Modular Decompilation of Low-Level Code by Partial Evaluation

Miguel Gómez-Zamalloa
DSIC, Complutense University of Madrid

Elvira Albert
DSIC, Complutense University of Madrid

Germán Puebla
CLIP, Technical University of Madrid

Abstract

Decompiling low-level code to a high-level intermediate representation facilitates the development of analyzers, model checkers, etc. which reason about properties of the low-level code (e.g., bytecode, .NET). Interpretive decompilation consists in partially evaluating an interpreter for the low-level language (written in the high-level language) w.r.t. the code to be decompiled. There have been proofs-of-concept that interpretive decompilation is feasible, but there remain important open issues when it comes to decompile a real language: does the approach scale up? is the quality of decompiled programs comparable to that obtained by ad-hoc decompilers? do decompiled programs preserve the structure of the original programs? This paper addresses these issues by presenting, to the best of our knowledge, the first modular scheme to enable interpretive decompilation of low-level code to a high-level representation, namely, we decompile bytecode into Prolog. We introduce two notions of optimality. The first one requires that each method/block is decompiled just once. The second one requires that each program point is traversed at most once during decompilation. We demonstrate the impact of our modular approach and optimality issues on a series of realistic benchmarks. Decompilation times and decompiled program sizes are linear with the size of the input bytecode program. This demostrates empirically the scalability of modular decompilation of low-level code by partial evaluation.

1 Introduction

Decompilation of low-level code (e.g., bytecode) to an intermediate representation has become a usual practice nowadays within the development of analyzers, verifiers, model checkers, etc. For instance, in the context of *mobile* code, as the source code is not available, decompilation facilitates the reuse of existing analysis and model checking tools. In general, high-level intermediate representations allow abstracting away the particular language features and developing the tools on simpler representations. As a representative example, Java bytecode is decompiled to a rule-based representation in [2], to clause-based programs in [23], to a three-address code view of bytecodes in Soot [26] and to the typed procedural language BoogiePL in [7]. Also, PIC programs are transformed to logic programs in [13]. Rule-based representations used in declarative programming in general—and in Prolog in particular—provide a convenient formalism to define such intermediate representations. E.g., as it can be seen in [2, 23, 26, 13], the operand stack used in a low-level language can be represented by means of explicit logic variables and that its unstructured control flow can be transformed into recursion. As further examples of the use of declarative programming, Java programs are transformed prior to be analyzed to term rewriting systems in [11] and C++ programs to logic programs in [22].

All above cited approaches (except [13]) develop *ad-hoc* decompilers to carry out the particular decompilations. An appealing alternative to the development of dedicated decompilers is the so-called *interpretive* decompilation by *partial evaluation* (PE) [14]. PE is an automatic program transformation technique which specializes a program w.r.t. part of its known input data. Interpretive compilation was proposed in Futamura's seminal work [8], whereby compilation of a program P written in a (*source*) programming language L_S into another (*object*) programming language L_O is achieved by specializing an interpreter for L_S written in L_O w.r.t. P. The advantages of interpretive (de-)compilation w.r.t. dedicated (de-)compilers are well-known and discussed in the PE literature (see, e.g., [4]). Very briefly, they include: *flexibility*, it is easier to modify the interpreter in order to tune the decompilation (e.g., observe new properties of interest); *easier to trust*, it is more difficult to prove that ad-hoc decompilers preserve the program semantics; *easier to maintain*, new changes in the language semantics can be easily reflected in the interpreter.

There have been several proofs-of-concept of interpretative (de-)compilation (e.g., [4, 13, 17]), 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 (de-)compiled programs preserve the structure of the original ones? (c) is the "quality" of decompiled programs comparable to that obtained by dedicated decompilers?* This paper answers these questions positively by proposing a modular decompilation scheme which can be steered to control the structure of decompiled code and ensure quality decompilations which preserve the original program's structure. Our main contributions are summarized as follows:

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

For the sake of concreteness, our decompilation scheme is formalized in the context of logic programming but the techniques to enable modularity can also be applied to compilation for any instantiation of languages (not necessarily low-level languages).

2 Basics of Partial Deduction

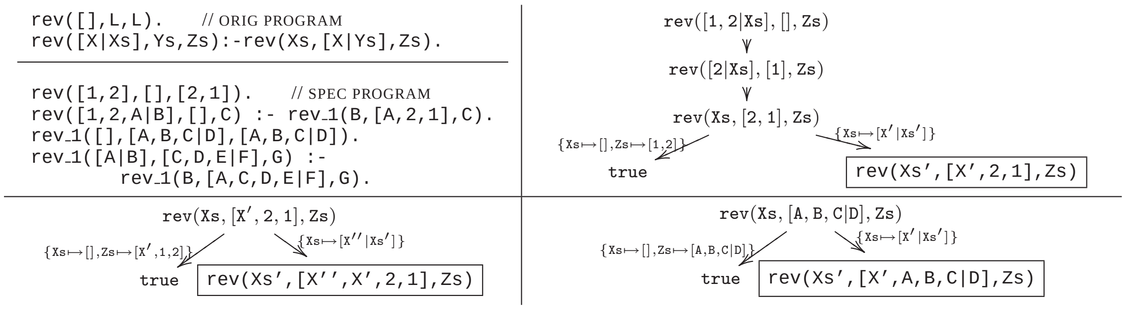
We assume familiarity with basic notions of logic programming [20]. Executing a program P for a call A consists in building an *SLD tree* for $P \cup \{A\}$ and then extracting the *computed answers* from every non-failing branch of the tree. PE in logic programming (see e.g. [9]) builds upon the SLD trees mentioned above. We now introduce a generic function PE, which is parametric w.r.t. the *unfolding rule*, unfold, and the *abstraction operator*, abstract and captures the essence of most algorithms for PE of logic programs:

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

Function PE differs from standard ones in the use of the set of annotations $\mathcal{A}$, whose role is described below. PE starts from a program P, a (possibly empty) set of annotations $\mathcal{A}$ and an initial set of calls S_0. At each iteration, the so-called *local control* is performed by the unfolding rule unfold (Line 3), which takes the current set of terms S_i, the program and the annotations and constructs a *partial* SLD tree for each call in S_i. Trees are partial in the sense that, in order to guarantee termination of the unfolding process, it must be possible to choose *not* to further unfold a goal, and rather allow leaves in the tree with a non-empty, possibly non-failing, goal (these goals appear in the figure within a frame). Let us consider the PE of program `rev` (first two rules at the top left of Fig. 1) w.r.t. the initial set $S = \{\mathtt{rev}([1,2|\mathtt{Xs}],[],\mathtt{Zs})\}$ and $\mathcal{A} = \emptyset$. We show in the figure the three partial SLD-trees computed by unfold during the PE process. The particular unfold operator determines which call to select from each goal and when to stop unfolding. For the SLD-trees shown in the figure, the unfolding rule stops the derivation when the selected call *embeds* [16] a previous call, i.e., it is syntactically larger than a previous call and thus threatens termination. In the top right tree, the call in the frame $\mathtt{rev}(\mathtt{Xs}',[\mathtt{X}',2,1],\mathtt{Zs})$ embeds the previous call $\mathtt{rev}(\mathtt{Xs},[2,1],\mathtt{Zs})$, hence the derivation is stopped.

The partial evaluator may have to build several SLD-trees to ensure that all calls left in the leaves (named *leaves*(T^{pe}) in L4) are "covered" by the root of some tree. This is known as the *closedness* condition of PE [21]. E.g., after having built the first SLD-tree for the call $\mathtt{rev}([1,2|\mathtt{Xs}],[],\mathtt{Zs})$, the call $\mathtt{rev}(\mathtt{Xs}',[\mathtt{X}',2,1],\mathtt{Zs})$ is not covered by $\mathtt{rev}([1,2|\mathtt{Xs}],[],\mathtt{Zs})$ because it is not an instance of it. In the *global control*, those calls in the leaves which are not covered are added to the new set of terms to be partially evaluated, by the operator abstract (L4). At the next iteration, an SLD-tree is built for such call, shown at the bottom left tree. Thus, basically, the algorithm iteratively (L2-6) constructs partial SLD trees until all their leaves are covered by the root nodes. An essential point of the operator abstract is that it has to perform "generalizations" on the calls that have to be partially evaluated in order to avoid computing partial SLD trees for an infinite number of calls. E.g., the framed calls $\mathtt{rev}(\mathtt{Xs},[\mathtt{X}',\mathtt{X}'',2,1],\mathtt{Zs})$ and $\mathtt{rev}(\mathtt{Xs},[\mathtt{X},2,1],\mathtt{Zs})$ are generalized, resulting in $\mathtt{rev}(\mathtt{Xs},[\mathtt{A},\mathtt{B},\mathtt{C}|\mathtt{D}],\mathtt{Zs})$. Usually, the generalized call is added to the set S_{i+1} and the instances (i.e., $\mathtt{rev}(\mathtt{Xs},[\mathtt{X},2,1],\mathtt{Zs})$) removed. At the next iteration, an SLD tree is built for the generalized term (bottom right). Without such generalization, the algorithm would keep on adding calls $\mathtt{rev}(\mathtt{Xs},[\mathtt{X},\mathtt{X}',\mathtt{X}'',2,1],\mathtt{Zs})$, $\mathtt{rev}(\mathtt{Xs},[\mathtt{X},\mathtt{X}',\mathtt{X}'',\mathtt{X}''',2,1],\mathtt{Zs})$,... infinitely.

A partial evaluation of P w.r.t. S is then systematically extracted from the resulting set of calls T^{pe} in the final phase, codegen in L7. The notion of *resultant* is used to generate a program rule associated to each root-to-leaf derivation of the SLD-trees for the final set of terms T^{pe}. Given an SLD derivation of $P \cup \{A\}$ with $A \in T^{pe}$ ending in B and θ be the composition of the mgu's in the



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

```
rev([1,2],[],[2,1]).    // SPEC PROGRAM
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).
```

Figure 1. Partial Evaluation (and unfolding SLD trees) for $\texttt{rev}([1,2|\texttt{Xs}],[],\texttt{Zs})$**.**

derivation steps, the rule $\theta(A) :- B$ is called the *resultant* of the derivation. A PE is defined as the set of resultants associated to the derivations of the constructed partial SLD trees for all $P \cup T^{pe}$. The final set T^{pe} contains the calls $\texttt{rev}([1,2|\texttt{Xs}],[],\texttt{Zs})$ and $\texttt{rev}(\texttt{Xs},[\texttt{A},\texttt{B},\texttt{C}|\texttt{D}],\texttt{Zs})$ from which the PE, at the top left, is generated. The first two resultants are obtained from each derivation (branch) of the top right tree, the last two ones from the bottom right tree.

The notions of *completeness* and *correctness* of PE [9] ensure that the specialized program produces no less resp. no more answers than the original program. A sufficient condition to ensure completeness is that the specialized program is *closed* by the resulting set of terms T^{pe}. Intuitively, the closedness condition ensures that all calls which may arise during the computation of $P \cup S$ are instances of T^{pe} and hence there is a matching resultant for them (solutions are not lost). The abstraction operator is encharged of ensuring that the closedness condition is met by means of a proper generalization of calls. Correctness is achieved when the resulting set T^{pe} is independent, i.e., there are no two calls in T^{pe} which unify. Independence can be recovered by a post-processing of renaming, which often does argument filtering [9]. In addition, renaming has benefits for performance because it reduces the number of rules per predicate. Thus, though the calls in T^{pe} for our example are independent, we rename the second call for predicate `rev` to `rev_1`.

Finally, the role of the annotations $\mathcal{A}$ (often manually provided) in *offline* PE is to give information to the control operators to decide when to stop derivations in the local control and how to perform generalizations in the global control to ensure termination. In *online* PE, all control decisions are taken during the specialization phase, without the use of annotations. We trivially turn function PE into online by just ignoring the annotations. In our method, though they are not needed to ensure termination, we use annotations to improve the quality of decompilation . Hence, according to the above classification, we will adopt in this work a (hybrid) online PE algorithm enhanced with some offline annotations (automatically computed).

3 Non-Modular Interpretive Decompilation

This section describes the state of the art in interpretive decompilation of low-level languages to Prolog, including recent work in [13, 3, 12, 4]. We do so by formulating non-modular decompilation in a generic way and identifying its limitations. The low-level language we consider, denoted as $\mathcal{L}_{bc}$, is a simple imperative bytecode language in the spirit of Java bytecode but, to simplify the presentation, without object-oriented features (our implementation supports full Java bytecode). It uses an operand stack to perform computations. It has an unstructured control flow with explicit conditional and unconditional `goto` instructions and manipulates only integer numbers. A bytecode program P_{bc} is organized in a set of methods which are the basic (de-)compilation units of $\mathcal{L}_{bc}$. The code of a method m, denoted *code*(m), consists of a sequence of bytecode instructions $BC_m = < pc_0 : bc_0, \ldots, pc_{n_m} : bc_{n_m} >$ with $pc_0, \ldots, pc_{n_m}$ being consecutive natural numbers. The $\mathcal{L}_{bc}$ instruction set is:

BcInst ::= `push(x)` | `load(v)` | `store(v)` | `add` | `sub` | `mul` | `div` | `rem` | | `neg` | `if` $\diamond$ `(pc)` | `if0` $\diamond$ `(pc)` | `goto(pc)` | `return` | `call(mn)`

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

3.1 Non-modular, Online, Interpretive Decomp.

We rely on the so-called "interpretive approach" to compilation by PE described in Sect. 1, also known as first Futamura projection [8]. In particular, the decompilation of a $\mathcal{L}_{bc}$-bytecode program P_{bc} to LP (for short LP-decompilation) might be obtained by specializing (with an LP partial evaluator) a $\mathcal{L}_{bc}$-interpreter written in LP w.r.t.

```
main(Method,InArgs,Top) :-
   build_s0(Method,InArgs,S0), execute(S0,Sf),
   Sf = st(fr(_,_,[Top|_],_),_)).
execute(S,S) :-
   S = st(fr(M,PC,[_Top|_],_),[]),
   bytecode(M,PC,return,_).
execute(S1,Sf) :-
   S1 = st(fr(M,PC,_,_),_), bytecode(M,PC,Inst,_),
   step(Inst,S1,S2), execute(S2,Sf).

step(goto(PC),S1,S2) :-
   S1 = st(fr(M,_,S,LV),FrS),
   S2 = st(fr(M,PC,S,LV),FrS).
step(push(X),S1,S2) :-
   S1 = st(fr(M,PC,S,L),FrS), next(M,PC,PC2),
   S2 = st(fr(M,PC2,[X|S],L),FrS).
...
step(call(M2),S1,S2) :-
   S1 = st(fr(M,PC,OS,LV),FrS), split_OS(M2,OS,Args,OS3),
   build_s0(M2,Args,st(fr(M2,PC2,OS2,LV2),_)),
   S2 = st(fr(M2,PC2,OS2,LV2),[fr(M,PC,OS3,LV)|FrS]).
step(return,S1,S2) :-
   S1 = st(fr(_,_,[RV|_],_),[fr(M,PC,OS,LV)|FrS]),
   next(M,PC,PC2), S2 = st(fr(M,PC2,[RV|OS],LV),FrS).
```

Figure 2. Fragment of (small-step) $\mathcal{L}_{bc}$ interpreter

P_{bc}. In Fig. 2 we show a fragment of a (small-step) $\mathcal{L}_{bc}$ interpreter implemented in Prolog, named *Int*$_{\mathcal{L}_{bc}}$. We assume that the code for every method in the bytecode program P_{bc} is represented as a set of facts `bytecode/3` such that, for every pair $pc_i : bc_i$ in the code for method m, we have a fact `bytecode(`m`,`pc_i`,`bc_i`)`. The state carried around by the interpreter is of the form `st(Fr,FrStack)` where `Fr` represents the current frame (environment) and `FrStack` the stack of frames (call stack) implemented as a list. Frames are of the form `fr(M,PC,OStack,LocalV)`, where `M` represents the current method, `PC` the program counter, `OStack` the operand stack and `LocalV` the list of local variables. Predicate `main/3`, given the method to be interpreted `Method` and its input method arguments `InArgs`, first builds the initial state by means of predicate `build_s0/3` and then calls predicate `execute/2`. In turn, `execute/2` calls predicate `step/3`, which produces `S2`, the state after executing the bytecode, and then calls predicate `execute/2` recursively with `S2` until we reach a `return` instruction with the empty stack. For brevity, we only show the definition of `step/3` for a selected set of instructions and omit the code of `build_s0/3` and `localVar_update/4`. The latter simply updates the value of a local variable. By using this interpreter, in a purely online setting, we define a *non-modular* decompilation scheme in terms of the generic function PE as follows.

Definition [DECOMP$_{\mathcal{L}_{bc}}$] Given a self-contained $\mathcal{L}_{bc}$-bytecode program P_{bc}, the (non-modular) LP-decompilation of P_{bc} can be obtained as:

$$\text{DECOMP}_{\mathcal{L}_{bc}}(P_{bc}) = PE(\mathit{Int}_{\mathcal{L}_{bc}} \cup P_{bc}, \emptyset, S)$$

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

Recent work in interpretive, online decompilation has focused on ensuring that the layer of interpretation is completely removed from decompiled programs, i.e., *effective*

```
int gcd(int x,int y){            int lcm(int x,int y){
  int res;                         int gcd = gcd(x,y);
  while (y != 0){                  if (gcd == 0) return 0;
    res = x%y; x = y;              else return x*y/gcd;}
    y = res;}
  return abs(x);}                int fact(int x){
                                   if (x == 0)
int abs(int x){                      return 1;
  if (x < 0) return -x;            else
  else return x; }                   return x*fact(x-1);}
```

```
Method gcd/2                   Method lcm/2
 0:load(1)                      0:load(0)
 1:if0eq(11)                    1:load(1)       Method fact/1
 2:load(0)                      2:call(gcd)      0:load(0)
 3:load(1)     Method abs/1     3:store(2)       1:if0ne(4)
 4:rem          0:load(0)       4:load(2)        2:push(1)
 5:store(2)     1:if0ge(5)      5:if0ne 8        3:return
 6:load(1)      2:load(0)       6:push(0)        4:load(0)
 7:store(0)     3:neg           7:return         5:load(0)
 8:load(2)      4:return        8:load(0)        6:push(1)
 9:store(1)     5:load(0)       9:load(1)        7:sub
 10:goto 0      6:return        10:mul           8:call(fact)
 11:load(0)                     11:load(2)       9:mul
 12:call(abs)                   12:div           10:return
 13:return                      13:return
```

Figure 3. Source code and $\mathcal{L}_{bc}$-bytecode for working example

decompilations are obtained. This requires the use of the following advanced control techniques. Type-based homeomorphic embedding ($\trianglelefteq_T$) [3] 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 [5]. Operator abstract must incorporate a polyvariance control mechanism [12] 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 partial evaluator based on an unfolding operator unfold$_{\trianglelefteq_T}$ and abstraction operator abstract$_{\trianglelefteq_T}$ which incorporate such advanced techniques and is able to remove the layer of interpretation.

3.2 Limitations of Non-Modular Decompilation

This section illustrates by means of the bytecode example in Fig. 3 that non-modular decompilation does not ensure a satisfactory handling of issues *(a)* and *(b)*. In the examples, we often depict the Java source code for clarity, but the partial evaluator works directly on the bytecode. The program consists of a set of methods that carry out arithmetic operations. Method `gcd` computes the greatest-common divisor, `abs` the absolute value, `lcm` the least-common multiple and `fact` the factorial recursively. The LP-decompilation obtained by applying Def. 3.1 is shown in Fig. 4. The partial evaluator performs a post-processing of renaming and argument filtering [9] for all calls except for calls to the `main` predicate, (as they represent calls to methods whose name we want to preserve). We identify the following limitations of non-modular decompilation.

(L1) Method invocations from `lcm` to `gcd` (index 2) and from `gcd` to `abs` (index 12) do not appear in the decompiled code. Instead, such calls have been *inlined* within their calling contexts and, as a consequence, the structure of the original code has been lost. For instance, the last two rules in the decompilation for `lcm`, `execute_1`, correspond to the `while` loop of `gcd`. 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 might become 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 Sect. 2. This is worse in the case of nested loops.

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

(L4) The decompiled program does not contain the code corresponding to recursive `fact` due to space limitations, as the decompiled code contains basically the whole interpreter. The problem with recursion is: assume we want to decompile method $m1$ whose code is $< pc_0 : bc_0, \ldots, pc_j : call(m1), \ldots, pc_n : return >$. There is an initial decompilation for $A_k = \mathtt{execute}(\mathtt{st}(\mathtt{fr}(\mathtt{m1}, \mathtt{pc_j}, \mathtt{os}, \mathtt{lv}), []), \mathtt{S_f})$ 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 get to the recursive call. At this point, the derivation must be stopped as $A_k \trianglelefteq_T A_l$. In order to ensure termination, the global control generalizes the above calls into $\mathtt{execute}(\mathtt{st}(\mathtt{fr}(\mathtt{m1}, \mathtt{pc_j}, \_, \_), \_), \mathtt{S_f})$, 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.

```
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 4. Decompiled (non-modular) code for working example

4 A Modular Decompilation Scheme

By *modular* decompilation, we refer to a decompilation technique whose decompilation unit is the method, i.e., we decompile a method at a time. We show that this approach overcomes the four limitations of non-modular decompilation described in Sect. 3.2 and answers issues *(a)* and *(b)* positively. In essence, we need to: (i) give a compositional treatment to method invocations, we show that this can be achieved by considering a *big-step* interpreter; (ii) provide a mechanism to residualize calls in the decompiled program, we automatically generate program annotations for this purpose; (iii) study the conditions which ensure that *separate* decompilation of methods is sound.

4.1 Big-step Interpreter to Enable Modularity

Traditionally, two different approaches have been considered to define language semantics, *big-step* (or *natural*) semantics and *small-step* semantics (see, e.g., [15]). 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 *invoke*. The transition for *invoke* in small-step defines the next step of the computation, i.e., the current frame is pushed on the call-stack and a new environment is initialized for the execution of the invoked method. Note that, after performing this step, we do not distinguish anymore between the code of the caller method and that of the callee. This avoids modularity of decompilation.

In the context of interpretive (de-)compilation of imperative languages, small-step interpreters are commonly used (see e.g. [24, 13, 4]). We argue that the use of a big-step interpreter is a necessity to enable modular decompilation which scales to realistic languages. In Fig. 5, we depict the

```
execute(S,S) :-
  S = st(M,PC,[_Top|_],_),
  bytecode(M,PC,return,_).
execute(S1,Sf) :-
  S1 = st(M,PC,_,_),
  bytecode(M,PC,Inst,_),
  step(Inst,S1,S2),
  execute(S2,Sf).
```

```
step(invoke(M2),S1,S2) :-
  S1 = st(M,PC,OS,LV),
  next(M,PC,PC2),
  split_OS(M2,OS,Args,OSRs),
  main(M2,Args,RV),
  S2 = st(M,PC2,[RV|OSRs],LV).
```

Figure 5. Fragment of big-step $\mathcal{L}_{bc}$ interpreter $\mathit{Int}^{bs}_{\mathcal{L}_{bc}}$

relevant part of the big-step interpreter for $\mathcal{L}_{bc}$-bytecode, named $\mathit{Int}^{bs}_{\mathcal{L}_{bc}}$. We can see that the *invoke* statement, after extracting the method parameters from the operand stack, calls recursively predicate `main/3` in order to execute the callee. Upon return from the method execution, the return value is pushed on the operand stack of the new state and execution proceeds normally. Also, we do not need to carry the call-stack explicitly within the state, but only the information for the current environment. I.e., states are of the form `st(M,PC,OStack,LocalV)`. This is because the call-stack is already available by means of the calls for predicate `main/3`.

The compositional treatment of methods in $\mathit{Int}^{bs}_{\mathcal{L}_{bc}}$ 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 $\mathit{Int}^{bs}_{\mathcal{L}_{bc}}$ does not present L4. E.g., the decompilation of a recursive method $m1$ starts from the call $\mathtt{main}(\mathtt{m1},\_,\_)$ and then reaches a call $\mathtt{main}(\mathtt{m1},\mathtt{args},\_)$ where `args` represents the particular arguments in the recursive call. This call is flagged as dangerous by local control and the derivation is stopped. The important points are that, unlike before, no recomputation is needed as the second call is necessarily an instance of the first one and, besides, there is no information loss associated to the generalization of the call-stack, as there is no stack. The recursion problem was first detected in [10] 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 [13], where a simpler control-flow analysis is performed before PE in order to collect all possible instructions which might follow the $return$. Such information may then be used to recover information lost by the generalization. This solution turns out to be also impractical for our purposes when considering realistic programs that make intensive use of library code (e.g. Java bytecode) as many continuations can follow. Our solution does not require the use of static analysis and, as our experiments show, does not pose scalability problems.

4.2 Guiding Online PE with Annotations

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

- **memo**: The current call is not further unfolded. Therefore, the call is later transferred to the global control to carry out its specialization separately.
- **rescall**: The current call is not further unfolded. Unlike calls marked **memo**, the current call is not transferred to the global control.

In the following, we denote by $\mathsf{unfold}^{\mathcal{A}}_{\trianglelefteq_T}$ the unfolding operator of Sect. 2 enhanced to use the above annotations. We adopt the same names for the annotations as in offline PE [19]. However, in offline PE they are the *only* means to control termination[1] and **rescall** annotations are in principle only used for builtins.

4.3 Modular Decompilation

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

[1]Hybrid approaches like [18] use online techniques to control termination in offline PE.

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

Definition [MOD-DECOMP$_{\mathcal{L}_{bc}}$] Given a $\mathcal{L}_{bc}$-bytecode program P_{bc}, a modular LP-decompilation of P_{bc} can be obtained as:

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

where the set of annotations $\mathcal{A}_{mod} = \{(m \in \textit{calls}(P_{bc})) \Rightarrow \textbf{rescall}\ main(m, \_, \_)\}$ and the initial sets of calls $S_m = \{main(m, \_, \_)\}$ for each $m \in \textit{defs}(P_{bc})$.

Let us briefly explain the above definition. Now the function PE is executed once per method defined in P_{bc}, starting each time from a set of calls, S_m, which contains a call of the form main(m, _, _) for method m. The set $\mathcal{A}_{mod}$ contains a **rescall** annotation which affects all methods invoked (but not necessarily internal) inside P_{bc}. When a method invocation is to be decompiled, the call step(invoke(m'),_,_) occurs during unfolding. We can see that, by using the big-step interpreter in Fig. 5, 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. 3 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) :-         | exec_1(A,0,C) :-
   main(gcd,[B,C],D),        |    main(abs,[A],C).
   D\=0, E is B*C,           | exec_1(A,B,F) :- B\=0,
   A is E//D.                |    H is A rem B, exec_1(B,H,F).
main(lcm,[A,B],0) :-         |
   main(gcd,[A,B],0).        | main(abs,[A],A) :- A>=0.
                             | main(abs,[B],A) :- B<0, A is-B.
main(gcd,[B,0],A) :-         |
   main(abs,[B],A).          | main(fact,[B],A) :-
main(gcd,[B,C],A) :-         |    B\=0, C is B-1,
   C\=0, D is B rem C,       |    main(fact,[C],D), A is B*D.
   exec_1(C,D,A).            | main(fact,[0],1).
```

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

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

Theorem 1 (soundness) *Consider a $\mathcal{L}_{bc}$-bytecode program P_{bc} and a concrete input I. Let P'_{bc} be the result of* MOD-DECOMP$_{\mathcal{L}_{bc}}(P_{bc})$ *and I' the LP representation of I. Then, A' is an answer for $P'_{bc} \cup \{I'\}$ iff A is the result of executing P_{bc} for the input I, where A' is the LP representation of A.*

The proof sketch might be found in Appendix A.

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

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

The proof sketch might be found in Appendix B.

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

5 Decompilation of Low-Level Languages

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

Let us explain *block-level* decompilation by means of an example. Consider the method m_{bl} to the left of Fig. 6, where we only show the relevant bytecode instructions, and its CFG in the center. 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* (C point) is a program point where two or more branches merge. In the CFG of m_{bl}, the only divergence (resp. convergence) point is pc_i (resp. pc_k).

By using the decompilation scheme presented so far, we obtain the SLD-tree shown to the right of Fig. 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 Sect. 2), each rule contains the decompiled code associated to the whole branch of the tree. This code duplication brings in two problems: it increases considerably the size of decompiled programs and also makes their execution slower. For instance, when $\overline{\mathtt{cond_i}}$ holds, the execution goes unnecessarily through $\{\mathtt{res_A}\}$ in the first rule, fails to prove $\mathtt{cond_i}$ and, then, attempts the second rule.

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

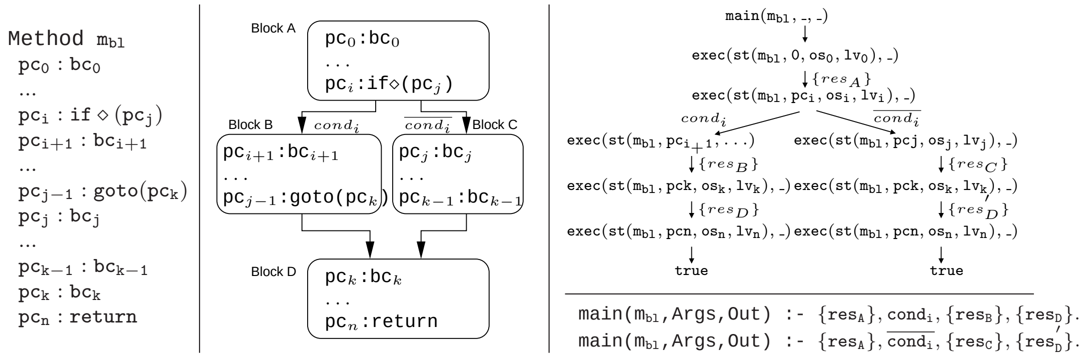
The amount of repeated residual code grows exponentially with the number of C and D points and the amount of re-evaluated code grows exponentially with the number of C points. Thus, we now aim for a *block-level* decompilation that helps overcome problems D and C above. Intuitively, a block-level decompilation must produce a residual rule for each block in the CFG. This can be achieved by building SLD-trees which correspond to each single block, rather than expanding them further.

The **memo** annotations presented in Sect. 4.2 facilitate the design of the block-level interpretive decompilation scheme. In particular, we can easily force the unfolding process to stop at D points by including a **memo** annotation for $\mathtt{execute}/2$ 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 Sect. 4.3, by viewing entries to blocks as method calls. Not surprisingly, the solution can be achieved similarly in our setting by: (1) stopping the derivation at $\mathtt{execute}/2$ calls whose PC corresponds to C points and (2) passing the call to the global control, and ensuring that it is evaluated in a sufficiently generalized context which covers all incoming contexts. The former point is ensured by the use of **memo** annotations and the latter by including in the initial set of terms a generalized call of the form $\mathtt{execute(st(m_{bl}, pc_k, \_, \_), \_)}$ for all C points, which forces such generalization. The next definition presents the *block-level* decompilation scheme where div_points(m) and conv_points(m) denote, resp., the set of D points and C points of a method m.

Definition [BLOCK-MOD-DECOMP$_{\mathcal{L}_{bc}}$] Given a $\mathcal{L}_{bc}$-bytecode program P_{bc}, a block-level, modular LP-decompilation of P_{bc} can be obtained as:

$$\text{BLOCK-MOD-DECOMP}_{\mathcal{L}_{bc}}(P_{bc}) = \bigcup_{\forall m \in \mathit{defs}(P_{bc})} PE(\mathit{Int}^{bs}_{\mathcal{L}_{bc}} \cup \mathit{code}(m), \mathcal{A}_m, S_m)$$

$$\begin{array}{lcl} \mathcal{A}_{blocks} & = & \{pc \in \mathsf{div\_points}(m) \cup \mathsf{conv\_points}(m) \Rightarrow \\ & & \quad \mathbf{memo}\ execute(st(m, pc, \_, \_), \_)\} \\ S_m & = & \{main(m, \_, \_)\} \cup \\ & & \{execute(st(m, pc, \_, \_), \_) \mid pc \in \mathsf{conv\_points}(m)\} \\ \mathcal{A}_m & = & \mathcal{A}_{mod} \cup \mathcal{A}_{blocks}, \text{ for each } m \in \mathit{defs}(P_{bc}). \end{array}$$



```
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
pc₀:bc₀
...
pcᵢ:if◇(pcⱼ)
Block B
cond_i
$\overline{cond_i}$
Block C
pc_{i+1}:bc_{i+1}
...
pc_{j-1}:goto(pc_k)
pc_j:bc_j
...
pc_{k-1}:bc_{k-1}
Block D
pc_k:bc_k
...
pc_n:return
main(m_bl, _, _)
exec(st(m_bl, 0, os_0, lv_0), _)
{res_A}
exec(st(m_bl, pc_i, os_i, lv_i), _)
cond_i
$\overline{cond_i}$
exec(st(m_bl, pc_{i+1}, ...)
exec(st(m_bl, pcj, os_j, lv_j), _)
{res_B}
{res_C}
exec(st(m_bl, pck, os_k, lv_k), _)
exec(st(m_bl, pck, os_k, lv_k), _)
{res_D}
{res_D'}
exec(st(m_bl, pcn, os_n, lv_n), _)
exec(st(m_bl, pcn, os_n, 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. $\mathcal{L}_{bc}$**-bytecode, CFG, unfolding tree and decompiled code of** m_{bl} **method**

An important point is that, unlike annotations used in offline PE [17] which are generated by only taking the interpreter into account, our annotations for block-level decompilation are generated by taking into account the particular program to be decompiled. Importantly, both the annotations and the initial set of calls can be computed automatically by performing two passes on the bytecode (see, e.g., [2, 26]). The result of performing block-level decompilation on m_{bl} is:

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

Now, the residual code associated to each block appears once in the code. This ensures that block-level decompilation preserves the CFG shape as dedicated decompilers do. Thus, the quality of our decompiled code is as good as that obtained by state-of-the-art decompilers [2, 23] but with the advantages of interpretive decompilation (see Sect. 1). We formalize the quality of block-level decompilation.

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

The proof sketch might be found in Appendix C.

6 Experimental Evaluation

We report on our implementation of a decompiler for full (sequential) Java Bytecode into Prolog. For the experimental evaluation we have used the set of benchmarks in the JOlden suite [6]. Most programs make an extensive use of library methods. Hence, non-modular decompilation cannot be assessed as we run into memory problems when trying to decompile the code of library calls. The experiments have been performed on an Intel Core 2 Duo 1.86GHz with 2GB of RAM, running Linux. Figure 7 depicts four charts measuring different aspects of the decompilation. We assess the differences between the *modular* and the *modular+block-level* (just *block-level* for short) approaches; as well as how the size of the programs affects the decompilation. We measure two aspects of the decompilation: the decompilation time (in milliseconds) per instruction and the decompiled program size (in bytes) per instruction. The decompilation time indicates the efficiency of the process and the size of decompiled programs is directly related to the decompilation quality. Each point $[X, Y]$ in the charts corresponds to the decompilation of a single method in the JOlden suite, where X represents the number of instructions of the method and Y the measured data (time or decompiled program size). The tables in the left-hand side show the data obtained (times in the top chart and sizes in the bottom one) for both the modular and the block-level decompilation. The variations in the block-level decompilation cannot be appreciated when combined with modular. Thus, we include in the tables on the right-hand side the figures for the block-level decompilation in isolation such that we adjust the scale on the Y-axis to the domain of the data.

From the charts, we conclude: (1) Times per instruction are notably larger for the smallest methods, as can be seen by looking at the initial curve in the charts. This is because the overhead introduced for starting a new decompilation is more noticeable when the time for decompilation itself is small, while it becomes negligible for larger methods. The same happens for the size of the decompiled programs. (2) Block-level decompilation achieves important speedups in general (for all methods with more than 40 instructions). Besides, it obtains significantly smaller decompiled programs. The speedups per package range from 3.36 in **power** to 31.4 in **bisort** for the decompilation times; and from 2.5 times smaller in **power** to 9 times smaller in **bisort** for the decompiled program sizes. Note that there is a clear correspondence between both measures, since C points introduce both inefficiency and size increase in decompilation, as explained in Sect. 5. Moreover, modular decompilation runs out of memory for some of the largest methods. This is again related to code duplication (C and D points) and (re-)evaluation (C points), which grow exponentially. (3) The most important conclusion is that, while in modu-

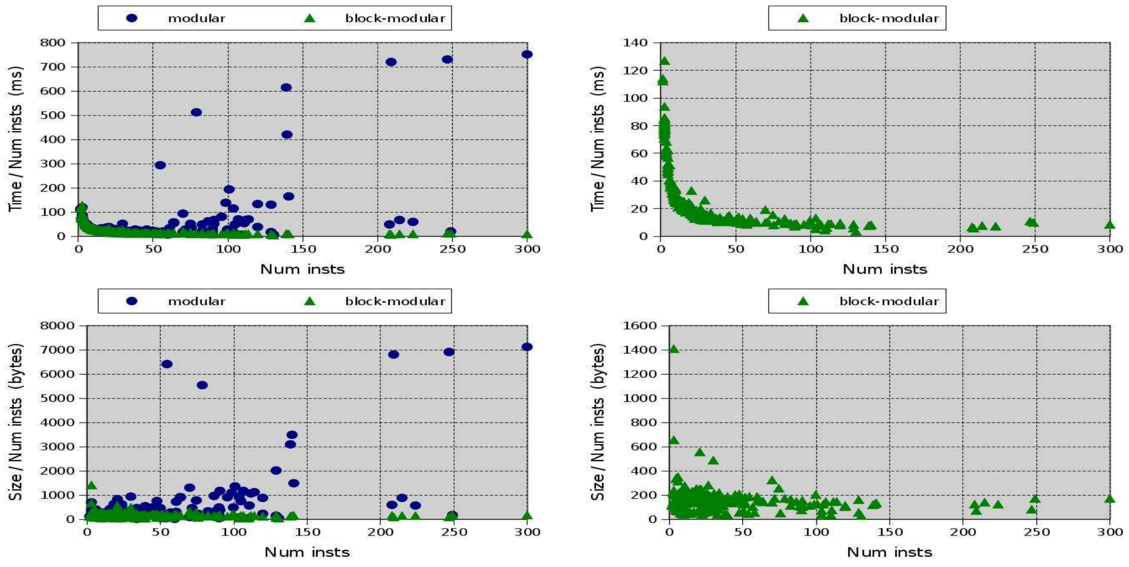


Figure 7. Evaluating *modular* decompilation vs. *modular+block-level* deompilation with the JOlden Suite

lar decompilation both the times and the sizes per instruction greatly increase with the size of the benchmarks, this does not happen in the block-level scheme. In block-level decompilation, these figures are totally stable (mostly constant) for all methods with more than 40 instructions. This demonstrates that both the decompilation times and the decompiled program sizes are *linear* with the size of the input bytecode program, thus demonstrating the scalability of the block-level decompilation. One might wonder why there are still small variations in the ratio. In our experience, the following points also matter: 1) the complexity of the control flow of the methods, 2) the relative complexity of the bytecode instructions used, e.g., instructions which operate in the heap tend to produce more residual code, 3) the structure w.r.t. methods of the program, e.g., classes with methods of medium size tend to result in better decompilations than those with few large methods or many small ones.

7 Conclusions and Related Work

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

Previous work in *interpretative* (de-)compilation has mainly focused on proving that the approach is feasible for small interpreters and medium-sized programs. The focus has been on demonstrating its *effectiveness*, i.e., that the so-called interpretation layer can be removed from the compiled programs. To achieve effectiveness, offline [17], online [4, 13, 24] and hybrid [18] PE techniques have been assessed and novel control strategies have been proposed and proved effective [12, 3]. Our work starts off from the premise that interpretive decompilation is feasible and effective as proved by previous work and studies further issues which have not been explored before. A main objective of our work is to investigate, and provide the necessary techniques, to make interpretive decompilation scale in practice. A further goal is to ensure, and provide the techniques, that decompiled programs preserve the structure of the original programs and that its quality is comparable to that obtained by dedicated decompilers. We believe that the techniques proposed in this paper, together with their experimental evaluation, provide for the first time actual evidence that the interpretive theory proposed by Futamura in the 70s is indeed an appealing and feasible alternative to the development of ad-hoc decompilers from modern languages to intermediate representations.

References

[1] 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 R. D. Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

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

[4] 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.

[5] 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.

[6] J. S. Collection. `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`.

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

[8] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

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

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

[11] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE. In *Proc. of 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume LNCS 3091, pages 210–220. Springer-Verlag, 2004.

[12] 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.

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

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

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

[16] M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 379–403. Springer, 2002.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] G. Marpons, M. Carro, J. Mariño, A. Herranz, L.-Å. Fredlund, and J. J. M. Navarro. Towards Checking Coding Rule Conformance Using Logic Programming. Poster session at SAS 2007, August 2007.

[23] 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.

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

[25] 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.

[26] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of CASCON 1999*, pages 125–135, 1999.

(Appendix included for reviewer convenience.)

A Proof of Theorem 1

Proof [sketch] Regarding completeness, we first have to exclude calls to external predicates 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 ones, closedness is already ensured by traditional PE [21]. We can 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 T^{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 T^{pe}. Regarding correctness, the full code of the interpreter must be studied. 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 [25]), 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. □

B Proof of Proposition 1

Proof Clearly, 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. □

C Proof of Proposition 2

Proof [sketch] The proof follows easily by contradiction. In order to prove (I), consider that two resultants contain residual code for the same bytecode instruction. This can be due to two reasons. (a) There is in the SLD-tree a D point which leads to two derivations. This is not possible because D points are annotated as **memo** and hence the derivation must have been stopped. (b) There are two separate trees which contain derivations for instructions of the same block. Then, this block must be a C block. Hence, it is not possible because C points are annotated as **memo** and hence the derivation must have stopped before. We focus now on D blocks to prove (II). Consider that there have been two evaluations of an instruction pc_x within a D block B starting at $pc_1 \in \mathsf{conv\_points}(M)$. Then, there must have been two different instances $\mathtt{execute}(\mathtt{st}(\mathtt{M}, \mathtt{pc}_1, \mathtt{A}, \mathtt{B}), \mathtt{C}))$ and, later, $\mathtt{execute}(\mathtt{st}(\mathtt{M}, \mathtt{pc}_1, \mathtt{D}, \mathtt{E}), \mathtt{F}))$. This is not possible because there exists the initial call $execute(st(M, pc_1, \_, \_), \_))$ in S_m which does not allow the evaluation of $\mathtt{execute}(\mathtt{st}(\mathtt{M}, \mathtt{pc}_1, \mathtt{D}, \mathtt{E}), \mathtt{F}))$. □