (TDG) of imperative bytecode by *partial evaluation* (PE) of CLP which consists in two phases: (1) the bytecode program is first transformed into an equivalent CLP program by means of interpretive compilation by PE, (2) a second PE is performed in order to supervise the generation of test-cases by execution of the CLP decompiled program. The main advantages of TDG by PE include flexibility to handle new coverage criteria, the possibility to obtain test-case generators and its simplicity to be implemented. The approach in principle can be directly applied for TDG of any imperative language. However, when one tries to apply it to a declarative language like Prolog, we have found as a main difficulty the generation of test-cases which cover the more complex control flow of Prolog. Essentially, the problem is that an intrinsic feature of PE is that it only computes non-failing derivations while in TDG for Prolog it is essential to generate test-cases associated to failing computations. Basically, we propose to transform the original Prolog program into an equivalent Prolog program with *explicit failure* by partially evaluating a Prolog interpreter which captures failing derivations w.r.t. the input program. Another issue that we discuss in the paper is that, while in the case of bytecode the underlying constraint domain only manipulates integers, in Prolog it should properly handle the symbolic data manipulated by the program. The resulting scheme is of interest for bringing the advantages which are inherent in TDG by PE to the field of logic programming.

1 Introduction

Test data generation (TDG) aims at automatically generating test-cases for interesting test *coverage criteria*. The coverage criteria measure how well the program is exercised by a test suite. Examples of coverage criteria are: *statement coverage* which requires that each line of the code is executed; *path coverage* which requires that every possible trace through a given part of the code is executed; etc. There are a wide variety of approaches to TDG (see [22] for a survey). Our work focuses on *glass-box* testing, where test-cases are obtained from the concrete program in contrast to *black-box* testing, where they are deduced from a specification of the program. Also, our focus is on *static* testing, where we assume no knowledge about the input data, in contrast to *dynamic* approaches [6] which execute the program to be tested for concrete input values.

The standard approach to generating test-cases statically is to perform a *symbolic* execution of the program [18,14,11], where the contents of variables are expressions rather than concrete values. The symbolic execution produces a system of *constraints* consisting of the conditions to execute the different paths. This happens, for instance, in branching instructions, like if-then-else, where we might want to generate test-cases for the two alternative branches and hence accumulate the conditions for each path as constraints. The symbolic execution approach is usually combined with the use of *constraint solvers* in order to: handle the constraints systems by solving the feasibility of paths and, afterwards, to instantiate the input variables.

TDG for declarative languages has received comparatively less attention than for imperative languages. In general, declarative languages pose different problems to testing related to their own execution models, like laziness in functional programming (FP) and failing derivations in constraint logic programming (CLP). The majority of existing tools for FP are based on black-box testing (see e.g. [4]). An exception is [7] where a glass-box testing approach is proposed to generate test-cases for Curry. In the case of CLP, test-cases are obtained for Prolog in [16,3,21]; and very recently for Mercury in [5]. Basically the test-cases are obtained by first computing constraints on the input arguments that correspond to execution paths of logic programs and then solving these constraints to obtain test inputs for such paths.

In recent work [2], we have proposed to employ existing *partial evaluation* (PE) techniques developed for CLP in order to automatically generate *test-case generators* for glass-box testing of bytecode. PE [13] is an automatic program transformation technique which has been traditionally used to specialise programs w.r.t. a known part of its input data and, as Futamura predicted, can also be used to compile programs in a (source) language to another (object) language (see [8]). The approach to TDG by PE of [2] consists of two independent CLP PE phases. (1) First, the bytecode is transformed into an equivalent (decompiled) CLP program by specialising a bytecode interpreter by means of existing PE techniques. (2) A second PE is performed in order to supervise the generation of test-cases by execution of the CLP decompiled program. Interestingly, it is possible to employ control strategies previously defined in the context of CLP PE in order to capture *coverage criteria* for glass-box testing of bytecode. A unique feature of this approach is that, this second PE phase allows generating not only test-cases but also test-case *generators*. Another important advantage is that, in contrast to previous work to TDG of bytecode, it does not require devising a dedicated symbolic virtual machine.

In this work, we study the application of the above approach to TDG by means of PE to the Prolog language. Compared to TDG of an imperative language [2], dealing with Prolog brings in as the main difficulty to generate test-cases associated to failing computations. This happens because an intrinsic feature of PE is that it only produces results associated to the *non-failing* derivations. While this is what we need for TDG of an imperative language (like bytecode above), we now want to capture non-failing derivations in Prolog and

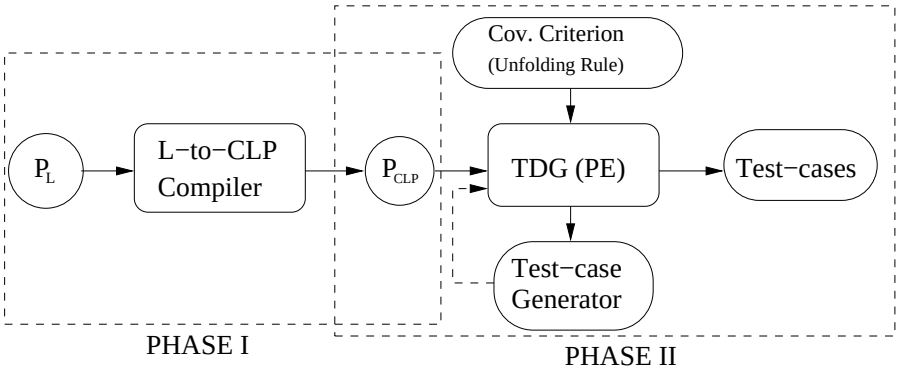


Fig. 1. General scheme of TDG by Partial Evaluation of CLP

still rely on a standard partial evaluator. Our proposal is to transform the original Prolog program into an equivalent Prolog program with explicit failure by partially evaluating a Prolog interpreter which captures failing derivations w.r.t. the input program. This transformation is done in the phase (1) above. As another difference, in the case of bytecode, the underlying constraint domain only manipulates integers. However, the above phase (2) should properly handle the data manipulated by the program in the case of Prolog. Compared to existing approaches to TDG of Prolog [3,16], our approach basically is of interest for bringing the advantages which are inherent in TDG by PE to the field of Prolog:

(i) It is *more powerful* in that we can produce test-case generators which are CLP programs whose execution in CLP returns further test-cases on demand without the need to start the TDG process from scratch;
(ii) It is more *flexible*, as different coverage criteria can be easily incorporated to our framework just by adding the appropriate local control to the partial evaluator.
(iii) It is *simpler* to implement compared to the development of a dedicated test-case generator, as long as a CLP partial evaluator is available.

The rest of the paper is organized as follows. In the next section, we give some basics on PE of logic programs and describe in detail the approach to TDG by PE proposed in [2]. Sect. 3 discusses some fundamental issues like the Prolog control-flow and the notion of computation path. Then, Sect. 4 describes the program transformation to make failure explicit, Sect. 5 outlines existing methods to properly handle symbolic data during the TDG phase, and finally Sect. 6 concludes and discusses some ideas for future work.

2 Basics of TDG by Partial Evaluation

In this section we recall the basics of partial evaluation of logic programming and summarize the general approach of relying on partial evaluation of CLP for TDG of an imperative language, as proposed in [2].

2.1 Partial Evaluation and its Application to Compilation

We assume familiarity with basic notions of logic programming and partial evaluation (see e.g. [9]). Partial evaluation is a semantics-based program transformation technique which specialises a program w.r.t. given input data, hence, it is often called *program specialisation*. Essentially, partial evaluators are non-standard interpreters which evaluate goals as long as termination is guaranteed and specialisation is considered profitable. In logic programming, the underlying technique is to construct (possibly) *incomplete* SLD trees for the set of atoms to be specialised. In an incomplete tree, it is possible to choose *not* to further unfold a goal. Therefore, the tree may contain three kinds of leaves: failure nodes, success nodes (which contain the empty goal), and non-empty goals which are not further unfolded. The latter are required in order to guarantee termination of the partial evaluation process, since the SLD being built may be infinite. Even if the SLD trees for fully instantiated initial atoms (as regards the *input* arguments) are finite, the SLD trees produced for partially instantiated initial atoms may be infinite. This is because the SLD for partially instantiated atoms can have (infinitely many) more branches than the actual SLD tree at run-time.

The role of the *local control* is to determine how to construct the (incomplete) SLD trees. In particular, the *unfolding rule* decides, for each resolvent, whether to stop unfolding or to continue unfolding it and, if so, which atom to select from the resolvent. On the other hand, partial evaluators need to compute SLD-trees for a number of atoms in order to ensure that all atoms which appear in non-failing leaves of incomplete SLD trees are "covered" by the root of some tree (this is known as the closedness condition of partial evaluation [9]). The role of the *global control* is to ensure that we do not try to compute SLD trees for an infinite number of atoms. The usual way of achieving this is by applying an *abstraction operator* which performs "generalizations" on the atoms for which SLD trees are to be built. The global control returns a set of atoms T. Finally, the partial evaluation can then be systematically extracted from the set T (see [9] for details).

Traditionally, there have been two different approaches regarding the way in which control decisions are taken, *on-line* and *off-line* approaches. In *online* PE, all control decisions are dynamically taken during the specialisation phase. In *offline* PE, a set of previously computed annotations (often manually provided) gives information to the control operators to decide, 1) when to stop unfolding (*memoise*) in the local control, and 2) how to perform generalizations in the global control.

The development of PE techniques has allowed the so-called "interpretative approach" to compilation which consists in specialising an interpreter w.r.t. a fixed object code. Interpretive compilation was proposed in Futamura's seminal work [8], whereby compilation of a program P written in a (*source*) programming language L_S into another (*object*) programming language L_O is achieved by partially evaluating an interpreter for L_S written in L_O w.r.t. P. The advantages of interpretive (de-)compilation w.r.t. dedicated (de-)compilers are well-known and discussed in the PE literature (see, e.g., [1]). Very briefly, they include: *flexibility*,

it is easier to modify the interpreter in order to tune the decompilation (e.g., observe new properties of interest); *easier to trust*, it is more difficult to prove that ad-hoc decompilers preserve the program semantics; *easier to maintain*, new changes in the language semantics can be easily reflected in the interpreter.

2.2 A General Scheme to TDG of Imperative Languages by PE

In recent work, we have proposed an approach to Test Data Generation (TDG) by PE of CLP [2] and used it for TDG of bytecode. The approach is generic in that the same techniques can be applied to TDG other both low and high-level imperative languages. In Figure 1 we overview the main two phases of this technique. In **Phase I**, the input program written in some (imperative) language L is compiled into an equivalent CLP program P_{CLP}. This compilation can be achieved by means of an ad-hoc decompiler (e.g., an ad-hoc decompiler of bytecode to Prolog [17]) or, more interestingly, can be achieved automatically by relying on the first Futamura projection by means of PE for logic programs as explained above (e.g., [12,1,10]).

Now, the aim of **Phase II** is to generate test-cases which traverse as many different execution paths of P_L as possible, according to a given coverage criteria. From this perspective, different test data will correspond to different execution paths. With this aim, rather than executing the program starting from different input values, the standard approach consists in performing *symbolic execution* such that a single symbolic run captures the behavior of (infinitely) many input values. The central idea in symbolic execution is to use constraint variables instead of actual input values and to capture the effects of computation using constraints. Hence, the compilation from L to CLP allows us to use the standard CLP execution mechanism to carry out this phase. In particular, by running the P_{CLP} program without input values, each successful execution corresponds to a different computation path in P_L.

Rather than relying on the standard execution mechanism, we have proposed in [2] to use PE of CLP to carry out **Phase II**. Essentially, we can rely on a CLP partial evaluator which is able to solve the constraint system, in much the same way as a symbolic abstract machine would do. Note that performing symbolic execution for TDG consists in building a finite (possibly unfinished) evaluation tree by using a non-standard execution strategy which ensures both a certain coverage criterion and termination. This is exactly the problem that *unfolding rules*, used in partial evaluators of (C)LP, solve. In essence, partial evaluators are non-standard interpreters which receive a set of partially instantiated atoms and evaluate them as determined by the so-called unfolding rule. Thus, the role of the unfolding rule is to supervise the process of building finite (possibly unfinished) SLD trees for the atoms. This view of TDG as a PE problem has important advantages. First, we can directly apply existing, powerful, unfolding rules developed in the context of PE. Second, it is possible to explore additional abilities of partial evaluators in the context of TDG. In particular, the generation of a residual program from the evaluation tree returns a program which can be used as a *test-case generator*, i.e., a CLP program whose execution in CLP

returns further test-cases on demand without the need to start the TDG process from scratch. In the rest of the paper, we study the application of this general approach to TDG of Prolog programs.

3 Computation Paths for Test Data Generation of Prolog

As we have already mentioned, test data generation is about producing test-cases which traverse as many different execution paths as possible. From this perspective, different test data should correspond to different execution paths. Thus, a main concern is to specify the computation paths for which we will produce test-cases. This requires first to determine the control flow of the considered language. In this section, we aim at defining the control flow of Prolog programs that we will use for TDG. Test data will be generated for the computation paths in the control flow.

3.1 The Control Flow of Prolog

As usual a Prolog program consists of a set of predicates, where each predicate is defined as a sequence of clauses of the form H :- $B_1, \ldots, B_m$ with $m \geq 0$. A predicate is univocally determined by its *predicate signature* p/n, being p the name of the predicate and n its arity. Throughout the rest of the paper we will consider Prolog programs with the following features:

- Rules are normalized, i.e., arguments in the head of the rule are distinct variables. The corresponding bindings will appear explicitly in the body as unifications.
- Atoms appearing in the bodies of rules can be: unifications (considered as builtins), calls to defined predicates, term checking builtins (`==/2`, `\==/2`, etc), and arithmetic builtins (`is/2`, `</2`, `=</2`, etc). Other typical Prolog builtins like `fail/0`, `!/0`, `if/3`, etc, have been deliberately left out to simplify the presentation.
- All predicates must be moded and well-typed. We will assume the existence of a "`:- pred`" declaration associated with each predicate specifying the type expected for each argument (see as example the declarations in Fig. 2). Note that this assumption is sensible in the context of TDG (as the aim is the automatic generation of test *input*). Also, it should not be a limitation as analyses that can automatically infer this information exist.

The control flow in Prolog programs is significantly more complex than in traditional imperative languages. The declarative semantics of Prolog implies some additional features like: 1) several forms of backtracking, induced by the failure of a sub-goal, or by non-deterministic predicates; or 2) forced control flow change by the predicate "cut". Traditionally, control-flow graphs (CFGs for short) are used to statically represent the control-flow of programs. Typically, in a CFG, nodes are blocks containing a set of sequential instructions, and edges represent the flows that the program can follow w.r.t. the semantics of the corresponding

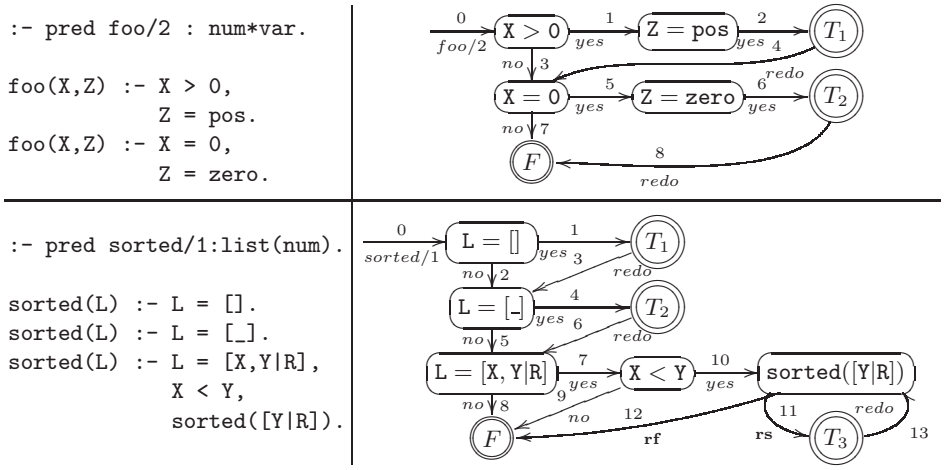


Fig. 2. Working example. Prolog code and CFGs.

programming language. In the literature, CFGs for Prolog (and Mercury) have been used for the aim of TDG in [16,21] ([5] for Mercury). In particular, CFGs determine the computation paths for which test-cases will be produced. Our framework relies on the CFGs of [16,21] which are known as *p-flowgraph*'s.[3] As will be explained later, there are some differences between these CFGs and the ones in [5] which lead to different test-cases.

Figure 2 depicts the Prolog code together with the corresponding CFGs for predicates `foo/2` and `sorted/1`. Predicate `foo/2`, given a number in its first argument, returns, in the second one, the value `pos` if the number is positive and `zero` if it is zero. If the number is negative, it just fails. Predicate `sorted/1`, given a list of numbers, checks whether the list is strictly sorted, in that case it succeeds, otherwise it fails. The CFGs contain the following nodes:

- a non-terminal node associated to each atom in the body of each clause,
- a set of terminal nodes "T$_i$" representing the success of the i-th clause, and
- the terminal node "F" to represent failure.

As regards edges, in principle all non-terminal nodes have two output flows, corresponding to the cases where the builtin or predicate call succeeds or fails respectively. They are labeled as "yes" or "no" for builtins (including unifications), and as "**rs**" (*return-after-success*) or "**rf**" (*return-after-failure*) for predicate calls. There is an exception in the case of unifications where one of the arguments is a variable, in which case the unification cannot fail. This can be known statically by using the mode information. See for example nodes "`Z=pos`"

[3] The difference with the CFGs in [16,21] is that they consider one additional node per clause to explicitly represent the unification of the head of the rule. This is not needed in our case since predicates are normalised.

and "`Z=zero`" in the `foo/2` CFG. Both "yes" and "**rs**" edges point to the node representing the next atom in the clause or to the corresponding "T_i" node if the atom is the last one. Finally, each "T_i" node has an output edge labeled as "redo" to represent the case in which the predicate is asked for more solutions. All "no", "**rf**" and "redo" edges point either to the node corresponding to the first previous non-deterministic call in the same clause, or the first node of the following clause, or the "F" node if no node meets the above conditions. See as an example the "**rs**" and "**rf**" edges from the non-terminal node for `sorted([Y|R])`.

3.2 Generating Test Data for Computation Paths

In order to define the computation paths determined by the CFGs, every edge in every CFG is labeled with a unique natural number. An special edge labeled with "0" and p/n represents the entry of predicate p/n.

Definition 1 (Computation sub-path). *Given the CFG for predicate P, a* computation sub-path *is a sequence of numeric labels (natural numbers) $\langle l_1, \ldots, l_n \rangle$ s.t.:*

- *l_1 corresponds to either an entry, an "**rs**", an "**rf**" or a "redo" edge,*
- *l_n leads to a terminal node or to a predicate call, and*
- *for all consecutive labels l_i, l_j, there exists a node corresponding to a builtin in the CFG of P, for which l_i is an input flow and l_j is an output flow.*

Definition 2 (Computation path). *Given the CFGs corresponding to the set of predicates defining a program, a* computation path *(CP for short) for predicate p is a concatenation $sp_1 \cdots sp_m$ ($m \geq 1$) of computation sub-paths such that:*

- *First label in sp_1 is either 0, in which case we say it is a* full *CP, or corresponds to a "redo" edge, in which case we say it is a* partial *CP (PCP for short).*
- *Last label in sp_m leads to a terminal node in the CFG of p. If it is a T node the CP is said to be* successful *otherwise it is called* failing.
- *For all sp_k whose last label leads to a node corresponding to a predicate call, $cp = sp_{k+1} \cdots sp_j$, $j > k$ is a CP for the called predicate, and:*
  - *if cp is successful then the first label in sp_{j+1} corresponds to an "**rs**" edge,*
  - *otherwise (cp is failing), it corresponds to an **rf** edge.*
- *For all sp_k whose first label corresponds to a "redo" edge flowing from a "T_a" node in the CFG of predicate q, $\exists sp_j$, $j < k$, whose first label corresponds either to an entry edge or to a "redo" edge flowing from "T_b", $b < a$, of the CFG of q.*

If a CP contains at least one label corresponding to a "redo" flow, then the CP is said to be an after-retry *CP. The rest of the CPs are* first-try *CPs.*

For example in `foo/2`, p_1=$\langle 0,1,2\rangle$ and p_2=$\langle 0,3,5,6\rangle$ are first-try successful CPs; p_3=$\langle 0,3,7\rangle$ is a first-try failing branch; p_4=$\langle 0,1,2\rangle \cdot \langle 4,5,6\rangle$ is an after-retry successful CP (although this one is unfeasible as $X > 0$ and $X = 0$ are disjoint conditions), and p_5=$\langle 0,1,2\rangle \cdot \langle 4,7\rangle$ is an after-retry failing branch. In `sorted/1`, p_6=$\langle 0,2,5,7,10\rangle \cdot \langle 0,2,4\rangle \cdot \langle 11\rangle$ is a first-try successful CP and p_7=$\langle 0,2,5,7,10\rangle \cdot \langle 0,2,5,7,9\rangle \cdot \langle 12\rangle$ is a first-try failing CP. It is interesting to observe the correspondence between the CPs and the test data that make the program traverse them. In `foo/2`, p_1 is followed by goal `foo(1,Z)`, p_2 by goal `foo(0,Z)`, p_3 by `foo(-1,Z)`, p_4 is an unfeasible path, and p_5 is followed by `foo(0,Z)` when we ask for more solutions. As regards `sorted/1`, p_6 is followed by the goal `sorted([0,1])` and p_7 by `sorted([0,1,0])`. As we will see in Sect. 5, these will become part of the test-cases that we automatically infer.

A key feature of our CFGs is that they make explicit the fact that after failing with a clause the computation has to re-try with the following clause, unless a non-deterministic call is left behind. E.g., in `foo/2` the CFG makes explicit that the only way to get a first-try failing branch is through the CP $\langle 0,3,7\rangle$, hence traversing, and failing in, both conditions $X > 0$ and $X = 0$. Therefore, a test data to obtain such a behavior will be a negative number for argument X. Other approaches, like the one in [5], do not handle flows after failure in the same way. In fact, in [5], edge "3" in `foo/2` goes directly to node "F". It is not clear if these approaches are able to obtain such a test data. As another difference with previous approaches to TDG of Prolog, we want to highlight that we use CFGs just to reason about the program transformation that will be presented in the following section and, in particular, to clarify which features we want to capture. However, in previous approaches, test-cases are deduced directly from the CFGs.

4 A Program Transformation to Make Failure Explicit

As we outlined in Sect. 1, an intrinsic feature of the second phase of our approach is that it can only produce results associated to non-failing derivations. This is the main reason why the general approach to TDG by PE sketched in Sect. 2 is directly applicable only to TDG of imperative languages. To enable its application to Prolog, we propose a program transformation which makes failure explicit in the Prolog program. The specialisation of meta-programs has been proved to have a large number of interesting applications[9]. Futamura projection's to derive compiled code, compilers and compiler generators fall into this category. The specialization of meta-interpreters for non-standard computation rules has also been studied. Furthermore, language extensions and enhancements can be easily expressed as meta-interpreters which perform additional operations to the standard computation. In short, program specialisation offers a general compilation technique for the wide variety of procedural interpretations of logic programs. Among them, we propose to carry out our transformation which makes failure in logic programs explicit by partially evaluating a Prolog meta-interpreter which captures failing derivations w.r.t. the original program. First, in Sect. 4.1 we describe such a meta-interpreter emphasizing the Prolog control features which

we want to capture. Then, Sect. 4.2 describes the control strategies which have to be used in PE in order to produce an effective transformation.

4.1 A Prolog Meta-Interpreter to Capture Failure

Given a Prolog program and given a goal, our aim is to define an interpreter in which the computation of the program and goal produces the same results as the ones obtained by using the standard Prolog computation but with the difference that failure is never reported. Instead, an additional argument *Answer* will be bound to the value "yes", if the computation corresponds to a successful derivation, and to "no" if it corresponds to a failing derivation. Predicate `solve/4` is the main predicate of our meta-interpreter whose first and second arguments are the predicate signature and arguments of the goal to be executed; and its third argument is the answer; by now we ignore the last argument. For instance, the call `solve(foo/2,[0,Z],Answer,_)` succeeds with `Z` = `zero` and `Answer` = `yes`, and `solve(foo/2,[-1,Z],Answer,_)` also succeeds, but with `Answer` = `no`. The interpreter has to handle the following issues:

1. The Prolog *backtracking* mechanism has to be explicitly implemented. To this aim, a stack of *choice points* is carried along during the computation so that:
   - if the derivation fails: (1) when the stack is empty, it ends up with success and returns the value "no", (2) otherwise, the computation is resumed from the last choice point, if any;
   - if it successfully ends: (1) when the stack is empty, the computation finishes with answer "yes", (2) otherwise, the computation is resumed from the last choice point.
2. When backtracking occurs, all variable bindings, between the current point and the choice point to resume from, have to be undone.
3. The interpreter has to be implemented in a *big-step* fashion. This is a requirement for obtaining an effective decompilation. More details are given in Sect. 4.2.

Figure 3 shows an implementation of a meta-interpreter which handles the above issues. The fourth argument of the main predicate `solve/4`, named `TNCPS`, contains upon success the total number of choice points not yet considered, whose role will be explained later. The interpreter assumes that the program is represented as a set of `pred/2` and `clause/3` facts. There is a `pred/2` fact per predicate providing its predicate signature, number of clauses and mode information; and a `clause/3` fact per clause providing the actual code and clause identifier. Predicate `solve/4` basically builds an initial state on `S0`, by calling `build_s0/4`, and then delegates on `exec/3` to obtain the final state `Sf` of the computation. The output information, `OutVs`, is taken from `Sf`. The state carried along is of the form `st(PP,G,CPs,OutVs,Ans,NCPs)`, where `PP` is the current program point, `G` the current goal, `CPs` is the stack of choice points (list of program points), `OutVs` the list of variables in `G` corresponding to the output

```
solve(P/Ar,Args,Answer,TNCPs) :-
   pred(P/Ar,_),
   build_s0(P/Ar,Args,S0,OutVs),
   exec(Args,S0,Sf),
   Sf = st(_,_,_,OutVs',Answer,TNCPs/_),
   OutVs' = OutVs.

exec(_,S,Sf) :-
   S = st(_,[],[],OutVs,yes,NCPs),
   Sf = st(_,_,_,OutVs,yes,NCPs).
exec(_,S,Sf) :-
   S = st(_,[],[_|_],OutVs,yes,NCPs),
   Sf = st(_,_,_,OutVs,yes,NCPs).
exec(_,S,Sf) :-
   S = st(_,_,[],OutVs,no,TNCPs/0),
   Sf = st(_,_,_,OutVs,no,TNCPs/0).
exec(Args,S,Sf) :-
   S = st(_,[],[CP|CPs],_,yes,TNCPs/0),
   build_retry_state(Args,CP,CPs,TNCPs,S'),
   exec(Args,S',Sf).
exec(Args,S,Sf) :-
   S = st(_,_,[CP|CPs],_,no,TNCPs/0),
   build_retry_state(Args,CP,CPs,TNCPs,S'),
   exec(Args,S',Sf).

exec(Args,S,Sf) :-
   S = st(PP,[A|As],CPs,OutVs,yes,TNCPs/ENCPs),
   PP = pp(P/Ar,ClId,Pt),
   internal(A),
   functor(A,A_f,A_ar),
   A =..[A_f|A_args],
   next(Pt,Pt'),
   solve(A_f/A_ar,A_args,Ans,ENCPs'),
   TNCPs' is TNCPs + ENCPs',
   ENCPs'' is ENCPs + ENCPs',
   PP' = pp(P/Ar,ClId,Pt'),
   S' = st(PP',As,CPs,OutVs,Ans,TNCPs'/ENCPs''),
   exec(Args,S',Sf).
exec(Args,S,Sf) :-
   S = st(PP,[A|As],CPs,OutVs,yes,NCPs),
   PP = pp(P/Ar,ClId,Pt),
   builtin(A),
   next(Pt,Pt'),
   run_builtin(PP,A,Ans),
   PP' = pp(P/Ar,ClId,Pt'),
   S' = st(PP',As,CPs,OutVs,Ans,NCPs),
   exec(Args,S',Sf).
```

Fig. 3. Code of Prolog meta-interpreter to capture failure

parameters of the original goal, `Ans` the current answer ("yes" or "no") and `NCPs` the number of choice points left behind. A program point is of the form `pp(P/Ar,ClId,Pt)`, where `P/Ar`, `ClId` and `Pt` are the predicate signature, the clause identifier and the program point of the clause at hand. Predicate `exec/3` implements the main loop of the interpreter. Given the current state in its second argument it produces the final state of the computation in the third one. It is defined by the seven clauses which are applied in they following situations:

$1^{st}cl$. *The current goal is empty, the answer "yes" and there are no pending choice points.* Then, the computation finishes with answer "yes". The current answer is actually used as a flag to indicate whether the previous step in the computation succeeded or failed (see the last two `exec/3` clauses).

$2^{nd}cl$. *As* $1^{st}cl$. *but having at least one choice point.* This clause represents the solution in which the computation ends. The 4^{th} clause takes the other alternatives.

$3^{rd}cl$. *The previous step failed and there are no pending choice points.* Then, the computation ends with answer "no".

$4^{th}cl$. *The current goal is empty, the answer "yes" and there is at least one pending choice point.* This is the same situation as in the 2^{nd} clause, however in this case the alternative of resuming from the last choice point is taken. The corresponding state `S'` is built by means of `build_retry_state/5` and the computation is resumed from `S'` by recursively calling `exec/3`.

$5^{th}cl$. *The previous step failed and there is at least one pending choice point.* Then, the computation is resumed from the last choice point in the same way as in the previous clause.

$6^{th}cl$. *The first atom to be solved is user-defined.* A call to `solve/4` handles the atom, and the computation proceeds with the next program point of the same clause which was the current one before calling `solve/4`. This way of solving a predicate call makes the interpreter *big-step* (issue (3) above).

$7^{th}cl$. *The first atom to be solved is a builtin.* Then, `run_builtin/3` produces the corresponding answer, and the computation proceeds with the following program point. An interesting observation (also applicable for the previous clause) is that the answer obtained from `run_builtin/3` (or `solve/4`) is now set up as the answer of the next state. This will make the computation go through the 3^{rd} or 5^{th} clauses in the following step, if the obtained answer was "no".

The correspondence between these clauses and the flows in the CFGs is as follows: clauses 1^{st}, 2^{nd} and 4^{th} represent the output edges from every "T" node. Clause 3^{rd} represents the "no" edges to "F" nodes and 5^{th} the "no" edges to non-terminal nodes. Finally clauses 6^{th} and 7^{th} represents the execution of builtins and predicate calls in non-terminal nodes and their corresponding "yes" edges.

Let us now explain how the interpreter handles the above three issues. To handle (1), a stack of choice points is carried along within the state, initialised to contain all initial program points of each clause defining the predicate to be solved, except for the first one. E.g., the initial stack of choice points for `sorted/1` is `[pp(sorted/1,2,1),pp(sorted/1,3,1)]`. How this stack is used to perform the backtracking is already explained in the description of the 4^{th} and 5^{th} `exec/3` clauses above. As regards issue (2), a quite simple way to implement this in Prolog is to produce the necessary fresh variables every time the computation is resumed. This is done inside `build_retry_state/5`. The corresponding unification to link the fresh variables with the original goal variables is made at the end (see last line of `solve/4`). This is the reason why 1) the list of the actual variables used in the current goal needs to be carried along within the state; and 2) the original arguments are carried along as the first argument of `exec/3`, as the original ground arguments provided, have to be used when resuming from a choice point.

Finally, it is worth mentioning that `solve/4` does not return the actual stack of choice points but only the number of them. This means that during a computation the interpreter only considers choice points of the predicate being solved. The question is then, how can the interpreter backtrack to the last choice point, including those induced by other computations of `solve/4`? E.g., how can the interpreter follow edge "13" in the CFG of `sorted/1`? The interpreter performs the backtracking in the following way: 1) The total number of choice points left behind, `TNCPs`, is carried along within the state and finally returned in the last argument of `solve/4`. 2) The number of choice points corresponding to invoked predicates, `ENCPs`, is also carried along. It is updated right after the call to `solve/4` in the 6^{th} clause of `exec/3`. Both numbers are stored in the last argument of the state as `TNCPs/ENCPs`. 3) Execution is resumed from choice points of the current predicate only if `ENCPs` $= 0$, as it can be seen in the 4^{th} and 5^{th} clauses. Otherwise, the computation just fails and Prolog's backtracking

mechanism is used to ask the last invoked predicate for more solutions. This indeed means that the non-determinism of the program is still implicit.

4.2 Controlling Partial Evaluation

The specialisation of interpreters has been studied in many different contexts, see e.g. [9,10,19]. Very recently, [10] proposed control strategies to successfully specialise low-level code interpreters w.r.t. non trivial programs. Here we demonstrate how such guidelines can be, and should be, used in the specialisation of non-trivial Prolog meta-interpreters. They include:

1. *Big-step* interpreter. This solves the problem of handling recursion (see [10]) and enables a compositional specialisation w.r.t. the program procedures (or predicates). Note that an effective treatment of recursion is specially important in Prolog programs where recursion is heavily used.
2. *Optimality* issues. Optimality must ensure that: a) the code to be transformed is traversed exactly once, and b) residual code is emitted once in the transformed program. To achieve optimality, during unfolding, all atoms corresponding with *divergence* or *convergence points* in the CFG of the program to be transformed, has to be *memoised* (see Sect. 2.1). A divergence (convergence) point is a program point from (to) which two or more flows originate (converge).

We already explained that the interpreter in Fig. 3 is big-step. As regards optimality, by looking at the CFGs of Fig. 2, we can observe: 1) all program points are divergence points except those corresponding with unifications in which one argument is a variable, and 2) the first program point of every clause, except for the one of the first clause, is a convergence point. We assume that conv_points(P) and div_points(P) denote, respectively, the set of convergence points and divergence points of a predicate P. We follow the syntax of [10] for PE annotations. An annotation is of the form "[*Precond*] ⇒ *Ann Pred*" where *Precond* is an optional precondition defined as a logic formula, *Ann* is the kind of annotation (only **memo** in this case), and *Pred* is a predicate descriptor, i.e., a predicate function and distinct free variables. Then, to achieve an effective transformation, we specialise the interpreter in Fig. 3 w.r.t. the program to be transformed by using the following annotation for each predicate `P/Ar` in the program:

$$\texttt{PP} \in \mathsf{div\_points}(\texttt{P/Ar}) \cup \mathsf{conv\_points}(\texttt{P/Ar}) \Rightarrow \textbf{memo}\ \texttt{exec}(\_, \texttt{st}(\texttt{PP}, \_, \_, \_, \_, \_), \_)$$

Additionally `solve/4` and `run_builtin/3` are also annotated to be memoised always to avoid code duplications.

This already describes how the specialisation has to be steered in the local control. As regards the global control, the only predicate which can introduce non-termination is `exec/3`. Its first and third arguments contain a fixed structure with variables. The second one might be problematic as it ranges over the set of all computable states at specialisation time. Note that the number of computable states remains finite thanks to the big-step nature of the interpreter. Still, it can

```
solve(foo/2,[C,D],A,B) :-
   run_builtin_1(E,C),
   exec_1(C,E,F,A,B), F = [D].

exec_1(A,no,F,G,H) :- exec_2(A,F,G,H).
exec_1(_,yes,[pos],yes,1).
exec_1(A,yes,F,G,H) :- exec_2(A,F,G,H).

exec_2(A,G,H,I) :-
   run_builtin_2(K,A), exec_3(K,G,H,I).

exec_3(no,[_],no,0).
exec_3(yes,[zero],yes,0).

run_builtin_1(yes,A) :- A#>0.
run_builtin_1(no,A) :- \+ A#>0.

run_builtin_2(yes,A) :- A#=0.
run_builtin_2(no,A) :- \+ A#=0.
```

Fig. 4. Transformed code with explicit failure for `foo/2`

happen that the same program point is reached with different values for the `NCPs` sub-term of the state. Therefore, if one wants to achieve the optimality criterion above, such argument has to be always generalised in global control.

Example 1. Figure 4 depicts the transformed code we obtain for predicate `foo/2`. It can be observed that there is a clear correspondence between the transformed code and the CFG in Fig. 2. Thus, predicate `solve/4` represents the node "`X>0`", `exec_1/5` implements its continuation, whose three clauses correspond to the three sub-paths $\langle 3 \rangle$, $\langle 1, 2 \rangle$ and $\langle 1, 2, 4 \rangle$ respectively. Predicate `exec_2/4` represents the node "`X=0`" and `exec_3/5` implements its continuation, whose two clauses correspond to the sub-paths $\langle 7 \rangle$ and $\langle 5, 6 \rangle$. Note that edge "8" is not considered in the meta-interpreter (nor in the transformed program) as it is meaningless for TDG. It is worth mentioning that the transformed program captures the way in which variable bindings are undone. For instance in `solve(foo/2,[C,D],...)`, if we keep track of variables `C` and `D`, it can be seen that `D`, which corresponds to variable `Z` in the original code, is only used for the final unification `F=[D]`, while new fresh variables are used for the unifications with `pos` and `zero`. However, variable `C`, which corresponds to variable `X` in the original code, is actually used for the checks in `run_builtin_1/2` and `run_builtin_2/2`. This turns out to be fundamental when trying to obtain test data associated to the *first-try failing* CP $\langle 0, 3, 7 \rangle$. It must be the same variable the one which, at the same time, is not "> 0" and not "$=0$". Otherwise we cannot obtain a negative number as test data for such CP. Finally, observe that the original Prolog arithmetic builtins have been (automatically) transformed into their `clpfd` counterparts [4].

5 Generating Test Cases by Partial Evaluation

Once the original Prolog program has been transformed into an equivalent Prolog program with explicit failure, we can use the approach of [2] to carry out **phase**

[4] We are using the `clpfd` library of `Sicstus` Prolog. See [20] for details.

II (see Fig. 1) and generate test data both for successful and failing derivations. As we have explained in Sect. 2.2, the idea is to perform a second PE over the CLP transformed program where the unfolding rule plays the role of the coverage criterion. In [2] an unfolding rule implementing the *block-count(k)* coverage criterion was proposed. A set of computation paths satisfies the *block-count*(k) *criterion* if it includes all terminating computation paths which can be built in which the number of times each block is visited does not exceed the given k. The blocks the criterion refers to are the blocks or nodes in the CFGs of the original Prolog program. As the only form of loops in Prolog are recursive calls, the "k" in the *block-count*(k) actually corresponds to the number of recursive calls which are allowed.

Unfortunately, the presence of Prolog's negation in our transformed programs complicates this phase. The negation will appear in the transformed program for "no" branches originating from nodes corresponding to a (possibly) failing builtin. See for example predicates `run_builtin_1/3` and `run_builtin_2/3` in the transformed code of `foo/2` in Fig. 4. While Prolog's negation works well for ground arguments, it gives no information for free variables, as it is required in the evaluation performed during this TDG phase. In particular, in the `foo/2` example, given the computation which traverses the calls "`\+ A#>0`" and "`\+ A#=0`" (corresponding to the path $\langle 0, 3, 7\rangle$ in the CFG), we need to infer that "`A<0`". In other words, we need somehow to turn the *negative* information into *positive* information. This transformation is straightforward for arithmetic builtins: we just have to replace "`\+` e_1`#=`e_2" by "e_1`#\=`e_2" and "`\+` e_1`#>`e_2" by "e_1`#=<`e_2", etc.

Example 2. This transformation allows us to obtain the following set of test-cases for `foo/2`:

$$\left\{\begin{array}{l}\langle\texttt{[1],[pos],yes/first-try}\rangle,\ \ \langle\texttt{[1],[\_],no/after-retry}\rangle,\\ \langle\texttt{[0],[zero],yes/first-try}\rangle,\ \langle\texttt{[-100],[\_],no/first-retry}\rangle\end{array}\right\}$$

They correspond respectively (reading by rows) to the CPs $\langle 0, 1, 2\rangle$, $\langle 0, 1, 2\rangle \cdot \langle 4, 7\rangle$, $\langle 0, 3, 5, 6\rangle$ and $\langle 0, 3, 7\rangle$. Each test-case is represented as a 3-tuple $\langle Ins, Outs, Ans\rangle$ being Ins the list of input arguments, $Outs$ the list of output arguments and Ans the answer. The answer takes the form A/B with $A \in \{$`yes,no`$\}$ and $B \in \{$`first-try,after-retry`$\}$[5], so that we obtain sufficient information about the kind of CP to which the test-case corresponds (see Sect. 3). As there are no recursive calls in `foo/2` such test-cases are obtained using the *block-count*(k) criterion for any k (greater than 0). The domain used for the integer number is $\{-100..100\}$.

However, it can be the case that negation involves unifications with symbolic data. For example, the transformed code for `sorted/1` includes the negations "`\+ L=[]`" and "`\+ L=[_|_]`". As before, we might write transformations for the negated unifications involving lists, so that at the end it is inferred that "

[5] To simplify the presentation in Sect. 4.1, we decided not include in the interpreter the support to calculate the `first-try/after-retry` value.

`L=[_,_|_]`". However this would be too an ad-hoc solution as many distinct term structures, different from lists, can appear on negated unifications. A solution for this problem has been recently proposed for Mercury in the same context [5]. It roughly consists in the following: 1) It is assumed that each predicate argument is well-typed. 2) A domain is initialised for each variable, containing the set of possible functors the variable can take. 3) When a negated unification involving an output variable is found (in their terminology a negated *decomposition*), the corresponding functor is removed from the variable domain. It is crucial at this point the assumption that complex unifications are broken down into simple ones. 4) Finally, a search algorithm is described to generate particular values from the type definition and final domain for the variable. The technique is implemented using CHR and can be directly used in principle for our purposes as well.

On the other hand, advanced declarative languages like TOY [15] make possible the co-existence of different constraint domains. In particular, the co-existence of boolean and numeric constraint domains enables the possibility of using *disequalities* involving both symbolic data and numbers. This allows for example expressing the negated unifications "`\+ L=[]`" and "`\+ L=[_|_]`" as disequality constraints " `L/=[]`" and " `L/=[_|_]`". Additionally, by relying on the boolean constraint solver, the negated arithmetic builtins "`\+ A#>0`" and "`\+ A#=0`" can be encoded as "`(A#>0) == false`" and "`(A#=0) == false`". This is in principle a more general solution that we want to explore, although a thorough experimental evaluation needs to be carried out to demonstrate its applicability to our particular context.

Example 3. Now, by using any of the techniques outlined above, we obtain the following set of test-cases for `sorted/1`, using *block-count*(2) as the coverage criterion:

$$\left\{\begin{array}{ll} \langle \texttt{[[]],[],yes/first-try} \rangle, & \langle \texttt{[[0]],[],yes/first-try} \rangle, \\ \langle \texttt{[[0,1]],[],yes/first-try} \rangle, & \langle \texttt{[[0,1,2]],[],yes/first-try} \rangle, \\ \langle \texttt{[[0,1,2,0|\_]],[],no/first-try} \rangle, & \langle \texttt{[[0,1,0|\_]],[],no/first-try} \rangle, \\ \langle \texttt{[[0,0|\_]],[],no/first-try} \rangle & \end{array}\right\}$$

They correspond respectively (reading by rows) to the CPs "$\langle 0, 1\rangle$", "$\langle 0, 2, 4\rangle$", "$\langle 0, 2, 5, 7, 10\rangle \cdot \langle 0, 2, 4\rangle \cdot \langle 11\rangle$", "$\langle 0, 2, 5, 7, 10\rangle \cdot \langle 0, 2, 5, 7, 10\rangle \cdot \langle 0, 2, 4\rangle \cdot \langle 11\rangle \cdot \langle 11\rangle$", "$\langle 0, 2, 5, 7, 10\rangle \cdot \langle 0, 2, 5, 7, 10\rangle \cdot \langle 0, 2, 5, 7, 9\rangle \cdot \langle 12\rangle \cdot \langle 12\rangle$", "$\langle 0, 2, 5, 7, 10\rangle \cdot \langle 0, 2, 5, 7, 9\rangle \cdot \langle 12\rangle$", "$\langle 0, 2, 5, 7, 9\rangle$". They are indeed all the paths that can be followed with no more than 3 recursive calls. This time the domain has been set up to $\{0..100\}$.

6 Conclusions and Ongoing work

Very recently, we proposed in [2] a generic approach to TDG by PE which in principle can be used for any imperative language. However, applying this approach to TDG of a declarative language like Prolog introduces some difficulties like the handling of failing derivations and of symbolic data. In this work, we

have sketched solutions to overcome such difficulties. In particular, we have proposed a program transformation, based on PE, to make failure explicit in the Prolog programs. To handle Prolog's negation in the transformed programs, we have outlined existing solutions that make it possible to turn the negative information into positive information. Though our preliminary experiments already suggest that the approach can be very useful to generate test-cases for Prolog, we plan to carry out a thorough practical assessment. This requires to cover additional Prolog features like the module system, builtins like `cut/0`, `fail/0`, `if/3`, etc. and also to compare the results with other TDG systems. We also want to study the integration of other kinds of coverage criteria like *data-flow* based criteria. Finally, we would like to explore the use of static analyses in the context of TDG. For instance, the information inferred by a *failure analysis* can be very useful to prune some of the branches that our transformed programs have to consider.

Acknowledgments This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Proc. PADL*, number 4354 in LNCS. Springer-Verlag, 2007.
2. E. Albert, M. Gómez-Zamalloa, and G. Puebla. Test Data Generation of Bytecode by clp Partial Evaluation. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, LNCS. Springer-Verlag, July 2008. To appear.
3. F. Belli and O. Jack. Implementation-based analysis and testing of prolog programs. In *ISSTA*, pages 70–80, 1993.
4. Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.
5. F. Degrave, T. Schrijvers, and W. Vanhoof. Automatic generation of test inputs for mercury. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, LNCS. Springer-Verlag, 2008. To appear.
6. R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
7. S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *PPDP*, pages 63–74, 2007.
8. Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
9. J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98. ACM Press, 1993.
10. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Modular Decompilation of Low-Level Code by Partial Evaluation. In *8th International Working Conference on*

Source Code Analysis and Manipulation (SCAM'08). IEEE Computer Society, September 2008. To appear.
11. A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Computational Logic*, pages 399–413, 2000.
12. Kim S. Henriksen and John P. Gallagher. Abstract interpretation of pic programs through logic programming. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 184–196. IEEE Computer Society, 2006.
13. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
14. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
15. F.J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
16. G. Luo, G. Bochmann, B. Sarikaya, and M. Boyer. Control-flow based testing of prolog programs. In *In Proc. of the 3rd International Symposium on Software Reliability Engineering*, pages 104–113, 1992.
17. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.
18. C. Meudec. Atgen: Automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.
19. J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In *Proc. of SAS'98*, volume 1503 of *LNCS*, pages 246–261, 1998.
20. Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog 3.8 User's Manual*, 3.8 edition, October 1999. Available from `http://www.sics.se/sicstus/`.
21. L. Zhao, T. Gu, J. Qian, and G. Cai. A novel test case generation method for prolog programs based on call patterns semantics. In *APLAS*, pages 105–121, 2007.
22. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Heap Space Analysis for Java Bytecode

Elvira Albert
DSIC, Complutense University of Madrid, Spain
elvira@sip.ucm.es

Samir Genaim
CLIP, Technical University of Madrid, Spain
samir@clip.dia.fi.upm.es

Miguel Gómez-Zamalloa
DSIC, Complutense University of Madrid, Spain
mzamalloa@fdi.ucm.es

Abstract

This article presents a heap space analysis for (sequential) Java bytecode. The analysis generates *heap space cost relations* which define at compile-time the heap consumption of a program as a function of its data size. These relations can be used to obtain upper bounds on the heap space allocated during the execution of the different methods. In addition, we describe how to refine the cost relations, by relying on *escape analysis*, in order to take into account the heap space that can be safely deallocated by the garbage collector upon exit from a corresponding method. These refined cost relations are then used to infer upper bounds on the *active heap space* upon methods return. Example applications for the analysis consider inference of constant heap usage and heap usage proportional to the data size (including polynomial and exponential heap consumption). Our prototype implementation is reported and demonstrated by means of a series of examples which illustrate how the analysis naturally encompasses standard data-structures like lists, trees and arrays with several dimensions written in object-oriented programming style.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]: General; D3.2 [*Programming Languages*]

General Terms Languages, Theory, Verification, Reliability

Keywords Heap Space Analysis, Heap Consumption, Low-level Languages, Java Bytecode

1. Introduction

Heap space analysis aims at inferring *bounds* on the heap space consumption of programs. Heap analysis is more typically formulated at the source level (see, e.g., [24, 17, 25, 19] in the context of functional programming and [18, 13] for high-level imperative programming languages). However, there are situations where one has only access to compiled code and not to the source code. An example of this is *mobile code*, where the code consumer receives code to be executed. In this context, Java bytecode [20] is widely used, mainly due to its security features and the fact that it is platform-independent. Automatic heap space analysis has interesting applications in this context. For instance, *resource bound certification* [14, 4, 5, 16, 12] proposes the use of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on the resource consumption. Also, heap bounds are useful on embedded systems, e.g., smart cards in which memory is limited and cannot easily be recovered. A general framework for the cost analysis of sequential Java bytecode has been proposed in [2]. Such analysis statically generates *cost relations* which define the cost of a program as a function of its input data size. The cost relations are expressed by means of *recursive equations* generated by abstracting the recursive structure of the program and by inferring size relations between arguments. Cost relations are parametric w.r.t. a *cost model*, i.e., the cost unit associated to the bytecode b appears as an abstract value T_b within the equations.

This article develops a novel application of the cost analysis framework of [2] to infer bounds on the heap space consumption of sequential Java bytecode programs. In a first step, we develop a cost model that defines the cost of memory allocation instructions (e.g., `new` and `newarray`) in terms of the number of heap (memory) units it consumes. E.g., the cost of creating a new object is the number of heap units allocated to that object. The remaining bytecode instructions do not add any cost. With this cost model, we generate heap space cost relations which are then used to infer upper bounds on the heap space usage of the different methods. These upper bounds provide information on the maximal heap space required for executing each method in the program. In a second step, we refine this cost model to consider the effect of garbage collection. This is done by relying on escape analysis [15, 8] to identify those memory allocation instructions which create objects that will be

ISMM'07, October 21–22, 2007, Montréal, Québec, Canada.

garbage collected upon exit from the corresponding method. With this information available, we can generate heap space cost relations which contain annotations for the heap space that will be garbage collected. The annotated cost relations in turn are used to infer upper bounds on the *active heap space* upon exit from methods, i.e., the heap space consumed and that might not be garbage collected upon exit.

A distinguishing feature of the approach presented in this article w.r.t. previous type-based approaches (e.g., [5, 17]) is that it is not restricted to linear bounds since the generated cost relations can in principle capture any complexity class. Moreover, in many cases, the relations can be simplified to a *closed form* solution from which one can glean immediate information about the expected consumption of the code to be run. The approach has been assessed by means of a prototype implementation, which originates from the one of [3]. It should be noted that the examples in [3] are simple imperative algorithms which did not make use of the heap, since they were aimed at demonstrating that traditional complexity schemata can be handled by the cost analysis of [2]. In contrast, we demonstrate our heap analysis by means of a series of example applications written in an object-oriented style which make intensive use of the heap and which present novel features like heap consumption that depends on the class fields, multiple inheritance, virtual invocation, etc. These examples allow us to illustrate the most salient features of our analysis: inference of constant heap usage, heap usage proportional to input size, support of standard data-structures like lists, trees, arrays, etc. To the best of our knowledge, this is the first analysis able to infer arbitrary heap usage bounds for Java bytecode.

The rest of the paper is structured as follows: Sec. 2 presents an example that illustrates the ideas behind the analysis. Sec. 3 briefly describes the Java bytecode language. Sec. 4 defines a cost model for heap consumption and describes the analysis framework. Sec. 5 demonstrates the different features of the analysis by means of examples. In Sec.6, we extend our cost model to consider the effect of garbage collection. Sec. 7 reports on a prototype implementation and some experimental results. Finally, Sec. 8 concludes and discusses the related work.

2. Worked Example

Consider the Java classes and their corresponding (structured) Java bytecode depicted in Fig. 1 which define a linked-list data structure in an object-oriented style, as it appears in [18]. The class *Cons* is used for data nodes and the class *Nil* plays the role of *null* to indicate the end of a list. Both classes define a copy function which is used to clone the corresponding object. In the case of *Nil* the copy method just returns *this* since it is the last element of the list, and in the case of *Cons* it clones the current object and its successors recursively (by calling the *copy* method of *next*). The rest of this section describes the different steps applied by the analyzer to approximate the heap consumption of the program depicted in Fig. 1. Note that the Java program is provided here just for clarity, the analyzer works directly on the bytecode which is obtained, for example, by compiling the Java program.

Step I: In the first step, the analyzer recovers the structure of the Java bytecode program by building a control flow graph (CFG) for its methods. The CFG consists of basic blocks which contain a sequence of non-branching bytecode instructions, these blocks are connected by edges that describe the possible flows that originate from the branching instructions like conditional jumps, exceptions, virtual method invocation, etc. In Fig. 1, the CFG of the method *Nil.copy* consists of the single block $Block_0^{Nil}$ and the CFG of the method *Cons.copy* consists of the rest of the blocks. $Block_0^{Cons}$ corresponds to the bytecode of *Cons.copy* up to the recursive method call *this.next.copy()*. Then, depending on the type of the object stored in *this.next* the execution is transferred to either *Nil.copy* or *Cons.copy*. This is expressed by the (guarded) branching to $Block_1^{Cons}$ and $Block_2^{Cons}$. In both cases, the control returns to $Block_3^{Cons}$ which corresponds to the rest of the statements.

Step II: In the second step, the analyzer builds an intermediate representation for the CFG and uses it to infer information about the changes in the sizes of the different data-structures (or in the values of integer variables) when the control passes from one part of the program (e.g., a block or a method) to another part. For example, this step infers that when *Nil.copy* or *Cons.copy* are called recursively, the length of the list decreases by one. This information is essential for defining the heap consumption of one part of the program in terms of the heap consumption of other parts.

Step III: In the third step, the intermediate representation and the size information are used together with the cost model for heap consumption to generate a set of cost relations which describe the heap consumption behaviour of the program. The following equations are the ones we get for the example in Fig. 1:

Heap Space Cost Equations			Size relations
$C_{copy}^{Nil}(a)$	=	0	$\{a{=}1\}$
$C_{copy}^{Cons}(a)$	=	$C_0(a)$	
$C_0(a)$	=	$size(Cons) + CC_0(a,b)$	$\{a{\geq}1, b{\geq}0, a{=}b{+}1\}$
$CC_0(a,b)$	=	$\begin{cases} C_1(a,b) & \hat{b} \in Nil \\ C_2(a,b) & \hat{b} \in Cons \end{cases}$	
$C_1(a,b)$	=	$C_{copy}^{Nil}(b) + C_3(a)$	$\{a{=}1\}$
$C_2(a,b)$	=	$C_{copy}^{Cons}(b) + C_3(a)$	$\{a{\geq}2\}$
$C_3(a)$	=	0	

Each of these equations corresponds to a method entry, block or branching in the CFG. An equation is composed by the left hand side which indicates the block or the method it represents, and the right hand side which defines its heap consumption behaviour. In addition, size relations might be attached to describe how the data size changes when using another equation.

```
abstract class List {
    abstract List copy();
}
class Nil extends List {
    List copy() {
        return this;
    }
}
class Cons extends List {
    int elem;
    List next;
    List copy(){
        Cons aux = new Cons();
        aux.elem = this.elem;
        aux.next = this.next.copy();
        return aux;
    }
}
```

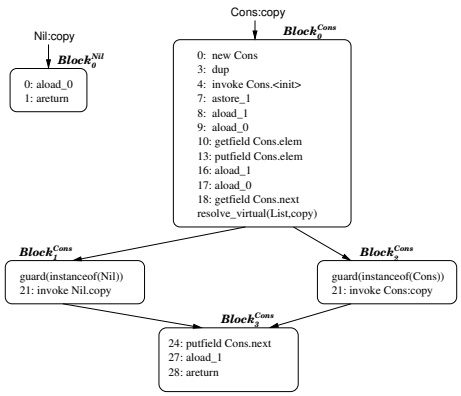


Figure 1. Java source code and CFG bytecode of example

The equation $C^{Nil}_{copy}(a)$ defines the heap consumption of *Nil.copy* in terms of (the size of) its first argument a which corresponds to its *this* reference variable (in Java bytecode the *this* reference variable is the first argument of the method). In this case the heap consumption is zero since the method does not allocate any heap space. The equation $C^{Cons}_{copy}(a)$ defines the heap consumption of *Cons.copy* as the heap consumption of $Block^{Cons}_0$ using the corresponding equation C_0, which in turn defines the heap consumption as the amount of heap units allocated by the *new* bytecode instructions, namely $size(Cons)$, plus the heap consumption of its successors which is defined by the equation CC_0. All other instructions in $Block^{Cons}_0$ contribute zero to the heap consumption. Note that in C_0, the variable b corresponds to *this.next* of *Cons.copy* and that the size analysis is able to infer the relation $a{=}b{+}1$ (i.e., the list a is longer than b by one). The equation CC_0 corresponds to the heap consumption of the branches at the end of $Block^{Cons}_0$, depending on the type of b (denoted as $\hat{b}$) it is equal to the heap consumption of $Block^{Cons}_1$ or $Block^{Cons}_2$ which are respectively defined by the equations C_1 and C_2. The equation C_1 defines the heap consumption of $Block^{Cons}_1$ as the heap consumption of *Nil.copy* (since it is called in $Block^{Cons}_1$) plus the heap consumption of $Block^{Cons}_3$ (using the equation C_3). Similarly C_2 defines the heap consumption of $Block^{Cons}_2$ in terms of the heap consumption of *Cons.copy*. The equation C_3 defines the heap consumption of $Block^{Cons}_3$ to be zero since it does not allocate any heap space.

Step IV: In the fourth step, we can simplify the equations and try to obtain an upper bound in closed form for the cost relation by applying the method described in [1]. In particular, assuming that $size(Cons)$ equals 8 (4 bytes for the integer field *data* and 4 bytes for the reference field *next*), we obtain the following simplified equations:

Equation			Size relations
$C^{Nil}_{copy}(a)$	=	0	$\{a{=}1\}$
$C^{Cons}_{copy}(a)$	=	8	$\{a{=}2\}$
$C^{Cons}_{copy}(a)$	=	$8 + C^{Cons}_{copy}(b)$	$\{a{\geq}3, b{\geq}1, a{=}b{+}1\}$

and then obtain an upper bound in closed form $C^{Cons}_{copy}(a) = 8 * (a-1)$.

The main focus of this paper is on the generation of heap space cost relations, as illustrated in Step III. Steps I and II are done as it is proposed in [2] and Step IV as it is described in [1] and hence we will not give many details on how they are performed in this paper.

3. The Java Bytecode Language

Java bytecode [20] is a low-level object-oriented programming language with unstructured control and an *operand stack* to hold intermediate computational results. Moreover, objects are stored in dynamic memory: the *heap*. A Java bytecode program consists of a set of *class files*, one for each class or interface. A class file contains information about its *name* $c \in Class\_Name$, the class it extends, the interfaces it implements, and the fields and methods it defines. In particular, for each method, the class file contains: a method signature which consists of its name and its type; its bytecode $bc_m = \langle pc_0{:}b_0, \ldots, pc_{n_m}{:}b_{n_m}\rangle$, where each b_i is a *bytecode instruction* and pc_i is its address; and the method's exceptions table. In this work we consider a subset of the JVM [20] language which is able to handle operations on integers and references, object creation and manipulation (by accessing fields and calling methods), arrays of primitive and reference types, and exceptions (either generated by abnormal execution or explicitly thrown by the program). For simplicity, we omit static fields and initializers and primitive types different from integers. Such features could be handled by making the underlying abstract interpretation support them by assuming the worst case approximation for them. Thus, our bytecode instruction set ($bcInst$) is:

$bcInst$ ::=
  push x | istore v | astore v | iload v | aload v | iconst a
| iadd | isub | imul | idiv | if$\diamond$ pc | goto pc | ireturn | areturn
| return | new $Class\_Name$ |
| newarray int | anewarray $Class\_Name$ | iaload | aaload
| iastore | aastore | athrow | dup
| invokevirtual/invokespecial $Class\_Name.Meth\_Sig$
| getfield/putfield $Class\_Name.Field\_Sig$

where $\diamond$ is a comparison operator (ne,le,_icmpgt, etc.), v a local variable, a an integer, pc an instruction address, and x an integer or the special value `null`.

4. The Heap Space Analysis Framework

Cost analysis of a low-level object-oriented language such as Java bytecode is complicated mainly due to its unstructured control flow (e.g., the use of goto statements rather than recursive structures), its object-oriented features (e.g., virtual method invocation) and its stack-based model. The recent work of [2] develops a generic framework for the automatic cost analysis of Java bytecode programs. Essentially, the complications of dealing with a low-level language are handled in this framework by abstracting the *recursive structure* of the program and by inferring *size relations* between arguments. As we have seen in Sect. 2, this analysis framework is based on transforming the Java bytecode program to an intermediate representation which fits inside the same setting all possible forms of loops. Then, using this intermediate representation, the analysis infers information about the change in the sizes of the relevant data structures as the program goes through its loops (Steps I and II). Finally, this information is used to set up a cost relation which defines the cost of the program in terms of the sizes of the corresponding data structures.

In this section, we present a novel application of this generic cost analysis framework to infer bounds on the heap space consumption of sequential Java bytecode programs. So far, this framework has been only used in [3] to infer the complexity of some classical algorithms while in this paper our purpose is completely different: we aim at computing bounds on the heap usage for programs written in object-oriented programming style which make intensive use of the heap. In Sect. 4.1 and Sect. 4.2, we briefly present the notions of recursive representation and calls-to size-relation in a rather informal style. Then, we introduce our cost model for heap consumption and our notion of heap space cost relation in Sect. 4.3.

4.1 Recursive Representation

Cost relations can be elegantly expressed as systems of *recursive* equations. In order to automatically generate them, we need to capture the iterative behaviour of the program by means of recursion. One way of achieving this is by computing the CFG of the program. Also, advanced features like virtual invocation and exceptions are simply dealt as additional nodes in the graph. To analyze the bytecode, its CFG can be represented by using some auxiliary recursive representation (see, e.g., [2]). In this approach, a *bytecode* is transformed into a set of *guarded rules* of the form $\langle head \leftarrow guard, body\rangle$ where the *guard* states the applicability conditions for the rule. Rules are obtained from blocks in the CFG and *guards* indicate the conditions under which each block is executed. As it is customary in deterministic imperative languages, guards provide mutually exclusive conditions because paths from a block are always exclusive (i.e., alternative) choices.

DEFINITION 4.1 (rec. representation). *Consider a block* $\mathtt{p}$ *in a CFG, which contains a sequence of bytecode instructions* $\mathtt{B}$ *guarded by the condition* $\mathtt{G_b}$ *and whose successor blocks are* $\mathtt{q_1}, \cdots, \mathtt{q_n}$. *The* recursive representation *of* $\mathtt{p}$ *is:*

$$\mathtt{p}(\bar{\mathtt{l}}, \bar{\mathtt{s}}, \mathtt{r}) \leftarrow \mathtt{G_p}, \mathtt{B}, (\mathtt{q_1}(\bar{\mathtt{l}}, \bar{\mathtt{s}}', \mathtt{r}); \cdots ; \mathtt{q_n}(\bar{\mathtt{l}}, \bar{\mathtt{s}}', \mathtt{r}))$$

where:

- $\bar{l}$ *is a tuple of variables which corresponds to the method's local variables,*
- $\bar{s}$ *and* $\bar{s}'$ *are tuples of variables which respectively correspond to the active stack elements at the block's entry and exit,*
- r *is a single variable which corresponds to the method's return value (omitted if there is not return value),*
- G_p *and* B *are obtained from the block's guard and bytecode instructions by adding the local variables and stack elements on which they operate as explicit arguments.*

We denote by $\mathtt{calls}(\mathtt{B})$ *the set of method invocation instructions within* $\mathtt{B}$ *and by* $\mathtt{bytecode}(\mathtt{B})$ *the other instructions.* □

The formal translation of bytecode instructions in `B` to calls within the recursive rules is presented in [2]. In this translation, it is interesting to note that the stack positions are visible in the rules by explicitly defining them as local variables. This intermediate representation is convenient for analysis as in one pass we can eliminate almost all stack variables which results in a more efficient analysis.

EXAMPLE 4.2. *The rules that correspond to the blocks* $Block_0^{Cons}$, $Block_1^{Cons}$ *and* $Block_2^{Cons}$ *in Fig. 1 are:*

$$\begin{array}{l}
\mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{r}) \leftarrow \\
\quad \mathtt{new}(\mathtt{Cons}, \mathtt{s}_0), \mathtt{dup}(\mathtt{s}_0, \mathtt{s}_1), \mathtt{Cons.<init>}(\mathtt{s}_1), \\
\quad \mathtt{astore}(\mathtt{s}_0, \mathtt{aux}'), \mathtt{aload}(\mathtt{aux}', \mathtt{s}_0'), \mathtt{aload}(\mathtt{this}, \mathtt{s}_1'), \\
\quad \mathtt{getfield}(\mathtt{Cons.elem}, \mathtt{s}_1', \mathtt{s}_1''), \\
\quad \mathtt{putfield}(\mathtt{Cons.elem}, \mathtt{s}_0', \mathtt{s}_1''), \\
\quad \mathtt{aload}(\mathtt{aux}', \mathtt{s}_0''), \mathtt{aload}(\mathtt{this}, \mathtt{s}_1'''), \\
\quad \mathtt{getfield}(\mathtt{Cons.next}, \mathtt{s}_1''''), \\
\quad (\mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}) ; \\
\qquad \mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r})).
\end{array}$$

$$\begin{array}{l}
\mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1, \mathtt{r}) \leftarrow \\
\quad \mathtt{guard}(\mathtt{instanceof}(\mathtt{s}_1, \mathtt{Nil})), \\
\quad \mathtt{Nil.copy}(\mathtt{s}_1, \mathtt{s}_1'), \\
\quad \mathtt{copy}_3^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1', \mathtt{r}).
\end{array}$$

$$\begin{array}{l}
\mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1, \mathtt{r}) \leftarrow \\
\quad \mathtt{guard}(\mathtt{instanceof}(\mathtt{s}_1, \mathtt{Cons})), \\
\quad \mathtt{Cons.copy}(\mathtt{s}_1, \mathtt{s}_1'), \\
\quad \mathtt{copy}_3^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1', \mathtt{r}).
\end{array}$$

The rule $\mathtt{copy}_0^{\mathtt{Cons}}$ *is not guarded and has two continuation blocks, while the other rules are guarded by the type of*

the object of s_1 (the top of the stack) and have only one successor. The bytecode instructions were transformed to include explicitly the stacks elements and the local variables on which they operate, moreover, all variables are in single static assignment *form. Note that calls to methods take the same form as calls to blocks, which makes all different forms of loops to fit in the same setting.* □

4.2 Size Analysis

A size analysis is then performed on the recursive representation in order to infer the *calls-to size-relations* between the variables in the head of the rule and the variables used in the calls (to rules) which occur in the body for each program rule. Derivation of constraints is a standard abstract interpretation over a constraints domain such as Polyhedra [2, 3]. Such relations are essential for defining the cost of one block in terms of the cost of its successors. The analysis is done by abstracting the bytecode instructions into the linear constraints they impose on their arguments, and then computing a fixpoint that collects *calls-to* relations.

DEFINITION 4.3 (calls-to size-relations). *Consider the rule in Def. 4.1, its calls-to* size-relations *are triples of the form*

$$\langle \mathtt{p}(\bar{\mathtt{x}}), \mathtt{p}'(\bar{\mathtt{z}}), \varphi \rangle \quad \textit{where } \mathtt{p}'(\bar{\mathtt{z}}) \in \mathtt{calls(B)} \cup \mathtt{q_1}(\bar{\mathtt{y}}) \cup \ldots \cup \mathtt{q_n}(\bar{\mathtt{y}})$$

The size-relation φ *is given as a conjunction of linear constraints. The tuples of variables $\bar{x}$, $\bar{y}$ and $\bar{z}$ correspond to the variables of the corresponding block.* □

In Java bytecode, we consider three cases within size relations: for integer variables, *size-relations* are constraints on the possible values of variables; for reference variables, they are constraints on the length of the longest reachable paths [21], and for arrays they are constraints on the length of the array. Note that using the path-length notion cyclic structures are not handled since to guarantee soundness the corresponding references are abstracted to "unknown-length" and therefore cost that depends on them cannot be inferred.

EXAMPLE 4.4. *The calls-to-size relation for the first rule in Ex. 4.2 is formed by the triples:*

$$\langle \mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}), \mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}), \varphi \rangle$$
$$\langle \mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}), \mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}), \varphi \rangle$$

where φ includes, among others, the constraint $this{=}s_1''''{+}1$ which states that the list that $this$ points to is longer by one than the list that s_1'''' points to (s_1'''' corresponds to $this.next$). The meaning of the above relations is explained in Section 2. Note that the call to the constructor Cons.<init> *is ignored for simplicity.* □

4.3 Heap Space Cost Relations

In order to define our heap space cost analysis, we start by defining a cost model which defines the cost of memory allocation instructions (e.g., new, newarray and anewarray) as the the number of heap (memory) units they consume. The remaining bytecode instructions do not add any cost.

DEFINITION 4.5 (cost model for heap space). *We define a cost model $\mathcal{M}_{heap}$ which takes a bytecode instruction* bc *and returns a positive expression as follows:*

$$\mathcal{M}_{heap}(\mathtt{bc}) = \begin{cases} size(\mathtt{Class}) & \textit{if bc=}\mathtt{new(Class,\_)} \\ S_{\mathtt{PrimType}} * \mathtt{L} & \textit{if bc=}\mathtt{newarray(PrimType,L,\_)} \\ S_{\mathtt{ref}} * \mathtt{L} & \textit{if bc=}\mathtt{anewarray(Class,L,\_)} \\ 0 & \textit{otherwise} \end{cases}$$

where $S_{\mathtt{PrimType}}$ and $S_{\mathtt{ref}}$ denote, respectively, the heap consumption of primitive types and references. Function size is defined as follows:

$$size(O) = \begin{cases} \sum_{F \in \mathtt{Class.field}} size(type(F)) & \textit{if O=}\mathtt{Class} \\ S_{\mathtt{PrimType}} & \textit{if O is a primitive type} \\ S_{\mathtt{ref}} & \textit{if O is a reference type} \end{cases}$$

where the type of a field in a Class *(i.e.,* Class.field*) can be either primitive or reference.* □

In Java bytecode, types are classified into primitive (its size is represented by $S_{\mathtt{PrimType}}$ in our model) and reference types ($S_{\mathtt{ref}}$). In a particular assessment, one has to set the concrete values for $S_{\mathtt{PrimType}}$ and $S_{\mathtt{ref}}$ of the JVM implementation.

For each rule in the recursive representation of the program and its corresponding size relation, the analysis generates the cost equations which define the heap consumption of executing the block (or possibly a method call) by relying on the above cost model. A *heap space cost relation* is defined as the set of cost equations for each block of the bytecode (or rule in the recursive representation).

DEFINITION 4.6 (heap space cost relation). *Consider a rule $\mathcal{R}$ of the form $\mathtt{p}(\bar{\mathtt{x}}) \leftarrow \mathtt{G_p}, \mathtt{B}, (\mathtt{q_1}(\bar{\mathtt{y}}); \cdots; \mathtt{q_n}(\bar{\mathtt{y}}))$ and let the linear constraints φ be a conjunction of all call-to size-relations within the rule. The* heap space cost *equations for $\mathcal{R}$ are generated as follows:*

$$C_p(\bar{x}) = \sum_{b \in bytecode(B)} \mathcal{M}_{heap}(b) + \sum_{r(\bar{z}) \in calls(B)} + C_r(\bar{z}) + C_{p\_cont}(\bar{y}) \quad \varphi$$
$$C_{p\_cont}(\bar{y}) = C_{q_1}(\bar{y}) \quad G_{q_1}$$
$$\ldots$$
$$C_{p\_cont}(\bar{y}) = C_{q_n}(\bar{y}) \quad G_{q_n}$$

where $\mathtt{G_{q_i}}$ is the guard of $\mathtt{q_i}$. The heap space cost relation associated to the recursive representation of a method is defined as the set of cost equations for its blocks. □

When the rule has multiple *continuation*s, it is transformed into several equations. We specify the cost of each continuation in a separate equation because the guards for determining the alternative path $\mathtt{q_i}$ that the execution will take (with $i = 1, \ldots, n$) are only known at the end of the execution of the bytecode B; thus, they cannot be evaluated before B is executed. The guards appear also decorating the equations. In the implementation, when a rule has only one continuation, it gives rise to a single equation which contains the size relation φ as an attachment.

Java source code

```
abstract class Data{
  abstract public Data copy();
}
class Polynomial extends Data{
  private int deg;
  private int[] coefs;
  public Polynomial() {
    coefs = new int[11];}
  public Data copy() {
    Polynomial aux = new Polynomial();
    aux.deg = deg;
    for (int i=0;i<=deg && i<=10;i++)
      aux.coefs[i] = coefs[i];
    return aux;}}
```

```
class Vector3D extends Data{
  private int x;
  private int y;
  private int z;
  public Vector3D(int x,int y,int z) {
    this.x = x;
    this.y = y;
    this.z = z;}
  public Data copy() {
    return new Vector3D(x,y,z);
  }
}
```

```
class Results{
  Data[] rs;
  public Results() {
    rs = new Data[25];
  }
```

```
public Results copy() {
  Results aux = new Results();
  for (int i = 0;i < 25;i++)
    aux.rs[i] = rs[i].copy();
  return aux;}}
```

Heap space cost equations

Equation	Guard	Size rels.
$C_{copy}(a) = S_{ref} + 25{*}S_{ref} + C_0(a,0)$		
$C_0(a,i) = \underbrace{S_{int} + S_{ref}}_{size(Polyn)} + 11{*}S_{int} + C_0(a,j)$	$\langle \hat{a}.rs[i] \in Polyn \rangle$	$\{i{<}25,\ j = i{+}1\}$
$C_0(a,i) = \underbrace{3{*}S_{int}}_{size(Vect3D)} + C_0(a,j)$	$\langle \hat{a}.rs[i] \in Vect3D \rangle$	$\{i{<}25,\ j = i{+}1\}$
$C_0(a,i) = 0$		$\{i{>=}25\}$

Figure 2. Constant heap space example

EXAMPLE 4.7. *The heap space cost equations generated for the rule* $copy_0^{Cons}$ *of Ex. 4.2 and the size relation of Ex. 4.4 are (see Sect. 2):*

$$C_0^{Cons}(this, aux) = size(Cons) + CC_0^{Cons}(this, aux', s_0'', s_1'''')\{this = s_1'''' + 1, \ldots\}$$

$$CC_0^{Cons}(this, aux', s_0'', s_1'''') = \begin{cases} C_1^{Cons}(this, aux', s_0'', s_1'''') & \hat{s}_1'''' \in Nil \\ C_2^{Cons}(this, aux', s_0'', s_1'''') & \hat{s}_1'''' \in Cons \end{cases}$$

The cost of $Block_0^{Cons}$ *is captured by* C_0^{Cons}*, among all bytecode instructions in* $Block_0^{Cons}$*, we count only the creation of the object of class* `Cons`*. The continuation of* $Block_0^{Cons}$ *is captured in the relation* CC_0^{Cons}*, where depending on the type of the object* s_1''''*, we choose between two* mutually exclusive *equations* C_1^{Cons} *or* C_2^{Cons}*.* □

In addition, the analyzer performs a slicing step, which aims at removing variables that do not affect the cost. And also tries to simplify the equations as much as possible by applying unfolding steps. These steps lead to simpler cost relations. Due to lack of space, during the rest of the paper we will apply them without giving details on how they were performed.

5. Example Applications of Heap Space Analysis

In this section, we show the most salient features of our heap space analysis by means of a series of examples. All examples are written in object-oriented style and make intensive use of the heap. We intend to illustrate how our analysis is able to deal with standard data-structures like lists, trees and arrays with several dimensions as well as with multiple inheritance, class fields, virtual invocation, etc. We show examples which present heap usage which depends proportionally to the *data size*, namely in some cases it depends on class fields while in another one on the input arguments. An interesting point is that heap consumption is, in the different examples, constant, linear, polynomial or exponentially proportional to the data sizes.

For each example, we show the Java source code and its heap space cost relation. Each relation consists of three parts: the equations, the guards and the size relations. The applicability conditions of each equation are defined by the guards and the size relations. Guards usually provide non-numeric conditions while size relations provide conditions on the sizes of the corresponding variables. In addition, size relations describe how the data changes when the control moves from one to another part of the program. Since our

Java source code

```
abstract class List{
  abstract public List copy();
}
class Nil extends List{
  public List copy() {
    return this;
}
```

```
class Cons extends List
  private Data elem;
  private List next;
  public List copy() {
    Cons aux = new Cons();
    aux.elem = this.elem.copy();
    aux.next = this.next.copy();
    return aux;}
```

Heap space cost equations

Equation	Guard	Size rels.
$C_{copy}(a) = \overbrace{2{*}S_{ref}}^{size(Cons)} + \overbrace{S_{int}+S_{ref}+11{*}S_{int}}^{this.elem.copy()}$	$\left\langle \begin{array}{l} \hat{a}.elem \in Polyn \wedge \\ \hat{a}.next \in Nil \end{array} \right\rangle$	$\{a=2\}$
$C_{copy}(a) = 2{*}S_{ref}+S_{int}+S_{ref} +11{*}S_{int}+C_{copy}(b)$	$\left\langle \begin{array}{l} \hat{a}.elem \in Polyn \wedge \\ \hat{a}.next \in Cons \end{array} \right\rangle$	$\left\{ \begin{array}{l} a{\geq}3, b{\geq}2 \\ a{>}b \end{array} \right\}$
$C_{copy}(a) = \underbrace{2{*}S_{ref}}_{size(Cons)} + \underbrace{3{*}S_{int}}_{this.elem.copy()}$	$\left\langle \begin{array}{l} \hat{a}.elem \in Vect3D \wedge \\ \hat{a}.next \in Nil \end{array} \right\rangle$	$\{a=2\}$
$C_{copy}(a) = 2{*}S_{ref}+3{*}S_{int}+C_{copy}(b)$	$\left\langle \begin{array}{l} \hat{a}.elem \in Vect3D \wedge \\ \hat{a}.next \in Cons \end{array} \right\rangle$	$\left\{ \begin{array}{l} a{\geq}3, b{\geq}2 \\ a{>}b \end{array} \right\}$

Figure 3. Generic list example

system only deals with integer primitive types, we use the cost model presented in Sect. 4.3 with the constants S_{int} and S_{ref} to denote the basic sizes for integers and reference types, respectively. Also note that we provide the Java source code instead of the bytecode just for clarity and space limitations. The analyzer works directly on the bytecode which can be found in the appendix.

5.1 Constant Heap Space Usage

In the first example we consider a method with constant heap space usage, i.e., its heap consumption does not depend on any input argument. Fig. 2 shows both the source code and the heap space cost equations generated by the analyzer. The program implements a data hierarchy which will be used throughout the section. It consists of an abstract class, *Data* and two subclasses, *Polynomial* and *Vector3D*. The class *Polynomial* defines a polynomial expression of degree up to 10 with integer coefficients, the coefficients are stored in the array field *coefs* and the degree in the integer field *deg*. Its *copy* method returns a deep copy of the corresponding polynomial by creating a new array of 11 integers and copying the first $deg{+}1$ original coefficients. The class *Vector3D* represents an integer vector with 3 dimensions. The class *Results* stores 25 objects of type *Data*, which in execution time will be *Polynomial* or *Vector3D* objects. Its *copy* method produces a deep copy of the whole structure where each of the 25 elements is copied by its corresponding *copy* method (hence dynamically resolved).

The cost equations generated by the analyzer for the method *Results.copy* are shown in Fig. 2 (at the bottom left). The first equation $C_{copy}(a)$ defines the heap consumption of the method in terms of its first argument a which corresponds to the abstraction of its *this* reference variable (i.e., its size). It counts the heap space allocated for the creation of an object of type *Results*, namely S_{ref}; the space allocated by its constructor, namely $25{*}S_{ref}$; and the space allocated when executing the loop. The heap space allocated by the loop is captured by C_0 and it depends on the type of the object at the current position of the array (which is specified in the guards by checking the class of $\hat{a}.rs[i]$) such that the call to its corresponding *copy* method contributes $S_{int}+S_{ref}{+}11{*}S_{int}$ if it is an instance of *Polynomial* and $3{*}S_{int}$ if it is an instance of *Vector3D*.

As already mentioned, a further issue is how to automatically infer *closed form* solutions (i.e., without recurrences) from the generated cost relations. In our examples, we can directly apply the method of [1] to compute an upper bound in closed form. However, we will not go into details of this process as it is not a concern of this paper and we will simply show the asymptotic complexity that can be directly obtained from such upper bounds. We can observe from the equations that the asymptotic complexity is $O(1)$, as equation C_{copy} is a constant plus C_0, and C_0 is called a constant number of times (in this case 25 times). By assuming that $S_{int}=4$ and $S_{ref}=4$, we can obtain the following upper bound $C_{copy}=4+25*4+25*52=1404$.

5.2 Bounds Proportional to the Input Data Size

For the second example, we consider a generic data structure of type *List*. Both the source code and the heap space cost equations obtained by our analyzer are depicted in Fig. 3. The list is implemented taking advantage of the polymorphism as in the style of the example in Sect. 1, but in this case the elements of the list are objects extending from *Data*

Java source code

```
class Score{
  private int gt1, gt2;
  public Score() {
    gt1 = 0;
    gt2 = 0;
  }
class Scoreboard{
  private Score[][][] scores;
public Scoreboard(int a,int b) {
  scores = new Score[a][][];
  for (int i = 1;i <= a;i++) {
    scores[i-1] = new Score[i][];
    for (int j = 0;j < (i-1);j++) {
      scores[i-1][j] = new Score[b];
      for (int k = 0;k < b;k++)
        scores[i-1][j][k] = new Score();}}}}
```

Heap space cost equations

Equation			Guard	Size rels.
$C_{<init>}(a,b)$	=	$a{*}S_{ref} + C_1(a,b,1)$		
$C_1(a,b,i)$	=	$i{*}S_{ref} + C_2(b,i,0) + C_1(a,b,d)$		$\{i{\leq}a,\ d{=}i{+}1\}$
$C_1(a,b,i)$	=	0		$\{i > a\}$
$C_2(b,i,j)$	=	$b{*}S_{ref} + C_3(b,0) + C_2(b,i,d)$		$\{j{<}(i{-}1),\ d{=}j{+}1\}$
$C_2(b,i,j)$	=	0		$\{j \geq (i{-}1)\}$
$C_3(b,k)$	=	$2{*}S_{int} + C_3(b,c)$		$\{k{<}b,\ c{=}k{+}1\}$
$C_3(b,k)$	=	0		$\{k \geq b\}$

Figure 4. Multi-dimensional arrays example

(see the classes in Fig. 2) rather than integer primitive types. The *List.copy* method returns a deep copy of the list which, in addition to copying the whole list structure, it copies each element by using the corresponding *Data.copy* method (resolved at execution time).

At the bottom of Fig. 3, we show the heap space cost equations our analyzer generates for the method *List.copy* of class *Cons*. The equation $C_{copy}(a)$ defines the heap consumption of the whole method in terms of its first argument a which represents the size of its *this* reference variable. There are four equations for C_{copy}, two of them (the second and the fourth one) are recursive and correspond to the case in which the rest of the list is not empty, i.e., $\hat{a}.next \in Cons$. Note that, in such recursive equations, the size analysis is able to infer the constraint $a > b$, thus ensuring that recursive calls are made with a strictly decreasing value. The other two equations are constant and correspond to the base case (i.e. the rest of the list is empty). This is abstracted in the size relations with the constraint $a = 2$. Note that the heap usage depends on whether we invoke the *copy* method of a *Polynomial* or a *Vector3D* object. By considering the worst cases for all equations, we can infer the upper bound $C_{copy}(a){\leq}(5{*}S_{ref}{+}15{*}S_{int}){*}a \equiv O(a)$ which describes a heap consumption linear in a, the size of the list.

5.3 Multi-Dimensional Arrays

Let us consider the example in Fig. 4. The class *Scoreboard* is instrumental to show how our heap space analysis deals with complex multi-dimensional array creation. The class has a 3-dimensional array field. The constructor takes two integers a and b and creates an array such that: the first dimension is a; the second dimension ranges from 1 to a; and the third dimension is b. Each array entry $scores[i][j][k]$ stores an object of type *Score*.

At the bottom of Fig. 4 we can see the heap space cost equations generated by the analyzer for the constructor of class *Scoreboard*. The equation $C_{<init>}(a,b)$ represents the heap space consumption of the constructor where a and b correspond to the size of its input parameters. It counts the heap consumed by constructing the first array dimension, $a{*}S_{ref}$, plus the heap consumption when executing the outermost loop which is represented by the call $C_1(a,b,1)$. The heap consumption modeled by C_1 includes the amount of heap allocated for the second array dimension in each iteration, $i{*}S_{ref}$, and the consumption of executing the middle loop which is represented by the call $C_2(b,i,0)$. Note that size analysis infers that within C_1, the value of i increases by 1 at each iteration ($d{=}i{+}1$) until it converges to a ($i{\leq}a$). The equation C_2 defines the heap consumption of the middle loop, which includes the heap allocated for the third array dimension, $b{*}S_{ref}$, plus the consumption of executing the innermost loop which is represented by the call $C_3(b,0)$. Finally, C_3 models the heap space required for creating $b{-}k$ *Score* objects by the innermost loop. In this case, we can infer the upper bound $C_{<init>}(a,b) \leq (((2{*}S_{int}{*}b) + b{*}S_{ref}){*}a + a{*}S_{ref}){*}a + a{*}S_{ref} \equiv O(b{*}a^2)$.

5.4 Complex Data Structures

For the last example, let us consider a more complex tree-like data structure which is depicted in Fig. 5. The class *MultiBST* implements a binary search tree data structure where each node has an object of type List (from Fig. 3) and two successors of type *MultiBST* which correspond to the right and left branches of the tree. The constructor method creates an empty tree whose *data* field is initialized to an empty list, i.e., an instance of class *Nil*. The *copy* method performs a deep copy of the whole tree by relying on the *copy* method of class *List*.

Java source code

```
class BST {
  private List data;
  private MultiBST lc;
  private MultiBST rc;
  public MultiBST() {
    data = new Nil();
    lc = null; rc = null; }

public MultiBST copy() {
  MultiBST aux = new MultiBST();
  aux.data = data.copy();
  if (l==null) aux.lc=null; else aux.lc=lc.copy();
  if (r==null) aux.rc=null; else aux.rc=rc.copy();
  return aux;}}
```

Heap space cost equations

Equation	Guard	Size rels.
$C(a) = 3{*}S_{ref} + D(d)$	$\langle \hat{a}.lc = null, \hat{a}.rc = null \rangle$	$\{a{>}0, a{>}d\}$
$C(a) = 3{*}S_{ref} + D(d) + C(l)$	$\langle \hat{a}.lc \neq null, \hat{a}.rc = null \rangle$	$\{a{>}0, a{>}d, a{>}l\}$
$C(a) = 3{*}S_{ref} + D(d) + C(r)$	$\langle \hat{a}.lc = null, \hat{a}.rc \neq null \rangle$	$\{a{>}0, a{>}d, a{>}r\}$
$C(a) = 3{*}S_{ref} + D(d) + C(l) + C(r)$	$\langle \hat{a}.lc \neq null, \hat{a}.rc \neq null \rangle$	$\{a{>}0, a{>}d, a{>}l, a{>}r\}$

Figure 5. Multi binary search tree example

The heap space cost equations generated by the analyzer for the method *MultiBST.copy* are depicted in Fig. 5 (at the bottom). The equations defining $C(a)$ represent the heap space usage of the whole method in terms of the parameter a, which corresponds to the maximal path-length in the tree. There are four cases which correspond to the different possible values for the left and right branches (equal or different from null). Consider for example the last equation: $3 * S_{ref}$ is the heap allocated by the new instruction; $D(d)$ is the heap consumption for copying an object of type *List* which corresponds to C_{copy} from Fig. 3; $C(l)$ and $C(r)$ correspond to the heap consumption of copying the left and right branches respectively. From the cost relation, we infer the upper bound $C(a){\leq}(3{*}S_{ref}{+}D(a)){*}2^a$ where $D(a)$ corresponds to the cost of copying the *data* field (see Sec. 5.2).

6. Active Heap Space with Garbage Collection

One of the safety principles in the Java language is ensured by the use of a garbage collector which avoids errors by the programmer related to deallocation of objects from the heap. The aim of this section is to furnish the heap usage cost relations with *safe annotations* which mark the heap space that will be deallocated by the garbage collector upon exit from the corresponding method. The annotations are then used to infer heap space upper bounds for methods upon exit.

In order to generate such annotations, we rely on the use of *escape analysis* (see, e.g., [8, 15]). Essentially, we assume that the heap allocation instructions `new`, `newarray` and `anewarray` have been respectively transformed by new instructions `new_gc`, `newarray_gc` and `anewarray_gc` as long as it is guaranteed that the lifetime of the corresponding allocated heap space does not exceed the instruction's static scope. In this case, the heap space can be safely deallocated upon exit from the corresponding method. This preprocessing transformation can be done in a straightforward way by using the information inferred by escape analysis. In the following, we refer by *transformed* bytecode instructions to the above transformation performed on the heap allocation instructions. Also, we use $gc(H)$ to denote that the heap space H will safely be garbage collected upon exit from the corresponding method (according to escape analysis) and $ngc(H)$ to denote that it might not be garbage collected.

DEFINITION 6.1. *We define a cost model for heap space with garbage collection which takes a* transformed *bytecode instruction* bc *and returns a positive symbolic expression as:*

$$\mathcal{M}^{gc}_{heap}(\mathtt{bc}) = \begin{cases} ngc(size(\mathtt{Class})) & \text{if } bc{=}\mathtt{new(Class,\_)} \\ gc(size(\mathtt{Class})) & \text{if } bc{=}\mathtt{new\_gc(Class,\_)} \\ ngc(S_{\mathtt{PrimType}} * \mathtt{L}) & \text{if } bc{=}\mathtt{newarray(PrimType,L,\_)} \\ gc(S_{\mathtt{PrimType}} * \mathtt{L}) & \text{if } bc{=}\mathtt{newarray\_gc(PrimType,L,\_)} \\ ngc(S_{\mathtt{ref}} * \mathtt{L}) & \text{if } bc{=}\mathtt{anewarray(Class,L,\_)} \\ gc(S_{\mathtt{ref}} * \mathtt{L}) & \text{if } bc{=}\mathtt{anewarray\_gc(Class,L,\_)} \\ 0 & \textit{otherwise} \end{cases}$$

where $S_{\mathtt{PrimType}}$, $S_{\mathtt{ref}}$ *and* $size()$ *are as in Def. 4.5.* □

The above cost model returns a *symbolic* positive expression which contains the annotations gc and ngc as described above. Therefore, when generating the heap space cost relations as described in Def. 4.6 w.r.t. $\mathcal{M}^{gc}_{heap}$, the cost relations will be of the following form:

$$C(\bar{x}) = gc(H_{gc}) + ngc(H_{ngc}) + \sum C_r(\bar{z}) \quad \varphi$$

where we assume that all symbolic expressions wrapped by gc (resp. by ngc) are grouped together within each cost equation and denote the total heap space H_{gc} that will be garbage collected (resp. H_{ngc} which might not be) after the application of such equation.

EXAMPLE 6.2. *Suppose we add the following methods*

```
abstract List map(Func o);                    // List
List map(Func o) { return this; }             // Nil
List map(Func o) {                            // Cons
    List tail = this.next.map(o);
    Cons head = new Cons();
    head.next = tail;
    head.elem = o.f(new Integer(this.elem));
    return head;
}
```

respectively to the classes List*,* Nil *and* Cons *which are depicted in Fig. 1. The method* map *clones the corresponding list structure, but the value of the field* elem *in the clone is the result of applying the method* o.f *on the corresponding value in the cloned list. Note that the method* o.f*, takes as input an object of type* Integer*, therefore* this.elem *(which is of type* int*) is first converted to* Integer *by creating a temporary corresponding* Integer *object. For simplicity, we do not give a specific definition for* Func*, but we assume that its method f (which is called using* o.f*) does not allocate any heap space and that it returns a value of type* int*. Using* escape analysis*, the creation of the temporary* Integer *object can be annotated as local to* map*, therefore we replace the corresponding* `new` *instruction by* `new_gc`*. Assuming that the size of an* Integer *object is 4 bytes, and using the cost model of Def. 6.1, we obtain the following cost equations:*

Equation	*Size relations*
$C^{Nil}_{map}(a) = 0$	$\{a{=}1\}$
$C^{Cons}_{map}(a) = gc(4) + ngc(8)$	$\{a{=}2\}$
$C^{Cons}_{map}(a) = gc(4) + ngc(8) + C^{Cons}_{map}(b)$	$\{a{\geq}3, b{\geq}1, a{=}b{+}1\}$

The symbolic expression $gc(4)$ in the above equations corresponds to the heap space allocated for the temporary Integer *object which can be garbage collected upon exit from* map*, and $ngc(8)$ corresponds to the heap space allocated for the* Cons *object. As before,* a *corresponds to the size of the* this *reference variable (i.e., the list length) and* b *to* this.next. □

Using the refined cost relations we can infer different information about the heap space usage depending on the interpretation given to the gc and ngc annotations. Let us first consider the following definitions:

$$\boxed{\forall\, H,\, gc(H) = 0 \text{ and } ngc(H) = H} \tag{1}$$

where we do not count the heap space that will be deallocated upon exit from the corresponding method. By applying Eq. (1) to a cost relation C_m of a method m, we can infer an upper bound U^{gc}_m of the *active heap space* upon the exit from m, i.e., the heap space consumed by m which might not be deallocated upon exit. In this setting, for the cost relations of Ex. 6.2 we infer the closed form $U^{gc}_{map} \equiv C^{Cons}_{map}(a) = 8 * (a-1)$. It is important to note that, in general, such upper bound does not ensure that the heap space required for executing m does not exceed U^{gc}_m, i.e., it is not an upper bound of the heap usage *during* the execution of m but rather only *after* its execution. Actually, in this simple example, we can observe already that during the execution of the method *map*, if all objects are heap allocated, we need more than $8 * (a-1)$ heap units (as the objects of type *Integer* will be heap allocated and they are not accounted in the upper bound). However, one of the applications of escape analysis is to determine which objects can be stack allocated instead of heap allocated in order to avoid invoking the garbage collector which is time consuming [8]. For instance, in the above example, the objects of type *Integer* can be safely stack allocated. When this stack allocation optimization is performed, then U^{gc}_m is indeed an upper bound for the heap space required to execute m.

In order to infer upper bounds for the heap space required during the execution of m, we define gc and ngc as follows:

$$\boxed{\forall\, H,\, gc(H) = H \text{ and } ngc(H) = H} \tag{2}$$

In this case, we obtain the same cost relations as in Def. 4.6 which correspond to the worst case heap usage in which we do not discount any deallocation by the garbage collector. In this setting, for the cost relation of Ex. 6.2 we infer the closed form $U_{map}(a) \equiv C^{Cons}_{map}(a) = 12 * (a-1)$.

Analysis for finding upper bounds on the memory high-watermark cannot be directly done using cost relations as introducing decrements in the equations requires computing lower bounds. As a further issue, the *active heap space* upper bound, U^{gc}_m, can be used to improve the accuracy of the upper bound on the heap space required for executing a sequence of method calls. For example, an upper bound of the heap space required for executing a method m_1 and upon its return immediately executing a method m_2 can be approximated by $max(U_{m_1}, U^{gc}_{m_1} + U_{m_2})$ which is more precise than taking $U_{m_1} + U_{m_2}$ as it takes into account that after executing m_1 we can apply garbage collection and only then executing m_2. This idea is the basis for a post-processing that could be done on the program in order to obtain more accurate upper bounds on the heap usage at a *program point* level. This is a subject of ongoing research.

7. Experiments

In order to assess the practicality of our heap space analysis, we have implemented a prototype inter-procedural analyzer in `Ciao` [10] as an extension of the one in [3]. We still have not incorporated an escape analysis in our implementation and hence the upper bounds inferred correspond to those generated using Eq. (2) of Sect. 6. The experiments have been performed on an Intel P4 Xeon 2 GHz with 4 GB of RAM, running GNU Linux FC-2, 2.6.9. Table 1 shows the run-times of the different phases of the heap space analysis process. The name of the main class to be analyzed is given in the first column, *Benchmark*, and its size (the sum of all its *class* file sizes) in KBytes is given in the second column, *Size*. Columns 3-6 shows the runtime of the different phases in milliseconds, they are computed as the arithmetic mean of five runs: *RR* is the time for obtaining the recursive representation (building CFG, eliminating stack elements, etc., as outlined in Sec. 4.1); *Size An.* is the time for the abstract-interpretation based size analysis for computing size relations; *Cost* is the time taken for building the heap space cost relations for the different blocks and representing them in a simplified form; and *Total* shows the total times of the whole analysis process. In the last column, *Complexity*, we depict the asymptotic complexity of the (worst-case) heap space cost obtained from the cost relations.

Benchmark	Size	RR	Size An.	Cost	Total	Complexity	
ListInt	0.86	24	53	7	83	$O(n)$	$n \equiv$ list length
Results	1.31	83	275	15	374	$O(1)$	–
BSTInt	0.48	37	113	5	156	$O(2^n)$	$n \equiv$ tree depth
List	1.79	71	207	16	293	$O(n)$	$n \equiv$ list length
Queue	1.93	219	570	24	813	$O(n)$	$n \equiv$ queue length
Stack	1.38	89	643	17	749	$O(n)$	$n \equiv$ stack length
BST	1.43	97	238	14	349	$O(2^n)$	$n \equiv$ tree depth
Scoreboard	0.65	280	1539	12	1830	$O(a^2 * b)$	$\{a, b\} \equiv$ input args.
MultiBST	2.35	166	510	34	709	$O(n * 2^n)$	$n \equiv$ tree depth

Table 1. Measured time (in ms) of the different phases of cost analysis

Regarding the benchmarks we have used, on one hand, we have benchmarks implementing some classic data structures using an object-oriented programming style, which expose the analyzer's ability in handling such classical data structures as well as sophisticated object-oriented programming features. In particular, *ListInt*, *List*, *Queue*, *Stack*, *BSTInt*, *BST* and *MultiBST* implement respectively integer and generic lists, generic queues, generic stacks, integer and generic binary search trees which allow data repetitions. On the other hand, we have some benchmarks which expose more particular issues of heap space analysis, such as *Results* which has constant heap space usage and *Scoreboard* which presents a multidimensional arrays creation. For all benchmarks, we have analyzed the corresponding *copy* method which performs a deep copy of the corresponding structure.

We can observe in the table that computing size relations is the most expensive step as it requires a global analysis of the program, whereas *RR* and *Cost* basically involve a single pass on the code. Our prototype implementation supports the full instructions set of sequential Java bytecode, however, it is still preliminary, and there is plenty of room for optimization, mainly in the size analysis phase, which in addition assumes the absence of cyclic data structures, which can be verified using the non-cyclicity analysis [23].

8. Conclusions and Related Work

We have presented an automatic analysis of heap usage for Java bytecode, based on generating at compile-time cost relations which define the heap space consumption of an input bytecode program. By means of a series of examples which allocate lists, trees, trees of lists, arrays, etc. in the heap, we have shown that our analysis is able to infer non-trivial bounds for them (including polynomial and exponential complexities). We believe that the experiments we have presented show that our analysis improves the state of the practice in heap space analysis of Java bytecode.

Related work in heap space analysis includes advanced techniques developed in functional programming, mainly based on type systems with resource annotations (see, e.g., [24, 17, 25, 19]) and, hence, they are quite different technically to ours. But heap space analysis is comparatively less developed for low-level languages such as Java bytecode. A notable exception is the work in [11], where a memory consumption analysis is presented. In contrast to ours, their aim is to verify that the program executes in bounded memory by simply checking that the program does not create new objects inside loops, but they do not infer bounds as our analysis does. Moreover, it is straightforward to check that new objects are not created inside loops from our cost relations. Another related work includes research in the MRG project [5, 7], which focuses on building a proof-carrying code [22] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds. The analysis is developed for a functional language which then compiles to a (subset of) Java bytecode and it is restricted to linear bounds. In [6] the Bytecode Specification Language is used to annotate Java bytecode programs with memory consumption behaviour and policies, and then verification tools are used to verify those policies.

For Java-like languages, the work of [18] presents a type system for heap analysis without garbage collection, it is developed at the level of the source code and based on amortised analysis (hence it is technically quite different to our work) and, unlike us, they do not present an inference method for heap consumption. On the other hand, the work of [9] deals also with Java source code, it is able to infer polynomial complexity though it does not handle recursion.

Some works consider explicit deallocation of objects by decreasing the cost by the size of the deallocated object (see, e.g., [18, 17]). This approach is interesting when one wants to observe the heap consumption at certain program points. However, it cannot be directly incorporated in our cost relations because they are intended to provide a *global* upper bound of a method's execution. Naturally, it should happen that allocated objects are correctly deallocated and hence our cost relations would provide zero as (global) upper bound. Other work which considers cost with garbage collection is [24]. Unlike ours, it is developed for pure functional programs where the garbage collection behaviour is easier to predict as programs do not have assignments.

In the future, we want to extend our work in several directions. On the practical side, we want to incorporate an escape

analysis to transform the bytecode as outlined in Sect. 6. Regarding scalability, it is a question of performance vs. precision trade-off and depends much on the underlying abstract domain used by the size analysis. We believe our analysis would scale without sacrificing precision if an efficient domain like octagons is used together with [1]. On the theoretical side, we plan to adapt our analysis to infer upper bounds on the heap usage at given program points in the presence of garbage collection. We also would like to develop an analysis which infers upper bounds on the call stack usage.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Cost Equation Systems. *Submitted*, 2007.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *16th European Symposium on Programming, ESOP'07*, Lecture Notes in Computer Science. Springer, March 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *Proc. of BYTECODE'07*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2007.

[4] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, number 3452 in LNAI, pages 380–397. Springer-Verlag, 2005.

[5] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, number 3362 in LNCS. Springer, 2005.

[6] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 86–95. IEEE Computer Society, 2005.

[7] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *Proc. of LPAR'04*, LNCS 3452, pages 347–362. Springer, 2004.

[8] Bruno Blanchet. Escape Analysis for Java$^{\text{tm}}$: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, 2003.

[9] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

[10] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[11] D. Cachera, D. Pichardie T. Jensen, and G. Schneider. Certified memory usage analysis. In *FM'05*, number 3582 in LNCS. Springer, 2005.

[12] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing resource bounds via static verification of dynamic checks. In *Proc. of ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2005.

[13] W. Chin, H. Nguyen, S. Qin, and M. Rinard. Memory Usage Verification for OO Programs. In *Proc. of SAS'05*, LNCS 3672, pages 70–86. Springer, 2005.

[14] K. Crary and S. Weirich. Resource bound certification. In *Proc. of POPL'00*, pages 184–198. ACM Press, 2000.

[15] Patricia M. Hill and Fausto Spoto. Deriving Escape Analysis by Abstract Interpretation. *Higher-Order and Symbolic Computation*, (19):415–463, 2006.

[16] M. Hofmann. Certification of Memory Usage. In *Theoretical Computer Science, 8th Italian Conference, ICTCS*, volume 2841 of *Lecture Notes in Computer Science*, page 21. Springer, 2003.

[17] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *30th ACM Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM Press, 2003.

[18] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[19] J. Hughes and L. Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *Proc. of ICFP'99*, pages 70–81. ACM Press, 1999.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[21] Patricia M.Hill, Etienne Payet, and Fausto Spoto. Path-length analysis of object-oriented programs. In *Proc. EAAI*, 2006.

[22] G. Necula. Proof-Carrying Code. In *POPL'97*. ACM Press, 1997.

[23] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of the 7th workshop on Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110, Charleston, SC, USA, January 2006. Springer-Verlag.

[24] L. Unnikrishnan, S. Stoller, and Y. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, Lecture Notes in Computer Science, pages 70–85. Springer, 2003.

[25] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.

Live Heap Space Analysis for Languages with Garbage Collection

Elvira Albert
Complutense University of Madrid
elvira@sip.ucm.es

Samir Genaim
Complutense University of Madrid
samir.genaim@fdi.ucm.es

Miguel Gómez-Zamalloa
Complutense University of Madrid
mzamalloa@fdi.ucm.es

Abstract

The *peak heap consumption* of a program is the maximum size of the *live* data on the heap during the execution of the program, i.e., the minimum amount of heap space needed to run the program without exhausting the memory. It is well-known that garbage collection (GC) makes the problem of predicting the memory required to run a program difficult. This paper presents, the best of our knowledge, the first *live heap space* analysis for garbage-collected languages which infers accurate upper bounds on the peak heap usage of a program's execution that are not restricted to any complexity class, i.e., we can infer exponential, logarithmic, polynomial, etc., bounds. Our analysis is developed for an (sequential) object-oriented bytecode language with a *scoped-memory* manager that reclaims unreachable memory when methods return. We also show how our analysis can accommodate other GC schemes which are closer to the *ideal* GC which collects objects as soon as they become unreachable. The practicality of our approach is experimentally evaluated on a prototype implementation. We demonstrate that it is fully automatic, reasonably accurate and efficient by inferring live heap space bounds for a standardized set of benchmarks, the JOlden suite.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]: General; D3.2 [*Programming Languages*]

General Terms Languages, Theory, Verification, Reliability

Keywords Live Heap Space Analysis, Peak Memory Consumption, Low-level Languages, Java Bytecode

1. Introduction

Predicting the memory required to run a program is crucial in many contexts like in embedded applications with stringent space requirements or in real-time systems which must respond to events or signals within a predefined amount of time. It is widely recognized that memory usage estimation is important for an accurate prediction of running time, as cache misses and page faults contribute directly to the runtime. Another motivation is to configure real-time garbage collectors to avoid mutator starvation. Besides, upper bounds on the memory requirement of programs have been proposed for resource-bound certification [10] where certificates encode security properties involving resource usage requirements, e.g., the (untrusted) code must adhere to specific bounds on its memory usage. On the other hand, automatic memory management (also known as garbage collection) is a very powerful and useful mechanism which is increasingly used in high-level languages such as Java. Unfortunately, GC makes the problem of predicting the memory required to run a program difficult.

A first approximation to this problem is to infer the *total memory allocation*, i.e., the *accumulated* amount of memory allocated by a program ignoring GC. If such amount is available it is ensured that the program can be executed without exhausting the memory, even if no GC is performed during its execution. However, it is an overly pessimistic estimation of the actual memory requirement. *Live heap space analysis* [18, 5, 8] aims at approximating the size of the *live* data on the heap during a program's execution, which provides a much tighter estimation. This paper presents a general approach for inferring the *peak heap consumption* of a program's execution, i.e., the maximum of the live heap usage along its execution. Our live heap space analysis is developed for (an intermediate representation of) an object-oriented *bytecode* language with automatic memory management. Programming languages which are compiled to bytecode and executed on a *virtual machine* are widely used nowadays. This is the approach used by Java bytecode and .NET.

Analysis of live heap usage is different from total memory allocation because it involves reasoning on the memory consumed at *all program states* along an execution, while total allocation needs to observe the consumption at the *final* state only. As a consequence, the classical approach to static cost analysis proposed by Wegbreit in 1975 [20] has been applied only to infer total allocation. Intuitively, given a program, this approach produces a *cost relation system* (CR for short) which is a set of recursive equations that capture the cost *accumulated* along the program's execution. Symbolic closed-form solutions (i.e., without recursion) are found then from the CR. This approach leads to very accurate cost bounds as it is not limited to any complexity class (infers polynomial, logarithmic, exponential consumption, etc.) and, besides, it can be used to infer different notions of resources (total memory allocation, number of executed instructions, number of calls to specific methods, etc.). Unfortunately, it is not suitable to infer peak heap consumption because it is not an accumulative resource of a program's execution as CR capture. Instead, it requires to reason on all possible states to obtain their maximum. By relying on different techniques which do not generate CR, live heap space analysis is currently restricted to polynomial bounds and non-recursive methods [5] or to linear bounds dealing with recursion [8].

Inspired by the basic techniques used in cost analysis, in this paper, we present a general framework to infer accurate bounds on the peak heap consumption of programs which improves the state-of-the-art in that it is not restricted to any complexity class and deals with all bytecode language features including recursion. To pursue our analysis, we need to characterize the behavior of

ISMM'09, June 19–20, 2009, Dublin, Ireland.

the underlying garbage collector. We assume a standard *scoped-memory* manager that reclaims memory when methods return. In this setting, our main contributions are:

1. *Escaped Memory Analysis.* We first develop an analysis to infer upper bounds on the *escaped memory* of method's execution, i.e., the memory that is allocated during the execution of the method *and* which remains upon exit. The key idea is to infer first an upper bound for the total memory allocation of the method. Then, such bound can be manipulated, by relying on information computed by *escape analysis* [4], to extract from it an upper bound on its escaped memory.
2. *Live Heap Space Analysis.* By relying on the upper bounds on the escaped memory, as our main contribution, we propose a novel form of *peak consumption* CR which captures the peak memory consumption over all program states along the execution for the considered scoped-memory manager. An essential feature of our CRs is that they can be solved by using existing tools for solving standard CRs.
3. *Ideal Garbage Collection.* An interesting, novel feature of our approach is that we can refine the analysis to accommodate other kinds of scope-based managers which are closer to an *ideal* garbage collector which collects objects as soon as they become unreachable.
4. *Implementation.* We report on a prototype implementation which is integrated in the COSTA system [2] and experimentally evaluate it on the JOlden benchmark suite. Preliminary results demonstrate that our system obtains reasonably accurate live heap space upper bounds in a fully automatic way.

2. Bytecode: Syntax and Semantics

Bytecode programs are complicated for both human and automatic analysis because of their unstructured control flow, operand stack, etc. Therefore, it is customary to formalize analyses on intermediate representations of the bytecode (e.g., [3, 19, 13]). We consider a rule-based *procedural* language (in the style of any of the above) in which a *rule-based program* consists of a set of *procedures* and a set of classes. A procedure p with k input arguments $\bar{x} = x_1, \ldots, x_k$ and m output arguments $\bar{y} = y_1, \ldots, y_m$ is defined by one or more *guarded rules*. Rules adhere to the following grammar:

$$
\begin{array}{rl}
\mathit{rule} & ::= p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) ::= g, b_1, \ldots, b_t \\
g & ::= \mathit{true} \mid \mathit{exp}_1 \ \mathit{op} \ \mathit{exp}_2 \mid \mathsf{type}(x, c) \\
b & ::= x := \mathit{exp} \mid x := \mathsf{new}\ c^i \mid x := y.f \mid x.f := y \mid q(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \\
\mathit{exp} & ::= \mathsf{null} \mid \mathit{aexp} \\
\mathit{aexp} & ::= x \mid n \mid \mathit{aexp} - \mathit{aexp} \mid \mathit{aexp} + \mathit{aexp} \mid \mathit{aexp} * \mathit{aexp} \mid \mathit{aexp} / \mathit{aexp} \\
\mathit{op} & ::= > \mid < \mid \leq \mid \geq \mid = \mid \neq
\end{array}
$$

where $p(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ is the *head* of the rule; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_t$ the body of the rule; n an integer; x and y variables; f a field name, and $q(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ a procedure call by value. The language supports class definition and includes instructions for object creation, field manipulation, and type comparison through the instruction $\mathsf{type}(x, c)$, which succeeds if the runtime class of x is exactly c. A class c is a finite set of typed field names, where the type can be integer or a class name. The superscript i on a class c is a unique identifier which associates objects with the program points where they have been created. The key features of this language are: (1) *recursion* is the only iterative mechanism, (2) *guards* are the only form of conditional, (3) there is no operand stack, (4) objects can be regarded as records, and the behavior induced by dynamic dispatch in the original bytecode program is compiled into *dispatch* blocks

```
class Test {
  static Tree m(int n) {
    if ( n>0 ) return new
      Tree(m(n-1),m(n-1),f(n));
    else return null;
  }
  static int f(int n) {
    int a=0,i=n;
    while ( n>1 ) {
      a += g(n).intValue();
      n=n/2;
    }
    for(; i>1; i=i/2)
      a *= h(i).intValue();
    return a;
  }
  static Integer g(int n) {
    Integer x=new Integer(n);
    return new Integer(x.intValue()+1);
  }
  static Long h(int n) {
    return new Long(n-1);
  }
} // end of class Test

class Tree {
  Tree l,r;
  int d;
  Tree(Tree l,Tree r,int d) {
    this.l = l;
    this.r = r;
    this.d = d;
  }
}
```

Figure 1. Java code of running example

(1) $\mathrm{m}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::=$
$\mathbf{n > 0}$,
$\mathrm{s}_0 := \mathsf{new}\ \mathrm{Tree}^1$;
$\mathrm{s}_1 := \mathrm{n} - 1$,
$\mathrm{m}(\langle \mathrm{s}_1 \rangle, \langle \mathrm{s}_1 \rangle)$,
$\mathrm{s}_2 := \mathrm{n} - 1$,
$\mathrm{m}(\langle \mathrm{s}_2 \rangle, \langle \mathrm{s}_2 \rangle)$,
$\mathrm{f}(\langle \mathrm{n} \rangle, \langle \mathrm{s}_3 \rangle)$,
$\mathrm{init}(\langle \mathrm{s}_0, \mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3 \rangle, \langle \rangle)$,
$\mathrm{r} = \mathrm{s}_0$.

(2) $\mathrm{m}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::=$
$\mathbf{n \leq 0}$,
$\mathrm{r} := \mathsf{null}$.

(3) $\mathrm{f}(\langle \mathrm{n} \rangle, \langle \mathrm{a} \rangle) ::=$
$\mathrm{a} := 0$,
$\mathrm{i} := \mathrm{n}$,
$\mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle)$,
$\mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle)$.

(4) $\mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle) ::=$
$\mathbf{n > 1}$,
$\mathrm{g}(\langle \mathrm{n} \rangle, \langle \mathrm{s}_0 \rangle)$,
$\mathrm{intValue}_1(\langle \mathrm{s}_0 \rangle, \langle \mathrm{s}_0 \rangle)$
$\mathrm{a} := \mathrm{a} + \mathrm{s}_0$,
$\mathrm{n} := \mathrm{n}/2$,
$\mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle)$.

(5) $\mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle) ::=$
$\mathbf{n \leq 1}$.

(6) $\mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle) ::=$
$\mathbf{i > 1}$,
$\mathrm{h}(\langle \mathrm{i} \rangle, \langle \mathrm{s}_0 \rangle)$,
$\mathrm{intValue}_2(\langle \mathrm{s}_0 \rangle, \langle \mathrm{s}_0 \rangle)$
$\mathrm{a} := \mathrm{a} * \mathrm{s}_0$,
$\mathrm{i} := \mathrm{i}/2$,
$\mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle)$.

(7) $\mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle) ::=$
$\mathbf{i \leq 1}$.

(8) $\mathrm{g}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::=$
$\mathrm{x} := \mathsf{new}\ \mathrm{Integer}^2$,
$\mathrm{init}_1(\langle \mathrm{x}, \mathrm{n} \rangle, \langle \rangle)$,
$\mathrm{intValue}_1(\langle \mathrm{x} \rangle, \langle \mathrm{s}_0 \rangle)$,
$\mathrm{s}_0 := \mathrm{s}_0 + 1$.
$\mathrm{r} := \mathsf{new}\ \mathrm{Integer}^3$,
$\mathrm{init}_1(\langle \mathrm{r}, \mathrm{s}_0 \rangle, \langle \rangle)$.

(9) $\mathrm{h}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::=$
$\mathrm{s}_0 := \mathrm{n} - 1$.
$\mathrm{r} := \mathsf{new}\ \mathrm{Long}^4$,
$\mathrm{init}_2(\langle \mathrm{r}, \mathrm{s}_0 \rangle, \langle \rangle)$.

(10) $\mathrm{init}(\langle \mathrm{this}, \mathrm{l}, \mathrm{r}, \mathrm{d} \rangle, \langle \rangle) ::=$
$\mathrm{this.l} := \mathrm{l}$,
$\mathrm{this.r} := \mathrm{r}$,
$\mathrm{this.d} := \mathrm{d}$.

Figure 2. Intermediate representation of running example.

guarded by a type check, and (5) procedures may have *multiple return* values. The translation from (Java) bytecode to the rule-based form is performed in two steps. First, a *control flow graph* (CFG) is built. Second, a *procedure* is defined for each basic block in the graph and the operand stack is *flattened* by considering its elements as additional local variables. E.g., this translation is explained in more detail in [3]. For simplicity, our language does not include advanced features of Java bytecode, such as exceptions, interfaces, static methods and fields, access control (e.g., the use of public, protected and private modifiers) and primitive types besides integers and references. Such features can be easily handled in our framework and indeed our implementation deals with full (sequential) Java bytecode.

EXAMPLE 2.1. *Fig. 1 depicts our running example in Java, and Fig. 2 depicts its corresponding rule-based representation where the procedures are named as the method they represent and "*$\mathrm{f_c}$*" and "*$\mathrm{f_d}$*" denote intermediate procedures for* f. *The Java program is included only for clarity as the analyzer generates the rule-based representation from the corresponding bytecode only. As an example, we explain rules (1) and (2) which correspond to method* m. *Each rule is guarded by a corresponding condition, resp.* $\mathbf{n} > \mathbf{0}$ *and* $\mathbf{n} \leq \mathbf{0}$. *Variable names of the form* $\mathrm{s_i}$ *indicate that they originate from stack positions. In rule (1), the "*$\mathsf{new}\ \mathrm{Tree}^1$*" instruction creates an object of type* Tree *(the superscript* 1 *is the unique identifier for this program point) and assigns the variable* $\mathrm{s_0}$ *to its reference (which corresponds to pushing the reference on the stack in the original bytecode). Then, the local variable* n *is decremented by one and the result is assigned to* $\mathrm{s_1}$. *Next, the method* m *is recursively invoked which receives as input argument the result of the previous operation (*$\mathrm{s_1}$*) and returns its result in* $\mathrm{s_1}$. *Similar invocations to methods* m, f *and* init *follow. In Java bytecode, constructor methods are named* init. *In both rules, the return value is* r *which in (1) is assigned to the object reference and in (2) to* null. *It can be observed that, like in bytecode, all guards and instructions correspond to three-address code, except for calls to procedures which may involve more variables as parameters. The methods* $\mathrm{intValue_1}$ *and* $\mathrm{init_1}$ *belong to class* Integer, *and* $\mathrm{intValue_2}$ *and* $\mathrm{init_2}$ *belong to class* Long. □

Observe in the example that, in our syntax, with the aim of simplifying the presentation, we do not distinguish between calls to methods and calls to intermediate procedures. For instance, $\mathrm{f_c}$ and $\mathrm{f_d}$ are intermediate procedures while f is the method. This distinction can be made observable in the translation phase trivially and, when needed, we assume such distinction is available.

2.1 Semantics

The execution of bytecode in rule-based form is exactly like standard bytecode; a thorough explanation is outside the scope of this paper (see [14]). An *operational semantics* for rule-based bytecode is shown in Fig. 3. An *activation record* is of the form $\langle p, bc, tv\rangle$, where p is a procedure name, bc is a sequence of instructions and tv a variable mapping. Executions proceed between *configurations* of the form $A; h$, where A is a stack of activation records and h is the *heap* which is a partial map from an infinite set of *memory locations* to *objects*. We use $h(r)$ to denote the object referred to by the memory location r in h and $h[r \mapsto o]$ to indicate the result of updating the heap h by making $h(r) = o$. An object o is a pair consisting of the object class tag and a mapping from field names to values which is consistent with the type of the fields.

Intuitively, rule (1) accounts for all instructions in the bytecode semantics which perform arithmetic and assignment operations. The evaluation $eval(exp, tv)$ returns the evaluation of the arithmetic or Boolean expression exp for the values of the corresponding variables from tv in the standard way, and for reference variables, it returns the reference. Rules (2), (3) and (4) deal with objects. We assume that $\mathsf{newobject}(c^i)$ creates a new object of class c and initializes its fields to either 0 or null, depending on their types. Rule (5) (resp., (6)) corresponds to calling (resp., returning from) a procedure. The notation $p[\bar{y}, \bar{y}']$ records the association between the formal and actual return variables. It is assumed that newenv creates a new mapping of local variables for the corresponding method, where each variable is initialized as newobject does.

An execution starts from an *initial configuration* of the form $\langle \bot, p(\langle \bar{x}\rangle, \langle \bar{y}\rangle), tv\rangle; h$ and ends when we reach a *final configuration* $\langle \bot, \epsilon, tv'\rangle; h'$ where tv and h are initialized to suitable initial values, tv' and h' include the final values, and $\bot$ is a special symbol indicating an initial state. We assume that any object stored in the initial heap h is reachable from (at least) one of the x_i, namely there are not *collectable* objects that can removed from h at the initial state. Note that $\mathrm{dom}(tv) = \mathrm{dom}(tv') = \bar{x} \cup \bar{y}$. Finite executions can be regarded as *traces* $S_0 \rightsquigarrow S_1 \rightsquigarrow \cdots \rightsquigarrow S_\omega$, denoted $S_0 \rightsquigarrow^* S_\omega$, where S_ω is a final configuration. Infinite traces correspond to non-terminating executions.

3. Total Memory Allocation Analysis

Let us first define the notion of total memory consumption. We let $size(c)$ denote the amount of memory required to hold an instance object of class c, $size(o)$ denotes the amount of memory occupied by an object o, and $size(h)$ denotes the amount of memory occupied by all objects in the heap h, namely $\Sigma_{r \in \mathrm{dom}(h)} size(h(r))$. We consider the semantics in Fig. 3 where no GC is performed. Given a trace $t \equiv A_1; h_1 \rightsquigarrow^* A_n; h_n$, the *total memory allocation* of t is defined as $total(t) = size(h_n) - size(h_1)$.

In this section, we briefly overview the application of the cost analysis framework, originally proposed by Wegbreit [20], to total memory consumption inference of bytecode as proposed in [3]. The original analysis framework [1] takes as input a program and a cost model $\mathcal{M}$, and outputs a closed-form upper bound that describes its execution cost w.r.t. $\mathcal{M}$. The cost model $\mathcal{M}$ defines the cost that we want to accumulate. For instance, if the cost model is the number of executed instructions, $\mathcal{M}$ assigns cost 1 to all instructions. The application of this framework to total memory consumption of bytecode takes as input a bytecode program and the following cost model $\mathcal{M}^t$, which is a simplification for our language of the cost model for heap space usage of [3].

DEFINITION 3.1 (heap consumption cost model [3]). *Given a bytecode instruction b, the heap consumption cost model is defined as*

$$\mathcal{M}^t(b) = \begin{cases} size(c^i) & b \equiv x := \mathsf{new}\ c^i \\ 0 & \textit{otherwise} \end{cases}$$

For a sequence of instructions, $\mathcal{M}^t(b_1 \cdots b_n) = \mathcal{M}^t(b_1) + \cdots + \mathcal{M}^t(b_n)$. □

3.1 Inference of Size Relations

The aim of the analysis is to approximate the memory consumption of the program as an upper bound function in terms of its input *data size*s. As customary, the *size* of data is determined by its variable type: the size of an integer variable is its value; the size of an array is its length; and the size of a reference variable is the length of the longest path that can be traversed through the corresponding object (e.g., length of a list, depth of a tree, etc.). To keep the presentation simple, we use the original variable names (possible primed) to refer to the corresponding abstract (size) variables; but we write the size in *italic* font. For instance, let x be a reference to a tree, then x represents the depth of x. When we need to compute the sizes $\bar{v}$ of a given tuple of variables $\bar{x}$, we use the notation $\bar{v} = \alpha(\bar{\mathrm{x}}, tv, h)$, which means that the integer value v_i is the size of the variable $\mathrm{x_i}$ in the context of the variables table tv and the heap h. For instance, if x is the reference to a tree, we need to access the heap h where the tree is allocated to compute its depth and obtain v. If x is an integer variable, then its size (value) can be obtained from the variable table tv.

Standard *size analysis* is used in order to obtain relations between the sizes of the program variables at different program points. For instance, associated to procedure $\mathrm{f_c}$, we infer the size relation $n' = n/2$ which indicates that the value of n decreases by half when calling $\mathrm{f_c}$ recursively. We denote by φ_r the conjunction of linear constraints which describes the relations between the abstract variables of a rule r and refer to [9, 3] for more information.

$$(1)\quad \frac{b \equiv x := exp,\ \ v = eval(exp, tv)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv[x \mapsto v]\rangle{\cdot}A; h}$$

$$(2)\quad \frac{b \equiv x := \mathsf{new}\ c^i,\quad o{=}\mathsf{newobject}(c^i),\quad r{\notin}dom(h)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv[x \mapsto r]\rangle{\cdot}A; h[r \mapsto o]}$$

$$(3)\quad \frac{b \equiv x := y.f,\ \ tv(y) \neq \mathsf{null},\ \ o = h(tv(y))}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv[x \mapsto o.f]\rangle{\cdot}A; h}$$

$$(4)\quad \frac{b \equiv x.f := y,\ \ tv(x) \neq \mathsf{null},\ \ o = tv(x)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv\rangle{\cdot}A; h[o.f \mapsto tv(y)]}$$

$$(5)\quad \frac{\begin{array}{c} b \equiv q(\langle\bar{x}\rangle, \langle\bar{y}\rangle),\ \text{there is a program rule } q(\langle\bar{x}'\rangle, \langle\bar{y}'\rangle){:=}g, b_1, \cdots, b_k \\ \text{such that } tv'{=}\mathsf{newenv}(q),\ \forall i.tv'(x'_i) = tv(x_i),\ eval(g, tv') = true \end{array}}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle q, b_1 \cdot \ldots \cdot b_k, tv'\rangle{\cdot}\langle p[\bar{y}, \bar{y}'], bc, tv\rangle{\cdot}A; h}$$

$$(6)\quad \frac{}{\langle q, \epsilon, tv\rangle{\cdot}\langle p[\bar{y}, \bar{y}'], bc, tv'\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv'[\bar{y} \mapsto tv(\bar{y}')]\rangle{\cdot}A; h}$$

Figure 3. Operational semantics of bytecode programs in rule-based form

(1)	$m(n){=}size(\mathrm{Tree}^1){+}m(s_1){+}m(s_2){+}$ $f(n){+}init(s_0, s_1, s_2, s_3)$	$\{n{>}0, s_0{=}1,$ $s_1{=}n{-}1, s_2{=}n{-}1\}$
(2)	$m(n){=}0$	$\{n{\leq}0\}$
(3)	$f(n){=}f_c(n, a) + f_d(i, a')$	$\{a{=}0, i{=}n\}$
(4)	$f_c(n, a){=}g(n){+}f_c(n', a')$	$\{n{>}1, n'{=}n/2\}$
(5)	$f_c(n, a){=}0$	$\{n{\leq}1\}$
(6)	$f_d(i, a){=}h(i){+}f_d(i', a')$	$\{i{>}1, i'{=}i/2\}$
(7)	$f_d(i, a){=}0$	$\{i{\leq}0\}$
(8)	$g(n){=}size(\mathrm{Integer}^2){+}size(\mathrm{Integer}^3)$	$\{x{=}1\}$
(9)	$h(n){=}size(\mathrm{Long}^4)$	$\{r{=}1, s_0{=}n{-}1\}$
(10)	$init(this, l, r, d){=}0$	$\{\}$

Figure 4. Total Allocation CR.

3.2 Generation of Cost Relations

In a nutshell, given a bytecode program P, the analysis of [3] proceeds in three steps: (1) it first transforms it into an equivalent rule-based program (our work directly starts from such rule-based form), (2) it infers size relations as explained above, (3) it generates a CR which describes the total memory consumption of the program as follows.

DEFINITION 3.2 (total allocation CR [3]). *Consider a rule* $\mathrm{r} \equiv \mathrm{p}(\langle\bar{\mathrm{x}}\rangle, \langle\bar{\mathrm{y}}\rangle){::=}\mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$ *and the size relations* φ_r *computed for* r*. We distinguish the subsequence of all calls to procedures* $\mathrm{b}_{\mathrm{i}_1} \ldots \mathrm{b}_{\mathrm{i}_k}$ *in* r*, with* $1 \leq i_1 \leq \cdots \leq i_k \leq n$ *and assume* $\mathrm{b}_{\mathrm{i}_\mathrm{j}} = \mathrm{q}_{\mathrm{i}_\mathrm{j}}(\langle\bar{\mathrm{x}}_{\mathrm{i}_\mathrm{j}}\rangle, \langle\bar{\mathrm{y}}_{\mathrm{i}_\mathrm{j}}\rangle)$*. Then, the cost equation for* r *is:*

$$p(\bar{x}) = \mathcal{M}^t(g, b_1, \ldots, b_n) + \Sigma_{j=1}^{k}\, q_{i_j}(\bar{x}_{i_j}),\ \ \varphi_r$$

Given a program P*, we denote by* $\mathcal{S}_P$ *the cost relation generated for each rule in* P *w.r.t. the heap consumption cost model* $\mathcal{M}^t$*.* □

Note that each call in the rule $\mathrm{q}_{\mathrm{i}_\mathrm{j}}(\langle\bar{\mathrm{x}}_{\mathrm{i}_\mathrm{j}}\rangle, \langle\bar{\mathrm{y}}_{\mathrm{i}_\mathrm{j}}\rangle)$ has a corresponding abstract version $b^{\alpha}_{i_j} = q_{i_j}(\bar{x}_{i_j})$ where $\bar{x}_{i_j}$ are the size abstractions of $\bar{\mathrm{x}}_{\mathrm{i}_\mathrm{j}}$. The output variables are ignored in the CR as the cost is a function of the *input* data sizes, however it should be noted that the possible effect of output variables on the cost has been already modeled by the size relation φ_r. For simplicity, the same procedure name is used to define its associated cost relation, but in *italic* font.

EXAMPLE 3.3. *The CR generated for the rule-based program in Fig. 2 w.r.t.* $\mathcal{M}^t$ *is depicted in Fig. 4. To simplify the presentation, we assume that the total heap consumption of all external methods (*init_1, $\mathrm{intValue}_1$, init_2 *and* $\mathrm{intValue}_2$*) is* 0 *and we do not show them in the equations from now on. Consider, for example, equation (4). It states that the memory consumption of executing* $\mathrm{f_c}(\langle\mathrm{n}, \mathrm{a}\rangle, \langle\mathrm{n}, \mathrm{a}\rangle)$ *is the total memory consumption of executing* $\mathrm{g}(\langle\mathrm{n}\rangle, \langle\mathrm{r}\rangle)$ *plus the one of* $\mathrm{f_c}(\langle\mathrm{n}', \mathrm{a}'\rangle, \langle\mathrm{n}', \mathrm{a}'\rangle)$*. The set of constraints attached to equation (4) includes information on: (1) how the sizes of the data change when the program moves from one rule to another, e.g., the constraint* $n' = n/2$ *indicates that the value of* n *decreases by half when calling* $\mathrm{f_c}$ *recursively; and (2) numeric conditions (obtained by abstracting the guards) under which the corresponding rule is applicable, e.g.,* $n > 1$ *indicates that the equation can be applied only when* n *is greater than* 1*.* □

An important observation is that, as discussed in Sec. 1, this analysis approach is intrinsically designed to infer the *total cost* (memory allocation in this case) of the program's execution and not to infer its peak consumption. This is because the equations accumulate the cost of all instructions and rules together as it can be observed in the CR for the example above.

3.3 Closed-Form Upper Bounds

Once the CR is generated, a cost analyzer has to use a CR solver in order obtain closed-form upper bounds, i.e., expressions without recurrences. The technical details of this process are not explained in the paper as our analysis does not require any modification to such part. In what follows, we rely on the CR solver of [3] (which can be accessed online through a web interface) to obtain closed-form upper bounds for our examples. The soundness of the overall analysis, as stated in the next theorem, requires that the equations generated as well as their closed-form upper bounds are sound.

THEOREM 3.4 (soundness [3]). *Given a procedure* p*, and a trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle\bar{\mathrm{x}}\rangle, \langle\bar{\mathrm{y}}\rangle) \cdot bc, tv_1\rangle \cdot A; h_1 \rightsquigarrow^* \langle \mathrm{q}, bc, tv_n\rangle{\cdot}A; h_n$*, then* $p(\bar{v}) \geq total(t)$ *where* $\bar{v} = \alpha(\bar{x}, tv_1, h_1)$*.* □

Observe that the trace t in the theorem represents an execution of procedure p for some specific input data (properly stored in tv_1 and h_1) where the first configuration corresponds to calling p and the last one to returning from that specific call. As already mentioned in Sec. 3.1, $\bar{v}$ denotes the size of the input data.

EXAMPLE 3.5. *Solving the equations of Fig. 4 results in the following closed-form upper bounds for* f, m, g *and* h*:*

$$\begin{array}{rl} m(n) = & (2^{\mathsf{nat}(n)} - 1)*(f(n)+size(\text{Tree}^1)) \\ f(n) = & \log_2(\mathsf{nat}(n-1)+1)*(size(\text{Integer}^2)+size(\text{Integer}^3))+ \\ & \log_2(\mathsf{nat}(n-1)+1)*size(\text{Long}^4)) \\ f_c(n,a) = & \log_2(\mathsf{nat}(n-1)+1)*(size(\text{Integer}^2)+size(\text{Integer}^3)) \\ f_d(i,a) = & \log_2(\mathsf{nat}(i-1)+1)*size(\text{Long}^4)) \\ g(n) = & size(\text{Integer}^2)+size(\text{Integer}^3) \\ h(n) = & size(\text{Long}^4) \end{array}$$

where the expression $\mathsf{nat}(l)$ *is defined as* $\max(l, 0)$ *to avoid negative evaluations. As expected, method* m *has an exponential memory consumption due to the two recursive calls, which in turn is multiplied by the allocation at each iteration (i.e., the consumption of* f *plus the creation of a* Tree *object). The solver indeed substitutes* $f(n)$ *by its upper bound shown below. The memory consumption of* f *has two logarithmic parts: the leftmost one corresponds to the first loop which accumulates the allocation performed along the execution of* $g(n)$*, the rightmost one corresponds to the second loop with the allocation of* $h(n)$*.* □

A fundamental observation is that the above upper bounds on the memory consumption can be tighter if one considers the effect of GC. For instance, a more precise upper bound for m can be inferred if we take into account that the memory allocated by f can be entirely garbage collected upon return from f. Likewise, the upper bound for f can be more precise if we take advantage of the fact that not all memory escapes from g. The goal of the rest of the paper is to provide automatic techniques to infer accurate bounds by taking into account the memory freed by scoped-GC.

4. Escaped Memory Upper Bounds

In a real language, GC removes objects which become *unreachable* along the program's execution. Given a configuration $A; h$, we say that an object $o = h(r)$ where $r \in \text{dom}(h)$ is not reachable, if it cannot be accessed (directly or indirectly) through the variables table tv of any activation record in A. To develop our analysis, we assume a scoped-memory manager, which at the level of the source language, meets these conditions: (1) it reclaims memory *only* upon return from methods and, (2) it collects *all* unreachable objects which have been created *during* the execution of the corresponding method call.

In order to simulate the behavior of such garbage collector at the level of the corresponding rule-based bytecode, it is enough to assume that the memory manager reclaims memory only upon return from procedures that correspond to methods but not from procedures that correspond to intermediate states like f_c and f_d. We use $\rightsquigarrow_{gc}$ to denote $\rightsquigarrow$-transitions with a scoped-memory manager which meets the two conditions above. In this context, the *escaped memory* of a procedure execution is defined as follows. Given a trace $t \equiv \langle q, p(\langle\bar{x}\rangle, \langle\bar{y}\rangle) \cdot bc, tv_1\rangle \cdot A; h_1 \rightsquigarrow^*_{gc} \langle q, bc, tv_n\rangle \cdot A; h_n$ whose first configuration corresponds to calling p and the last one to returning from that specific call, the escaped memory of t is $escaped(t) = size(h_n) - size(h_1)$, which corresponds to the amount of memory allocated during the execution of p and still live in the memory upon exit from p. Our first contribution is an automatic technique to infer *escaped memory upper bounds*.

4.1 Inference of Escape Information

We say that an object *escapes* from a procedure p, in the context of a scoped-memory manager, if it is created during the execution of p, and still available in the heap upon exit from p. Note that if p corresponds to an intermediate procedure, such object might be unreachable but still has not been garbage collected because GC is applied only when exiting from procedures that correspond to methods in the original program. As a preprocessing phase, for each procedure p, we need to over-approximate the set of allocation instructions "new c^i" that might be executed when calling p and its transitive calls such that it is guaranteed that *all* objects they create are not in memory upon exit from p, i.e., they have been garbage collected. Recall that an allocation instruction "new c^i" is uniquely identified by the *tagged* class c^i. We use the notation $A \setminus B$ for the difference on sets.

DEFINITION 4.1 (collectable objects). *Given a procedure* p*, we denote by* $collectable(\text{p})$ *the set of all allocation instructions, identified by their* tagged *classes, defined as follows.*

$c^i \in collectable(\text{p})$ *iff the following conditions hold:*

1. *"*new c^i*" is a reachable instruction from* p*;*
2. *for any trace* $t \equiv \langle \text{q}, \text{p}(\langle\bar{\text{x}}\rangle, \langle\bar{\text{y}}\rangle) \cdot bc, tv_1\rangle \cdot A; h_1 \rightsquigarrow^*_{gc} \langle \text{q}, bc, tv_n\rangle \cdot A; h_n$*, it holds that* $\forall r \in \text{dom}(h_n) \setminus \text{dom}(h_1)$ *the object* $h_n(r)$ *is not an instance of* c^i*.* □

The set of collectable objects can be approximated from the information computed by escape analysis [15, 4]. The goal of escape analysis is to determine all program points where an object is reachable and whether the lifetime of the object can be proven to be restricted only to the current method. In our implementation, we use the approach described in [21] which, as our experiments show, behaves well in practice.

EXAMPLE 4.2. *The escape information is computed for all procedures (both methods and intermediate rules) defined in Fig. 2:*

$$\begin{array}{l} collectable(\text{m}) = collectable(\text{f}) = \{\text{Integer}^2, \text{Integer}^3, \text{Long}^4\} \\ collectable(\text{f}_\text{c}) = collectable(\text{g}) = \{\text{Integer}^2\} \\ collectable(\text{f}_\text{d}) = collectable(\text{h}) = \emptyset \end{array}$$

As an example, the information in the set $collectable(\text{f})$ *states that the objects created with class tags* Integer^2*,* Integer^3 *and* Long^4 *during the execution of* f *by the transitive calls to* g *and* h*, do not escape from* f*. Also,* $collectable(\text{f}_\text{d}) = \emptyset$ *means that the object* Long^4 *created in* h *might escape from* f_d*. An important observation is that this object is not reachable upon exit from* f_d*, but since GC is applied only upon exit from procedures that correspond to methods, it will be collected only upon exit from* f*. This issue will be further discussed in Sec. 6.* □

4.2 Upper Bounds on the Escaped Memory

Intuitively, our technique to infer upper bounds on the escaped memory consists of two steps. In the first step, we generate equations for the total allocation (exactly as stated in Def. 3.2) which accumulate *symbolic* expressions of the form $size(c^i)$ to represent the heap allocation for the instruction new c^i, rather than its concrete allocation size. From these equations, we obtain an upper bound for the total memory allocation as a symbolic expression which contains residual $size(c^i)$ sub-expressions. The main novelty is that, in a second step, we tighten up such total allocation upper bound to extract from it only its escaped memory as follows. Given a procedure p, and its total heap consumption upper bound $p(\bar{x})$, we obtain the upper bound on the escaped memory by replacing expressions of the form $size(c^i)$ by 0 if it is guaranteed that all corresponding objects are not available in the memory upon exit from p, namely $c^i \in collectable(\text{p})$. Given an expression exp and a substitution σ from sub-expressions to values, $exp[\sigma]$ denotes the application of σ on exp.

DEFINITION 4.3 (escaped memory upper bound). *Given a procedure* p*, its escape information* $collectable(\text{p})$*, and its (symbolic) upper-bound for the total memory allocation* $p(\bar{x}) = exp$*, the escaped memory upper-bound of* p *is defined as:* $\check{p}(\bar{x}) = exp[\forall c^i \in collectable(\text{p}).size(c^i) \mapsto 0]$*.* □

Observe that, in the above definition, it is required that the set $collectable(\mathrm{p})$ contains the information for objects created in transitive calls from p, as stated in Def. 4.1, because escaped memory upper-bounds for a method p are obtained by using only the information in $collectable(\mathrm{p})$ and not in any other $collectable(\mathrm{q})$ with $\mathrm{q} \neq \mathrm{p}$. This is an essential difference w.r.t. existing work [3] which does not compute information for transitive calls, but instead computes the escape information only for the objects which are created inside each method (excluding its transitive calls). We obtain strictly more accurate bounds as the following example illustrates.

EXAMPLE 4.4. *Applying Def. 4.3 to the total heap allocation inferred in Ex. 3.5, by using the escape information of Ex. 4.2, results in the escaped memory upper bounds:*

$$\begin{array}{ll} \check{m}(n) = (2^{\mathsf{nat}(n)} - 1) * size(\mathrm{Tree}^1) & \check{f}(n) = 0 \\ \check{f}_c(n,a) = \log(\mathsf{nat}(n-1)+1) * size(\mathrm{Integer}^3) & \check{g}(n) = size(\mathrm{Integer}^3) \\ \check{f}_d(i,a) = \log(\mathsf{nat}(i-1)+1) * size(\mathrm{Long}^4) & \check{h}(n) = size(\mathrm{Long}^4) \end{array}$$

We can see that the escaped memory upper bound for m *does not accumulate the allocations of* Long^4 *nor* $\mathrm{Integer}^2$ *and* $\mathrm{Integer}^3$ *objects because they do not escape from* f. *In [3], the allocations corresponding to* $\mathrm{Integer}^3$ *and* Long^4 *are accumulated because they escape from the method where these objects have been created. The problem is that in [3] they are accumulated in the CR and hence in all upper bounds for methods that transitively invoke* g *and* h. □

The following theorem states the soundness of our escaped memory upper bounds.

THEOREM 4.5 (soundness). *Given a procedure* p*, and a trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) \cdot bc, tv_1 \rangle \cdot A; h_1 \leadsto^*_{gc} \langle \mathrm{q}, bc, tv_n \rangle \cdot A; h_n$*, then* $\check{p}(\bar{v}) \geq escaped(t)$ *where* $\bar{v} = \alpha(\bar{\mathrm{x}}, tv_1, h_1)$.

Proof.
(sketch) First, by Theorem 3.4, we have the soundness of the total allocation upper bound $p(\bar{v}) \geq total(t)$. Second, by the soundness of escape analysis [4], we know that $collectable(\mathrm{p})$ gives a safe approximation of the objects that escape from t. Now, by combining both parts, we have that $\check{p}(\bar{v}) \geq escaped(t)$ and, hence, the soundness of $\check{p}(\bar{v})$ follows. □

5. Live Heap Space Analysis

This section presents a novel live heap space analysis for garbage-collected languages which obtains precise upper bounds including logarithmic, exponential, etc. complexity classes. Achieving accuracy is crucial because live heap bounds represent the minimum amount of memory required to execute a program.

5.1 The Notion of Peak Consumption

Essentially, given a trace $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) \cdot bc, tv_1 \rangle \cdot A; h_1 \leadsto^*_{gc} \langle \mathrm{q}, bc, tv_n \rangle \cdot A; h_n$, the peak consumption can be defined as $peak(t) = max(size(h_2), \ldots, size(h_n)) - size(h_1)$. We decrement $size(h_1)$ because the objects created in an outer scope (i.e., those in h_1) cannot be collected during the execution t, as stated in condition (2) of scoped-GC in Sec. 4.

Let us illustrate this notion by means of this simple method "void r() {A; p(); B; q(); C;}" whose memory consumption is showed in Fig. 5. A, B and C are sequences of instructions that do not contain any method invocation. We use the notation $\hat{p}$ to the note the peak consumption of executing the method p. We can observe that the peak heap consumption $\hat{r}$ is the maximal of three possible scenarios: (1) In the leftmost column, we depict a scenario where we allocate A and then execute p, thus we add the peak heap consumption of p. (2) In the next alternative scenario, we still have A and then return from p's execution, thus we add the memory escaped upon return from p (i.e., $\check{p}$) and we continue until the execution of q. Hence we add B plus the peak of q. (3) In the next column, we have A, plus the memory escaped from p, plus B, plus the memory escaped from q, plus C. Observe that any of these scenarios may correspond to the actual peak and we need to infer upper bounds for all of them and then take the maximal. The rightmost column indicates the upper bound for total allocation which is clearly much less accurate.

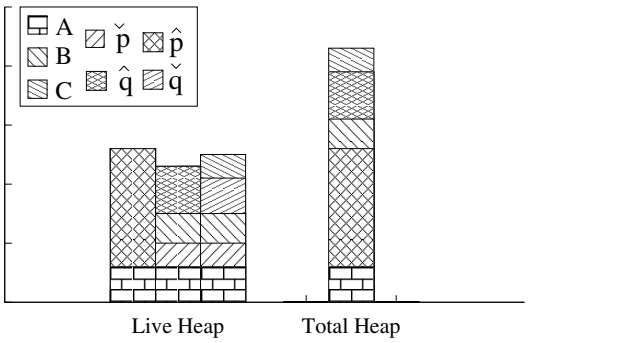


Figure 5. Memory Consumption of simple program

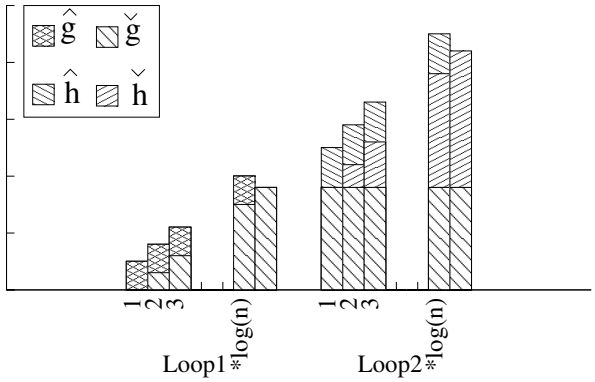


Figure 6. Memory Consumption of running example

In general the problem is more complicated, e.g., when method invocations occur within loops. Fig. 6 depicts the actual memory consumption of the execution of method f in our running example. Column 1 captures the heap allocation of executing g at the first iteration of the first loop (defined by procedure $\mathrm{f_c}$). Column 2 represents the escaped memory from g plus the next iteration of the loop where g allocates again $\hat{\mathrm{g}}$ memory and so on. As the loop in $\mathrm{f_c}$ is executed $\log(n)$ times we have all such possible scenarios over the tag Loop 1. Then, we start the execution of the second loop with an initial heap usage of $log(n)$ times the memory escaped from g. Similarly, at each iteration of the second loop, method h is invoked which allocates a maximal of memory $\hat{\mathrm{h}}$ and upon return, we need to consider the escaped memory from h plus the next execution. As the loop is executed $\log(n)$ times, we have all possible scenarios to the right grouped over the tag Loop 2. The peak heap allocation of executing f is the maximal of all such scenarios, namely the maximal between the two scenarios marked with $*$. The important point is that we need to infer upper bounds for $\hat{\mathrm{h}}$, $\hat{\mathrm{g}}$, $\check{\mathrm{h}}$, $\hat{\mathrm{g}}$ and generate as peak heap consumption the expression $\hat{\mathrm{f}} = \max(\hat{\mathrm{g}} + (\log(\mathrm{n}) - 1) * \check{\mathrm{g}}, \hat{\mathrm{h}} + (\log(\mathrm{n}) - 1) * \check{\mathrm{h}} + \log(\mathrm{n}) * \check{\mathrm{g}})$. Note that, in principle, it could happen that $\hat{\mathrm{g}} > (\log(\mathrm{n}) - 1) * \check{\mathrm{h}} + \hat{\mathrm{h}}$.

5.2 Peak Consumption Cost Relation

We now propose a novel approach for generating CR that, by relying on the escaped memory bounds, capture the peak heap consumption by considering all possible states of a program's execution. Our proposal is based on the following intuition: Let m_1 and m_2 be two methods, and let $\hat{m}_1(\bar{x}_1)$ and $\hat{m}_2(\bar{x}_2)$ be the peak heap consumption of executing m_1 and m_2 respectively, then the peak heap consumption of the two consecutive calls $\mathrm{m}_1;\mathrm{m}_2$ is $\max(\hat{m}_1(\bar{x}_1), \check{m}_1(\bar{x}_1) + \hat{m}_2(\bar{x}_2))$. The following definition generalizes this idea for an arbitrary sequence of statements.

DEFINITION 5.1 (peak consumption CR). *Consider a rule* $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) ::= \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$ *and its corresponding size relations* φ_r. *Then, its* peak consumption *equation is* $\check{p}(\bar{x}) = \mathcal{T}(b_1, \ldots, b_n),\ \varphi_r$ *where* $\mathcal{T}$ *is defined as follows:*

$$\mathcal{T}(b_1,\ldots,b_n)::= \begin{cases} 0 & \text{if } n = 0 \\ \max(\hat{q}(\bar{x}_1), \check{q}(\bar{x}_1) + \mathcal{T}(b_2,\ldots,b_n)) & \text{if } b_1 = \mathrm{q}(\langle \bar{\mathrm{x}}_1 \rangle, \langle \bar{\mathrm{y}}_1 \rangle) \text{ is a call} \\ \mathcal{M}^t(b_1) + \mathcal{T}(b_2,\ldots,b_n) & \text{if } b_1 \text{ is an instruction} \end{cases}$$

Given a program P, *we denote by* $\hat{\mathcal{S}}_P$ *the* peak consumption cost relation *generated for each rule in* P. □

In the above definition, it can be observed that, in the second case, we generate two possible scenarios not only for methods, but also for intermediate procedures. These scenarios correspond to either the peak of the first procedure call or to the escaped memory from the first procedure call plus the peak of the rest of the instructions sequence. Considering the two scenarios at the level of procedures (no only of methods) allows us to gain further accuracy in situations, like in the method f, in which intermediate procedures correspond to loops which contain method invocations. The next example illustrates this point.

EXAMPLE 5.2. *The peak consumption* CR $\hat{\mathcal{S}}_P$ *of the rule-based program is different from the one in Fig. 4 in equations (1), (3), (4) and (6) which are now as follows:*

$$\begin{array}{lll} (1) & \hat{m}(n) = & size(\mathrm{Tree}^1) + \max(\hat{m}(s_1), \check{m}(s_1) + \max(\hat{m}(s_2), \\ & & \check{m}(s_2) + \max(\hat{f}(n), \check{f}(n) + \hat{init}(s_0,s_1,s_2,s_3)))) \\ (3) & \hat{f}(n) = & \max(\hat{f}_c(n,a), \check{f}_c(n,a) + \hat{f}_d(i,a')) \\ (4) & \hat{f}_c(n,a) = & \max(\hat{g}(n), \check{g}(n) + \hat{f}_c(n',a')) \\ (6) & \hat{f}_d(i,a) = & \max(\hat{h}(i), \check{h}(i) + \hat{f}_d(i',a')) \end{array}$$

with the same constraints as those of Fig. 4. We can now replace the escaped memory upper bounds $\check{g}$, $\check{h}$, $\check{m}$ *and* $\check{f}$ *by the ones in Ex. 4.4. As an optimization, we do not apply the transformation to the last call in the rules, for instance, to the call to* $init$ *in equation (1), since trivially* $\hat{init} \geq \check{init}$. *Observe that in equation (3) we have applied also two possible scenarios to the intermediate procedure* f_c *which does not correspond to a method by introducing the max operator. This is essential to keep the two possible peaks (marked with "*" in the figure) separate instead of accumulating both of them, which would lead to a larger, less accurate upper bound. Besides, it is sound w.r.t. scoped-GC because the corresponding escaped memory bounds for* $\check{f}_c$ *and* $\check{f}_d$ *are obtained by considering that GC takes place upon method's return only.*

The most important point is that equation (4) accurately captures the memory consumption of all scenarios in Loop 1 of Fig. 6 and equation (6) captures those in Loop 2 to the right of the figure, as it will become clear after solving the equations in Ex. 5.3. □

An important feature of our CR $\hat{\mathcal{S}}_P$ is that they can still be solved by relying on a standard upper bound solver for CR produced by cost analysis like the one in [3]. The only adjustment is that our CR use the $\max$ operator which is frequently not supported. This is handled by a further preprocessing which transforms one equation that uses $\max$ into an equivalent set of equations that do not use $\max$ by creating nondeterministic equations whenever we have $\max$. In particular, an equation of the form $p(\bar{x}) = A + max(B, C), \varphi$ is translated into the two equations $p(\bar{x}) = A + B, \varphi$ and $p(\bar{x}) = A + C, \varphi$. Since an upper bound solver looks for an upper bound for all possible paths, it is guaranteed that this transformation simulates the effect of the $\max$ operator. Nested $\max$ are translated iteratively. For instance, the translation of equation (1) in Ex. 5.2, results in the following equations:

$$\begin{array}{l} \hat{m}(n) = size(\mathrm{Tree}^1) + \hat{m}(s_1), \varphi_1 \\ \hat{m}(n) = size(\mathrm{Tree}^1) + \check{m}(s_1) + \hat{m}(s_2), \varphi_1 \\ \hat{m}(n) = size(\mathrm{Tree}^1) + \check{m}(s_1) + \check{m}(s_2) + \hat{f}(n), \varphi_1 \\ \hat{m}(n) = size(\mathrm{Tree}^1) + \check{m}(s_1) + \check{m}(s_2) + \check{f}(n) + \hat{init}(s_0,s_1,s_2,s_3), \varphi_1 \end{array}$$

EXAMPLE 5.3. *Solving the transformed equations results in the following closed-form upper bounds:*

$$\begin{array}{lll} \hat{m}(n) & = 2^{\mathsf{nat}(n)} * size(\mathrm{Tree}^1) + \hat{f}(n) \\ \hat{f}(n) & = \max(\hat{f}_c(n,a), \check{f}_c(n,a) + \hat{f}_d(n,a')) \\ \hat{f}_c(n,a) & = (\log(\mathsf{nat}(n-1)+1) + 1) * size(\mathrm{Integer}^3) + size(\mathrm{Integer}^2) \\ \hat{f}_d(i,a) & = (\log(\mathsf{nat}(i-1)+1) + 1) * size(\mathrm{Long}^4) \\ \hat{g}(n) & = size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3) \\ \hat{h}(n) & = size(\mathrm{Long}^4) \end{array}$$

We can observe that the peak bound for f *accurately captures the maximal of the two scenarios in the figure: (1)* $\hat{f}_c(n,a)$ *corresponds to the leftmost column of Fig. 6 (since* $\check{g}$ *is* $size(\mathrm{Integer}^3)$ *which is accumulated* $log(n)-1$ *times and* $\hat{g}(n)$ *is* $size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3)$ *and (2)* $\check{f}_c(n,a) + \hat{f}_d(n,a')$ *corresponds to the rightmost column where, as expected, we accumulate* $log(n) - 1$ *times the escaped* $size(\mathrm{Long}^4)$ *object plus an additional one which is the peak consumption of* h *(and nothing escapes from* $\mathrm{f_c}$*).*

It is fundamental to observe the difference between the above live heap space bound for m *and the total allocation computed in Ex. 3.5. In our live bound, since the allocation required by* f *can be entirely garbage collected upon exit from* f*, the required heap is not proportional to the number of times that* f *is invoked (i.e., exponential on* n*) but rather the memory required for a single execution of* f. □

The following theorem states that the upper bounds computed by our analysis are *sound*, i.e., for any input values, they evaluate to a larger value than the actual peak consumption.

THEOREM 5.4 (soundness). *Given a procedure* p*, and a trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) \cdot bc, tv_1 \rangle \cdot A; h_1 \rightsquigarrow^*_{gc} \langle \mathrm{q}, bc, tv_n \rangle \cdot A; h_n$ *then* $\hat{p}(\bar{v}) \geq peak(t)$ *where* $\bar{v} = \alpha(\bar{\mathrm{x}}, tv_1, h_1)$. □

6. Approximating the Ideal Garbage Collector

In this section, we show how the analysis of Sec. 5 can be refined to consider other GC schemes and, in particular, to get closer to the *ideal* GC manager where objects are collected as soon as they become unreachable. For instance, the peak consumption upper bound inferred in Ex. 5.3 for f is accurate when using a scope-GC scheme, since all objects created inside the loops are collected only upon exit from f. However, it is clearly inaccurate for an ideal GC scheme, since the lifetime of each object created in f is limited to one iteration of the corresponding loop, and therefore f can be executed in constant heap space.

Luckily, we can take advantage of scopes in the rule-based representation in order to infer accurate upper bounds for such GC schemes without modifying our analysis. In Def. 4.1 the effect of GC is considered only on exit from procedures that correspond to methods, this is essential in order to obtain safe upper bounds for scoped-GC, since in the original language GC is assumed to be applied upon exit from method scopes. However, the rule-based language distinguishes scopes that correspond to code fragments (in

the original program) smaller than methods, e.g., $\mathrm{f_c}$ and $\mathrm{f_d}$ respectively correspond to the first and second loop of f. Considering the effect of GC on exit from these (non-method) smaller scopes corresponds to applying more often GC than in the original language, and therefore getting closer to the ideal GC. In order to support this, we need to compute the set of collectable objects for blocks exactly as we do for methods in Def. 4.1. Let us see an example.

EXAMPLE 6.1. *If we apply GC upon exit from* $\mathrm{f_c}$*, then the collectable objects are* $collectable(\mathrm{f_c}) = \{\mathrm{Integer}^2, \mathrm{Integer}^3\}$*, and hence* $\hat{f}_c(n,a) = 0$*. Observe that in Ex. 4.2* $collectable(\mathrm{f_c})$ *contains only* $\mathrm{Integer}^3$*. This in turn improves the peak consumption for* f *to* $\hat{f}(n) = \max(\hat{f}_c(n,a), \hat{f}_d(n,a'))$*, which is clearly more precise than the one in Ex. 5.3.* □

Interestingly, the above upper-bound can be even further improved in order to obtain one which is as close as possible to the ideal behavior. Consider Rule (6) in Fig. 2 which corresponds to the second loop in f. The object created in h, and escaped to the calling context, becomes unreachable immediately after executing $\mathrm{intValue_2}$. Thus, if we separate the loop's body into a separate procedure $\mathrm{f'_d}$, we make this behavior observable to our analysis. This can be done by transforming the rules associated to the loops as follows:

(4) $\mathrm{f_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle) ::=$
$\quad \mathrm{f'_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle).$
$\quad \mathrm{f_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle).$

$\mathrm{f'_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle) ::=$
$\quad \mathbf{n > 1},$
$\quad \mathrm{g}(\langle \mathrm{n}\rangle, \langle \mathrm{s_0}\rangle),$
$\quad \mathrm{intValue_1}(\langle \mathrm{s_0}\rangle, \langle \mathrm{s_0}\rangle)$
$\quad \mathrm{a} := \mathrm{a} + \mathrm{s_0},$
$\quad \mathrm{n} := \mathrm{n}/2.$

(6) $\mathrm{f_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle) ::=$
$\quad \mathrm{f'_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle).$
$\quad \mathrm{f_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle).$

$\mathrm{f'_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle) ::=$
$\quad \mathbf{i > 1},$
$\quad \mathrm{h}(\langle \mathrm{i}\rangle, \langle \mathrm{s_0}\rangle),$
$\quad \mathrm{intValue_2}(\langle \mathrm{s_0}\rangle, \langle \mathrm{s_0}\rangle)$
$\quad \mathrm{a} := \mathrm{a} * \mathrm{s_0},$
$\quad \mathrm{i} := \mathrm{i}/2.$

Now the peak consumption equations for $\mathrm{f_c}$ and $\mathrm{f_d}$ are:

$$\begin{array}{ll}
\hat{f}_c(n,a) = \max(\hat{f}'_c(n,a), \check{f}'_c(n,a) + \hat{f}_c(n',a')) & \{n{>}1, n'{=}n/2\}\\
\hat{f}_c(n,a) = 0 & \{n{\leq}1\}\\
\hat{f}_d(i,a) = \max(\hat{f}'_d(i,a), \check{f}'_d(i,a) + \hat{f}_d(i',a')) & \{i{>}1, i'{=}i/2\}\\
\hat{f}_d(i,a) = 0 & \{i{\leq}1\}\\
\hat{f}'_c(n,a) = size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3) & \\
\hat{f}'_d(i,a) = size(\mathrm{Long}^4) &
\end{array}$$

and, since $\check{f}'_c(n,a){=}\check{f}'_d(i,a){=}0$, solving them results in

$$\begin{array}{l}
\hat{f}_c(n,a){=}size(\mathrm{Integer}^2){+}size(\mathrm{Integer}^3)\\
\hat{f}_d(i,a){=}size(\mathrm{Long}^4)
\end{array}$$

which in turn improves the upper bound of f to

$$\hat{f}(n) = max(size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3), size(\mathrm{Long}^4))$$

which is indeed the minimal amount of memory required in order to execute f in the presence of an ideal GC.

In order to support such transformations, one should guide the transformation from the bytecode to the rule-based program by the information previously computed on the lifetime of the different objects. Such analysis should give us indications about when it is profitable to make smaller scopes. Currently, we do this transformation only for scopes that correspond to loops. Also, it should be noted that there is an efficiency versus accuracy trade-off here, as we generate more equations in this case which thus will be more expensive to solve. Note that the same ideas are useful for supporting region-based memory management. The idea is to infer regions and use this information to separate the scopes, such that the exit from scopes coincides with the removal of the corresponding region.

7. Experiments

In this section, we assess the practicality of our proposal on realistic programs, the standardized set of benchmarks in the JOlden suite [12]. This benchmark suite was first used by [7] in the context of memory usage verification for a different purpose, namely for checking memory adequacy w.r.t. given specifications, but there is no inference of upper bounds as our analysis does. It has been also used by [5] for our same purpose, i.e., the inference of peak consumption. However, since [5] does not deal with memory-consuming recursive methods, the process is not fully automatic in their case and they have to provide manual annotations. Also, they require invariants which sometimes have to be manually provided. In contrast, our tool is able to infer accurate live heap upper bounds in a fully automatic way, including logarithmic and exponential complexities.

The first column of Table 1 contains the name of the benchmark. For most examples, we analyze the method `main` which transitively requires the analysis of the majority of the methods in the package. Only in those benchmarks whose name appears in two different rows, we do not analyze the `main` but rather all those methods invoked within the `main` that we succeed to analyze. In particular, benchmarks Health(cV), Health(gR), Bh(cTD), Bh(eB), Voronoi(cP), and Voronoi(b) correspond, respectively, to methods `createVertex`, `getResults`, `createTreeData`, `expandBox`, `createPoints`, and `buildDelaunayTriangulation` in the corresponding packages. In benchmark Bh, we cannot obtain an upper bound for the method `stepSystem` which is invoked within `main`. The reason is that this method contains a loop whose termination condition does not depend on the size of the data structure, but rather on the particular value stored at certain locations within the data structure. In general, it is complicated to bound the number of iterations of this kind of loops. Basically, the same situation happens in the method `simulate` of benchmark Health. In Voronoi, we are able to analyze all methods when they are not connected together. Unfortunately, we cannot analyze the `main` which, first invokes the method `createPoints` which returns an object `point` and then invokes the method `point.buildDelaunayTriangulation` on such object. The problem is that the upper bound of `buildDelaunayTriangulation` depends on the size of the object `point` returned by `createPoints` and the size analysis is not able to propagate such relation. It should be noted that, in these three cases, the limitations are not related to our proposal in this paper but to external components which can be independently improved.

The second and third columns in the table show, respectively, the upper bounds for total allocation and for live heap space usage. Note that the cost model we use for the experiments substitutes the symbolic expressions $size(Obj)$ by their corresponding numerical values, so that the system can perform mathematical simplifications. In particular, the size of primitive types is $1, 2, 4$, etc. bytes respectively for byte, char, int, etc.; the size of a reference is set to 4 bytes; and the size of an object is the sum of the sizes of all its fields (including those inherited).[1]

Let us first explain the examples Tsp, Bisort, Health, TreeAdd, Perimeter and Voronoi which follow a similar pattern. Basically, they contain methods (in rule-based form) which have this shape $p(X) ::= alloc(k), p(Y_1), \ldots, p(Y_n)$, i.e., a certain allocation k is accumulated by several recursive calls to the method. The size of the arguments in the recursive calls decrease by half in examples Tsp, Bisort and Voronoi and there are two recursive calls. Thus, their resulting upper bounds are linear. In benchmarks Health, Perimeter and TreeAdd, the size of the argument decreases by a constant; the first two examples contain 4 recursive calls and the

[1] This is just an estimation. The sizes depend on the particular JVM

Bench	Total Allocation Upper Bounds	Live Heap Space Upper Bounds
Mst	$\texttt{nat(A+1)*nat}(\frac{\texttt{A}}{4}) + \texttt{33*nat(A+1) + 8}$	$\texttt{nat(A+1) + 8 + max(nat(A+1)}^2 \texttt{ + 18*nat(A+1) + nat}(\frac{\texttt{A}}{4}) \texttt{ + 72, nat(A+1)*nat}(\frac{\texttt{A}}{4}) \texttt{ + 25*nat(A+1) + 2*nat}(\frac{\texttt{A}}{4}) \texttt{ + 48)}$
Em3d	2*nat(D-1)*(32+nat(B)) + 2*nat(B) + 16*nat(C) + 2*nat(D) + 89	max(4*nat(B) + nat(C) + 2*nat(D) + 2*nat(D-1) + 153, 4*nat(B) + max(16,nat(C)) + 2*nat(D) + 2*nat(D-1) + 153), (34 + nat(B))*nat(D-1) + 6*nat(B) + 3*nat(D) + 313)
Bisort	4*nat(A) + 12*nat(B-1) + 52	max(4*nat(A),12*nat(B-1) + 36)
Tsp	46*nat(2*B-1) + 138	28*nat(2*B-1) + 84
Power	258544	5992
Health(cV)	$104*4^{\texttt{nat(A)}} + 416$	$104*4^{\texttt{nat(A)}} + 416$
Health(gR)	$28*4^{\texttt{nat(A-1)}} + 36$	$28*4^{\texttt{nat(A-1)}} + 36$
TreeAdd	$40*2^{\texttt{nat(B-1)}} + 4*\texttt{nat(A)} + 76$	$24*2^{\texttt{nat(B-1)}} + 60$
Bh(cTD)	96*nat(B) + 128	92*nat(B) + nat(B-1) + 308
Bh(eB)	96	92
Perimeter	$56*4^{\texttt{nat(B)}} + 4*\texttt{nat(A)} + 128$	$56*4^{\texttt{nat(B)}} + 112$
Voronoi(cP)	20*nat(2*A-1) + 60	20*nat(2*A-1) + 60
Voronoi(b)	$88*2^{\texttt{nat(A-1)}} + 8$	$88*2^{\texttt{nat(A-1)}} + 8$

Table 1. Upper bounds for Total Allocation and Live Heap Usage

latter one 2 recursive calls. Thus, their resulting upper bounds are exponential. The upper bounds for live heap and total heap for the methods in Health and Voronoi are the same. This happens because the analyzed methods are encharged of creating the data structures and there is no memory that can be garbage collected. In the remaining examples, the method `main` first calls the method `parseCmdLine` which creates a (linear) number of objects that do not escape to the `main` and, then, calls other methods that construct (and modify) a data structure which escapes to the `main`. The fact that some memory can be garbage collected explains that the live heap bounds are smaller than the total allocation. Tsp is interesting because some auxiliary `Double` objects are created during the execution of the methods `uniform` and `median` which do not escape from such methods and hence the difference between the live bound and the total allocation is bigger.

Benchmark Power has a constant memory consumption. Its live bound is much smaller than the total allocation because many objects are created by the constructor of `Lateral` which become unreachable and hence can be garbage collected. In the examples Mst and Em3d, most of the memory is allocated during the construction of a graph and all such memory cannot be garbage collected. As before, the live bound is slightly smaller because of the memory created by `parseCmdLine` which can be entirely garbage collected. Finally, the methods analysed for the benchmark Bh also create a number of auxiliary objects that can be garbage collected and the live heap bounds become tighter than the total allocation.

It is not easy to compare our upper bounds with those obtained by [5] since the cost models are different (we count sizes of objects as explained above while they count number of objects), they consider a region-based memory model while our analysis is developed for a scope-based model and, besides, for recursive methods (which occur in most benchmarks) [5] requires manual annotations that are not shown in their paper. In spite of these differences, as expected, our upper bounds coincide with those of [5] asymptotically (i.e., by ignoring the coefficients and constants).

An interesting experimentation that we plan to do for future work is to compare our upper bounds with actual observed values. This is however a rather complicated task. Note that it would require choosing particular inputs, and the memory consumption of the program could highly vary depending on such choice. We are confident about the positive results since, as we saw above, our UBs are coherent with those in [5], which in turn have already been compared to actual observed values.

8. Related Work

There has been much work on analyzing program cost or resource complexities, but the majority of it is on time analysis (see, e.g., [22]). Analysis of live heap space is different because it involves explicit analysis of all program states. Most of the work of memory estimation has been studied for functional languages. The work in [11] statically infers, by typing derivations and linear programming, linear expressions that depend on functional parameters while we are able to compute non-linear bounds (exponential, logarithmic, polynomial). The technique is developed for functional programs with an explicit deallocation mechanism while our technique is meant for imperative bytecode programs which are better suited for an automatic memory manager. The techniques proposed in [18, 17] consist in, given a function, constructing a new function that symbolically mimics the memory consumption of the former. Although these functions resemble our cost equations, their computed function has to be executed over a concrete valuation of parameters to obtain a memory bound for that assignment. Unlike our closed-form upper bounds, the evaluation of that function might not terminate, even if the original program does. Other differences with the work by Unnikrishnan et al. are that their analysis is developed for a functional language by relying on *reference counts* for the functional data constructed, which basically count the number of pointers to data and that they focus on particular aspects of functional languages such as tail call optimizations.

For imperative object-oriented languages, related techniques have been recently proposed. Previous work on heap space analysis [3] cannot be used to infer upper bounds on the maximum live memory as their cost relation systems are generated to accumulate cost, as explained in Sec. 3. Their refinement to infer escaped memory bounds is strictly less precise than ours as explained in Sec. 4, besides, there is no solution there to infer peak consumption. Later work improves [3] by taking garbage collection into account. In particular, for an assembly language, [8] infers memory resource bounds (both stack usage and heap usage) for low-level programs (assembly). The approach is limited to linear bounds, they rely on explicit disposal commands rather than on automatic memory management. In their system, dispose commands can be automatically generated only if alias annotations are provided. For a Java-like language, the approach of [5] infers upper bounds of the peak consumption by relying on an automatic memory manager as we do. They do not deal with recursive methods and are restricted to polynomial bounds. Besides, our approach is more flexible as regards its adaptation to other GC schemes (see Sec. 6). We believe that

our system is the first one to infer upper bounds on the live heap consumption which are not restricted to simple complexity classes.

9. Conclusions and Future Work

We have presented a general approach to automatic and accurate live heap space analysis for garbage-collected languages. As a first contribution, we propose how to obtain accurate bounds on the memory *escaped* from a method's execution by combining the total allocation performed by the method together with information obtained by means of escape analysis. Then, we introduce a novel form of *peak consumption cost relation* which uses the computed escaped memory bounds and precisely captures the actual heap consumption of programs' execution for garbage-collected languages. Such cost relations can be converted into closed-form upper bounds by relying on standard upper bound solvers. For the sake of concreteness, our analysis has been developed for object-oriented bytecode, though the same techniques can be applied to other languages with garbage collection. We first develop our analysis under a scoped-memory management which reclaims memory on method's return. The amount of memory required to run a method under such model can be used as an over-approximation of the amount required to run it in the context of an ideal garbage collection which frees objects as soon as they become dead. We also show how to approximate such ideal behavior with our analysis. For future work, we also plan to consider how to adapt our techniques to region based memory management [16, 6].

Finally, the idea developed in Sec. 5 can be used to estimate other (non accumulative) resources which require to consider the maximal consumption of several execution paths. For example, it can be used to estimate the maximal height of the frames stack as follows. Given a rule $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) ::= \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_n$, where $b_{i_1} \ldots b_{i_k}$ are the calls in r, with $1 \leq i_1 \leq \cdots \leq i_k \leq n$ and $\mathrm{b}_{\mathrm{i_j}} = \mathrm{q}_{\mathrm{i_j}}(\langle \bar{\mathrm{x}}_{\mathrm{i_j}} \rangle, \langle \bar{\mathrm{y}}_{\mathrm{i_j}} \rangle)$, its corresponding equation would be

$$p(\bar{x}) = \max(1 + q_{i_1}(\bar{x}_{i_j}), \ldots, 1 + q_{i_1}(\bar{x}_{i_k}))\ \ \varphi_r$$

which takes the maximal height from all possible call chains. Each "1" corresponds to a single frame created for the corresponding call. Note that in this setting, tail call optimization can be also supported, by using an analysis that detects calls in tail position, and then replace their corresponding 1's by 0's. This is a subject for future work.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* ,TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Post-proceedings of Formal Methods for Components and Objects (FMCO'07)*, number 5382 in LNCS, pages 113–133. Springer-Verlag, October 2008.

[3] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.

[4] Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java(TM). In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34. ACM, November 1999.

[5] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Proceedings of the International Symposium on Memory management (ISMM)*, New York, NY, USA, 2008. ACM.

[6] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In David F. Bacon and Amer Diwan, editors, *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, pages 85–96. ACM, 2004.

[7] W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard. Memory Usage Verification for OO Programs. In *Proc. of SAS'05*, volume 3672 of *LNCS*, pages 70–86, 2005.

[8] W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *Proceedings of the International Symposium on Memory management (ISMM)*, New York, NY, USA, 2008. ACM.

[9] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–97. ACM Press, 1978.

[10] K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.

[11] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[12] JOlden Suite Collection. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.

[13] H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In *Bytecode'07*, ENTCS, pages 35–50. Elsevier, 2007.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[15] Y. G. Park and B. Goldberg. Escape analysis on lists. In *PLDI*, pages 116–127, 1992.

[16] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[17] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic Accurate Live Memory Analysis for Garbage-Collected Languages. In *Proc. of LCTES/OM*, pages 102–111. ACM, 2001.

[18] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, volume 2575 of *LNCS*, pages 70–85, 2003.

[19] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON'99*, pages 125–135, 1999.

[20] B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.

[21] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.