Towards Modular Interpretive Decompilation of Low-Level Code to Prolog

Miguel Gómez-Zamalloa[1] Elvira Albert[1] Germán Puebla[2]

[1] *DSIC, Complutense University of Madrid,* {mzamalloa,elvira}@fdi.ucm.es

[2] *Technical University of Madrid,* german@fi.upm.es

Abstract

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 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. The proposed scheme ensures that: 1) each method/block is decompiled just once, and 2) each program point is traversed at most once during decompilation.

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. In particular, rule-based representations used in declarative programming in general—and in Prolog in particular—provide a convenient formalism to define such representations. E.g., as it can be seen in [1,18,20,10], the operand stack used in a low-level language can be represented by means of explicit logic variables and that its unstructured control flow can be transformed into recursion.

All above cited approaches (except [10]) develop *ad-hoc* decompilers to carry out the particular decompilations. An appealing alternative to the development of dedicated decompilers is the so-called *interpretive* decompilation by *partial evaluation* (PE) [11]. PE is an automatic program transformation technique which specializes a program w.r.t. part of its known input data. Interpretive compilation was proposed

in Futamura's seminal work [5], whereby compilation of a program P written in a (*source*) programming language L_S into another (*object*) programming language L_O is achieved by specializing an interpreter for L_S written in L_O w.r.t. P. The advantages of interpretive (de-)compilation w.r.t. dedicated (de-)compilers are well-known and discussed in the PE literature (see, e.g., [3]). 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 (or trust) that ad-hoc decompilers preserve the program semantics; *easier to maintain*, new changes in the language semantics can be easily reflected in the interpreter.

There have been several proofs-of-concept of interpretive (de)compilation (e.g., [3,10,13]), but there remain interesting open issues when it comes to assess its power and/or limitations to decompile a real language: *(a) does the 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 ensures quality decompilations which preserve the original program's structure. Our main contributions are summarized as follows:

(i) 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.

(ii) We present a modular decompilation scheme which is correct and complete for the proposed 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.

(iii) 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.

2 Basics of Partial Deduction

We assume familiarity with basic notions of logic programming [17]. 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. [6]) builds upon the SLD trees mentioned above. We now introduce a generic function PE, which is parametric w.r.t. the *unfolding rule*, unfold, and the *abstraction operator*, abstract and captures the essence of most algorithms for PE of logic programs:

```
1: function PE (P, A, S_0)
2:     repeat
3:         T^pe := unfold(S_i, P, A);
4:         S_{i+1} := abstract(S_i, leaves(T^pe), A);
5:         i := i + 1;
6:     until S_i = S_{i-1}      % (modulo renaming)
7:     return codegen(T^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_0. At each iteration, the so-called *local control* is performed by the unfolding rule unfold (L3), which takes the current set of terms S_i, the program and the annotations and constructs a *partial* SLD tree for each call in S_i. Trees are partial in the sense that, in order to guarantee termination of the unfolding process, it must be possible to choose *not* to further unfold a goal, and rather allow leaves in the tree with a non-empty, possibly non-failing, goal. The particular unfold operator determines which call to select from each goal and when to stop unfolding. The partial evaluator may have to build several SLD-trees to ensure that all calls left in the leaves are "covered" by the root of some tree. This is known as the *closedness* condition of PE [16]. In the *global control*, those calls in the leaves which are not covered are added to the new set of terms to be partially evaluated, by the operator abstract (L4). At the next iteration, an SLD-tree is built for such call. Thus, basically, the algorithm iteratively (L2-6) constructs partial SLD trees until all their leaves are covered by the root nodes. An essential point of the operator abstract is that it has to perform "generalizations" on the calls that have to be partially evaluated in order to avoid computing partial SLD trees for an infinite number of calls. A partial evaluation of P w.r.t. S is then systematically extracted from the resulting set of calls T^{pe} in the final phase, codegen in L7.

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 Decomp. and Limitations

This section describes the state of the art in interpretive decompilation of low-level languages to Prolog, including recent work in [10,2,8,3]. We do so by formulating non-modular decompilation in a generic way and identifying its limitations.

We consider a very simple imperative, stack-based, low-level language with unstructured control flow, denoted as $\mathcal{L}_{bc}$. It goes in the spirit of Java Bytecode but manipulating only integer numbers and without object-oriented features [1]. 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:

$$\begin{aligned} BcInst ::= \ & \mathsf{push(x)} \mid \mathsf{load(v)} \mid \mathsf{store(v)} \mid \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{rem} \mid \mathsf{neg} \mid \\ & \mathsf{if} \diamond \mathsf{(pc)} \mid \mathsf{if0} \diamond \mathsf{(pc)} \mid \mathsf{goto(pc)} \mid \mathsf{return} \mid \mathsf{invoke(mn)} \end{aligned}$$

[1] Our implementation supports full Java Bytecode

```
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(invoke(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).
```

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

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 variable array to the stack (and viceversa); add, sub, mul, div, rem and neg perform the usual arithmetic operations, if, if0 and goto are the usual branching instructions; return marks the end of methods and invoke calls 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 method 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 Decompilation

We rely on the so-called "interpretive approach" to compilation described in Sect. 1, also known as first Futamura projection [5]. In particular, the decompilation of a $\mathcal{L}_{bc}$-bytecode program P_{bc} to LP (for short LP-decompilation) might be obtained by specializing (with an LP partial evaluator) a $\mathcal{L}_{bc}$-interpreter written in LP w.r.t. P_{bc}. In Fig. 1 we show a fragment of a (small-step) $\mathcal{L}_{bc}$ interpreter implemented in Prolog, named $Int_{\mathcal{L}_{bc}}$. We assume that the code for every method in the bytecode program P_{bc} is represented as a set of facts `bytecode/3` such that, for every pair $pc_i : bc_i$ in the code for method m, we have a fact `bytecode(`m`,`pc_i`,`bc_i`)`. The state carried around by the interpreter is of the form `st(Fr,FrStack)` where `Fr` represents the current frame (environment) and `FrStack` the stack of frames (call stack) implemented as a list. Frames are of the form `fr(M,PC,OStack,LocalV)`, where `M` represents the current method, `PC` the program counter, `OStack` the operand stack and `LocalV` the list of local variables. Predicate `main/3`, given the method to be interpreted `Method` and its input method arguments `InArgs`, builds the initial state, calling `build_s0/3`, and then iterates on predicate `execute/2` until a `return` instruction with the empty stack is reached. The state transition function is modelled by means of predicate `step/3`. 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 3.1 [$\text{DECOMP}_{\mathcal{L}_{bc}}$] Given a self-contained $\mathcal{L}_{bc}$-bytecode program P_{bc}, the (non-modular) LP-decompilation of P_{bc} can be obtained as:

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

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

Recent work in interpretive, online decompilation has focused on ensuring that

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

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

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

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

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

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

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

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

the layer of interpretation is completely removed from decompiled programs, i.e., *effective* decompilations are obtained. This requires the use of some advanced control techniques like: the *Type-based homeomorphic embedding* ($\trianglelefteq_T$) [2], an unfolding operator able to safely perform non-leftmost unfolding steps in presence of builtins [4] and an `abstract` operator with an advanced polyvariance control mechanism [8]. Our starting point is thus a state-of-the-art partial evaluator based on such advanced techniques which 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. 2 that non-modular decompilation does not ensure a satisfactory handling of issues *(a)* and *(b)*. In the examples, we often depict the Java source code for clarity, but the PE works directly on the bytecode. The program consists of a set of methods that carry out arithmetic operations. Method `gcd` computes the greatest-common divisor, `abs` the absolute value, `lcm` the least-common multiple and `fact` the factorial recursively. The LP-decompilation obtained by applying Def. 3.1 is shown in Fig. 3. We identify the following limitations of non-modular decompilation:

(L1) Method invocations from `lcm` to `gcd` (index 2) and from `gcd` to `abs` (index 12) do not appear in the decompiled code. Instead, such calls have been *inlined* within their calling contexts and, as a consequence, the structure of the original code has been lost. This happens because calls to methods are dealt with in a *small-step* fashion within the interpreter.

(L2) As a consequence decompilation might become very inefficient. For instance, if n calls for the same method appear within a code, such method will be decompiled n times. Even worse, when there is a method invocation inside a loop, it code might be evaluated unnecessarily various times.

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

(L4) The decompiled program does not contain the code corresponding to recursive `fact` due to space limitations, as the decompiled code contains basically the whole interpreter. The problem with recursion was first detected in [7] and particu-

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

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).
```

Fig. 3. Decompiled (non-modular) code for working example

lar details together with a discussion on partial solutions proposed in the literature can be found in [9]. Limitations L1 and L4 answer issue *(b)* negatively.

4 An Interpretive 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.

4.1 Big-step Interpreter to Enable Modular Decompilation

Traditionally, two different approaches have been considered to define language semantics, *big-step* semantics and *small-step* semantics (see, e.g., [12]). Essentially, in big-step semantics, transitions relate the initial and final states for each statement, while in small-step semantics transitions define the *next* step of the execution for each statement. In the context of bytecode interpreters, 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, then after performing this step, we do not distinguish anymore between the code of the caller method and that of the callee. This avoids modularity of decompilation.

In the context of interpretive (de)compilation of imperative languages, small-step interpreters are commonly used (see e.g. [19,10,3]). We argue that the use of a big-step interpreter is a necessity to enable modular decompilation which scales to realistic languages. Now we depict the relevant part of the big-step interpreter for $\mathcal{L}_{bc}$-bytecode, named $Int^{bs}_{\mathcal{L}_{bc}}$.

```
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,OSRest),
    main(M2,Args,RV),
    S2 = st(M,PC2,[RV|OSRest],LV).
```

We can see that the *invoke* statement 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)`.

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.

4.2 Guiding Online Partial Evaluation with Annotations

We now present the annotations we use to provide additional control information to PE. 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 ($Ann \in \{\textbf{memo}, \textbf{rescall}\}$) 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.

We adopt the same names for the annotations as in offline PE [15]. However, in offline PE they are the *only* means to control termination [2] 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. Soundness and completeness of separate decompilation are discussed in [9]. In the following we present a modular decompilation scheme which, by combining the big-step interpreter with the use of **rescall** annotations, enables separate decompilation.

Definition 4.1 [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:

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

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

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 (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 $Int^{bs}_{\mathcal{L}_{bc}}$, a subsequent call `main(m',_,_)` will be generated. As there is a **rescall** annotation affecting 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.

[2] Hybrid approaches like [14] use online techniques to control termination in offline PE.

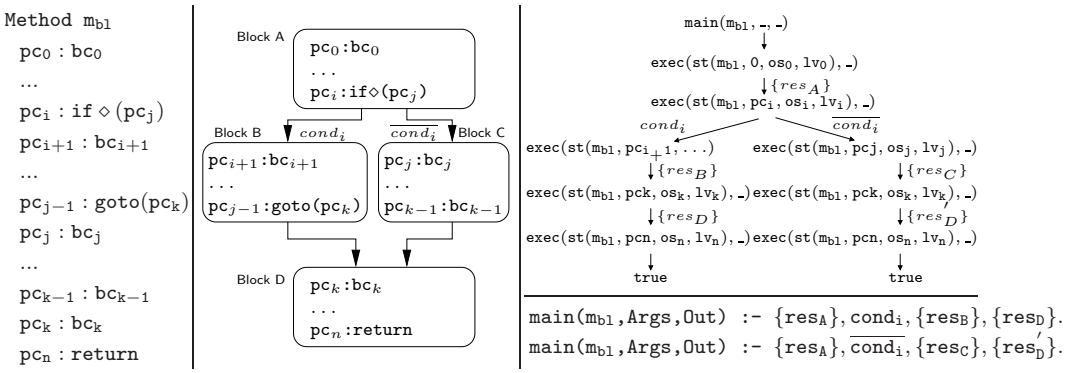

Fig. 4. $\mathcal{L}_{bc}$-bytecode, CFG, unfolding tree and decompiled code of m_{bl} method

Example 4.2 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(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(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(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 `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 has been completely removed. Thus, L1 and L4 from Sect. 3.2 have been successfully solved.

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

By using the decompilation scheme presented so far, we obtain the SLD-tree shown to the right of Fig. 4, in which all calls are completely unfolded as there is no termination risk (nor **rescall** annotation). The decompiled code is shown under the tree. We use $\{\texttt{res}_\texttt{X}\}$ to refer to the residual code emitted for BlockX and $\texttt{cond}_\texttt{i}$ to refer to the condition associated to the branching instruction at $\texttt{pc}_\texttt{i}$ ($\overline{\texttt{cond}_\texttt{i}}$ denotes

its negation). The quality of the decompiled code is not optimal due to:

D. Decompiled code $\{\texttt{res}_\texttt{A}\}$ for BlockA is duplicated. During PE, this code is evaluated once but each rule contains the 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. E.g., 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 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 (endangering scalability) and quality of decompilation (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. 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.

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 `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 `execute/2` calls whose PC corresponds to C points and (2) passing the call to the global control, and ensuring that it is evaluated in a sufficiently generalized context which covers all incoming contexts. The former point is ensured by the use of **memo** annotations and the latter by including in the initial set of terms a generalized call of the form $\texttt{execute}(\texttt{st}(\texttt{m}_\texttt{bl}, \texttt{pc}_\texttt{k}, \_, \_), \_)$ for all C points, which forces such generalization. The next definition presents the *block-level* decompilation scheme where $\mathsf{div\_points}(m)$ and $\mathsf{conv\_points}(m)$ denote, resp., the set of D points and C points of a method m.

Definition 5.1 [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:

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

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

Importantly, both the annotations and the initial set of calls can be computed

automatically by performing two passes on the bytecode (see, e.g., [1,20]). The result of performing block-level decompilation on m_{bl} is:

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

Now, the residual code associated to each block appears once in the decompiled 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 like [1,18] but with the advantages of interpretive decompilation (see Sect. 1).

6 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 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. 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 (see [9]), 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
This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

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

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

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Modular Decompilation of Low-Level Code by Partial Evaluation. Technical Report CLIP2/2008.0, UPM, 2008.

[10] 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.

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

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

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

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

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

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

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

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

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

[20] 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.