Decompilation of Java Bytecode to Prolog by Partial Evaluation

Miguel Gómez-Zamalloa[a], Elvira Albert[a], Germán Puebla[b]

[a]*DSIC, Complutense University of Madrid, E-28040 Madrid, Spain*
[b]*CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain*

Abstract

Reasoning about Java bytecode (JBC) is complicated due to its unstructured control-flow, the use of three-address code combined with the use of an operand stack, etc. Therefore, many static analyzers and model checkers for JBC first convert the code into a higher-level representation. In contrast to traditional decompilation, such representation is often not Java source, but rather some intermediate language which is a good input for the subsequent phases of the tool. *Interpretive decompilation* consists in partially evaluating an interpreter for the compiled language (in this case JBC) written in a high-level language w.r.t. the code to be decompiled. There have been proofs-of-concept that interpretive decompilation is feasible, but there remain important open issues when it comes to decompile a real language such as JBC. This paper presents, to the best of our knowledge, the first modular scheme to enable interpretive decompilation of a realistic programming language to a high-level representation, namely of JBC to Prolog. We introduce two notions of optimality which together require that decompilation does not generate code more than once for each program point. We demonstrate the impact of our modular approach and optimality issues on a series of realistic benchmarks. Decompilation times and decompiled program sizes are linear with the size of the input bytecode program. This demonstrates empirically the scalability of modular decompilation of JBC by partial evaluation.

Key words: program transformation, partial evaluation, decompilation, interpreters, Java bytecode, logic programming, Prolog

1. Introduction

Decompilation of Java bytecode (JBC for short) to an intermediate representation has become a usual practice nowadays within the development of analyzers, verifiers, model checkers, etc. For instance, in the context of *mobile* code, as the source code is not available, decompilation facilitates the reuse of existing analysis and model checking tools. In general, high-level intermediate representations allow abstracting away the particular language features and developing the tools on simpler representations. In particular, JBC is decompiled to a rule-based representation in [2], to clause-based programs in [35], to a three-address code representation in Soot [43] and to the typed procedural language BoogiePL in [13]. Also, analysis of Java programs is formalized and performed using Datalog in [44] and in [20] PIC assembly is transformed into logic

Email addresses: `mzamalloa@fdi.ucm.es` (Miguel Gómez-Zamalloa), `elvira@fdi.ucm.es` (Elvira Albert), `german@fi.upm.es` (Germán Puebla)

programs. This shows that the rule-based representations used in declarative programming in general—and in Prolog in particular—provide a convenient formalism to define such intermediate representations. For instance, as it can be seen in [2, 35, 20], the operand stack used in a bytecode language can be represented by means of explicit logic variables and its unstructured control flow can be transformed into recursion.

All above cited approaches (except [20]) develop *ad-hoc*, or dedicated, decompilers to carry out the particular decompilations. An appealing alternative to the development of dedicated decompilers is the so-called *interpretive* decompilation by *partial evaluation* (PE for short) [23]. PE is an automatic program transformation technique which specializes programs w.r.t. part of their input data. Interpretive compilation was proposed in Futamura's seminal work [14], whereby compilation of a program P written in a (*source*) programming language L_S into another (*target*) programming language L_T is achieved by specializing an interpreter for L_S written in L_T w.r.t. P. The advantages of interpretive (de-)compilation w.r.t. dedicated (de-)compilers are well-known and discussed in the PE literature (see, e.g., [5]). Very briefly, they include:

1. *Flexibility*: it is easier to modify the interpreter in order to tune the decompilation (e.g., observe new properties of interest). As an interesting example, in [5], a Java bytecode interpreter is instrumented with an additional argument which computes the *trace* of bytecode instructions in order to collect the computation history. A program decompiled by using this interpreter contains an additional argument with the execution trace at the level of Java bytecode. This trace will allow observing a good number of interesting properties about the program, e.g., runtime error freeness can be ensured when the trace does not contain instructions which issue any kind of run-time error.
2. *Easier to trust*: it is more difficult to prove that ad-hoc decompilers preserve the program semantics. For example, the formal specification chosen for defining our bytecode interpreter is Bicolano [40], which is written with the Coq Proof Assistant [7]. This allows checking that the specification is consistent and also proving properties on the behavior of some programs.
3. *Easier to maintain*: new changes in the language semantics can be easily reflected in the interpreter. This will become apparent later when we see that defining a bytecode interpreter in Prolog is a rather easy task and, hence, also maintaining it.

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

1.1. Summary of Contributions

There have been several proofs-of-concept of interpretive (de-)compilation (e.g., [5, 20, 29]), but there remain interesting open issues when it comes to assess its power and/or limitations to decompile a real language:

a) *does the approach scale?*

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

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

This article answers these questions positively by proposing a modular decompilation scheme which can be steered to control the structure of decompiled code and ensure quality decompilations which preserve the original program's structure. Our main contributions are summarized as follows:

1. We present the problems of *non-modular* decompilation and identify the components needed to enable a modular scheme. This includes how to write an interpreter and how to control an *online* partial evaluator in order to preserve the structure of the original program w.r.t. method invocations.
2. We present a modular decompilation scheme which is correct and complete for the proposed big-step interpreter. The *modular-optimality* of the scheme allows addressing issue *(a)* by avoiding decompiling the same method more than once, and *(b)* by ensuring that the structure of the original program can be preserved.
3. We introduce an interpretive decompilation scheme which answers issue *(c)* by producing decompiled programs whose *quality* is similar to that of dedicated decompilers. This requires a *block-level* decompilation scheme which avoids code duplication and code re-evaluation.
4. We report on experimental results on an set of realistic JBC programs which demonstrate the scalability and the efficiency of our proposal.

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

1.2. Outline of the Article

The article is organized as follows. The next section recalls some preliminary definitions and presents the background on PE of logic programs. We recall the correctness issues that a partial evaluator must guarantee. We also sketch the differences between online and offline partial evaluators. Section 3 briefly describes the interpretive approach to (de-)compilation. We present the first Futamura projection in generic terms and then instantiate it to the particular decompilation we want to carry out: decompile JBC to Prolog. Section 4 presents the subset of JBC we consider to define our decompilation scheme. It also describes non-modular decompilation (originally presented in [5]) and explains its limitations for the decompilation of real applications. These limitations are not tied to the decompilation of bytecode. They also occur in any application of interpretive decompilation.

Our first contribution is a modular decompilation scheme which is introduced in Section 5. We start by presenting a big-step interpreter and explain why it is necessary to enable a modular decompilation scheme. Then, we define the annotations that must be generated to obtain such modular decompilation. An important property of the resulting method is that it is ensured that each method is decompiled once.

Our second important contribution is the refinement of the modular decompilation scheme in Section 6 to ensure the scalability of our approach. This requires, among other things, that the decompiler does not emit code more than once for each bytecode instruction. This leads to what we call *block-optimality* in decompilation.

In Section 7 we extend the subset of JBC considered in previous sections in order to support a realistic language with object-oriented features. We show how our scheme can be easily adapted to handle the new features: the decompilation of the heap and associated instructions, the representation of classes by means of Prolog modules and virtual invocations by module-qualified calls. Our experimental results are reported in Section 8, where both the scalability and efficiency of our approach are assessed using the JOlden suite of benchmarks [22]. Finally, Section 9 reviews related work and Section 10 concludes.

2. Background on Partial Evaluation of Logic Programs

This section presents some preliminary notions and the background on PE of logic programs (often called *partial deduction*) required to formalize our decompilation scheme. We assume some basic knowledge on the terminology of logic programming and refer to [34] for details.

2.1. Logic Programming

Very briefly, an *atom* (or call) A is a syntactic construction of the form $p(t_1, \ldots, t_n)$, with $n \geq 0$, where p/n is a predicate signature and $t_1, \ldots, t_n$ are terms. A *clause* is of the form H :- $B_1, \ldots, B_m$, with $m \geq 0$, where its head H is an atom and its body $B_1, \ldots, B_m$ is a conjunction of m atoms. Note that in this context commas denote conjunctions. When $m = 0$ the clause is called a *fact* and is written "H.". A *program* is a finite set of clauses. A *goal* is a conjunction of atoms. We denote by $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ the *substitution* σ with $\sigma(X_i) = t_i$ for $i = 1, \ldots, n$ (with $X_i \neq X_j$ if $i \neq j$), and $\sigma(X) = X$ for all other variables X. Given an atom A, $\theta(A)$ denotes the application of substitution θ to A. Given two substitutions θ_1 and θ_2, we denote by $\theta_1\theta_2$ their composition. The identity substitution is denoted by id. An atom A' is an *instance* of A if there is a substitution σ with $A' = \sigma(A)$.

The operational semantics of logic programs is based on derivations.

Definition 1 (derivation step). *Let G be $A_1, \ldots, A_R, \ldots, A_k$ and $C = H$:- $B_1, \ldots, B_m$ be a renamed apart clause in P (i.e., it has no common variables with G). Let A_R be the selected atom for its evaluation. Then G' is* derived *from G if the following conditions hold:*

$$\begin{aligned} \theta &= mgu(A_R, H) \\ G' \text{ is the goal } &\theta(A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k) \end{aligned}$$

As customary, given a program P and a goal G, an *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P (i.e. C_i has no common variables with any G_j nor C_j with $j < i$), and a sequence of *computed answer substitutions* $\theta_1, \theta_2, \ldots$ (or *most-general unifiers*, mgus for short) such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. Finally, we say that the SLD derivation is composed of the *subsequent* goals $G_0, G_1, G_2, \ldots$.

A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees*. A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer for goal G. Such a derivation is called *failing* if it is not possible to perform a derivation step with G_n.

Executing a program P for a call A consists in building an *SLD tree* for $P \cup \{A\}$ and then extracting the *computed answers* from every non-failing branch of the tree.

2.2. Partial Deduction

Partial evaluation in logic programming (see e.g. [16]) builds upon the SLD trees mentioned above. We now introduce a generic function *PE*, which is parametric w.r.t. the *unfolding rule*, unfold, and the *abstraction operator*, abstract, and captures the essence of most algorithms for PE of logic programs:

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

Function PE differs from standard ones in the use of the set of annotations $\mathcal{A}$, whose role is described below. PE starts from a program P, a (possibly empty) set of annotations $\mathcal{A}$ and an initial set of calls S. At each iteration, the so-called *local control* is performed by the unfolding rule unfold (Line 4), which takes the current set of atoms S_i, the program and the annotations and constructs a *partial* SLD tree for each call in S_i. Trees are partial in the sense that, in order to guarantee termination of the unfolding process, it must be possible to choose *not* to further unfold a goal, and rather allow leaves in the tree with a non-empty, possibly non-failing, goal (these goals appear in the next examples within a frame). The atoms corresponding to such goals are returned by unfold and store in L^{pe} (Line 4). Then, in the *global control*, which is performed by the abstraction operator abstract, when some calls in the leaves of the trees are not properly *covered*, the operator abstract adds them to the new set of atoms to be partially evaluated in a proper "generalized" form such that termination is ensured (i.e., the condition $S_i = S_{i-1}$ is reached).

Let us consider the PE of the following program to reverse a list using an accumulator (predicate `rev/3`) w.r.t. the initial set $S = \{\texttt{rev}([1,2|\texttt{Xs}],[],\texttt{Zs})\}$ and $\mathcal{A} = \emptyset$:

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

Prolog lists use the notation $[X|L]$ to denote the list with X as head and L as continuation and [] to denote the empty list. The particular unfold operator determines which atom to select from each goal and when to stop unfolding. Let us consider an unfolding rule based on the *homeomorphic embedding* [28] relation, a *well-quasi order* used in state-of-the-art specialization tools. Intuitively, the homeomorphic embedding is a structural ordering under which an expression e_1 is greater than (i.e., it *embeds*), another expression e_2 if e_2 can be obtained from e_1 by deleting some parts, e.g., $\underline{\texttt{s}}(\texttt{s}(\underline{\texttt{U}}+\texttt{W})\underline{\times}(\underline{\texttt{U}+}\texttt{s}(\underline{\texttt{V}})))$ embeds $\texttt{s}(\texttt{U}\times(\texttt{U}+\texttt{V}))$. Such unfolding rule always selects the leftmost atom and stops the derivation when the selected call *embeds* a previous call and thus threatens termination. We start by constructing the following SLD-tree:

$$\begin{tikzcd}
& \mathtt{rev([1,2|Xs],[],Zs)} \arrow[d] & \\
& \mathtt{rev([2|Xs],[1],Zs)} \arrow[d] & \\
& \mathtt{rev(Xs,[2,1],Zs)} \arrow[dl, "\{\mathtt{Xs}\mapsto[],\mathtt{Zs}\mapsto[1,2]\}"'] \arrow[dr, "\{\mathtt{Xs}\mapsto[\mathtt{X'}\mid\mathtt{Xs'}]\}"] & \\
\mathtt{true} & & \boxed{\mathtt{rev(Xs',[X',2,1],Zs)}}
\end{tikzcd}$$

It can be observed that the call in the frame rev(Xs′, [X′, 2, 1], Zs) embeds the previous call rev(Xs, [2, 1], Zs), hence the derivation is stopped. Such call is said to be transferred to the global control in the sense that it is returned by unfold as an element of L^{pe} and hence it is passed away as an argument to abstract.

The partial evaluator may have to build several SLD-trees to ensure that all calls left in the leaves (L^{pe} in Line 4) are "covered" by the root of some tree. This is known as the *closedness* condition of PE [33]. E.g., after having built the first SLD-tree for the call rev([1, 2|Xs], [], Zs), the call rev(Xs′, [X′, 2, 1], Zs) is not covered by rev([1, 2|Xs], [], Zs) because it is not an instance of it. At this point the abstract operator adds the framed call to the new set of atoms to be partially evaluated. At the next iteration, the following SLD-tree is built for such call:

$$\begin{tikzcd}
& \mathtt{rev(Xs,[X',2,1],Zs)} \arrow[dl, "\{\mathtt{Xs}\mapsto[],\mathtt{Zs}\mapsto[\mathtt{X'},1,2]\}"'] \arrow[dr, "\{\mathtt{Xs}\mapsto[\mathtt{X''}\mid\mathtt{Xs'}]\}"] & \\
\mathtt{true} & & \boxed{\mathtt{rev(Xs',[X'',X',2,1],Zs)}}
\end{tikzcd}$$

Thus, basically, the algorithm iteratively (Lines 3-7) constructs partial SLD trees until all their leaves are covered by the root nodes. An essential point of the operator abstract is that it has to perform "generalizations" on the calls that have to be partially evaluated in order to avoid computing partial SLD trees for an infinite number of calls. The homeomorphic embedding can be again used here to ensure termination and detect which calls have to be generalized. A classical way of performing generalizations is to use the *most-specific generalizer* operator (*msg* for short) in the following way. Suppose that a call A is to be added to the set S_k, and that there is a call B in S_k s.t. A embeds B, then the *msg* of A and B is added to the set S_{k+1} (and usually B is removed). In the example, the framed call rev(Xs, [X′, X′′, 2, 1], Zs) embeds rev(Xs, [X, 2, 1], Zs) (also framed), therefore both are generalized using the *msg* resulting in rev(Xs, [A, B, C|D], Zs). The generalized call is added to the set S_{i+1} and rev(Xs, [X, 2, 1], Zs)) removed. At the next iteration, the following SLD tree is built for the generalized atom:

$$\begin{tikzcd}
& \mathtt{rev(Xs,[A,B,C|D],Zs)} \arrow[dl, "\{\mathtt{Xs}\mapsto[],\mathtt{Zs}\mapsto[\mathtt{A},\mathtt{B},\mathtt{C}\mid\mathtt{D}]\}"'] \arrow[dr, "\{\mathtt{Xs}\mapsto[\mathtt{X'}\mid\mathtt{Xs'}]\}"] & \\
\mathtt{true} & & \boxed{\mathtt{rev(Xs',[X',A,B,C|D],Zs)}}
\end{tikzcd}$$

Without such generalization, the algorithm would keep on adding calls rev(Xs, [X, X′, X′′, 2, 1], Zs), rev(Xs, [X, X′, X′′, X′′′, 2, 1], Zs),. . . infinitely.

A partial evaluation of P w.r.t. S is then systematically extracted from the resulting set of calls L^{pe} in the final phase, codegen in L8. The notion of *resultant* is used to generate a program rule associated to each root-to-leaf derivation of the SLD-trees for the final set of atoms L^{pe}. Given an SLD derivation of $P \cup \{A\}$ with $A \in L^{pe}$ ending in B and θ being the composition of the mgu's in the derivation steps, the rule $\theta(A) : -B$ is called the *resultant* of the derivation. A PE is defined as the set of resultants (clauses) associated to the derivations of the constructed partial SLD trees for all $P \cup L^{pe}$. The resulting program is often referred to as the *specialized program* or *residual program*. In the example, the final set L^{pe} contains the calls $\texttt{rev([1, 2|Xs], [], Zs)}$ and $\texttt{rev(Xs, [A, B, C|D], Zs)}$ from which the following PE (residual program) is generated:

```
rev([1,2],[],[2,1]).
rev([1,2,A|B],[],C) :- rev_1(B,[A,2,1],C).
rev_1([],[A,B,C|D],[A,B,C|D]).
rev_1([A|B],[C,D,E|F],G) :- rev_1(B,[A,C,D,E|F],G).
```

The first two resultants are obtained from each derivation (branch) of the first tree above and the last two ones from the last tree above.

It can be also observed that a post-processing of renaming has been performed by codegen as explained below. Such a post-processing use to perform in addition some form of *argument filtering* [32]. This is because automatically generated programs, and in particular those generated by PE, very often contain redundant arguments which do not affect the correctness of the program. Throughout the rest of the paper we will consider a codegen function which is able to remove arguments which are actually not used in any computation but rather just passed around.

2.3. Correctness of Partial Deduction

Intuitively, the notions of, respectively, *completeness* and *correctness* of PE [16] ensure that the specialized program produces no less, respectively, no more answers than the original program. A sufficient condition to ensure completeness is that the specialized program is *closed* by the resulting set of atoms L^{pe}. As informally explained in Section 2.2, the closedness condition ensures that all calls which may arise during the computation of $P \cup S$ are instances of L^{pe} and hence there is a matching resultant for them (solutions are not lost).

Definition 2 (closedness). *Let T and S be two sets of atoms. Then, S is T-closed iff each atom in S is an instance of an atom in T. Given a program P and a set of atoms T, we say that $P \cup T$ is S-closed if the set of atoms which occur in the computation of $P \cup T$ are S-closed.*

The abstraction operator ensures that the closedness condition is met by means of a proper generalization of calls. For instance, as the set of atoms $\{\texttt{rev(Xs, [X', X'', 2, 1], Zs)}\}$ is not closed w.r.t. this set $\{\texttt{rev(Xs, [X, 2, 1], Zs)}\}$, the abstraction operator has generalized both terms to the term $\texttt{rev(Xs, [A, B, C|D], Zs)}$ which covers both terms.

Let us see an example where the closedness condition does not hold and hence we lose completeness. Consider a program defined by these two clauses:

```
p(X) :- q(X).
q(X).
```

The following partially evaluated program has been obtained by specializing the above program w.r.t. the set of atoms $S = \{\texttt{q(a)}\}$:

```
p(X) :- q(X).
q(a).
```

The closedness condition w.r.t. the set S does not hold because the atom in the left-hand side of the first rule is not an instance of any atom in S. It can be seen that the partially evaluated program is not complete since the goal p(b) succeeds in the original program while it fails in the residual one.

Correctness is achieved when the resulting set L^{pe} is independent, i.e., there are no two calls in L^{pe} which unify.

Definition 3 (independence). *Let S be a set of atoms. Then, S is* independent *if no pair of atoms in S have a common instance.*

Let us see an example where the independence condition does not hold and hence we lose correctness. Consider again the above program and the set of atoms $S = \{\mathtt{q(X)}, \mathtt{q(a)}\}$ which is not independent. The following program is a partial evaluation w.r.t. the set S:

```
p(X) :- q(X).
q(X).
q(a).
```

It can be seen that the residual program produces more answers than the original one. In particular, for the goal q(Y) it returns two answers $\{\mathtt{Y} \mapsto \mathtt{X}\}$ and $\{\mathtt{Y} \mapsto \mathtt{a}\}$ while the original program generates only the first one.

Independence can be recovered by a post-processing of renaming [16]. In the previous program, the two atoms in S could be renamed as $\mathtt{q}_1(\mathtt{X})$ and $\mathtt{q}_2(\mathtt{a})$ and the residual program would contain one clause defining $\mathtt{q}_1$ and another one for $\mathtt{q}_2$. In addition, renaming has benefits for performance because it reduces the number of rules per predicate. Thus, though the calls in L^{pe} for our running example are independent, we rename the second call for predicate rev to rev_1.

Theorem 1 (correctness). *Let P be a program, L^{pe} be a finite, independent set of atoms and P' be a partial evaluation of L^{pe} in P. For every goal G such that $P' \cup \{G\}$ is L^{pe}-closed, the following conditions hold:*

- Soundness: *$P' \cup \{G\}$ has a successful derivation with answer θ only if $P \cup \{G\}$ does.*
- Completeness: *$P \cup \{G\}$ has a successful derivation with an answer θ only if $P' \cup \{G\}$ does.*

The above theorem is proven in early work on PE of logic programs [33, 25].

2.4. Online vs. Offline Partial Deduction

It is well-known that both the quality of the specialized programs and the time required for the PE process greatly vary with the control strategies used. Traditionally, two approaches to PE have been considered, *online* and *offline* PE. In online PE, all control decisions are taken on the fly during the specialization phase by keeping track of the specialization history. This is the case of the control rules used in the example of Section 2.2. In the offline approach, all control decisions are taken before the proper specialization phase. These control decisions are based on abstract descriptions of the data instead of the actual data. The control strategy is usually represented as program annotations which are the sole decision criteria for control of the partial evaluator. For instance, in the local control, an annotation can explicitly indicate that an atom should not be unfolded. In the global control, annotations typically specify for each call which arguments have to be generalised away (i.e. replaced by variables). Such annotations are in some partial evaluators automatically generated by a *binding-time analysis* and in other partial evaluators they are manually provided by the user, either in part or in full.

Under this classification, the PE algorithm we propose can be considered a hybrid approach since the $\mathcal{A}$ annotations provide information to the control operators, as in offline PE, and the algorithm includes control rules based on the actual specialization history, as in online PE. The advantages of the offline approach are that, once all control annotations are available, PE is quite simple and efficient. On the other hand, online PE, though less efficient, has a strictly more powerful control strategy since control decisions are based on actual data instead of abstract descriptions of data. Therefore, though all offline PEs can be replicated using online techniques, many online PEs cannot be reproduced using offline techniques.

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

3. The Interpretive Approach to Compilation

The development of PE, program specialization and related techniques [14, 23, 15] has led to an alternative approach to compilation (known as the first Futamura projection) based on specializing an interpreter with respect to a fixed object program. Let us explain intuitively the interpretive approach. We denote by `mix` a generic partial evaluator, by `p` a program and by `in1` (resp. `in2`) the static (resp. dynamic) input data. Given a program $\mathtt{P}$, we write $\mathtt{P}_{\mathtt{L}}$ to denote that $\mathtt{P}$ is written in language $\mathtt{L}$. When the program is a meta-program, we write as a super-index the language the meta-program manipulates. For instance, $\mathtt{mix}_{\mathtt{S}}^{\mathtt{L}}$ denotes a partial evaluator, written in $\mathtt{S}$, which manipulates programs written in $\mathtt{L}$. We omit the languages (both sub. and super-indexes) when they are not relevant. Finally, we use the notation $[[\mathtt{P}]][\mathtt{d}]$ to denote the execution of $\mathtt{P}$ with input data $\mathtt{d}$. A partial evaluator can be defined as a program which behaves as follows:

$$\begin{array}{rcl} \mathtt{p\_in1} & = & [[\mathtt{mix}]]\,[\mathtt{p},\mathtt{in1}] \\ \mathtt{output} & = & [[\mathtt{p\_in1}]]\,[\mathtt{in2}] \;\; = \;\; [[\mathtt{p}]]\,[\mathtt{in1},\mathtt{in2}] \end{array}$$

Essentially, the execution of `mix` for `p` and `in1` returns a specialized program `p_in1` whose execution for the dynamic data `in2` must be the same as executing the original program `p` w.r.t. all dynamic plus static data $[\mathtt{in1},\mathtt{in2}]$. This implies the following:

$$[[\mathtt{p}]]\,[\mathtt{in1},\mathtt{in2}] = [[\;[[\mathtt{mix}]]\,[\mathtt{p},\mathtt{in1}]\;]]\,[\mathtt{in2}] \qquad (1)$$

This means that the result of PE is a program which is semantically equivalent w.r.t. the original for the static data. We now define by means of equations the behavior of an interpreter $\mathtt{int}_{\mathtt{L}}^{\mathtt{S}}$, which interprets programs written in $\mathtt{S}$, and is written in (a possibly different) language $\mathtt{L}$:

$$\mathtt{output} = [[\mathtt{source}_{\mathtt{S}}]]\,[\mathtt{d}] = [[\mathtt{int}_{\mathtt{L}}^{\mathtt{S}}]]\,[\mathtt{source}_{\mathtt{S}},\mathtt{d}] \qquad (2)$$

This captures the idea that executing a source program `source` for some input data `d` in the interpreter gives the same output as the execution of the program yields. Similarly, we can define a compiler $\mathtt{comp}_{L}^{S\to T}$ from $\mathtt{S}$ to $\mathtt{T}$ written in (a possibly different) language $\mathtt{L}$ as follows:

$$\begin{array}{rcll} [[\mathtt{source}_{\mathtt{S}}]]\,[\mathtt{d}] & = & [[\;[[\mathtt{comp}_{\mathtt{L}}^{\mathtt{S}\to\mathtt{T}}]]\,[\mathtt{source}_{\mathtt{S}}]\;]]\,[\mathtt{d}] & (3) \\ [[\mathtt{comp}_{\mathtt{L}}^{\mathtt{S}\to\mathtt{T}}]]\,[\mathtt{source}_{\mathtt{S}}] & = & \mathtt{target}_{\mathtt{T}} & \end{array}$$

Consider now a partial evaluator $\mathtt{mix}_L^T$ (written in $\mathtt{L}$) for programs written in $\mathtt{T}$, and an interpreter $\mathtt{int}_\mathtt{T}^\mathtt{S}$ (written in $\mathtt{T}$) for programs written in $\mathtt{S}$. Now, the compilation of a program $\mathtt{source}_\mathtt{S}$ to a program $\mathtt{target}_\mathtt{T}$ by using a partial evaluator $\mathtt{mix}_L^T$ can be performed as follows:

$$\mathtt{target}_\mathtt{T} = [[\mathtt{mix}_\mathtt{L}^\mathtt{T}]]\,[\mathtt{int}_\mathtt{T}^\mathtt{S}, \mathtt{source}_\mathtt{S}]$$

which is justified by combining equations (1) and (2) in this way:

$$[[\mathtt{p}_\mathtt{S}]]\ \mathtt{d} = [[\mathtt{int}_\mathtt{T}^\mathtt{S}]]\ [\mathtt{p}, \mathtt{d}] = [[\ [[\mathtt{mix}_\mathtt{L}^\mathtt{T}]]\ [\mathtt{int}_\mathtt{T}^\mathtt{S}, \mathtt{p}]\ ]]\ \mathtt{d}$$

Now, by comparing the above equation with equation (3), it can be observed the essence of compilation by means of PE of interpreters: we obtain the compilation of the program $\mathtt{p}$ written in $\mathtt{S}$ into another language $\mathtt{T}$. The application of this interpretative approach to compilation within our framework consists in specializing a bytecode ($\mathtt{BC}$) interpreter $\mathtt{int}_\mathtt{LP}^\mathtt{BC}$ written in logic programming $\mathtt{LP}$ where the static data is the actual bytecode program $\mathtt{p}_\mathtt{BC}$ to be decompiled:

$$[[\mathtt{p}_\mathtt{BC}]]\ \mathtt{d} = [[\mathtt{int}_\mathtt{LP}^\mathtt{BC}]]\ [\mathtt{p}_\mathtt{BC}, \mathtt{d}] = [[\ [[\mathtt{mix}^\mathtt{LP}]]\ [\mathtt{int}_\mathtt{LP}^\mathtt{BC}, \mathtt{p}_\mathtt{BC}]\ ]]\ \mathtt{d} = [[\mathtt{p}_\mathtt{LP}]]\ \mathtt{d}$$

It can be observed that the result is a decompiled program $\mathtt{p}_\mathtt{LP}$ in $\mathtt{LP}$ which, given the actual input data $\mathtt{d}$ produces the same result as the original program $\mathtt{p}_\mathtt{BC}$.

4. Non-Modular Interpretive Decompilation

This section describes the state of the art in interpretive decompilation of low-level languages to Prolog, including recent work in [20, 4, 18, 5]. We do so by formulating non-modular decompilation in a generic way and identifying its limitations.

4.1. The Bytecode Language

The bytecode language we consider, denoted as $\mathcal{L}_{bc}$, is a simple imperative bytecode language in the spirit of Java bytecode. To simplify the presentation, it does not include advanced features of Java bytecode such as exceptions, arrays, object-oriented features, access control (e.g. public, protected, private) and it manipulates only integer numbers. The extensions to consider such advanced features will be discussed later in Sections 7 and 8. As in Java bytecode, $\mathcal{L}_{bc}$ uses an operand stack to perform intermediate computations and an array of variables to store the formal parameters of the method and the actual method variables. Also, the global *heap* is not yet considered. Support for object-oriented features will be provided later. Finally, $\mathcal{L}_{bc}$, has an unstructured control flow, i.e., there are no explicit block markers, hence it includes explicit conditional and unconditional `goto` instructions.

A bytecode program P_{bc} consists of a set of methods which are the basic (de-)compilation units of $\mathcal{L}_{bc}$. The code of a method m, denoted $code(m)$, consists of a sequence of indexed bytecode instructions $\langle pc_0 : bc_0, \ldots, pc_{n_m} : bc_{n_m}\rangle$ with $pc_0, \ldots, pc_{n_m}$ being consecutive natural numbers. The $\mathcal{L}_{bc}$ instruction set is:

$$\begin{array}{rl} Inst_{\mathcal{L}_{bc}} ::= & \mathtt{push(x)} \mid \mathtt{load(v)} \mid \mathtt{store(v)} \mid \mathtt{add} \mid \mathtt{sub} \mid \mathtt{mul} \mid \mathtt{div} \mid \mathtt{rem} \mid \\ & \mathtt{neg} \mid \mathtt{if} \diamond \mathtt{(pc)} \mid \mathtt{if0} \diamond \mathtt{(pc)} \mid \mathtt{goto(pc)} \mid \mathtt{return} \mid \mathtt{invoke(mn)} \end{array}$$

where $\diamond$ is a comparison operator (eq, le, gt, etc.), v a local variable, x an integer, pc an instruction index and mn a method name. Instructions `push`, `load` and `store` transfer values or constants from the local variables to the stack (and vice-versa); `add`, `sub`, `mul`, `div`, `rem`

```
main(Method,InArgs,Top) :-
   build_s0(Method,InArgs,S0),
   execute(S0,Sf),
   Sf = st(fr(_,_,[Top|_],_),_)).
execute(S,S) :-
   S = st(fr(M,PC,[_Top|_],_),[]),
   bytecode(M,PC,return,_).
execute(S,Sf) :-
   S = st(fr(M,PC,_,_),_),
   bytecode(M,PC,Inst,_),
   step(Inst,S,S'),
   execute(S',Sf).

step(push(X),S,S') :-
   S = st(fr(M,PC,OS,L),FrS),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[X|OS],L),FrS).
step(add,S,S') :-
   S = st(fr(M,PC,[X,Y|OS],L),FrS),
   next(M,PC,PC'), Z is X + Y,
   S' = st(fr(M,PC',[Z|OS],L),FrS).
```

```
step(goto(PC),S,S') :-
   S = st(fr(M,_,OS,LV),FrS),
   S' = st(fr(M,PC,OS,LV),FrS).
step(load(I),S,S') :-
   S = st(fr(M,PC,OS,L),FrS),
   next(M,PC,PC'), nth(L,I,V),
   S' = st(fr(M,PC',[V|OS],L),FrS).
step(store(I),S,S') :-
   S = st(fr(M,PC,[V|OS],L),FrS),
   next(M,PC,PC'), replace_nth(L,I,V,L'),
   S' = st(fr(M,PC',OS,L'),FrS).
...
step(invoke(M'),S,S') :-
   S = st(fr(M,PC,OS,LV),FrS),
   split_OS(M',OS,Args,OS''),
   build_s0(M',Args,st(fr(M',PC',OS',LV'),_)),
   S' = st(fr(M',PC',OS',LV'),
           [fr(M,PC,OS'',LV)|FrS]).
step(return,S,S') :-
   S = st(fr(_,_,[RV|_],_),[fr(M,PC,OS,LV)|FrS]),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[RV|OS],LV),FrS).
```

Figure 1: Fragment of (small-step) $\mathcal{L}_{bc}$ interpreter

and `neg` perform the usual arithmetic operations, being `rem` the division remainder and `neg` the arithmetic negation; `if` and `if0` are conditional branching instructions (with the special case of comparisons with 0); `goto` is an unconditional branching; `return` marks the end of methods and `invoke` invokes a method. For simplicity, all methods are supposed to return an integer value. A method m is uniquely determined by its name. We write $calls(m)$ to denote the set of all method names invoked within the code of m. We use $defs(P_{bc})$ to denote the set of *internal* method names defined in P_{bc}. The remaining methods are *external*. We say that P_{bc} is *self-contained* if $\forall m \in P_{bc}, calls(m) \subseteq defs(P_{bc})$, i.e., P_{bc} does not include calls to external methods.

Though very simple, $\mathcal{L}_{bc}$ will be enough for our purposes when presenting the main ideas of the different decompilation schemes. Nevertheless it will be gradually extended as needed when we present more advanced features until the point of covering the full Java bytecode language in the experimental evaluation in Section 8.

4.2. Non-modular, Online Decompilation

We rely on the interpretive approach to compilation by PE described in Section 3. As it has been already explained, the decompilation of a $\mathcal{L}_{bc}$-bytecode program P_{bc} to LP (for short LP-decompilation) might be obtained by specializing (with an LP partial evaluator) a $\mathcal{L}_{bc}$-interpreter written in LP w.r.t. P_{bc}. In Fig. 1 we show a fragment of a (small-step) $\mathcal{L}_{bc}$ interpreter implemented in Prolog, named $Int_{\mathcal{L}_{bc}}$. We assume that the code for every method in the bytecode program P_{bc} is represented as a set of facts `bytecode/3` such that, for every pair $pc_i : bc_i$ in the code for method m, we have a fact `bytecode(`m, pc_i, bc_i`)`. The state carried around by the interpreter is of the form `st(Fr,FrStack)` where `Fr` represents the current frame (environment) and `FrStack` the stack of frames (call stack) implemented as a list. Frames are of the form `fr(M,PC,OStack,LocalV)`, where `M` represents the current method, `PC` the program counter, `OStack` the operand stack and `LocalV` the list of local variables. Predicate `main/3`, given the method to be interpreted `Method` and a concrete input (method arguments) `InArgs`, first builds the initial state by means of predicate `build_s0/3` and then calls predicate `execute/2`. In turn, `execute/2` calls predicate `step/3`, which produces `S'`, the state after executing the bytecode, and then calls predicate `execute/2` recursively with `S'` until we reach a `return` instruction with the empty stack. For brevity, we only show the definition of `step/3` for a selected set of instructions and omit the code of some auxiliary predicates. Namely `build_s0/3`, which was explained below, `next/3`, which produces the next program counter given the current one, `nth/3` and `replace_nth/4`, which respectively get and set the i-th element of a list, and `split_OS/4`, which splits the current operand stack between the parameters list to be used in the called method and the rest. By using this interpreter, we define a *non-modular* decompilation scheme in terms of the generic function *PE* as follows:

Definition 4 ($\textsc{decomp}_{\mathcal{L}_{bc}}$). *Given a self-contained $\mathcal{L}_{bc}$-bytecode program P_{bc}, the (non-modular) LP-decompilation of P_{bc} can be obtained as:*

$$\textsc{decomp}_{\mathcal{L}_{bc}}(P_{bc}) = PE(\mathrm{Int}_{\mathcal{L}_{bc}} \cup P_{bc}, \emptyset, S)$$

where S is the set of calls $\{main(m, \_, \_) \mid m \in \mathrm{defs}(P_{bc})\}$.

Observe in the above definition that the set of annotations is empty. Following the PE terminology, the above definition corresponds to *online* PE as we have explained in Section 2.4.

Recent work in interpretive, online decompilation has focused on ensuring that the layer of interpretation is completely removed from decompiled programs, i.e., *effective* decompilations are obtained. This requires the use of the following advanced control techniques. Type-based homeomorphic embedding ($\trianglelefteq_T$) [4] has been used both at the local and global control to decide when to stop derivations and when to generalize calls so that effectiveness of the decompilation can be obtained in the presence of integers without compromising termination. The unfolding operator must also be able to accurately handle built-in predicates and to safely perform non-leftmost unfolding steps as in [6]. The operator $\mathsf{abstract}$ must incorporate a polyvariance control mechanism [18] which avoids performing useless specializations that can introduce superfluous decompiled code and thus degrade the decompilation effectiveness. Our starting point is thus a state-of-the-art online partial evaluator based on an unfolding operator $\mathsf{unfold}_{\trianglelefteq_T}$ and abstraction operator $\mathsf{abstract}_{\trianglelefteq_T}$ which incorporate such advanced techniques and is able to remove the layer of interpretation. Such advanced partial evaluator is used in the following both for running examples and experiments.

4.3. Limitations

This section illustrates by means of the bytecode example in Fig. 2 that non-modular decompilation does not ensure a satisfactory handling of issues *(a)* and *(b)* in Section 1. In the examples, we often depict the Java source code for clarity, but the decompiler works directly on the bytecode. The program consists of a set of methods that carry out arithmetic operations. Method `gcd` computes the greatest-common divisor, `abs` the absolute value, `lcm` the least-common multiple and `fact` the factorial recursively. The LP-decompilation obtained by applying Definition 4 is shown in Fig. 3. The partial evaluator performs a post-processing of renaming and argument filtering [16] for all calls except for calls to the `main` predicate (as they represent calls to methods whose name we want to preserve). We identify below four limitations, which we identify as

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

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

```
int lcm(int x,int y){
   int gcd = gcd(x,y);
   if (gcd == 0) return 0;
   else return x*y/gcd;}

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

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

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

```
Method lcm
 0:load(0)
 1:load(1)
 2:invoke(gcd)
 3:store(2)
 4:load(2)
 5:if0ne 8
 6:push(0)
 7:return
 8:load(0)
 9:load(1)
 10:mul
 11:load(2)
 12:div
 13:return
```

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

Figure 2: Source code and $\mathcal{L}_{bc}$-bytecode for working example

(L1). . . **(L4)**, of non-modular decompilation. It is important to note that such limitations, and the way to avoid them which we propose in Section 5 below, are also relevant to the case of offline PE.

(L1) Calls to methods are *inlined* within their calling contexts and, as a consequence, the structure of the original code is lost. For example, method invocations from `lcm` to `gcd` (index 2) and from `gcd` to `abs` (index 12) do not appear in the decompiled code. As a result, the last two rules in the decompilation for `lcm`, `execute_1`, correspond to the `while` loop of `gcd`. This happens because calls to methods are dealt with in a *small-step* fashion within the interpreter, i.e., the code of invoked methods is unfolded as if it was inlined inside the "caller" method.

(L2) As a consequence, decompilation becomes very inefficient. E.g., if *n* calls to the same method appear within a code, such method will be decompiled *n* times. Even worse, if there is a method invocation inside a loop, its code will be evaluated twice in the best case, as we have to perform the corresponding generalizations in the global control before reaching a fixpoint, as in the example of Section 2.2. This is worse in the case of nested loops.

(L3) The non-modular approach does not work incrementally, in the sense that it does not support *separate* decompilation of methods but rather has to (re)decompile all method calls.

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

execute_1(A,0,B,C,D) :- A>0, E is B*C, D is E//A.
execute_1(0,0,_,_,0).
execute_1(A,0,B,C,D) :- A<0, E is -A,
                        F is B*C, D is F//E.
execute_1(A,B,C,D,I) :- B\=0, K is A rem B,
                        execute_1(B,K,C,D,I).
```

```
main(gcd,[A,0],A) :- A>=0.
main(gcd,[B,0],A) :- B<0, A is -B.
main(gcd,[B,C],A) :- C\=0,
                     D is B rem C,
                     execute_2(C,D,A).

execute_2(A,0,A) :- A>=0.
execute_2(A,0,C) :- A<0, C is -A.
execute_2(A,B,G) :- B\=0,
                    I is A rem B,
                    execute_2(B,I,G).

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

Figure 3: Decompiled (non-modular) code for working example

Thus, decompiling a real language becomes unfeasible, as one needs to consider system libraries, whose code might not be available. Limitation L2 together with L3 answer issue *(a)* negatively.

(L4) The decompiled code contains basically the whole interpreter when there are recursive methods. This is why the decompiled program in Figure 3 does not contain the code corresponding to the recursive `fact` method. The problem with recursion is as follows. Assume we want to decompile method $m1$ whose code is $\langle pc_0 : bc_0, \ldots, pc_j : invoke(m1), \ldots, pc_n : return\rangle$. There is an initial decompilation for $A_k = \mathtt{execute}(\mathtt{st}(\mathtt{fr}(\mathtt{m1}, \mathtt{pc_j}, \mathtt{os}, \mathtt{lv}), []), \mathtt{S_f})$ in which the call stack is empty. During its decompilation, a call of the form $A_l = \mathtt{execute}(\mathtt{st}(\mathtt{fr}(\mathtt{m1}, \mathtt{pc_j}, \mathtt{os}', \mathtt{lv}'),$ $[\mathtt{fr}(\mathtt{m1}, \mathtt{pc_j}, \mathtt{os}, \mathtt{lv})]), \mathtt{S_f})$ with the call stack containing the previous frame appears when we arrive to the recursive call. At this point, the derivation must be stopped as $A_k \trianglelefteq_T A_l$. In order to ensure termination, global control generalizes the above calls into $\mathtt{execute}(\mathtt{st}(\mathtt{fr}(\mathtt{m1}, \mathtt{pc_j}, \_, \_), \_),$ $S_f)$, where $\_$ denotes a fresh variable and thus the call-stack has become unknown. As a consequence, after evaluating the *return* statement, the continuation obtained from the call-stack is unknown and we produce the call $\mathtt{execute}(\mathtt{st}(\mathtt{fr}(\_, \_, \_, \_), \_), \mathtt{S_f})$ to be decompiled. Here, the fact that the method and the program counter are unknown prevents us from any chance of removing the interpretation layer, i.e., the decompiled code will potentially contain the whole interpreter. This indeed happens during the decompilation of `fact`. Partial solutions to the recursion problem exist and will be discussed later. Limitations L1 and L4 answer issue *(b)* negatively.

5. A Modular Decompilation Scheme

By *modular* decompilation, we refer to a decompilation technique whose decompilation unit is the method, i.e., we decompile a method at a time. We show that this approach overcomes the four limitations of non-modular decompilation described in Section 4.3 and answers issues *(a)* and *(b)* positively. In essence, we need to: (i) Give a compositional treatment to method invocations. We show that this can be achieved by considering a *big-step* interpreter. (ii) Provide a mechanism to residualize calls in the decompiled program (i.e. do not unfold them and add them without modifications to the residual code). We automatically generate program annotations for this purpose. (iii) Study the conditions which ensure that *separate* decompilation of methods is sound.

5.1. Big-step Semantics Interpreter to Enable Modularity

Traditionally, two different approaches have been considered to define language semantics, *big-step* (or *natural*) semantics and *small-step* (or *structural operational*) semantics (see, e.g., [26]). Essentially, in big-step semantics, transitions relate the initial and final states for each statement, while in small-step semantics transitions define the *next* step of the execution for each statement. In the context of bytecode interpreters, it turns out that most of the statements execute in a single step, hence making both approaches equivalent for such statements. This is the case for our $\mathcal{L}_{bc}$-bytecode interpreter for all statements except for *call*. The transition for *call* in small-step defines the next step of the computation, i.e., the current frame is pushed on the call-stack and a new environment is initialized for the execution of the invoked method. Note that, after performing this step, we do not distinguish anymore between the code of the caller method and that of the callee. This prevents us from having modularity in decompilation.

In the context of interpretive (de-)compilation of imperative languages, small-step interpreters are commonly used (see e.g. [39, 20, 5]). We argue that the use of a big-step interpreter is a necessity to enable modular decompilation which scales to realistic languages. In Fig. 4, we depict the relevant part of the big-step interpreter for $\mathcal{L}_{bc}$-bytecode, named $Int^{bs}_{\mathcal{L}_{bc}}$. We can see that the *call* statement, after extracting the method parameters from the operand stack, calls recursively predicate `main/3` in order to execute the callee. Upon return from the method execution, the return value is pushed on the operand stack of the new state and execution proceeds normally. Also, we do not need to carry the call-stack explicitly within the state, but only the information for the current environment, i.e., states are of the form `st(M,PC,OStack,LocalV)`. This is because the call-stack is already available by means of the calls for predicate `main/3`.

The compositional treatment of methods in $Int^{bs}_{\mathcal{L}_{bc}}$ is not only essential to enable modular decompilation (overcome L1, L2 and L3) but also to solve the recursion problem in a simple and elegant way. Indeed, the decompilation based on the big-step interpreter $Int^{bs}_{\mathcal{L}_{bc}}$ does not present L4. E.g., the decompilation of a recursive method $m1$ starts from the call `main(m1, _, _)` and then reaches a call `main(m1, args, _)` where `args` represents the particular arguments in the recursive call. This call is flagged as dangerous by local control and the derivation is stopped. The important points are that, unlike before, no re-computation is needed as the second call is necessarily an instance of the first one and, besides, there is no information loss associated to the generalization of the call-stack, as there is no stack. The recursion problem was first detected in [17] and a solution based on computing regular approximations during PE was proposed. Although feasible in theory, such technique might be too inefficient in practice and problematic to scale it up to realistic applications due to the overhead introduced by the underlying analysis. Another solution is proposed in [20], where a simpler control-flow analysis is performed before PE in order to collect all possible instructions which might follow the *return*. Such information may then be used to recover information lost by the generalization. This solution turns out to be also impractical for our purposes when considering realistic programs that make intensive use of library code (e.g. Java bytecode) as many continuations can follow. Our solution does not require the use of static analysis and, as our experiments show, does not pose scalability problems.

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

```
execute(S,S) :-                        
   S = st(M,PC,[_Top|_],_),            step(invoke(M'),S,S') :-
   bytecode(M,PC,return,_).               S = st(M,PC,OS,LV),
execute(S,Sf) :-                          next(M,PC,PC'),
   S = st(M,PC,_,_),                      split_OS(M',OS,Args,OSRs),
   bytecode(M,PC,Inst,_),                 main(M',Args,RV),
   step(Inst,S,S'),                       S' = st(M,PC',[RV|OSRs],LV).
   execute(S',Sf).
```

Figure 4: Fragment of big-step $\mathcal{L}_{bc}$ interpreter $Int^{bs}_{\mathcal{L}_{bc}}$

5.2. Guiding Online PE with Annotations

We now present the annotations we use to provide additional control information to PE. They are instrumental for obtaining the quality decompilation we aim at. We use the annotation schema: "[*Precond*] $\Rightarrow$ *Ann Pred*" where *Precond* is an optional precondition defined as a logic formula, *Ann* is the kind of annotation (*Ann* $\in$ {**memo**, **rescall**}) and *Pred* is a predicate descriptor, i.e., a predicate name and distinct free variables. Such annotations are used by local control when a call for *Pred* is found as follows:

- **memo**: The current call is not further unfolded and is later transferred to the global control to carry out its specialization separately. It is then replaced by a call to the specialized version.
- **rescall**: The current call is not further unfolded. Unlike calls marked **memo**, the current call is not transferred to the global control. Therefore the call is added to the residual code without modification.

In the following, we denote by $\mathsf{unfold}^{\mathcal{A}}_{\trianglelefteq_T}$ the unfolding operator of Section 2.2 enhanced to use the above annotations. We adopt the same names for the annotations as in offline PE [31] (**rescall** stands for residual call while **memo** stands for *memoise*, i.e., pass the call to the *memo* table[1]). However, in offline PE they are the *only* means to control termination while in our method they are only used to improve the accuracy in the local control. As another difference, in offline PE, **rescall** annotations are used only for builtins. In principle, their use for internal predicates could threaten PE-completeness if a call is residualized but it is not an instance of some call in the final set L^{pe} (i.e., it is not closed by L^{pe}). In the next section, we illustrate the importance of **rescall** annotations also for internal predicates to enable *separate* PE. The role of **memo** becomes important to control the structure of the decompiled programs as we will see in Section 6.

5.3. Modular Decompilation

In order to achieve modular decompilation, it is instrumental to allow performing *separate* decompilation. In the interpretive approach this requires being able to perform separate PE, i.e., to be able to specialize parts of the program independently and then join the specializations

[1]This is how the list of atoms to be partially evaluated, named L^{pe} in Section 2.2, is usually denoted in offline PE.

together to form the residual program. For instance, consider a self-contained logic program P partitioned in a set $\{P_1, \ldots, P_n\}$ of mutually disjoint subprograms which preserve predicate boundaries, i.e., for any predicate *pred* in P we have that all rules for *pred* are in the same partition P_j, for some $j \in \{1, \ldots, n\}$. Consider also the sets of terms $S_1, \ldots, S_n$ such that all calls in S_i correspond to predicates defined in P_i, $i = 1, \ldots, n$. We can now define $S = S_1 \cup \cdots \cup S_n$ and the usual notions of closedness and independence are applicable. A *separate* partial evaluation for P and S is obtained as the union of the individual specializations w.r.t. each corresponding set of calls, i.e., $\bigcup_{P_i \in P} PE(P_i, \emptyset, S_i)$. One additional difficulty for separate PE is related to the use of renaming for guaranteeing independence (see Definition 3), since renaming requires a global table which is not available when generating code for the individual subprograms. A simple strategy which we will use in our modular decompilation is to allow polyvariant specialization (i.e. multiple specialized versions per predicate) for calls to predicates locally defined in the subprogram P_i being partially evaluated, but to resort to monovariant specialization (i.e. only one specialized version per predicate) for predicates used across subprogram boundaries. Then, the renaming can use a local renaming table, which must guarantee that there will be no name clash with renamed calls from other subprograms.

We present now a modular decompilation scheme which, by combining the big-step interpreter with the use of **rescall** annotations, enables separate decompilation and ensures *correctness* (i.e., it is sound and complete w.r.t. internal methods).

Definition 5 (MOD-DECOMP$_{\mathcal{L}_{bc}}$). *Given a $\mathcal{L}_{bc}$-bytecode program P_{bc}, a modular LP-decompilation of P_{bc} can be obtained as:*

$$\textsc{mod-decomp}_{\mathcal{L}_{bc}}(P_{bc}) = \bigcup_{\forall m \in \text{defs}(P_{bc})} PE(\text{Int}^{bs}_{\mathcal{L}_{bc}} \cup \text{code}(m), \mathcal{A}_{mod}(m), S(m))$$

*where the set of annotations $\mathcal{A}_{mod}(m) = \{(m \in \text{calls}(m)) \Rightarrow$ **rescall** $main(m, \_, \_)\}$ and the initial set of calls $S(m) = \{main(m, \_, \_)\}$.*

Let us briefly explain the above definition. Now the function PE is executed once per method defined in P_{bc}, starting each time from a set of calls, S_m, which contains a call of the form `main(m, _, _)` for method `m`. The set $\mathcal{A}_{mod}$ contains a **rescall** annotation which affects all methods invoked (but not necessarily internal) inside P_{bc}. When a method invocation is to be decompiled, the call `step(invoke(m'),_,_)` occurs during unfolding. We can see that, by using the big-step interpreter in Fig. 4, a subsequent call `main(m',_,_)` will be generated. As there is a **rescall** annotation which affects all methods invoked in the program, such call is not unfolded but rather remains residual. If `m'` is internal, a corresponding decompilation from the call `main(m',_,_)` will be, or has already been, performed since function PE is executed for every method in P_{bc}. Thus, completeness is ensured for internal predicates.

Example 1. By applying function MOD-DECOMP$_{\mathcal{L}_{bc}}$ to the $\mathcal{L}_{bc}$-bytecode program in Fig. 2 we execute PE once for each of the four methods in the program. In each execution we specialize the interpreter w.r.t. the calls `main(fact,_,_)`, `main(gcd,_,_)`, `main(lcm,_,_)`, and `main(abs,_,_)`. We obtain the following LP-decompilation:

```
main(lcm,[B,C],A) :- main(gcd,[B,C],D),
                     D\=0,
                     E is B*C,
                     A is E//D.
main(lcm,[A,B],0) :- main(gcd,[A,B],0).

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

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

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

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

The structure of the original program w.r.t. method calls is preserved, as the residual predicate for `lcm` contains an invocation to the definition of `gcd`, which in turn invokes `abs`, as it happens in the original bytecode. Moreover, we now obtain an effective decompilation for the recursive method `fact` where the interpretive layer is completely removed without the need of any analysis. Thus, L1 and L4 have been successfully solved.

The following theorem ensures the correctness of modular decompilation for the big-step bytecode interpreter. Completeness can be ensured by excluding calls to external methods not defined in the bytecode. It is independent of the way the interpreter is defined, as the closedness condition for the internal methods is enforced by our definitions of $\mathcal{A}_{mod}$ and S_m. Soundness holds in the case of our interpreter, because the only calls which are transferred to the global control are instances of `main/3` and `execute/2` and their first argument is the method's name, which makes them mutually exclusive. A post-processing of renaming is thus optional, but it can be necessary to ensure that the independence condition is met for other interpreters.

Theorem 2 (correctness). *Consider a $\mathcal{L}_{bc}$-bytecode program P_{bc}, a concrete input I and the $\mathcal{L}_{bc}$-bytecode interpreter* $\mathrm{Int}^{bs}_{\mathcal{L}_{bc}}$. *Let P'_{bc} be the result of* $\textsc{mod-decomp}_{\mathcal{L}_{bc}}(P_{bc})$. *Then, A is an answer for $P'_{bc} \cup \{I\}$ iff A is an answer obtained running P_{bc} on* $\mathrm{Int}^{bs}_{\mathcal{L}_{bc}}$ *with input I.*

Proof 1. Let us first prove the completeness of modular decompilation. This requires to prove the closedness condition as stated in Definition 2. We first have to exclude calls to *external* predicates not defined in the bytecode for which we do not obtain an answer in P'_{bc}. Thus, we need to ensure closedness for the calls which have `rescall` annotations and are internal. For the remaining internal calls, closedness is already ensured by traditional PE (Theorem 1). We reason by contradiction. Consider a method invocation to m' which has a **rescall** annotation $true \Rightarrow$ **rescall** $main(m', \_, \_)$ but it is not covered by L^{pe}. This leads to a contradiction because, function PE is executed $\forall m \in defs(P_{bc})$, including m'. Thus, there is an initial call $\mathtt{main}(\mathtt{m}', \_, \_)$ in $S_{m'}$ and hence it is covered by the final set L^{pe}.

In order to prove the correctness of our modular decompilation scheme, the full code of the interpreter must be studied. Here we focus on proving independence as stated in Definition 3. In the case of $Int^{bs}_{\mathcal{L}_{bc}}$, it is implied by the facts that: 1) the only recursive definitions are `main/3` and `execute/2` and the remaining predicates are always evaluable (in the sense of [41]), 2) thus every call manipulated by the global control is an instance of `main/3` or `execute/2` and 3) all such instances include the method name in some of their (sub-)arguments, which makes them mutually exclusive and hence independent. Since we have proved independence and closedness of the resulting terms, by Theorem 1, we have the correctness of modular decompilation. □

We now characterize the notion of *modular-optimality* in decompilation which ensures that (1) only the code associated to internal methods is decompiled, thus, we can have external calls (e.g., to libraries) which are not decompiled and overcome L3; (2) and each method is decompiled only once and thus we overcome L2.

Proposition 1 (modular-optimality). *Given a $\mathcal{L}_{bc}$-bytecode program P_{bc}, function* MOD-DECOMP$_{\mathcal{L}_{bc}}$ *only decompiles the code corresponding to internal methods defined in P_{bc}, and the code of each method is decompiled once.*

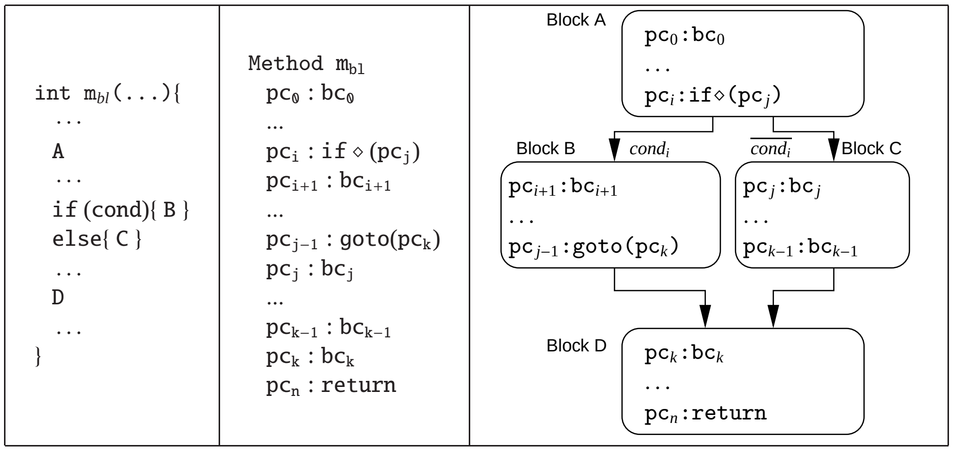
Proof 2. Only internal methods of P_{bc} are decompiled because all calls are annotated as **rescall** and hence they are not transferred to the global control. Then, we must prove that each method is decompiled once. The proof follows by contradiction. Assume that a method m is decompiled $n > 1$ times. This means that during the PE process, there have been n calls of the form $main(m, \_, \_)$ that have been unfolded. This leads to a contradiction as there is a **rescall** annotation which affects every method which is called in the program $main(m, \_, \_)$. This prevents from unfolding $main(m, \_, \_)$ and the result follows. $\square$

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

6. An Optimal Decompilation Scheme

The main issue is whether it is possible to obtain, by means of interpretive decompilation, programs whose *quality* is equivalent to that obtained by dedicated decompilers; issue *(c)* in Section 1. We will show now that, using the most effective unfolding strategies of PE, code for the same program point can be emitted (i.e. it can be decompiled) several times, which degrades both the efficiency and the quality of decompilation. In order to obtain results which are comparable to that of dedicated decompilers, it makes sense to use similar heuristics. Since decompilers first build a *control flow graph* (CFG) for the method, which guides the decompilation process, we now study how a similar notion can be used for controlling PE of the interpreter.

Let us explain *block-level* decompilation by means of an example. Consider the method m_{bl} in Fig. 5. The source code is shown to the left, the relevant bytecode in the center and its CFG to the right. As customary, the CFG [1] consists of basic blocks which contain a sequence of non-branching bytecode instructions and which are connected by edges which describe the possible flows originated from the branching instructions (like conditional jumps, exceptions, virtual method invocation, etc.). In our small language $\mathcal{L}_{bc}$, conditional jumps (i.e., `if`$\diamond$ and `if0`$\diamond$) are the only branching instructions. A *divergence point* (D point) is a program point (bytecode index) from which more than one branch originates; likewise, a *convergence point*

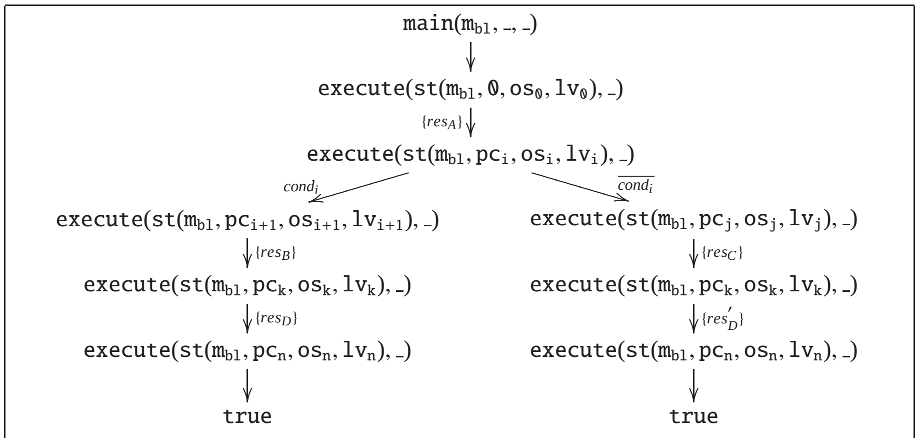


```
int m_bl(...){
  ...
  A
  ...
  if (cond){ B }
  else{ C }
  ...
  D
  ...
}
```

```
Method m_bl
  pc_0 : bc_0
  ...
  pc_i : if ◇ (pc_j)
  pc_i+1 : bc_i+1
  ...
  pc_j-1 : goto(pc_k)
  pc_j : bc_j
  ...
  pc_k-1 : bc_k-1
  pc_k : bc_k
  pc_n : return
```

Block A
$\mathtt{pc}_0$:$\mathtt{bc}_0$
...
$\mathtt{pc}_i$:if◇($\mathtt{pc}_j$)
Block B
$cond_i$
$\overline{cond_i}$
Block C
$\mathtt{pc}_{i+1}$:$\mathtt{bc}_{i+1}$
...
$\mathtt{pc}_{j-1}$:goto($\mathtt{pc}_k$)
$\mathtt{pc}_j$:$\mathtt{bc}_j$
...
$\mathtt{pc}_{k-1}$:$\mathtt{bc}_{k-1}$
Block D
$\mathtt{pc}_k$:$\mathtt{bc}_k$
...
$\mathtt{pc}_n$:return

Figure 5: Source code, $\mathcal{L}_{bc}$-bytecode and CFG of m_{bl} method



main($\mathtt{m_{bl}}$, _, _)
execute(st($\mathtt{m_{bl}}$, 0, $\mathtt{os_0}$, $\mathtt{lv_0}$), _)
$\{res_A\}$
execute(st($\mathtt{m_{bl}}$, $\mathtt{pc_i}$, $\mathtt{os_i}$, $\mathtt{lv_i}$), _)
$cond_i$
$\overline{cond_i}$
execute(st($\mathtt{m_{bl}}$, $\mathtt{pc_{i+1}}$, $\mathtt{os_{i+1}}$, $\mathtt{lv_{i+1}}$), _)
execute(st($\mathtt{m_{bl}}$, $\mathtt{pc_j}$, $\mathtt{os_j}$, $\mathtt{lv_j}$), _)
$\{res_B\}$
$\{res_C\}$
execute(st($\mathtt{m_{bl}}$, $\mathtt{pc_k}$, $\mathtt{os_k}$, $\mathtt{lv_k}$), _)
execute(st($\mathtt{m_{bl}}$, $\mathtt{pc_k}$, $\mathtt{os_k}$, $\mathtt{lv_k}$), _)
$\{res_D\}$
$\{res'_D\}$
execute(st($\mathtt{m_{bl}}$, $\mathtt{pc_n}$, $\mathtt{os_n}$, $\mathtt{lv_n}$), _)
execute(st($\mathtt{m_{bl}}$, $\mathtt{pc_n}$, $\mathtt{os_n}$, $\mathtt{lv_n}$), _)
true
true

```
main(m_bl,Args,Out) :- {res_A}, cond_i, {res_B}, {res_D}.
main(m_bl,Args,Out) :- {res_A}, ¬cond_i, {res_C}, {res'_D}.
```

Figure 6: Unfolding SLD-tree and decompiled code of m_{bl} method

(C point) is a program point where two or more branches merge. In the CFG of m_{bl}, the only divergence (resp. convergence) point is pc_i (resp. pc_k).

By using the decompilation scheme presented so far, we obtain the SLD-tree shown in Fig. 6, in which all calls are completely unfolded as there is no termination risk (nor **rescall** annotation). The decompiled code is shown under the tree. We use $\{\mathtt{res_X}\}$ to refer to the residual code emitted for BlockX and $\mathtt{cond_i}$ to refer to the condition associated to the branching instruction at $\mathtt{pc_i}$ ($\overline{\mathtt{cond_i}}$ denotes its negation). The quality of the decompiled code is not optimal due to:

D. Decompiled code $\{\mathtt{res_A}\}$ for BlockA is duplicated in both rules. During PE, this code is evaluated once but, due to the way resultants are defined (see codegen in Section 2.2), each rule contains the decompiled code associated to the whole branch of the tree. This code duplication brings in two problems: it increases considerably the size of decompiled programs and also makes their execution slower. For instance, when $\overline{\texttt{cond}_\texttt{i}}$ holds, the execution goes unnecessarily through $\{\texttt{res}_\texttt{A}\}$ in the first rule, fails to prove $\texttt{cond}_\texttt{i}$ and, then, attempts the second rule.

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

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

The **memo** annotations presented in Section 5.2 facilitate the design of the optimal interpretive decompilation scheme. In particular, we can easily force the unfolding process to stop at D points by including a **memo** annotation for `execute/2` calls whose *PC* corresponds to a D point. In the example, unfolding stops at pc_i as desired. Regarding C points, an additional requirement is to partially evaluate the code on blocks starting at these points at most once. The problem is similar to the polyvariant vs monovariant treatment in the decompilation of methods in Section 5.3, by viewing entries to blocks as method calls. Not surprisingly, the solution can be achieved similarly in our setting by: (1) stopping the derivation at `execute/2` calls whose *PC* corresponds to C points and (2) passing the call to the global control, and ensuring that it is evaluated in a sufficiently generalized context which covers all incoming contexts. The former point is ensured by the use of **memo** annotations and the latter by including in the initial set of atoms a generalized call of the form $\texttt{execute}(\texttt{st}(\texttt{m}_\texttt{bl}, \texttt{pc}_\texttt{k}, \_, \_), \_)$ for all C points, which forces such generalization. The next definition presents the optimal decompilation scheme where div_points(m) and conv_points(m) denote, respectively., the set of D points and C points of a method m.

Definition 6 (OPTIMAL-DECOMP$_{\mathcal{L}_{bc}}$). *Given a $\mathcal{L}_{bc}$-bytecode program P_{bc}, an optimal, modular LP-decompilation of P_{bc} can be obtained as:*

$$\textsc{optimal-decomp}_{\mathcal{L}_{bc}}(P_{bc}) = \bigcup_{\forall m \in \text{defs}(P_{bc})} PE(\text{Int}^{bs}_{\mathcal{L}_{bc}} \cup \text{code}(m), \mathcal{A}_{opt}(m), S(m))$$

$$\begin{array}{lcl}
\mathcal{A}_{blocks}(m) & = & \{pc \in \mathsf{div\_points}(m) \cup \mathsf{conv\_points}(m) \Rightarrow \mathbf{memo}\ execute(st(m, pc, \_, \_), \_)\} \\
S(m) & = & \{main(m, \_, \_)\} \cup \{execute(st(m, pc, \_, \_), \_) \mid pc \in \mathsf{conv\_points}(m)\} \\
\mathcal{A}_{opt}(m) & = & \mathcal{A}_{mod}(m) \cup \mathcal{A}_{blocks}(m)
\end{array}$$

An important point is that, unlike annotations used in offline PE [29] which are generated by only taking the interpreter into account, our annotations for the optimal decompilation are generated by taking into account the particular program to be decompiled. Importantly, both the annotations and the initial set of calls can be computed automatically by performing two passes on the bytecode (see, e.g., [2, 43]).

The result of performing an optimal decompilation on m_{bl} is:

$$\begin{array}{l}
\texttt{main(m}_{\texttt{bl}}\texttt{,Args,Out) :- }\{\texttt{res}_{\texttt{A}}\}, \texttt{execute}_1(\ldots). \\
\texttt{execute}_1(\ldots)\texttt{ :- }\texttt{cond}_{\texttt{i}}, \{\texttt{res}_{\texttt{B}}\}, \texttt{execute}_2(\ldots). \\
\texttt{execute}_1(\ldots)\texttt{ :- }\overline{\texttt{cond}_{\texttt{i}}}, \{\texttt{res}_{\texttt{C}}\}, \texttt{execute}_2(\ldots). \\
\texttt{execute}_2(\ldots)\texttt{ :- }\{\texttt{res}_{\texttt{D}}\}.
\end{array}$$

Now, the residual code associated to each block appears once in the code. This ensures that the optimal decompilation preserves the CFG shape as dedicated decompilers do. Thus, the quality of our decompiled code is as good as that obtained by state-of-the-art decompilers [2, 35] but with the advantages of interpretive decompilation (see Section 1). We formally prove the quality of our proposed decompilation scheme in the next proposition.

Proposition 2 (block-optimality). *Given a bytecode program P_{bc}, the optimal decompilation function* OPTIMAL-DECOMP$_{\mathcal{L}_{bc}}$ *ensures that: (I) residual code for each bytecode instruction in P_{bc} is emitted once in the decompiled program, (II) each bytecode instruction in P_{bc} is evaluated at most once during PE and (III) there is at most one residual rule for each block in the bytecode.*

Proof 3. The proof follows easily by contradiction.

In order to prove (I), consider that two resultants contain residual code for the same bytecode instruction. This can be due to two reasons. (a) There is in the SLD-tree a D point which leads to two derivations. This is not possible because D points are annotated as **memo** and hence the derivation must have been stopped. (b) There are two separate trees which contain derivations for instructions of the same block. Then, this block must be a C block. Hence, it is not possible because C points are annotated as **memo** and hence the derivation must have stopped before.

We focus now on D blocks to prove (II). Consider that there have been two evaluations of an instruction pc_x within a D block B starting at $pc_1 \in \mathsf{conv\_points}(M)$. Then, there must have been two different instances $\texttt{execute}(\texttt{st}(\texttt{M}, \texttt{pc}_1, \texttt{A}, \texttt{B}), \texttt{C}))$ and, later, $\texttt{execute}(\texttt{st}(\texttt{M}, \texttt{pc}_1, \texttt{D}, \texttt{E}), \texttt{F}))$. This is not possible because there exists the initial call $\texttt{execute}(\texttt{st}(\texttt{M}, \texttt{pc}_1, \_, \_), \_))$ in S_m which does not allow the evaluation of $\texttt{execute}(\texttt{st}(\texttt{M}, \texttt{pc}_1, \texttt{D}, \texttt{E}), \texttt{F}))$.

For (III) to be false there must exist a block in the CFG which includes a sequence of bytecode instructions $\langle pc_1 : b_1, \ldots, pc_i : b_i, \ldots, pc_n : b_n \rangle$, with $i \geq 2$ and $n \geq i$ such that the residual program contains a rule for the subsequence of bytecode instructions $\langle pc_i : b_i, \ldots, pc_j : b_j \rangle$ with $i \in \{2, \ldots, n\}$ and $j \in \{i, \ldots, n\}$. This requires that the local control stops unfolding for a call of the form $\texttt{execute}(\texttt{st}(\texttt{M}, \texttt{pc}_\texttt{i}, \texttt{A}, \texttt{B}), \texttt{C}))$. According to our optimal local control strategy, execution of a bytecode instruction is only left residual if the instruction at position pc_i in method M is a C point or a D point, which contradicts the assumption that the sequence of instructions $\langle pc_1 : b_1, \ldots, pc_i : b_i, \ldots, pc_n : b_n \rangle$ belongs to the same block in the CFG. □

After taking into account the central observation from Section 5 that the interpreter should be written in big-step semantics, each of the optimality criteria above is simpler or more complicated to achieve depending on the local control strategy we use. For example, if we start from a modular decompiler as discussed in Section 5 above, optimality criterion (III) will in general be satisfied, but not criteria (I) nor (II) since the local control rule tends to over-specialize calls which results in re-evaluating expressions and emitting code multiple times.

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

Another important observation is that the costly mechanisms, namely the type-based homeomorphic embedding [4] and the polyvariance control from [18], used for controlling the PE that were used earlier to achieve the results in Sections 4.2 and 5.3 are not needed anymore using the optimal decompilation scheme. Instead, the following trivial control operators can be used: unfold unfolds all calls except those matching a **memo** or **rescall** annotation, and abstract adds to the set S_{i+1} every call in L^{pe} which is not an instance of any call in S_i. It can be easily proved that termination is ensured both at the local and at the global control level thanks to the annotations and the initial set of atoms provided to the PE in Definition 6. Intuitively, in the local control, the only source of potential non-termination is a loop in the bytecode program, and there is always a convergence point associated with it, therefore termination is guaranteed as the corresponding **memo** annotation associated with the divergence point will force unfolding to stop. In the global control, we have to ensure that the set of atoms to be specialized does not grow infinitely. The only atoms which can potentially occur in the set are those of the form $\texttt{execute}(\texttt{st}(\texttt{m}, \texttt{pc}, \_, \_), \_))$ with $pc \in \textsf{div\_points}(m) \cup \textsf{conv\_points}(m)$. Those with $pc \in \textsf{conv\_points}(m)$ are always an instance of an atom already present in the set thus they are never added. As regards those with $pc \in \textsf{div\_points}(m)$, it can be derived from Proposition 2 that only one single version of the atom can be added to the set, otherwise the corresponding bytecode will be traversed more than once. The complete proof of termination will require a complete formalisation of the control rules and a complete definition of the bytecode interpreter used, and is not given in this paper.

7. Decompiling Object-Oriented Bytecode

In this section we present the main extensions that are needed to apply interpretive decompilation to a bytecode language with object-oriented features. Such features include: dynamic memory allocation, arrays, classes and objects, inheritance and polymorphism. We first present an extension of $\mathcal{L}_{bc}$, denoted as $\mathcal{L}_{bc}^{O}$, which includes all these features in the spirit of Java bytecode. An $\mathcal{L}_{bc}^{O}$-bytecode program P_{bc} consists of a set of classes $classes(P_{bc}) = \mathcal{C}$, partially ordered w.r.t the subclass relation. The class is the basic (de-)compilation unit of $\mathcal{L}_{bc}^{O}$. Each class $c \in \mathcal{C}$ contains information about the class it extends[2] and the fields and methods it declares.

[2]If a class does not explicitly extend any class, it implicitly extends class `Object`.

A method (field) is uniquevocally identified by its method (field) signature which is of the form $c : mn$ ($c : fn$), where c is the class in which it is declared and mn (fn) the method (field) name. The name `init` is reserved for the class initialization methods (constructors). We write $defs(c)$ to denote the set of *internal* method signatures defined in the class c. Some other features of Java bytecode like interfaces, static methods and static fields, exceptions, access control and types besides integers and references are not yet considered to simplify the presentation. We show later in Section 8 that they do not add any complication to the decompilation process. As in $\mathcal{L}_{bc}$, the code associated to a method m, denoted $code(m)$, consists of a sequence of indexed bytecode instructions. The $\mathcal{L}_{bc}^{O}$ instruction set is:

$$\begin{array}{rl} Inst_{\mathcal{L}_{bc}^{O}} ::= & \mathtt{push(x)} \mid \mathtt{load(v)} \mid \mathtt{store(v)} \mid \mathtt{add} \mid \mathtt{sub} \mid \mathtt{mul} \mid \mathtt{div} \mid \mathtt{rem} \mid \mathtt{neg} \mid \\ & \mathtt{if} \diamond \mathtt{(pc)} \mid \mathtt{if0} \diamond \mathtt{(pc)} \mid \mathtt{goto(pc)} \mid \mathtt{return} \mid \mathtt{invoke(ms)} \\ & \mathtt{newarray(}\tau\mathtt{)} \mid \mathtt{arrload} \mid \mathtt{arrstore} \mid \mathtt{arraylength} \\ & \mathtt{new(c)} \mid \mathtt{getfield(fs)} \mid \mathtt{putfield(fs)} \mid \mathtt{dup} \mid \mathtt{ifnull} \mid \mathtt{ifnonnull} \end{array}$$

where τ is a type signature, $\tau \in \mathcal{C} \cup \{int\}$, `c` is a class, $\mathtt{c} \in \mathcal{C}$, `ms` a method signature and `fs` a field signature. The first two rows correspond to the instructions in $\mathcal{L}_{bc}$, which are already described in Section 4.1. The third row comprises the instructions to manipulate arrays: creation (`newarray(`τ`)`), loading and storing an element (resp. `arrload` and `arrstore`), and consulting the array length (`arraylength`). The last row contains instructions to manipulate objects: object creation (`new`), accessing and modifying fields (resp. `getfield` and `putfield`), the `dup` instruction duplicates the reference stored on top of the operand stack and new conditional branching instructions for references `ifnull` and `ifnonnull`. As we are omitting static methods, the `invoke` instruction always corresponds to virtual invocations. For simplicity, all methods are supposed to return a value (except for constructors).

7.1. *Handling the Heap during Decompilation*

An $\mathcal{L}_{bc}^{O}$-bytecode program manipulates both integers and references to objects and arrays[3]. Therefore, besides using an operand stack and an array of local variables, it uses a *heap* where objects and arrays are allocated. Thus, the first design decision which we have to take is how to represent the $\mathcal{L}_{bc}^{O}$ heap in Prolog. A first alternative would be to represent objects as Prolog terms. Each object could have as main functor an identifier for its class and as many arguments as fields there are in the corresponding class. The problem with this approach is that, though apparently simple, logic programs do not allow destructive updates, i.e., once an argument (variable) gets associated (unified) to a functor, it cannot be associated to a different functor, as the subsequent unification would fail. A possible way out would be the use of the non-pure Prolog predicate `setarg`, which allows overwriting values. However, the programs thus obtained are not very amenable to static analysis since the use of `setarg` breaks the declarative nature of logic programs and introduces all difficulties associated to the analysis of shared mutable data structures, which is well-known to pose major difficulties to static analysis. Since one of our main motivations is to analyze the programs obtained by our decompilation, we opt for another alternative which produces declarative programs. In this other alternative, the heap is passed as an explicit argument which is not overwritten, but rather modified as needed. We now describe how the $Int_{\mathcal{L}_{bc}}^{bs}$ interpreter is extended to handle the heap, denoted $Int_{\mathcal{L}_{bc}^{O}}$. The extensions include:

[3]We use the special functor `ref/1` to distinguish references in the Prolog representation.

- The main predicate of the interpreter is now of the form `main(M,InArgs,Hin,Top,Hout)`, where the new additional parameters `Hin` and `Hout` stand, respectively, for the input and the output heap of the method. Note that now `M` is not just a method name but a method signature.
- The state carried out by the interpreter has to include an extra argument for the heap. Thus, it is of the form `st(M,PC,OS,LV,H)`, where `H` is the current heap. Again, `M` is a method signature.
- The corresponding rules for the `step/3` associated with the new added bytecode instructions have to be provided. As an example, consider the implementation in our Prolog interpreter of the `getfield(f)` operation:

```
step(getfield(F),S,S') :-
    S = st(M,PC,[ref(R)|S],L,H),
    S' = st(M,PC',[V|S],L,H),
    next(M,PC,PC'),
    getfield(H,R,F,V).
```

There is an important difference between the heap and the operand stack which affects the decompilation process. While the operand stack is a local data structure of each method execution, the heap is a global entity which stores objects that can be created at any point during a program's execution and besides objects can be aliased. Basically, the consequence is that the heap becomes unknown at PE time (typically it is a logic variable) and, hence, most operations involving the heap cannot be fully evaluated and have to appear residual in the decompiled code. Essentially, we treat the heap during decompilation as an abstract data type with a set of operations which manipulate it. For instance, this is the case of the atom `getfield(H,R,F,V)` in the code above. In Figure 7 we list all the predicates used in the interpreter, which use the heap, together with a description of their functionality. Output arguments are underlined (the rest are input). Note that these are exactly the set of predicates that can appear residual in our decompiled programs besides arithmetic operations, calls to `main/5` (bytecode methods) and calls to `execute_i/n` (bytecode blocks). Figure 8 depicts to the right side an example of a decompiled program containing heap operations.

7.2. Decompilation with Classes

Object-oriented programs, both high-level and bytecode, are structured as a set of classes. It makes sense then to devise a decompilation scheme where the decompilation is done at the level of classes: we decompile one class at a time, and within each class, we decompile each declared method at a time. Clearly, it is convenient to structure decompiled programs at a similar level. A natural choice in a module-structured language like Prolog is to make use of modules such that each class is decompiled in a corresponding module. A Prolog module consists of a module name, a list of exported predicates, a list of imported modules (optionally together with the list of predicates imported from each module) and the code (set of clauses) of the exported and auxiliary predicates. We propose a decompilation scheme with the following characteristics:

1. There is a Prolog module per class in the bytecode program, with the same name.
2. A Prolog predicate is associated with each declared method, with the same name. As we will see later this is done via a simple post-renaming of the `main/5` atoms.

Creation operations:	
`new(H,C,R,H')`	`H'` is the heap obtained from the heap `H` by creating a new object of type `C`. The new object is stored at location `R`.
`newarray(H,T,N,R,H')`	`H'` is the heap obtained from the heap `H` by creating a new array with `N` elements of type `T`. The new array is stored in location `R`.
Accessing operations:	
`getfield(H,R,F,V)`	Field `F` of object at location `R` in the heap `H` has the value `V`.
`arrload(H,R,I,V)`	The element at index `I` of the array at location `R` in the heap `H` has the value `V`.
`arraylength(H,R,N)`	The length of the array at location `R` in the heap `H` is `N`.
Setting operations:	
`putfield(H,R,F,V,H')`	The heap `H'` results from setting the field `F` of object at location `R` in the heap `H` with the value `V`.
`arrstore(H,R,I,V,H')`	The heap `H'` results from setting the element at index `I` of the array at location `R` in the heap `H` with the value `V`.

Figure 7: Residual heap operations

3. Each Prolog module: 1) exports all predicates corresponding to the methods declared in the corresponding class and, 2) imports all needed external predicates from the corresponding modules.

The decompilation scheme with classes is formalized as follows:

Example 2. Definition 7 (CLASS-DECOMP$_{\mathcal{L}^O_{bc}}$**).** *Given a class of an $\mathcal{L}^O_{bc}$-bytecode program, an optimal, LP-decompilation of c is defined as:*

$$\text{CLASS-DECOMP}_{\mathcal{L}^O_{bc}}(c) = \phi(\bigcup_{\forall m \in \text{defs}(c)} PE(\text{Int}_{\mathcal{L}^O_{bc}} \cup \text{code}(m), \mathcal{A}_{class}(m), S(m)))$$

*where $\mathcal{A}_{class}(m) = \mathcal{A}_{opt}(m) \cup \mathcal{A}_{heap}$, being $\mathcal{A}_{opt}(m)$ and $S(m)$ the sets in Definition 6 adapted for the new interpreter $\text{Int}_{\mathcal{L}^O_{bc}}$. The set $\mathcal{A}_{heap}$ denotes the set of **rescall** annotations to residualize the heap operations in Figure 7.*

The function ϕ denotes a simple post-processing which is applied over the set of predicates resulting from the successive PEs, producing a Prolog module with the characteristics enumerated above. Basically, 1) it produces the corresponding module header, with the lists of imported and exported predicates, and 2) it renames all atoms of the form `main(c:mn,Args,Hin,Out,Hout)` as `c:Mn(Args,Hin,Out,Hout)`. This is interpreted in Prolog as a module-qualified call, i.e., a call to predicate `mn` of module `c`.

We can now define an object-oriented decompilation of an $\mathcal{L}^O_{bc}$-bytecode program as follows.

$$\text{OO-DECOMP}_{\mathcal{L}^O_{bc}}(P_{bc}) = \bigcup_{\forall c \in classes(P_{bc})} \text{CLASS-DECOMP}_{\mathcal{L}^O_{bc}}(c).$$

Observe that OO-DECOMP$_{\mathcal{L}^O_{bc}}$ takes a set of $\mathcal{L}^O_{bc}$ classes and produces a set of Prolog modules. An example of the application of OO-DECOMP$_{\mathcal{L}^O_{bc}}$ is shown in Figure 8. On the left side, we show the Java-like source code of our example program, together with the $\mathcal{L}^O_{bc}$-bytecode which is shown within brackets. Again, we show the source code for clarity but the decompilation works on the bytecode. It has three classes, `A`, `B` and `Foo`. `B` extends `A` inheriting field `n` and re-defining

```
class A {
public int n;
public A(){this.n=1;}
[ 0:load(0)
  1:invoke(Object:init)
  2:load(0)
  3:push(1)
  4:putfield(A:n)
  5:return ]
int m(){return n+1;}
[ 0:load(0)
  1:getfield(A:n)
  2:push(1)
  3:add
  4:return ]
}
```

```
:- module('A',[init/4,m/4]).

:- use_module('Object',[init/4]).

init([ref(T)],H0,_Out,H2) :-
   'Object':init([ref(T)],H0,_,H1),
   putfield(H1,T,'A':n,1,H2).

m(ref(T),H0,N',H0) :-
   getfield(H0,T,'A':n,N),
   N' is N+1.
```

```
class B extends A {
[ 0:load(0)
  1:invoke(A:init)
  2:return ]
int m(){return n+2;}
[ 0:load(0)
  1:getfield(A:n)
  2:push(1)
  3:add
  4:return ]
}
```

```
:- module('B',[init/4,m/4]).

:- use_module('A',[init/4]).

init([ref(T)],H0,_,H1) :-
   'A':init([ref(T)],H0,_,H1).

m(ref(T),H0,N',H0) :-
   getfield(H0,T,'A':n,N),
   N' is N+2.
```

```
class Foo {
int foo(A a){return a.m();}
[ 0:load(1)
  1:invoke(A:m)
  2:return ]
}
```

```
:- module('Foo',[foo/4]).
:- use_module('A',[m/4]).
:- use_module('B',[m/4]).

foo([_,Ref],H0,Out,H1) :-
   resolve(Ref,H0,C),
   C:m([Ref],H0,Out,H1).
```

Figure 8: Example of decompilation with classes

method `m`. Method `foo` of `Foo` invokes method `m` on an object declared of type `A`. The $\mathcal{L}_{bc}^{O}$-bytecode of each declared method is shown within brackets. On the right side, we show the Prolog decompiled program. It has three modules `A`, `B` and `Foo`. Note that, in Prolog, strings starting with an uppercase letter are interpreted as variables and the rest as functors or constants. Thus, if one wants to use the special constant *A*, the notation *'A'* has to be used. The `Foo` module will be explain in the next section. The corresponding predicates are exported/imported. See for example how module `A` exports predicates `init/4` and `m/4`, and imports predicate `init/4` from module `Object`[4]. Note that all predicates have four arguments as they come from the corresponding instance of `main/5`. There are several calls in the decompiled program which are module-qualified call, e.g., the call to `init/4` inside module `B`.

7.3. Virtual Invocations

An important feature of object-oriented languages is *polymorphism* in the presence of virtual invocations. In a virtual invocation, the method to be executed is determined at run-time depending on the actual type of the corresponding object. As it happens with heap operations, the operation to *resolve* the method to be called cannot be performed at PE time, and then has to be residualized. In fact, the information needed (object type) for the resolution is in the heap which is in general unknown at PE time as we saw in Section 7.1. In the following we show the code corresponding to the `invoke` operation for virtual invocations in our $Int_{\mathcal{L}_{bc}^{O}}$ interpreter:

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

Predicate `resolve/3` is encharged of performing the above method resolution. Given the call `resolve(Ref,H,C')` it proceeds as follows: 1) the class of the current object at location `Ref` in the heap `H` is obtained, and 2) due to inheritance, it can happen that the method is not declared in such class, then it has to go up in the classes hierarchy until reaching a class in which the method is declared. This class is finally returned in `C'`. Then, the call to `main/4` is done with the method signature `C':M'`. As with heap operations, the corresponding **rescall** annotation has to be provided to make the corresponding `resolve/3` atom appear in the decompiled code. It always appears immediately before calls corresponding to method invocations (except calls to constructors[5]).

Example 3. As an example, consider method `foo` of class `Foo` in Fig. 8. The method `m` is invoked on an object declared of type `A`. However, variable `a` can actually store at run-time a reference to an object of class `A` or of any class extending `A`, in this case `B`. Whether to execute method `m` of class `A` or of class `B` is thus determined at run-time. In the Prolog code, we can observe the call to `resolve/3` immediately before the call to predicate `m/4`, which is module-qualified with the obtained module.

8. Experimental Evaluation

We report on two different implementations of a decompiler for full (sequential) Java Bytecode into Prolog. For the first one we extend an already existing powerful online PE, the one integrated in the `CiaoPP` analysis and specialization system [21]. This partial evaluator implements several unfolding rules and abstraction operators. This allows us to compare the different decompilation schemes explained in the paper, in particular, to compare to the non-optimal ones. Such comparison is presented in Section 8.1. However, the overhead introduced by using such

[4]Constructor methods first call the constructor of its super-class, in this case `Object`.
[5]Invocations to constructors are never virtual. They can be statically resolved.

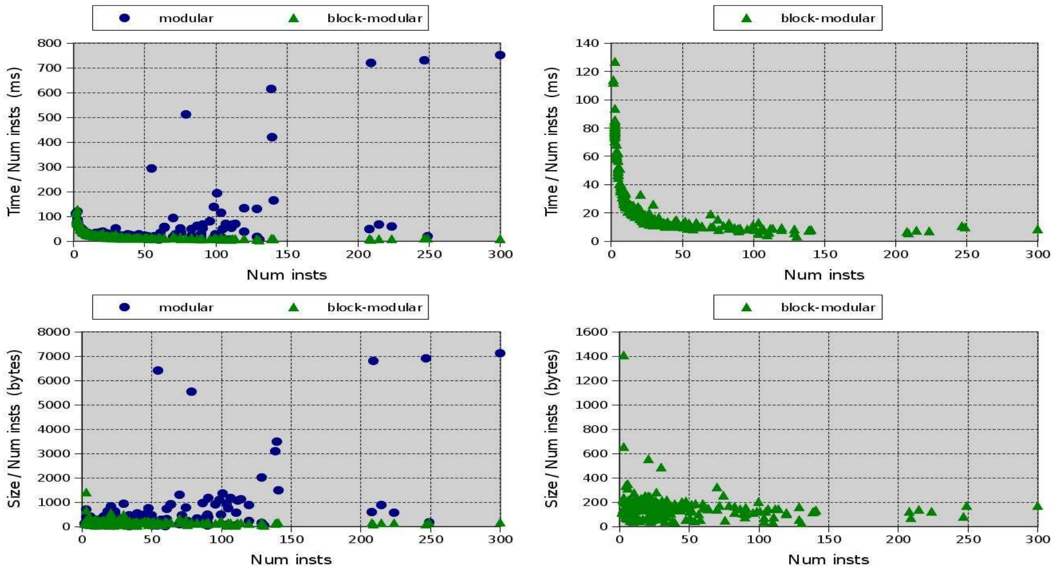


Figure 9: Evaluating the scalability of *optimal* decompilation with the JOlden Suite

generic and powerful tool prevents us from competing with ad-hoc decompilers as regards efficiency (decompilation times). For this reason, we have carried out a second implementation for which we have written a stand-alone PE which only contains the local and global strategies required by an optimal decompilation. This partial evaluator is integrated into a decompilation tool called jbc2prolog which also includes a Java bytecode interpreter. This makes it possible to both obtain optimal decompilation and be competitive in terms of efficiency with ad-hoc decompilers. A thorough comparison against the decompiler in the COSTA [3] system and against the JDec [8] decompiler is presented in Section 8.2.

Both implementations consider full sequential Java bytecode. The extensions needed to handle the features not considered in $\mathcal{L}_{bc}^{O}$ (exceptions, static fields and methods, access control, etc.) do not add any special complication to the decompilation scheme. For instance, exception handling is simply dealt with as another source of branching. This certainly makes the size of our decompiled programs grow considerably, although this is something every decompiler of a real-life language has to deal with. Solutions based on static analysis exist which allow avoiding some exception branches. E.g., *nullity* analysis can be used to avoid considering branches corresponding to null-pointer exceptions which are proved to be non-null, which reduces the size of the code considerably. These analyses can be easily incorporated in our decompilation tool and it is indeed a subject of future work.

8.1. Assessing the Scalability of Decompilation

For the experimental evaluation, we have used the standardized set of benchmarks in the JOlden suite [22]. In particular, our first goal is to compare the scalability of the *optimal* decompilation scheme (see Definition 7) against that of the *modular* (non-optimal) one (see Definition 5). Here it comes the need to use the partial evaluator of `CiaoPP`, as it combines the power of online control operators like type-based homeomorphic embedding [4], with the ability of adding conditional annotations as described in Definition 6. As most programs in the JOlden suite make an extensive use of library methods, non-modular decompilation cannot be assessed as we run into memory problems when trying to decompile the code of library calls. Figure 9 depicts four charts measuring different aspects of the decompilation. In order to reason about scalability, we assess the differences between the non-optimal and the optimal approaches, as well as how the size of the programs affects the decompilation. The times are computed as the arithmetic mean of five runs on an Intel Core 2 Duo at 1.86GHz with 2GB of RAM, running Linux 2.6.24-21. We measure two aspects of the decompilation: the decompilation time (in milliseconds) and the decompiled program size (in bytes). It should be noticed that absolute data are not required to assess scalability issues. We rather need relative data per instruction in order to prove that it does not increase with the size of the programs. The relative decompilation time indicates the efficiency of the process and the size of decompiled programs is directly related to the decompilation quality. Each point $[X, Y]$ in the charts corresponds to the decompilation of a single method in the JOlden suite, where X represents the number of instructions of the method and Y the measured data (time per instruction or decompiled program size per instruction). The charts in the left-hand side show the data obtained (times in the top chart and sizes in the bottom one) for both the non-optimal and the optimal decompilation. The variations in the optimal decompilation cannot be appreciated when combined with the non-optimal. Thus, we include in the charts on the right-hand side the figures for the optimal decompilation in isolation such that we adjust the scale on the Y-axis to the domain of the data.

From the charts, we conclude: (1) Times per instruction are notably larger for the smallest methods, as can be seen by looking at the initial curve in the charts. This is because the overhead introduced for starting a new decompilation is more noticeable when the time for decompilation itself is small, while it becomes negligible for larger methods. The same happens for the size of the decompiled programs. (2) The optimal decompilation achieves important speedups in general (for all methods with more than 40 instructions). Besides, it obtains significantly smaller decompiled programs. The speedups per package range from 3.36 in **power** to 31.4 in **bisort** for the decompilation times; and from 2.5 times smaller in **power** to 9 times smaller in **bisort** for the decompiled program sizes. Note that there is a clear correspondence between both measures, since C points introduce both inefficiency and size increase in decompilation, as explained in Section 6. Moreover, modular decompilation runs out of memory for some of the largest methods. This is again related to code duplication (C and D points) and (re-)evaluation (C points), which grow exponentially. (3) The most important conclusion is that, while in the non-optimal decompilation both the times and the sizes per instruction greatly increase with the size of the benchmarks, this does not happen in the optimal scheme. In the optimal decompilation, these figures are totally stable (mostly constant) for all methods with more than 40 instructions. This shows that both the decompilation times and the decompiled program sizes are *linear* with the size of the input bytecode program, thus demonstrating the scalability of our optimal decompilation. One might wonder why there are still small variations in the ratio. In our experience, the following points also matter: 1) the complexity of the control flow of the methods, 2) the relative complexity of the bytecode instructions used, e.g., instructions which operate in the heap tend to produce more residual code, 3) the structure w.r.t. methods of the program, e.g., classes with methods of medium size tend to result in better decompilations than those with few large methods or many small ones.

Benchmark				**jbc2prolog**						COSTA	JDec
Pack	$\mathbf{N}_{cls}$	$\mathbf{N}_{mths}$	$\mathbf{N}_{ins}$	$\mathbf{T}_{bl}$	$\mathbf{T}_{sps}$	$\mathbf{T}_{ge}$	$\mathbf{T}_{pe}$	$\mathbf{T}_{cg}$	$\mathbf{T}_{j2p}$	$\mathbf{T}_{costa}$	$\mathbf{T}_{jdec}$
bisort	2	15	554	10	10	0	147	10	177	170	1802
bh	9	73	2012	57	28	0	652	70	807	860	7394
em3d	4	22	713	27	7	0	184	26	243	347	3386
health	6	27	973	37	13	0	224	26	300	420	4822
mst	5	31	703	14	4	0	173	20	210	317	3958
perimeter	9	46	838	37	9	0	134	13	193	363	6564
power	6	32	1927	43	24	4	566	64	701	693	5330
treeadd	2	12	308	6	3	0	67	14	90	143	1600
tsp	2	16	946	17	13	4	367	26	427	380	1948
voronoi	6	73	1781	50	19	7	673	62	810	1023	5270
overall	51	347	10755	297	131	14	3186	330	3958	4717	42074

Table 1: Efficiency of jbc2prolog

8.2. Efficiency: Comparing against other Decompilers

To assess the efficiency of our approach we compare the decompilation times we get using our tool jbc2prolog w.r.t. those obtained using the decompiler in the COSTA system [3] and those of the well-known Java decompiler JDec [8]. COSTA is a COSt and Termination analysis tool for Java bytecode. It performs a decompilation of the bytecode into a rule-based representation before the actual analysis phase with the aim of making the analysis design simpler. This decompilation basically consists of two parts. First, the CFG for each method is built and then, for each block in the CFG, an associated rule is produced. We have chosen the COSTA decompiler to compare the efficiency of interpretive decompilation because COSTA is also implemented in Prolog and hence the underlying implementation language performance is identical. The resulting decompiled program is a set of rules which resemble our Prolog clauses in several aspects: recursion is the only form of iteration and conditional instructions are captured by guarded rules. However, there are still some differences w.r.t. our decompiled programs: a) in COSTA the operand stack is explicitly flattened and represented by means of local variables whereas in jbc2prolog PE together with argument filtering automatically achieve this effect, and b) we represent the heap explicitly in the residual programs as explained in Section 7.1. These two features together are important since in the programs decompiled using COSTA (or `CiaoPP`) all bytecode instructions remain residual and have to be taken as builtins, i.e., predefined procedures by analysis. In contrast, in jbc2prolog bytecode instructions are interpreted at decompilation time and converted into basic Prolog instructions such as unifications and arithmetic or into the ADT operations in Figure 7 for those instructions involving the heap. As a result, extending an existing Prolog analyzer to analyze JBC decompiled programs is simpler using our decompiler than using those in COSTA [2] or `CiaoPP` [35], since the decompiled programs are executable and the analysis does not need to be extended with any further builtins.

Again we use the set of benchmarks in the JOlden suite [22]. Table 1 shows the times taken (in milliseconds) by each of the different phases of jbc2prolog together with the total time used by the COSTA and JDec decompiler for each package of the JOlden suite. All times are computed as the arithmetic mean of five runs, in this case on a Intel Core 2 Quad Q9300 at 2.5GHz with 1.95GB of RAM, running Linux 2.6.27-9. In particular, for each JOlden package we measure:

the total number of classes, methods and instructions in the package (columns $\mathbf{N}_{cls}$, $\mathbf{N}_{mths}$ and $\mathbf{N}_{ins}$), the time taken by the different phases of jbc2prolog, namely, the parsing and loading time of the `.class` file (column $\mathbf{T}_{bl}$), the pre-processing time to infer the divergence and convergence points of the bytecode program (column $\mathbf{T}_{sps}$), the generation of the entries to the PE (column $\mathbf{T}_{ge}$), the actual specialization time (column $\mathbf{T}_{pe}$) and the time taken by the code generation phase (column $\mathbf{T}_{cg}$). Finally, last three columns show respectively the total times taken by jbc2prolog (column $\mathbf{T}_{j2p}$), the COSTA decompiler (column $\mathbf{T}_{costa}$) and the JDec decompiler (column $\mathbf{T}_{jdec}$). The last row shows the overalls of all measurements. We can see that the whole JOlden suite is decompiled by jbc2prolog in less than 4 seconds versus the 4.7 secs in COSTA and the 42 secs in JDec. It can be concluded that our results our competitive with those of an ad-hoc decompiler. In particular, we see that they are similar to those obtained in COSTA. Furthermore, in most examples, jbc2prolog is more efficient than COSTA, especially in **voronoi**, **perimeter** and **treeAdd**. On the other hand we can see that jbc2prolog is about ten times faster than JDec. Our conclusion in this regard is that it is very difficult to compare with decompilers written in other programming languages, since the performance of the implementation language heavily influences the decompilation time.

9. Related Work

Previous work in *interpretative* (de-)compilation has mainly focused on proving that the approach is feasible for small interpreters and medium-sized programs. The focus has been on demonstrating its *effectiveness*, i.e., that the so-called interpretation layer can be removed from the compiled programs. To achieve effectiveness, offline [29], online [5, 20, 39] and hybrid [30] PE techniques have been assessed and novel control strategies have been proposed and proven effective [18, 4]. Our work starts off from the premise that interpretive decompilation is feasible and effective as proved by previous work and studies further issues which have not been explored yet. Let us review now related work both in the field of decompilation of low-level code. Related work on on the PE of interpreters has been already compared in Section 1 and in several places throughout the paper.

The work by Breuer and Bowen [9] is only tangentially related to ours. They propose a general method for compiling decompilers from the specifications of (non-optimizing) compilers. The main idea is that a data type specification for a programming-language grammar can be remolded into a functional program that enumerates all of the abstract syntax trees of the grammar. It is showed that by relying on this technique a decompiler can be generated from a simple Occam-like compiler specification. The only similarity with our work is that decompiled programs are somehow obtained from specifications (in our case of the interpreter and in their case of the compiler). However, the underlying methods are technically different and also they do not provide a practical solution for ensuring applicable conditions for their technique.

As regards (direct) decompilation of low-level back to source code, it has been the subject of a good amount of research. Decompilation can be attempted at different levels, with different levels of success. The most complicated case is when decompiling binary executables. There are a good number of associated complications, such as recovering the control flow. One intrinsic problem in this approach is that it is not possible in general to distinguish code from data statically. See e.g. [10, 42] and their references for a discussion on the problems and techniques for binary decompilation. The next level is decompilation of assembly, see e.g. [11]. This shares many of the complications associated to the decompilation of binaries, since current hardware architectures are rather complex, but at least it is possible to separate code from data. The following level is decompilation of code to be run on a virtual machine. This is in general easier to perform since virtual machines are usually simpler than current hardware architectures and because often the code for this virtual machines (bytecode) must satisfy certain behavior restrictions (must be *verifiable* [27]) and types of variables are available. As a result, in the particular case of decompilation of Java bytecode back to Java source, a number of successful commercial and free software decompilers exist which are able to handle a large class of bytecode programs, especially those generated by common Java compilers, i.e., `javac`. Nevertheless, things become more complicated when the Java bytecode has been generated by an obfuscator, and especially when an optimizing compiler, or a compiler from other programming languages such as Haskell, Eiffel, ML, Ada, and Fortran is used. See e.g. [37] and its references for a good account on the existing Java bytecode decompilers and the difficulties associated to its decompilation.

As already mentioned, there exist several analyzers for Java bytecode which use a higher-level intermediate representation and which can be seen as ad-hoc decompilers. In particular, both the COSTA [3] and `CiaoPP` [21] systems have a front-end which converts bytecode into an intermediate representation which is then the input to the subsequent analysis. Though in both cases the intermediate representation is similar, in the case of COSTA it is formalized as a rule-based representation [2], whereas in `CiaoPP` it is formalized as Horn clauses, i.e., a logic program [35]. The reason for doing that in `CiaoPP` is that, at least in principle, that allows using the analysis which are already available in `CiaoPP`. However, there is a crucial difference between the logic programs generated in [35] and those generated by our decompiler. Whereas the programs generated by [35] are only meant to be the subject of static analysis and are not executable, the programs we generate can both be subject to analysis or be executed. The reason why the programs in [35] nor those in [2] are executable is because they basically capture the control-flow of the bytecode program, but the basic bytecode instructions themselves remain as *builtins*, i.e., predefined predicates, to the analysis. Analysis results are correct as long as the behavior of such bytecode instructions is safely approximated by the analysis. Producing fully executable logic programs as the result of decompilation is not trivial since many of the bytecode instructions operate on the heap in a way or another. Thus, in order to make an executable decompiled program we need to introduce the JVM heap explicitly in the logic program. All this is done automatically in our approach.

10. Conclusions

We argue that *declarative languages* and the technique of *partial evaluation* have nowadays a large application field within the development of analysis, verification, and model checking tools for modern programming languages. On one hand, declarative languages provide a convenient intermediate representation which allows (1) representing all iterative constructs (loops) as recursion, independently of whether they originate from iterative loops (conditional and unconditional jumps) or recursive calls, and (2) all variables in the local scope of the methods (formal parameters, local variables, fields, and stack values in low-level languages) can be represented uniformly as explicit arguments of a declarative program. On the other hand, the technique of PE enables the automatic (de-)compilation of a (complicated) modern program to a simple declarative representation by just writing an interpreter for the modern language in the corresponding declarative language and using an existing partial evaluator.

The resulting intermediate representation greatly simplifies the development of the above tools for modern languages and, more interestingly, existing advanced tools developed for declarative programs (already proven correct and effective) can be directly applied on it. In previous work [5], by reasoning on our decompiled residual programs, we have automatically proved in the CiaoPP system some non-trivial properties of Java bytecode programs such as termination, run-time error freeness and infer bounds on its resource consumption. In order to prove run-time error freeness, we have proposed an enhanced bytecode interpreter which computes, in addition to the return value of the method called, also the trace which captures the computation history. Such traces represent the semantic steps used and therefore do not only represent instructions, as the context has also some importance. They have allowed us to distinguish, for example, for a same instruction, the step that throws an exception from the normal behavior. E.g., `invokevirtual_step_ok` and `invokevirtual_step_NullPointerException` represent, respectively, a normal method call and a method call on a null reference that throws an exception. Such additional flexibility of interpretive decompilation has allowed to prove run-time error freeness in a straightforward way by simply specifying the property of being error-free as verifying that the corresponding trace in the decompiled program does not contain an exceptional step.

A unique feature of our decompiled programs is that they represent the whole program state, i.e., in contrast to [35, 2, 43], our decompiled programs contain a representation of the heap in addition to the operand stack. The advantage is decompiled programs are fully *executable* which in turn broadens their application field. As an example, recently we have developed a novel framework for *test case generation* [45] of bytecode by relying on our decompiled Prolog programs. Basically, the standard approach to generating test-cases statically is to perform a *symbolic* execution of the program [12, 36, 38, 24, 19], where the contents of variables are expressions rather than concrete values. The symbolic execution produces a system of *constraints* consisting of the conditions to execute the different paths. This happens, for instance, in branching instructions, like if-then-else, where we might have to generate test-cases for the two alternative branches and hence accumulate the conditions for each path as constraints. The fact that our decompiled program are executable Prolog programs allows us to directly rely on available techniques for *constraint* logic programs (where backtracking is inherent to the language) to carry out such symbolic execution.

Finally, a main objective of our work has been to investigate, and provide the necessary techniques, to make interpretive decompilation scale in practice. A further goal has been to ensure, and provide the techniques, that decompiled programs preserve the structure of the original programs and that their quality is comparable to that obtained by dedicated decompilers. We believe that the techniques proposed in this paper, together with their experimental evaluation, provide for the first time actual evidence that the interpretive theory proposed by Futamura in the 70s is indeed an appealing and feasible alternative to the development of ad-hoc decompilers from modern languages to intermediate representations.

Acknowledgments

We gratefully acknowledge the anonymous referees for many useful comments and suggestions. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* and TIN-2008-05624 *DOVES* projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

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

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

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

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

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

[6] 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.

[7] B. Barras et al. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, 1997. citeseer.ist.psu.edu/barras97coq.html.

[8] Swaroop Belur and Kartik Bettadapura. Jdec: Java Decompiler. http://jdec.sourceforge.net/.

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

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

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

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

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

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

[15] J. Gallagher. Transforming logic programs by specializing interpreters. In *Proc. of the 7th. European Conference on Artificial Intelligence*, 1986.

[16] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98. ACM Press, 1993.

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

[18] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Improving the Decompilation of Java Bytecode to Prolog by Partial Evaluation. In *ETAPS Ws on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, volume 190 of *ENTCS*, pages 85–101, 2007.

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

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

[21] 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.

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

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

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

[25] J. Komorovski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta Programming in Logic, Proceedings of META'92*, volume 649 of *LNCS*, pages 49–69. Springer-Verlag, 1992.

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

[27] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

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

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

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

[31] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in prolog using a hand-written compiler generator. *TPLP*, 4(1–2):139 – 191, 2004.

[32] Michael Leuschel and Morten Heine Sørensen. Redundant argument filtering of logic programs. In *LOPSTR*, pages 83–103, 1996.

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

[34] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[35] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.

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

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

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

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

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

[41] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *Proc. of LOPSTR'04*, pages 149–165. Springer LNCS 3573, 2005.

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

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

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

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