UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

Análisis y Verificación de Programas Modulares

(Analysis and Verification of Modular Programs)

Tesis Doctoral

Jesús Correas Fernández
Licenciado en Informática
Junio de 2008

Departamento de Lenguajes y Sistemas
Informáticos e Ingeniería del Software

Facultad de Informática

Tesis Doctoral

Análisis y Verificación de Programas Modulares

**presentada en la Facultad de Informática
de la Universidad Politécnica de Madrid
para la obtención del título de
Doctor en Informática**

Candidato:	**Jesús Correas**
	Licenciado en Informática
	Universidad Politécnica de Madrid
Director:	**Germán Puebla**
	Doctor en Informática
	Licenciado en Informática

Madrid, Junio de 2008

UNIVERSIDAD POLITECNICA DE MADRID

Tribunal nombrado por el Magfco. y Excmo. Sr. Rector de la Universidad Politécnica de Madrid, el día......de...........................de 200...

Presidente:	________________________________
Vocal:	________________________________
Vocal:	________________________________
Vocal:	________________________________
Secretario:	________________________________
Suplente:	________________________________
Suplente:	________________________________

Realizado el acto de defensa y lectura de la Tesis el día ...... de.................. de 200... en la Facultad de Informática.

EL PRESIDENTE LOS VOCALES

EL SECRETARIO

A Beatriz

Agradecimientos

En primer lugar, quiero expresar mi gratitud a mi director de tesis, Germán Puebla, y al director del Grupo de Investigación CLIP, Manuel Hermenegildo, por todo el tiempo, energía y recursos que me han permitido terminar esta tesis. Además de su disponibilidad en todo momento, me han animado a continuar en los momentos más difíciles y cuando más lo necesitaba. Ha sido un gran honor para mí formar parte de este grupo de investigación, y espero haber alcanzado el nivel científico que se requiere para participar en las actividades que desarrolla el grupo. También quiero agradecer a Paweł Pietrzak por su colaboración y su amistad, que ha resultado en una fructífera serie de trabajos en común que forman una parte fundamental de esta tesis, y que espero podamos continuar en otros proyectos futuros.

Junto a ellos, los demás coautores de los artículos y ponencias que forman parte de esta tesis han contribuido a mi formación científica: José Manuel Gómez, Manuel Carro, Francisco Bueno, Daniel Cabeza, María García de la Banda, Peter Stuckey y Kim Marriott.

No puedo dejar de mencionar al resto de miembros del grupo CLIP, con los que he compartido buenos tiempos y largas jornadas de trabajo. No dispongo de espacio aquí para referirme a todos ellos sin dejar a ninguno, pero quiero agradecer en especial a Astrid Beascoa, que ha sido tantas veces la clave para terminar el trabajo a tiempo y para compartir las dificultades del día a día.

También quiero aprovechar la oportunidad para agradecer a los profesores que me han invitado a estancias en sus respectivas universidades, por su apoyo y acogida: John Gallagher en la Universidad de Roskilde, y Deepak Kapur en la Universidad de Nuevo Méjico.

Por último, quiero agradecer a mis padres que me dieran lo que no tenían para que yo pudiera llegar hasta aquí. Y todo esto hubiera sido imposible sin el apoyo incondicional y la paciencia de Beatriz, a quien dedico esta Tesis.

Sinopsis

Existe un gran número de técnicas avanzadas de verificación y optimización estática de programas que han demostrado ser extremadamente útiles en la detección de errores de programación y en la mejora de la eficiencia, y que tienen como factor común la necesidad de información precisa de análisis global del programa. La interpretación abstracta es una de las técnicas de análisis más establecidas, lo que ha permitido el desarrollo de métodos innovadores para la verificación de programas.

Por otra parte, uno de los desafíos más importantes en la investigación informática actual consiste en mejorar la capacidad de detectar automáticamente errores en programas y asegurar que un programa es correcto respecto a una determinada especificación, con el objetivo de producir *software* fiable. Por ello, la verificación de programas es un área importante de investigación, y por ello proporcionar técnicas avanzadas para detectar errores y verificar sistemas en programas reales complejos es una de las áreas más relevantes en la industria informática actual. Un enfoque interesante de la verificación de programas es la denominada verificación abstracta, una técnica que tiene como objetivo la verificación de un programa mediante sobre-aproximaciones de la semántica concreta del programa.

Sin embargo, estos métodos no son directamente aplicables a programas reales, pues técnicas avanzadas como las mencionadas están en muchos casos disponibles como prototipos, y los avances conseguidos hasta ahora en esta dirección solamente han permitido su aplicación de modo restringido.

El objetivo de esta Tesis Doctoral es desarrollar técnicas de análisis y verificación para su uso eficiente y preciso en grandes programas modulares o incompletos y mostrar su factibilidad en sistemas reales. Con el fin de evaluar la utilidad práctica de las técnicas propuestas, los algoritmos resultantes han sido implementados e integrados en el sistema `Ciao` y se han comprobado experimentalmente, lo que ha permitido aplicarlos en casos de estudio reales.

Resumen*

Introducción y motivaciones

La descomposición del código de un programa en módulos es una técnica fundamental en el proceso de desarrollo de *software*, pues permite construir programas complejos a partir de módulos sencillos.

A pesar de la importancia de la programación modular, existen muchas técnicas avanzadas de análisis y verificación de programas (tales como dependencia del contexto y polivarianza) que todavía no son aplicables a sistemas reales. Estas técnicas han demostrado ser extremadamente útiles en la detección de errores y en la mejora de la eficiencia, pero desafortunadamente todavía están orientadas a programas pequeños sin las complejidades de los sistemas reales, tales como interfaces con sistemas externos, bases de datos, librerías externas, componentes implementados en otros lenguajes, así como todas las características que proporciona la modularidad. Además, el área de programación lógica se ha visto afectado por la ausencia (hasta el año 2000) de un estándar para describir la modularidad, y que ha dado lugar a la coexistencia de varios enfoques en distintas implementaciones del lenguaje. El objetivo de esta tesis consiste en hacer que estas técnicas puedan utilizarse en programas lógicos grandes, modularizados e incompletos.

Las técnicas de manipulación de programas tienen como factor común la necesidad de información precisa de un análisis global del programa. La interpretación abstracta es una de las técnicas más establecidas para obtener información sobre la ejecución de un programa sin ejecutarlo realmente. No obstante, para la apli-

*Este resumen de la Tesis Doctoral, presentada en lengua inglesa para su defensa ante un tribunal internacional, es preceptivo según la normativa de doctorado vigente en la Universidad Politécnica de Madrid.

cación de las técnicas de análisis existentes a programas complejos y modulares, o bien se imponen restricciones al dominio abstracto para el que se analiza el programa, o bien exigen renunciar a resultados tan precisos como los que se obtienen para programas monolíticos pequeños. Esta tesis, como se verá en detalle, propone un marco de trabajo para el análisis preciso de programas modulares sin ninguna de estas dos limitaciones. Este marco de trabajo ha sido implementado satisfactoriamente y evaluado mediante diversos experimentos que han permitido comparar las distintas configuraciones de los parámetros del sistema. Además se han realizado experimentos de reanálisis incremental de programas modulares en presencia de modificaciones en algunos de los módulos del programa.

Por otra parte, uno de los desafíos más importantes en la investigación informática actual consiste en mejorar la capacidad de detectar automáticamente errores en programas y asegurar que un programa es correcto respecto a una determinada especificación, con el objetivo de producir *software* fiable. Por ello, la verificación de programas es un área importante de investigación. En el caso habitual de programas desarrollados por un equipo de programadores, es especialmente interesante el uso de técnicas de análisis y depuración que puedan considerar programas incompletos, de forma que se permita su uso aun cuando parte del programa no esté disponible. Esta característica permitiría utilizar técnicas avanzadas en el ciclo de edición-compilación-pruebas en fases anteriores a las que se han utilizado hasta ahora. Sin embargo, resulta habitual que los enfoques actuales de verificación de programas requieran que tanto el código que compone el programa como una especificación completa del mismo estén disponibles para el sistema verificador. Este requerimiento no es realista para programas grandes y complejos, y en particular en aquellos sistemas desarrollados hace tiempo, de los que ni siquiera se dispone de una documentación de usuario completa. La mejor solución en estos casos consiste en permitir una especificación parcial del programa, y utilizar técnicas de análisis avanzadas para inferir lo que falta en la especificación. Posteriormente, esta información puede utilizarse para verificar la corrección del programa respecto a la especificación parcial mencionada anteriormente. Este planteamiento resulta, evidentemente, en una fase de verificación más débil, pues se supone que los aspectos del programa para los que no se dispone de especificación están correctamente satisfechos por el código. No obstante, la información inferida puede utilizarse para verificar la corrección de la especificación disponible.

Esta tesis utiliza como lenguaje objeto el disponible en el sistema `Ciao`. `Ciao` es un sistema de programación lógica con restricciones, desarrollado en el grupo de investigación al que pertenece el doctorando, con un gran número de librerías y extensiones del lenguaje. El lenguaje del sistema `Ciao` contiene estructuras específicas para representar modularidad estricta y que comparten las mismas directrices de diseño descritas en el estándar ISO. Además, el sistema `Ciao` incluye `CiaoPP`, un preprocesador que implementa un amplio conjunto de herramientas avanzadas tales como analizadores, paralelizadores, especializadores, etc. En concreto, `CiaoPP` incluye un analizador dependiente del contexto y polivariante basado en la técnica de interpretación abstracta que es paramétrico respecto a los dominios abstractos utilizados. El analizador es utilizado para obtener información del programa por el resto de las herramientas de `CiaoPP`. Entre las herramientas disponibles en `CiaoPP`, también existe un comprobador de aserciones que toma una especificación (posiblemente parcial) del usuario en forma de aserciones, y comprueba automáticamente si el programa verifica esta especificación. En ambos casos, para esta tesis se han utilizado estas herramientas no modulares y se han extendido para tratar programas modulares e incompletos. Los marcos de trabajo desarrollados en el ámbito de esta tesis han sido integrados completamente en `CiaoPP` y actualmente forman parte del sistema. Aunque esta tesis se ha centrado en el lenguaje `Ciao`, las técnicas desarrolladas en esta tesis pueden extenderse fácilmente a otros paradigmas de programación.

A pesar de las ventajas de los lenguajes (C)LP debido a su fundamentación teórica, cuando se utilizan para problemas reales deben aplicarse muchas características extralógicas. En esta tesis se consideran también algunas de estas características.

Para cada una de las partes de esta tesis se han investigado los trabajos recientemente publicados por otros investigadores. En cada uno de los capítulos más importantes se puede encontrar una sección en la que se compara el trabajo realizado en esta tesis con lo que ya se ha publicado.

Objetivos de la tesis

El objetivo principal de esta tesis es el desarrollo, implementación y evaluación experimental de nuevas técnicas de compilación para tratar con programas lógicos (con restricciones) modulares.

Por una parte, esta tesis propone el desarrollo de herramientas de análisis basadas en la técnica de interpretación abstracta para inferir información precisa de la ejecución de programas lógicos en tiempo de compilación.

Además, objetivos fundamentales de la tesis son el uso de la información obtenida del análisis para desarrollar nuevos algoritmos y herramientas para verificar programas estructurados en módulos, así como resaltar las ventajas e inconvenientes de este enfoque.

Los objetivos concretos de esta Tesis son los siguientes:

- Extensión de las técnicas de análisis polivariante y dependiente del contexto a programas estructurados en módulos [PCH$^+$04], con la máxima precisión y eficiencia posibles. Desarrollo y obtención de un algoritmo de punto fijo intermodular.

- Análisis intermodular incremental: reanálisis eficiente de un programa modular después de la realización de cambios en algunos de los módulos del programa.

- Estudio experimental de los algoritmos anteriores [CPHB06], e investigación de nuevas técnicas que permitan mejorar la eficiencia del sistema de análisis. Evaluación de distintas alternativas de análisis módulo a módulo, así como exploración de otros enfoques.

- Extensión de las técnicas de verificación de programas para su aplicación a programas modulares [PCPH06]. Desarrollo de distintos algoritmos en función de la especificación proporcionada por el usuario y del coste del análisis.

- Aplicación de las técnicas de análisis y verificación modular a diversos casos de estudio. Aplicación a programas con interfaces con sistemas externos, en particular sistemas que acceden a bases de datos relacionales [CGC$^+$04a], y a la reducción del tamaño de las librerías del sistema.

- Integración de todas las técnicas desarrolladas en el sistema `CiaoPP` [HPBLG05].

Estructura del trabajo

Esta tesis está formada por tres partes principales, más una parte introductoria adicional para proporcionar los conceptos básicos sobre los que se construye el resto de la tesis. Las dos partes centrales de la tesis están dedicadas al análisis y a la verificación abstracta de programas modulares, respectivamente. Finalmente, La cuarta parte describe varias aplicaciones de las técnicas desarrolladas en diversos casos de estudio. En los siguientes apartados se describen en más detalle cada una de estas partes.

Parte I. Conceptos Fundamentales

La primera parte de esta tesis establece los conceptos básicos que se utilizan en las partes posteriores. Está compuesta por tres capítulos. En el primero, se clarifica el concepto de módulo que se utilizará a lo largo de la tesis. Durante los últimos años se han desarrollado diferentes enfoques de la programación lógica modular, con diferentes características y fundamentaciones semánticas, y sólo recientemente se ha definido un estándar oficial [PRO00]. El enfoque utilizado en esta tesis es similar a los utilizados más ampliamente en los sistemas actuales de programación lógica, tanto comerciales como no comerciales, y se ajusta al estándar. Más aún, debido a su naturaleza extralógica, puede extenderse fácilmente a otros paradigmas de programación diferentes de la programación lógica con restricciones.

La modularidad no sólo es de utilidad para la compilación separada, sino que también proporciona posibilidades adicionales. La más importante es que provee modularidad al propio análisis. Esto quiere decir que, si el lenguaje se extiende por medio de módulos de librería (como es el caso en el sistema `Ciao` utilizado para esta tesis) y el sistema de módulos es estricto, las características específicas del lenguaje que resultan especialmente difíciles para una herramienta de procesamiento de código fuente son solamente aplicables en los módulos que importan el módulo de librería que las define. Entre estas características se encuentran los procedimientos de orden superior, los procedimientos dinámicos y de manejo de bases de datos, la programación con restricciones, etcétera. Este enfoque permite configurar las herramientas para cada módulo. Por ejemplo, el tipo de análisis a utilizar puede ser diferente para distintos módulos: en algunos módulos puede ser más apropiado utilizar un algoritmo de análisis ascendente, mientras que en otros módulos un análisis descendente proporciona mejores resultados. Del mismo modo, se puede seleccionar el dominio abstracto de análisis para analizar un módulo determinado con precisión, y no utilizar el mismo dominio para otros módulos en los que no se requiere tan alta precisión. También los parámetros del dominio abstracto pueden ajustarse para proporcionar distintos niveles de eficiencia y precisión en diferentes módulos.

El segundo capítulo es una descripción introductoria de la interpretación abstracta en el contexto de la programación lógica. La interpretación abstracta es una de las técnicas más establecidas para obtener información sobre la ejecución de un programa sin necesidad de ejecutarlo [CC77a, CC92], que consiste en simular el funcionamiento de un programa utilizando una *abstracción* de los valores reales, en lugar de estos propios valores concretos. El resultado de la interpretación abstracta será una *aproximación segura* del comportamiento del programa. La interpretación abstracta tiene varias ventajas respecto al diseño de casos de prueba para el análisis de la ejecución de un programa, pues permite capturar el comportamiento del programa en todas las ejecuciones y contextos posibles, en lugar de los casos particulares de los casos de prueba diseñados por un desarrollador. Además, no requiere la ejecución del programa, que en algunos casos podría no ser posible.

Aunque la técnica de interpretación abstracta puede aplicarse a cualquier paradigma de programación, en el caso de la programación lógica presenta diversas ventajas por su fundamentación teórica, y permite el uso de técnicas adicionales que producen información de análisis más precisa. Mediante la interpretación abstracta se pueden analizar programas con diversos grados de precisión: por una parte, mediante la utilización de diferentes dominios abstractos de análisis, y por otra parte a través de la utilización de algoritmos sofisticados. Entre estos últimos, caben destacar el análisis dependiente del contexto y la polivarianza. El análisis dependiente del contexto permite obtener, sobre un procedimiento, información que depende del contexto en el que dicho procedimiento fue llamado, en lugar de obtener información más general (pero menos precisa) independiente del contexto, es decir, común a todas las llamadas a dicho procedimiento en el programa. Por otra parte, la polivarianza permite obtener resultados de análisis separados para distintos contextos de llamada, en lugar de unificar los contextos de las diversas llamadas al procedimiento para producir un resultado de análisis único, aunque menos preciso. Tanto la polivarianza como la dependencia del contexto proporcionan una mayor precisión en los resultados, pero a cambio requieren mayor cantidad de recursos de cómputo y añaden complejidad al algoritmo de análisis. Estas dos técnicas aplicadas conjuntamente hacen de la interpretación abstracta una herramienta extremadamente precisa para obtener información de un programa.

El tercer capítulo de esta primera parte introduce `CiaoPP`, el procesador del sistema `Ciao`, y se describen brevemente en forma de tutorial los componentes más relevantes del preprocesador, introduciendo algunos conceptos del lenguaje de aserciones de `Ciao` y sus aplicaciones. El objetivo de este capítulo es proporcionar al lector una introducción rápida a `CiaoPP`, sistema en el que se han integrado las técnicas desarrolladas en esta tesis.

Parte II. Análisis de Programas Modulares

Como se ha mencionado antes, las técnicas de análisis de programas constituyen la herramienta básica para el resto de los objetivos de esta tesis. Sin embargo, el análisis plantea por sí mismo importantes problemas cuando se tratan grandes programas modulares.

A pesar de los avances realizados hasta ahora en este área, las técnicas actuales de análisis polivariantes y dependientes del contexto utilizadas exigen que la totalidad del programa se encuentre disponible para el analizador. Sin embargo, frecuentemente se produce la situación en la que no es posible cargar el programa completo en el analizador y realizar el análisis de todo el programa monolíticamente. En algunos casos, los requerimientos de memoria del analizador desbordan la memoria disponible, de forma que el analizador no cabe en memoria si se carga y se analiza a la vez todo el código del programa. En otros casos, no es posible cargar en memoria todo el código porque algunos fragmentos están todavía en desarrollo, y por tanto no están completamente implementados. Y, por último, también puede producirse el caso en el que, después de algunas modificaciones del código, sea más rápido reanalizar incrementalmente los módulos implicados en la modificación, en lugar de ejecutar un costoso análisis de todo el código desde el principio. En esta parte de la tesis se tratan todos estos aspectos, proponiendo un marco de trabajo general y paramétrico que proporciona una visión unificada de estos problemas.

Una posible solución a los problemas anteriormente mencionados consiste en utilizar únicamente dominios abstractos con ciertas características especiales [CDG93, MJB00, CC02a] (por ejemplo los dominios denominados composicionales), en los cuales no se pierde precisión si se hace análisis sin conocer de antemano los patrones de entrada del módulo. Para analizar el programa, bastaría con ir analizando los módulos en orden ascendente, utilizando la información previamente obtenida de los módulos ya analizados. Sin embargo, existen muchos dominios abstractos que han mostrado su interés práctico y no cumplen dichas propiedades. En [PH00], el grupo de investigación al que pertenece el doctorando identificó los problemas más importantes relacionados con el análisis dependiente del contexto y polivariante de programas modulares, pero sin proponer soluciones concretas. En [BdlBH$^+$01] se propone el primer algoritmo que resuelve en gran medida dichos problemas. Sin embargo, la primera visión global del problema y la propuesta de un marco paramétrico capaz de englobar diferentes algoritmos no se realiza hasta [PCH$^+$04], ya como parte de la presente Tesis.

Por otra parte, en el apartado anterior se han descrito dos técnicas adicionales que permiten mejorar los resultados de la interpretación abstracta de programas lógicos (con restricciones). Tanto la posibilidad de disponer de información dependiente del contexto como la polivarianza permiten tener información más precisa de las llamadas y resultados de los procedimientos del programa. No obstante, ambas requieren el desarrollo de un algoritmo de punto fijo intermodular para obtener la precisión máxima en los resultados del análisis. La forma en la que se obtiene este punto fijo intermodular es relevante tanto para la precisión como para la eficiencia del análisis. En esta parte de la tesis se resaltan diversos parámetros del algoritmo intermodular y se evalúan empíricamente, y se propone un algoritmo de punto fijo intermodular paramétrico, instanciándolo para diversas políticas de selección de módulos (scheduling). Además, se ha evaluado experimentalmente el análisis intermodular incremental para algunos cambios específicos en el código, mostrando las ventajas de utilizar un enfoque intermodular, en lugar de analizar

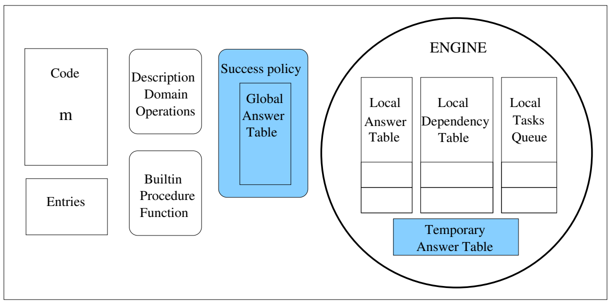


Figura 1: Análisis de programas modulares: el nivel local.

todo el programa desde el principio.

Análisis de Programas Modulares: El Nivel Local

El análisis de programas modulares presentado en esta tesis tiene como objetivo el analizar un programa compuesto por varios módulos a partir del análisis sucesivo de cada uno de los módulos que lo constituyen. Se define como nivel local el conjunto de tareas y elementos de información que se consideran cuando se analiza uno de los módulos del programa. El análisis de un módulo de un programa modular tiene características propias, diferentes del análisis monolítico de un programa no estructurado en módulos. El aspecto más importante que debe considerarse en este nivel es que el código a analizar es *incompleto*, pues el código de los procedimientos importados de otros módulos no está disponible para el analizador. Es decir, durante el análisis de un módulo m pueden producirse llamadas de la forma $P : CD$ tales que el procedimiento P esté importado desde otro módulo m', en lugar de estar definido en m. El problema de determinar cuál es el valor AD de respuesta de las llamadas a P se denomina *problema de los resultados importados*. Por otra parte, para obtener de m' la información de análisis lo más precisa posible, es necesario propagar el patrón de llamada $P : CD$ desde m a m', con el fin de utilizarlo cuando se analice m'. Denominamos este problema como el *problema de las llamadas importadas*.

Se puede observar que estos dos problemas son característicos de un análisis dependiente del contexto y polivariante. Los analizadores monovariantes solamente necesitan propagar un único patrón de llamada entre módulos, aquel que engloba a todos los patrones de entrada de todos los módulos que lo importen. Por otra parte, los analizadores independientes del contexto no necesitan propagar información de patrones de entrada, pues es suficiente analizar los módulos en orden ascendente para disponer de la información de análisis de los procedimientos importados. El inconveniente de estos analizadores es que obtienen información menos precisa que el tipo de análisis considerado en esta tesis.

Para poder realizar el análisis de un módulo en el contexto del análisis modular, es necesario mantener por tanto una estructura de datos para intercambiar información entre los análisis de los distintos módulos del programa. Esta estructura de datos se denomina *Tabla de Resultados Global (GAT, Global Answer Table)*, y contiene el conjunto de resultados de análisis obtenidos hasta el momento para cada uno de los módulos del programa. Como el análisis es dependiente del contexto y polivariante, por cada procedimiento de cada módulo es posible que existan varias entradas en la GAT de la forma $P : CD \mapsto AD$, que indica que el resultado de analizar el procedimiento P con el patrón de llamada CD produce como resultado el patrón de respuesta (también denominado de resultado) AD.

El *problema de los resultados importados* se soluciona aplicando una *política de resultados (SP, success policy)*. Esta política es necesaria porque, cuando al analizar un módulo m se necesita el resultado de un patrón de llamada $P : CD$ a un procedimiento P importado de otro módulo m', se producirá frecuentemente la situación en la que no exista en la GAT una entrada con exactamente el mismo patrón de llamada. En este caso, es posible aprovechar la información existente en ese momento en la GAT para calcular una respuesta temporal para el patrón de llamada $P : CD$ y continuar con el análisis del módulo m. La forma de calcular esta respuesta temporal viene dada por la SP. Se pueden definir diversas SP con distintos grados de precisión, si bien pueden clasificarse como sobreaproximaciones o subaproximaciones del resultado exacto $AD^{=}$ (que es el que calcularía el análisis no modular, o *monolítico*). Utilizaremos la denominación SP^{+} para referirnos a políticas que producen sobreaproximaciones, y SP^{-} para las que producen subaproximaciones.

Por su parte, la solución al *problema de las llamadas importadas* requiere que se almacenen en otra estructura de datos global, la *tabla de respuestas temporal (TAT, temporary answer table)* los patrones de entrada a procedimientos importados de m' generados por el análisis del módulo m, y cuyos resultados han sido aproximados mediante la SP. Cuando se realice el análisis de m', estarán disponibles para el analizador los patrones de entrada contenidos en la TAT. La *política*

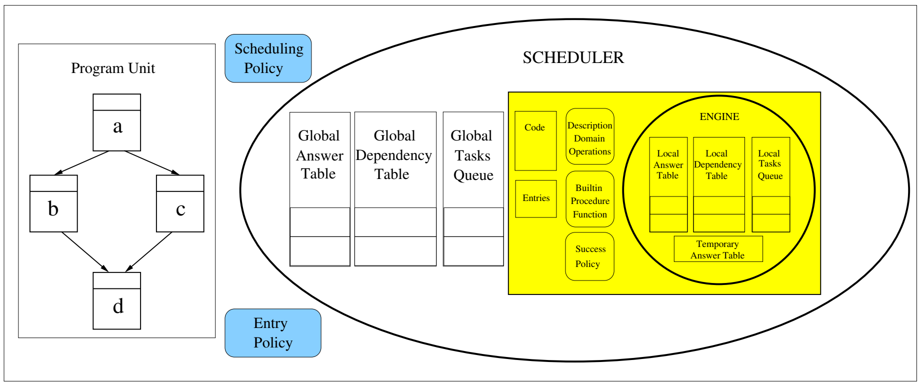


Figura 2: Análisis de programas modulares: el nivel global

de entrada (entry policy) permite seleccionar qué entradas se van a considerar para cada módulo. Dependiendo de cómo se analice un módulo, y de cuál sea el objetivo del análisis, se pueden definir diversas políticas de entrada: si el objetivo es obtener la mayor precisión posible (por ejemplo, para calcular un punto fijo intermodular), pueden utilizarse las entradas de la *TAT* para obtener la información más precisa posible para los módulos que importan el módulo que se está analizando; por otra parte, si el objetivo es verificar la corrección de un único módulo, puede ser suficiente analizar el módulo para los patrones más generales de los procedimientos exportados.

Análisis de Programas Modulares: El Nivel Global

El nivel global del análisis de programas modulares considera las relaciones que se producen entre los análisis de los distintos módulos y los parámetros que controlan el sistema.

Las estructuras de datos que intervienen en el nivel global son las siguientes. En primer lugar, la tabla de respuestas global *GAT*, descrita anteriormente. La segunda estructura de datos es la *tabla de dependencias global (GDT, global dependency table)*, que contiene las relaciones entre patrones de entrada de procedimientos exportados de distintos módulos. De esta forma, es posible propagar de la forma más precisa posible qué partes del programa necesitan reanalizarse. Esta tabla es necesaria para mantener la tercera estructura de datos, la *cola de tareas global (GTQ, global task queue)*, que contiene los eventos por procesar en el nivel global, que normalmente serán los módulos que requieren reanálisis.

El nivel global tiene tres parámetros. El primero de ellos es el programa (también denominado *program unit*), conjunto de módulos relacionados mediante la relación de importación, que tienen un nodo raíz, que corresponde con el módulo principal (denominado *top level*). El segundo parámetro es la *política de entrada*, que determina la forma en que deben inicializarse las estructuras GTQ y GAT. Por último, el tercer parámetro es la *política de planificación* (*scheduling policy*), que define el orden en el que se deben procesar las entradas de la GTQ.

El análisis de un programa modular comienza inicializando las estructuras de datos globales según indica la *política de entrada*. En cada paso, la *política de planificación* determina el conjunto de entradas de la GTQ que se procesarán de un módulo m, y se eliminan de la GTQ. Se analiza el módulo m, y posteriormente se actualizan las estructuras de datos globales. Como resultado de esto, se pueden añadir nuevos eventos a la GTQ. El análisis termina cuando no hay eventos pendientes de procesar en la GTQ, o cuando la estrategia de planificación no selecciona más eventos.

Las tres estructuras de datos globales se actualizan después del análisis. Por una parte, en la GAT se cambian los resultados del análisis que mejoran los valores anteriores en la tabla (dependiendo de la *SP*), y se introducen los valores nuevos para los procedimientos importados. Por otra parte, se introducen en la GDT las dependencias nuevas encontradas entre los procedimientos exportados por el módulo m y los importados de otros módulos. Por último, se introducen en la GTQ los elementos de los módulos que importan m que se vean afectados por los cambios en la GAT.

El esquema del nivel global permite el análisis incremental de programas: si después de analizar el programa se producen modificaciones en el código fuente del módulo m, basta con lanzar el análisis modular, marcando todas las entradas de la GTQ de m.

Políticas de Planificación

La *política de planificación* determina el orden en el que se procesan los eventos que aparecen en la GTQ. Se pueden definir diversas políticas de planificación en el análisis de programas modulares.

Por una parte, se puede considerar el caso en el que varios programadores o equipos desarrollan diferentes partes de un programa. En este caso, los programadores pueden utilizar el análisis separado de sus respectivos módulos para detectar lo antes posible errores sin tener que esperar a que el resto del programa esté completamente terminado. se puede realizar una planificación manual, de forma que sean los programadores los que decidan cuándo y qué módulo se analiza en cada momento. El requisito fundamental de la política de planificación manual es que en todo momento los resultados deben ser correctos. Por ello, es imprescindible utilizar una *política de resultados* que obtenga sobreaproximaciones (SP^+).

Por otro lado, se puede plantear una política de planificación automática, de manera que es el sistema el que decide qué módulo se debe analizar en cada momento, y respecto a qué patrones de entrada. Existen situaciones en las que el usuario está interesado en obtener los resultados de análisis lo más precisos posible. El sistema debe por tanto planificar los módulos de forma adecuada para llegar a un punto fijo intermodular. A diferencia de la política manual, en este caso se puede utilizar tanto SP^+ como SP^-, pues los resultados deben ser correctos solamente cuando se llega al punto fijo intermodular.

Resultados Experimentales

El esquema de análisis modular descrito ha sido implementado completamente en CiaoPP. Esta implementación permite parametrizar el análisis modular de diversas formas, como se ha indicado en los apartados anteriores. Entre los aspectos evaluados, se pueden destacar:

Monolítico vs. modular En primer lugar, se ha comparado el análisis monolítico (todo el programa a la vez) con la obtención de un punto fijo intermodular. Aunque es esperable que el análisis modular de un programa por primera vez sea más lento que el análisis monolítico (debido a la sobrecarga de cargar y descargar módulos, etc.), es interesante estudiar en qué medida se produce.

Políticas de planificación intermodular Otro aspecto a estudiar está relacionado con la influencia de la política de selección de módulos en la eficiencia del análisis. Se han estudiado dos políticas sencillas de selección de módulos que básicamente recorren el grafo de dependencias intermodulares en orden ascendente o descendente, respectivamente.

Políticas de resultados Se han comparado dos políticas de resultados (*success policies*) de uso común, que obtienen sobreaproximaciones y subaproximaciones de las llamadas a procedimientos importados.

Análisis incremental de programas modulares Por último, se ha comparado el análisis monolítico de un programa con el reanálisis del mismo después de realizar determinadas modificaciones en el código fuente. Esta comparación ilustra las ventajas de analizar exclusivamente el módulo modificado (y los módulos relacionados que se vean afectados por el cambio) en lugar de analizar todo el programa desde cero.

Se han estudiado tres tipos diferentes de cambios en el código fuente: en primer lugar, un cambio simple que mantiene los mismos resultados de análisis (por tanto, ningún otro módulo se ve afectado por el cambio); una modificación en el código que hace que los patrones de salida de los procedimientos exportados tengan una respuesta más precisa; y por último un cambio que produce resultados más generales para los procedimientos exportados.

Para los experimentos se han considerado dos dominios abstractos de análisis de “modos”: Def [dlBH93], que almacena propiedades (en particular, información de *groundness*) mediante implicaciones proposicionales, y *Sharing-freeness* [MH91], que contiene información sobre compartición de variables lógicas y de variables libres de forma conjunta.

Análisis de un programa modular desde cero En este caso, el análisis intermodular resulta, como es de esperar, más lento que el análisis monolítico (con todo el programa cargado en memoria), debido al coste de carga y descarga del código y de la información de análisis relacionada, y la limitación determinada por la imposibilidad de analizar procedimientos que no estén en el módulo que se está procesando. Este experimento está detallado en la sección 6.2.2. Sin embargo, el tiempo de análisis desde cero es todavía razonable, excluyendo el caso del dominio $Sharing - freeness$ con una política de resultados sobreaproximadora.

Los resultados experimentales también propocionan evidencia de que el análisis modular implica un consumo de memoria máximo inferior al análisis monolítico, que en algunos casos puede ser conveniente pues permitiría analizar programas de tamaño crítico que el análisis monolítico no es capaz de procesar.

Respecto a las dos estrategias de selección de módulos estudiadas, se pueden observar solamente pequeñas diferencias en el tiempo de ejecución. Este resultado parece reflejar que el orden de los módulos no es tan relevante como se podría esperar a priori cuando se analiza un programa modular.

Reanálisis de un programa modular después de un cambio en el código En todos los tipos de modificaciones evaluadas se puede constatar que el análisis modular incremental es más eficiente que volver a realizar el análisis monolítico desde cero. Además, es importante señalar que la complejidad del dominio influye en el resultado de las pruebas de análisis incremental: cuanto más complejo es el dominio mayores son las ventajas del análisis incremental. Esto sugiere que el análisis intermodular puede hacer factible el uso de dominios precisos pero costosos sobre programas modulares. En la sección 6.2.3 se muestra que, incluso en los peores casos, el coste de reanálisis, aunque próximo, siempre es inferior al análisis monolítico desde cero. Esto indica que, aun en presencia de los cambios más agresivos en un módulo, el análisis intermodular no es más costoso que el análisis monolítico.

Análisis de tipos para la verificación de programas modulares

Por último, el capítulo 7 presenta dos técnicas para mejorar la eficiencia del análisis modular en dominios de tipos. En el análisis de tipos tradicional para programas lógicos [JB92, VB02] se infieren nuevos tipos durante el análisis, lo que hace que el marco de trabajo general propuesto en los apartados anteriores sea excesivamente costoso. Para mejorar la eficiencia del análisis, se propone en primer lugar un modelo mediante el cual los tipos inferidos por el analizador son reemplazados por tipos definidos en el código fuente (tanto definidos por el usuario como presentes en las librerías del sistema). De este modo, además de reducir el coste del análisis, se garantiza una convergencia más rápida hacia el punto fijo intermodular.

La segunda técnica está basada en el uso de aserciones de tipos paramétricas en la especificación del programa. Este tipo de aserciones es especialmente útil en las librerías que implementan procedimientos de manipulación de datos genéricos (como por ejemplo listas o árboles binarios), de forma que no sea necesario reanalizarlos cada vez que se analiza un programa que utilice la librería. De este modo, se pueden instanciar parámetros en la aserción de la librería (estas aserciones cumplen el rol de interfaz del módulo de librería) de acuerdo al patrón de llamada real, y se limitan a reutilizar el patrón de salida sin necesidad de analizar el módulo de librería. Esto permite incorporar algunas características específicas de lenguajes de programación lógica fuertemente tipados [MO84, HL94b] sin tener que cambiar el lenguaje y mientras se siguen utilizando tipos descriptivos, tipos que describen aproximaciones de la semántica no tipada.

Estas técnicas se han evaluado experimentalmente, constatando la relevancia de su aplicación en el contexto de programas modulares, especialmente en el caso del análisis incremental, al reanalizar un programa sobre el que se han realizado cambios en algunos de sus módulos.

Parte III. Comprobación de Aserciones en Tiempo de Compilación en Programas Modulares

En el área de verificación de programas se ha mostrado gran interés en la utilización de la información obtenida mediante técnicas de interpretación abstracta [Bou93, CLMV99, PBH00a, Cou03a]. Sin embargo, como en el caso del análisis, los enfoques existentes para la verificación de programas modulares [CLV98] no permiten el uso de las técnicas más sofisticadas, tales como la polivarianza y la dependencia del contexto. Además, estos enfoques requieren una especificación completa del programa, como los desarrollados para el paradigma orientado a objetos (véase [LM05]).

Por otra parte, muchas de las herramientas de depuración y verificación existentes comprueban los programas respecto a una especificación proporcionada en forma de aserciones [PBH00b, DNTM89] escritas por el usuario. Sin embargo, la mayor parte de estas herramientas no consideran programas estructurados en módulos. Además, la especificación debe ser completa, en el sentido de describir todos los procedimientos del programa.

En esta parte de la tesis se propone un marco de trabajo para la comprobación de aserciones en programas lógicos modulares, basado en información de análisis global. La especificación del programa mediante aserciones puede ser parcial: las aserciones no introducidas por el usuario son inferidas mediante el análisis. La especificación se escribe en términos de las aserciones del sistema

Ciao [BCHP96, BDM97, DNTM89, M9́5, PBH00b]. Obsérvese que este sistema de aserciones está diseñado para tratar las características específicas del paradigma de programación lógica, en particular para permitir diversos usos de un procedimiento. Por ejemplo, el procedimiento `append/3` pude utilizarse para concatenar listas, descomponer listas, comprobar o buscar el prefijo de una lista determinada, etc.

El trabajo realizado en esta parte está basado en la propuesta de [PBH00d] de comprobación de aserciones en programas no modulares. Además, en esta tesis se utiliza la información polivariante generada por el análisis.

Cuando se considera un programa compuesto por varios módulos, la corrección de un módulo puede verse como condicional, de forma que esta noción de corrección depende de la corrección del resto de los módulos del programa. Usando este enfoque, en esta tesis se estudian diferentes conceptos de corrección de aserciones en programas modulares. De esta forma, este trabajo ha permitido desarrollar diversas variantes de la verificación intermodular, que implican diferentes formas de obtener la información de análisis del programa, como se verá a continuación.

Aserciones

En esta tesis se consideran dos tipos fundamentales de aserciones [PBH00c].[†] El primer tipo es el de aserciones `success`, que se utilizan para expresar propiedades que deben cumplirse a la finalización correcta del cómputo de un procedimiento (*postcondiciones*). En el momento de llamar al procedimiento, debe satisfacerse una determinada *precondición*. Las aserciones `success` pueden expresarse en este lenguaje mediante una expresión de la forma: `success` $P : \mathit{Pre} \Rightarrow \mathit{Post}$, donde P es un descriptor de procedimiento y *Pre* y *Post* son pre- y post-condiciones respectivamente. Sin pérdida de generalidad, se considerará que *Pre* y *Post* se corresponden con sustituciones abstractas (λ_{Pre} y λ_{Post} respectivamente) sobre el conjunto de argumentos del procedimiento P.

El segundo tipo de aserciones expresa propiedades que deben cumplirse en cualquier llamada a un procedimiento dado. Estas propiedades son similares a las *precondiciones* utilizadas en verificación de programas. Estas aserciones tienen la

[†][PBH00c] incluye también otros tipos de aserciones que están fuera del objetivo de esta tesis.

forma: `calls` $P : Pre$, y deben interpretarse como: “en todas las activaciones de P, Pre debe cumplirse en el estado de la llamada.”

A cada aserción se le puede asignar un *estado*. El estado indica si la aserción se refiere a propiedades previstas (*intended*) o reales, y la relación entre esta propiedad y la semántica del programa. De este modo, si una aserción se cumple en un conjunto de consultas Q se dice que la aserción está *comprobada* (*checked*) respecto a Q. Una aserción también puede ser *falsa*, si se sabe que hay al menos un patrón de llamada (o de finalización) en la semántica concreta que incumple la propiedad de la aserción. Si se puede probar la falsedad de una aserción, se le da el estado `false`. Finalmente, una aserción que expresa una propiedad que se cumple para cualquier consulta inicial es una aserción *verdadera* (*true*). Si se puede probar esto durante la comprobación en tiempo de compilación, de forma independiente del contexto de llamada, la aserción se rescribe con el estado `true`.

Obsérvese que la diferencia entre aserciones comprobadas y verdaderas es que estas últimas se cumplen para cualquier contexto. Por tanto, el hecho de que una asercion sea verdadera implica que también está comprobada.

Por último, una aserción para la que no se puede determinar ninguno de los estados anteriores en tiempo de compilación se dice que es una aserción pendiente de comprobar (*check*). Esta aserción expresa una propiedad prevista. Puede cumplirse o no en la versión actual del programa. Este es el estado por defecto: si una aserción no tiene un estado explícitamente, se supone que su estado es *check*. Antes de realizar la comprobación de aserciones en tiempo de compilación, todas las aserciones escritas por el usuario tienen este estado.

En el sistema desarrollado, la comprobación de aserciones debe estar precedida por una fase de análisis, y consiste fundamentalmente en comparar las aserciones con la información obtenida por el análisis. Por tanto, la comprobación de aserciones se realiza realmente en el dominio abstracto. Si es posible verificar que una aserción es correcta en el contexto determinado por el dominio abstracto, entonces se puede cambiar su estado a `checked`. Del mismo modo, si se puede determinar que una aserción es incorrecta en el dominio abstracto, se puede cambiar su estado a `false`. La corrección de la interpretación abstracta garantiza de forma suficiente la comprobación de las aserciones sobre el dominio concreto.

Es importante destacar que la precisión del análisis influye directamente sobre el número de aserciones que el sistema es capaz de comprobar que son correctas (o incorrectas). Por ello, cuanto más preciso sea el análisis del programa (por ejemplo, mediante analizadores dependientes del contexto y polivariantes), más aserciones se pueden comprobar o bien determinar su falsedad.

Comprobación de aserciones en un módulo

El marco de trabajo de análisis modular descrito en los apartados anteriores es independiente del lenguaje de aserciones. Sin embargo, las aserciones pueden contener información relevante para el analizador. Con este objetivo, cuando se computa el análisis del módulo m, $LAT = analysis(m, E, AT)$ –LAT es la tabla de resultados local del análisis del módulo m con el conjunto de patrones de entrada del análisis E y utilizando la tabla de resultados global AT–, también puede referirse a información recogida directamente de las aserciones, en lugar de hacerlo de otros pasos anteriores del análisis modular. Esto da lugar a políticas de entrada y de resultados adicionales respecto a las descritas anteriormente.

Cuando se comprueban las aserciones de programas modulares, un módulo determinado puede considerarse tanto en el contexto de un programa o separadamente, tomando en consideración solamente los procedimientos importados. Cuando un módulo se trata en el contexto de un programa, el contexto de llamada de un módulo m se denomina *conjunto de llamadas iniciales*.

Se dice que el conjunto de llamadas iniciales a un módulo m es *válido* si y sólo si todas las aserciones `calls` de cada procedimiento exportado de m obtienen estado `checked` respecto a dicho conjunto de llamadas.

Se puede definir que un módulo m es *parcialmente correcto en contexto* con respecto a un conjunto de llamadas iniciales si y sólo si: (1) todas las aserciones `calls` de m obtienen estado `checked` respecto al conjunto de llamadas; (2) todas las aserciones `success` tienen estado `true` o `checked` respecto a dicho conjunto; y (3) todas las aserciones `calls` de procedimientos importados por m tienen estado `checked` respecto al conjunto de llamadas inicial.

Un módulo m es *parcialmente correcto* si y sólo si es parcialmente correcto en contexto respecto a cualquier conjunto válido de llamadas iniciales.

Como se ha dicho antes, las aserciones se comprueban respecto a toda la información de análisis disponible después de analizar el código. Esta información de análisis es polivariante y cubre todos los puntos de programa en los que se llama a cada procedimiento. Se puede por tanto demostrar que se verifica lo siguiente:

Sea m un módulo y $LAT = analysis_{SP^+}(m, \mathcal{CP}_m^{Asst}, AT)$, donde AT es una tabla de respuestas con sobreaproximaciones de (algunos módulos de) $imports(m)$.

El módulo m es parcialmente correcto si todas las aserciones `success` *obtienen estado* `true` *con respecto a LAT y todas las aserciones* `calls` *en m y los módulos importados por m tienen estado* `checked` *respecto a LAT.*

Esta proposición considera la corrección de un módulo independientemente del contexto de llamada del módulo, pues el punto de inicio del análisis es el conjunto de precondiciones de las aserciones `pred` (representado con $\mathcal{CP}_m^{Asst}$). Respecto a la tabla de respuestas de los procedimientos importados AT, ésta debe contener sobreaproximaciones, pero puede estar incompleta o incluso vacía (pues la política de resultados es sobreaproximadora: SP^+). No obstante, cuanto más precisa es AT, más aserciones obtienen estado `true` o `checked`.

Si los módulos importados por m no están implementados, todavía se puede utilizar la información de aserciones (si existe) para obtener una LAT más precisa. En este caso, La corrección de m no se puede garantizar. En su lugar, se puede proporcionar una noción más débil, corrección parcial condicional. Es importante tener en cuenta que en este caso el análisis se basa en aserciones escritas por el usuario que posiblemente no han sido verificadas.

Sea m un módulo, y $LAT = analysis_{SP^+}(m, \mathcal{CP}_m^{Asst}, \mathcal{AT}_m^{Asst})$. m es condicionalmente parcialmente correcto si todas las aserciones `success` *tienen estado* `true` *y todas las aserciones* `calls` *de procedimientos en m y los módulos importados por m tienen estado* `checked` *respecto de LAT.*

Esta noción más débil de corrección se convierte en corrección parcial cuando se considera todo el programa en conjunto, como se verá a continuación.

Comprobación de aserciones en un programa modular

La comprobación de aserciones en un programa modular compuesto por varios módulos (también denominado *program unit*) difiere de la comprobación de aserciones en un único módulo en diversos aspectos. Por una parte, se puede obtener el conjunto más preciso de llamadas iniciales a un módulo del programa, a partir de las llamadas realizadas por los módulos del programa que lo importan. Por otra parte, los patrones de entrada de procedimientos importados de otros módulos también pueden ser más precisos si se considera todo el programa modular. Esto da lugar a la noción de corrección de un programa modular:

> *Sea m_{top} el módulo principal que define un programa modular U. U es parcialmente correcto si y sólo si m_{top} es parcialmente correcto, y todos los módulos que dependen de m_{top} son parcialmente correctos en contexto respecto de los conjuntos de llamadas iniciales inducidas por las llamadas iniciales de m_{top}.*

A continuación se proponen tres algoritmos para la verificación de programas modulares, en función de la información de análisis disponible.

Verificación de un programa modular con información de análisis intermodular. En este primer caso se describe un algoritmo básico para comprobar aserciones en un programa modular, aprovechando la modularidad del programa.

Antes de la comprobación de aserciones se realiza el análisis intermodular del programa. Una vez hecho esto, se recorren los módulos uno a uno y se comprueban las aserciones locales al módulo y las de los procedimientos importados.

Como la información de análisis utilizada en cada módulo ha sido generada por el análisis intermodular, los resultados de la comprobación de aserciones son correctos independientemente de la política de resultados SP utilizada. Es más, si se utiliza una política SP^-, la precisión del análisis intermodular es similar a la del análisis monolítico, y los resultados de la comprobación de aserciones igual de precisos (con la excepción de algunas aserciones que sean `true` con el análisis monolítico, y `checked` con el análisis intermodular).

Verificación de un programa modular sin información de análisis intermodular. Como se ha comentado anteriormente, cada aserción `calls` de un procedimiento exportado P se comprueba en todos los módulos que importan dicho procedimiento. Si en todos los módulos del programa modular se puede determinar que dicha aserción `calls` obtiene el estado `true`, entonces esto quiere decir que la precondición de la aserción es una aproximación de todas las llamadas a P en el programa. Por tanto, las aserciones `calls` se pueden utilizar como el punto de inicio del análisis de cada módulo en el programa.

Esto da lugar a un escenario para la comprobación de aserciones en el que no es necesario realizar un análisis intermodular previo, y que tiene como objetivo probar que todos los módulos son condicionalmente correctos, en lugar de probar que son correctos en contexto. Este nuevo algoritmo analiza una sola vez cada uno de los módulos del programa y comprueba sus aserciones, utilizando la información de las aserciones tanto para definir el conjunto de llamadas iniciales de cada módulo, como para obtener información de los procedimientos importados de otros módulos. Con este nuevo enfoque se puede afirmar lo siguiente:

> *Sea m_{top} un módulo que define un programa modular U cuyo módulo principal es m_{top}. Si cada módulo $m \in U$ es condicionalmente parcialmente correcto, y m_{top} es parcialmente correcto, entonces U es parcialmente correcto.*

De esta forma, si las aserciones obtienen el estado `true` o `checked` con este nuevo algoritmo, también lo obtendrán si se utiliza la información del análisis intermodular. Por tanto, si este algoritmo comprueba que las aserciones son correctas, entonces no es necesario ejecutar un costoso análisis intermodular de todo el programa.

Intercalado de análisis modular y comprobación de aserciones. El mayor inconveniente del algoritmo anterior es que puede no ser capaz de determinar la corrección parcial de un programa modular si el usuario ha introducido pocas aserciones para los procedimientos exportados, o bien éstas no son suficientemente precisas. En este caso, es necesario suministrar esta información, así como incorporar cierto grado de propagación automática de patrones de entrada y de salida entre los módulos durante el proceso de comprobación.

La idea básica consiste en intercalar análisis y comprobación de aserciones en el algoritmo de análisis intermodular. La ventaja principal de este enfoque es que se detectarán los errores lo antes posible, sin necesidad de calcular un costoso punto fijo intermodular, y al mismo tiempo se propagan los patrones de entrada y salida entre los módulos.

En este nuevo algoritmo, se puede concluir lo siguiente: si se detecta que una aserción obtiene el estado `checked` o `false` en un paso intermedio del algoritmo, al final del proceso continuará en ese mismo estado. Si la aserción no se ha verificado o demostrado que es falsa, su estado podría cambiar en pasos posteriores, pues la información de análisis podría ser más precisa en las iteraciones subsiguientes.

Por otra parte, si en el algoritmo de análisis intermodular se utiliza una política de resultados subaproximante SP^-, se verifica el siguiente resultado adicional: si en cualquier paso intermedio del algoritmo el estado de una aserción es `check` o pasa a ser `false`, entonces al final del proceso el estado de dicha aserción será como mucho `check`. De este modo, en este caso se puede parar el procesamiento del programa modular tan pronto como se detecten aserciones con estado `false` o `check`, pues en ningún caso se conseguirán verificar estas aserciones.

Parte IV. Aplicaciones

La parte final de la tesis está formada por dos aplicaciones sobre programas reales de las técnicas desarrolladas. En el primer caso, una aplicación del análisis de programas modulares de especial relevancia es la posibilidad de utilizar esta técnica en programas con interfaces con sistemas externos de los que no se dispone el código fuente. Dentro de éstos, es particularmente importante la utilización de bases de datos relacionales: por su propia estructura, desde el punto de vista del análisis es posible determinar en tiempo de compilación gran cantidad de propiedades (tipos de los argumentos, terminación, determinismo, modos y grado de instanciación de los argumentos, etc.). Esta técnica se aplica por tanto a programas con accesos a sistemas de bases de datos, con el fin de mejorar los accesos a éstos con información en tiempo de compilación obtenida mediante interpretación abstracta. Este caso de estudio es un ejemplo claro de cómo se pueden utilizar técnicas avanzadas en programas reales.

En segundo lugar, la interpretación abstracta ha jugado un papel fundamental en el área de especialización y optimización de programas, gracias al concepto de especialización abstracta [PH03]. En particular, el análisis polivariante ha permitido el desarrollo de la especialización abstracta múltiple. No obstante, tampoco en este área se han desarrollado técnicas efectivas de especialización múltiple de programas modulares. En el segundo caso de estudio, la especialización de programas modulares se utilizará en particular para la especialización de las librerías del sistema utilizadas por un programa dado, con el objetivo de reducir el tamaño del mismo para su ejecución en dispositivos con recursos limitados de computación (dispositivos móviles, etc.). Este algoritmo está basado en el algoritmo de especialización múltiple no modular de [PH99]. Este enfoque es innovador respecto al modelo utilizado tradicionalmente, consistente en proporcionar librerías especiales para estos dispositivos, con importantes limitaciones[‡].

Contribuciones

A continuación se enumeran las principales contribuciones de esta tesis. Varios resultados obtenidos han sido publicados y presentados en foros internacionales; esta situación se menciona explícitamente. Se puede observar que la mayor parte de los trabajos publicados han sido realizados en colaboración con otros investigadores tanto del grupo de investigación CLIP, al que pertenece el doctorando, como de otros centros de investigación. En todos los trabajos la contribución del doctorando ha sido muy relevante, como lo demuestra el hecho de que el doctorando está entre los primeros autores en todas las publicaciones, en las que los autores están ordenados por su grado de aportación en los artículos. Además, esta colaboración ha permitido al doctorando aprender nuevos enfoques debido a la distinta procedencia y área de investigación de los demás coautores.

Entre las contribuciones más importantes que se deben destacar, se encuentran las siguientes:

- Se ha considerado el problema de analizar un programa compuesto por varios módulos utilizando un analizador dependiente del contexto y polivariante, y se han identificado los principales aspectos relacionados con el análisis modular separado. Este marco de trabajo general ha sido publicado, conjuntamente con Manuel Hermenegildo, Francisco Bueno, María García de la Banda, Kim Marriott y Peter Stuckey, en *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development* (Springer-Verlag, Lecture Notes in Computer Science) [PCH$^+$04].

[‡]Por ejemplo, el entorno de ejecución Java 2 Platform Micro Edition (accesible en la dirección `http://java.sun.com/products`).

- Se ha implementado un algoritmo de punto fijo paramétrico para el análisis incremental de programas modulares, y se ha evaluado experimentalmente para distintos valores de los parámetros. Además, se han examinado y evaluado diversos escenarios incrementales característicos utilizando este algoritmo. Esta evaluación experimental ha sido presentada en el *15th International Symposium on Logic-based Program Synthesis and Transformation* (LOPSTR'05) [CPHB06], como un trabajo conjunto con Manuel Hermenegildo y Francisco Bueno.

- Se han desarrollado dos técnicas específicas para permitir el análisis de tipos de programas modulares utilizando el marco de trabajo desarrollado para esta tesis, y se han evaluado experimentalmente las mejoras producidas por estas técnicas. Este trabajo ha sido aceptado para su presentación en *ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation* (PEPM'08) [PCPH08], como un trabajo conjunto con Paweł Pietrzak y Manuel Hermenegildo.

- Se ha estudiado el problema de verificar que un programa dividido en módulos satisface una especificación parcial expresada mediante aserciones. Se han propuesto diversos algoritmos para implementar la verificación de programas modulares, dependiendo de la calidad de la información proporcionada sobre la especificación del programa y de la eficiencia del algoritmo. Este trabajo fue desarrollado conjuntamente con Paweł Pietrzak y Manuel Hermenegildo, y presentado en el *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning* (LPAR'06) [PCPH06].

- Se han estudiado en detalle dos casos de estudio de especialización de programas. En el primero de ellos, se incide en la especialización de un programa que accede a un sistema externo de bases de datos para generar sentencias SQL optimizadas en tiempo de compilación. En el segundo, las librerías del sistema Ciao son especializadas con respecto a un programa dado con el objetivo de reducir el tamaño del conjunto final de módulos de librería necesarios para la ejecución del programa. El primer caso de estudio ha sido presentado en diversos foros internacionales, conjuntamente con José Manuel Gómez, Manuel Carro, Daniel Cabeza y Manuel Hermenegildo. Este caso de estudio fue presentado inicialmente en el taller de trabajo *Second CoLogNet*

Workshop on Implementation Technology for Computational Logic Systems (*Formal Methods '03 Workshop*), (ITCLS'03) [CGC+03b]. Posteriormente, fue presentada una versión mejorada como contribución de congreso en el *6th International Symposium on Practical Aspects of Declarative Languages*, (PADL'04) [CGC+04a]. Asímismo, un resumen de este trabajo ha sido presentado en forma de poster en 19th Internacional Conference on Logic Programming (ICLP'04) [CGC+03a].

- Como resultado de las contribuciones mencionadas, se ha desarrollado una herramienta, implementada e integrada en el sistema Ciao/CiaoPP, y contribuyendo al sistema global con más de 4000 líneas de código (se puede encontrar más información sobre el sistema en [BLGPH06b, BLGPH06a])

- Por último, esta tesis demuestra la factibilidad de la utilización de técnicas de compilación avanzadas en programas reales.

UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

Analysis and Verification of Modular Programs

PhD Thesis

Jesús Correas Fernández
June 2008

PhD Thesis

Analysis and Verification of Modular Programs

presented at the Computer Science School
of the Technical University of Madrid
in partial fulfillment of the degree of
Doctor in Computer Science

PhD Candidate: Jesús Correas

Licenciado en Informática
Universidad Politécnica de Madrid

Advisor: Germán Puebla

Profesor Titular de Universidad

Madrid, June 2008

To Beatriz

Acknowledgements

First of all, I want to express my gratitude to my thesis advisor, Germán Puebla, and to the director of the CLIP Research Group, Manuel Hermenegildo, for all their time, energy and resources that have allowed me to finish this thesis. In addition to their availability at any time, they have encouraged me to continue with this work in the hardest moments and when I needed it most. It has been a great honor to be part of this research group, and I hope I have achieved the required scientific level for participating in the activities developed by the group. I also want to thank Paweł Pietrzak for his collaboration and friendship, that have resulted in a fruitful series of common works that are a fundamental part of this thesis, and that I hope we can continue in other future projects.

Together with them, the rest of co-authors of the papers that are part of this thesis have contributed to my scientific education: José Manuel Gómez, Manuel Carro, Francisco Bueno, Daniel Cabeza, María García de la Banda, Peter Stuckey, and Kim Marriott.

I have to mention also the members of the CLIP group, with whom I have shared good times. I do not have enough space here to refer to all of them without missing any one, but I want to thank especially to Astrid Beascoa, who was so many times the key for finishing the work on time, and for sharing with her the daily difficulties.

I also want to seize the opportunity to thank the professors that invited me to stay in their respective universities their support and welcome: John Gallagher at Roskilde University, and Deepak Kapur at the University of New Mexico.

Finally, I want to thank my parents who gave me what they did not have to make me reach this point. And nothing of this could have been possible without the unconditional support and patience of Beatriz, to whom I dedicate this thesis.

Abstract

There are many advanced techniques for static program verification and optimization which have been demonstrated extremely useful in detecting bugs and improving the efficiency and which have as common factor the need for precise global analysis information. Abstract interpretation is one of the most established analysis techniques, which has allowed the development of innovative methods for program verification.

The ability to automatically detect bugs in programs and to make sure that a program is correct with respect to a given specification is one of the most important challenges in computer science, in order to produce reliable software. Program verification is an important area of research, and providing advanced techniques for detecting errors and verifying systems in complex, real-life programs is among the most relevant areas in today's computer industry. An interesting approach to program verification is abstract verification, a technique that aims at the verification of a program by means of over-approximations to the concrete semantics of the program.

Nevertheless, most of these methods are not directly applicable to modular, real-life programs, since advanced techniques like the ones aforementioned are in many cases in a prototypical or proof-of-concept state and the advances made so far in this direction have allowed their application in a restricted way only.

The purpose of this PhD Thesis is to enable and show the feasibility of precise analysis and verification techniques to be used in large, modularized and incomplete programs in an accurate and efficient way. In order to assess the practical usefulness of the proposed techniques, the resulting algorithms have been implemented and integrated into the `Ciao` system and experimentally evaluated, that has allowed us to apply them to real-life case studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are many advanced techniques for automatic program verification and optimization which have been demonstrated extremely useful in detecting bugs and improving the efficiency. These techniques have as common factor the need for precise global analysis information. Abstract interpretation is one of the most established techniques to infer information about the execution of the program without actually executing it [CC77a, CC92]. Since the general framework of abstract interpretation appeared, a great number of techniques and tools based on it have been developed, which obtain precise information about the behavior of the program.

The ability to automatically detect bugs in programs and to make sure that a program is correct with respect to a given specification is one of the most important challenges in computer science. Program verification is an important area of research, and providing advanced techniques for detecting errors and verifying systems in complex, real-life programs is among the most relevant areas in today's computer industry. An interesting approach to program verification is abstract verification, a technique that aims at the verification of a program by means of over-approximations to the concrete semantics of the program [HPBLG05, HPBLG03a]. Abstract verification takes advantage of the information obtained by abstract interpretation-based analyses, and uses it to verify the correctness of the program. Among the advantages of abstract verification, there are some of them of special relevance. First, it allows an automatic verification, with no need of user interaction during the verification process. Second, it is possible to provide a partial specification of the program, instead of a complete specification required by other approaches.

Unfortunately, advanced techniques like the ones aforementioned are in many cases in a prototypical or proof-of-concept state. Therefore, they often address small programs without the complexity of real-life systems, including: interfaces to external systems, databases, third-party libraries, components implemented in other languages, and using all the features provided by modularity. This is in contrast to the fact that, in some cases, the existing approaches to the analysis and verification of complex, modular programs either impose strong requirements to the abstract domain for which the program is analyzed, or demand giving up any accuracy achievable in the case of monolithic, small programs. In the particular case of program verification, current systems often require that all the code composing the program and a complete specification of the program are available to the verifier. This requirement is not realistic for large and complex programs, and in particular those legacy systems developed some time ago, and for which there is often not even a complete, up-to-date human-readable user documentation available. The best solution in those cases is to allow a partial specification of the program, and to use advanced analysis techniques to infer what is missing from the specification. This results, of course, in weaker verification, since those aspects about the program not specified are assumed to be satisfied by the code. However, the inferred information can be used to verify the correctness of the program with respect to that partial specification.

The purpose of this thesis is to enable and show the feasibility of precise analysis and verification techniques to be used in large, modularized and incomplete programs. Two frameworks for the analysis and verification of modular programs have been fully implemented in the scope of this thesis, and have been evaluated with several experiments that measure up the different configurations of the framework parameters. In addition, some experiments have also been made addressing the interesting case of incremental reanalysis of modular programs after certain modifications in some of the modules in the program.

This thesis has used `Ciao` [BCC$^+$02] as target programming language. `Ciao` is a multi-paradigm programming system, allowing programming in logic, constraint, and functional styles (as well as a particular form of object-oriented programming). At the heart of Ciao is an efficient logic programming-based kernel language. The (constraint) logic programming area has been affected by the lack (until 2000) of a standard for describing modularity, and that led to the coexistence of several approaches to modularity for different variants of the language. The `Ciao` language includes specific language constructs to represent strict modularity, and share the same guidelines described in the ISO standard. In contrast, the state of the art in the analysis of logic programming languages is pretty advanced. This allows the use of the very large body of approximation domains, inference techniques, and tools for abstract interpretation-based semantic analysis which have been developed to a powerful and mature level in this area. In addition, we take as starting point `CiaoPP` [HPBLG05], a preprocessor containing a large set of advanced tools such as analyzers, parallelizers, specializers, etc.

The `CiaoPP` framework uses incremental abstract interpretation to obtain information about the program, which is then used to verify the program, to detect bugs with respect to partial specifications written using assertions (in the program itself and/or in system libraries), to generate run-time tests for properties which cannot be checked completely at compile-time and simplify them, and to perform high-level program transformations such as multiple abstract specialization, partial evaluation, parallelization, and resource usage control, all in a provably correct way.

Specifically, `CiaoPP` includes a context-sensitive, multivariant analyzer based on abstract interpretation which is parametric with respect to the abstract domains used. In this thesis some of the tools available in `CiaoPP` have been extended in order to accurately and efficiently deal with modular and incomplete programs. The frameworks developed in the scope of this thesis have been fully integrated in `CiaoPP` and are now part of the system. Although this thesis has focused in the `Ciao` language, the techniques developed are also applicable to other programming paradigms.

1.1 Thesis objectives

The main objective of this thesis is the development, implementation and experimental evaluation of novel techniques for accurately and efficiently dealing with modular, real-life (constraint) logic programs.

First, this thesis aims at the development of analysis tools based on the well known technique of abstract interpretation for inferring precise run-time information of modular programs at compile-time.

Based on the previous objective, the second objective is to use the information obtained from the inter-modular analysis to develop new algorithms and tools to verify programs structured in modules, and to highlight the advantages and drawbacks of this approach.

1.2 Structure of the Work

This thesis is composed of three main parts, plus an additional introductory part for providing background concepts on which the rest of the thesis will be built. The central parts of the thesis focus on analysis and abstract verification of modular programs, respectively. Finally, the fourth part describes several applications of the techniques developed to real programs. The following subsections describe them in more detail.

1.2.1 Part I. Background

The first part of this thesis states the basis that will be used in subsequent parts. It is composed of three chapters. In the first chapter of this part, the concept of module to be used throughout the thesis is addressed. Several approaches to modular logic programming have been developed over the years, with different characteristics and semantic backgrounds, and only relatively recently an official standard has been defined [PRO00]. The approach used in this thesis is close to the ones most widely used in current commercial and non-commercial systems, and follows the guidelines described in the standard. Furthermore, due to its extra logical nature, this approach can be easily extended to other programming paradigms different from (constraint) logic programming.

In addition to the advantages offered by modularity to the programmer, it has been used for improving compilation performance by means of separate compilation. Nonetheless, modularity not only helps in separate compilation. It also contributes with additional features. The most relevant one is that it may provide modularity to the analysis as well. This means that, if the language is extended by using library modules (as it is the case in the `Ciao` system used for this thesis) and the module system is strict, some specific features of the language which are specially difficult for a source code handling tool need only be taken into account when analyzing modules that import the library modules defining such features. Among such especially difficult features we can mention higher order procedures, database handling and dynamic procedures, constraint programming features, to name a few. This approach allows the user to set up the most appropriate tools for each module. For example, the analysis approach to be used can be different for different modules: in some modules a bottom-up analysis algorithm is more appropriate, whereas in other modules a top-down algorithm provides better results. In the same way, we can choose an accurate but expensive abstract domain for analysing a given module accurately, and then choose a cheap, though less accurate, abstract domain for other modules in which a very high precision is not required. Similarly, parameters for the abstract domain can also be fine tuned to bring different levels of efficiency and precision in different modules.

The second chapter in this part is an introductory description of abstract interpretation in the context of (constraint) logic programming. As already mentioned, abstract interpretation [CC77a, CC92] is one of the most established techniques to infer provably correct information about the execution of the program without actually executing it. It has several advantages with respect to designing test cases for the analysis of the execution of a program, since it allows capturing the behavior of the program in all possible executions and contexts, instead of the particular cases of the run-time tests designed by a developer. In addition, it does not need to actually execute the program, which in some cases is not possible. When abstractly interpreting a program, an abstraction of the data is used, instead of the data themselves, and the program is therefore interpreted with respect to that abstract domain.

Some optional features of abstract interpretation have been applied in the context of logic programming in order to produce more accurate analysis information. First, *context sensitivity* allows having analysis information about a procedure that depends on the context for which that procedure is called, instead of obtaining more general (but less precise) information about the procedure regardless the context in which it is called. Second, *polyvariance* allows having separate analysis results for each calling context, instead of joining together the calling contexts of a procedure to obtain a single, but less accurate, result of the analysis of the procedure. These two techniques applied together often result in important accuracy gains in the analysis of logic programs.

The third chapter of this part introduces `CiaoPP`, the `Ciao` system preprocessor, and briefly describes in a tutorial way the most relevant components of the preprocessor, introducing some concepts of the `Ciao` assertion language and its applications. The goal of this chapter is to provide the reader with a quick introduction to `CiaoPP`, where the techniques developed in this thesis have been integrated.

1.2.2 Part II. Analysis of Modular Programs

It is sometimes the case that loading the entire program into the analyzer and performing the analysis of all the program monolithically is not possible. In some cases, the cost of the analysis in terms of memory usage is expensive enough to make the analyzer run out of memory if all the program code is loaded into memory and analyzed at the same time. In other cases, it is not possible to load all the code because parts of it are under development, and therefore not completely implemented yet, or they are implemented in other programming languages, or even the code may be developed by a third party, and only compiled code may be available. Also, even if it is possible to analyze the whole program at once, after some modification of the code, it is usually faster to incrementally reanalyze the modules involved in the modification, instead of running an expensive analysis from scratch. In this part of the thesis all these aspects are dealt with, proposing a general, parametric framework, that provides a unified view of those issues.

On the other hand, in the previous part of the thesis two main additional techniques have been described that allow improving the results of abstract interpretation in (constraint) logic programs: both context-sensitivity and polyvariance allow having precise information on calls and successes of program procedures. However, they require the development of an intermodular fixed-point algorithm to obtain the maximum precision in the analysis results. The way in which the intermodular fixed-point is performed is relevant for both the precision and efficiency of the analysis. In this part of the thesis several parameters of the intermodular analysis are highlighted and empirically evaluated, and a parametric intermodular fixed-point algorithm proposed, instantiating it for several scheduling policies. In addition, incremental intermodular analysis is also experimentally assessed for some predefined, characteristic changes in the code, showing the advantages of using an intermodular approach, instead of analyzing the program from scratch again.

As a final chapter of this part, two techniques are described to improve the efficiency of the analysis for type domains. This is important since the most widely used domains for descriptive types turn out not to be directly appropriate for the analysis of modular programs due to efficiency issues. In this case, new types are inferred during the analysis. In order to speed up analysis, the accuracy of the inferred types can be optionally reduced by using only the types predefined by the user or present in the libraries. In every iteration, analysis replaces the inferred types by the most precise approximation using defined types. We show that in this way we ensure faster convergence to the fixpoint and that analysis times are reduced significantly. The second technique is based on the use of parametric type assertions in the specification. Such assertions are especially useful in the module interface of libraries implementing generic data manipulation predicates (like, e.g., lists or binary trees) in order not to reanalyze them for different calling patterns every time we analyze a program that uses the library. In this case, parameters can be instantiated in the trusted assertion (now playing the role of module interface) according to the actual call pattern, and then simply reuse the resulting success pattern without analyzing the library module. By applying the two techniques proposed, the existing analysis domains can be used in a modular context by sharing some specific characteristics of strongly typed logic programming languages [MO84, HL94b], but without changing the source language and while remaining in descriptive types, i.e., types which describe approximations of the untyped semantics. The techniques developed have also been implemented and evaluated experimentally, checking their applicability to real programs, showing the important improvement obtained with this techniques. Incremental reanalysis in the type domain has been assessed as well.

1.2.3 Part III. Compile-time Assertion Checking of Modular Programs

Many existing debugging and verification tools check programs with respect to a specification provided in the form of assertions [PBH00b, DNTM89] written by the user. In some cases, the specification of the program must be complete, whereas in other tools a partial specification may be provided to the verifier. An interesting approach to program verification is abstract verification, a technique that aims at the verification of a program by means of over-approximations to the concrete semantics of the program. Using the exact semantics for verification is in general not realistic, since it can be infinite, too expensive, or only partially known. The abstract verification approach has the advantage of computing *safe* approximations of the program semantics.

In the scope of abstract verification, a framework for static (i.e., compile-time) checking of assertions in modular logic programs is proposed in this part of the thesis. In our framework, the specification of the program through assertions may be partial: those assertions not directly written by the user are inferred by the global analysis.

When considering a program split in several modules, the correctness of a module can be seen as conditional, depending on the correctness of the rest of the modules in the program. Using this approach, different concepts for correctness of assertions in modular programs are studied. As a consequence, this work has allowed us to develop several approaches to intermodular verification, which affect how the analysis information of the program is obtained. For example, in the case that there are enough assertions specified by the user, computing an expensive intermodular fixed-point may be avoided in the case that it is possible to perform a single traversal of the modules in the program during the analysis phase. Another alternative, if there are not enough user assertions, is to interleave intermodular analysis and checking in order to obtain as soon as possible (in)correctness information for the partial specification of the program. Several algorithms are provided in this thesis, and the relation between the analysis parameters and correctness conditions are studied in depth.

1.2.4 Part IV. Applications

This final part of the thesis consists of two applications of program analysis to the handling of real-life programs. In the first case, an application of modular program analysis is presented in the context of programs with interfaces to external systems for which the source code is not available. Among them, is particularly important the use of relational database systems: its own structure, from the point of view of the analysis, brings a great deal of information (types of the arguments, termination, determinism, modes and instantiation level of arguments, etc.). This technique is therefore applied to programs which access to external database systems, in order to improve accesses to the database with compile-time information obtained by means of abstract interpretation. This case study is a clear example about how advanced techniques can be used for real-life programs.

In the second case, a novel modular specialization algorithm is developed in order to reduce the size of code in libraries for a given program. This algorithm is based in the non-modular multiple specialization algorihm of [PH03], and propagates specialized versions of predicates across the modular graph. In this case, it has been applied to strip-down the code of the libraries used by a program, in order to reduce the total size of the object code and make them fit in a small device.

1.3 Contributions

This thesis has contributed to the state of the art in several ways, detailed as follows:

- The problem of analyzing a program split in modules using a context-sensitive, polyvariant analyzer has been addressed, and the main issues related to the separate modular analysis approach have been identified. The general analysis framework, co-authored with Manuel Hermenegildo, Francisco Bueno, María García de la Banda, Kim Marriott, and Peter Stuckey, has been published in *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development* (Springer-Verlag, Lecture Notes in Computer Science) [PCH$^+$04].

- A parametric fixed point algorithm for the incremental analysis of a modular program has been implemented and experimentally evaluated for different parameters of the algorithm. In addition, several characteristic incremental scenarios have been examined and assessed using that algorithm. This experimental evaluation has been presented at *15th International Symposium on Logic-based Program Synthesis and Transformation* (LOPSTR'05) [CPHB06]. It was co-authored with Manuel Hermenegildo and Francisco Bueno.

- Two techniques for enabling type analysis for modular programs have been developed, and their impact experimentally evaluated. This work has been co-authored with Paweł Pietrzak and Manuel Hermenegildo, and is to be presented at the *ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation* (PEPM'08) [PCPH08].

- The issue of verifying that a program split in modules satisfies a (possibly partial) specification given as assertions has been studied. Several algorithms for implementing modular program verification have been proposed, depending on the quality of the information provided about the specification of the program. This work was co-authored with Paweł Pietrzak and Manuel Hermenegildo, and presented at *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning* (LPAR'06) [PCPH06].

- Two case studies have been studied in detail. In the first case study, special emphasis is made in programs that access external database systems and in the generation or efficient SQL statements at compilation time. This case study, co-authored with José Manuel Gómez, Manuel Carro, Daniel Cabeza, and Manuel Hermenegildo, has been presented at *6th International Symposium on Practical Aspects of Declarative Languages*, (PADL'04) [CGC$^+$04a]. An abstract of this work, co-authored with the same authors, was presented as a poster at the *19th Internacional Conference on Logic Programming* [CGC$^+$03a].

- In the second case study, Ciao system libraries are specialized with respect to a given program in order to reduce the size of the runtime system needed for the execution of the program. This has been applied in the context of the EU IST Programme project *Advanced Specialization and Analysis for Pervasive Systems* (ASAP, IST-2001-38059).

- The contributions listed above have been implemented and integrated into the Ciao/CiaoPP system, contributing to the overall system with more than 4,000 lines of code (more information about the system can be found in [BLGPH06b, BLGPH06a])

- Finally, this thesis demonstrates the feasibility of using multivariant, context-sensitive analysis and verification for real-life programs.

Part I

Background

Chapter 2

Module Systems for Logic Programs

Modularity is a programming concept that has received much attention in the context of logic programming over the years. Diverse approaches, with different features and limitations have been proposed. Nevertheless, the logic programming community has not agreed in a common standard for describing modular programs until relatively recently [PRO00]. This has led to the coexistence of several concepts of modularity in different logic programming systems. An early survey of the most important approaches to modular logic programming can be found in [BLM94]. There, the authors identify two main approaches to modularity. The first approach refers to modular programming in-the-small: in this approach, basic constructs in the modular language are integrated in the underlying logical foundations, and in most cases they are extended. That enables the main features of modularity (encapsulation and code reuse), together with other characteristics provided by the extended frameword. The second approach, modular programming in-the-large, is focused in how to structure the code of complex programs. In this case, the language constructs are extra-logical, and provide the facilities of splitting the code in modules, although do not extend the logical foundations. The ISO Prolog standard fits in this second approach.

In our context, a program is said to be modular when its source code is distributed in several source units named modules, and they contain language constructions to clearly define the *interface* of every module with the rest of the modules in the program. This interface is composed of two sets of predicates:

the set of *exported* predicates (those accessible from other modules), and the set of *imported* modules. The module system we consider is *strict* in the sense that procedures defined in a module m are visible in another module m' which uses m only if such procedures are exported by m, i.e., procedures which are not exported are not visible outside the module in which they are defined. Non strict module systems can be used for separate compilation, but they are not useful to separate context-sensitive analysis: there is no way to know at compile time which procedures will be called from outside, nor the corresponding call patterns. As a result, all procedures must be analyzed for the most general call pattern, which defeats the spirit of context sensitivity.

We now introduce some notation. We will use m and m' to denote *modules*. Given a module m, by $imports(m)$ we denote the set of modules which m imports. Figure 2.1 presents a modular program. Modules are represented as boxes and there is an arrow from m to m' iff m imports m'. In our example, $imports(a) = \{b, c\}$. By $depends(m)$ we refer to the set generated by the transitive closure of $imports$, i.e. $depends(m)$ is the least set such that $imports(m) \subseteq depends(m)$ and $m' \in depends(m)$ implies that $imports(m') \subseteq depends(m)$. In our example, $depends(a) = \{b, c, d, e, f\}$. Note that there may be circular dependencies among modules. In our example, $e \in depends(d)$ and $d \in depends(e)$. A module m is a *leaf* if $depends(m) = \emptyset$. In our example, the only leaf module is f. By $callers(m)$ we denote the set of modules which import m. In the example, $callers(e) = \{b, c, d\}$. Also, we define $related(m) = callers(m) \cup imports(m)$. In our example, $related(b) = \{a, d, e\}$.

The *program unit* of a given module m is the finite set of modules containing m and the modules on which m depends: $program\_unit(m) = \{m\} \cup depends(m)$[*]. The module m is called the *top-level* module of its program unit. In our example, $program\_unit(a) = \{a, b, c, d, e, f\}$ and $program\_unit(c) = \{c, d, e, f\}$. A program unit U is self-contained in the sense that $\forall\ m \in U\ : m' \in imports(m) \rightarrow m' \in U$.

Finally, $exported\_preds(m)$ is the set of predicate names exported by module m, and $imported\_preds(m)$ is the set of predicate names imported by m from any module ni $imports(m)$. Given a program unit $program\_unit(m)$, we can always obtain a single-module program that behaves like $program\_unit(m)$ [BLM94]: we

[*]As discussed later, library modules and *builtins* require special treatment in order to avoid reanalysis of all used library predicates every time a user program is analyzed.

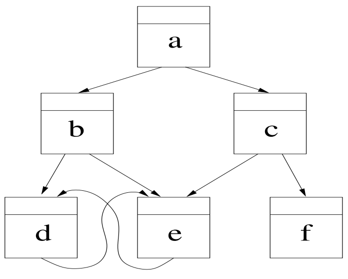


Figure 2.1: An Example of Module Dependencies

will denote such program as $flatten(m)$, defined as

$$flatten(m) = \bigcup_{n \in program\_unit(m)} ren(n)$$

where $ren(n)$ stands for a suitable predicate renaming so that no conflicts arise between predicates with the same name in different modules, and $exported\_preds(flatten(m)) = exported\_preds(m)$.

The frameworks included in this thesis assume that the module system used is *static*, in the sense that module-related language constructs like module importation and exported procedures specification must be described in the program by means of static declarations, instead of specifying it as a result of the execution of some program code. This aspect is particularly important for modular preprocessing tools, since otherwise the modules of the program cannot be processed separately at compile time, because the list of modules to be used by the program would be unknown at compile time.

In particular, there are two aspects of the modular system that are required to be static:

- On one hand, the modular structure of the program must be known in advance. That means that $imports(m)$, for all $m \in program\_unit(m_{top})$, cannot change during the execution of the program. The preprocessing tools must know which are the modules and the relations among them in order to build the graph of inter-modular relations and to be able to decide how the information obtained for a given module affects the rest of the program.

- On the other hand, the list of exported predicates $exported\_preds(m)$ must be statically fixed. One of the most important advantages of modularity from the point of view of preprocessing tools (like analyzers or verifiers) is that the interface of the module is known in advance, and therefore it is guaranteed that the only calls to a non-exported predicate are the ones inside the module defining it.

2.1 The `Ciao` Module System

For concreteness, and because of its appropriateness for global analysis, in our implementation we will use the module system of `Ciao` [CH00]. The `Ciao` language implements an extra-logical approach to modularity, and to a great extent it complies with the guidelines described in the ISO standard.

The `Ciao` language has been designed to facilitate the work of global preprocessing tools, but at the same time to implement advanced language features (as e.g. higher-order programming or dynamic modular constructs). That means that some features of the language that prevent from using the frameworks developed in this thesis are actually available in the `Ciao` system. Nevertheless, the modular structure itself is of great help in this issue: such features are only activated if a specific module (the one defining the compiler itself) is loaded from a module in the user program. Therefore, it is easy to identify which modules use dynamic modular constructs. [Cab04] describes extensively the `Ciao` module system and its design guidelines.

The basic language declarations available in `Ciao` for expressing modularity are the following:

`:- module(M,Exp,Pkg).` This declaration is the main module declaration. It must appear before any code that belongs to this module. `M` is the module name (which usually must match with the file name that contains the source code), `Exp` is the list of exported procedures of the module. It represents the public interface of the module. Finally, `Pkg` is optional, and indicates whether the module uses `Ciao` syntax extensions, such as DCGs, functional syntax, or assertions for providing information about the program.

`:- export(Exp).` This is another way to declare a predicate or a list of predicates as exported. It is equivalent to adding `Exp` to the second argument of the module declaration.

`:- use_module(IM,Proc).` The set of imported modules is declared in `Ciao` using this construct. A `use_module` declaration must be used for each imported module `IM`. `Proc` is an optional argument that indicates the predicates to be used from that imported module. The predicates in `Proc` must have been exported by `IM`.

Example 2.1.1. *Figure 2.2 shows an example of a modular program. It is composed of three modules, namely* `test`*,* `qsort`*, and* `lists`*.*

The top-level module of this program is the module `test`*, that sorts a list given as input, and then checks that the resulting list is indeed sorted and has the same length as the input list. The module declaration states that the name of the module is* `test`*, exports a procedure,* `test/1`*, and uses the package* `assertions`*, that extends the language allowing assertions for describing procedures. It is important to note that module* `test` *defines two procedures,* `test/1` *and* `sorted/1`*, although only* `test/1` *is exported.* `sorted/1` *is internal to the module, and it is therefore guaranteed by the strictness property of the module system that there will be no calls from anywhere in any program using this module apart from those inside module* `test`*. Module* `test` *also imports the other two modules,* `qsort` *and* `lists` *(lines 2 and 3). Finally, in line 4 there is an example of assertion, stating that the exported predicate will be called using a list of numbers as argument. Chapter 4 includes an overall description of the assertion language.*

Module `qsort` *contains two procedures, of which only one procedure is exported,* `qsort/2`*. In addition, it uses module* `lists` *(line 2), but only imports* `append/3`*.*

Finally, module `lists` *is an excerpt of the* `Ciao` *library module with the same name. It contains four predicates, of which only* `append/3` *and* `length/2` *are exported.*

```
:- module(test,[test/1],[assertions]).
:- use_module(qsort).
:- use_module(lists).
:- entry test(X) : list(X,num).

test(L) :-  length(L,Length), qsort(L,Res), sorted(Res), length(Res,Length).

sorted([]).
sorted([_]).
sorted([X,Y|Z]) :-  X =< Y, sorted([Y|Z]).
%%----------------------------------------------------
:- module(qsort, [qsort/2],[]).
:- use_module(lists, [append/3]).

qsort([X|L],R) :-
        partition(L,X,L1,L2), qsort(L2,R2), qsort(L1,R1), append(R1,[X|R2],R).
qsort([],[]).

partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right) :- E < C, !,partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]) :- E >= C, partition(R,C,Left,Right1).
%%----------------------------------------------------
:- module(lists,[append/3,length/2],[]).

append([], L, L).
append([E|Es], L, [E|R]) :- append(Es, L, R).

length(L, N) :- var(N), !, llength(L, 0, N).
length(L, N) :- dlength(L, 0, N).

llength([], I, I).
llength([_|L], I0, I) :- I1 is I0+1, llength(L, I1, I).

dlength([], I, I) :- !.
dlength([_|L], I0, I) :- I0<I, I1 is I0+1, dlength(L, I1, I).
```

Figure 2.2: A modular program.

Chapter 3

Abstract Interpretation of Logic Programs and its Applications

3.1 Semantics of Logic Programs and Abstract Interpretation

We recall some classical definitions in logic programming. An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. A *predicate descriptor* is an atom $p(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are distinct variables. We shall use predicate descriptors to refer to a certain form of atoms, as well as to predicate symbols. A *clause* is of the form $H\texttt{:-}B_1, \ldots, B_n$ where H, the *head*, is an atom and $B_1, \ldots, B_n$, the *body*, is a possibly empty finite conjunction of atoms. In the following we assume that all clause heads are normalized, i.e., H is of the form of a predicate descriptor. Furthermore, we require that each clause defining a predicate p has an identical sequence of variables $X_{p_1}, ..., X_{p_n}$ in the head. We call this the *base form* of p. This is not restrictive since programs can always be normalized, and it will facilitate the presentation of the algorithms later. However, both in the examples and in the implementation we handle non-normalized programs. A *definite logic program*, or *program*, is a finite sequence of clauses. *ren* denotes a set of renaming substitutions over variables in the program at hand.

We will consider two kinds of concrete semantics for this thesis. On one hand we can use the important class of semantics referred to as *fixpoint semantics.*

In this setting, a (monotonic) semantic operator (which we refer to as S_R) is associated with each program R. This S_R function operates on a semantic domain D which is generally assumed to be a complete lattice or, more generally, a chain complete partial order. The meaning of the program (which we refer to as $[\![R]\!]$) is defined as the least fixpoint of the S_R operator, i.e., $[\![R]\!] = \mathrm{lfp}(S_R)$. A well-known result is that if S_R is continuous, the least fixpoint is the limit of an iterative process involving at most ω applications of S_R and starting from the bottom element of the lattice.

On the other hand, another concrete semantics used for reasoning about programs will use the notion of generalized AND trees, as they are described in [Bru91]. We will use a standard notion of substitutions and use notation $\theta_{|V}$ to denote a projection of the substitution θ over the set variables V. Every node of a generalized AND tree, denoted $\langle \theta_c, (A, H), \theta_s \rangle$, contains a call to a predicate A and a head H of the matching clause (not present if A is a built-in), with a call substitution θ_c and corresponding success subtitution θ_s, defined over variables of the clause where A occurs. Sometimes one or both substitutions, or the clause head of some nodes might be not known, or simply not applicable. In such cases we will write them as $-$.

Definition 3.1.1. *A generalized* AND *tree is defined as follows:*

Initialization *An initial call A with call substitution θ_c forms a single-node generalized* AND *tree $\langle \theta_c, (A, -), - \rangle$.*

Procedure entry *Given a generalized* AND *tree with a leaf node $\langle \theta_c, (A, -), - \rangle$ corresponding to a call, and a renamed apart clause $C = H \leftarrow B_1, \ldots, B_n$ defining A, the node becomes $\langle \theta_c, (A, H), - \rangle$, and $B_1, \ldots, B_n$ are added as its children, with B_1 being adorned to the left by the call substitution $\psi = (\theta \circ \theta_c)|_{vars(C)}$, where $\theta = mgu(A, H) \neq fail$. If $n = 0$ then ψ is a success substitution for C.*

Interpretation of a built-in *Given a generalized* AND *tree with a leaf node $\langle \theta_c, A, - \rangle$ corresponding to a call to a built-in, the new tree is obtained by transforming the node to $\langle \theta_c, A, \theta_s \rangle$ where θ_s is a success substitution for A. If A is the last literal in the clause body then θ_s becomes a success substitution of the clauses, otherwise it is a call substitution for the next call.*

Procedure exit *Given a generalized* AND *tree with a node* $\langle\theta_c, (A, H), -\rangle$ *being a parent of a clause body with a success substitution* ψ, *the node becomes* $\langle\theta_c, (A, H), \theta_s\rangle$, *where* $\theta_s = (\theta \circ \psi)|_{vars(C)}$, *where* $\theta = mgu(A, H)$, *and* C *is a clause containing a call* A.

The concrete semantics of a program R for a given set of queries Q, $[\![R]\!]_Q$, is the set of generalized AND trees that represent the execution of the queries in Q for the program R*. When the set of queries is not relevant, we will refer to the concrete semantics simply as $[\![R]\!]$.

Definition 3.1.2. *calling_context*(P, R, Q) *of a predicate given by the predicate descriptor* P *defined in* R *for a set of queries* Q *is the set* $\{\theta_c | \exists T \in [\![R]\!]_Q$ *s.t.* $\exists\langle\theta'_c, (A, -), -\rangle$ *in* $T \wedge \exists\, \psi$ *s.t.* $P\psi = A \wedge \theta_c = \psi \circ \theta'_c\}$

success_context(P, R, Q) *of a predicate given by the predicate descriptor* P *defined in* R *for a set of queries* Q *is the set of pairs* $\{(\theta_c, \theta_s) | \exists T \in [\![R]\!]_Q$ *s.t.* $\exists\langle\theta'_c, (A, H), \theta'_s\rangle$ *in* $T \wedge \exists\, \psi$ *s.t.* $P\psi = A \wedge \theta_c = \psi \circ \theta'_c \wedge \theta_s = \psi \circ \theta'_s\}$

Abstract interpretation [CC77a] is a technique for static program analysis in which execution of the program is simulated on a description (or abstract) domain (D_α) which is simpler than the actual (or concrete) domain (D). Values in the description domain and sets of values in the actual domain are related via a pair of monotonic mappings $\langle\alpha, \gamma\rangle$: *abstraction* $\alpha : 2^D \rightarrow D_\alpha$ and *concretization* $\gamma : D_\alpha \rightarrow 2^D$ which form a Galois connection, i.e.

$$\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall\lambda \in D_\alpha : \ \alpha(\gamma(\lambda)) = \lambda.$$

The set of all possible descriptions represents a description domain D_α which is usually a complete lattice or cpo for which all ascending chains are finite. Note that in general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall\lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$). Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in some precise sense. A description $\lambda \in D_\alpha$ *approximates* a set of concrete values $x \in 2^D$ if $\alpha(x) \sqsubseteq \lambda$. Correctness of abstract interpretation guarantees that the descriptions computed approximate all of the actual values which occur during execution of the program.

*We find this formalization more suitable than the derivation-based one used in a previous work [PBH00d] because it simplifies the presentation of the subsequent material. This formalization will be used for Part III mainly.

One of the fundamental results of abstract interpretation is that an abstract semantic operator S_R^α for a program R can be defined which is correct w.r.t. S_R in the sense that $\gamma(\mathrm{lfp}(S_R^\alpha))$ is an approximation of $[\![R]\!]$, and, if certain conditions hold (e.g., ascending chains are finite in the D_α lattice), then the computation of $\mathrm{lfp}(S_R^\alpha)$ terminates in a finite number of steps. We will denote $\mathrm{lfp}(S_R^\alpha)$, i.e., the result of abstract interpretation for a program R, as $[\![R]\!]_\alpha$.

Typically, abstract interpretation guarantees that $[\![R]\!]_\alpha$ is an *over*-approximation of the abstract semantics of the program itself, $\alpha([\![R]\!])$. Thus, we have that $[\![R]\!]_\alpha \supseteq \alpha([\![R]\!])$, which we will denote as $[\![R]\!]_{\alpha^+}$. Alternatively, the analysis can in principle be designed to safely *under*-approximate the actual semantics [Sch07], and then we have that $[\![R]\!]_\alpha \subseteq \alpha([\![R]\!])$, which we denote as $[\![R]\!]_{\alpha^-}$.

Different description domains may be used which capture different properties with different accuracy and cost. Also, for a given description domain, program, and set of initial call patterns there may be many different analysis graphs. However, for a given set of initial call patterns, a program and abstract operations on the descriptions, there is a unique *least analysis graph* which gives the most precise information possible.

Goal-dependent abstract interpretation takes as input a program R and a call pattern[†] $P{:}\lambda$, where P is an atom, and λ is a restriction of the run-time bindings of P expressed as an abstract substitution in the abstract domain D_α. Such an abstract interpretation (denoted $analysis(R, P{:}\lambda)$) returns OR-nodes of the abstract AND-OR trees in the form of triples $\{\langle P_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle P_n, \lambda_n^c, \lambda_n^s\rangle\}$. Let $P{:}\lambda$ be an abstract initial call pattern, and let Q be the set of concrete queries described by $P{:}\lambda$, i.e. $Q = \{P\theta \mid \theta \in \gamma(\lambda)\}$. Correctness of abstract interpretation guarantees:

- The abstract success substitutions cover all the concrete success substitutions which appear during execution, i.e., $\forall i = 1..n\ \forall \theta_c \in \gamma(\lambda_i^c)$ if $P_i\theta_c$ succeeds in R with computed answer θ_s then $\theta_s \in \gamma(\lambda_i^s)$.

- The abstract call substitutions cover all the concrete calls which appear during execution of initial queries in Q. Formally, $\forall P'$ in R $\forall \theta_c \in$ $calling\_context(P', R, Q)$ $\exists \langle P', \lambda^c, \lambda^s\rangle \in analysis(R, P{:}\lambda)$ s.t. $\theta_c \in \gamma(\lambda^c)$.

[†]Note that we shall use sets of call patterns instead in the subsequent chapters –the extension is trivial.

As usual in abstract interpretation, $\perp$ denotes the abstract substitution such that $\gamma(\perp) = \emptyset$. A tuple $\langle P_j, \lambda_j^c, \perp\rangle$ indicates that all calls to predicate p_j with substitution $\theta \in \gamma(\lambda_j^c)$ either fail or loop, i.e., they do not produce any success substitutions.

An analysis is said to be *multivariant (on calls)* if more than one entry $P{:}\lambda_1^c \mapsto \lambda_1^s, \ldots, P{:}\lambda_n^c \mapsto \lambda_n^s$ $n \geq 0$ with $\lambda_i^c \neq \lambda_j^c$ for some i, j may be computed for the same predicate. As it is shown in this thesis, multivariant analyzers may provide valuable information (for example, for assertion checking, as it will be seen in Part III) not obtainable otherwise. Note that if $n = 0$ then the corresponding predicate is not needed for solving any goal in the considered class (p, λ) and is thus dead code and may be eliminated, as explained in the application shown in Chapter 10. An analysis is said to be *multivariant on successes* if more than one entry $P{:}\lambda^c \mapsto \lambda_1^s, \ldots, P{:}\lambda^c \mapsto \lambda_n^s$ $n \geq 0$ with $\lambda_i^s \neq \lambda_j^s$ for some i, j may be computed for the same predicate p and call substitution λ^c. Different analyses may be defined with different levels of multivariance [VDLM93]. Many implementations of abstract interpreters are multivariant on calls. However, most of them are not multivariant on successes, mainly for efficiency reasons (such as the analyzer in `CiaoPP` [HPMS00] used in this thesis, that allows both types of multivariance, but multivariance on success is switched off by default). An abstract interpretation process is monotonic, in the sense that the more specific the initial call pattern is, the more precise the results of the analysis are. The output of the abstract interpreter is kept in an *answer table* (AT), as it will be detailed in Chapter 5. In the rest of this thesis, entries of AT will be denoted $P{:}\lambda^c \mapsto \lambda^s$.

Example 3.1.3. *The traditional* `append/3` *predicate for concatenating lists, defined as*

```
app([],Y,Y).
app([A|X1],Y,[A|Z1]):- app(X1,Y,Z1).
```

may be called in different ways. If it is called with first and second arguments instantiated to lists, then on success the third argument will be instantiated to a list, the result of concatenating them. On the other hand, if it is called with the third argument instantiated to a list, then on success first and second arguments will be both lists, a possible decomposition of the third argument in two sublists. As instance, the following predicate for inserting an element into a list

```
insert(X,L1,L2):- app(A,B,L1), app(A,[X|B],L2).
```

produces both calling patterns for `append/3` *when it is called with* `L1` *instantiated to a list.*

When analyzing `append/3` *using a monovariant, context-sensitive analyzer with respect to both different calling patterns, it actually analyzes it with respect to their least upper bound. If A, B, and C represent the arguments of* `append/3`*, the calling abstract substitution in a type domain would be:*

$$(list(A) \wedge list(B) \wedge any(C)) \sqcup (any(A) \wedge any(B) \wedge list(C))$$

$$= (any(A) \wedge any(B) \wedge any(C)) = \top$$

The result of analyzing `append/3` *with respect to $\top$ would be*

$$append(A, B, C) : \top \mapsto (list(A) \wedge any(B) \wedge any(C))$$

In contrast, a multivariant, context-sensitive analysis considers both calling patterns separately, obtaining more accurate results:

$append(A, B, C) : (list(A) \wedge list(B) \wedge any(C)) \mapsto (list(A) \wedge list(B) \wedge list(C))$
$append(A, B, C) : (any(A) \wedge any(B) \wedge list(C)) \mapsto (list(A) \wedge list(B) \wedge list(C))$

A partial ordering on answer tables over a given module can be defined in the following sense:

Definition 3.1.4. *Let AT_1 and AT_2 be answer tables over a given module m.*

$AT_1 \preceq AT_2$ iff $\forall(P : CP_1 \mapsto AP_1) \in AT_1, (\exists(P : CP_2 \mapsto AP_2) \in AT_2$ s.t. $CP_1 \sqsubseteq CP_2$ and $\forall(P : CP_2' \mapsto AP_2') \in AT_2$, if $CP_1 \sqsubseteq CP_2'$ then $AP_1 \sqsubseteq AP_2')$.

3.2 Applications of Abstract Interpretation

In this section some applications of abstract interpretation will be described, namely abstract verification and debugging, and program transformation, which will be used in the rest of this thesis. In this section we follow [HPBLG05].

Property	**Definition**
P is partially correct w.r.t. $\mathcal{I}$	$[\![P]\!] \subseteq \mathcal{I}$
P is complete w.r.t. $\mathcal{I}$	$\mathcal{I} \subseteq [\![P]\!]$
P is incorrect w.r.t. $\mathcal{I}$	$[\![P]\!] \not\subseteq \mathcal{I}$
P is incomplete w.r.t. $\mathcal{I}$	$\mathcal{I} \not\subseteq [\![P]\!]$

Table 3.1: Set theoretic formulation of verification problems

3.2.1 Abstract Verification and Debugging

Both program verification and debugging compare the *actual semantics* of the program, i.e., $[\![P]\!]$, with an *intended semantics* for the same program, which we will denote by $\mathcal{I}$. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. In Table 3.1 we define classical verification problems in a set-theoretic formulation as simple relations between $[\![P]\!]$ and $\mathcal{I}$.

Using the exact actual or intended semantics for automatic verification and debugging is in general not realistic, since the exact semantics can be typically only partially known, infinite, too expensive to compute, etc. On the other hand the abstract interpretation technique allows computing *safe* approximations of the program semantics. The key idea in our approach [BDD$^+$97, HPB99, PBH00d] is to use the abstract approximation $[\![P]\!]_\alpha$ directly in program verification and debugging tasks.

A number of approaches have already been proposed which make use to some extent of abstract interpretation in verification and/or debugging tasks. Abstractions were used in the context of algorithmic debugging in [LS88]. Abstract interpretation for debugging of imperative programs has been studied by Bourdoncle [Bou93], by Comini et al. for the particular case of algorithmic debugging of logic programs [CLV95] (making use of partial specifications) [CLMV99], and very recently by P. Cousot [Cou03b].

Our first objective herein is to present the implications of the use of *approximations* of both the intended and actual semantics in the verification and debugging process. As we will see, the possible loss of accuracy due to approximation prevents full verification in general. However, and interestingly, it turns out that in many cases useful verification and debugging conclusions can still be derived by comparing the approximations of the actual semantics of a program to the (also possibly approximated) intended semantics.

In our approach we actually compute the abstract approximation $[\![P]\!]_\alpha$ of the concrete semantics of the program $[\![P]\!]$ and compare it directly to the (also approximate) intention (which is given in terms of *assertions* [PBH00b]), following almost directly the scheme of Table 3.1. This approach can be very attractive in programming systems where the compiler already performs such program analysis in order to use the resulting information to, e.g., optimize the generated code, since in these cases the compiler will compute $[\![P]\!]_\alpha$ anyway. Alternatively, $[\![P]\!]_\alpha$ can always be computed on demand.

For now, we assume that the program specification is given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$. Comparison between actual and intended semantics of the program is most easily done in the same domain, since then the operators on the abstract lattice, that are typically already defined in the analyzer, can be used to perform this comparison. Thus, it is interesting to study the implications of comparing $\mathcal{I}_\alpha$ and $[\![P]\!]_\alpha$, which is an approximation of $\alpha([\![P]\!])$.

In Table 3.2 we propose (sufficient) conditions for correctness and completeness w.r.t. $\mathcal{I}_\alpha$, which can be used when $[\![P]\!]$ is approximated. Several instrumental conclusions can be drawn from these relations.

Analyses which over-approximate the actual semantics (i.e., those denoted as $[\![P]\!]_{\alpha^+}$), are specially suited for proving partial correctness and incompleteness with respect to the abstract specification $\mathcal{I}_\alpha$. It will also be sometimes possible to prove incorrectness in the extreme case in which the semantics inferred for the program is incompatible with the abstract specification, i.e., when $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset$. We also note that it will only be possible to prove total correctness if the abstraction is *precise*, i.e., $[\![P]\!]_\alpha = \alpha([\![P]\!])$. According to Table 3.2 completeness requires $[\![P]\!]_{\alpha^-}$ and partial correctness requires $[\![P]\!]_{\alpha^+}$. Thus, the only possibility is that the abstraction is precise.

On the other hand, we use $[\![P]\!]_{\alpha^-}$ to denote the less frequent case in which analysis under-approximates the actual semantics. In such case, it will be possible to prove completeness and incorrectness. In this case, partial correctness and incompleteness can only be proved if the analysis is precise.

If analysis information allows us to conclude that the program is incorrect

Property	**Definition**	**Sufficient condition**
P is partially correct w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \subseteq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \subseteq [\![P]\!]_{\alpha^-}$
P is incorrect w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \not\subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^-} \not\subseteq \mathcal{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \not\subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \not\subseteq [\![P]\!]_{\alpha^+}$

Table 3.2: Validation problems using approximations

or incomplete w.r.t. $\mathcal{I}_\alpha$, an (abstract) symptom has been found which ensures that the program does not satisfy the requirement. Thus, debugging should be initiated to locate the program construct responsible for the symptom. Since $[\![P]\!]_{\alpha^+}$ often contains information associated to program points, it is often possible to use the this information directly and/or the analysis graph itself to locate the earliest program point where the symptom occurs (see Section 4.2).

It is important to point out that the use of safe approximations is what gives the essential power to the approach. As an example, consider that classical examples of assertions are type declarations. However, herein we are interested in supporting a much more powerful setting in which assertions can be of a much more general nature, stating additionally other properties, some of which cannot always be determined statically for all programs. These properties may include properties defined by means of user programs and extend beyond the predefined set which may be natively understandable by the available static analyzers. Also, only a small number of (even zero) assertions may be present in the program, i.e., the assertions are *optional*. In general, we do not wish to limit the programming language or the language of assertions unnecessarily in order to make the validity of the assertions statically decidable (and, consequently, the proposed framework needs to deal throughout with approximations).

Additional discussions and more details about the foundations and implementation issues of our approach can be found in [BDD$^+$97, HPB99, PBH00d, PBH00a].

Property	**Definition**	**Sufficient condition**
L is abstractly executable to *true* in P	$RT(L,P) \subseteq TS(L,P)$	$\exists \lambda' \in A_{TS}(\overline{B}, D_\alpha) : \lambda_L \sqsubseteq \lambda'$
L is abstractly executable to *false* in P	$RT(L,P) \subseteq FF(L,P)$	$\exists \lambda' \in A_{FF}(\overline{B}, D_\alpha) : \lambda_L \sqsubseteq \lambda'$

Table 3.3: Abstract Executability

3.2.2 Abstract Executability and Program Transformation

In our program development framework, abstract interpretation also plays a fundamental role in the areas of program transformation and program optimization. Optimizations are performed by means of the concept of *abstract executability* [GH91, PH97]. This allows reducing at compile-time certain program fragments to the values *true*, *false*, or *error*, or to a simpler program fragment, by application of the information obtained via abstract interpretation. This allows optimizing and transforming the program (and also detecting errors at compile-time in the case of *error*).

For simplicity, we will limit herein the discussion to reducing a procedure call or program fragment L (for example, a "literal" in the case of logic programming) to either *true* or *false*. Each run-time invocation of the procedure call L will have a *local environment* which stores the particular values of each variable in L for that invocation. We will use θ to denote this environment (composed of assignments of values to variables, i.e., *substitutions*) and the restriction (projection) of the environment θ to the variables of a procedure call L is denoted $\theta|_L$.

We now introduce some definitions. Given a procedure call L without side-effects in a program P we define the *trivial success set* of L in P as $TS(L,P) = \{\theta|_L : L\theta$ succeeds exactly once in P with empty answer substitution $(\epsilon)\}$. Similarly, given a procedure call L from a program P we define the *finite failure set* of L in P as $FF(L,P) = \{\theta|_L : L\theta$ fails finitely in $P\}$.

Finally, given a procedure call L from a program P we define the *run-time substitution set* of L in P, denoted $RT(L,P)$, as the set of all possible substitutions (run-time environments) in the execution state just prior to executing the procedure call L in any possible execution of program P.

Table 3.3 shows the conditions under which a procedure call L is abstractly executable to either *true* or *false.* In spite of the simplicity of the concepts, these definitions are not directly applicable in practice since $RT(L, P)$, $TS(L, P)$, and $FF(L, P)$ are generally not known at compile time. However, it is usual to use a *collecting semantics* as concrete semantics for abstract interpretation so that analysis computes for each procedure call L in the program an abstract substitution λ_L which is a safe approximation of $RT(L, P)$, i.e. $\forall L \in P \;\; RT(L, P) \subseteq \gamma(\lambda_L)$.

Also, under certain conditions we can compute either automatically or by hand sets of abstract values $A_{TS}(\overline{L}, D_\alpha)$ and $A_{FF}(\overline{L}, D_\alpha)$ where $\overline{L}$ stands for the *base form* of L, i.e., where all the arguments of L contain distinct free variables. Intuitively they contain abstract values in domain D_α which guarantee that the execution of $\overline{L}$ trivially succeeds (resp. finitely fails). For soundness it is required that $\forall\lambda \in A_{TS}(\overline{L}, D_\alpha)\ \gamma(\lambda) \subseteq TS(\overline{L}, P)$ and $\forall\lambda \in A_{FF}(\overline{L}, D_\alpha)\ \gamma(\lambda) \subseteq FF(\overline{L}, P)$.

Even though the simple optimizations illustrated above may seem of narrow applicability, in fact for many builtin procedures such as those that check basic types or which inspect the structure of data, even these simple optimizations are indeed very relevant. Two non-trivial examples of this are their application to simplifying independence tests in program parallelization [PH99] (Section 4.3) and the optimization of delay conditions in logic programs with dynamic procedure call scheduling order [PdlBMS97].

These and other more powerful abstract executability rules are embedded in the multivariant abstract interpreter in our program development framework. The resulting system performs essentially all high- and low-level program optimizations and transformations during program development and in compilation. In fact, the combination of the concept of abstract executability and multivariant abstract interpretation has been shown to be a very powerful program transformation and optimization tool, capable of performing essentially all the transformations traditionally done via partial evaluation [PH99, PHG99, CC02b, Leu98]. Also, the class of optimizations which can be performed can be made to cover traditional lower-level optimizations as well, provided the lower-level code to be optimized is “reflected” at the source level or if the abstract interpretation is performed directly at the object level.

Chapter 4

The Ciao System Preprocessor

In this chapter we present CiaoPP, the preprocessor of the Ciao program development system. CiaoPP uses incremental abstract interpretation to obtain information about the program, which is then used to verify programs, to detect bugs with respect to partial specifications written using assertions (in the program itself and/or in system libraries), to generate run-time tests for properties which cannot be checked completely at compile-time and simplify them, and to perform high-level program transformations such as multiple abstract specialization, parallelization, and resource usage control, all in a provably correct way. The usage of CiaoPP in this thesis is twofold. On one hand, the tools available in CiaoPP like static analysis algorithms, abstract verification code, etc. are taken as starting point for the frameworks developed in this thesis; on the other hand, in this thesis we provide frameworks which allow the use of analysis and verification on modular programs and integrate them into CiaoPP. In this chapter we follow [HPBLG05].

Ciao is a multi-paradigm programming system, allowing programming in logic, constraint, and functional styles (as well as a particular form of object-oriented programming). At the heart of Ciao is an efficient logic programming-based kernel language. This allows the use of the very large body of approximation domains, inference techniques, and tools for abstract interpretation-based semantic analysis which have been developed to a powerful and mature level in this area (see, e.g., [MH92, LV94, GdW94, BCHP96, dlBHB$^+$96a, HBPLG99] and their references). These techniques and systems can approximate at compile-time, always safely, and with a significant degree of precision, a wide range of properties which is much richer than, for example, traditional types. This includes data structure shape (including pointer sharing), independence, storage reuse, 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).

4.1 Static Analysis and Program Assertions

The fundamental functionality behind `CiaoPP` is static global program analysis, based on abstract interpretation. For this task, `CiaoPP` uses the PLAI abstract interpreter [MH92, BdlBH99] including extensions for, e.g., incrementality [HPMS00, PH96], analysis of constraints [dlBHB^{+}96b], and analysis of concurrency [MdlBH94].

The system includes several abstract analysis domains developed by several groups in the LP and CLP communities and can infer information on variable-level properties such as moded types, definiteness, freeness, independence, and grounding dependencies: essentially, precise data structure shape and pointer sharing. It can also infer bounds on data structure sizes, as well as procedure-level properties such as determinacy, termination, non-failure, and bounds on resource consumption (time or space cost). `CiaoPP` implements several techniques for dealing with "difficult" language features (such as side-effects, meta-programming, higher-order, etc.). As a result, it can for example deal safely with arbitrary ISO-Prolog programs [BCHP96]. A unified language of assertions [BCHP96, PBH00b] is used to express the results of analysis, to provide input to the analyzer, and, as we will see later, to provide program specifications for debugging and verification, as well as the results of the comparisons performed against the specifications.

Assertions and Properties: `CiaoPP` *Assertions* are a means of specifying *properties* which are (or should be) true of a given predicate, predicate argument, and/or *program point.* If an assertion has been proved to be true it has a prefix `true` –like the ones above. Assertions can also be used to provide information to the analyzer in order to increase its precision or to describe predicates which have not been coded yet during program development. These assertions have a `trust` prefix [BCHP96]. For example, if we commented out the `use_module/2`

```
:- module(qsort, [qsort/2], [assertions]).
:- use_module(compare,[geq/2,lt/2]).

qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
        lt(E,C),  partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        geq(E,C), partition(R,C,Left,Right1).

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
```

Figure 4.1: A module defining qsort.

declaration in Figure 4.1, we could describe the mode of the (now missing) `geq` and `lt` predicates to the analyzer for example as follows:

```
:- trust pred geq(X,Y) => ( ground(X), ground(Y) ).
:- trust pred lt(X,Y)  => ( ground(X), ground(Y) ).
```

The same approach can be used if the predicates are written in, e.g., an external language such as, e.g., C or Java. Finally, assertions with a `check` prefix are the ones used to specify the *intended* semantics of the program, which can then be used in debugging and/or verification, as we will see in Section 4.2. Interestingly, this very general concept of assertions is also particularly useful for generating documentation automatically (see [Her00] for a description of their use by the `Ciao` auto-documenter).

Assertions refer to certain program points. The `true pred` assertions above specify in a combined way properties of both the entry (i.e., upon calling) and exit (i.e., upon success) points of *all calls* to the predicate. It is also possible to express properties which hold at points between clause literals. As an example of this, the following is a fragment of the output produced by CiaoPP for the program in Figure 4.1 when information is requested at this level:

```
:- true pred qsort(A,B)
         : mshare([[A],[A,B],[B]])
        => mshare([[A,B]]).

qsort([X|L],R) :-
  true((ground(X),ground(L),var(R),var(L1),var(L2),var(R2), ...
  partition(L,X,L1,L2),
  true((ground(X),ground(L),ground(L1),ground(L2),var(R),var(R2), ...
  qsort(L2,R2), ...
```

In Chapter 8 there is a more detailed discussion about the assertions used for describing the intended semantics of programs.

In CiaoPP properties are just predicates, which may be builtin or user defined. For example, the property `var` used in the above examples is the standard builtin predicate to check for a free variable. The same applies to `ground` and `mshare`. The properties used by an analysis in its output (such as `var`, `ground`, and `mshare` for the previous mode analysis) are said to be *native* for that particular analysis. The system requires that properties be marked as such with a `prop` declaration which must be visible to the module in which the property is used. In addition, properties which are to be used in run-time checking (see later) should be defined by a (logic) program or system builtin, and also visible. Properties declared and/or defined in a module can be exported as any other predicate. For example:

```
:- prop list/1.
list([]).
list([_|L]) :- list(L).
```

defines the property "list". A list is an instance of a very useful class of user-defined properties called *regular types* [YS87, DZ92, GdW94, GP02, VB02], which herein are simply a syntactically restricted class of logic programs. We can mark this fact by stating "`:- regtype list/1.`" instead of "`:- prop list/1.`" (this can be done automatically). The definition above can be included in a user program or, alternatively, it can be imported from a system library, e.g.:
`:- use_module(library(lists),[list/1]).`

Type Analysis: CiaoPP can infer (parametric) types for programs both at the predicate level and at the literal level [GdW94, GP02, VB02]. The output for Figure 4.1 at the predicate level, assuming that we have imported the lists library, is:

```
:- true pred qsort(A,B)
         : ( term(A), term(B) )
        => ( list(A), list(B) ).
:- true pred partition(A,B,C,D)
         : ( term(A), term(B), term(C), term(D) )
        => ( list(A), term(B), list(C), list(D) ).
:- true pred append(A,B,C)
         : ( list(A), list1(B,term), term(C) )
        => ( list(A), list1(B,term), list1(C,term) ).
```

where term is any term and prop list1 is defined in library(lists) as:

```
:- regtype list1(L,T) # "@var{L} is a list of at least one @var{T}'s."
list1([X|R],T) :- T(X), list(R,T).
:- regtype list(L,T) # "@var{L} is a list of @var{T}'s."
list([],_T).
list([X|L],T) :- T(X), list(L).
```

We can use entry assertions [BCHP96] to specify a restricted class of calls to the module entry points as acceptable:

```
:- entry qsort(A,B) : (list(A, num), var(B)).
```

This informs the analyzer that in all external calls to qsort, the first argument will be a list of numbers and the second a free variable. Note the use of builtin properties (i.e., defined in modules which are loaded by default, such as var, num, list, etc.). Note also that properties natively understood by different analysis domains can be combined in the same assertion. This assertion will aid goal-dependent analyses obtain more accurate information. For example, it allows the type analysis to obtain the following, more precise information:

```
:- true pred qsort(A,B)
         : ( list(A,num), term(B) )
        => ( list(A,num), list(B,num) ).
:- true pred partition(A,B,C,D)
```

```
          : ( list(A,num), num(B), term(C), term(D) )
        => ( list(A,num), num(B), list(C,num), list(D,num) ).
:- true pred append(A,B,C)
          : ( list(A,num), list1(B,num), term(C) )
        => ( list(A,num), list1(B,num), list1(C,num) ).
```

Chapter 8 shows an alternative way for specifying entry points to a module, by means of `check` assertions. Chapter 7 proposes some techniques to improve the performance of the general type analysis domain, in order to enable it for the modular analysis framework.

The `CiaoPP` framework includes other analysis domains which are out of the scope of this thesis. Among them, there are:

Non-failure and Determinacy Analysis: `CiaoPP` includes a non-failure analysis, based on [DLGH97] and [BLGH04], which can detect procedures and goals that can be guaranteed not to fail, i.e., to produce at least one solution or not terminate. It also can detect predicates that are "covered", i.e., such that for any input (included in the calling type of the predicate), there is at least one clause whose "test" (head unification and body builtins) succeeds. `CiaoPP` also includes a determinacy analysis based on [LGBH05], which can detect predicates which produce at most one solution, or predicates whose clause tests are mutually exclusive, even if they are not deterministic (because they call other predicates that can produce more than one solution).

Size, Cost, and Termination Analysis: `CiaoPP` can also infer lower and upper bounds on the sizes of terms and the computational cost of predicates [DLGHL94, DLGHL97]. The cost bounds are expressed as functions on the sizes of the input arguments and yield the number of resolution steps. Various measures are used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. Note that obtaining a non-infinite upper bound on cost also implies proving *termination* of the predicate.

Decidability, Approximations, and Safety: As a final note on the analyses, it should be pointed out that since most of the properties being inferred are in general undecidable at compile-time, the inference technique used, abstract interpretation, is necessarily *approximate*, i.e., possibly imprecise.

On the other hand, such approximations are also always guaranteed to be safe, in the sense that (modulo bugs, of course) they are never *incorrect*.

4.2 Program Debugging and Assertion Validation

`CiaoPP` is also capable of combined static and dynamic verification, and debugging using the ideas outlined so far. To this end, it implements the framework described in [HPB99, PBH00a] which involves several of the tools which comprise `CiaoPP`. Figure 4.2 depicts the overall architecture. Hexagons represent the different tools involved and arrows indicate the communication paths among them.

Program verification and detection of errors is first performed at compile-time by using the sufficient conditions shown in Table 3.2, i.e., by inferring properties of the program via abstract interpretation-based static analysis and comparing this information against (partial) specifications I_α written in terms of assertions.

Dynamic checking verifies at runtime that the program execution complies with the specification. This is usually done only for those specifications which cannot be checked at compile-time.

Both the static and the dynamic checking are provably *safe* in the sense that all errors flagged are definite violations of the specifications.

In this thesis we will focus on static checking only.

Static Debugging: The idea of using analysis information for debugging comes naturally after observing analysis outputs for erroneous programs. Consider the program in Figure 4.3. The result of regular type analysis for this program includes the following code:

```
:- true pred qsort(A,B)
         : ( term(A), term(B) )
        => ( list(A,t113), list(B,^x) ).

:- regtype t113/1.
t113(A) :- arithexpression(A).
t113([]).
```

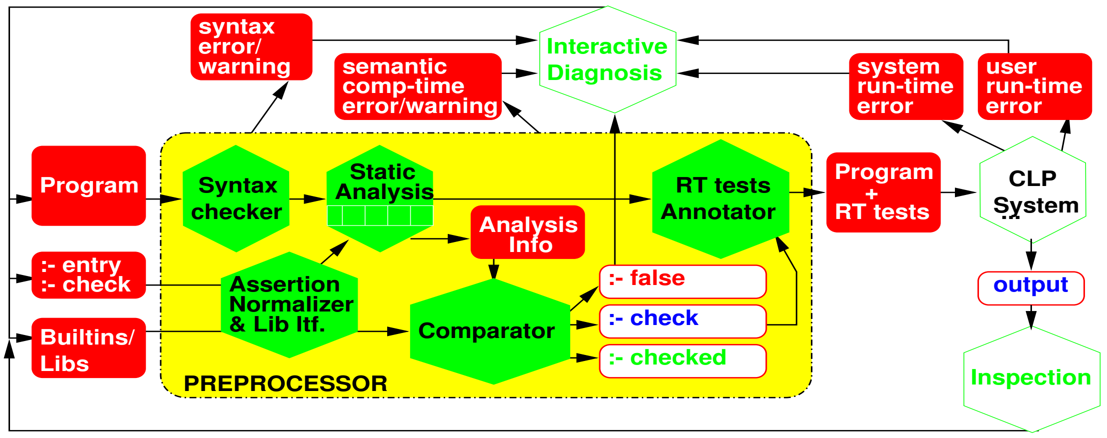


Figure 4.2: Architecture of the Preprocessor

```
:- module(qsort, [qsort/2], [assertions]).
:- entry qsort(A,B) : (list(A, num), var(B)).

qsort([X|L],R) :-
        partition(L,L1,X,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R2,[x|R1],R).
qsort([],[]).

partition([],_B,[],[]).
partition([e|R],C,[E|Left1],Right):-
        E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E >= C,   partition(R,C,Left,Right1).

append([],X,X).
append([H|X],Y,[H|Z]):- append(X,Y,Z).
```

Figure 4.3: A tentative qsort program.

```
t113([A|B]) :- arithexpression(A), list(B,t113).
t113(e).
```

where `arithexpression` is a library property which describes arithmetic expressions and `list(B,^x)` means "a list of `x`'s." A new name (`t113`) is given to one of the inferred types, and its definition included, because no definition of this type was found visible to the module. In any case, the information inferred does not seem compatible with a correct definition of `qsort`, which clearly points to a bug in the program.

Static Checking of Assertions in System Libraries: In addition to manual inspection of the analyzer output, `CiaoPP` includes a number of automated facilities to help in the debugging task. For example, `CiaoPP` can find incompatibilities between the ways in which library predicates are called and their intended mode of use, expressed in the form of assertions in the libraries themselves. Also, the preprocessor can detect inconsistencies in the program and check the assertions present in other modules used by the program.

For example, turning on compile-time error checking and selecting type and mode analysis for our tentative `qsort` program in Figure 4.3 we obtain the following messages:

```
WARNING: Literal partition(L,L1,X,L2) at qsort/2/1/1 does not succeed!
ERROR: Predicate E>=C at partition/4/3/1 is not called as expected:
         Called:   num>=var
         Expected: arithexpression>=arithexpression
```

where `qsort/2/1/1` stands for the first literal in the first clause of `qsort` and `partition/4/3/1` stands for the first literal in the third clause of `partition`.*

The first message warns that all calls to `partition` will fail, something normally not intended (e.g., in our case). The second message indicates a wrong call to a builtin predicate, which is an obvious error. This error has been detected by comparing the mode information obtained by global analysis, which at the corresponding program point indicates that `E` is a free variable, with the assertion:

```
:- check calls A<B (arithexpression(A), arithexpression(B)).
```

which is present in the default builtins module, and which implies that the two arguments to `</2` should be ground. The message signals a compile-time, or

*In the actual system line numbers and automated location of errors in source files are provided.

abstract, incorrectness symptom [BDD+97], indicating that the program does not satisfy the specification given (that of the builtin predicates, in this case). Checking the indicated call to `partition` and inspecting its arguments we detect that in the definition of `qsort`, `partition` is called with the second and third arguments in reversed order – the correct call is `partition(L,X,L1,L2)`.

After correcting this bug, we proceed to perform another round of compile-time checking, which produces the following message:

```
WARNING: Clause 'partition/4/2' is incompatible with its call type
         Head:      partition([e|R],C,[E|Left1],Right)
         Call Type: partition(list(num),num,var,var)
```

This time the error is in the second clause of `partition`. Checking this clause we see that in the first argument of the head there is an `e` which should be `E` instead. Compile-time checking of the program with this bug corrected does not produce any further warning or error messages.

Static Checking of User Assertions and Program Validation: Though, as seen above, it is often possible to detect error without adding assertions to user programs, if the program is not correct, the more assertions are present in the program the more likely it is for errors to be automatically detected. Thus, for those parts of the program which are potentially buggy or for parts whose correctness is crucial, the programmer may decide to invest more time in writing assertions than for other parts of the program which are more stable. In order to be more confident about our program, we add to it the following `check` assertions:[†]

```
:- calls qsort(A,B) : list(A, num).                     % A1
:- success qsort(A,B)  => (ground(B), sorted_num_list(B)). % A2
:- calls partition(A,B,C,D) : (ground(A), ground(B)).     % A3
:- success partition(A,B,C,D) => (list(C, num),ground(D)). % A4
:- calls append(A,B,C) : (list(A,num),list(B,num)).       % A5
:- comp partition/4 + not_fails.                          % A6
:- comp partition/4 + is_det.                             % A7
:- comp partition(A,B,C,D) + terminates.                  % A8


:- prop sorted_num_list/1.
```

[†]The `check` prefix is assumed when no prefix is given, as in the example shown.

```
sorted_num_list([]).
sorted_num_list([X]):- number(X).
sorted_num_list([X,Y|Z]):-
        number(X), number(Y), X=<Y, sorted_num_list([Y|Z]).
```

where we also use a new property, `sorted_num_list`, defined in the module itself. These assertions provide a partial specification of the program. They can be seen as integrity constraints: if their properties do not hold at the corresponding program points (procedure call, procedure exit, etc.), the program is incorrect. `Calls` assertions specify properties of all calls to a predicate, while `success` assertions specify properties of exit points for all calls to a predicate. Properties of successes can be restricted to apply only to calls satisfying certain properties upon entry by adding a "`:`" field to `success` assertions. Finally, `Comp` assertions specify *global* properties of the execution of a predicate. These include complex properties such as determinacy or termination and are in general not amenable to run-time checking. They can also be restricted to a subset of the calls using "`:`". More details on the assertion language can be found in [PBH00b].

`CiaoPP` can perform compile-time checking of the assertions above, by comparing them with the assertions inferred by analysis (see Table 3.2 and [BDD$^+$97, PBH00d] for details), producing as output the following assertions (refer also to Figure 4.2, output of the comparator):

```
:- checked calls qsort(A,B) : list(A,num).                         % A1
:- check success qsort(A,B)  => sorted_num_list(B).                % A2
:- checked calls partition(A,B,C,D) : (ground(A),ground(B)).       % A3
:- checked success partition(A,B,C,D) => (list(C,num),ground(D) ).% A4
:- false calls append(A,B,C) : ( list(A,num), list(B,num) ).       % A5
:- checked comp partition/4 + not_fails.                           % A6
:- checked comp partition/4 + is_det.                              % A7
:- checked comp partition/4 + terminates.                          % A8
```

Note that a number of initial assertions have been marked as `checked`, i.e., they have been *verified*. If all assertions had been moved to this `checked` status, the program would have been *verified*. In these cases `CiaoPP` is capable of generating certificates which can be checked efficiently for, e.g., mobile code applications [APH04]. However, in our case assertion `A5` has been detected to be false. This indicates a violation of the specification given, which is also flagged by `CiaoPP` as follows:

```
ERROR: (lns 22-23) false calls assertion:
   :- calls append(A,B,C) : list(A,num),list(B,num)
      Called append(list(^x),[^x|list(^x)],var)
```

The error is now in the call `append(R2,[x|R1],R)` in `qsort` (`x` instead of `X`). Assertions `A1`, `A3`, `A4`, `A6`, `A7`, and `A8` have been detected to hold, but it was not possible to prove statically assertion `A2`, which has remained with `check` status. Note that though the predicate `partition` may fail in general, in the context of the current program it can be proved not to fail. Note also that `A2` has been simplified, and this is because the mode analysis has determined that on success the second argument of `qsort` is ground, and thus this does not have to be checked at run-time. On the other hand the analyses used in our session (types, modes, non-failure, determinism, and upper-bound cost analysis) do not provide enough information to prove that the output of `qsort` is a *sorted* list of numbers, since this is not a native property of the analyses being used. While this property could be captured by including a more refined domain (such as constrained types), it is interesting to see what happens with the analyses selected for the example.[‡]

Part III of this thesis describes a framework and proposes several scenarios for extending the static checking of assertions to modular programs.

Performance Debugging and Validation: Another very interesting feature of `CiaoPP` is the possibility of stating assertions about the efficiency of the program which the system will try to verify or falsify. This is done by stating lower and/or upper bounds on the computational cost of predicates (given in number of execution steps).

[‡]Note that while property `sorted_num_list` cannot be proved with only (over approximations) of mode and regular type information, it may be possible to prove that it does *not* hold (an example of how properties which are not natively understood by the analysis can also be useful for detecting bugs at compile-time): while the regular type analysis cannot capture perfectly the property `sorted_num_list`, it can still approximate it (by analyzing the definition) as `list(B, num)`. If type analysis for the program were to generate a type for `B` not compatible with `list(B, num)`, then a definite error symptom would be detected.

4.3 Source Program Optimization

We now turn our attention to the program optimizations that are available in CiaoPP. These include abstract specialization, parallelization (including granularity control), multiple program specialization, and integration of abstract interpretation and partial evaluation. All of them are performed as source to source transformations of the program. In most of them static analysis is instrumental, or, at least, beneficial.

Abstract Specialization: Program specialization optimizes programs for known values (substitutions) of the input. It is often the case that the set of possible input values is unknown, or this set is infinite. However, a form of specialization can still be performed in such cases by means of abstract interpretation, specialization then being with respect to abstract values, rather than concrete ones. Such abstract values represent a (possibly infinite) set of concrete values.

CiaoPP can also apply abstract specialization to the optimization of programs with dynamic scheduling (e.g., using `delay` declarations) [PdlBMS97]. The transformations simplify the conditions on the *delay declarations* and also move delayed literals later in the rule body, leading to substantial performance improvement. This is used by CiaoPP, for example, when supporting complex computation models, such as Andorra-style execution [HBC$^+$99].

Parallelization: An example of a non-trivial program optimization performed using abstract interpretation in CiaoPP is program parallelization [BdlBH99]. It is also performed as a source-to-source transformation, in which the input program is *annotated* with parallel expressions. The parallelization algorithms, or annotators [MBdlBH99], exploit parallelism under certain *independence* conditions, which allow guaranteeing interesting correctness and no-slowdown properties for the parallelized programs [HR95, dlBHM00]. This process is complicated by the presence of shared variables and pointers among data structures at run-time.

The tests in the above example aim at *strict* independent and-parallelism. However, the annotators are parameterized on the notion of independence. Different tests can be used for different independence notions: non-strict independence [CH94], constraint-based independence [dlBHM00], etc. Moreover, all forms of and-parallelism in logic programs can be seen as independent and-parallelism, provided the definition of independence is applied at the appropriate granularity level.[§]

Resource and Granularity Control: Another application of the information produced by the `CiaoPP` analyzers, in this case cost analysis, is to perform combined compile–time/run–time resource control. An example of this is task granularity control [LGHD96] of parallelized code. Such parallel code can be the output of the process mentioned above or code parallelized manually.

In general, this run-time granularity control process involves computing sizes of terms involved in granularity control, evaluating cost functions, and comparing the result with a threshold[¶] to decide for parallel or sequential execution. Optimizations to this general process include cost function simplification and improved term size computation, both of which are illustrated in the following example.

Multiple Specialization: Sometimes a procedure has different uses within a program, i.e. it is called from different places in the program with different (abstract) input values. In principle, (abstract) program specialization is then allowable only if the optimization is applicable to all uses of the predicate. However, it is possible that in several different uses the input values allow different and incompatible optimizations and then none of them can take place. In `CiaoPP` this problem is overcome by means of "multiple program specialization" where different versions of the predicate are generated for each use. Each version is then optimized for the particular subset of input values with which it is to be used. The abstract multiple specialization technique used in `CiaoPP` [PH99] has the advantage that it can be incorporated with little or no modification of some existing abstract interpreters, provided they are *multivariant* (PLAI and similar frameworks have this property).

This specialization can be used for example to improve automatic parallelization in those cases where run-time tests are included in the resulting program. In such cases, a good number of run-time tests may be eliminated and invariants extracted automatically from loops, resulting generally in lower overheads and in several cases in increased speedups.

[§]For example, stream and-parallelism can be seen as independent and-parallelism if the independence of "bindings" rather than goals is considered.

[¶]This threshold can be determined experimentally for each parallel system, by taking the average value resulting from several runs.

Integration of Abstract Interpretation and Partial Evaluation: In the context of `CiaoPP` we have also studied the relationship between abstract multiple specialization, abstract interpretation, and partial evaluation. Abstract specialization exploits the information obtained by multivariant abstract interpretation where information about values of variables is propagated by simulating program execution and performing fixpoint computations for recursive calls. In contrast, traditional partial evaluators (mainly) use unfolding for both propagating values of variables and transforming the program. It is known that abstract interpretation is a better technique for propagating success values than unfolding. However, the program transformations induced by unfolding may lead to important optimizations which are not directly achievable in the existing frameworks for multiple specialization based on abstract interpretation. In [PAH06] we present a specialization framework which integrates the better information propagation of abstract interpretation with the powerful program transformations performed by partial evaluation. We have added state of the art local and global control strategies [PAH05, APG06] which makes `CiaoPP`'s partial evaluator a quite competitive system

Part II

Analysis of Modular Programs

Chapter 5

Intermodular, Context-sensitive Analysis of Logic Programs

Context-sensitive analysis provides information which is potentially more accurate than that provided by context-free analysis. Such information can then be applied in order to validate/debug the program and/or to specialize the program obtaining important improvements. Unfortunately, context-sensitive analysis of modular programs poses important theoretical and practical problems. One solution, used in several proposals, is to resort to context-free analysis. Other proposals do address context-sensitive analysis, but are only applicable when the description domain used satisfies rather restrictive properties. In this chapter, it is argued that a general framework for context-sensitive analysis of modular programs, i.e., one that allows using all the domains which have proved useful in practice in the non-modular setting, is indeed feasible and very useful. Driven by the experience in the design and implementation of context-sensitive analysis and specialization techniques in the CiaoPP system, the Ciao system preprocessor, in this chapter a number of design goals for context-sensitive analysis of modular programs are discussed, as well as the problems which arise in trying to meet these goals. A high-level description of a framework for analysis of modular programs is also provided, which does substantially meet these objectives. This framework is generic in that it can be instantiated in different ways in order to adapt to different contexts. Finally, the behavior of the different instantiations w.r.t. the design goals that motivate this work is also discussed.

5.1 Introduction

Analysis of logic programs has received considerable theoretical and practical attention. A number of successful compile-time techniques have been proposed and implemented which allow obtaining useful information on the program and using such information to debug, validate, and specialize the program, obtaining important improvements in correctness and efficiency. Unfortunately, most of the existing techniques are still only used in prototypes and, though numerous experiments demonstrate their effectiveness, they have not made their way into existing real-life systems. Perhaps one of the reasons for this is that most of these techniques were originally designed to be applied to a complete, monolithic program, while programs in practice invariably have a more complex structure combining a number of user modules with system libraries. Clearly, organizing program code in this modular way has many practical advantages for both program development and maintenance. On the other hand, performing global techniques such as program analysis on modular programs differs from doing so in a monolithic setting in several interesting ways and poses non-trivial problems which must be solved.

Driven by our experience in the design and implementation of context-sensitive analysis and specialization techniques in the CiaoPP system [PH03, HPBLG03b], in this chapter a high level description of a framework for analysis of modular programs is presented. This framework is generic in that it can be instantiated in different ways in order to adapt to different contexts. The correctness, accuracy, and efficiency of the different instantiations is discussed and compared.

Performing global analysis on modular programs differs from doing so in a monolithic setting in several interesting ways and poses non-trivial problems which must be solved, and it has been addressed in a number of previous works. (see, for example, [CC02a] and its references where the main approaches to separate modular static analysis by abstract interpretation are described)

However, most of them have focused on specific analyses with particular properties and using more or less ad-hoc techniques. In [CDG93] a framework is proposed for performing compositional analysis of logic programs in a modular fashion, using the concept of an *open program*, introduced in [BGLM94]. An open program is a program in which part of the code is not available to the analyzer. Nevertheless, this interesting framework is valid only for a particular set of abstract domains of analysis—those which are *compositional*.

Another interesting framework for compositional analysis for logic programs is presented in [VB00], in this case, for *binding-time analysis*. Although the most natural way to describe abstract interpretation-based binding-time analyses is arguably to use a top-down, goal-dependent framework, in this work a goal-independent analysis framework is used in order to simplify the handling of the issues stemming from modularity. The choice is based on the fact that context-sensitivity brings important problems to a top-down analysis framework. Both this paper and [CDG93] stress compositionality as a very attractive property, since it greatly facilitates modular analysis. However, there are many useful abstract domains which do not meet this property, and thus these approaches are not of general applicability.

In [Pro02] a control-flow analysis-based technique is proposed for call graph construction in the context of object oriented languages. Although there has been other work in this area, the novelty of this approach w.r.t. previous proposals is that it is context-sensitive. Also, [BJ03] shows a way to perform modular class analysis by translating the object oriented program into *open* DATALOG programs, in the sense of [BGLM94]. These two contributions are tailored to specific analysis domains with particular properties, so an important part of their work is not generally applicable nor reusable in a general framework.

In [RRL99] a two-phase analysis is proposed for incomplete imperative programs, starting with a fast, imprecise global analysis and then continuing with a (possibly context sensitive) analysis for each module in the program. This approach is not abstract interpretation-based. It is interesting to see that it appears to follow from the theory of abstract interpretation that if in such a two-pass approach the first pass "overshoots" the fixed-point, the maximum precision may not be recovered in the second pass.

In [TJ94] a method for performing separate control-flow analysis by means of abstract interpretation is proposed. This paper does not deal with the inter-modular approach studied in the present work, although it does have points in common with our module-aware analysis framework (Section 5.5). However, in this work the initial information needed by the abstract interpretation-based analyzer is provided by other analysis techniques (types and effects techniques), instead of taking advantage of the actual results from the analysis of the rest of the modules in the program.

And finally, in [Log04b, Log04a] an abstract interpretation based approach to the analysis of class-based, object-oriented languages is presented. The analysis is split in two separate semantic functions, one for the analysis of an object and another one for the analysis of the context that uses that object. The interdependence between context and object is expressed by two mutually recursive equations, that can be solved using a fixpoint computation, which somehow resembles the technique used in this thesis. In addition, the work presented in [Log04b] is context-sensitive and multivariant, since it keeps track of what is called "interaction history" at every program point. The overall fixed point (equivalent to our intermodular fixed point) starts assuming a worst-case approximation, similar to SP^+ success policies that will be described below in Section 5.5.1. This may lead to a non-least fixed point if there are mutually recursive methods in the program. In addition, the imperative nature of the target language makes the theoretical framework much more complex than the one used for logic programs and applied in this thesis. In [Log07], a later paper from the same author, `Cibai` is presented as an abstract interpretation-based analyzer for modular analysis of Java classes. However, in this case the intermodular analysis is not context sensitive, and a single bottom-up traversal of the class dependency graph is performed for a given domain (dynamic octagons). These works are quite close to the approach proposed in this thesis. However, there is no experimental evaluation of the framework, except for some preliminary tests performed on `Cibai`, where no intermodular fixed point is evaluated.

A preliminary study of the extension of context-sensitive analysis and specialization to the case of modular programs was presented in [PH00]. A full practical proposal for context-sensitive modular program analysis was presented in [BdlBH+01], which also proposed a collection of models and reported some very preliminary data from its implementation in the context of the Ciao system. Also, an implementation of [BdlBH+01] in the context of the HAL system [GDMS02] has been reported in [Net02].

These early experimental results provided initial evidence on the overall potential of the approach, but were limited in that they studied only a partial implementation. It was left as future work to perform further experimentation in order to understand the many issues and trade-offs left open in the design, and to study whether the proposed models scale and are usable in the context of large, real-life modular programs.

The rest of this chapter proceeds as follows: Section 5.2 presents a review of program analysis based on abstract interpretation and of the non-modular framework that we use as a starting point. Section 5.3 then presents some additional notation related to modular programs and a first, simple approach to extending the framework to handling such modular programs: the "flattening" approach. This approach is used as baseline for comparison throughout the rest of the thesis. Section 5.4 then identifies a number of characteristics that are desirable of a modular analysis system and which the simple approach does not meet in general. Achieving (at least a subset of) these characteristics justifies the more involved approach presented in the rest of the chapter. To this end, Section 5.5 first discusses the modifications made to the analysis framework for non-modular programs in order to be able to handle one module at a time. Section 5.6 then presents the actual full framework for analysis of modular programs. The framework proposed is parametric on the *scheduling policies*. The following sections discuss two scheduling policies which are fundamentally different: *manual scheduling* (Section 5.7), which corresponds to a scenario where one or more users decide when and what modules to analyze individually (but in a context-sensitive way), such as in distributed program development, and *automatic scheduling* (Section 5.8), where a full scheduling policy automatically determines in which order the modules will be analyzed and continues until the process is completed (a fixed-point is reached). Section 5.9 addresses some practical implementation issues, including persistence and handling of libraries. Finally, Section 5.10 compares the behavior of the different instantiations of the generic framework proposed together with that of the flattening approach w.r.t. the desirable design features discussed in Section 5.4, and presents some conclusions.

5.2 A Non-Modular Context-Sensitive Analysis Framework

The aim of context-sensitive program analysis is, for a particular description domain, to take a program and a set of initial call patterns and to annotate the program with information about the current environment at each program point whenever that point is reached when executing calls described by the initial call patterns.

5.2.1 The Generic Non-Modular Analysis Framework

We will now briefly describe the main ingredients of a generic context-sensitive analysis framework which computes the least analysis graph. This framework generalizes the particular analysis algorithms used in systems such as PLAI [MH90a, MH92], GAIA [LV94], and the CLP($\mathcal{R}$) analyzer [KMM$^+$98], and we believe captures the essence of most context-sensitive, non-modular analysis systems. More details on this generic framework can be found in [HPMS00, PH96].

We first introduce some notation. CD and AD stand for descriptions in the abstract domain. The expression $P : CD$ denotes a *call pattern*. This consists of a predicate call together with a call description for that predicate call. Similarly, $P : AD$ denotes an answer pattern, though it will be referred to as AD when it is associated to a call pattern $P : CD$ for the same predicate call.

The least analysis graph for the program is implicitly represented in the algorithm by means of two data structures, the *answer table* and the *dependency table*. Given the information in these data structures it is straightforward to construct the graph and the associated program point annotations. The answer table contains entries of the form $P : CD \mapsto AD$. It is interpreted as: the answer pattern for calls of the form CD to P is AD. A dependency is of the form $P : CD_0 \Rightarrow B_{key} : CD_1$. This is interpreted as follows: if the procedure P is called with description CD_0 then this causes the procedure B to be called with description CD_1. The subindex *key* can be used in order to uniquely identify the program point within P where B is called with calling pattern CD_1. Dependency arcs represent the arcs in the program analysis graph from procedure calls to the corresponding call pattern.

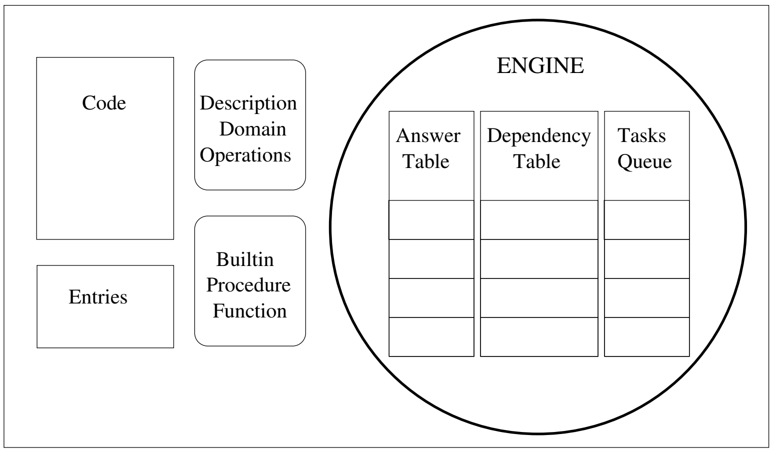


Figure 5.1: Non-Modular Analysis Framework

Intuitively, different analysis algorithms correspond to different graph traversal strategies which place entries in the answer table and dependency table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixed-point algorithms, we use a priority queue. The queue contains the events to process. Different priority strategies correspond to different analysis algorithms. Thus, the third, and final, structure used in our generic framework is a *tasks queue*.

When an event being added to the tasks queue is already in the queue, a single event with the maximum of the priorities is kept in the queue. Also, only one arc of the form $P : CD \Rightarrow B_{key} : CD'$ for each tuple (P, CD, B_{key}) exists in the dependency table: the last one added. The same holds for entries $P : CD \mapsto AD$ for each tuple (P, CD) in the answer table.

Figure 5.1 shows the architecture of the framework. The *Code* corresponds to the (source) code of the program to be analyzed. By *Entries* we denote the initial starting points for analysis. The box *Description Domain Operations* represents the definition of operations which are domain dependent. The circle represents the *Analysis Engine*, which has the three data-structures mentioned above, i.e., the answer table, the dependency table, and the tasks queue. Initially, for each analysis these three structures are empty and the analysis engine takes care of processing the events on the priority queue by repeatedly removing the highest priority event and calling the appropriate event-handling function. This in turn consults and modifies the contents of the answer and dependency tables. When the tasks queue becomes empty then the analysis engine has reached a fixed-point. This implies that the least analysis graph has been found. We will use $Analysis_{D_\alpha}(Q, E) = (AT, DT)$ to denote that the analysis of program Q for initial descriptions E in domain D_α produces the answer table AT with dependency table DT.

5.2.2 Predefined Procedures

In order to simplify their presentation, formalizations of program analysis often do not consider *predefined* procedures. However, in practice, program analysis implementations allow the use of predefined (language built-in and/or library) procedures[*] in the programs to be analyzed. These *external* procedures whose code is not available in the program being analyzed are often handled in an *ad-hoc* way. Thus, in fairness, non-modular program analyses are more accurately represented by adding to the framework a *builtin procedure function* which essentially hardwires the answer table for these external procedures. This function is represented in Figure 5.1 by the box *builtin procedure function.* We will use $\mathcal{CP}$ and $\mathcal{AP}$ to denote, respectively, the set of all call patterns and the set of all answer patterns. The builtin procedure function can be formalized as a function $BF : \mathcal{CP} \rightarrow \mathcal{AP}$. For all call pattern $P : CD$ where P is a builtin procedure $BF(P : CD)$ returns a description AD which is assumed to be correct in the sense that it is a safe approximation, i.e. an over-approximation of the actual answer pattern for $P : CD$.

It is important to note that the data structures which are outside the analysis engine, *code*, *entries*, *description domain operations*, and *builtin procedure function* are read-only. However, though the code and entries are supposed to change for the analysis of each particular program, the *builtin procedure function* can be considered to be fixed, for each description domain D_α, in that it does not vary from the analysis of one program to another. Indeed, it can be considered to be part of the analyzer. Thus, the builtin procedure function is not explicitly represented as an input to the analysis algorithm.

[*]In our modular design, a library can be treated simply as (yet another) module in the program. However, special practical considerations for them will be discussed in Section 5.9.3.

5.3 The Flattening Approach to Modular Processing

Several *compilation tasks* such as program analysis and specialization are traditionally considered *global*, as opposed to *local*. Usually, local tasks process one procedure or module at a time and all the information required for performing the task can be obtained by inspecting that procedure. In contrast, in global tasks the results of processing a part of the program (say, a procedure) may be needed in order to process other parts of the program. Thus, global processing often requires iterating on the whole program until a fixed-point is reached.

In a modular setting, it may well be the case that part of the information needed to perform the task on (a procedure in) module m has to be computed in modules other than m. We will refer to the information originated in modules different from m as *inter-modular* information in contrast to the information originated in m itself, which we will call *intra-modular*.

Example 5.3.1. *In context-sensitive program analysis there is an information flow of both call and success patterns to and from procedures in different modules. Thus, program analysis requires inter-modular information. For example, the module c receives call patterns from module a since* $\mathrm{callers}(c) = \{a\}$*, and it has to propagate the corresponding success patterns to a. In turn, c provides* $\{e, f\} = \mathrm{imports}(c)$ *with call patterns and receives success patterns from them.*

5.3.1 Flattening a Program Unit vs. Modular Processing

Applying a framework for non-modular programs to a module m has the difficulty that m may not be self-contained. However, there should be no problem in applying the framework if m is a leaf module. Furthermore, given a global process such as program analysis, at least in principle, it is not obvious that it makes much sense to apply the process to a module m alone. In principle, it makes more sense to apply it to program units since they are conceptually self-contained. Thus, given a module m one natural approach seems to be to apply the tool (simultaneously) to all the modules in $U = \mathit{program\_unit}(m)$.

Given a program unit U it is always possible to build a single module m_{flat} which is equivalent to U and which is a leaf. The process of constructing such a module m_{flat} usually only amounts to renaming apart identifiers in the different modules in U so as to avoid name clashes. We will use $\mathit{flatten}(U) = m_{flat}$ to denote that the module m_{flat} is the result of renaming apart the code in each module in U and concatenating its code into a monolithic module m_{flat}. This points to a simple solution to the problem of processing modular programs (at least for the case in which all the code is available): to transform $\mathit{program\_unit}(m)$ into the equivalent monolithic program m_{flat}. It is then straightforward to apply any tool for non-modular programs to the leaf module m_{flat}. Figure 5.2 represents the case in which the non-modular analysis framework is used on the flattened program.

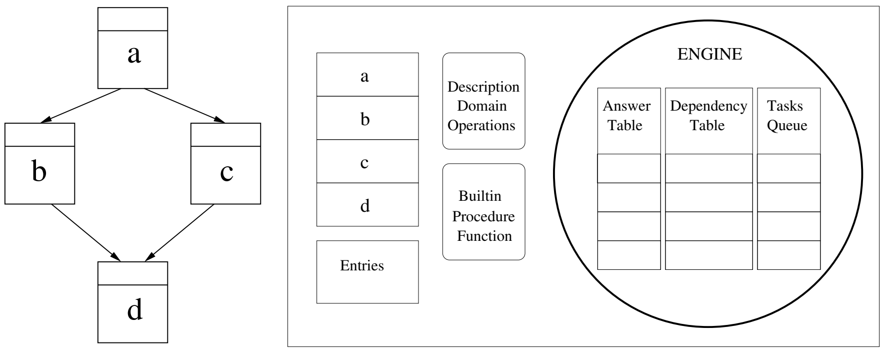


Figure 5.2: Using non-modular analysis on a flattened program

Given the existence of an implementation for non-modular analysis, this approach is often simple to apply. Also, this flattening approach has theoretical interest. It can be used, for example, in order to compare the efficiency of different approaches to modular handling of programs w.r.t. the flattening approach. However, as a practical way in which to actually perform analysis of program units this approach has important drawbacks. This issue will be discussed in more detail in Section 5.10. The most obvious one is that the complete program must be loaded into the analyzer, and thus large programs may make the analyzer run out of memory. Moreover, as the internal analysis data structures include information for all the program source code, in the monolithic case analysis of a given procedure may take more time than when keeping in memory only the module in which such procedure resides. Another, perhaps more important drawback is that the program must be self-contained: this can be a problem if the analyzer is used while developing the program, when some modules are not yet implemented, or if there are calls to external procedures, i.e., procedures for which the source code is not available, or which are implemented in other languages.[†]

5.4 Design Goals for Analysis of Modular Programs

Before presenting our proposals for analysis of modular programs, we will discuss the main features which should be taken into account when designing and/or implementing a tool for context-sensitive analysis of modular programs. As often happens in practice, some of the features presented are conflicting with others and this might make it impossible to find a framework which behaves optimally w.r.t. all of them.

Module-Awareness We consider a framework *module-aware* when it has been designed with modules in mind. Thus, it is applicable to a module m by using the code of m and some "interface" information for the modules in $imports(m)$. Such interface information will in general consist of a summary of previous analysis results for such modules, if such results are available, or a safe approximation if they are not.

Though transforming a non-modular framework into a module-aware one may seem trivial, it requires identifying precisely which is the required information on the result of applying the tool in each of the modules in $imports(m)$ which should be stored in order to apply the tool to m. This corresponds in general to the inter-modular information. It is also desirable that the amount of such information be minimal.

Example 5.4.1. *The framework for non-modular analysis in Section 5.2 is indeed non-modular since it requires the code of all procedures (except possibly for some predefined ones) to be available to the analyzer. It will produce wrong results when applied to non-leaf modules since a missing procedure can only be deemed as an error, unless the framework is aware that such a procedure can be imported.*

[†]However, several approaches have been proposed for the analysis of incomplete programs (*open programs*), for example [BCHP96, BJ03].

Correctness The results of applying the tool to a module m should produce results which are *correct*. The notion of correctness itself can in general be lifted from the non-modular case to the modular case without great difficulties. A more complex issue is how to extend a framework to the modular case in such a way that correctness is preserved.

Accuracy Similarly, the analysis results for a module m should be as accurate as possible. The notion of accuracy can be defined by comparing the analysis results with those which would be obtained using the flattening approach presented in Section 5.3.1 above, since the latter always computes the most accurate information possible, which corresponds to the least analysis graph.

Termination A framework for analysis of modular programs should guarantee termination (at least) in all cases in which the flattening approach terminates (which, typically, is for every program). Such termination is guaranteed by choosing description domains with some specific characteristics such as having finite height, finite ascending chains, etc., and/or incorporating a *widening operator*.

Efficiency in Time The time required to apply the tool should be reasonable. We will understand "reasonable" as not over an acceptable threshold on the time taken using the flattening approach.

Efficiency in Memory In general, one of the main expected advantages of the modular approach is that the total amount of memory required to handle each module separately should be smaller than that needed in the flattening approach.

No Need for Analyzing All Call Patterns Under certain circumstances, applying a tool on a module m may require processing only a subset of the call patterns rather than all call patterns for m. In order to achieve this, the model must keep track of fine-grained dependencies. This will allow marking exactly those call patterns which need processing. Other call patterns not marked do not need to be processed.

Support for the Co-Existence of Multiple Program Units/Applications In a modular setting it is often the case that a particular module is used in several applications. Support for software reuse is thus a desirable feature. However, this poses additional and interesting challenges to the tools, some of which will be discussed in Section 5.9.

Support for Source Changes What happens if the source of a module changes during processing? Some tools will not allow this at all and if it happens all the processing has to start again from scratch. This has the disadvantage that the tool is then not incremental since a (possibly minor) change in a module invalidates the information for all the program unit. Other tools may delete the information which may depend on the changed code, but still keep the information which does not depend on it.

Persistence This feature indicates that the inter-modular information can be stored in a persistent medium, such as a file stored on disk or a database, and allow later recovery of such information.

5.5 Analysis of Modular Programs: The Local Level

As a first step towards introducing our analysis framework for modular programs, which will be presented in Section 5.6 below, in this section we discuss the main ingredients which have to be added to an analysis framework for non-modular programs in order to be able to handle one module at a time.

Analyzing a module separately presents the difficulty that, from the point of view of analysis, the code to be analyzed is *incomplete* in the sense that the code for procedures imported from other modules is not available to analysis. More precisely, during analysis of a module m there may be calls $P : CD$ such that the procedure P is not defined in m but instead it is imported from another module $m' \in \mathit{imports}(m)$. We refer to determining the value of AD to be used for $P : CD \mapsto AD$ as the *imported success problem.* In addition, in order to obtain analysis information for m' which is as accurate as possible we need to somehow propagate the call $P : CD$ from m to m' so that the next time m' is analyzed such a call pattern is taken into account. We refer to this as the *imported calls*

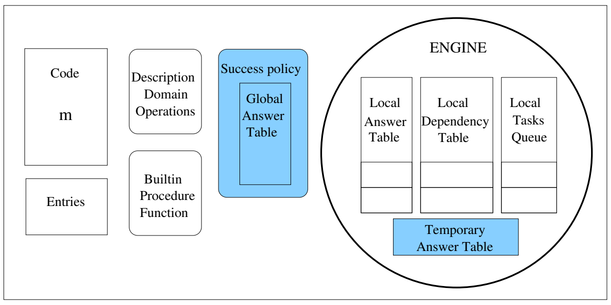


Figure 5.3: Module-aware analysis framework

problem. Note that in this case analysis has to be module-aware in order to determine whether a given procedure is either local or imported (or predefined).

Figure 5.3 shows the architecture of an analysis framework which is module-aware. This framework is an extension of the non-modular framework in Figure 5.1. One minor change is that the read/write data structures internal to the analysis engine have been renamed with the prefix "local". So now we have the *local answer table*, the *local dependency table*, and the *local task queue*. Also, the box which represents the code now contains m indicating that it contains the single module m.

The shaded boxes in Figure 5.3 indicate the main differences w.r.t. the non-modular framework. One is that in the module-aware framework there is an additional read-only[‡] data structure, the *global answer table*, or GAT for short. Its contents are identical in format to those in the answer table of the non-modular framework. There are however some differences: (1) the GAT contains analysis results which were obtained previously to the current analysis step. (2) The GAT contains entries which correspond to predicates defined in $imports(m)$, whereas all entries in the local answer table (or LAT for short) are for predicates defined in m itself. (3) Only information of exported predicates is available in GAT. The LAT contains information for all predicates in m regardless of whether they are exported or not.

[‡]In fact, this data structure is read/write at the global level discussed in Section 5.6 below, but it is read-only as regards our engine for analysis of one module.

5.5.1 Solving the Imported Success Problem

The second important difference is that the module-aware framework requires the use of a *success policy*, or SP for short, which is represented in Figure 5.3 with a shaded box surrounding the GAT. The SP can be seen as an intermediator between the GAT and the analysis engine. The behavior of the analysis engine for predicates defined in m remains exactly as before. SP is needed because though the information in the GAT will be used in order to obtain answer patterns for imported predicates, given a call pattern $P : CD$ it will often be the case that an entry of exactly the form $P : CD \mapsto AD$ does not exist in GAT. In such case, the information already present in GAT may be of value in order to obtain a (temporary) answer pattern AD, and continue the analysis of the module. Note that the GAT together with SP will allow solving the "imported success problem".

In contrast, in many formalizations of non-modular analysis there is no explicit success policy. This is because if the call pattern $P : CD$ has not been analyzed yet, the analysis algorithm forces its computation. Thus, the results of analysis do not depend on any particular success policy: when analysis reaches a fixed-point there is always an entry of the form $P : CD \mapsto AD$ for any call pattern $P : CD$ which appears in the analysis graph. Unfortunately, in a modular setting it is not directly possible to force the analysis of predicates defined in other modules. Those modules may have already been analyzed or they may be analyzed in the future. We will simply do the best possible given the information available in GAT.

We will use $\mathcal{GAT}$ to denote the set of all global answer tables. The success policy can be formalized as a function $SP : \mathcal{CP} \times \mathcal{GAT} \rightarrow \mathcal{AP}$. Several success policies can be defined which provide over- or under-approximations of the exact answer pattern $AD^=$ with different degree of accuracy. Note that this exact value $AD^=$ is the one which the flattening approach (that we will thus denote $SP^=$) would compute. In this work we consider two kinds of success policies, those which are guaranteed to always provide over-approximations, i.e. $AD^= \sqsubseteq SP(P : CD, AT)$, and those which provide under-approximations, i.e., $SP(P : CD, AT) \sqsubseteq AD^=$. We will use the superscript $^+$ (resp $^-$) to indicate that a success policy over-approximates (resp. under-approximates). As will be discussed later in this chapter, both over- and under-approximations are useful in different contexts and for different purposes. Since it is always required to know whether a success policy over- or under-approximates we will mark all success policies in either way.

Example 5.5.1. *A very precise over-approximating success policy is the function SP^+_{All} defined below, already proposed in [PH00]:*

$$\begin{array}{l} SP^+_{All}(P : CD, GAT) = topmost(CD) \sqcap_{AD' \in app} AD' \text{ where} \\ \quad app = \{AD' \mid (P : CD' \mapsto AD') \in \ GAT \text{ and } CD \sqsubseteq CD'\} \end{array}$$

The function topmost obtains the topmost answer pattern for a call pattern. The notion of topmost description *was already introduced in [BCHP96]. Informally, a topmost description preserves the information on those properties which are* downwards closed *whereas it loses information for those which are not. Note that taking $\top$ as answer pattern is a correct over-approximation, but often less accurate than using topmost substitutions. For example, if a variable is known to be ground in the call pattern, it will continue being ground in the answer pattern and taking $\top$ as the answer pattern would lose this information. However, the fact that a variable is free on call does not guarantee that it will keep on being free on success.*

We refer to this success policy as SP^+_{All} because it uses all *entries in GAT which are* applicable *to the call pattern in the sense that the call pattern already computed is more general than the call being analyzed.*

Example 5.5.2. *The counter-part of SP^+_{All} is the function SP^-_{All} which is defined as:*

$$\begin{array}{l} SP^-_{All}(P : CD, GAT) = \sqcup_{AD' \in app} AD' \text{ where} \\ app = \{AD' \mid (P : CD' \mapsto AD') \in \ GAT \text{ and } CD' \sqsubseteq CD\} \end{array}$$

Note the change in the direction of the applicability relation (the call pattern in the GAT *has to be more particular than the one being analyzed) and the use of the lub operator instead of the glb. Also, note that taking, for example, $\perp$ as an under-approximation is correct but SP^-_{All} is more precise.*

5.5.2 Solving the Imported Calls Problem

The third important difference w.r.t. the non-modular framework is the use of the *temporary answer table* (or *TAT* for short) and which is represented as a shaded box within the analysis engine of Figure 5.3. This answer table will be used to store call patterns for imported predicates which are not yet present in GAT and whose answer pattern has been obtained (approximated) using the success policy on the entries currently stored in GAT. The TAT is used as a cache for imported call patterns and their corresponding answer patterns, thus avoiding having to repeatedly apply the success policy on the GAT for equivalent call patterns, which is an expensive operation. Also, after analysis of the current module is finished, the existence of the TAT simplifies the way in which the global data structures need to be updated. This will be discussed in more detail in Section 5.6 below.

We use $analysis_{SP}(m, E_m, GAT) = (LAT_m, LDT_m, TAT_M)$ to denote that the module-aware analysis framework returns (LAT_m, LDT_m, TAT_M) when applied to module m with respect to the set of initial call patterns E_m with success policy SP and GAT. In subsequent chapters in which LDT and TAT are irrelevant, we will assume that the result of the module-aware analysis of a module will simply be the local answer table, $analysis_{SP}(m, E_m, GAT) = LAT_m$. The success policy and the global answer table will be omitted as well when they are irrelevant or not needed for describing the analysis of a module.

Depending on how the module is being analyzed and what will be the analysis results used for, there are several *entry policies* applicable. For example, if the module m is analyzed during the computation of an intermodular fixed point, as it will be shown in Section 5.8, the best entry policy applicable is to take as entry points of the analysis the GAT entries of the modules which import m. If on the other hand it will be used during the development or verification of a single module, as described in Section 5.7, an entry policy that takes into account all exported predicates will be preferred.

5.6 Analysis of Modular Programs: The Global Level

After discussing the *local-level* issues which appear when analyzing a module, in this section we present a complete framework for the analysis of modular programs. Since analysis is a global task, an analysis framework should not only deal with local-level information, but also with global-level information. A

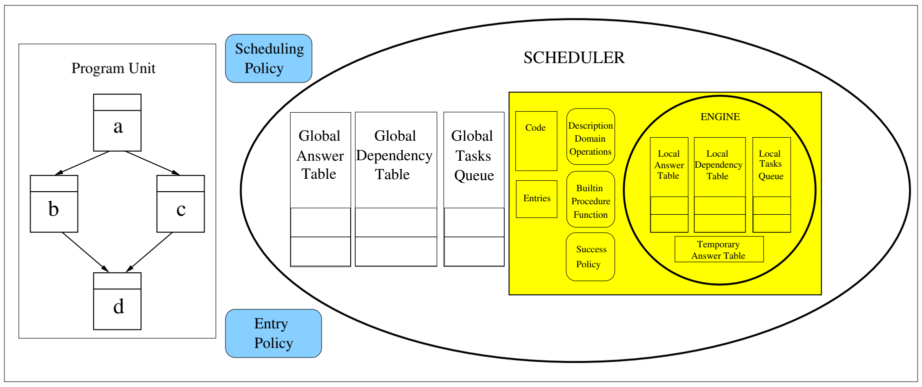


Figure 5.4: A two-level framework for analysis of modular programs

graphical representation of our framework is depicted in Figure 5.4. The main idea is that we have to add a higher-level component to the framework which takes care of the *inter-modular* information, as opposed to the *intra-modular* information which is handled by the local-level subsystem described in the previous section.

As a result, analysis of modular programs is best seen as a two-level process. Note that the inner, lightly shaded, rectangle corresponds exactly to Figure 5.3 as it is a module-aware analysis system. It is interesting to see how the data structures in the global and local levels are indeed very similar. The similarities and differences between the GAT and LAT have been discussed already in Section 5.5 above. Regarding the global and local dependency tables (GDT and LDT respectively), they are used in order to be able to propagate as precisely as possible which parts of the analysis graph have to be recomputed. The GDT is used in order to add events to the global task queue (GTQ) whereas the LDT is used to add events ($arcs$) to be (re-)analyzed to the local task queue (LTQ). We can define the events to be processed at the global level using different levels of granularity. As usual, the finer-grained these events are, the more detailed and thus more effective the handling of the events can be. One obvious possibility is to use modules as events. This means that all call patterns which correspond to a module are handled simultaneously whenever the module is selected at the global level. A more refined possibility is to keep events at the call pattern level. This, together with sufficiently detailed information in the GDT will allow incrementality at the call pattern level rather than module level.

5.6.1 Parameters of the Framework

The framework has three parameters. The *program unit* corresponds to the program unit to be analyzed. Note that the code may not be physically stored in the tool's memory since it is already on external storage. However, the framework may maintain some information on the program unit, such as dependencies among modules, *strongly connected components*, and any other information which may be useful in order to guide analysis. In the figure the *program unit* is represented, as an example, containing a program unit composed of four modules. The second parameter is the *entry policy*, which determines the way in which the GTQ and GAT should be initialized whenever analysis of a program unit is started. Depending on how the success policy is defined, entries for all procedures exported in each of the modules in the program unit may be required in GAT and GTQ or not.

Finally, the *scheduling policy* determines the order in which the entries in the GTQ should be processed. The efficiency with which the fixed-point is reached can differ very much from some scheduling policies to others. Since the framework presented in Figure 5.4 has just one analysis engine, processing a call pattern in a different module from that currently loaded has a relevant cost associated to it, since this often requires context switching from the current module to a new module. Thus, it is often a good idea to process all or many of the call patterns in GTQ which correspond to the module which is being analyzed in order to minimize the number of times the analysis tool has to switch from one module to another. In the rest of the chapter we consider that events in GTQ are answer patterns which would benefit from (re-)analysis. The role of the scheduling policy is to select a set of patterns from GTQ which must necessarily belong to the same module m to be analyzed. Note that a scheduling policy based on modules can always be obtained by simply processing at each analysis step all events in GTQ which correspond to m.

5.6.2 How the Global Level Works

As already mentioned, analysis of a modular program starts by initializing the global data structures as indicated by the entry policy. At each step, the scheduling policy is used to determine the set E_m of entries for module m which are to be processed. They are removed from GTQ and copied into the data structure *Entries*. The code of the module m is also copied to *code*. Then, $analysis_{SP}(m, E_m, GAT) = (LAT_m, LDT_m, TAT_m)$ is computed. Then, the global data structures are updated, as detailed in Section 5.6.3 below. As a result of this, new events may be added to GTQ. Analysis terminates when there are no more events to process in GTQ or when the scheduling strategy does not select any further events.

Each entry in GTQ is of one of the following three types: *over-approximation*, *under-approximation*, or *invalid*, according to the reason why they should be re-analyzed. An entry $P : CP \mapsto AP$ which is an over-approximation is marked $P : CP \mapsto^{+} AP$. This indicates that the answer pattern AP is possibly an over-approximation since it depends on a call pattern whose answer pattern has been determined to be an over-approximation. In other words, the accuracy of $P : CP \mapsto AP$ may be improved by re-analysis. Similarly, under-approximations are marked $P : CP \mapsto^{-} AP$ and they indicate that AP is probably an under-approximation since it depends on a call pattern whose success pattern has increased. As a result, the call pattern should be re-analyzed to guarantee correctness. Finally invalid entries are marked $P : CP \mapsto^{\perp} AP$. They indicate that the relation between the current answer pattern AP and one resulting from recomputing it for $P : CP$ is unpredictable. This often indicates that the source code of the module has changed in a way that the analysis results for some of the exported procedures are just incompatible with previous ones. Handling this kind of events is discussed in more detail in Section 5.6.4 below.

5.6.3 Updating the Global State

In Section 5.5 it has been presented how the local level subsystem, given a module m, can compute the corresponding LAT_m, LDT_m, and TAT_m. However, once analysis of module m is done, the analysis results of module m have to be used in order to update the global state prior to starting analysis of any other module.

We now briefly discuss how this updating is done. For each initial call pattern $P : CP$ in *Entries* we compare the previous answer pattern AP with the newly computed one AP'. If $AP = AP'$ then this call pattern has not been affected by the latest analysis. However, it is also possible that the answer pattern "evolves" in different analysis iterations. If we use SP^{+}, the natural thing is that the new answer pattern is more specific than the previous one, i.e., $AP' \sqsubset AP$. In such case those call patterns which depend on $P : CP$ can also improve their success pattern. We use the GDT to locate all such patterns and we add them to the GTQ with the $^+$ mark. Conversely, if we use SP^-, the natural thing is that $AP \sqsubset AP'$. We then add events marked $^-$.

In a typical situation, and if modules do not change, all events in GTQ will be approximations of the same sign. This depends on the success policy used. If the success policy is of kind SP^+ (resp. SP^-) then the events which will be added to GTQ will also be over-approximations (resp. under-approximations). In turn, when they are processed they will introduce other over-approximations (resp. under-approximations).

The TAT_m is also used to update the global state. All entries in TAT_m are added to GAT and GTQ marked with the same sign as the success policy used. Last, we also have to update the GDT. For this, we first erase all entries for any of the call patterns which we have just analyzed, and which are thus stored in $entries_m$. Then we add an entry of the form $P : CP \rightarrow H : CP'$ for each imported procedure H which is reachable with call pattern CP' from an initial call pattern $P : CP$. Note that this can easily be determined using LDT.

5.6.4 Recovering from an Invalid State

If code of a module m has changed since it was last analyzed, it can be the case that the global information available is invalid. This happens when in the results of re-analysis of m any of the exported predicates has an answer pattern which is incompatible with the previous results. In this case, all information dependent on the new answer patterns might have become invalid, as discussed in Section 5.6.2. The question is how to minimize the impact of such a situation.

The simplest solution is to (transitively) erase any information of other modules which depends on the invalidated one. This solution may not be very efficient, as it ignores all results of previous analyses of other modules even if the changes performed in the module are minor, or only affect directly related modules. Another alternative is to launch an automatic recovery process as soon as invalid analysis results are detected (see [BdlBH$^+$01]). This process has to reanalyze the modules directly affected by the invalidated answer pattern(s). If the new answer patterns coincide with the old ones then the changes do not affect this module and the process terminates. Otherwise, it continues transitively with the directly related modules.

5.7 Using a Manual Scheduling Policy

Consider, for example, the relevant case of independent development of different parts of the program, which can then even be performed in parallel by different teams. In this setting, it makes sense that the analyzer performs its job on the current module without analyzing other modules in the program unit, i.e., it allows separate analysis. This will typically allow early detection of compile-time errors in the current module without having to wait for the code of the dependent modules to be fully developed. Moreover, in this setting, it is the user (or users) who decide when and what to analyze. Thus, we refer to this as the *manual* setting. Furthermore, we assume that in this setting analysis for a module m has to do its best with only the code for m plus the results of previous analyses (if any) of the modules in $depends(m)$. These assumptions have important implications. The setting allows the users of different modules to decide when they should be processed. And thus, any module could be (re-)analyzed at any point. As a result, strong requirements must hold for the whole approach to be correct. In return, the results obtained may not be optimal (in terms of error detection, degree of optimization, etc., depending on the particular tools) w.r.t. those achievable using automatic scheduling.

So the question is, is there any combination of the three parameters of the global analysis framework which allows handling the manual setting? The answer to this question is yes. Our earlier paper [BdlBH$^+$01] essentially describes such an instantiation of the analysis framework. In the terminology of the current chapter, the model in [BdlBH$^+$01] corresponds to waiting until the user requests that a module m in the program unit U be analyzed. The success policy is over-approximating. This guarantees that in the absence of invalidated entries in the GTQ all events will be marked $^+$. This means that the analysis information available is correct, though perhaps not as accurate as possible. Since the scheduling is manual, no other analyses should be triggered until the user requires so. Finally, the entry policy is simply to include in GTQ an event such as $P : \top \mapsto^+ \top$ per predicate exported by any of the modules in U to be analyzed (it is called *all* entry policy). The initial events are required to be so general to keep the overall correctness of the analysis while allowing the users to choose the order of the modules to be analyzed.[§] The model in [BdlBH$^+$01] has the very important feature of being guaranteed to always provide correct results without the need of reaching a global fixed-point.

5.8 Using an Automatic Scheduling Policy

In spite of the evident interest of the manual setting, there are situations in which the user is interested in obtaining the most accurate analysis results possible. For this, it may be required to analyze the modules in the program unit several times in order to converge to a distributed global fixed-point. We will refer to this as the *automatic* setting, in which the user decides when to start global analysis of a program unit. From then on it is the global analysis framework by means of its *scheduling policy* who decides when and what to analyze. Note that the manual and automatic settings roughly correspond to scenario 1 and scenario 2 of [PH00] respectively. Since we admit circular dependencies among modules, the strategy has to be able to deal with such circularities correctly and efficiently without entering infinite loops. The question now is what are the values for the different parameters to our generic framework which should be used in order to obtain satisfactory results? One major difference of the automatic setting w.r.t. the manual setting is that in addition to over-approximations, now also under-approximations can be used. This is because though under-approximations do not guarantee correctness in general, when an inter-modular fixed-point is reached, analysis results are guaranteed to be correct. Below we consider the use of SP^+ and SP^- separately.

We will use the function $GAT = modular\_analysis(m)$ to refer to the analysis of the program unit m reaching an intermodular fixed-point, that returns as result the global answer table. When computing the intermodular fixed point, $analysis(n, E, AT)$ is invoked for each module n in the program unit, where E is the set of calling patterns in GAT for predicates defined in n which need to be (re)analyzed, AT is the current state of the GAT, and the GAT is updated after

[§]In the case of the Ciao system it is possible to use *entry* declarations (see for example [PBH00b]) in order to improve the set of initial call patterns for analysis.

analysis with information from the resulting LAT. See [PCH+04] for details. We can define a partial ordering on answer tables over a given module in the following sense: $AT_1 \preceq AT_2$ iff $\forall(P : CP_1 \mapsto AP_1) \in AT_1, (\exists(P : CP_2 \mapsto AP_2) \in AT_2$ s.t. $CP_1 \sqsubseteq CP_2$ and $\forall(P : CP'_2 \mapsto AP'_2) \in AT_2$, if $CP_1 \sqsubseteq CP'_2$ then $AP_1 \sqsubseteq AP'_2)$.

5.8.1 Using Over-Approximating Success Policies

If a success policy SP^+ is used, we are in a situation similar to the one in Section 5.7 in that independently of how many times each module has been analyzed, if there have not been any code changes, the analysis results are guaranteed to be correct. The main difference is that now the system keeps on automatically requesting further analysis steps until a fixed-point is reached.

The drawback is that when the fixed point is reached it may not be minimal, i.e., information is not as precise as it could be. It can be verified that, in the common case in which there are not mutually recursive calls that traverse module boundaries, the least fixed point will be reached.

Regarding the entry policy, an important observation is that in the automatic mode, much as in the case of intra-modular analysis, inter-modular analysis will eventually compute all call patterns which are needed in order to obtain information which is correct w.r.t. calls, i.e., the set of computed call patterns covers all possible calls which may occur at run-time for the class of initial calls considered, i.e., those for the top-level of the program unit U. This will allow us to use a different entry policy from that used in the manual mode: rather than introducing events of the form $P : \top \mapsto^+ \top$ in the GTQ for exported predicates in all modules in U, it suffices to introduce them for predicates exported by the top-level of U (this entry policy is named *top-level* entry policy). This has several important advantages: (1) It avoids analyzing all predicates for the most general call pattern, since this may end up introducing plenty of call patterns which are not used in our particular program unit U. (2) It will help to have a more guided scheduling policy since there are no requests for processing a module until it is certain that a call pattern should be analyzed. (3) If multiple specialization is being performed based on the set of call patterns for each procedure (possibly proceeded by a minimization step for eliminating useless versions [PH99]), the fact that a call pattern with the most general call pattern exists implies that a non-optimized version of the predicate must always exist. Another way out of this problem is to eliminate useless call patterns once an inter-modular fixed-point has been reached.

Since reaching a global fixed-point can be a costly task, one interesting possibility can be the introduction of a time-out. The user can ask the system to request (re-)analysis as needed towards improving the analysis information. However, if after performing n analysis steps the time-out is reached before analysis $n + 1$ is finished, the global state corresponding to state n is guaranteed to be correct. In this case, the entry policy used has to be to introduce most general call patterns for all exported predicates, either before starting analysis or when a time-out is reached.

5.8.2 Using Under-Approximating Success Policies

Another alternative is to use SP^-. As a result, the analysis results are not guaranteed to be correct until an inter-modular fixed-point is reached. Thus, it may take a large amount of time to perform this global analysis. On the other hand, once a fixed-point is reached, the accuracy which will be obtained is optimal, since it corresponds to the least analysis graph, which is exactly the same which the flattening approach would have obtained.

Regarding the entry policy, the same discussion as above applies. The only difference being that the GTQ should be initialized with events of the form $P : \top \mapsto^- \bot$ since now the framework computes under-approximations. Clearly, $\bot$ is an under-approximation of any description.

Another important thing to note is that, since the final results of automatic analysis are optimal, they do not depend on the use of a particular success policy SP_1^- or another SP_2^-. Of course, the efficiency using SP_1^- can be very different from that obtained using SP_2^-.

5.8.3 Hybrid policy

In practice we may wish to use a manual scheduling policy with an over-approximating success policy during program development, and then use an automatic scheduling policy with an under-approximating success policy just before program release, so as to ensure that the analysis is as precise as possible, thus allowing as much optimization as possible in the final version.

Fortunately, in such a situation we can often reuse much of the analysis information obtained using the over-approximating success policy. The reason is that if the analysis with the over-approximating success policy has reached a fixed-point, the answers obtained for module m are as accurate as those obtained with an under-approximating success policy as long as there are no cyclic dependencies between the modules in $\mathit{depends}(m)$. Thus in the common case that no modules are mutually dependent we can simply use the answer tables from the manual scheduling policy and use an automatic scheduling policy with an over-approximating success policy to obtain the fixed-point. Even in the case that some modules are mutually dependent we can use this technique to compute the answers for the modules which do not contain cyclic dependencies or do not depend on modules that contain them (e.g., leaf-modules).

5.8.4 Computation of an Intermodular Fixed-Point

Determining the optimal order in which the different modules in the program unit should be analyzed in order to get to a fixed-point as efficiently as possible is not trivial.

Finding good scheduling strategies for intra-modular analysis is a topic which has received considerable attention and highly optimized algorithms exist which converge to a fixed-point quickly. Unfortunately, it is not possible to directly translate the same heuristics used in the intra-modular case to the inter-modular case. In the inter-modular case we have to take into account the time required to change from analysis of one module to another since this typically means reading a new module from disk. Thus, requests to process call patterns have to be grouped by modules in order to reduce the number of times we change context.

Taking the heuristics in [PH96, HPMS00] as a starting point we have experimented with different scheduling policies which take into account different aspects of the structure of the program unit such as dependencies, strongly connected components, etc. with promising results. In the current implementation two simple strategies have been used which allow studying the behavior of the analysis of modular programs in two clearly different situations. Both strategies take the list of modules in a given order (a top-down and a bottom-up traversal of the intermodule dependency graph, respectively),[¶] and traverse the list analyzing the modules which have pending call patterns, updating the corresponding global tables with the analysis results. This process is repeated until there are no pending call patterns for any module in the program. It has also been explored which of the approaches to success policy results in more efficiently reaching a global fixed-point in combination with specific abstract domains, and whether the heuristics to be applied in either case coincide or are mostly different. The results of all those experiments can be seen in Chapter 6.

5.9 Some Practical Implementation Issues

In this section we discuss several issues not addressed in the previous sections and which are very important in order to have practical implementations of context-sensitive analysis systems. These issues are related to the persistence of global information and the analysis of libraries.

5.9.1 Making Global Information Persistent

The two-level framework presented in Section 5.6 needs to keep information both at the local and global level. One relevant question, due to its practical implications, is where this global information actually resides. One possibility is to have the global analysis tool running continuously as a kind of “compilation server” which stores the global state in its program memory. In a *manual* setting, this global tool would wait for the user(s) to place requests to analyze modules. When a request is received, the corresponding module is analyzed for the appropriate call patterns and using the global information available at the time in the memory of the global analyzer. After analysis terminates, the global information is updated and remembered by the process for subsequent requests. If we are in an *automatic* setting, the global tool itself requests the analysis of different modules until a global fixed-point (or a time-out) is reached.

This approach outlined above is not fully persistent in the sense that if the computer crashes all information about the global state is lost and analysis would

[¶]All modules which belong to the same cycle in the graph have been considered at the same depth, and therefore those modules will be selected in any order.

have to start from scratch again. In order to implement the more general kind of persistence discussed in Section 5.4, a way to save and restore the global state of analysis is needed. This requires storing the value of the three global-level data-structures: GAT, GDT, and GTQ. A level of granularity which seems appropriate in this context is clearly the module level. I.e., the global state of analysis is saved and restored between two consecutive steps of (module) analysis, but not during the analysis of a given module, which, from the point of view of the two-level framework, is an atomic operation.
The ability to save and restore the global state of analysis has several advantages:

1. The global tool does not need to be running continuously: it can save its state, stop, restart when needed, and restore the global state. This is specially interesting when using a manual scheduling policy, since two consecutive analysis requests can be separated by large intervals.

2. Even if the automatic scheduling policy is used, any information about the global state which is still valid can be directly used. This means that analysis can be *incremental* in the sense that (global level) analysis information which is known to be valid is reused.

5.9.2 Splitting Global Information

Consider the analysis of module b in the program unit $U = \{a, b, c, d, e, f, g, h\}$ depicted in Figure 5.5. In principle, the global state includes information regarding exported predicates in any of the modules in U. As a result, if we can save the global state to disk and restore it, this would involve storing and retrieving information about all modules in U. However, analysis of b only requires retrieving the information for modules in *related(m)*. The small boxes which appear on the side of every module represent the portion of the global structures related to each module. To analyze the module b, the information of the global tables that we need is that of modules a, d and e, as indicated by the dashed curved line.

This is straightforward to do in practice by splitting the information in the global data structures into several parts, each one associated to a module. This allows easily identifying the pieces of global information which are needed in order to process a given module.

This optimization of the handling of global information has several advantages:

1. The time required to save and restore the information to disk is reduced since the total amount of information transferred is smaller.

2. The use of the data structures during analysis can be more efficient since search space is reduced.

3. The total amount of memory required in order to analyze a module can be significantly reduced: only the local data structures plus a possibly very reduced part of the global data structures are actually required to analyze the module.

One question which we have intentionally left open is where the persistent information should reside. In fact, all the discussion above is independent on how and where the global state is stored, as long as it is persistent. One possibility is to use a database which stores the global state and information is grouped by modules in order to minimize the amount of information which has to be retrieved or updated for each analysis. Another, very common, possibility is to store the global information associated to each module to disk, in the same way as temporary information (such as relocatable code) is stored in many traditional compilers. In fact, the actual implementation of modular analysis in both CiaoPP and HAL [Net02] systems is based on this idea: a module m has a `m.reg` file associated to it which contains the part of the global data structures which are associated to m.

5.9.3 Handling Libraries and Predefined Modules

Many compilers and program development systems include a large number of predefined modules and libraries which can be readily reused by programmers –an obviously interesting feature since it greatly reduces the time required to develop applications. From the point of view of analysis, these predefined modules and libraries differ from user programs in a number of ways:

1. They are designed with reusability in mind and thus they can be used by a comparatively large number of user programs.

2. Sometimes the source code for libraries and predefined modules may not be available. One common reason for this is that they are implemented in a lower-level language.

3. The total amount of code available as libraries can be extremely large. Thus, reanalyzing the libraries over and over again for slightly different call patterns can be costly.

Given these characteristics, it makes sense to develop a specialized treatment for libraries. We propose the following scheme. For each library module, the analysis results for a sufficient set of call patterns should be precomputed. This set should cover all possible correct call patterns for the library. In addition, the answer pattern for those call patterns have to be an over-approximation of the actual answers, independently of whether a SP^+ or SP^- success policy is used for the programs which use such library. In addition, in order to provide more accurate information, more particular call patterns which are expected to occur often in programs which use that library module can also be included. This information is added to the GAT of the program units which use the library. Thus, the success policy will be able to use this information directly for obtaining answer patterns. The reason for requiring pre-computed answer patterns for library modules to be over-approximations is that, much in the same way as for predefined procedures, even if an automatic scheduling policy is used, library modules are (in principle) not analyzed for calling patterns other than those which are pre-computed. Note that this is conceptually equivalent to considering the interface information of library modules *read-only*, since any program using them can read this information, but no additional call patterns will be analyzed. As a result, the global level framework will ignore new call patterns to library procedures that might be generated during the analysis of user programs. More precisely, entries of the form $P : CP \mapsto AP$ in TAT such that P is a library predicate do not need to be added to the GTQ since they will not be analyzed. In addition, no entries of the form $P : CP \rightarrow H : CP'$ need be added to GDT if H is a library predicate, since the answer pattern for library predicates is never modified and thus those dependencies are useless.

Deciding which is the best set of call patterns for which a library module should be analyzed is a non-trivial problem. One possibility can be to extract call patterns from correct programs which use the library and study which are the call patterns most often used. Another possibility is to have the library developer decide which are the call patterns of interest.

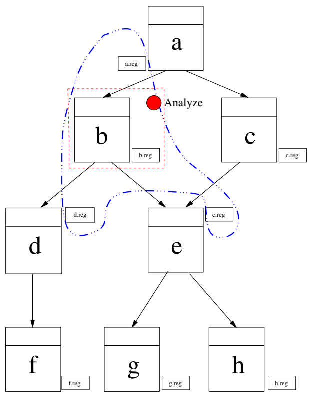


Figure 5.5: Using Distributed Scheduling and Local Data Structures

In spite of the considerations above, it is sometimes the case that we are interested in treating a library module using the general scheme, i.e., effectively considering the library information writable and allowing the analysis of new call patterns and the storage of the corresponding results. This can be interesting if the source code of a library is available and the set of initial call patterns for which it has been analyzed is not very representative. Note that hopefully this will happen often only when the library is relatively new. Once the code of the library stabilizes and a good set of initial patterns is obtained, it will generally be considered read-only. Allowing reanalysis of a library can also be useful when we are interested in using the analysis results from such call patterns to optimize the code of the library for the particular cases that correspond to those calls. For this case it may be interesting to store the corresponding information locally to the calling module, as opposed to inserting it into the library directories.

In summary, the implementation of the framework needs to treat libraries in a special way and also allow applying the general scheme for some designated library modules.

5.10 Discussion and Conclusions

Table 5.1 summarizes some characteristics of the different instantiations of the generic framework presented in this chapter, in terms of the design features discussed in Section 5.4. The corresponding entries for the flattening approach of Section 5.3 –our baseline as usual– are also provided for comparison, listed in the column labeled Flattening. The Manual column lists the characteristics of the manual scheduling policy described in Section 5.7. The last two columns correspond to the two instantiations of the automatic scheduling policy, which were presented in Sections 5.8.1 and 5.8.2 respectively. Automatic$^+$ (resp. Automatic$^-$) indicate that an over-approximating (resp. under-approximating) success policy is used.

The first three rows, i.e., Scheduling policy, Success policy, and Entry policy correspond to the values of these parameters in each instantiation.

All instances of the framework for modular analysis are *module-aware*, in contrast to Flattening, which is not. Both instances described of the modular framework proposed are incremental, in the sense that only a subset (instead of every module) in the program unit needs to be re-analyzed, and they also both achieve the goal of *not needing to reanalyze all call patterns* every time a module is considered for analysis.

Regarding correctness, both the Flattening and Automatic$^-$ approaches have in common that correctness is only guaranteed when analysis comes to an end. This is because the approximations used are under-approximations and thus the results are only guaranteed to be correct when a (global) fixed-point is reached. However, in the Manual and Automatic$^+$ approaches the information in the global state is correct after any number of local analysis steps.

On the other hand, both the Flattening and Automatic$^-$ approaches are guaranteed to obtain the most accurate information possible, i.e., the least analysis graph, when a fixed-point is reached. In contrast, the Manual approach cannot guarantee optimal accuracy for two reasons. The first one is that there is no guarantee that modules will be processed the number of times that is necessary for an inter-modular fixed-point to be reached. Second, even if such a fixed-point is reached, it may not be the least fixed-point. This is because this approach uses over-approximations of the analysis results which are improved ("narrowed") in the different analysis iterations until a fixed-point is reached. On the other hand, if there are no circular dependencies among predicates in different modules, then the fixed-point obtained will be the least one, i.e., the most accurate.

Regarding efficiency in time we will consider two cases. The first one is when we have to perform analysis of the program unit from scratch. In this case, Flattening can be highly optimized in order to converge quickly to a fixed-point.

In contrast, in this situation the instances of the modular framework have the disadvantage that loading and unloading modules during analysis introduces a significant overhead. As a result, in order to maintain the number of context changes low, call patterns may be solicited from imported modules which use temporary information and which are not needed in the final analysis graph. These call patterns which end up being useless are known as *spurious* versions. This problem also occurs in Flattening, though to a much lesser degree if good algorithms are used. Therefore, the modular approaches may end up performing work which is speculative, and thus the total amount of work performed in the automatic approaches to modular analysis is in principle an upper bound of that needed in Flattening.

On the other hand, consider the second case in which a relatively large amount of intra-modular analysis has already taken place for the modules to be analyzed in our programming unit and that the global information is persistent. In this case, the automatic approaches can update their global data structures using the precomputed information, rather than starting from scratch as is done in Flattening. In such a case the automatic approaches may perform much less work than Flattening. It is to be expected that once module m becomes stable, i.e., it is fully developed, it will quickly be analyzed for a relatively large set of calling patterns. In such a case it is likely that it will be possible to analyze any other module m' which uses m by simply reusing the existing analysis results for m. This is specially true in the case of *library modules*, as discussed in Section 5.9.3.

Regarding the efficiency in terms of memory, it is to be expected that the instances of the modular framework will outperform the non-modular, flattening approach. This was in fact already observed in the case of [BdlBH$^+$01]. Indeed, one important practical difficulty that appears during the (monolithic) analysis of large programs is that the amount of information which is kept in memory is very large and the storage needed can become too large to fit in memory. The modular framework proposed needs less memory because: a) at each point in time, only one module requires to be loaded in the code area, and b) the local answer table only needs to hold entries for the module being analyzed, and not for other modules. Also, in general, the total amount of memory required to store the global data structures is not very high when compared to the memory required locally for the different modules. In addition, not all the global data

Table 5.1: Comparison of Approaches to Modular Analysis

	Flattening	Manual	Automatic$^+$	Automatic$^-$
Scheduling policy	automatic	manual	automatic	automatic
Success policy	SP^-	SP^+	SP^+	SP^-
Entry policy	top-level	all	top-level	top-level
Module-aware	no	yes	yes	yes
No Rean. of all CPs	no	n/a	yes	yes
Correct	at fixed-point	yes	yes	at fixed-point
Accurate	yes	no	no circ.	yes
Efficient in time	yes	n/a	no	no
Efficient in memory	no	yes	yes	yes
Termination	finite asc. chains	finite asc. chains	finite chains	finite asc. chains

structures are required when analyzing a module m, but only that associated with the modules in *related*(m).

Finally, regarding termination, except for Flattening, in which only one level of termination is required, the three other cases require two levels of termination: at the intra-modular and at the inter-modular level. In Flattening, since analysis results increase monotonically until a fixed-point is reached, termination is often guaranteed by considering description domains which do not contain infinite ascending chains: no matter what the current description is, top ($\top$), which is trivially guaranteed to be a fixed-point, is only a finite number of steps away. Exactly the same condition is required for guaranteeing termination of Automatic$^-$. The manual approach only requires guaranteeing intra-modular termination since the number of call patterns analyzed is finite. However, in the case Automatic$^+$, finite ascending chains are required for ensuring local termination *and* finite descending chains are required for ensuring global termination. As a result, termination requires domains with finite chains, or appropriate widening operators.

In summary, the proposed two-level generic framework for analysis and its instantiations meet a good subset of our stated objectives. We hope the discussion and the concrete proposal presented in this work will provide a better understanding of the handling of context-sensitive program analysis on modular programs and contribute to the widespread use of such context-sensitive analysis techniques for modular programs in practical systems. An implementation of the framework, as a generalization of the pre-existing CiaoPP modular analysis components, is currently being completed. In this context, we are experimenting with different scheduling policies for the global level, for concrete, practical analysis situations.

Chapter 6

Experimental Evaluation of the Intermodular Analysis Algorithm

6.1 Introduction and Motivation

Several models for context-sensitive analysis of modular programs have been proposed, each with different characteristics and representing different trade-offs. The advantage of these context-sensitive analyses is that they provide information which is potentially more accurate than that provided by context-free analyses. Such information can then be applied to validating/debugging the program and/or to specializing the program in order to obtain important performance improvements. Some preliminary experimental results have also been reported for some of these models, providing some initial evidence on their potential. However, further experimentation, needed in order to understand the many issues left open and to show that the proposed modes scale and are usable in the context of large, real-life modular programs, was left as future work. The aim of this chapter is twofold. On one hand we provide an empirical comparison of the different models proposed in Chapter 5 and in [BdlBH$^+$01], as well as experimental data on the different choices left open in those designs. Our second aim is to explore the scalability of these models and of the implementation. To this end we have used some larger modular programs as benchmarks, including some real-life examples such as a working partial evaluator and parts of the Ciao compiler. To this end we have completed a full implementation in CiaoPP [HPBLG05] (the preprocessor of the Ciao system [BCC+04]) of the framework for context-sensitive analysis described in [PCH+04] and its different instances, and we have studied experimentally the behavior of the resulting system. These results have been compared with traditional, non modular analyses in terms of time and memory consumption. Our experimental results shed light on the practical implications of the different design choices and of the models themselves. We also show that context-sensitive analysis of modular programs is indeed feasible in practice, and that in certain critical cases it provides better performance results than those achievable by analyzing the whole program at once. This is specially the case regarding memory consumption and when reanalyzing after making changes to a program, as is often the case during program development.

Following Section describes the tests performed and analyzes the results obtained. Section 8.6 presents our conclusions.

6.2 Empirical results

As mentioned above, the framework has been fully implemented in CiaoPP. This implementation allows performing both monolithic and modular analysis, and the modular analysis is parametric in several ways. This makes it possible to study the overall behavior of the system for different strategies and policies and thus performing several experiments and comparisons:

Flattened vs. modular First, the flattened approach of Section 5.3.1 has been compared to the intermodular fixpoint of Section 5.8.4. Although it is predictable that the analysis of a program for the first time in a modular, separate analysis fashion will be slower than the flattened approach (due to the overhead in loading/unloading modules, etc.), it is interesting to study by how much. On the other hand, in some cases the analysis of a whole program may be unfeasible due to hardware (memory) limitations, but in the intermodular fixpoint approach this limitation can be overcome.

Intermodular scheduling policies Another aspect to study is related to the influence of the module selection policy in the efficiency of the analysis. The scheduling policies used have been already described in Section 5.6. We will refer to them as *naive_top_down* and *naive_bottom_up*, respectively.

Success policies Two success policies have been compared in both scheduling policies: an over-approximating policy, SP^{+}_{all}, and an under-approximating one, SP^{-}_{all}, as described in Section 5.5. Although there may be other success policies, we estimate that these ones are the most effective policies, as they bring the closest results to $SP^{=}$.

Incremental analysis of modular programs Finally, the analysis of a modular program from scratch using the monolithic approach has been compared to the reanalysis of that program after making specific modifications in the source code. This comparison illustrates the advantages of analyzing only the module which has changed (and the modules affected by that change) instead of reanalyzing the whole program from scratch.

Three different kinds of source code modifications have been studied: 1) a simple change that keeps the same analysis results, 2) a change that results in the exported predicates producing a more precise answer pattern, and 3) a modification in the source code such that after the change exported predicates produce more general analysis results.

Note that when there are changes in the source code which do not improve or invalidate previous analysis results, nor generate new call patterns for imported modules (i.e., case 1 above), using the modular approach is clearly advantageous (at least theoretically), since it is more incremental and only one module needs to be analyzed after each change. In contrast, in the monolithic (non-modular) analysis the whole program must be (re)analyzed.

The second kind of change studied represents a change that makes the analysis results for exported predicates be more precise than the ones obtained before. This is done by removing all clauses of exported predicates of a module except the first non recursive one.* This will bring in general analysis results which are more specific than the results previously obtained, making them invalid in most cases, and producing the reanalysis of the calling modules.

The third type of source change corresponds to performing a modification

*Mutually recursive predicates are also considered. If the exported predicate has only recursive clauses, they are replaced by a fact with all arguments ground.

in an exported predicate which results in this predicate providing more general analysis results. The change consists in the addition of a clause to all the exported predicates of a module in which all arguments are pairwise distinct and free variables.[†] This approach generally forces the reanalysis of the modules which use the changed module. In turn, this may transitively require reanalysis of other modules until analysis information stabilizes.

In the following subsections the selected benchmark programs are described, and the results of the tests are studied in detail. Two "modes" domains have been considered: *Def* [dlBH93], which keeps track of properties (in particular, groundness) through definite propositional implications and *Sharing-freeness* [MH91], which keeps track of information on variable sharing and freeness in a combined way.

6.2.1 Brief description of the benchmarks used

The central focus of this chapter is to study how the intermodular analysis framework of CiaoPP will behave with real-life programs. Therefore, we have striven in the selection of benchmark programs to include not only characteristic examples used in the LP analysis literature, but also other programs which are specially difficult to analyze in a modular setting (for example, because there are several mutually recursive predicates which conform intermodular cycles), and real-life programs. A brief description of the selected benchmarks follows:

ann This is the &-Prolog implementation of the MEL annotator (by K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo). In this case the code is distributed in 3 modules with no cycles in the intermodular dependency graph.

bid This program computes an opening bid for a bridge hand (by J. Conery). It is composed of 7 modules, with no cycles in the intermodular dependency graph.

[†]In the $Sharing-Freeness$ domain this addition might not provide a more general analysis result, as this kind of clause does not provide a top success substitution. However, the tests have been performed using the same change also in the case of $Sharing - Freeness$ to make the tests homogeneous across the different domains.

boyer The boyer benchmark is a reduced version of the Boyer/Moore theorem prover (by E. Tick). The program has been separated in four modules with a cycle between two modules.

peephole This program is the SB-Prolog peephole optimizer. In this case, the program is split in three modules, but there are two cycles in the intermodular dependency graph, and there are several intermodular cycles at the predicate call level.

prolog_read corresponds to a simplified version of the code used by the Ciao compiler for reading terms. It is composed by three modules, having a cycle between two of them.

unfold_ is a fragment of the CiaoPP preprocessor which contains the partial evaluator. It is distributed in 7 modules with no cycles between them, although many other modules of CiaoPP source code, while not analyzed, are consulted in order to get assertion information.

managing_proj is a program used by the authors for EU project management. It is distributed in 8 modules with no intermodular cycles.

check_links is an example program for the *Pillow* HTML/XML/HTTP connectivity package (by D. Cabeza and M. Hermenegildo) that checks that links contained in a given URL address are reachable. The whole Pillow package is analyzed together with the sample program, and it is composed of 6 modules without intermodular cycles.

It should be noted that for all these programs the number of modules indicated above correspond to the user modules of the benchmark. However, they are not the only ones processed: any benchmark is likely to use quite a large number of modules from the system libraries. In particular, in Ciao all builtins are in system libraries. For efficiency, library modules are pre-analyzed for a representative set of call patterns and the analysis results are expressed using the assertion language described in [PBH00b]. Instead of analysing library modules over and over again, the analysis algorithm computes success information from such assertions using a SP^+ policy.

Def					
Bench	**Mod**	**Load**	**Ana.**	**Gen.**	**Total**
ann	3	387	343	170	1151
bid	8	631	35	182	1177
boyer	4	385	161	100	871
peephole	3	350	205	175	907
prolog_read	3	343	279	370	1179
unfold_	7	1622	540	117	2744
managing_proj	8	1154	6128	302	8025
check_links	6	1492	3720	365	6002
Sharing-freeness					
Bench	**Mod**	**Load**	**Ana.**	**Gen.**	**Total**
ann	3	387	480	217	1513
bid	8	631	50	192	1400
boyer	4	385	181	102	1098
peephole	3	350	542	305	1643
prolog_read	3	343	3112	633	4490
unfold_	7	1622	521069	286	523692
managing_proj	8	1154	781	256	2911
check_links	6	1492	4044	484	6706

Table 6.1: Time spent (in milliseconds) by the monolithic analysis of different benchmark programs

The benchmarks have been run on a Dell PowerEdge 4600 with two Pentium processors at 2 Ghz and 4 Gb of memory, and normal workload. Each test has been run twice, reporting the arithmetic mean of these runs.

6.2.2 Analysis of a modular program from scratch

Table 7.3 shows the absolute times in milliseconds spent in analyzing the programs using the flattening approach. **Mod** reflects the number of modules comprising each benchmark (excluding system modules). For every benchmark, the total analysis time is divided into several categories, represented by the following columns:

Type of test		automatic SP^+_{all}				automatic SP^-_{all}			
		Load	**Ana.**	**Gen.**	**Total**	**Load**	**Ana.**	**Gen.**	**Total**
Def	**top_down**	2.25	1.30	1.44	1.47	3.64	1.60	2.39	2.08
	bot_up	2.26	1.25	1.44	1.45	3.80	1.66	2.47	2.16
Shfr	**top_down**	2.23	20.51	1.83	11.60	3.85	3.13	2.65	3.51
	bot_up	2.23	20.02	1.68	11.43	3.96	3.14	2.75	3.53

Table 6.2: Geometric overall results for analysis of modular programs from scratch using different global scheduling algorithms and success policies. Numbers relative to the monolithic approach.

Load This column corresponds to the time spent loading modules into CiaoPP. This time includes the time used for reading the module to be analyzed and the time spent in reading the assertions of the imported modules.

Ana. This is the time spent analyzing the program and applying the success policy for imported predicates together with some preprocessing of the code.

Gen. Corresponds to the task of generating the global information (referred to before as the GAT and TAT tables). The information generated is related to the analysis results of all exported and multifile predicates, new call patterns of imported predicates generated during the analysis of each module, and the modules that import the module and can improve their analysis results by reanalysis.

Total Time elapsed since the analyzer is called until it finishes completely. It is the sum of the previous columns, plus some extra time spent in other tasks, such as the generation of the intermodular dependency graph, handling the list of modules to get the next module to be analyzed, etc.

Table 6.2 gives the summary of the weighted geometric means of the comparative times for all benchmarks for the Def and *Sharing-freeness* analysis domains. The numbers in this table are relative to the monolithic case (shown in Table 7.3), and the number of clauses of each program is used as weight for each benchmark when computing the weighted geometric mean. The *naive_bottom_up* and *naive_top_down* global scheduling policies are compared, as well as the SP^-_{all} and SP^+_{all} success policies. Table columns have the same meaning as before.

This table shows the overall time spent in the analysis of the different benchmarks without previous analysis information. It is clear that the modular analysis from scratch, in general, is slower than monolithic analysis, as expected. Using Def the intermodular analysis from scratch is only somewhat slower compared to the monolithic analysis, and in particular the analysis time is not much larger than the monolithic time in most cases. However, in simple domains like Def, the analysis time is not the most important fraction of the total time, and therefore other tasks such as module loading or results generation can in fact be more relevant than the analysis itself. On the other hand, more complex domains as $Sharing-freeness$ increase the difference with respect to the monolithic case. It is important to note that using SP^+_{all} is clearly not recommended for performing modular analysis from scratch in the $Sharing - freeness$ domain. The result in this case is biased a great deal by the results of the analysis of `managing_proj`, in which most predicates have many arguments, resulting in large sharing sets that tend to approximate to $\top$ (which is the powerset of the variables in the clause). However, SP^-_{all} produces reasonable results.

On the other hand, when comparing the global scheduling policies, only a slight difference in the time taken using the *naive_top_down* or the *naive_bottom_up* strategies can be observed. This result seems to reflect that the order of the modules is not so relevant when analyzing a modular program as was initially expected.

Memory Consumption when analyzing from scratch. We have also compared the maximum memory required for the analysis in the flattened and the modular approaches to the analysis of modular programs from scratch. Table 6.3 shows the maximum memory consumption during the analysis of the flattened approach (column **Monolithic**), and the use of memory of the modular approach (using both global scheduling policies described before) relative to the monolithic case (columns SP^+_{all} and SP^-_{all} for the corresponding success policies). The results show that the modular approach is clearly better in terms of maximum memory consumption than the monolithic approach, except for the outlying result of `managing_proj` for the particular case of the combination SP^+_{all} and $Sharing - freeness$, as mentioned above. However, given a program split into N modules, the memory used for analyzing it in a modular way might be

Global scheduling policy: naive_top_down							
		Def			***Sharing-Freeness***		
Bench	**Mod**	**Monolith.**	SP^+	SP^-	**Monolith.**	SP^+	SP^-
ann	3	2825160	0.69	0.49	4070806	0.54	0.39
bid	8	2201134	0.54	0.54	3241842	0.36	0.36
boyer	4	2405980	0.42	0.42	3495038	0.61	0.35
peephole	3	2390936	0.68	0.68	3761442	0.42	0.43
prolog_read	3	2766260	0.53	0.51	5429194	0.84	0.84
unfold_	7	5775798	0.54	0.54	16168722	0.31	0.37
managing_proj	8	5555454	0.32	0.32	6565038	3.65	0.26
check_links	6	10431662	0.70	0.65	18643226	0.83	0.77
Weighted Geom. mean			0.48	0.46		1.12	0.40

Global scheduling policy: naive_bottom_up							
		Def			***Sharing-Freeness***		
Bench	**Mod**	**Monolith.**	SP^+	SP^-	**Monolith.**	SP^+	SP^-
ann	3	2825160	0.52	0.49	4070806	0.54	0.39
bid	8	2201134	0.57	0.54	3241842	0.36	0.36
boyer	4	2405980	0.42	0.42	3495038	0.61	0.40
peephole	3	2390936	0.68	0.68	3761442	0.42	0.43
prolog_read	3	2766260	0.53	0.51	5429194	0.84	0.84
unfold_	7	5775798	0.54	0.54	16168722	0.31	0.37
managing_proj	8	5555454	0.33	0.33	6565038	3.65	0.28
check_links	6	10431662	0.69	0.66	18643226	0.81	0.78
Weighted Geom. mean			0.47	0.47		1.11	0.41

Table 6.3: Overall memory consumption of Non-modular vs. SP^+ and SP^- policies.

expected to be M/N, where M is the memory required for the monolithic analysis. This is not true because the complexity of the program is in general not evenly distributed among its modules. Since Table 6.3 shows maximum memory consumption, figures are strongly influenced by the most complex modules.

6.2.3 Reanalysis of a modular program after a change in the code

As explained at the beginning of Section 6.2, we have also studied the incremental cost of reanalysis of a modular program after a change, for different typical changes, as explained above.

In the first case, shown in Table 6.4, a simple change in a module with no implications in the analysis results of that module has been tested. It has been implemented by "touching" a module, i.e., changing the modification time without actually modifying its contents, in order to force CiaoPP to reanalyze it. As before, numbers refer to the geometric overall results, relative to those obtained with the monolithic approach (Table 7.3). As it is suggested in the results shown in Table 6.4, the modular analysis is clearly better than the monolithic approach for this kind of change. Obviously, global scheduling and success policies are not relevant, since only the module which has been modified is reanalyzed.

The second case (summarized in Table 6.5) corresponds to a source code modification in which, as already mentioned, all the clauses of the exported predicates of a given module have been replaced by the first non-recursive clause of the predicate. As in the previous case, different policies do not seem to be very relevant for this change. It is interesting to note that this kind of change is even more efficient than just touching a module: since some part of the code is being removed, the analysis tends to be simplified (specially the recursive clauses, which cause more iterations of the fixed-point computation algorithm).

And, finally, the third case shown in Table 6.6 is implemented by adding a most general fact to all exported predicates of a given module. Like in the previous case, this kind of change is an extreme situation in which all exported predicates are affected. Even in this case modular analysis is more efficient than the monolithic approach. With respect to the differences between the success policies, the SP^{-} policy is slightly more efficient in complex domains such as $Sharing - freeness$, although both policies and domains behave incrementally. On the other hand, the bottom-up global scheduling policy produces better results than top-down scheduling.

The overall results in Tables 6.4,6.5, and 6.6 indicate that in many cases the reanalysis time is much better than in the monolithic case. It is important

		automatic SP^{+}_{all}				automatic SP^{-}_{all}			
Type of test		**Load**	**Ana.**	**Gen.**	**Total**	**Load**	**Ana.**	**Gen.**	**Total**
Def	**top_down**	0.68	0.39	0.19	0.44	0.63	0.38	0.19	0.43
	bot_up	0.66	0.41	0.20	0.45	0.65	0.38	0.19	0.43
Shfr	**top_down**	0.67	0.53	0.26	0.44	0.65	0.40	0.25	0.40
	bot_up	0.65	0.52	0.28	0.43	0.67	0.41	0.26	0.40

Table 6.4: Geometric overall results for reanalysis of modular programs after touching a module, using different global scheduling algorithms and success policies. Numbers relative to the monolithic approach.

		automatic SP^{+}_{all}				automatic SP^{-}_{all}			
Type of test		**Load**	**Ana.**	**Gen.**	**Total**	**Load**	**Ana.**	**Gen.**	**Total**
Def	**top_down**	0.97	0.18	0.33	0.45	0.99	0.20	0.33	0.46
	bot_up	0.97	0.18	0.32	0.45	1.00	0.20	0.34	0.46
Shfr	**top_down**	1.00	0.36	0.41	0.49	0.94	0.26	0.33	0.44
	bot_up	0.97	0.33	0.39	0.47	0.98	0.27	0.33	0.46

Table 6.5: Geometric overall results for reanalysis of modular programs after removing all clauses of exported predicates of a module except the first non-recursive one, using different global scheduling algorithms and success policies. Numbers relative to the monolithic approach.

		automatic SP^{+}_{all}				automatic SP^{-}_{all}			
Type of test		**Load**	**Ana.**	**Gen.**	**Total**	**Load**	**Ana.**	**Gen.**	**Total**
Def	**top_down**	1.09	0.63	0.43	0.70	1.05	0.59	0.42	0.66
	bot_up	1.02	0.58	0.40	0.64	1.04	0.60	0.43	0.66
Shfr	**top_down**	1.18	1.00	0.69	0.80	1.27	1.00	0.69	0.86
	bot_up	1.14	0.97	0.67	0.77	1.21	0.98	0.70	0.83

Table 6.6: Geometric overall results for reanalysis of modular programs after adding a most general fact to all exported predicates of a module, using different global scheduling algorithms and success policies. Numbers relative to the monolithic approach.

to note that the analysis domain used is very relevant to the efficiency of the modular approach: the analysis of a complete program in complex domains such as $Sharing - freeness$ is much more expensive than the reanalysis of a module, while the difference is smaller (although still significant) in the case of Def. This suggests that modular analysis can make it practical to use domains which are precise but rather costly. On the other hand, the results in Table 6.6 for reanalysing after a more general change using $Sharing - freeness$ are very close to monolithic analysis from scratch, although still below it. That means that even in the presence of the most agressive change in a module, modular analysis is not more time-consuming than analyzing from scratch. Simpler changes provide better results of the modular analysis with respect to the flattened approach, as is shown in Tables 6.4 and 6.5 for other kinds of changes.

6.3 Conclusions

We have presented an empirical study of several proposed models for context-sensitive analysis of modular programs, with the objective of providing experimental evidence on the scalability of these models and, specially, on the impact on performance of the different choices left open in those models.

Our results shed some light on the different choices available. In the case of analyzing a modular program from scratch, the modular analysis approach has been shown, as expected, to be slower than the flattening approach (i.e., having the complete program in memory, and analyzing it as a whole), due to the cost in time of loading and unloading code and related analysis information, and the restriction of not being able to analyze predicates in modules other than the one being processed. However, the modular analysis times from scratch are still reasonable, excluding the case of the $Sharing - freeness$ domain with SP^+_{all} success policy. In addition, our results also provide evidence that modular analysis does imply a lower maximum memory consumption which in some cases may be of advantage since it may allow analyzing programs of a certain critical size that would not fit in memory using the flattening approach.

Across the domains we can see that in simple domains SP^+_{all} and a naive bottom up scheduling policy appear to be the best. These strategies appear substantially better for some experiments (in particular, for more general changes) and not much worse than others on most experiments. Another conclusion which can be derived from our experiments is that, as already mentioned, no really significant difference has been observed between the top-down and bottom-up strategies.

We have also considered the case of reanalyzing a previously analyzed program, after making changes to it. This is relevant because it represents the standard situation during program development in which some modules change while others (and the libraries) remain unchanged. While in this phase the analysis results may not be needed in order to obtain highly optimized programs, they are indeed required for other important steps during development, such as static program debugging and validation. In this context our results show that modular analysis, because of its more incremental nature, can offer clear advantages in both time and memory consumption over the monolithic approach.

Chapter 7

A Practical Type Analysis for Verification of Modular Prolog Programs

Regular types are a powerful tool for computing very precise descriptive types for logic programs. However, in the context of real-life, modular Prolog programs, the accurate results obtained by regular types often come at the price of efficiency. In this chapter we propose a combination of techniques aimed at improving analysis efficiency in this context. As a first technique we allow optionally reducing the accuracy of inferred types by using only the types defined by the user or present in the libraries. We claim that, for the purpose of verifying type signatures given in the form of assertions the precision obtained using this approach is sufficient, and show that analysis times can be reduced significantly. Our second technique is aimed at dealing with situations where we would like to limit the amount of reanalysis performed, especially for library modules. Borrowing some ideas from polymorphic type systems, we show how to solve the problem by admitting parameters in type specifications. This allows us to compose new call patterns with some precomputed analysis info without losing any information. We argue that together these two techniques contribute to the practical and scalable analysis and verification of types in Prolog programs.

7.1 Introduction

Types are widely recognized as being useful for several purposes, which include early detection, i.e., at compile-time, of certain programming errors, enforcement of disciplined programming, and documentation of code. In the terminology of [Pie02], Pure Logic Programming is a *safe* programming language in that the semantics of programs is well-defined and the execution of a program does not depend on the particular compiler used. This is achieved, without the need for types, thanks to the declarative nature of Pure Logic Programming, which is *untyped*. However, as soon as we introduce predefined operations in the programming language, for example arithmetic, certain type checks are required in order to preserve the safety of the programming language. As a result, Prolog, which is the most widely used logic programming language, is no longer an untyped programming language, but rather it is a *dynamically checked* typed language: In order to preserve language safety, calls to predefined operations which do not satisfy their calling conventions result in run-time errors or exceptions. However, some of the desirable features of types mentioned above in fact only apply to *statically checked* typed languages.

A clear possibility in order to obtain a statically checked typed logic programming language is to design a new programming language from scratch with static checking in mind. Two proposals along these lines are Gödel [HL94b] and Mercury [SHC96]. In these languages, types (called *prescriptive types*) are a part of the language itself (both the syntax and semantics). In spite of the undoubtful contribution of these proposals, Gödel is no longer maintained and Mercury, while certainly interesting in many ways, deviates in a number of respects from logic programming. In practice, Prolog (including its different extensions) remains by far the most widely used logic programming language.

Another possibility, which is the one we will focus on in this work, is to provide a mechanism for performing static checking of types directly for Prolog. We believe that this will have more practical impact than designing yet another strongly typed logic programming language. Several proposals have been made and implemented in the direction of augmenting Prolog with static checking, such as, e.g., Ciao [HPBLG05, BCC$^+$06]. In this context, even though the language itself is not statically typed, it is possible to infer static information about the program in terms of *regular types*. The types inferred are called *descriptive types* in that they describe the program behavior, but they do not provide *directly* any assurance about the nonexistence of run-time errors since we can obtain descriptive types for any program. In Ciao, users can *optionally* provide type definitions and assign types to predicate arguments and thus describe the expected behavior of the program [HPBLG05]. Success of static checking occurs if the descriptive types inferred imply the type declarations provided by the user and those present in system libraries. Alternatively, type-related errors may be detected statically.

Abstract interpretation-based type analysis using regular types is a powerful technique for computing very precise descriptive types for logic programs in general and for Prolog in particular. In Ciao, a multi-variant, context sensitive analysis engine [HPMS00] is used which is parametric w.r.t. the abstract domain of interest and which can analyze [CPHB06, PCH$^+$04] and check [PCPH06] modular programs Unfortunately, in this setting, the analysis of real-life, modular Prolog programs, using regular types turns out to be too expensive in practice. The abstract domain of regular types is infinite and in order to guarantee termination of the analysis process a widening operator is required. Such operators may, in some cases, be quite sophisticated procedures (cf. [Mil99]). It is this ability of the widening-based analyses to create new types that brings the precise results, but at the same time the presence of a large number of very detailed types inevitably affects analysis performance.

In this chapter we propose a combination of techniques aimed at improving analysis efficiency in this context while preserving a reasonable accuracy. The techniques proposed are implemented as extensions of the generic analyzer in the Ciao Preprocessor, CiaoPP [HPMS00, HPBLG05], with the type domains of [JB92, VB02]. As a first technique we allow *optionally* reducing the accuracy of inferred types by using only the types defined by the user or present in the libraries. In every iteration our analysis replaces the inferred types with such types. We will show that in this way we ensure faster convergence to the fixed point and that analysis times can indeed be reduced significantly. Also, we claim that, for the purpose of verifying type signatures given in the form of assertions, the precision obtained using this approach is adequate.

Our second technique is aimed at dealing with situations where we would like to limit the amount of reanalysis performed for library modules while increasing precision. To this end we allow using parametric type assertions in the specification. Such assertions are specially useful in libraries implementing generic data manipulation predicates (like, e.g., lists or AVL-trees) which we do not want to have to reanalyze every time we analyze a program that uses the library. In this case we can instantiate parameters in the trusted assertion (now playing the role of module interface) according to the actual call pattern, and simply reuse the resulting success pattern without analyzing the library module. In this way we incorporate some specific characteristics of polymorphic type systems for logic programming [MO84, HL94b], without changing the source language and while remaining in descriptive types, i.e., types which describe approximations of the program semantics.

The main application of our analysis is in verification of programs with respect to a partial specification written in the form of a number of type assertions and type definitions (see [PBH00c, PCPH06]). Note that any assertion present in the program must refer to types which *are* defined in user or library modules anyway, and therefore in this case we may not lose many opportunities for verifying assertions.

7.2 Related work

The issue of introducing type systems for static checking of types in logic programming dates back to the early papers of Mishra [Mis84] and Mycroft–O'Keefe [MO84]. There has been a number of proposals since then for providing adequate notions of types and typing (see for example [Pfe92]).

As mentioned before, our work follows the *descriptive typing* approach in which types approximate the program semantics. This idea was first presented in [Mis84], where types are described by *regular term grammars* that reappeared in the literature in one formalism or another.

Deriving descriptive types from a program (this process is also called "type inference" or "type analysis"), essentially means finding, at compile-time, an approximate description (in our context, a safe approximation) of the values that program variables can take at run-time. Descriptive types can be inferred that approximate various semantics of a logic program. In [HJ90] descriptive types are computed using *set constraints* analysis. Their types approximate the declarative semantics of programs. Another approach to approximate the declarative semantics is to construct an abstract counterpart of the *immediate consequences operator*, T_P, (see e.g. [Llo87]) in order to obtain a superset of the success set of the program. An example of this approach is [GdW94], in which regular descriptive types are called "regular approximations"). Also, the T_P operator is approximated in [YS90], but with the goal of verifying a program w.r.t. given success types, rather than inferring the types. The relative power of different regular approximations of T_P is discussed in [HJ92].

In other approaches, descriptive types approximate operational semantics, following a top-down execution strategy with the Prolog selection rule. This allows distinguishing call and success types, which makes it feasible to verify call patterns for certain predicates. Examples of this line are [JB92, VHCL95, VB02]. In our approach we also deal with operational semantics. As already mentioned, we use a generic, context-sensitive, multi-variant analysis framework [HPMS00, Bru91, MH92, dlBHB$^+$96a] based on *abstract interpretation* [CC77b] and specialized to our type domain.

In the above mentioned approaches, types are constructed on the fly during the iterative analysis process over an abstract domain of types which is infinite. Therefore a *widening* operator is introduced, to ensure that no infinite ascending chain is generated during computing the fixed point, and thus that the computation terminates. For a comprehensive study of different widening operators see [Mil99].

In our approach, also new type definitions are generated on-the-fly. However, as soon as they are generated, the analyzer tries to replace them by picking a type from a predefined collection of definitions. These definitions correspond to the types which have been defined by the user or which are present in library modules used by the program. This type replacement has to be correct –we always replace types with super-types– and accurate –we never lose more precision that strictly required. Therefore, the type definitions present in the abstract descriptions at each iteration step originate from a finite set. This guarantees termination without the need for a widening operator.

Another alternative to the widening option is to generate a type domain which is specific to a given program. An example of such an approach is [GP02], which proposes an abstract interpretation over non-deterministic tree automata. The authors exploit the observation, due to [CC95], that for a particular program one may automatically build a finite domain of tree automata (or regular tree grammars), and thus make sure that the analysis (a fixed point iteration) terminates. Another approach that allows constructing program-specific type domains is proposed in [GH04]. The constructed domain incorporates also instantiation information and it is *condensing*, i.e., combining the result of bottom-up analysis with an initial goal pattern is as precise as the output of analyzing the program top-down in the goal-dependent fashion, for the same initial goal pattern. The fact that the domain is condensing is attractive for intermodular analysis as it enables fully compositional approach. Nevertheless, in our view, a disadvantage of these approaches w.r.t. considering the defined types is that since types are automatically generated the resulting types are not intuitive and are hard to understand.

Our approach is strongly related to other work in which types for each function symbol are defined prior to the analysis, and analysis itself infers types for predicates. In these analyses, the output shown to the user contains familiar types and, thus, it is easy to interpret. Among other papers, [Lu95] follows this line, and shows an analysis method that combines types with sharing and aliasing. A rather complex polymorphic analysis is presented in [Lu98]. In contrast, as our main concern is simplicity and efficiency, we do not infer polymorphic types, even if we do make use of parametric type rules for describing module interfaces (see Section 7.5). In some of this work, like [BG92], only a well-typed part of the program semantics (a success set in this case) is described by the analysis output. In this sense [BG92] is a prescriptive typing approach. Types for function symbols are also required by [CL00], where an elegant theory of ACI-unification (associative, commutative and idempotent) is used to infer (polymorphic) type information from the program (abstractly compiled before the analysis). The resulting domain is condensing. A technique which for given type definitions infers a combination of prescriptive and descriptive types is given in [SG95]. In all the above work, typing rules for function symbols are given prior to the analysis. In some cases, (like for example [CL00]) the rules are quite restrictive and require that each function symbol is of exactly one type. In our work, predefined types are used differently. There is no notion of a type signature for function symbols. Instead, during the analysis the inferred (descriptive) types are inspected and replaced by predefined types that match them as precisely as possible.

There are also some similarities between our work and strongly typed logic languages such as Gödel [HL94b] or Mercury [SHC96] (whose type systems are based on [MO84]), especially as regards the usage of parametric rules. However, in both Gödel and Mercury the programmer is required to write, together with the code, the types, both for function symbols and for predicates. Moreover, subtyping is often not permitted. In contrast, in our setting writing type definitions is optional and subtyping is allowed.

Our work is thus unique in combining both the flexibility of descriptive typing approaches, where type definitions are optional and have clear semantics, with some features of prescriptive types, where the output of analysis is presented in terms of types known to the user, and parametric type rules are allowed.

7.3 Preliminaries

7.3.1 Regular types domain

Assume a finite set $\mathcal{F}$ of ranked function symbols. Let $Term(\mathcal{F}, \mathcal{V})$ denote a set of terms built from function symbols $\mathcal{F}$ and variables $\mathcal{V}$. A *regular term grammar* is a tuple $G = \langle \mathcal{T}, \mathcal{F}, R \rangle$, where:

- $\mathcal{T}$ is a set of non-terminal symbols (constants), called here *type symbols*,
- R is a set of rules of the form $l \rightarrow r$, where $l \in \mathcal{T}$, $r = f(T_1, \ldots, T_n)$ $(f/n \in \mathcal{F}, T_i \in \mathcal{T})$.

We use the notation $t_1 \Rightarrow_G t_2$ (or $t_1 \Rightarrow t_2$ if G is clear from the context) to denote the usual derivability relation, i.e. if t_2 is obtained from t_1 by replacing t (where t is a subterm of t_1) by a term r where $t \rightarrow r \in R$. $\overset{*}{\Rightarrow}$ denotes the transitive and reflexive closure of $\Rightarrow$.

A type symbol T defined in grammar G denotes a (regular) set of ground terms $\mathit{Type}_G(T) = \{t \in Term(\mathcal{F}, \emptyset) \mid T \overset{*}{\Rightarrow} t\}$. As before, we drop the subscript G if it is clear from the context. In order to describe sets of numbers or Prolog atoms, we introduce *b*ase types and corresponding base type symbols, like *int*, *num*, *atm*, etc., denoting respectively sets of integers, all numbers, Prolog atoms, etc. The base types can be seen as defined by a set of rules, fixed for a fixed signature, with constants in the right hand sides. Moreover, we introduce the "top" type symbol $\top$, s.t. *Type*$(\top) = Term(\mathcal{F}, \emptyset)$ (i.e. $\top$ denotes the set of all ground terms) and "bottom" type symbol $\perp$, s.t. *Type*$(\perp) = \emptyset$.

The regular type domain (e.g., [DZ92]) is equipped with standard operations that satisfy the corresponding properties:

(inclusion $\sqsubseteq$) $T_1 \sqsubseteq T_2$ iff $\mathit{Type}(T_1) \subseteq \mathit{Type}(T_2)$[*]

(intersection $\sqcap$) $\mathit{Type}(T_1 \sqcap T_2) = \mathit{Type}(T_1) \cap \mathit{Type}(T_2)$

(union $\sqcup$) $\mathit{Type}(T_1 \sqcup T_2) \supseteq \mathit{Type}(T_1) \cup \mathit{Type}(T_2)$

Also, we use equality between type symbols $T_1 = T_2$ as a shortcut for $\mathit{Type}(T_1) = \mathit{Type}(T_2)$. Note that type union is approximate. This is due to the fact that we, as many other researchers, use *deterministic* (or *tuple distributive*) types, in which if $f(a,b) \in \mathit{Type}(T)$ and $f(c,d) \in \mathit{Type}(T)$ then also $f(a,d) \in \mathit{Type}(T)$ and $f(c,b) \in \mathit{Type}(T)$.

We use regular types as abstract domain for the analysis. An abstract substitution is then a mapping from variables to types. Let $dom(\lambda)$ denote a domain of an abstract substitution λ, and let $\lambda_{|\overline{X}}$ be a projection of λ over variables $\overline{X}$. For an abstract substitution $\lambda = \{X_1/T_1, \ldots, X_n/T_n\}$, a value of the concretization function γ is given by: $\gamma(\lambda) = \{\{X_1/t_1, \ldots, X_n/t_n\} \mid t_i \in \mathit{Type}(T_i), 1 \leq i \leq n\}$.

7.4 Type Analysis with Predefined Types

As mentioned before, our type analysis is a part of the Ciao Preprocessor, CiaoPP [HPBLG05], and uses one of its analysis engines [HPMS00]. Moreover, as an underlying type inference system, we use the type analysis of [JB92] and [VB02] with various widenings (see [Mil99] for a comprehensive study on widenings in type domains). These analyses synthesize new types out of function symbols and constants present in the program. However, as a first technique in order to speed up analysis, especially in the context of large programs, we introduce a key new feature: in our analysis, types synthesized during analysis can

[*]As pointed out in [Lu01] this operation is incorrectly defined in [DZ92]. Our system uses the correct version of $\sqsubseteq$, implemented by Pedro López García, independently from [Lu01].

be *optionally* replaced by predefined types which are in the scope of the module being analyzed. These may have been written by the user in the module being processed, or imported from other user modules or from a library. The predefined types chosen as replacements are less precise or equivalent to the inferred ones, so that a safe (albeit potentially less precise) approximation of the semantics is still obtained.

Let $\mathcal{T}_0$ denote a set of predefined type symbols, including $\top$, $\bot$, and some more symbols. Every iteration of the analysis contains two steps: (1) synthesizing new types $T_1, \ldots, T_n$, as explained elsewhere (e.g., [JB92, VB02]), and (2) replacing them with $\lceil T_1 \rceil, \ldots, \lceil T_n \rceil$ where $\lceil . \rceil$ is a replacement operator which for $T' = \lceil T \rceil$ satisfies the following:

- it returns a predefined type, i.e., $T' \in \mathcal{T}_0$,
- it safely approximates T, i.e., $T \sqsubseteq T'$,
- and it is as precise as possible: $\nexists T'' \in \mathcal{T}_0$ s.t. $(T'' \sqsubset T' \land T \sqsubseteq T'')$.

Note that it is not always possible to find a unique best matching predefined type for a given synthesized type. There may be two or more types that are incompatible (or equivalent) and at the same time match a given synthesized type. As a heuristics, in the case of conflicts, we give priority to types which are defined in user modules (over those in library modules) since they are likely to look more familiar to the user. Also, types that are closer in the module hierarchy (i.e., defined in the current module or a closer module) are preferred.

In order to speed up the analysis, the $\sqsubseteq$ relation over $\mathcal{T}_0$ (let us denote it $\sqsubseteq_0$) is initially precomputed (with library and builtin types) before the analysis starts and, during the analysis of each module, incrementally complemented with types specific to that module. Thanks to this, checking the subtyping is efficient. Note however, that since $\mathcal{T}_0$ contains arbitrary types (we make no assumptions about $\mathcal{T}_0$ except that it always contains $\top$ and $\bot$) $(\mathcal{T}_0, \sqsubseteq_0, \sqcap_0, \sqcup_0)$ cannot directly serve as an abstract domain, as the following does not hold $\forall_{T_1,T_2 \in \mathcal{T}_0}\, \mathit{Type}(T_1 \sqcap T_2) = \mathit{Type}(T_1) \cap \mathit{Type}(T_2)$. For example consider $\mathcal{T}_0 = \{\mathit{list}, \mathit{atm}, \top, \bot\}$. The g.l.b. of atm and list induced by $\sqsubseteq_0$ would give $\bot$, whereas properly computed $\mathit{atm} \sqcap \mathit{list}$ should contain the empty list. A remedy for this is to use the standard $\sqcap$ (like for example type intersection of [DZ92]) and apply $\lceil . \rceil$ to the result, and thus to redefine $\sqcap_0$ so that $\forall_{T_1,T_2 \in \mathcal{T}_0} T_1 \sqcap_0 T_2 = \lceil T_1 \sqcap T_2 \rceil$. Clearly, $\sqcup_0$ can be directly computed by traversing the graph corresponding to the $\sqsubseteq_0$ relation.

7.5 Using Parametric Rules and Type Assertions

In order to make any analysis which works with modular programs realistic in practice, there must exist some degree of *separate* handling of code fragments. I.e., for scalability reasons, it is not realistic to expect that all modules related to an application should be available to analysis. A clear example for this are library modules. For them, we would like to have analysis information readily available, without the need of analyzing them over and over again for each application which uses them. To this end, our second technique consists in allowing developers of libraries (and modules which can be reused) to write, besides the usual regular type definitions, *parametric type rules*. It is important to note that regular types do not include in principle parametric type rules. Therefore, analysis does not infer this kind of rules and checking them will require some additional mechanism, as we propose in Section 7.5.4 below.

Let us introduce some notation. Let $\mathcal{TV}$ be a set of *type variables* or *parameters*. Now we admit also non-nullary symbols in $\mathcal{T}$. The notion of type symbol changes a bit, now it is a ground (parameter-free) term built of symbols from $\mathcal{T}$ (i.e. an element of $Term(\mathcal{T}, \emptyset)$). Parametric type rules have the form $l \rightarrow r$ where $l \in Term(\mathcal{T}, \mathcal{TV})$, $r = f(T_1, \ldots, T_n)$ $(f/n \in \mathcal{F}, T_i \in Term(\mathcal{T}, \mathcal{TV}))$, and $vars(r) \subseteq vars(l)$.

Example 7.5.1. *Consider the standard definition of a list.*

$$\begin{array}{lcl} list(\alpha) & \rightarrow & [\,] \\ list(\alpha) & \rightarrow & [\alpha \mid list(\alpha)] \end{array}$$

In our framework parametric rules have no denotation unless the parameters are instantiated to regular types by a *parameter substitution*. Let $T = t(\alpha_1, \ldots, \alpha_n)$ $(t/n \in \mathcal{T})$ where $\alpha_1, \ldots, \alpha_n$ are parameters. Then, the parameter substitution Ψ is a mapping $\{\alpha_1 \mapsto T_1, \ldots, \alpha_n \mapsto T_n\}$ where $T_1, \ldots, T_n$ are regular type symbols. Applying Ψ to T, written $\Psi(T)$, means replacing any occurrence of α_i in rules defining T, by T_i. After that, the l.h.s. of the rules become a type symbol (likewise all symbols in the r.h.s.), and thus parametric rules becomes parameter-free grammar rules, as presented in Section 7.3.1, and can be added to the type grammar.

Naturally, a parametric rule can be instantiated multiple times with different types, and resulting in different types, e.g., $list(int)$ and $list(atm)$. Note also that type $list(\perp)$ denotes an empty list.

The process of replacing synthesized types by predefined ones also takes parametric rules into account. During analysis, types constructed by instantiating parametric rules are added to the set $\mathcal{T}_0$ of predefined types. The new instances are created on the fly, by generating type substitutions $\Psi : \mathcal{T} \mapsto \mathcal{T}_0$, such that for a synthesized type T and a parametric type symbol T_p, the instance of T_p can serve as a good approximation of T, i.e., $\Psi(T_p) = \lceil T \rceil$.

Example 7.5.2. *Assume that at some intermediate step of the analysis the following abstract substitution is generated $\lambda = \{X/T\}$, where* Type$(T) = \{[a]\}$, *i.e. T denotes a one-element list of a's. Assume also that the definition of list (see Example 7.5.1) is present in the system. Since the constant a is described by the built-in type atm the analyzer would generate the parameter substitution $\Psi = \{\alpha \mapsto atm\}$ and finally would replace T by $\lceil T \rceil = \Psi(list(\alpha))$.*

Obviously, an abstract domain constructed as described above contains an infinite number of types, e.g., $list(num)$, $list(list(num))$, $list(list(list(num)))$, ...etc. Similarly to [BG92], we restrict the maximum depth of terms in parametric type symbols to an arbitrary number. Type symbols that occur below the maximum depth are simply replaced by $\top$. Our experiments show that depth value 3 is seldom exceeded in many programs and thus, in practice, no precision is usually lost in this step.

7.5.1 Type assertions

Information about intended or inferred call and success patterns is given in the form of *assertions* [PBH00c]. In this chapter we limit ourselves (without loss of generality) to just one form of assertion, "pred" assertions, written (in simplified form) as `pred` P : $Pre \Rightarrow Post$. P is a *predicate descriptor*, i.e., it has a predicate symbol as main functor and all arguments are distinct free variables, and *Pre* and *Post* are pre- and post-conditions respectively. For our purposes it is sufficient to consider that *Pre* and *Post* correspond to abstract substitutions (λ_{Pre} and λ_{Post} resp.) over variables of P. A more detailed description of assertions can be found in Chapters 4 and 8.

The meaning of `pred` assertions is twofold. First, the precondition *Pre* expresses properties which should hold in calls to P. Second, the postcondition *Post* expresses properties which should hold on termination of a successful computation of P, provided that *Pre* holds on call. Types are by default understood in "instantiation" mode [PBH00c], i.e., `=> list(L)` implies that at procedure output `L` is *instantiated* to a list[†], and `=> list(L,T)` implies that at procedure output `L` is instantiated to a list whose elements are of type `T`. Note that type expressions in assertions differ from type expressions as defined in previous sections. Their first argument is a variable whose type is described by the expression (this first argument should not be confused with a type parameter). There can be more than one `pred` assertion per predicate, each one describing a different usage of the predicate, for example:

```
:- pred length(L,N) : (var(L), int(N))  => list(L).
:- pred length(L,N) : (var(N), list(L)) => int(N).
```

In this case the *union* (disjunction) of the *Pre* parts expresses the properties which should hold in any call and the *Post* parts apply for calls matching their respective *Pre* part.

Herein we are interested in call and success patterns conveying only type information. It is possible to write a `pred` assertion with parametric types like:

```
:- pred reverse(X,Y) : list(X,A) => list(Y,A).
```

In this case, arguments are not identified by name but rather by position. We use `*` to separate the type of different arguments. This assertion tells us that the predicate `reverse/2` is meant to be invoked with the first argument bound to a list whose element can be of any type, denoted by the type variable `A`. Upon success, the procedure returns in the second argument a list whose elements must be of type `A`.

[†]Alternatively we can consider "compatibility" mode, meaning that either `L` is already instantiated to a list, or it might be instantiated to a list in the subsequent computations.

7.5.2 Using parametric type assertions in modular analysis

Assume a scenario where assertions are written in a (library) module and that for efficiency we do not want to analyze this module if possible. If no assertions are present in the module for exported predicates or if the preconditions in such assertions do not match the calling patterns the module will simply be entered and analyzed during modular analysis, as described in Section 7.3 (see also [PCH+04, PCPH06]). However, if suitable parametric assertions are present, assume that both precondition *Pre* and postcondition *Post* contain parameters $\overline{A}$ and $\overline{B}$ respectively. The assertion takes the following form: $:-$`pred` $P : Pre(\overline{A}) \Rightarrow Post(\overline{B})$. We require $\overline{B} \subseteq \overline{A}$. Our goal is to find, for a given call pattern λ_c, a valuation Ψ of parameters $\overline{A}$ (and therefore $\overline{B}$) such that $\lambda_c \sqsubseteq \lambda_{\Psi(Pre(\overline{A}))}$. Moreover, we are interested in finding a Ψ that gives $\Psi(Pre(\overline{A}))$ that is as precise as possible. In order to achieve this we use the *matching* operation of [DMP02]. Matching resembles checking of type inclusion (see [DZ92, Lu01]). We match a parameterless type T_1 against a, possibly parametric, type T_2, and denote this operation $T_1 \dot{\sqsubseteq} T_2$. Matching finds a (possibly small) parameter valuation Ψ so that $T_1 \sqsubseteq \Psi(T_2)$, or fails if such a valuation does not exist. The whole procedure starts with empty Ψ. Then matching, similarly to inclusion checking, traverses the type rules and involved terms recursively, and compares the corresponding structures. If at some point matching is about to compare a type parameter α and a type symbol, say T, then $\alpha \mapsto T$ is added to Ψ. It might however happen that another binding $\alpha \mapsto T'$ is already present in Ψ. In this case, $\alpha \mapsto T'$ is replaced by $\alpha \mapsto T' \sqcup T$.

Example 7.5.3. *Assume the following assertion describing a use of* `append/3`*:*

```
:- pred append(X,Y,Z): (list(X,A), list(Y,A))
                     => list(Z,A).
```

If the analyzer finds a call to `append(`$X1, X2, X3$`)` *with an abstract substitution* $\{X1/list(int), X2/list(int), X3/term\}$ *matching will generate the parameter valuation* $\{\mathtt{A} \mapsto int\}$. *If however the call pattern has a substitution* $\{X1/list(int), X2/list(atm), X3/term\}$ *then the valuation computed by matching would be* $\{\mathtt{A} \mapsto int \sqcup atm\}$.

Note that parameter handling is substantially different in typed logic programming (e.g., [MO84, HL94b]), where type inference involves type unification that tries to bind a type parameter to a single type and fails if any subsequent binding is incompatible with the previous one. Obviously, if T_2 has no parameters in $T_1 \dot{\sqsubseteq} T_2$, matching reduces to inclusion checking.

Note that without admitting parameters, in order to avoid reanalysis of the library using our proposed rules, the library developer would have to write a specific assertion for each possible type of list elements, which obviously is not feasible. Similarly, standard modular analysis would also save triples for each type of list elements that occurs every time that analysis enters the library. Another remedy is to write the most general assertion:

```
:- pred reverse(X,Y): list(X,term) => list(Y,term).
```

but in this case we unnecessarily lose precision.

7.5.3 Parametric type assertions in verification and debugging

As mentioned above, proving an assertion with parameters, like the one of Example 7.5.3, cannot be directly done by using the results of analysis. E.g., consider the assertion $:-$`pred` $P : \mathit{Pre}(\overline{A}) \Rightarrow \mathit{Post}(\overline{B})$ (1) (where, as before, we assume that $\overline{B} \subseteq \overline{A}$). Essentially, proving (1) means that we want to show: $\forall \Psi(\lambda_c \sqsubseteq \lambda_{\Psi(\mathit{Pre}(\overline{A}))} \Rightarrow \lambda_s \sqsubseteq \lambda_{\Psi(\mathit{Post}(\overline{B}))})$ (2) where λ_c and λ_s are, respectively, the call and success patterns computed by abstract interpretation. We propose a proving method which resembles the well-known skolemization technique. For every parameter α_i we introduce a dummy type c_i, such that $\forall T \in \mathcal{T} : T \neq \top \Rightarrow c_i \sqcap T = \bot$ and $\forall T \in \mathcal{T} : T \neq \bot \Rightarrow c_i \sqcup T = \top$. The intuition is that $\mathit{Type}(c_i)$ is disjoint from any terms in the program and initial goal. Let Ψ_c be a parameter valuation $\{\alpha_1 \mapsto c_1, \ldots, \alpha_k \mapsto c_k\}$.

Proposition 7.5.4. *Consider assertion (1). If abstract interpretation for a top goal P with the initial call pattern* $\lambda_{\Psi_c(\mathit{Pre}(\overline{A}))}$ *computes a success pattern* $\lambda_c = \lambda_{\Psi_c(\mathit{Post}(\overline{B}))}$ *then (2) holds.*

Proof (outline): Inductively, for every abstract operation we show that if the operation preserves a dummy type c_i, it will preserve an arbitrary type. Consider *a conjunction of two abstract substitutions $\lambda_1 \wedge \lambda_2 = \lambda$. Assume that a dummy type c occurs in λ_1, i.e., for some variable $X \in dom(\lambda_1)$ we have $\lambda_{1|X} = \{X/c\}$. If c propagates to the result of the conjunction, meaning that $\lambda_{|X} = \{X/c\}$, then either $X \notin dom(\lambda_2)$, or $\lambda_{2|X} = \{X/c\}$, or $\lambda_{2|X} = \{X/\top\}$ (as otherwise we would have $\lambda_{|X} = \{X/\bot\}$). It is clear that in either case any other type would propagate the same way as c. A similar reasoning can be performed for disjunction and projection.* □

The intuition behind Proposition 7.5.4 is that if a dummy type can be passed through the entire analysis process, meaning that it has not been "touched" by any abstract operation, we can conclude that any other type would be passed the same way.

7.6 Example

In this section we illustrate with a simple example our type analysis and its application to program verification. Our example consists of three modules. We start by describing the top-level module of the application, called `main`, whose code is shown below:

```
:- module(main,[p/2],[assertions,regtypes,functional]).
:- use_module(qs,[qsort/2]).

:- pred p(X,Y): list(X,num) => dlist(Y).    % #1
p(X,Y) :-  dlist(X), qsort(X,Y).

:- regtype dlist/1. dlist := [] | [~digit|dlist].

:- regtype digit/1. digit := 0|1|2|3|4|5|6|7|8|9.
```

In this code, the first line of the program contains the module declaration (in Ciao [BCC+06]), which defines the module name and the list of exported predicates, as well as declaring that several *packages* should be used (e.g., for assertion processing). Next, the `use_module` declaration informs the compiler and analyzer that this module imports procedure `qsort/2` from module `qs`. The `main` module contains two definitions of regular types: `dlist` and `digit`. These definitions are in fact also ordinary Prolog procedures written using Ciao's functional syntax [CCH06], and therefore besides their use in the post-condition of assertion

#1 they can also be used as regular (test) procedures, as is actually done in the clause defining p/2.

We now present the module qs, which implements *quicksort*, and whose code is shown below:

```
:- module(qs, [qsort/2], [assertions]).
:- use_module(library(lists),[append/3]).

:- pred qsort(X,Y) : list(X,A)  => list(Y,A).% #2
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L1,R1),
        qsort(L2,R2),
        append(R1,[X|R2],R).
qsort([],[]).

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
        E @< C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E @>= C,
        partition(R,C,Left,Right1).
```

An interesting feature of this implementation of quicksort is that the qsort/2 procedure is not restricted to lists of numbers and can in fact accept lists of any kind of elements due to the use of @< and @>= in the comparisons. In this case, the developer of the module has opted to include a pred parametric assertion, marked in the code as #2. This assertion states that on call, the first argument of the qsort/2 procedure is meant to be a list of elements of any type A, whereas upon success, the second argument should be bound to a list of type A. As can be observed, qs imports the standard append/3 predicate, for list concatenation, from the lists library. Therefore, the third and final module in our example is the *lists* library. For our purposes, there is no need to show its code here.

Assume now that we want to prove assertion #1 by analyzing statically module main. If we apply the inter-modular static analysis in [PCH$^+$04] using standard regular types, we can prove such assertion since analysis obtains the type dlist (or equivalently list(digit)) for the second argument of p/2. However, and as already argued, this approach is too costly (both time- and memory-wise) since analysis iterates over all modules, including libraries, analyzing for different call patterns until a global fixed point is reached.

We now show on this example how our proposal preserves the required accuracy in order to perform the verification task at hand, i.e., proving assertion `#1`, while simplifying the analysis process. A first step in our approach is, as described in Section 7.5, to avoid analyzing libraries, which in general is desirable except in initial phases of library development, verification, and testing. For this, the following assertion is present in the `lists` library:

```
:- checked pred append(X,Y,Z):(list(X,A),list(Y,A))
                               =>  list(Z,A).     % #3
```

Which states that, as expected, the result of concatenating (appending) two lists of a given type `A` results in a list of exactly such type `A`. The `checked` flag [PBH00c] in front of the assertion indicates that the assertion has been automatically proved to hold using the method described in Proposition 7.5.4.

Next, assume that we restrict ourselves to defined types (using the $\lceil . \rceil$ operation) as described in Section 7.4. With this assumption, when we analyze the procedure `qsort/2` for the call pattern induced by the `main` module, we get the type `list(digit)` in its first argument. Since we know that the `main` module belongs to the set of modules being analyzed, we consider the types defined in `main` as defined types, and therefore no accuracy is lost due to the use of defined types. Next, analysis will reach the call to `append/3`. By using assertion `#3`, the analyzer can deduce, without reanalyzing append, that upon success of `append(R1,[X|R2],R)` in the first clause of `qsort/2`, `R` will be bound to `list(digit)`. This type is propagated through the success of `qsort/2` to the calling module `main`, consequently allowing the system to prove assertion `#1`. Note that if we would like module `qs` to be reusable in any context without reanalyzing `qs` over and over again for different calling patterns, our approach allows introducing parametric assertions, such as assertion `#2`. Then, Proposition 7.5.4 can again be used to prove this parametric assertion once and forall.

7.7 Experimental evaluation

In order to evaluate the practical impact of our proposal, we have performed some preliminary benchmarking of modular analysis, in the context of inferring regular types, both with and without our proposed optimizations.

Bench	**Mod**	**Cls**	**Orig**	**Opt**	**SU**
ann	3	227	9711	7738	1.25
bid	8	69	7399	3392	2.18
boyer	4	145	1789	1905	0.94
manag_proj	8	907	289564	30962	9.35
check_links	6	576	41862	32392	1.29
grades	4	168	19392	9255	2.09
grade_listing	10	1553	86410	17427	4.96
Wgt. Arith. mean			117194	21297	5.50
Wgt. Geom. mean			69387	18178	3.82

Table 7.1: Intermodular analysis from scratch, using an underapproximating success policy (SP^-) and a top-down scheduling policy.

	module touch			**more general clause**			**recursion removal**		
Bench	**Orig**	**Opt**	**SU**	**Orig**	**Opt**	**SU**	**Orig**	**Opt**	**SU**
ann	1798	1134	1.59	6026	3204	1.88	3335	1585	2.10
bid	620	206	3.01	2035	574	3.54	1068	297	3.60
boyer	222	559	0.40	331	625	0.53	401	595	0.67
manag_proj	15134	15077	1.00	41664	14668	2.84	53383	1419	37.62
check_links	7439	6449	1.15	18232	9958	1.83	10353	6964	1.49
grades	3385	734	4.61	5432	1196	4.54	4186	933	4.49
grade_listing	4119	4557	0.90	36458	7173	5.08	16707	7353	2.27
W. Arith. mean	6985	6843	1.02	29459	8570	3.44	22475	4757	4.72
W. Geom. mean	5054	4741	1.07	21724	6838	3.18	14364	3406	4.22

Table 7.2: Reanalysis after several kinds of changes, using an underapproximating success policy (SP^-) and a bottom-up scheduling policy (in the case of recursion removal, SP^+ and top-down scheduling have been used.)

The analysis framework implemented in CiaoPP and used for this chapter can be configured selecting specific values for several parameters of the framework. The main parameters that can be selected when performing intermodular analysis are the scheduling policy and the success policy (see [PCH$^+$04] for more information on those and other parameters.) The scheduling policy allows the user to select how the framework decides at each iteration which module must be the next one selected for analysis during the intermodular fixed point computation. Two main approaches have been implemented: a *top-down* policy, traversing the intermodular dependency graph and selecting first the module requiring analysis which is higher in the graph (the top-level module is the top of the graph). The *bottom-up* policy takes first the deepest module in the intermodular dependency graph which requires analysis. When a program is analyzed from scratch, the first module analyzed is always the top-level one. The success policy, as already mentioned, selects how temporary results for calls to imported predicates are approximated when the exact success pattern is not available in the Global Answer Table. During the experiments, the parameters for both policies have been set to the most advantageous setting for the original type analysis, namely top-down and SP^{-}, respectively, in order to highlight the speedup obtained with just defined types.

The benchmark programs used are modular programs of medium size, ranging from three to ten modules, and from 69 to 1553 clauses. The number of modules and clauses for each program is detailed in Table 7.1. A brief description of the selected benchmarks follows. **ann** is the &-Prolog implementation of the MEL automatic parallelizer (by K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo) [MH90b]. **bid** computes an opening bid for a bridge hand (by J. Conery). The **boyer** benchmark is a reduced version of the Boyer/Moore theorem prover (by E. Tick). The program has been separated in four modules with a cycle between two modules. **managing_project** is a program developed by the authors for EU project management. **check_links** is an sample program for the *Pillow* HTML/XML/HTTP connectivity package (by D. Cabeza and M. Hermenegildo) that checks that links contained in a given URL address are reachable. Note that the whole Pillow package is analyzed together with the sample program. And finally, **grades** and **grade_listing** are programs used by the authors for grading students, and are composed of 4 and 10 modules, respectively. The experiments have been run on a Dell PowerEdge 4600 with two Pentium processors at 2 Ghz and 4 Gb of memory, and normal workload. Analysis time in the experiments corresponds to the time spent (in milliseconds) analyzing code. Tasks related to program loading and unloading, and saving analysis results to disk are not part of the optimizations described in this chapter and have been excluded from the tables.

The first experiment involves analyzing from scratch the benchmark programs using the intermodular analysis algorithm. The results are shown in Table 7.1. Column **Mod** contains the number of modules that compose each benchmark, while column **Cls** includes the number of clauses. Column **Orig** is the time spent by the original regular type analysis, and column **Opt** is the time spent by the analysis with the optimizations described in this chapter. Finally, column **SU** (speedup) shows the improvement brought by the optimized version over the traditional type analysis. In this case the improvement is considerable, supporting our thesis that the optimizations proposed are specially appropriate for modular analysis. The last two lines of tables 7.1 and 7.2 show the arithmetic and geometric means of the results obtained for each column, weighted by the number of clauses in each program. On average, our proposed optimizations speed up intermodular analysis by a factor of 3.82-5.5.

Table 7.2 shows how the optimizations proposed improve the analysis results when benchmark programs are reanalyzed in an incremental way, after several specific modifications are made to the source code. For evaluating this, it is important to make experiments which are representative of the kind of changes which occur in real-life. We have followed here the approach used in [CPHB06], were three different kinds of source code modifications have been studied. For the three classes of changes, one module is modified each time and then the program is reanalyzed, in order to incrementally recompute the analysis results visiting only the modules that require reanalysis: the changed module, plus possibly other modules transitively affected by this change. The numbers shown in Table 7.2 are the average of the times taken by the reanalysis of the program when each of the modules in the benchmark programs is modified.

In the first kind of change, a simple modification is made in a single module in such a way that this modification does not change the results of analysis for that module (named **module touch** in that table). This has been implemented by "touching" the module, i.e., changing the modification time without actually modifying its contents, in order to force CiaoPP to reanalyze it. It can be observed that the optimizations introduced do not provide much speedup. The main reason for this behavior is that only the modified module is visited by the modular analysis. This means that the reduction in the number of intermodular iterations and number of types used is not relevant in this case. Also, a single module in a modular program is typically a rather small piece of code, which may not be large enough to take advantage of the use of just defined types.

The second kind of modification shown in Table 7.2 is a modification in the

Bench	**Orig**	**Optim**	**SU**
ann	1291	1605	0.80
bid	1972	731	2.70
boyer	973	935	1.04
manag_proj	37177	28849	1.29
check_links	10266	7158	1.43
grades	5653	2348	2.41
grade_listing	60552	16052	3.77

Table 7.3: Time spent by the monolithic analysis of different benchmark programs.

source code such that after the change exported predicates produce more general analysis results (named **more general clause** in Table 7.2). It is implemented by adding a most general fact to all exported predicates of a given module. This kind of change is an extreme situation in which all exported predicates are affected. This modification in general requires that not only the modified module be reanalyzed, but also some other related modules, since the analysis results for the modified module are different, and very likely affect the modules which import the modified one. It is encouraging to observe that the optimizations introduced by our approach appear to be specially relevant in this case, since due to them the process of reanalyzing the program is sped up by a factor of more than three.

The third case corresponds to a source code modification in which exported predicates produce a more precise answer pattern (named **recursion removal** in Table 7.2). In this case all the clauses of the exported predicates of a given module have been replaced by the first non-recursive clause of the predicate. As in the previous case, this is an extreme case in which all exported predicates are affected, except that now a more particular answer is obtained for them instead of a more general one. Again, the reanalysis of the program after the change generally requires analyzing other modules besides the modified one. As shown in the table the modular analysis is very competitive when using our proposed approach, bringing significant speedup also in this case, by a factor of more than four.

Finally, the third experiment, shown in Table 7.3, presents the time spent when analyzing the programs in a non-modular way, i.e., as if all the program code were located in a single module (we refer to this as the “monolithic analysis”). The results suggest that although the optimizations presented in this chapter aim at improving modular analysis, they are also useful in the case of large non-modular programs, since the analysis speed improves for most of the benchmarks. However, as can be seen for **ann**, there may be cases where the original analysis is somewhat faster. This can be explained by the fact that though using the $\lceil . \rceil$ operation results in analyzing fewer call patterns, the replacement operation also introduces some overhead. The results also show that modular analysis times remain, as expected, somewhat higher than when analyzing in a monolithic way, but are reasonable in comparison, and the comparison with the analysis times for reanalysis after partial changes are quite encouraging, since they often improve on the monolithic analysis times, notably for some of the programs with larger execution times. Note also that of course modular analysis is vital when programs cannot be analyzed monolithically due to, e.g., memory limitations.

7.8 Conclusions

We have proposed a combination of techniques aimed at improving analysis efficiency in type inference and verification for modular Prolog programs. In particular, we have presented a type analysis which optionally reduces the accuracy of inferred types during the analysis process by using only the types defined by the user or present in the libraries. Also, borrowing some ideas from polymorphic type systems, we have proposed a method that allows using polymorphic type rules for specifying library module boundaries, and we have proposed a novel method in order to use such type rules in the context of a regular type-based analysis system. Finally, we have also implemented our approach and reported experimental results from the analysis of a number of modular programs.

Our experimental results suggest that the optimizations presented do contribute significantly to increasing analysis efficiency both for the monolithic case and, even more, for the case of analyzing programs module by module. This holds both when analyzing programs from scratch as well when doing it incrementally after changes to a single module. Modular analysis times remain, as expected, somewhat higher than analyzing them in a monolithic way, but are reasonable, and the results from reanalysis after partial changes are quite encouraging, improving sometimes on the monolithic analysis times. Another advantage of the proposed approach is that the output of analysis is more readable, since it is presented to the user in terms of known types, rather than in terms of new, inferred ones, which are typically more detailed and have automatically generated names, and which can sometimes be difficult to interpret. Furthermore, the precision obtained appears to be sufficient for the purpose of verifying type signatures given in the form of assertions.

In summary, we argue that our proposal is of practical relevance, since it allows reducing analysis cost significantly while preserving a useful level of accuracy.

Part III

Compile-time Assertion Checking of Modular Programs

Chapter 8

Context-Sensitive Multivariant Assertion Checking in Modular Programs

We propose a *modular*, assertion-based system for verification and debugging of large logic programs, together with several interesting models for checking assertions statically in modular programs, each with different characteristics and representing different trade-offs. Our proposal is a *modular* and *multivariant* extension of our previously proposed abstract assertion checking model and we also report on its implementation in the CiaoPP system. In our approach, the specification of the program, given by a set of assertions, may be partial, instead of the complete specification required by traditional verification systems. Also, the system can deal with properties which cannot always be determined at compile-time. As a result, the proposed system needs to work with *safe* approximations: all assertions proved correct are guaranteed to be valid and all errors actual errors. The use of modular, context-sensitive static analyzers also allows us to introduce a new distinction between assertions checked in a particular context or checked in general.

8.1 Introduction

Splitting program code into modules is widely recognized as a useful technique in the process of software development. Also in the context of (constraint) logic programming (CLP) the modular approach to constructing programs has received considerable attention (see [BLM94] for an early survey). In addition, the development of large applications can greatly benefit from advanced analysis, verification, and debugging tools (see for example [DHM00]). Some of these tools check programs with respect to a specification provided in the form of assertions [PBH00b, DNTM89], written optionally by the user. In this chapter we propose a framework for static (i.e., compile-time) checking of assertions in modular logic programs, based on information from global analysis.

Within our framework, the programmer is expected to write a (partial) specification for a module (or a set of modules) being subject to the verification process. The specification is written in terms of (Ciao) assertions [BCHP96, BDM97, DNTM89, M9́5, PBH00b]. From the programmer's viewpoint, these assertions resemble the type (and mode) declarations used in strongly typed logic languages such as Gödel [HL94a] or Mercury [SHC96] and in functional languages. However, when compared to the latter, note that in logic programming arguments of procedures behave differently in the sense that arguments might be either input or output, depending on the specific *usage* (i.e., the context) of the procedure. For instance, the classical predicate `append/3` can be used for concatenating lists, for decomposing lists, for checking or finding a prefix of a given list, etc. Therefore, our assertion language and the checking procedure are designed to *allow various usages of a predicate.* Moreover, comparing to the former, herein we are interested in supporting a general setting in which, on one hand assertions can be of a quite general nature, including properties which are *undecidable*, and, on the other hand, only a small number of assertions may be present in the program, i.e., the assertions are *optional.* In particular, we do not wish to limit the programming language or the language of assertions unnecessarily in order to make the assertions decidable.

Consequently, the proposed framework needs to deal throughout with *approximations* [BDD$^+$97, CC77a, HPBLG05]. It is imperative that such approximations be performed in a safe manner, in the sense that if an "error" (more formally, a *symptom*) is flagged, then it is indeed a violation of the specification. However, while a system can be complete with respect to decidable properties (e.g., certain type systems), it cannot be complete in general, in the sense that when undecidable properties are used in assertions, there may be errors with respect to such assertions that are not detected at compile-time. This is a trade-off that we accept in return for a greater flexibility. However, in order to detect as many errors as possible, the framework combines *static* (i.e., compile-time) and *dynamic* (i.e., run-time) checking of assertions. In particular, run-time checks are (optionally) generated for assertions which cannot be statically determined to hold or not. In this thesis we will concentrate on compile-time checking only.

Our approach is strongly motivated by the availability of powerful and mature static analyzers for (constraint) logic programs (see, e.g., [BCHP96, LV94, dlBHB$^+$96a, MH92] and their references), generally based on abstract interpretation [CC77b]. Also, since we deal with modular programs, context-sensitive static analyses that handle modules (e.g., [CC02a, CDG93] among others) provide us with suitable background. Especially relevant is our recent work on context sensitive, multivariant modular analysis (see Part II and [PH00, BdlBH$^+$01, PCH$^+$04, CPHB06]). These analysis systems can statically infer a wide range of properties (from types to determinacy or termination) accurately and efficiently, for realistic modular programs. We would like to take advantage of such program analysis tools, rather than developing new abstract procedures, such as concrete [DNTM89] or abstract [CLMV99, PBH00d] diagnosers and debuggers, or using traditional proof-based methods, [AM94, AP93, Der93, Fer87, DM88].

This chapter builds on [PBH00c] where the assertion language that we use was introduced, and on [PBH00d] where a proposal for the formal treatment of assertion checking, both at compile-time and at run-time, was presented. We extend the above-mentioned work in four main directions. Most importantly, the solution of [PBH00d] is not modular. We show herein how to check assertions in modular programs in a way that ensures the soundness of the approach. Also, the formalization is different to that of [PBH00d], the present one being based on generalized AND trees. In addition, in this chapter *multivariant* information generated by the analysis is exploited. This essentially means that multiple usages of a procedure can result in multiple descriptions in the analysis output. In consequence, this enables us to verify the code in a more accurate way.

Modular verification has also been studied within OO programming (e.g., [LM05]) where the importance of contextual correctness, as in this thesis, has been recognized. Nevertheless that work differ from this thesis in several respects, the most important one being that they are based on traditional Hoare-like based verification techniques and the full specification is required, whereas our framework is based on abstract interpretation and allows for partial specifications.

In the context of Logic Programming [CLV98] shows how to perform abstract diagnosis of incomplete logic programs. Our approach is similar to theirs, since the correctness of a modular program is established in terms of the correctness of its modules. However, in [CLV98] the complete specification is needed and, more importantly, context-sensitive analysis information is not used, and therefore there is no concept of correctness in context. We claim that this is an important advantage of our approach, because it allows the validation of a module in a given program even when it is not possible to validate it in a context-independent way.

Our basic tool for checking assertions is *abstract interpretation* [CC77b], as it has been described in Part I. In order to formally represent results of concrete execution and abstract interpretation of logic programs, the formalism of generalized AND trees and abstract AND-OR trees will be used. In the rest of the chapter we will use CP and AP to refer to abstract substituions stored in the GAT, and λ for other abstract substitutions.

8.2 Assertions

We consider two fundamental kinds of (basic) assertions [PBH00c].* The first one is `success` assertions, which are used to express properties which should hold on termination of a successful computation of a given predicate (*postconditions*). At the time of calling the predicate, the computation should satisfy a certain *precondition.* `success` assertions can be expressed in our assertion language using an expression of the form: `success` $P : Pre \Rightarrow Post$, where P is a predicate descriptor, and *Pre* and *Post* are pre- and post-conditions respectively. Without loss of generality, we will consider that *Pre* and *Post* correspond to abstract

*[PBH00c] presents other types of assertions, but they are outside the scope of this work.

substitutions (λ_{Pre} and λ_{Post} resp.) over $vars(P)$. This kind of assertion should be interpreted as "in any invocation of P if *Pre* holds in the calling state and the computation succeeds, then *Post* should also hold in the success state." The postcondition stated in a `success` assertion refers to *all* the success states (possibly none). Note that `success` P : *true* $\Rightarrow$ *Post* can be abbreviated as `success` $P \Rightarrow$ *Post*.

A second kind of assertions expresses properties which should hold in any call to a given predicate. These properties are similar in nature to the classical *preconditions* used in program verification. These assertions have the form: `calls` P : *Pre*, and should be interpreted as "in all activations of P *Pre* should hold in the calling state." More than one assertion may be written for each predicate. That means that, in any invocation of P, at least one `calls` assertion for P should hold.

Finally, we write `pred` P : *Pre* $\Rightarrow$ *Post*, as a shortcut for the two assertions: `calls` P : *Pre* and `success` P : *Pre* $\Rightarrow$ *Post*. We write `pred` $P \Rightarrow$ *Post* if *Pre* = *true*. We claim that the `pred` form is a natural way to describe a usage of the predicate. In what follows, we will use `calls` (resp. `success`) assertions when we want to refer to the calls part (resp. success part) of a `pred` assertion. We will assume, for simplicity and with no loss of generality, that all assertions referring to a predicate P defined in module m are also provided in that module. We will denote with $assertions(m)$ the set of assertions appearing in module m, and $assertions(P)$ refers to the assertions for predicate P.

Example 8.2.1. *A possible set of* `calls` *assertions for the traditional* `length/2` *predicate that relates a list to its length, might be:*

```
:- calls length(L,N) : (var(L), int(N)).    %(1)
:- calls length(L,N) : (list(L), var(N)).   %(2)
```

These assertions describe different modes for calling that predicate: either for `(1)` *generating a list of length* `N`*, or* `(2)` *to obtain the length of a list* `L`.

Possible success assertions for that predicate are:

```
:- success length(L,N) : (var(L), int(N))  => list(L).
:- success length(L,N) : (var(N), list(L)) => int(N).
```

The following two assertions are equivalent to all the previous assertions for `length/2`*:*

```
:- pred length(L,N) : (var(L), int(N))  => list(L).
:- pred length(L,N) : (var(N), list(L)) => int(N).
```

We assign a *status* to each assertion. The status indicates whether the assertion refers to intended or actual properties, and the relation between the property and the program semantics. This section builds on [PBH00d], but it has been adapted to our use of generalized AND trees.

We say that a `calls` assertion A with predicate descriptor P is *applicable* to a node $N = \langle \theta_c, (A, P'), \theta_s \rangle$ of the generalized AND tree if there is $\sigma \in ren$ (a renaming substitution) s.t. $P' = P\sigma$ and N is adorned on the left, i.e., the call substitution θ_c of N has been already computed. A `success` assertion A with predicate descriptor P is applicable to a node N if $P' = P\sigma$ (where $\sigma \in ren$) and N is adorned on the right, i.e., the success substitution θ_s of the call at N has been computed (the procedure exit has been completed). In what follows, we will denote with ρ a suitable renaming substitution.

If an assertion holds within a fixed set of queries Q then the assertion is said to be *checked* with respect to Q. If this is proved, the assertion receives the corresponding status `checked`. Formally:

Definition 8.2.2 (Checked assertions). *Let R be a program.*

- *An assertion $A =$ `calls` $P : Pre$ in R is* checked *w.r.t. the set of queries Q iff $\forall \theta_c \in calling\_context(P, R, Q), \theta_c\rho \in \gamma(\lambda_{Pre})$.*
- *An assertion $A =$ `success` $P : Pre \Rightarrow Post$ in R is* checked *w.r.t. a set of queries Q iff $\forall (\theta_c, \theta_s) \in success\_context(P, R, Q), \theta_c\rho \in \gamma(\lambda_{Pre}) \rightarrow \theta_s\rho \in \gamma(\lambda_{Post})$.*

A calls or success assertion can also be *false*, whenever it is known that there is at least one call (or success) pattern in the concrete semantics that violates the property in the assertion. If we can prove this, the assertion is given the status `false`. In addition, an error message will be issued by the preprocessor.

Definition 8.2.3 (False assertions). *Let R be a program.*

- *An assertion $A =$ `calls` $P : Pre$ in R is* false *w.r.t. the set of queries Q iff $\exists \theta_c \in calling\_context(P, R, Q)$ s.t. $\theta_c\rho \notin \gamma(\lambda_{Pre})$.*

- *An assertion $A =$* `success` *$P : Pre \Rightarrow Post$ in R is* false *w.r.t. the calling context Q iff $\exists(\theta_c, \theta_s) \in success\_context(P, R, Q)$ s.t. $\theta_c\rho \in \gamma(\lambda_{Pre}) \wedge \theta_s\rho \notin \gamma(\lambda_{Post})$.*

Finally, an assertion which expresses a property which holds for any initial query is a *true* assertion. If it can be proven, independently on the calling context, during compile-time checking, the assertion is rewritten with the status `true`. Formally:

Definition 8.2.4 (True success assertion). *An assertion $A =$* `success` *$P : Pre \Rightarrow Post$ in R is* true *iff for every set of queries Q, $\forall(\theta_c, \theta_s) \in success\_context(P, R, Q), \theta_c\rho \in \gamma(\lambda_{Pre}) \rightarrow \theta_s\rho \in \gamma(\lambda_{Post})$.*

Note that the difference between checked assertions and true ones, is that the latter hold for any context. Thus, the fact that an assertion is true implies that it is also checked.

Assertions are subject to compile-time checking. An assertion which is not determined by compile-time checking to be given any of the above statuses is a *check* assertion. This assertion expresses an intended property. It may hold or not in the current version of the program. This is the default status, i.e., if an assertion has no explicitly written status, it is assumed that the status is `check`. Before performing a compile-time checking procedure all assertions written by the user have `check` status.

In our setting, checking assertions must be preceded by analysis, and basically it boils down to comparing assertions (whenever applicable) with the abstract information obtained by analysis. Below we present sufficient conditions for compile-time assertion checking in a program not structured in modules. The following sections will deal with assertion checking of modules and modular programs. In the case of proving a calls assertion, we would like to ensure that all concrete calls are included in the description λ_{Pre}. For disproving calls assertions, i.e., turning them to false, we want to show that there is some concrete call which is not covered by λ_{Pre}.

Definition 8.2.5 (Abstract assertion checking). *Let R be a program, and Q_α an abstract description of queries to R.*

- *An assertion* $A =$ `success` $P : Pre \Rightarrow Post$ *in* R *is* abstractly true *iff* $\exists P'{:}\lambda^c \mapsto \lambda^s \in analysis(R, \{P : \lambda_{Pre}\})$ *s.t.* $\exists \sigma \in ren,\ P' = P\sigma, \lambda^c = \lambda_{Pre} \wedge \lambda^s \sqsubseteq \lambda_{Post}$.

- *An assertion* $A =$ `success` $P : Pre \Rightarrow Post$ *in* R *is* abstractly checked *w.r.t.* Q_α *iff* $\forall P'{:}\lambda^c \mapsto \lambda^s \in analysis(R, Q_\alpha)$ *s.t.* $\exists \sigma \in ren,\ P' = P\sigma, \lambda^c \sqsubseteq \lambda_{Pre} \rightarrow \lambda^s \sqsubseteq \lambda_{Post}$.

- *An assertion* $A =$ `calls` $P : Pre$ *in* R *is* abstractly checked *w.r.t.* Q_α *iff* $\forall P'{:}\lambda^c \mapsto \lambda^s \in analysis(R, Q_\alpha)$ *s.t.* $\exists \sigma \in ren,\ P' = P\sigma, \lambda^c \sqsubseteq \lambda_{Pre}$.

- *An assertion* $A =$ `success` $P : Pre \Rightarrow Post$ *in* R *is* abstractly false *w.r.t.* Q_α *iff* $\forall P'{:}\lambda^c \mapsto \lambda^s \in analysis(R, Q_\alpha)$ *s.t.* $\exists \sigma \in ren,\ P' = P\sigma, \lambda^c \sqsubseteq \lambda_{Pre} \wedge (\lambda^s \sqcap \lambda_{Post} = \bot)$.

- *An assertion* $A =$ `calls` $P : Pre$ *in* R *is* abstractly false *w.r.t.* Q_α *iff* $\forall P'{:}\lambda^c \mapsto \lambda^s \in analysis(R, Q_\alpha)$ *s.t.* $\exists \sigma \in ren,\ P' = P\sigma, \lambda^c \sqcap \lambda_{Pre} = \bot$.

In this definition $analysis(R, Q_\alpha)$ is a generic analysis computation, and therefore the definition is parametric with respect to the analysis actually performed for checking the assertions, as will be shown below. Now we show that Definition 8.2.5 can indeed serve as a sufficient condition:

Proposition 8.2.6 (Checking a calls assertion). *Let* $A =$ `check calls` $P : Pre$ *be an assertion.*

- *If* A *is abstractly checked w.r.t.* Q_α*, then* A *is checked w.r.t.* $\gamma(Q_\alpha)$*.*

- *If* A *is abstractly false w.r.t.* Q_α*, then* A *is false w.r.t.* $\gamma(Q_\alpha)$*.*

- *otherwise, nothing can be deduced about* A *considered atomically (and it is left in check status).*

Soundness of the above statements can be derived directly from the correctness of abstract interpretation. In the case of checked assertions, we make sure that all call patterns that can appear at run-time belong to $\gamma(\lambda_{Pre})$. The "false" cases are a bit more involved. Due to the approximating nature of abstract interpretation, there is no guarantee that a given abstract call description λ^c corresponds to any call pattern that can appear at run-time. Thus, it is possible that the assertion is never applicable, but if it is, it will be invalid. What is known is that every run-time call pattern is described by one or more entries for P in AT. Thus, in order to ensure that no call pattern will satisfy λ_{Pre}, all λ^c's for P must be taken into account.

Finally, if a `calls` assertion is not abstractly checked nor abstractly false, we cannot deduce anything about A when it is considered atomically. However, we could still split it, and apply the same process to the parts.

Example 8.2.7. *Let us consider the* `calls` *assertions shown in Example 8.2.1, and suppose that after analyzing* `length/2` *we can check that in any call to* `length/2` *the first argument is a free variable, but there is no information available regarding the second argument. Then, the following assertion*

```
:- calls length(L,N) : (var(L), int(N)).    %(1)
```

cannot be abstractly checked if it is considered atomically. However, the assertion can be split in two parts, and different statuses can be set to each of the parts: `checked` *for the part referring to the first argument, and* `check` *for the one corresponding to the second argument.*

Proposition 8.2.8 (Checking a success assertion). *Let $A =$* `check success` *$P : Pre \Rightarrow Post$ be an assertion.*

- *If A is abstractly true, then A is true.*
- *If A is abstractly checked w.r.t. Q_α, then A is checked w.r.t. $\gamma(Q_\alpha)$.*
- *If A is abstractly false w.r.t. Q_α, then A is false w.r.t. $\gamma(Q_\alpha)$.*
- *otherwise, nothing can be deduced about A considered atomically (and it is left in check status).*

In order to show a `success` assertion to be true, a separate analysis with the entry point λ_{Pre} may run (as detailed in Definition 8.2.5), even though call patterns from $\gamma(\lambda_{Pre})$ may not appear in the program concrete execution starting from calls (satisfying `pred` assertions) to exported predicates. Consequently, the assertion is handled independently on the calling context. In this method, it is ensured that every success pattern described by λ^s originates from the call patterns that are described by the precondition λ_{Pre}. Thus, in order to prove the assertion to be true it suffices to perform the check $\lambda^s \sqsubseteq \lambda_{Post}$. The assertion is shown to be checked if either there is no call in the current calling context that satisfies the preconditions, or the postcondition holds. The assertion is proved false if it can be guaranteed that there are call patterns satisfying the preconditions, for which the postcondition surely does not hold for the corresponding success patterns. In the same way as before, a `success` assertion remains atomically `check` when it is not abstractly checked nor abstractly false. We can however simplify the assertion when part of the assertion can be proved to hold, like in a `calls` assertion. Note that the more precise analysis results are, the more assertions get status `true`, `checked` and `false`.

8.3 Checking assertions in a single module

The modular analysis framework described in Chapter 5 is independent from the assertion language. Nevertheless, assertions may contain relevant information for the analyzer. To this end when $analysis(m, E, AT)$ is computed for a module m, the parameters E and AT can also refer to information gathered directly from assertions, rather than from other analysis steps. This yields additional entry and success policies with respect to those described in Chapter 5:

- E can be extracted from the call parts of `pred` assertions for exported predicates in m. In this case, the module is analyzed separately from the actual code of the importing modules; in fact they need not even be implemented, they just have to obey the interface requirements, stated by the calls assertions of their exported predicates. Using this option we can reason about partial correctness of the module, since using the `pred` assertions as starting point for analysis ensures that we cover all initial queries that satisfy the assertions. Such set will be denoted as

$$\begin{aligned}\mathcal{CP}_m^{Asst} = \{&P : \lambda_{Pre} \mid P \in exported\_preds(m) \\ &\wedge \texttt{pred}\ P : Pre \Rightarrow Post\ \in assertions(m)\} \\ &\cup\ \{P : \top \mid P \in exported\_preds(m) \wedge assertions(P) = \emptyset\}.\end{aligned}$$

- AT can also be extracted from `pred` (or `success`) assertions found in the imported modules. This allows checking incomplete programs, as the imported predicates do not even have to be implemented. Again, we assume the topmost success pattern if no assertions are present. This provides a possibly inaccurate yet sound approximation. Given a module m, the answer table generated from the assertions for imported modules is denoted as

$$\begin{aligned}\mathcal{AT}_m^{Asst} = \bigcup_{n \in imports(m)} & (\{P : \lambda_{Pre} \mapsto \lambda_{Post} \mid P \in exported\_preds(n) \\ & \wedge \texttt{pred}\ P : Pre \Rightarrow Post\ \in assertions(n)\} \\ & \cup \{P : \top \mapsto \top \mid P \in exported\_preds(n) \wedge assertions(P) = \emptyset\}).\end{aligned}$$

Since the information used in this new success policy is not obtained by the analysis but written by the programmer, the correctness of the analysis results depends on the correctness of the assertions used. It will be studied below.

Note that we assume the topmost patterns if no assertions are present. In what follows, we will refer to the entry policy described in Chapter 5 as $\mathcal{CP}_m^{GAT}$.

When checking assertions of modular programs, a given module can be considered either in the context of a program unit or separately, taking into account only the imported predicates. When treated in the context of a program unit, the calling context of a module m is called the *set of initial queries* Q_m. We say that the set of initial queries Q_m to a module m is *valid* iff for every imported predicate p all the calls assertions related to p are checked w.r.t. Q_m.

Definition 8.3.1 (Partially correct in context module). *A module m is* partially correct in context *with respect to a set of initial queries Q_m iff (1) every calls assertion in m is checked w.r.t. Q_m, and (2) every success assertion in m is true, or checked w.r.t. Q_m, and (3) every calls assertion for a predicate imported by m is checked with respect to Q_m.*

Definition 8.3.2 (Partially correct module). *A module m is* partially correct *iff m is partially correct in context w.r.t. any valid set of initial queries.*

8.3.1 Single module edit-check cycle

Assertions are checked, as explained above, w.r.t. all analysis information available for a given (call or success of a) predicate after executing the analysis of the code. Such analysis information is multivariant, and covers all the program points in the analyzed code where a given predicate is called and how it succeeds. If available, a *GAT* table can be used to improve the analysis results with information from previous analyses of imported modules.

In our system, when checking a module, `calls` assertions for imported predicates are visible, and can therefore also be checked. This enables verifying whether a particular call pattern to an imported predicate satisfies its assertions. Of course, a `calls` assertion cannot be given status `true` or `checked`, as in general not all call patterns for the imported predicate occur in the calling module. Nevertheless, a warning or error is issued whenever the assertion is violated and/or cannot be shown to hold.

The following proposition allows us to determine a basic partial correctness result for a module:

Proposition 8.3.3. *Let m be a module, and $LAT = analysis_{SP^+}(m, \mathcal{CP}_m^{Asst}, AT)$, where AT is an over-approximating answer table for (some modules in) $imports(m)$.*

The module m is partially correct if all success assertions are abstractly true w.r.t. LAT and all calls assertions for predicates in m and $imported\_preds(m)$ are abstractly checked w.r.t. LAT.

This proposition considers correctness of a single module regardless of the calling context of the module, since the starting point of the analysis is the set of preconditions in `pred` assertions. Note that *LAT* must be computed using an over-approximating success policy, in order to obtain correct results (provided that *AT* is correct). The answer table *AT* used for the analysis may be incomplete, or even an empty set: this approach allows us to check the assertions of a given module even when there is no information available from the imported modules. However, the more accurate *AT* is, the more assertions get status `true` or `checked`. This proposition is especially useful during the development of a modular program (i.e., the "edit-check cycle"), when different programmers develop different modules of the program. A programmer can check the assertions of his/her module as soon as it is syntactically correct. If other programmers in the team have analyzed their modules already, a shared *GAT* can be used to generate the answer table *AT* for checking the module more accurately.

Example 8.3.4. *The following module computes the cardinality of a list (i.e., the number of distinct elements in a list):*

```
:- module(cardinality,[card/2],[assertions]).
:- use_module(lists,[length/2,remove_dups/2]).
:- pred card(L,C) : list(L) => num(C).
card(L,C):- remove_dups(L,L1), length(L1,C).
```

The first line of the program contains the module declaration in the Ciao language, that defines the module name and the list of exported predicates, as well as declaring the usage of the `assertions` package*, which is needed for assertion processing. The second line declares that* `length/2` *and* `remove_dups/2` *defined in the module* `lists` *will be imported.*

Predicate `card/2` *first removes duplicated elements of the list in the first argument by means of* `remove_dups/2`*, and then gets the length of that list using* `length/2`*.*

Since `length/2` *and* `remove_dups/2` *are imported from the module* `lists`*, their code is not accessed by the analyzer when processing* `cardinality` *separately. If there is no information available for those predicates, when analyzing* `card/2` *with an SP^+ success policy no useful information will be produced. If however* `lists` *has been analyzed before, the resulting information may be useful for the analysis of* `card/2`*.*

Unfortunately, if the modules imported by m are not implemented yet, there is no possibility to analyze them in order to provide more accurate information to the analyzer. In order to overcome that, we can use the assertion information for the exported predicates in imported modules to obtain a more precise LAT. In this case, correctness of the module cannot be guaranteed, but a weaker notion of correctness, conditional partial correctness, may be proved. Note that in this case the analysis relies on possibly unverified assertions written by the user.

Proposition 8.3.5. *Let m be a module, and $LAT = analysis_{SP^+}(m, \mathcal{CP}_m^{Asst}, \mathcal{AT}_m^{Asst})$.*

The module m is conditionally partially correct if all success assertions are abstractly true, and all calls assertions for predicates in m and $imported\_preds(m)$ are abstractly checked w.r.t. LAT.

This conditional partial correctness turns into partial correctness when the program unit is taken as a whole, as we will see in Section 8.4.

Example 8.3.6. *Let us assume that in Example 8.3.4 module* `lists` *contains the following assertions:*

```
:- pred length(L,N) : list(L) => num(N).
:- pred remove_dups(L,S) : list(L) => list(S).
```

In this case, module `cardinality` *can be analyzed using the information in those assertions, and then that module can be checked conditionally, where the condition is the correctness of those assertions.*

Example 8.3.7. *The power of the assertion language and the assertion checking system can be illustrated with the* `functor/3` *library predicate. The ISO standard for Prolog [PRO94] states that* `functor/3` *can only be invoked using two possible calling modes, and any other mode will raise a run-time error. The first mode allows obtaining the functor name and arity of a structure, while the second calling mode builds up a structure given its functor name and arity.*

Our assertion checking system is able to statically detect such calling patterns because several assertions are allowed for a given predicate, and the underlying analyzer captures context-sensitive, multivariant abstract information. They can be expressed by means of the following assertions:

```
:- pred functor(+T,Name,Arity) => (atomic(Name),nat(Arity)).
:- pred functor(T,+Name,+Arity) : (atomic(Name),nat(Arity)) => nonvar(T).
```

In these assertions, the plus sign before an argument has the usual meaning of a Prolog mode, i.e., that the argument cannot be a free variable on calls. The calls parts of these assertions will be used when analyzing and checking any module that uses this library predicate, in order to check the calling modes to it.

8.4 Checking assertions in a program unit

Checking assertions in a program unit consisting of several modules differs from checking assertions in a single module in some ways. First of all, the most accurate initial queries to a given module m are provided by the calls to m made by other modules in the program unit (except the top-level one). Secondly, the success patterns of imported predicates may also be more accurate if we consider a given program unit. This leads us to the notion of correctness for program units. Note that the following definition concerns the concrete semantics.

Definition 8.4.1 (Partially correct program unit). *Let m_{top} be a module defining a program unit $U = program\_unit(m_{top})$. U is partially correct iff m_{top} is partially correct and $\forall m \in depends(m_{top})$, m is partially correct in context w.r.t. the sets of initial queries induced by the initial queries to m_{top}.*

8.4.1 Verifying a program unit with intermodular analysis information

In this section we describe the basic, naïve algorithm for checking assertions in a completely implemented program unit. This procedure fully exploits the modularity of the program, and it is depicted in Algorithm 1. We assume that before checking assertions, an intermodular analysis fixed point is computed. This step aims at obtaining correct and precise analysis information for the exported predicates of every module in the program, and it is, to some extent, a source of inefficiency that we will deal with in next sections. In the second step, the algorithm selects every module one by one, and checks both local assertions and assertions for predicates imported from other modules. Observe that before the checking, an analysis must be performed for each individual module, in order to find call patterns[†] and success patterns for non-exported predicates, as this information is not captured in the GAT.[‡] We use the notation $check\_assertions(m, LAT)$ to denote activation of checking assertions in module m, w.r.t. the analysis information contained in a local answer table LAT.

The algorithm verifies whether each module in the input program unit is correct in context. Therefore, Algorithm 1 checks if a program unit is correct.

[†]Notice that predicates exported by a module m may be called from inside m as well as from modules importing m.

[‡]Local analysis results could be stored on disk as well as the GAT for efficiency purposes. However, it is conceptually like performing the local analysis of each module again in the checking step.

Algorithm 1 Checking assertions with intermodular analysis information

```
Input: top module $m_{top}$
Input: a global answer table $GAT = modular\_analysis(m_{top})$
Output: Warning/Error messages, new status in the assertions in $program\_unit(m_{top})$
  for all $m \in program\_unit(m_{top})$ do
    $LAT_m := analysis(m, \mathcal{CP}_m^{GAT}, GAT)$
    $check\_assertions(m, LAT_m)$
  end for
```

Correctness of checking is guaranteed for any success policy used during the intermodular analysis. We can state the following results:

Proposition 8.4.2. *Let m_{top} be a module defining a program unit $U = program\_unit(m_{top})$. Let A be an assertion in U. If Algorithm 1 decides that A is abstractly true (resp. checked or false) then A is true (resp. checked or false).*

Thus, in consequence, Algorithm 1 can be viewed as a sufficient condition for partial correctness of a program unit. Finally, we can also state a result concerning the accuracy of Algorithm 1, as compared with the flattening approach to modular analysis, since the results produced by $modular\_analysis_{SP^-}(m)$ are as precise as those obtained by the analysis of $flatten(m)$ [PCH$^+$04].

Proposition 8.4.3. *Let $LAT = analysis(flatten(m_{top}), \mathcal{CP}_{m_{top}}^{Asst}, \emptyset)$. The output of (1) Algorithm 1 run with input m_{top} and $GAT = modular\_analysis_{SP^-}(m_{top})$ is the same as the output of (2) $check\_assertions(flatten(m_{top}), LAT)$ with the exception that some assertions which are marked as checked by (2) may be marked as true by (1).*

Note that the use of SP^- policy is only required when there are cycles in the intermodule dependency graph (in order to reach the least fixed point, see Chapter 5).

Algorithm 2 Checking assertions without modular analysis

```
Input: top-level module $m_{top}$
Output: Warning/Error messages, new status in the assertions in $program\_unit(m_{top})$
  for all $m \in program\_unit(m_{top})$ do
    $LAT_m := analysis_{SP^+}(m, \mathcal{CP}_m^{Asst}, \mathcal{AT}_m^{Asst})$
    $check\_assertions(m, LAT_m)$
  end for
```

8.4.2 Verifying a program unit with no intermodular analysis information

As explained in the previous section, every assertion A of the form `check calls` $P : \lambda_{Pre} \in assertions(m)$ where $P \in exported\_preds(m)$ is verified in every module that imports P from m. If such calls assertions are abstractly true in all importing modules (i.e., for every call pattern CP found in a module importing P we have that $CP \sqsubseteq \lambda_{Pre}$), then that means that λ_{Pre} approximates all possible calling patterns to P from outside m. Therefore, the `calls` assertions can be used as starting points for analyzing every module in the program unit for checking the assertions. This leads us to a scenario for checking assertions, shown in Algorithm 2, where no prior intermodular analysis is required, and which aims at proving every module to be conditionally correct rather than correct in context.

Observe that Algorithm 2 does not use the modular analysis results as input. Instead, `pred` assertions of exported predicates are taken as input to the single-module analysis phase, $\mathcal{CP}_m^{Asst}$. A similar policy is applied when collecting success patterns of imported predicates.

This scenario can be viewed as proving conditional correctness of each module $m \in program\_unit(m_{top})$, where the conditions are the corresponding `pred` assertions from imported modules, as stated in Proposition 8.3.5. On the other hand, since we check all the modules in the program unit, and the program unit is self-contained, the `pred` assertions from imported modules are also the subject of checking. Assume that after checking all the modules in $program\_unit(m_{top})$ all the `pred` assertions get status `checked` or `true`.[§] This means that for ev-

[§]In this case the `calls` part originated from the `pred` assertion receives status `checked`, and ery exported/imported predicate P, the analysis information $P : CP \mapsto AP$ generated when analyzing individual modules satisfies the checking conditions of Propositions 8.2.6 and 8.2.8. Thus, the following result holds:

Proposition 8.4.4. *Let m_{top} be a module defining a program unit $U = program\_unit(m_{top})$. If each module $m \in U$ is conditionally partially correct, and m_{top} is partially correct, then U is partially correct.*

If the assertions get true or checked using Algorithm 2, it is easy to see that they would also get true or checked if the (full) modular analysis were used, as modular analysis computes the least fixed point, i.e., it returns the most accurate analysis information. Consequently, if the `calls` assertions receive status `checked` and the `success` assertions receive status `true` when checking with Algorithm 2, there is no need to run a costly modular analysis.

8.4.3 Interleaving analysis and checking

Algorithm 2 may not be able to determine that a program unit is partially correct if the user has provided either too few assertions for exported predicates or they are not accurate enough. In this case we have to replace information from the missing assertions and to incorporate a certain degree of automatic propagation of call/success patterns among modules during the checking process. The basic idea is to interleave analysis and compile-time checking during modular analysis. The main advantage of this approach is that errors will be detected as soon as possible, without computing an expensive intermodular fixpoint, yet having call and success patterns being propagated among modules. The whole process terminates as soon as an error is detected or when the modular analysis fixed point has been reached, as shown in Algorithm 3. Concrete procedures in steps 1 and 2 depend on a specific intermodular analysis algorithm, success and entry policies, etc. Note that in Algorithm 3 every module is analyzed for $\mathcal{CP}_m^{GAT}$, the set of all call patterns for a module m in the GAT.[¶]

the `success` part status `true`.

[¶] $\mathcal{CP}_m^{GAT}$ is used for simplicity of the presentation. In the actual implementation the modules are analyzed just for the *marked* entries, and only the assertions related to those entries are checked.

Algorithm 3 Interleaving analysis and checking

```
Input: top module $m_{top}$
Output: $GAT$, Warning/Error messages, new status in the assertions in $program\_unit\ (m_{top})$
  Set initial $GAT$ with marked entries for call patterns from $\mathcal{CP}^{Asst}_{m_{top}}$
  while there are modules with marked entries in $GAT$ do
1     select module $m$
      $LAT_m := analysis_{SP}(m, \mathcal{CP}^{GAT}_m, GAT)$
      $check\_assertions(m, LAT_m)$
      if an error is detected in $m$ then
         STOP
      end if
2     update $GAT$ with $LAT_m$
  end while
```

If an SP^+ success policy is used in Algorithm 3, then $LAT^1_m \succeq LAT^2_m \succeq \cdots \succeq LAT^n_m$ (see Chapter 3), where LAT^n_m coincides with the analysis results of module m when the intermodular fixed point has been reached, and each of the LAT^i_m corresponds to the status of the analysis answer table for m at every iteration of the algorithm that schedules m for analysis.

Proposition 8.4.5. *Let LAT_m be an answer table for module m. If an assertion is abstractly checked (resp. abstractly true or abstractly false) w.r.t. LAT_m it will also be abstractly checked (resp. abstractly true or abstractly false) w.r.t. any answer table LAT'_m s.t. $LAT'_m \preceq LAT_m$.*

Thus, the conclusions drawn about the assertions are sound in the following sense: if an assertion is detected to be checked or false in an intermediate step, it will surely remain checked or false at the end of the process. If the assertion is not yet proved not disproved, its status might change in the subsequent steps as the analysis information might be more accurate in future iterations. This observation allows us to define an alternative termination condition of Algorithm 3. The algorithm terminates with no error messages as soon as all the assertions in the program unit obtain status `checked` or `true`.

Algorithm 3 can be adapted to apply the SP^- success policy. The sequence of answer tables generated during the analysis using that policy is now $LAT^1_m \preceq$

$LAT^2_m \preceq \cdots \preceq LAT^n_m$, where only LAT^n_m, i.e. the one corresponding with the global fixpoint, is guaranteed to safely approximate the module's semantics. The following proposition holds.

Proposition 8.4.6. *Let LAT_m be an answer table for module m. If an assertion A is not abstractly checked w.r.t. LAT_m, then $\forall LAT'_m$ s.t. $LAT_m \preceq LAT'_m$, A will not be abstractly checked w.r.t. LAT'_m.*

Therefore, in this case the following conclusions can be made about the final status of assertions: if at any intermediate step the status of an assertion remains as `check` or becomes `false`, it will at most be `check` at the end of the whole process. Therefore, Algorithm 3 must stop and issue an error as soon as `false` or `check` assertions are detected (instead of stopping only when there are `false` assertions, as above).

Sufficient condition for partial correctness follows:

Proposition 8.4.7. *Let m_{top} be a module defining a program unit U = program_unit(m_{top}). If Algorithm 3 terminates without issuing error messages, then (1) if SP^+ is used and Algorithm 3 decides that an assertion A is abstractly true (resp. checked), then A is true (resp. checked); and (2) if SP^- is used then all assertions in U are checked.*

8.4.4 An example

Let us illustrate the process of checking assertions in a program unit. Consider the following main module `bdays` that identifies boring days as days when boring things have to be done.

```
:- module(bdays,[boring_day/1],[assertions]).
:- use_module(classify_days).

:- check pred boring_day(D) => tworkingday(D).    % (1)

boring_day(Day) :- is_boring(ToDo), what_to_do(Day,ToDo).

is_boring(teaching).
is_boring(faculty_meeting).
```

The above module imports from the `classify_days` module predicate `what_to_to/2` and the regular type `tworkingday`, used in assertion `(1)`.

```
:- module(classify_days,
     [what_to_do/2,tweekday/1,tactivity/1,tworkingday/1],
     [assertions, regtypes]).

:- regtype tweekday/1.
tweekday(mon).  tweekday(tue).  tweekday(wed).
tweekday(thu).  tweekday(fri).  tweekday(sat).
tweekday(sun).

:- regtype tworkingday/1.
tworkingday(mon).  tworkingday(tue). tworkingday(wed).
tworkingday(thu).  tworkingday(fri).

:- regtype tactivity/1.
tactivity(hacking).  tactivity(teaching).  tactivity(research).
tactivity(pub).  tactivity(nap).  tactivity(faculty_meeting).

:- check pred what_to_do(A,B) => (tweekday(A), tactivity(B)).  % (2)
what_to_do(mon,hacking).
what_to_do(tue,teaching).
what_to_do(wed,teaching).
what_to_do(thu,faculty_meeting).
what_to_do(fri,research).
what_to_do(sat,pub).
what_to_do(sun,nap).
```

Assume that we would like to check assertions in the above program unit. We will do it first without intermodular analysis, i.e. following Algorithm 2. The modules are analyzed, with type analysis, one by one, and the only information being propagated between the modules is related to assertions for exported predicates. During the analysis the variable `Day` in the clause defining `boring_day/1` obtains an abstract value (a type) `tweekday`, as this value has been propagated from the imported assertion `(2)` for `what_to_do/2`. Nothing more precise can be inferred in this scenario. Consequently assertion (1) in the main module remains check[||]. Assertion (2) obtains always status checked.

In order to validate assertion (1) in the module day, more accurate success patterns of what_to_do/2 must be computed, which essentially means that Algorithm 1 or 3 has to be applied. In this case those success patterns of what_to_do/2 found by the intermodular analyzer are used rather than information extracted from the assertion. Thus the variable Day takes an abstract value rt10, where rt10 is a regular type newly generated by the analyzer:

```
:- regtype rt10/1.
rt10(thu).
rt10(tue).
rt10(wed).
```

Since rt10 is a subtype of tworkingday assertion (2) obtains status checked. Note that this reasoning cannot be easily performed without intermodular analysis, even if we use more expressive type language in the assertions.

8.5 Implementation issues

This assertion checking schema has been implemented on CiaoPP, the Ciao System preprocessor [HPBLG05]. The different scenarios for compile-time assertion checking that have been detailed in previous sections determine the final implementation of the assertion system. In the following sections, the most relevant factors are revised, and the implementation with the interface is discussed.

8.5.1 Modules checked

The first question to be answered by the user concerns which modules should be subject to assertion checking. As we have already seen in Section 8.3.1, the programmer that develops a single module might be interested in checking assertions only in that module, possibly because other modules are not completely implemented yet, or because assertions in those modules have been already verified. The purpose of checking the assertions of a module in isolation is in general to

[||]A warning message is issued to the user.

check all the assertions in that module and the calls to imported predicates (as in propositions 8.3.3 and 8.3.5).

On the other hand, modifying a module may affect other modules which import the changed module. That might happen if call patterns for imported predicates or success patterns for exported predicates rise errors which are propagated to the related modules and prevent corresponding assertions from being true or checked. Of course, the user can also select an arbitrary subset of modules to be verified, or the whole program unit.

System libraries are excluded by default from the checking process. Nevertheless, the assertions regarding exported library predicates will be exported along with the predicates, in order to verify whether those predicates are used as expected. Checking assertions for the exported predicates of library modules has been found very useful, since even user programs with no assertions can be checked with respect to the assertions written in the system libraries.

Example 8.5.1. *The following module computes the average of a list of numbers:*

```
:- module(average,[avg/2],[assertions]).
:- use_module(lists,[length/2]).
:- use_module(numlists,[sum/2]).

:- pred avg(L,A) : list(L,num) => num(A).

avg(L,A):-
    length(L,N), sum(L,S), A is S / N.
```

Since `length/2` *and* `sum/2` *are imported from* `lists` *and* `numlists`*, respectively, their code is not accessed by the analyzer when analyzing* `average` *separately. If there is no information available for those predicates, when analyzing* `avg/2` *with an SP^+ success policy no useful information should be produced.*

However, since `is/2` *is also a predicate imported by default from the* `arithmetic` *system library, the assertions related to it are taken into account. In such library there exists the following assertion:*

```
:- pred is(X,+Y) : arithexpression(Y) => num(X).
```

After checking `average`*, the system issues the following warning message:*

```
WARNING (ctchecks_messages): (lns 6-10)
 at literal 3 not verified calls assertion:
   :- calls arithmetic:is(X,Y)
      : [[arithmetic:arithexpression(Y)]]
 because on call arithmetic:is(term,^ (term/term))
```

containing both expected properties extracted from the assertion and the analysis information at the call to `is/2`*. Observe that there is no specific information available for variables* `S` *and* `N`[**] *when the goal* `A is S / N` *is analyzed and therefore it cannot be guaranteed that* `S / N` *is actually an arithmetic expression. But the assertion for* `is/2` *still allows checking the assertion in* `average`*, since the assertion for* `is/2` *states that* `is/2` *will always return a number in its first argument upon success.*

8.5.2 Modules analyzed

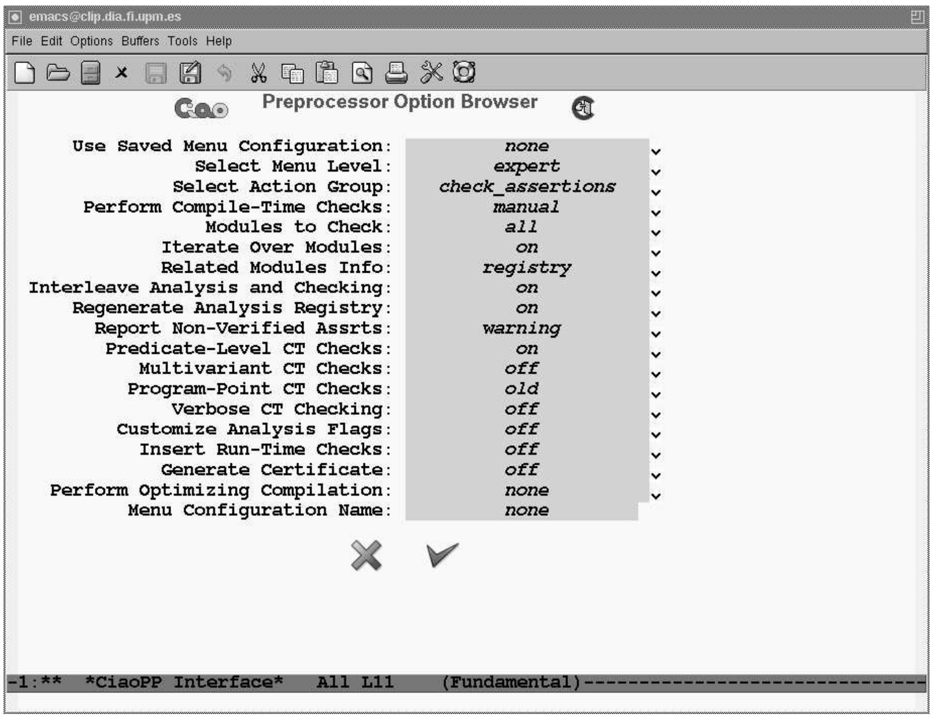
The analysis of the code that must be performed prior to checking the assertions is another factor that can yield different implementations. As we have already explained, when a single module is to be checked, the analysis phase can produce improved results if an intermodular analysis of the whole program unit is performed. As is pointed out in Proposition 8.3.3, more accurate analysis results may produce more assertions to become checked. This can happen not only when an intermodular fixed point has been reached. An interesting intermediate scenario is depicted for analysis in [PCH$^+$04], in which the modules are analyzed when their programmers request it, reusing the results from previous analyses of related modules in a shared GAT. This global table can be safely used (using an SP^+ success policy) at any time for checking a module.

If the rest of the modules in the program are not finished yet, or the user is confident that the inter-modular analysis does not contribute with any new information (for example because the assertions already set in other modules are very accurate), the modular analysis phase might be dropped completely. The inter-modular analysis may not be necessary even when all modules are to be checked: if the `pred` assertions are precise enough, the program can be verified by checking its modules one by one, as shown in Section 8.4.2.

[**]In the abstract domain used for this example, $\top$ is represented by `term`.

8.5.3 The interface

An interface to the CiaoPP system has been implemented as a part of Emacs Ciao mode. Figure 8.1 shows the interface in the state when the user is supposed to decide values of system parameters that determine the working mode and specific tasks performed by CiaoPP.



Preprocessor Option Browser

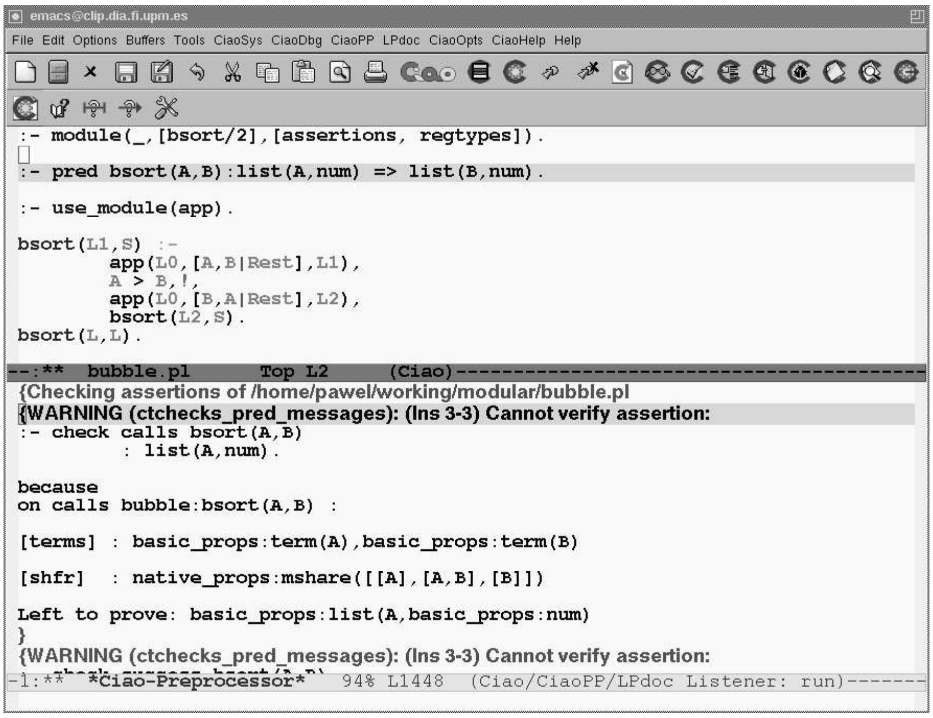
Use Saved Menu Configuration:	none
Select Menu Level:	expert
Select Action Group:	check_assertions
Perform Compile-Time Checks:	manual
Modules to Check:	all
Iterate Over Modules:	on
Related Modules Info:	registry
Interleave Analysis and Checking:	on
Regenerate Analysis Registry:	on
Report Non-Verified Assrts:	warning
Predicate-Level CT Checks:	on
Multivariant CT Checks:	off
Program-Point CT Checks:	old
Verbose CT Checking:	off
Customize Analysis Flags:	off
Insert Run-Time Checks:	off
Generate Certificate:	off
Perform Optimizing Compilation:	none
Menu Configuration Name:	none

```
-1:**  *CiaoPP Interface*   All L11    (Fundamental)------------------------------
```

Figure 8.1: An Interface.

We show the questions typically posted in the so-called *expert mode*. An alternative option is the *naive mode* in which CiaoPP takes default parameters values appropriate for a given task decided by the user in the **Action Group** option. It is set to `check_assertions`, as CiaoPP, besides checking assertions, can be used for other purposes like analyzing or specializing programs. The `manual` value in **Perform Compile-Time Checks** option indicates that the user can customize analysis options. Otherwise (`auto` value), the system, based on assertions found in the program, will determine automatically abstract domains needed for the checking process.

The next interesting option is **Modules to Check**. This one directly corre-


emacs@clip.dia.fi.upm.es
File Edit Options Buffers Tools CiaoSys CiaoDbg CiaoPP LPdoc CiaoOpts CiaoHelp Help

```
:- module(_,[bsort/2],[assertions, regtypes]).

:- pred bsort(A,B):list(A,num) => list(B,num).

:- use_module(app).

bsort(L1,S) :-
        app(L0,[A,B|Rest],L1),
        A > B,!,
        app(L0,[B,A|Rest],L2),
        bsort(L2,S).
bsort(L,L).
--:**  bubble.pl       Top L2      (Ciao)----------------------------------------
{Checking assertions of /home/pawel/working/modular/bubble.pl
{WARNING (ctchecks_pred_messages): (lns 3-3) Cannot verify assertion:
:- check calls bsort(A,B)
         : list(A,num).

because
on calls bubble:bsort(A,B)  :

[terms]  : basic_props:term(A),basic_props:term(B)

[shfr]   : native_props:mshare([[A],[A,B],[B]])

Left to prove: basic_props:list(A,basic_props:num)
}
{WARNING (ctchecks_pred_messages): (lns 3-3) Cannot verify assertion:
-1:**  *Ciao-Preprocessor*   94% L1448  (Ciao/CiaoPP/LPdoc Listener: run)-------
```

Figure 8.2: A warning message.

sponds to our discussion in Section 8.5.1 and can be set to values `all` or `curr_mod` (current module - a module in the current Emacs buffer). The next option **Iterate Over Modules** determines whether the checking algorithm should perform intermodular fixpoint analysis or not. The option **Related Modules Info** determines if the call (resp. success) patterns of exported (resp. imported) predicates are extracted from the assertions or from the *GAT* (the registry). If the answer to **Iterate Over Modules** is `off` then single traversal of all the modules in the program unit is performed. It results in Algorithm 1, if **Related Modules Info** is set to `registry` and intermodular fixpoint has been computed before, or it results in Algorithm 2 if **Related Modules Info** is set to `assertions`. If **Iterate Over Modules** is set to `on` then the interface displays another option **Interleave Analysis and Checking** which allows to accomplish Algorithm 3 of performing checking and intermodular analysis at the same time.

The remaining options displayed in Figure 8.1 control less importatant aspects of checking process or are simply irrelevant for our purpose.

Figure 8.2 demonstrates a warning message displayed by the interface. In the bottom Emacs buffer we can see the message itself, containing information about the assertion that could not be proved nor disproved, and the analysis information obtained for call of `bsort/2` for the two different abstract domains `terms` and `shfr`. Note that the message also contains a line saying which of the assertion properies have not been proved yet. The top buffer displays the corresponding file with the assertion highlighted.

8.6 Conclusions

Algorithms 1 and 2 have different levels of accuracy, computing cost, and verification power. The advantages of Algorithm 2 are that it is potentially more accurate and it does not impose any burden on the user, since no assertions are compulsory. On the other hand, Algorithm 1 has low computing cost, since modules only need to be analyzed once and it can be applied to incomplete programs. All this at the price of a development policy where module interfaces are accurately described using assertions.

Comparing this chapter with related work, the scenario described in Section 8.4.2 can be seen as an instance of the analysis with user-provided interface of [CC02a]. Our goal is however different than theirs: instead of computing the most precise analysis information we try to prove or disprove assertions, which makes this method more related in fact to the one of [CLV98], focused on program verification. Nevertheless, unlike [CLV98] we do not require the user to provide a complete specification, specially in Algorithm 2 –the missing parts are either described by topmost values or infered by the interleaved analysis algorithm.

Part IV

Applications

This final part of the thesis consists of two applications of program analysis to the transformation and specialization of real-life programs, respectively.

In the first case, an application of modular program analysis of special interest is the possibility of using this technique in programs with interfaces with external systems from which the source code is not available. There are several reasons for not having the code available to the analyzer, and among them, it is particularly important the use of interfaces to other languages. It is of special relevance the case of accessing to persistent information, and specifically the access to relational database systems through relational languages: its own structure, from the point of view of the analysis, brings a great deal of information (types of the arguments, termination, determinism, modes and instantiation level of arguments, etc.). The next chapter describes a specific implementation developed in the Ciao system to provide persistent storage capabilities, transparent to the underlying storage method. From the point of view of the Ciao language, the basic constructs for accessing and modifying data in the persistent storage are seen as simple procedure calls. That means that they can be considered just like calls to imported procedures, and therefore modular analysis and assertion checking algorithms can work seamlessly in that scenario. Persistent predicates are declared by means of the assertion language already described in Part III. The aim for using the assertion language is in this case twofold: on one hand, it declares which procedures are to reside in a persistent location; on the other hand, it facilitates the use of global preprocessing tools like analyzers and compile-time checkers for programs with persistent information and incomplete code, and provides additional type information about the data stored in persistent procedures. As a first approach, a simple implementation based on plain files is introduced, and then an implementation based on external relational databases is presented and global analysis results used for optimizing programs using that implementation.

In the second case, a preliminary modular specialization algorithm is developed in order to reduce the size of code in libraries for a given program. This algorithm is based in the non-modular multiple specialization algorihm of [PH03], and extends it by propagating specialized versions of predicates across the modular graph. In this case, it has been applied to strip-down the code of the libraries used by a program, in order to reduce the total size of the object code and make them fit in a small device. Program specialization has been successfully applied to user programs, but it has not been directly used for system libraries yet. This approach is specially relevant for pervasive systems, as it allows the use of generic libraries for developing applications on virtual machine runtime-based systems. It presents important differences with respect to current runtime systems for pervasive devices (like Java Micro Edition series of runtime environments), as they have a prefixed set of runtime libraries specific for such systems.

Some results have been obtained, showing that an important reduction in runtime library size has been achieved using a simple abstract domain. It is expected that the application of more complex abstract domains will provide even better results. This work is described in the second chapter of this part.

Chapter 9

Optimization of Interfaces to Persistent Information

This chapter describes a model of persistence in (C)LP languages and two different and practically very useful ways to implement this model in current systems. The fundamental idea is that persistence is a characteristic of certain dynamic predicates (i.e., those which encapsulate state). The main effect of declaring a predicate persistent is that the dynamic changes made to such predicates *persist* from one execution to the next one. After proposing a syntax for declaring persistent predicates, a simple, file-based implementation of the concept is presented and some examples shown. An additional implementation is presented which stores persistent predicates in an external database. The abstraction of the concept of persistence from its implementation allows developing applications which can store their persistent predicates alternatively in files or databases with only a few simple changes to a declaration stating the location and modality used for persistent storage. This chapter presents the model, the implementation approach in both the cases of using files and relational databases, a number of optimizations of the process (using information obtained from static global analysis and goal clustering), and performance results from an implementation of these ideas.

9.1 Introduction

State is traditionally implemented in Prolog and other (C)LP systems through the built-in ability to modify predicate definitions dynamically at runtime.[*] Generally, fact-only dynamic predicates are used to store information in a way that provides global visibility (within a module) and preserves information through backtracking. This internal rule database, albeit a non-declarative component of Prolog, has practical applications from the point of view of the needs of a programming language.[†]

However, Prolog internal rule database implementations associate the lifetime of the internal state with that of the process, i.e., they deal only with what happens when a given program is running and changes its private rule database. Indeed, the Prolog rule database lacks an important feature: data persistence. By data persistence we refer to rule database modifications surviving across program executions (and, as a later evolution, maybe being accessible to other programs –even concurrently). This feature, if needed, must be explicitly implemented by the programmer in traditional systems.

In this chapter we present a conceptual model of persistence by proposing the concept of *persistent predicates*, and a number of implementations thereof. A persistent predicate is a special kind of dynamic, data predicate that "resides" in some persistent medium (such as a set of files, a database, etc.) and which is typically external to the program using such predicates. The main effect is that any changes made to a persistent predicate from a program "survive" across executions , i.e., if the program is halted and restarted the predicate that the new process sees is in precisely the same state as it was when the old process was halted (provided no change was made in the meantime to the storage by other processes or the user). Notably, persistent predicates appear to a program as ordinary dynamic predicates: calls to these predicates can appear in clause bodies in the usual way without any need to wrap or mark them as "external" or "database" calls and updates to persistent predicates can be made calling the

[*]In the ISO standard these predicates have to be marked explicitly as *dynamic*.

[†]Examples of recent proposals to extend its applicability include using it to model reasoning in a changing world [Kow96], and as the basis for communication of concurrent processes [CH99] and objects [PB02].

standard `asserta/1`, `assertz/1`, `retract/1`, etc. predicates used for ordinary dynamic predicates, but suitably modified. Updates to persistent predicates are guaranteed to be atomic and transactional, in the sense that if an update terminates, then the external storage has definitely been modified. This model provides a high degree of conceptual compatibility with previously existing programs which access only the local rule database,[‡] while bringing at the same time several practical advantages:

- The state of dynamic predicates is, at all times, reflected in the state of the external storage device. This provides security against possible data loss due to, for example, a system crash.

- Since accesses to persistent predicates are *viewed* as regular accesses to the Prolog rule database, analyzers (and related tools) for full Prolog can deal with them in the same way as with the standard dynamic predicates, resulting in a series of optimizations, some of which will be shown. Using explicit accesses to files or external databases through low-level library predicates would make this task much more difficult.

Finally, perhaps the most interesting advantage of the notion of persistent predicates is that it abstracts away how the predicate is actually stored. Thus, a program can use persistent predicates stored in files or in external relational databases interchangeably, and the type of storage used for a given predicate can be changed without having to modify the program except for replacing a single declaration in the whole program. The program always contains standard internal database access and aggregation predicates, independently of whether the storage medium is the internal Prolog rule database, file-based, or database-based. It also minimizes impact on the host language, as the semantics of the access to the rule database is compatible with that of Prolog.

Our approach builds heavily on the well known and close relationship between (Constraint) Logic Programming and relational databases [Ull90]: for example, operations in the relational algebra can be easily modeled using Horn clauses (plus negation for some operations), where database tables are seen as fact-only

[‡]The "logical view" of updates [LO87] is not enforced in the case of using a relational database as storage, in the same way as with concurrent data predicates [CH99].

predicates, and every record is seen as a fact. On the other hand, the embedding into Prolog allows combining full Prolog code (beyond DATALOG) with the accesses to the persistent predicates.

A number of current Prolog systems offer external database interfaces, but often with ad-hoc access builtins. In those cases in which some kind of transparency is provided (e.g. Quintus *ProDBI*, SICStus and LPA *Prodata*, ECLiPSe), the system just allows performing queries on tables as if they were Prolog predicates, but does not allow updating tables using the same transparent approach. We argue that none of these cases achieve the same level of flexibility and seamless integration with Prolog achieved in our proposal.

Implementations of this model have been used in real-world applications such as the Amos tool (see `http://www.amosproject.org`), part of a large, ongoing international project aimed at facilitating the reuse of Open Source code by means of a powerful, ontology-based search engine working on a large database of code information.

9.2 Persistent Predicates in Prolog

We will now define a syntax for the declaration of persistent predicates. We will also present briefly two different implementations of persistent predicates which differ on the storage medium (files of Prolog terms in one case, and an external relational database in the other). Both implementations aim at providing a semantics compatible with that of the Prolog internal rule database, but enhanced with persistence over program executions.

9.2.1 Declaring Persistent Predicates

The syntax that we propose for defining persistent predicates is based on the assertion language of Ciao Prolog [PBH00c, HPBLG03b], which allows expressing in a compact, uniform way, types, modes, and, in general, different (even arbitrary) properties of predicates.

In order to specify that a predicate is persistent we have to flag it as such, and also to define where the persistent data is to be stored. Thus, a minimum declaration is:

```
:- include(library(persdb)).

:- pred employee/3 + persistent(payroll).
:- pred category/2 + persistent(payroll).

:- persistent_db(payroll, file('/home/clip/accounting')).
```

The first declaration states that the persistent database library is to be used to process the source code file: the `include`d code loads the `persdb` library support predicate definitions, and defines the local operators and syntactic transformations that implement the persdb package. The second and third line state that predicates `employee/3` and `salary/2` are persistent and that they live in the storage medium to be referred to as `payroll`, while the fourth one defines which type of storage medium the `payroll` identifier refers to.[§] It is the code in the persdb package that processes the `persistent/1` and `persistent_db/2` declarations, and which provides the code to access the external storage and keeps the information necessary to deal with it. In this particular case, the storage medium is a disk file in the directory specified in the directive. The predicates in Figure 9.2 use these declarations to compute the salary of some employee, and to increment the number of days worked:

```
salary(Emp,Salary):-
    employee(Emp,Cat,Days),
    category(Cat,PerDay),
    Salary is Days * PerDay.
```

```
one_more_day(Emp):-
    retract(employee(Emp,Cat,Days)),
    Days1 is Days + 1,
    assert(employee(Emp,Cat,Days1)).
```

Figure 9.1: Accessing and updating a persistent predicate

If the external storage is to be kept in an SQL database, argument type information is required in order to create the table (if the database is empty) and also to check that the calls are made with compatible types. It is also necessary to establish a mapping (views) between the predicate functor and arguments

[§]The `persistent_db/2` information can also be included in the argument of `persistent`, but using `persistent_db/2` declarations allows factoring out information shared by several predicates.

and table name and columns. In this example, suitable declarations are:

```
:- include(library(persdb)).

:- pred employee/3 :: string * string * int +
        persistent(employee(ident, category, time), payroll).
:- pred category/2 :: string * int          +
        persistent(category(category, money), payroll).

:- persistent_db(payroll, db(paydb, admin, 'Pwd', 'db.comp.org')).
```

The `db/4` structure indicates database name (`paydb`), database server (`db.comp.org`), database user (`admin`) and password (`Pwd`). This information is processed by the persdb package, and a number of additional formats can be used. For example, the port for the database server can be specified (as in `'db.comp.org':2020`), the precise database brand can be noted (as, for example `odbc/4` or `oracle/4` instead of the generic `db/4`), etc. This instructs the persdb package to use different connection types or to generate queries specialized for particular SQL dialects. In addition, values for the relevant fields can also be filled in at run time, which is useful for example to avoid storing sensitive information, such as password and user names, in program code. This can be done using hook facts or predicates, which can be included in the source code, or asserted by it, perhaps after consulting the user. These facts or predicates are then called when needed to provide values for the arguments whose value is not specified in the declaration. For example, a declaration such as:

```
:- persistent_db(payroll, db(paydb, puser/1, ppwd/1, 'db.comp.org')).
```

would call the hook predicates `puser/1` and `ppwd/1`, which are expected to be defined as `puser(User):- ...` and `ppwd(Password):- ...`.

Note also that, as mentioned before, the declarations corresponding to `employee/3` and `category/2` specify the name of the table in the database (which can be different from that of the predicate) and the name of each of its columns. It may also have a type signature. If a table is already created in the database, then this declaration of types is not strictly needed, since the system will retrieve the schema from the database. However, it may still be useful so that (compile-time or run-time) checking of calls to persistent predicates can be performed. Furthermore, types and modes can be read and inferred by a global analysis tool, such as, e.g., CiaoPP [HPBLG03b, HBPLG99], and used to optimize the generation of SQL expressions and to remove superfluous runtime checks at compile time (see Section 9.2.3).

A dynamic version of the `persistent` declaration exists, which allows defining new persistent predicates on the fly, under program control. Also, in order to provide greater flexibility, lower-level operations (of the kind available in traditional Prolog-SQL interfaces) are also available, which allow establishing database connections manually. These are the lower-level library operations the above examples are compiled into. Finally, a persistent predicate can also be made to correspond to a complex view of several database tables. For further illustration, Figure 9.2 shows an example queue elements are kept as persistent data facts so that the program state can be recovered in subsequent executions.

9.2.2 File-Based Implementation

The file-based implementation of persistent predicates provides a light-weight, simple, and at the same time powerful form of persistence. It has the advantage of being standalone in the sense that it does not require any external support other than the file management capabilities provided by the operating system: these persistent predicates are stored in files under direct control of the persistent library. This implementation is especially useful when building small to medium-sized standalone (C)LP applications which require persistent storage and which may have to run in an environment where the existence of an external database manager is not ensured. Also, it is very useful even while developing applications which will connect to databases, because it allows working with persistent predicates maintained in files when developing or modifying the code and then switching to using the external database for testing or "production" by simply changing a declaration.

The implementation pursues at the same time efficiency and security. Each predicate uses three files: the *data file*, which stores a *base state* for the predicate; the *operations file*, which stores the differential between the base state and the predicate state in the program (i.e., operations pending to be integrated into the data file); and the *backup file*, which stores a security copy of the data file. Such files, in plain ASCII format, can be edited by hand using any text editor, or even easily read and written by other applications.

Program	**Execution**

```
:- module(queue, [main/0]).
:- include(library(persdb)).

:- pred queue/1 +
   persistent(file('/tmp/queue')).

main:-
   write('Action:'),
   read(A),
   handle_action(A),
   main.

handle_action(halt) :-
   halt.
handle_action(in(Term)) :-
   assertz(queue(Term)).
handle_action(out) :-
   (  retract(queue(Term))
   -> write('Out '), write(Term)
   ;  write('EMPTY!') ), nl.
handle_action(list) :-
   findall(T,queue(T),Contents),
   write('Contents: '),
   write(Contents),nl.
```

```
$ ./queue
Action: in(first).
Action: in(second).
Action: list.
Contents: [first, second]
Action: halt.

$ ./queue
Action: out.
Out first
Action: list.
Contents: [second]
Action: out.
Out second
Action: out.
EMPTY!
Action: halt.
```

Figure 9.2: Queue example and execution trace

When no program is accessing the persistent predicate (because, e.g., no program updating that particular predicate is running), the data file reflects exactly the facts in the Prolog internal rule database. When any insertion or deletion is performed, the corresponding change is made in the Prolog internal rule database, and a record of the operation is *appended* to the operations file. In this moment the data file does not reflect the state of the internal Prolog rule database, but it can be reconstructed by applying the changes in the operations file to the state in the data file. This strategy incurs only in a relatively small, constant overhead per update operation (the alternative of keeping the data file always up to date would lead to an overhead linear in the number of records in it).

When a program using a file-based persistent predicate starts up, the data file is first copied to a backup file (preventing data loss in case of system crash during this operation), and all the pending operations are performed on the data file by loading it into memory, re-executing the updates recorded in the operations file, and saving a new data file. The order in which the operations are performed and the concrete O.S. facilities (e.g., file locks) used ensure that even if the process aborts at any point in its execution, the data saved up to that point can be completely recovered upon a successful restart. The data file can also be explicitly brought up to date on demand at any point in the execution of the program.

9.2.3 External Database Implementation

We present another implementation of persistent predicates which keeps the storage in a relational database. This is clearly useful, for example, when the data already resides in such a database, the amount of data is very large, etc. A more extensive description of this interface can be found in [CCG+98, CHGT98].

One of the most attractive features of our approach is that this view of external relations as just another storage medium for persistent predicates provides a very natural and transparent way to perform simple accesses to relational databases from (C)LP programs. This implementation allows reflecting selected columns of a relational table as a persistent predicate. The implementation also provides facilities for reflecting complex views of the database relations as individual persistent predicates. Such views can be constructed as conjunctions, disjunctions or projections of database relations.

The architecture of the database interface (Figure 9.3), has been designed with two goals in mind: simplifying the communication between the Prolog side and the relational database server, and providing platform independence, allowing inter-operation when using different databases.

The interface is built on the Prolog side by stacking several abstraction levels over the socket and native code interfaces (Figure 9.3). Typically, database servers allow connections using TCP/IP sockets and a particular protocol, while

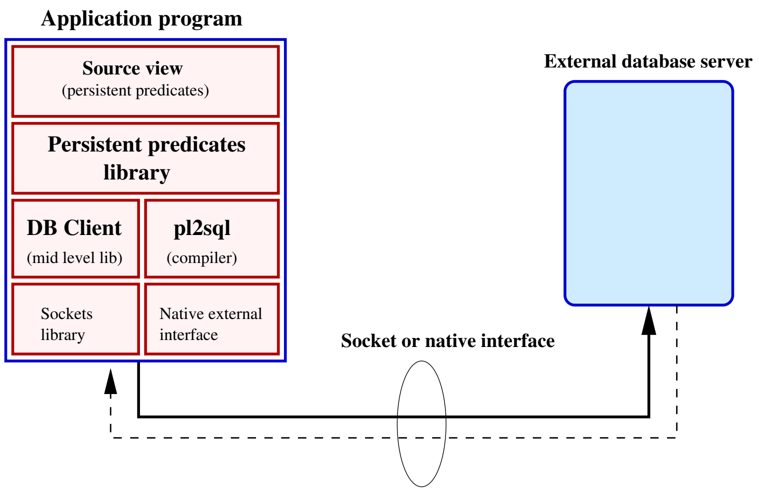


Figure 9.3: Architecture of the access to an external database

in other cases, linking directly a shared object or a DLL may be needed. For the cases where remote connections are not provided (e.g., certain versions of ODBC), a special-purpose mediator which acts as a bridge between a socket and a native interface has been developed [CCG+98, CHGT98]. Thus, the low level layer is highly specific for each database implementation (e.g. MySQL, Postgres, ORACLE, etc.). The mid-level interface (which is similar in level of abstraction to that present in most current Prolog systems) abstracts away these details.

The higher-level layer implements the concept of persistent predicates so that calls and database updates to persistent predicates actually act upon relations stored in the database by means of automatically generated mid-level code. In the base implementation, at compile-time, a “stub” definition is included in the program containing one clause whose head has the same predicate name and arity as the persistent predicates and whose body contains the appropriate mid-level code, which basically implies activating a connection to the database (logging on) if the connection is not active, compiling on the fly and sending the appropriate SQL code, retrieving the solutions (or the first solution and the DB handle for asking for more solutions, and then retrieving additional solutions on backtracking or eventually failing), and closing the connection (logging off the database), therefore freeing the programmer from having to pay attention to low-level details.

The SQL code in particular is generated using a Prolog to SQL translator based on the excellent work of Draxler [Dra91]. Modifications were made to the code of [Dra91] so that the compiler can deal with the different idioms used by different databases, the different types supported, etc. as well as blending with the high-level way of declaring persistence, types, modes, etc. that we have proposed (and which is in line with the program assertions used throughout in the Ciao system). Conversions of data types are automatically handled by the interface, using the type declarations provided by the user or inferred by the global analyzers.

In principle the SQL code corresponding to a given persistent predicate, literal, or group of literals needs to be generated dynamically at run-time for every call to a persistent predicate since the mode of use of the predicate affects the code to be generated and can change with each run-time call. Clearly, a number of optimizations are possible. In general, a way to improve performance is by reducing overhead in the run-time part of the Prolog interface by avoiding any task that can be accomplished at compile-time, or which can be done more efficiently by the SQL server itself. We study two different optimization techniques based on these ideas: the use of static analysis information to pre-compute the SQL expressions at compile time (which is related to adornment-based query optimization in deductive databases [RU93]), and the automatic generation of complex SQL queries based on Prolog query clustering.

Using static analysis information to pre-compute SQL expressions.

As pointed out, the computation of SQL queries can be certainly sped up by creating skeletons of SQL sentences at compile-time, and fully instantiating them at run-time. In order to create the corresponding SQL sentence for a given call to a persistent predicate at compile-time, information regarding the instantiation status of the variables that appear in the goal is needed. This mode information can be provided by the user by means of the Ciao assertion language. More interestingly, this information can typically be obtained automatically by using program analysis, which in the Ciao system is accomplished by **CiaoPP**, a powerful program development tool which includes a static analyzer, based on Abstract Interpretation [HPBLG03b, HBPLG99]. If the program is fed to CiaoPP, selecting the appropriate options, the output will contain, at every program point, the abstract substitution resulting from the analysis using a given domain. The essential information here is argument groundness (i.e., *modes*, which are computed using the sharing+freeness domain): we need to know which database columns must appear in the WHERE part of the SQL expression.

For example, assume that we have an database-based persistent predicate as in Section 9.2:

```
:- pred employee/3 :: string * string * int +
       persistent(employee(ident, category, time), payroll).
```

and consider also the program shown in the left side of Figure 9.2. The literal employee/3 will be translated by the persistence library to a mid-level call which will at run-time call the pl2sql compiler to compute an SQL expression corresponding to employee(Empl,Categ,Days) based on the groundness state of Empl, Categ and Days. These expressions can be precomputed for a number of combinations of the groundness state of the arguments, with still some run-time overhead to select among these combinations. For example, if the static analyzer can infer that Empl is ground when calling employee(Empl,Categ,Days), we will be able to build at compile-time the SQL query for this goal as:

```
SELECT ident, category, time FROM employee WHERE ident = '$Empl$';
```

The only task that remains to be performed at run-time, before actually querying the database, is to replace $Empl$ with the actual value that Empl is instantiated to and send the expression to the database server.

A side effect of (SQL-)persistent predicates is that they provide useful information which can improve the analysis results for the rest of the program: the assertion that declares a predicate (SQL-)persistent also implies that on success all the arguments will be ground. This additional groundness information can be propagated to the rest of the program. For instance, in the definition of salary/2 in Figure 9.2, category/2 happens to be a persistent predicate living in an SQL database. Hence, we will surely be provided with groundness information for category/2 so that the corresponding SQL expression will be generated at compile-time as well.

Query clustering.

The second possible optimization on database queries is query clustering. A simple implementation approach would deal separately with each literal calling a persistent predicate, generating an individual SQL query for every such literal.

Under some circumstances, mainly in the presence of intensive backtracking, the flow of tuples through the database connection generated by the Prolog backtracking mechanism will produce limited performance.

In the case of complex goals formed by consecutive calls to persistent predicates, it is possible to take advantage of the fact that database systems include a great number of well-developed techniques to improve the evaluation of complex SQL queries. The Prolog to SQL compiler is in fact able to translate such complex conjunctions of goals into efficient SQL code. The compile-time optimization that we propose requires identifying literals in clause bodies which call SQL-persistent predicates and are contiguous (or can be safely reordered to be contiguous) so that they can be clustered and, using mode information, the SQL expression corresponding to the entire complex goal compiled as a single unit. This is a very simple but powerful optimization, as will be shown.

For example, in predicate `salary/2` of Figure 9.2, assuming that we have analysis information which ensures that `salary/2` is always called with a ground term in its first argument, a single SQL query will be generated at compile-time for both calls to persistent predicates, such as:

```
SELECT ident, category, time, rel2.money
FROM employee, category rel2
WHERE ident = '$Empl$' AND rel2.category = category;
```

9.2.4 Concurrency and transactional behaviour

There are two main issues to address in these implementations of persistence related to transactional processing and concurrency. The first one is *consistency*: when there are several processes changing the same persistent predicate concurrently, the final state must be consistent w.r.t. the changes made by every process. The other issue is *visibility*: every process using a persistent predicate must be aware of the changes made by other processes which use that predicate. A further, related issue is what means exist in the source language to express that a certain persistent predicate may be accessed by several threads or processes, and how several accesses and modifications to a set of persistent predicates are grouped so that they are implemented as a single transaction.

Regarding the source language issue, the Ciao language already includes a way to mark dynamic data predicates as concurrent [CH99], stating that such predicates could be modified by several threads or processes. Also, a means has been recently developed for marking that a group of accesses and modifications to a set of dynamic predicates constitute a single atomic transaction [Pat04]. Space limitations do not allow describing locking and transactional behaviour in the implementation of persistent predicates proposed. The current solutions are outlined in [CGC+04b, Pat04] and these issues are the subject of future work.

9.3 Empirical results

We now study from a performance point of view the alternative implementations of persistence presented in previous sections. To this end, both implementations (file-based and SQL-based) of persistent predicates, as well as the compile-time optimizations previously described, have been integrated and tested in the Ciao Prolog development system [BCC+02].

9.3.1 Performance without Compile-time Optimizations

The objective in this case is to check the relative performance of the various persistence mechanisms and contrast them with the internal Prolog rule database. The queries issued involve searching on the database (using both indexed and non-indexed queries) as well as updating it.

The results of a number of different tests using these benchmarks can be found in Table 9.1, where a four-column, 25,000 record database table is used to check the basic capabilities and to measure access speed. Each one of the four columns has a different measurement-related purpose: two of them check indexed accesses —using `int` and `string` basic data types—, and the other two check non-indexed accesses. The time spent by queries for the different combinations are given in the rows *non-indexed numeric query*, *non-indexed string query*, *indexed numeric query*, and *indexed string query* (time spent in 1,000 consecutive queries randomly selected). Row *assertz* gives the time for creating the 25,000 record table by adding the tuples one by one. Rows *non-indexed numeric retract*, *non-indexed string retract*, *indexed numeric retract*, and *indexed string retract* provide the timings for the deletion of 1,000 randomly selected records by deleting the tuples one by one.

The timings were taken on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 1Gb of RAM memory, running Red Hat Linux 8.0, and averaging several runs and eliminating the best and worst values. Ciao version 1.9.78 and MySQL version 3.23.54 were used.

The meaning of the columns is as follows:

prologdb (data) Is the time spent when accessing directly the internal (assert/retract) state of Prolog.

prologdb (concurrent) In this case tables are marked as *concurrent*. This toggles the variant of the assert/retract database which allows concurrent access to the Prolog rule database. Atomicity in the updates is ensured and several threads can access concurrently the same table and synchronize through facts in the tables (see [CH99]). This measurement has been made in order to provide a fairer comparison with a database implementation, which has the added overhead of having to take into account concurrent searches/updates, user permissions, etc.¶

persdb This is the implementation presented in Section 9.2.2, i.e., the file-based persistent version. The code is the same as above, but marking the predicates as persistent. Thus, in addition to keeping incore images of the rule database, changes are automatically flushed out to an external, file-based transaction record. This record provides persistence, but also introduces the additional cost of having to save updates. The implementation ensures atomicity and also basic transactional behavior.

persdb/sql This is the implementation presented in Section 9.2.3, i.e., where all the persistent predicates-related operations are made directly on an external SQL database. The code is the same as above, but marking the predicates

¶Note, however, that this is still quite different from a database, apart, obviously, from the lack of persistence. On one hand databases typically do not support structured data, and it is not possible for threads to synchronize on access to the database, as is done with concurrent dynamic predicates. On the other hand, in concurrent dynamic predicates different processes cannot access the same data structures, which is possible in SQL databases. However, SQL databases usually use a server process to handle requests from several clients, and thus there are no low-level concurrent accesses to actual database files from different processes, but rather from several threads of a single server process.

as SQL-persistent. No information is kept incore, so that every database access imposes an overhead on the execution.‖

sql Finally, this is a native implementation in SQL of the benchmark code, i.e., what a programmer would have written directly in SQL, with no host language overhead. To perform these tests the database client included in MySQL has been used. The SQL sentences have been obtained from the Ciao Prolog interface and executed using the MySQL client in batch mode.

	prologdb (data)	**prologdb** (concur.)	**persdb**	**persdb /sql**	**sql**
assertz (25000 records)	590.5	605.5	5,326.4	16,718.3	3,935.0
non-indexed numeric query	7,807.6	13,584.8	7,883.5	17,721.0	17,832.5
non-indexed string query	8,045.5	12,613.3	9,457.9	24,188.0	23,052.5
indexed numeric query	1.1	3.0	1.1	1,082.4	181.3
indexed string query	1.1	3.0	1.5	1,107.9	198.8
non-indexed numeric retract	7,948.3	13,254.5	8,565.0	19,128.5	18,470.0
non-indexed string retract	7,648.0	13,097.6	11,265.0	24,764.5	23,808.8
indexed numeric retract	2.0	3.3	978.8	2,157.4	466.3
indexed string retract	2.0	3.1	1,738.1	2,191.9	472.5

Table 9.1: Speed in milliseconds of accessing and updating

Several conclusions can be drawn from Table 9.1:

Sensitivity to the amount of data to be transferred Some tests made to show the effect of the size of the data transferred on the access speed (which can be consulted in [CGC$^+$04b]) indicate that the methods which access to external processes (**persdb/sql** and **sql**) are specially sensitive to the data size, more than the file-based persistent database, whilst the internal Prolog rule database is affected to some extent only.

Incidence of indexing The impact of indexing is readily noticeable in the tables, especially for the internal Prolog rule database but also for the file-based persistent database. The MySQL-based tests do present also an

‖Clearly, it would be interesting to perform caching of read data, but note that this is not trivial since an invalidation protocol must be implemented, given there can be concurrent updates to the database. This is left as future work.

important speedup, but not as relevant as that in the Prolog-only tests. This behavior is probably caused by the overhead imposed by the SQL database requirements (communication with MySQL daemon, concurrency and transaction availability, much more complex index management, integrity constraint handling, etc). In addition to this, Prolog systems are usually highly optimized to take advantage of certain types of indexing, while database systems offer a wider class of indexing possibilities which might not be as efficient as possible in some determinate cases, due to their generality.

Impact of concurrency support Comparing the Prolog tests, it is worth noting that concurrent predicates bring in a non-insignificant load in rule database management (up to 50% slower than simple data predicates in some cases), in exchange for the locking and synchronization features they provide. In fact, this slow-down makes the concurrent Prolog internal rule database show a somewhat lower performance than using the file-based persistent database, which has its own file locking mechanism to provide inter-process concurrent accesses (but not from different threads of the same process: in that case both concurrency and persistence of predicates needs to be used).

Incidence of the Prolog interface in SQL characteristics Comparing direct SQL queries (i.e., typed directly at the database top-level interface) with using persistent predicates, we can see that only in the case of non-indexed queries times are similar, whereas indexed queries and database modifications show a significant difference. This is due to the fact that in the experiments the setting was used in which a different connection to the database server was open for every query requested, and closed when the query had finished (useful in practice to limit the number of open connections to the database, on which there is a limitation). We plan to perform additional tests turning on the more advanced setting in which the database connection is kept open.

9.3.2 Performance with Compile-time Optimizations

We have also implemented the two optimizations described in Section 9.2.3 (using static analysis information and query clustering) and measured the improvements brought about by these optimizations. The tests have been performed on two SQL-persistent predicates (`p/2` and `q/2`) with 1,000 facts each and indexed on the first column. There are no duplicate tuples nor duplicate values in any column (simply to avoid overloading due to unexpected backtracking). Both `p/2` and `q/2` contain exactly the same tuples.

Table 9.2 presents the time (in milliseconds) spent performing 1,000 repeated queries in a failure-driven loop. In order to get more stable measures average times were calculated for 10 consecutive tests, removing the highest and lowest values. The system used to run the tests was the same as in section 9.3.1.

The *single queries* part of the table corresponds to a simple call to `p(X,Z)`. The first row represents the time spent in recovering on backtracking all the 1,000 solutions to this goal. The second and third rows present the time taken when performing 1,000 queries to `p(X,Z)` (with no backtracking, i.e., taking only the first solution), with, respectively, the indexing and non-indexing argument being instantiated. The two columns correspond to the non-optimized case in which the translation to SQL is performed on the fly, and to the optimized case in which the SQL expressions are pre-computed at compile-time, using information from static analysis.

The '*complex queries*:`p(X,Z),q(Z,Y)`' part of the table corresponds to calling this conjunction with the rows having the same meaning as before. Information about variable groundness (on the first argument of the first predicate in the second row and on the second argument of the first predicate in the third row) obtained from global analysis is used in both of these rows. The two columns allow comparing the cases where the queries for `p(X,Z)` and `q(Z,Y)` are processed separately (and the join is performed in Prolog via backtracking) and the case where the compiler performs the clustering optimization and pre-compiles `p(X,Z),q(Z,Y)` into a single SQL query.

Finally, the '*complex queries*:`p(X,Z),r(Z,Y)`' part of the table illustrates the special case in which the second goal calls a predicate which only has a few tuples (but matching the variable bindings of the first goal). More concretely, `r/2` is a persistent predicate with 100 tuples (10% of the 1,000 tuples of `p/2`). All the tuples in `r/2` have in the first column a value which appears in the second column of `p/2`. Thus, in the non-optimized test, the Prolog execution mechanism will backtrack over the 90% of the solutions produced by `p/2` that will not succeed.

Single queries: `p(X,Y)`		
	on-the-fly SQL generation	pre-computed SQL expressions
Traverse solutions	36.6	28.5
Indexed ground query	1,010.0	834.9
Non-indexed ground query	2,376.1	2,118.1
Complex queries: `p(X,Z),q(Z,Y)`		
	non-clustered	clustered
Traverse solutions	1,039.6	51.6
Indexed ground query	2,111.4	885.8
Non-indexed ground query	3,550.1	2,273.8
Complex queries: `p(X,Z),r(Z,Y)`		
	non-clustered	clustered
Asymmetric query	1146.1	25.1

Table 9.2: Comparison of optimization techniques

Single queries: `p(X,Y)`		
	on-the-fly SQL generation	pre-computed SQL expressions
Indexed ground query	197.5	27.6
Non-indexed ground query	195.4	27.3
Complex queries: `p(X,Z),q(Z,Y)`		
	non-clustered on-the-fly	pre-computed clustered queries
Indexed ground query	406.8	33.3
Non-indexed ground query	395.0	42.6

Table 9.3: Comparison of optimization techniques (*Prolog time only*)

The results in Table 9.2 for single queries show that the improvement due to compile-time SQL expression generation is between 10 and 20 percent. These times include the complete process of a) translating (dynamically or statically) the literals into SQL and preparing the query (with our without optimizations), and b) sending the resulting SQL expression to the database and processing the query in the database. Since the optimization only affects the time involved in a), we measured also the effect of the optimizations when considering only a), i.e., only the time spent in Prolog. The results are shown in Table 9.3. In this case the run-time speed-up obtained when comparing dynamic generation of SQL at run time and static generation at compile time (i.e., being able to pre-compute the SQL expressions thanks to static analysis information) is quite significant. The difference is even greater if complex queries are clustered and translated as a single SQL expression: the time spent in generating the final SQL expression when clustering is pre-computed is only a bit greater than in the atomic goal case, while the non-clustered, on-the-fly SQL generation of two atomic goals needs twice the time of computing a single atomic goal. In summary, the optimization results in an important speedup on the Prolog side, but the overall weight of b) in the selected implementation (due to opening and closing DB connections) is more significant. We believe this overhead can be reduced considerably and this is the subject of ongoing work.

Returning to the results in Table 9.2, but looking now at the complex goals case, we observe that the speed-up obtained due to the clustering optimization is much more significant. Traversing solutions using non-optimized database queries has the drawback that the second goal is traversed twice for each solution of the first goal: first to provide a solution (as is explained above, `p/2` and `q/2` have exactly the same facts, and no failure happens in the second goal when the first goal provides a solution), and secondly to fail on backtracking. Both call and redo imply accessing the database. In contrast, if the clustering optimization is applied, this part of the job is performed inside the database, so there is only one database access for each solution (plus the last access when there are no more solutions). In the second and third rows, the combined effect of compile-time SQL expression generation and clustering optimization causes a speed-up of around 50% to 135%, depending on the cost of retrieving data from the database tables: as the cost of data retrieval increases (e.g., access based on a non-indexed column), the speed-up in grouping queries decreases.

Finally, the asymmetric complex query (in which the second goal succeeds for only a fraction of the solutions provided by the first goal) the elimination of useless backtracking yields the most important speed-up, as expected.

Chapter 10

Generation of Stripped-down Runtime Systems using Abstract Interpretation and Program Specialization

10.1 Introduction and Motivation

Libraries and modules are a fundamental tool for developing large applications, as they allow sharing common code between different programs and they provide a clean interface to widely used routines. Many development environments based on bytecode virtual machine emulators often provide a full-featured library with large amounts of code (as for example the Java run-time environment). Such systems are composed of two different environments: on one hand, a software development kit for program development, comprising a compiler and set of libraries, and on the other hand a runtime system, which contains a virtual machine interpreter (and/or a just-in-time compiler) and a bytecode version of the libraries. Such systems present a lot of advantages for the programmer: interoperability (to some extent in pervasive devices), a generic and independent programming interface, etc. However, these runtime systems tend to use an excessive amount of space in both memory and permanent storage, as their libraries are programmed for a very general usage, covering lots of possible cases. In addition, programmers tend to develop libraries that are more general purpose than is actually needed for a specific application, in order to make the library as reusable as possible. This approach, useful for program development, results however in applications which include significant amounts of useless code fragments.

In a pervasive system scenario, the space needed by a program is of vital importance in this kind of devices, and the use of general development tools and libraries is usually very restricted. In the case of Java, the alternative for developing software for small devices is to use a different development kit and runtime system, the Java Micro Edition.* It contains several runtime systems and development kits depending on how powerful the target device is, and several prepackaged sets or libraries for different functionalities (Java TV, Java Phone, etc.) This approach avoids the excessive use of resources done in the general approach, but at the same time constraints the range of runtime system libraries available to programmers. If a specific functionality not existing in the reduced runtime system is needed by the programmer, then it must be added to the program by hand in case it is possible, therefore losing the advantages of having a general programming library available in the extended runtime system. Moreover, it may be the case that the runtime system with additional libraries does not fit into the device's memory, but it would fit if the procedures in the library which are not used by the specific application being installed on it were removed.

In contrast, the approach presented in this chapter is to allow the programmer to use any part of a general runtime system library, and to apply abstract interpretation-based analysis and specialization techniques in order to remove all unnecessary code during execution. This dead code can be removed from both user programs or runtime system libraries, generating a specialized version of the runtime system for a given application.

Moreover, our approach can be easily extended to the specialization of runtime system libraries for a set of programs, instead of specializing them for only one program. Then, a specialized version for the runtime libraries can be generated to be installed in a pervasive device, including exactly the functionalities needed by the set of programs that will execute in such pervasive system.

This work will only take into consideration the specialization of user and runtime system libraries. It will not deal with the specialization of the bytecode

*`http://java.sun.com/products`

emulator itself, which is in general not written in the same source language as the libraries.

10.2 Execution Model Based on Abstract Machines

A basic execution model is generally composed of three different parts, depicted in Figure 10.1 and detailed as follows:

- The bytecode which corresponds to user programs.
- The bytecode which implements the runtime system libraries. This code is usually shared among all the user programs installed on the system.[†]
- The abstract machine emulator (or a platform-dependent just-in-time compiler), which interprets (or compiles) the bytecode.

When a runtime system is to be installed on a small device (like pervasive devices) the traditional approach is to define the version of the runtime system which best fits in the resources available in the device.

The main drawback of this approach is that the set of features provided by each runtime version is fixed. If a functionality is not included in the standard runtime system, it must be added manually by the programmer, even if there are pervasive devices powerful enough to host runtime libraries richer than the ones included in the runtime version designed for them.

10.3 Runtime library specialization

A different approach, taken in this work, is to allow the programmer to use the complete full-featured set of libraries of the most general version of the system. During compilation, both the program modules and the runtime libraries are stripped-down for the specific use of that program, using abstract interpretation-based techniques. The advantages are twofold: on one hand, the programmer

[†]Although in some cases a compiled user program may comprise the set of libraries needed.

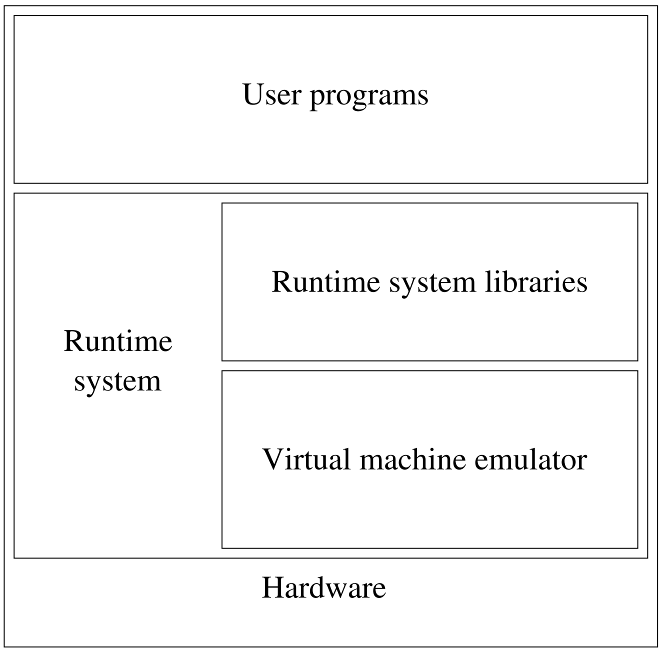


Figure 10.1: Generic runtime system

can use general libraries previously developed for other applications; on the other hand, the runtime libraries are included in the final runtime system only if they are to be used by the program. Moreover, the level of granularity when adding libraries to the runtime system is even finer than traditional compilers (that decide whether a library must be included or not if there are procedures invoked from the program, but they cannot decide if an individual procedure can be excluded from the library if it is not invoked from any part of the program), as abstract interpretation-based specialization detects and performs dead-code elimination, even when using very simple abstract domains.

This approach provides an additional advantage. When there are several applications that are to be executed in a given pervasive device, the runtime system is usually shared between them. The procedure depicted above can be easily adapted to perform the specialization of the runtime libraries for all the applications in the system. Therefore, the generated runtime system will include all the features needed by those applications and specialized for them, but it will not include other libraries or procedures inside libraries not used by the given set of programs.

10.4 A Practical application using `CiaoPP` and `Ciao`

10.4.1 The `Ciao` Runtime System Structure

The `Ciao` runtime system has the same general structure than other runtime systems like Java. As detailed before, it is composed of a bytecode emulator written in C and a wide set of engine and system libraries. As discussed throughout this thesis, the `Ciao` language includes a strict module system [CH00]. System libraries are encapsulated in `Ciao` modules, although some internal libraries are written in C for several technical reasons (some libraries are needed by the virtual machine, others have strict efficiency requirements, or they need to access low-level operating system resources).

`Ciao` libraries look like a user `Ciao` module, although they present slight differences: they are precompiled, and can be used from a user program using a `use_module(library(...))` construct, instead of including the complete path to the library module file. Even built-in procedures (not written in Prolog but embedded in the runtime engine) are listed in `Ciao` library modules, denoting with `impl_defined` declarations that they are not defined inside the module. The compiler then links the built-in predicate declaration with the actual fragment of runtime engine code. This approach to built-ins and libraries allows a very high degree of library specialization.

`Ciao` libraries are classified into two categories: *engine libraries*, those which are mandatory for the execution of any `Ciao` program, and the remaining libraries, which are necessary only if the user program needs their functionality. Figure 10.2 shows the structure of the `Ciao` runtime system.

By considering libraries as regular user files, the `Ciao` compiler is able to determine which libraries are needed for the user program: the compiled program will have the minimal set of libraries, instead of all runtime libraries as traditional runtime systems. This means that the compiler strips-down the runtime system at a module level, but if a library module is included, all procedures in the module will be included, even if they are not used by the program. In the following sections a finer-grained runtime system reduction is proposed.

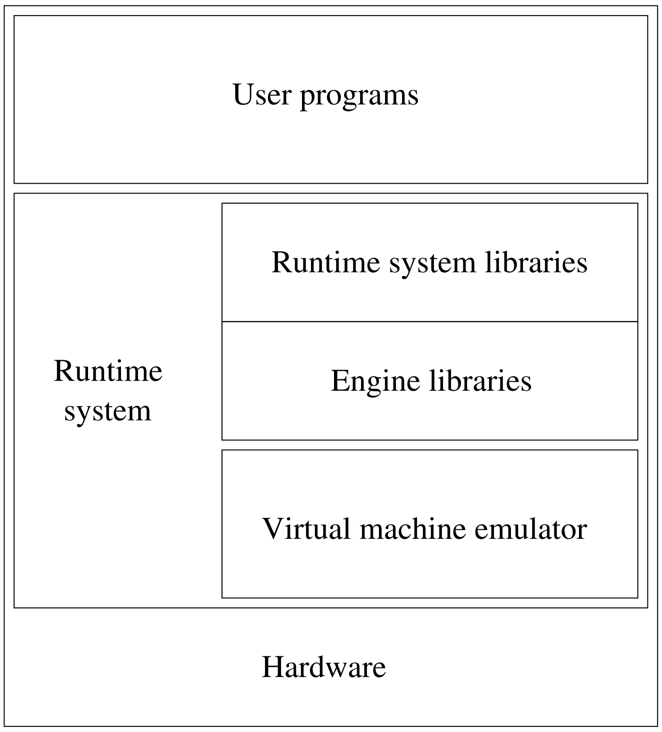


Figure 10.2: Ciao runtime system

10.4.2 Analysis and Specialization of modular `Ciao` programs

The analysis and specialization is performed using `CiaoPP`, the `Ciao` preprocessor, based on abstract interpretation already presented in Chapter 4. Only programs written in the `Ciao` language and its extensions can be processed, and library code implemented in C in the runtime engine cannot be processed nor specialized. Therefore, procedures declared as `impl_defined` are conservatively handled by the preprocessor.

The analysis of a modular program in `CiaoPP` is implemented following the framework described in Chapter 5. In summary, modules in the program are analyzed in turn, marking the call patterns for imported predicates as pending for analysis. When the imported module is analyzed, all pending patterns are processed. If the analysis results are more precise than those obtained in previous analyses of that module, the modules which import it are marked for reanalysis. This process terminates when there are no marked modules in the program with pending call patterns.

For this work a very simple abstract domain has been used. It only contains two abstract values, $\top$ and $\perp$, representing if a given predicate is used or not. More complex domains would bring more precise results on code reachability.

The results of this inter-modular analysis framework are the starting point of the specialization of modular programs. The specializer takes the list of calling patterns generated for every module, and removes the code that is unreachable from these calling patterns.

10.4.3 General algorithm for Runtime Generation

Given the analyzer and specializer for modular programs included in `CiaoPP`, the procedure for generating runtime libraries for a given program is as follows:

1. Determine the inter-modular graph of the program, including all needed libraries

2. Copy all these files to a separate place.

3. Perform the analysis of the copy of the user program and the copied libraries.

4. Perform the analysis of special startup code (in order not to lose code to be executed before the main predicate of the user program, as explained below).

5. Specialize the modular program, generating transformed source files for all the modules of the program and libraries.

10.4.4 Empirical results

In a first approach, user programs and non-engine libraries were considered for specialization. Engine libraries were excluded, as they were thought as not specializable. However, the results were not as good as one would expect: around a 10% of code reduction in a minimal program. Some engine libraries needed other libraries defined in `Ciao`, losing opportunities for specialization (e.g., the sorting

	Not specialized		Specialized
program	default libs.	manual	
minimal	2,260,293	816,835	501,081
qsort	2,277,134	822,275	504,374
queens	2,263,025	833,441	503,140

Table 10.1: Comparison of the size of compiled libraries (in bytes).

library is needed by aggregation predicates for implementing `setof/3`, needed in turn by the debugger, used by the engine).

The second approach is to generate a specialized version of all runtime system libraries, including engine libraries. All libraries are analyzed and specialized, leaving procedures implemented in C unchanged, but specializing all `Ciao` code.

In this approach, some specific engine modules require special treatment. During analysis and specialization, predicates are marked as needed by the program along the list of modules starting from the startup predicate (defined in the user program as `main/0` or `main/1`). Nevertheless, as mentioned before there is some startup code written in `Ciao` which is executed before the user program starts, and which therefore needs to be preserved in the final, specialized code. As this code is not called from any point of the user program nor the libraries, it will be removed by the specializer. Therefore, additional calling patterns for such code must be provided to `CiaoPP` together with the user program calling patterns.

The second approach brings much better results (even if they are still preliminary since the system can be improved significantly), and they are detailed in Table 10.1 for some simple examples.

In this table, Numbers correspond to a static compilation of the examples, which includes the libraries needed by the program, but does not include the virtual machine emulator itself. The first example is the smallest `Ciao` program, while `qsort` and `queens` are simple benchmarks which include some additional libraries: `qsort` uses `append/3` from the lists handling library and `write/1` for printing out the results, and `queens` uses the `Ciao` statistics library to get the time spent in the benchmark and `format/2` for formatted output.

the second and third columns correspond to the traditional compilation of the programs, including the default set of `Ciao` libraries in the first case, or just the minimal set of libraries, in the second case. The fourth column is the result of specializing the code of programs and libraries, removing dead code.

10.5 Conclusions and Future Work

Despite the preliminary nature of this work, We have already obtained an important reduction on the size of libraries by removing library procedures which are not used by a given program. Furthermore, a significant improvement is expected using richer domains, as other abstract domains can detect additional fragments of unused code (for example, using modes domains in programs with tests on the instantiation of variables). Another source of improvement is the detection of dead-code for built-ins written in C. Currently, procedures written in C are not removed from the system engine even if they are not called from anywhere in the program. A procedure can be implemented to get the annotations produced by `CiaoPP` analyzer to generate C code only for the library procedures which are used in the program.

Part V

Conclusions and Future Work

This thesis has focused in studying different aspects of analysis and verification of modular programs in the scope of context-sensitive, multivariant, abstract interpretation-based analysis of logic programs. Research and implementation have been centered in preserving as much as possible the accuracy of the existing well known techniques for non-modular programs, as well as enabling these technologies for the common case of incremental processing of programs after source code modifications.

Among the most relevant conclusions that can be drawn from this thesis we can mention:

- We have designed and implemented the first analysis framework for logic programs based on abstract interpretation that is capable of performing context-sensitive, multivariant analysis without losing precision in non-compositional domains. It allows incrementally reanalyzing a program in which some of its modules changed since the last time it was analyzed. We have performed several experiments that show the practicality of this approach, especially in the case of incremental reanalysis. The main advantages offered by our approach are:

  It is a module-centered approach. The basic unit of our approach is the module. That means that our framework has to deal with modules which are incomplete, in the sense that they may call procedures which are not locally defined but rather imported from other modules; and furthermore, it must properly handle the interactions between modules, and be able to make the analysis results converge to a fixed point.

  It is domain-independent. The analysis framework works with both compositional and non-compositional domains. Several abstract domains have been tested in this thesis, including Def, $Sharing-freeness$, and type domains, as well as other simpler domains like the one used for the case study showed in Chapter 10. The special case of type domains has been specifically addressed, due to the particular characteristics and issues they pose for modular analysis. New domains can be easily plugged into the framework.

It is based on a parametric framework. Since the purpose of Part II of this thesis is to study how to analyze a modular program, different parameters have been identified and evaluated. As a result, the developed framework can be configured with respect to those parameters, producing different scenarios.

It allows different usages. The analysis framework can be used either for analyzing a modular program from scratch, or for analyzing modules while the program is being developed, or for reanalyzing the program after changes in the source code.

It is usable from early phases of program development. This approach does not need that the whole program is completely implemented to start analyzing it. The manual scheduling policy allows programmers to decide which module needs to be analyzed and when it can be analyzed. The framework keeps the information obtained from the analysis of other modules in the program in order to take advantage of it and to produce the most accurate results.

It allows the analysis of incomplete programs. In addition to analyzing modules with incomplete code located in imported modules, the analysis framework can also analyze programs that use third party code or compiled code, or that interface to other languages or systems, as illustrated in Chapter 9 when dealing with the interface to relational database systems.

New techniques for types domains have been developed. In addition to the domain-independence mentioned above, the use of type domains has been specifically studied, since they have characteristics that make them special. Two new techniques for handling type domains have been developed for improving their efficiency and make them amenable for modular analysis.

The incremental analysis if efficient in terms of time and memory. Experimental results show that the reanalysis of a program after changes in a module within this framework is more efficient than monolithically analyzing it from scratch, even in the extreme case of changing all exported procedures in the module.

It is integrated into `CiaoPP`. This framework has been successfully integrated into the `CiaoPP` system. That means that it is integrated in the `CiaoPP` user interface, and that, furthermore, it can take advantage of the rest of tools and technologies developed in that system, including any of the abstract interpretation-based domains implemented in `CiaoPP`.

- A compile-time assertion checking framework has also been designed and implemented. It is the first assertion checking framework for logic programs that is capable of checking modular programs with partial user specifications, and that uses context-sensitive, multivariant analysis results based on abstract interpretation. The most remarkable advantages that can be drawn are:

It builds on top of modular analysis framework. Instead of developing an ad-hoc system for extracting information for compile-time assertion checking, it is based on the results obtained by the modular analysis framework. This allows the assertion checker to rely on well established technology for obtaining accurate program information, and provides the basics for innovative modular assertion checking algorithms.

It allows a partial specification of the program. Since the assertion checking framework relies on the analysis results, the specification of the program may be incomplete. The analysis results replace user specifications. Even in the case in which the user specification is non-existent, the user program can still be checked with respect of the specification of library procedures called from the program.

The notion of conditional correctness is introduced. In order to make use of the modular nature of programs being checked, the concept of conditional partial correctness has been introduced, and some important results on overall program correctness obtained.

Several algorithms have been developed and implemented. An initial naive algorithm has been implemented, in which the assertions are checked after analyzing the program. A second scenario has been described for programs which contain a rather complete specification and in which there is no need of reaching an inter-modular analysis fixed-point. Instead, the new concept of conditional partial correctness of modules is proved, and using it as a starting point, program partial correctness obtained under some conditions. And finally, an algorithm has been designed and implemented for interleaving inter-modular analysis and assertion checking. It makes use of the internal structure of the analysis framework to stop analysis processing as soon as an error is detected in the program, instead of requiring the analysis reach an intermodular fixed point to start checking its assertions.

Issues regarding program verification with type domains have been addresed. When using type domains for compile time checking, several issues regarding the analysis of modular programs with those domains must be addressed, as explained above. Specifically, the expressivity of assertions based on types has been enhanced by the use of type variables in the assertions, and at the same time remain in descriptive types (types which describe approximations of the untyped semantics).

It is integrated into `CiaoPP`. Also the assertion checking framework has been successfully integrated into the `CiaoPP` system, taking advantage of the `CiaoPP` user interface and the rest of tools and technologies developed in that system.

- Two real-life applications have been evaluated. On one hand, it has been tested and evaluated the application of modular program analysis to incomplete programs for the specific case of programs interfacing relational databases. The analysis of the program helps in detecting whether there is room for program optimization, generating SQL queries at compile time, and improving their efficiency by collapsing several queries in one complex SQL query that joins several database tables. The second application deals with analyzing the `Ciao` system libraries in order to save space in a pervasive device. In this case, a preliminar modular specialization framework algorithm has been developed.

As can be easily seen, this thesis is a first step in several research lines that may be of practical and theoretical interest. Among them, it is important to highlight the following future work:

- It would be interesting to study in depth the scheduling policy for modular analysis. The policies used in this thesis are simple approaches, that demonstrated the usefulness of the approach, but a thorough study should be carried out.

- In parallel with the previous line, there is also an important line of research in the improvement of the analysis of modular programs in other ways: for instance, distributing the analysis of different modules in a network of analysis servers.

- The use of infinite abstract domains and the application of widenings in the modular analysis framework poses interesting issues that should be carefully studied. The way a widening could be defined to overcome module boundaries limitation is of special interest.

- Another important issue is to establish a complete set of assertions for libraries.

- The frameworks designed in this thesis have been applied to (constraint) logic programming languages like the one in the `Ciao` system. Nevertheless, it can be extended to other programming paradigms.

- The assertion checking scenarios described in this thesis can be trivially extended to allow incremental assertion checking. This approach would allow verifying that a program is correct after changes in some modules at a much lower cost than checking it again from scratch.

Bibliography

[AM94] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

[AP93] K. R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 1(106):109–157, 1993.

[APG06] 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.

[APH04] E. Albert, G. Puebla, and M. Hermenegildo. An Abstract Interpretation-based Approach to Mobile Code Safety. In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, Electronic Notes in Theoretical Computer Science 132(1), pages 113–129. Elsevier - North Holland, April 2004.

[BCC+02] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual (v1.8). The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of Madrid (UPM), May 2002. System and on-line version of the manual available at `http://www.ciaohome.org`.

[BCC+04] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.10). The ciao system documentation series–TR, School of Computer Science, Technical University of Madrid (UPM), June 2004. System and on-line version of the manual available at `http://www.ciaohome.org`.

[BCC+06] 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`.

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[BDD+97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[BdlBH99] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[BdlBH+01] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[BDM97] J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.

[BG92] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 19(3):281–313, 1992.

[BGLM94] A. Bossi, M. Gabbrieli, G. Levi, and M.C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122(1,2):3–47, 1994.

[BJ03] F. Besson and T. Jensen. Modular class analysis with datalog. In *10th International Symposium on Static Analysis, SAS 2003*, number 2694 in LNCS. Springer, 2003.

[BLGH04] F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[BLGPH06a] F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. A Tutorial on Program Development and Optimization using the Ciao Preprocessor. Technical Report CLIP2/06, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2006.

[BLGPH06b] F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. The Ciao Preprocessor. Technical Report CLIP1/06, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2006.

[BLM94] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19–20:443–502, July 1994.

[Bou93] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[Cab04] D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System.* PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.

[CC77a] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC77b] 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.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. Rapport de recherche lix/rr/92/08, LIX, Ecole Polytechniqe (Palaiseau, France), 1992. Also appeared in Journal of Logic Programming (1992).

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, 1995. ACM Press, New York, NY.

[CC02a] P. Cousot and R. Cousot. Modular Static Program Analysis, invited paper. In *Eleventh International Conference on Compiler Construction, CC 2002*, number 2304 in LNCS, pages 159–178. Springer, 2002.

[CC02b] P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *POPL'02: 29ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, January 2002. ACM.

[CCG+98] I. Caballero, D. Cabeza, S. Genaim, J.M. Gomez, and M. Hermenegildo. persdb_sql: SQL Persistent Database Interface. Technical Report CLIP10/98.0, December 1998.

[CCH06] A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *The 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, pages 142–162, Fuji Susono (Japan), April 2006.

[CDG93] M. Codish, S. K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proc. POPL'93*, 1993.

[CGC+03a] J. Correas, J. M. Gomez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for CLP Systems. In *2003 International Conference on Logic Programming*, number 2916 in LNCS, pages 481–482, Heidelberg, Germany, December 2003. Springer-Verlag. Extended abstract.

[CGC+03b] J. Correas, J. M. Gomez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for CLP Systems (And Two Useful Implementations). In M. Carro and J. Correas, editors, *Second CoLogNet Workshop on Implementation Technology for Computational Logic Systems (Formal Methods '03 Workshop)*, pages 51–64, School of Computer Science, Technical University of Madrid, September 2003. Facultad de Informatica.

[CGC+04a] J. Correas, J. M. Gomez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for CLP Systems (And Two Useful Implementations). In *Proceedings of the Sixth*

International Symposium on Practical Aspects of Declarative Languages, number 3057 in LNCS, pages 104–119, Heidelberg, Germany, June 2004. Springer-Verlag.

[CGC$^+$04b] J. Correas, J. M. Gomez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for (C)LP Systems (and two useful implementations). Technical Report CLIP3/2003.1(2004), Technical University of Madrid, School of Computer Science, UPM, April 2004. `http://cliplab.org/papers/persdb-tr1.pdf`.

[CH94] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.

[CH99] M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, USA, November 1999.

[CH00] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[CHGT98] D. Cabeza, M. Hermenegildo, S. Genaim, and C. Taboch. Design of a Generic, Homogeneous Interface to Relational Databases. Technical Report D3.1.M1-A1, CLIP7/98.0, September 1998.

[CL00] M. Codish and V. Lagoon. Type Dependencies for Logic Programs using ACI-unification. *Journal of Theoretical Computer Science*, 238:131–159, 2000.

[CLMV99] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.

[CLV95] M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In *1995 International Logic Programming Symposium*, pages 275–287, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.

[CLV98] M. Comini, G. Levi, and G. Vitiello. Modular abstract diagnosis. In *APPIA-GULP-PRODE'98 Joint Conference on Declarative Programming*, pages 409–420, 1998.

[Cou03a] P. Cousot. Automatic Verification by Abstract Interpretation. In *Proc. of VMCAI'03*, pages 20–24. Springer LNCS 2575, 2003.

[Cou03b] P. Cousot. Automatic Verification by Abstract Interpretation, Invited Tutorial. In *Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, number 2575 in LNCS, pages 20–24. Springer, January 2003.

[CPHB06] J. Correas, G. Puebla, M. Hermenegildo, and F. Bueno. Experiments in Context-Sensitive Analysis of Modular Programs. In *15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, number 3901 in LNCS, pages 163–178. Springer-Verlag, April 2006.

[Der93] P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

[DHM00] P. Deransart, M. Hermenegildo, and J. Maluszynski. *Analysis and Visualization Tools for Constraint Programming.* Number 1870 in LNCS. Springer-Verlag, September 2000.

[dlBH93] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 437–455. MIT Press, October 1993.

[dlBHB$^+$96a] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, September 1996.

[dlBHB$^+$96b] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Trans. on Programming Languages and Systems*, 18(5):564–615, 1996.

[dlBHM00] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems*, 22(2):269–339, March 2000.

[DLGH97] S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[DLGHL94] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[DLGHL97] 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.

[DM88] W. Drabent and J. Małuszyński. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59:133–155, 1988.

[DMP02] W. Drabent, J. Małuszyński, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs. *Theory and Practice of Logic Programming*, 2(4–5):549–611, 2002.

[DNTM89] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.

[Dra91] Christoph Draxler. *Accessing Relational and Higher Databases through Database Set Predicates in Logic Programming Languages.* PhD thesis, Zurich University, Department of Computer Science, 1991.

[DZ92] P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[Fer87] G. Ferrand. Error diagnosis in logic programming. *J. Logic Programming*, 4:177–198, 1987.

[GDMS02] María J. García de la Banda, Bart Demoen, Kim Marriott, and Peter J. Stuckey. To the Gates of HAL: A HAL Tutorial. In *International Symposium on Functional and Logic Programming*, pages 47–66, 2002.

[GdW94] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[GH91] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[GH04] J. Gallagher and K. Henriksen. Abstract domains based on regular types. In B. Demoen and V. Lifschitz, editors, *ICLP 2004, Proceedings*, volume 3132 of *LNCS*, pages 27–42. Springer-Verlag, 2004.

[GP02] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, number 2257 in LNCS, pages 243–261. Springer-Verlag, January 2002.

[HBC$^+$99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[HBPLG99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 Int'l. Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[Her00] M. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.

[HJ90] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proc. 17th POPL*, pages 197–209, 1990.

[HJ92] N. Heintze and J. Jaffar. Semantic types for logic programs. In Pfenning [Pfe92], pages 141–155.

[HL94a] P. Hill and J. Lloyd. *The Goedel Programming Language.* MIT Press, Cambridge MA, 1994.

[HL94b] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* Logic Programming Series. MIT Pres, 1994.

[HPB99] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[HPBLG03a] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Abstract Verification and Debugging of Constraint Logic Programs. In *Recent Advances in Constraints*, number 2627 in LNCS, pages 1–14. Springer-Verlag, January 2003.

[HPBLG03b] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[HPBLG05] 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.

[HPMS00] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[HR95] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[JB92] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

[KMM$^+$98] A. Kelly, A. Macdonald, K. Marriott, H. Søndergaard, and P.J. Stuckey. Optimizing compilation for CLP($\mathcal{R}$). *ACM Transactions on Programming Languages and Systems*, 20(6):1223–1250, 1998.

[Kow96] R. A. Kowalski. Logic Programming with Integrity Constraints. In *Proceedings of JELIA*, pages 301–302, 1996.

[Leu98] M. Leuschel. Program Specialisation and Abstract Interpretation Reconciled. In *Joint International Conference and Symposium on Logic Programming*, June 1998.

[LGBH05] P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

[LGHD96] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21:715–734, 1996.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer, 2nd Ext. Ed., 1987.

[LM05] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods (FM)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer-Verlag, 2005.

[LO87] Timothy G. Lindholm and Richard A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the Fourth Int'l. Conference and Symposium*, pages 21–39. The MIT Press, 1987.

[Log04a] Francesco Logozzo. *Modular Static Analysis of Object-oriented Languages.* PhD thesis, Ecole Polytechnique, 2004.

[Log04b] Francesco Logozzo. Separate compositional analysis of class-based object-oriented languages. In *Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST'2004)*, volume 3116 of *Lectures Notes in Computer Science*, pages 332–346. Springer-Verlag, July 2004.

[Log07] Francesco Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In *VMCAI'07*, number 4349 in LNCS. Springer, Jan 2007.

[LS88] Y. Lichtenstein and E. Y. Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 512–531, Seattle, Washington, August 1988. MIT.

[Lu95] L. Lu. Type Analysis of Logic Programs in the Presence of Type Definitions. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based program manipulation*, pages 241–252. The ACM Press, 1995.

[Lu98] L. Lu. Polymorphic Type Analysis of Logic Programs by Abstract Interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998.

[Lu01] L. Lu. On Dart-Zobel Algorithm for Testing Regular Type Inclusion. *SIGPLAN NOTICES*, 36(9):81–85, 2001.

[LV94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[M9́5] D. Le Métayer. Proving properties of programs defined over recursive data structures. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–99, 1995.

[MBdlBH99] K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.

[MdlBH94] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[MH90a] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[MH90b] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[Mil99] P. Mildner. *Type Domains for Abstract Interpretation: A Critical Study.* PhD thesis, Computing Science Department - Uppsala University, Uppsala, 1999.

[Mis84] P. Mishra. Towards a theory of types in prolog. In *International Symposium on Logic Programming*, pages 289–298, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[MJB00] N. Mazur, G. Janssens, and M. Bruynooghe. A module based analysis for memory reuse in mercury. In *First International Conference on Computational Logic (CL 2000)*, number 1861 in LNCS, pages 1255–1269. Springer-Verlag, July 2000.

[MO84] A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[Net02] Nicholas Nethercote. The Analysis System of HAL. Master's thesis, Monash University, 2002.

[PAH05] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 149–165. Springer-Verlag, August 2005.

[PAH06] G. Puebla, E. Albert, and M. Hermenegildo. Abstract Interpretation with Specialized Definitions. In *The 13th International Static Analysis Symposium (SAS'06)*, number 4134 in LNCS, pages 107–126. Springer, August 2006.

[Pat04] N. D. Pattengale. Transactional semantics. Technical Report CLIP3/04.0, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, February 2004.

[PB02] A. Pineda and F. Bueno. The O'Ciao Approach to Object Oriented Logic Programming. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[PBH00a] G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[PBH00b] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[PBH00c] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.

[PBH00d] G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[PCH$^+$04] G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey. 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*, number 3049 in LNCS, pages 234–261. Springer-Verlag, Heidelberg, Germany, August 2004.

[PCPH06] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo. Context-Sensitive Multivariant Assertion Checking in Modular Programs. In *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06)*, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

[PCPH08] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo. A Practical Type Analysis for Verification of Modular Prolog Programs. In *ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation (PEPM'08)*, pages 61–70. ACM Press, January 2008.

[PdlBMS97] G. Puebla, M. García de la Banda, K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on Logic Programming*, pages 93–107, Cambridge, MA, June 1997. MIT Press.

[Pfe92] F. Pfenning, editor. *Types in Logic Programming.* MIT Press, 1992.

[PH96] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[PH97] G. Puebla and M. Hermenegildo. Abstract Specialization and its Application to Program Parallelization. In J. Gallagher, editor, *Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.

[PH99] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[PH00] G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[PH03] G. Puebla and M. Hermenegildo. Abstract Specialization and its Applications. In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003. Invited talk.

[PHG99] G. Puebla, M. Hermenegildo, and J. Gallagher. An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework. In O Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, number NS-99-1 in BRISC Series, pages 75–85. University of Aarhus, Denmark, January 1999.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, February 2002.

[PRO94] International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England. *PROLOG. ISO/IEC DIS 13211 — Part 1: General Core*, 1994.

[PRO00] International Organization for Standardization, 1, rue de Varembé, Case postale 56, CH-1211 Geneva 20, Switzerland. *PROLOG. ISO/IEC DIS 13211-2 — Part 2: Modules*, 2000.

[Pro02] Christian W. Probst. Modular Control Flow Analysis for Libraries. In *Static Analysis Symposium, SAS'02*, volume 2477 of *LNCS*, pages 165–179. Springer-Verlag, 2002.

[RRL99] A. Rountev, B.G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *ESEC/FSE'99*, volume 1687 of *LNCS*, pages 235–252. Springer-Verlag, 1999.

[RU93] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[Sch07] David A. Schmidt. A calculus of logical relations for over- and underapproximating static analyses. *Sci. Comput. Program.*, 64(1):29–53, 2007.

[SG95] H. Saglam and J. Gallagher. Approximating constraint logic programs using polymorphic types and regular descriptions. Technical Report CSTR-95-17, Department of Computer Science, University of Bristol, Bristol BS8 1TR, 1995.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *JLP*, 29(1–3), October 1996.

[TJ94] Y. M. Tang and P. Jouvelot. Separate abstract interpretation for control-flow analysis. In *Theoretical Aspects of Computer Software (TACS '94)*, number 789 in LNCS. Springer, 1994.

[Ull90] J. D. Ullman. *Database and Knowledge-Base Systems, Vol. 1 and 2*. Computer Science Press, Maryland, 1990.

[VB00] W. Vanhoof and M. Bruynooghe. Towards modular binding-time analysis for first-order mercury. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[VB02] C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.

[VDLM93] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michael. The Impact of Granularity in Abstract Interpretation of Prolog. In *Workshop on Static Analysis*, number 724 in LNCS, pages 1–14. Springer-Verlag, September 1993.

[VHCL95] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

[YS87] E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Concurrent Prolog: Collected Papers*, pages 211–244, 1987.

[YS90] E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.