Modular Interpretive Decompilation of Low-Level Code by Partial Evaluation

Elvira Albert[1]

joint work with

Miguel Gómez-Zamalloa[1] **and Germán Puebla**[2]

(1) Complutense University of Madrid (Spain)
(2) Technical University of Madrid (Spain)

Beijing, September 2008

Introduction

Motivation

Low-level code ⇒ Intermediate representations

- **Mobile environments**: only *low-level code* available.
- Analysis tools unavoidably more complicated.
  - unstructured control flow,
  - use of operand stack,
  - use of heap, etc.

Introduction

Motivation

Low-level code ⇒ Intermediate representations

- **Mobile environments**: only *low-level code* available.
- Analysis tools unavoidably more complicated.
  - unstructured control flow,
  - use of operand stack,
  - use of heap, etc.
- Decompiling to intermediate representations:
  - abstracts away particular language features.
  - simplifies development of analyzers, model checkers, etc.
  - variants: *clause-based*, *BoogiePL*, *Soot*, etc.

Introduction

Motivation

Low-level code ⇒ Intermediate representations

- **Mobile environments**: only *low-level code* available.
- Analysis tools unavoidably more complicated.
  - unstructured control flow,
  - use of operand stack,
  - use of heap, etc.
- Decompiling to intermediate representations:
  - abstracts away particular language features.
  - simplifies development of analyzers, model checkers, etc.
  - variants: *clause-based*, *BoogiePL*, *Soot*, etc.

High-level (declarative) languages

- Convenient intermediate representation:
  - iterative constructs (loops) ⇒ recursion.
  - all variables in local scope of methods represented uniformly.
- Advanced tools (for declarative) languages re-used.

Introduction

Interpretive Decompilation

- Most of the approaches develop hand-written decompilers.
- Appealing alternative: interpretive decompilation
- PE allows specializing a program w.r.t. some part of its input.

Introduction

Interpretive Decompilation

- Most of the approaches develop hand-written decompilers.
- Appealing alternative: interpretive decompilation
- PE allows specializing a program w.r.t. some part of its input.

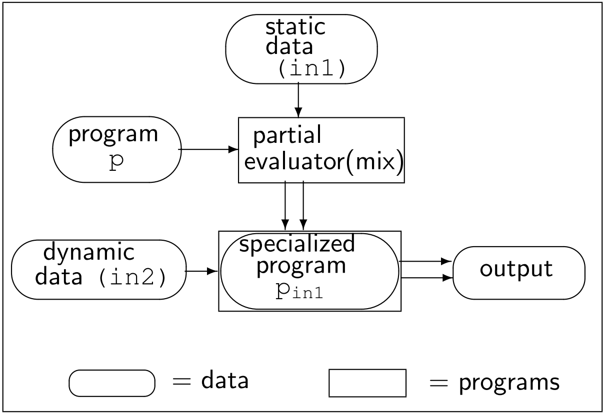
Definition (1st Futamura Projection)

A program P written in L_S can be compiled into another language L_O by specializing an interpreter for L_S written in L_O w.r.t. P.

First Futamura Projection

Partial Evaluation and the Interpretive Approach

$\texttt{p}(\texttt{in1}, \texttt{in2}) = \texttt{output}$

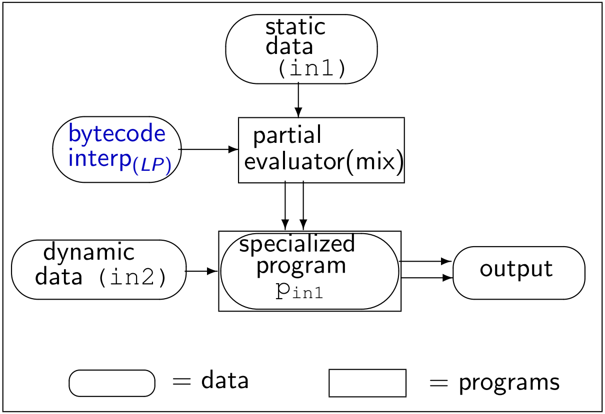


$[[\texttt{p}]]\ [\texttt{in1}, \texttt{in2}] = [[\ [[\texttt{mix}]]\ [\texttt{p}, \texttt{in1}]\ ]]\ [\texttt{in2}]$

First Futamura Projection

Partial Evaluation and the Interpretive Approach

$\texttt{p}(\texttt{in1}, \texttt{in2}) = \texttt{output}$

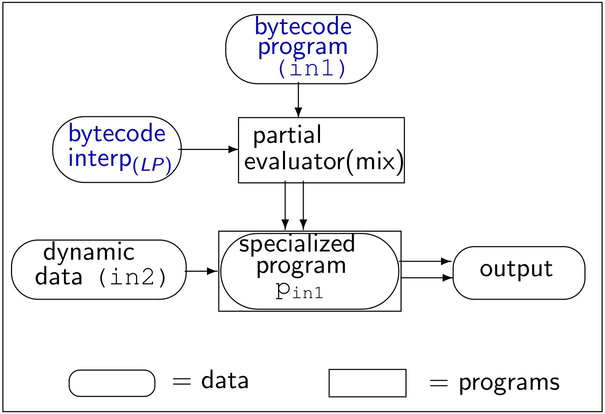


$$[\![\texttt{bc\_interp}]\!]\ [\texttt{in1}, \texttt{in2}] = [\![\ [\![\texttt{mix}]\!]\ [\texttt{bc\_interp}, \texttt{in1}]\ ]\!]\ [\texttt{in2}]$$

First Futamura Projection

Partial Evaluation and the Interpretive Approach

$\texttt{p}(\texttt{in1}, \texttt{in2}) = \texttt{output}$

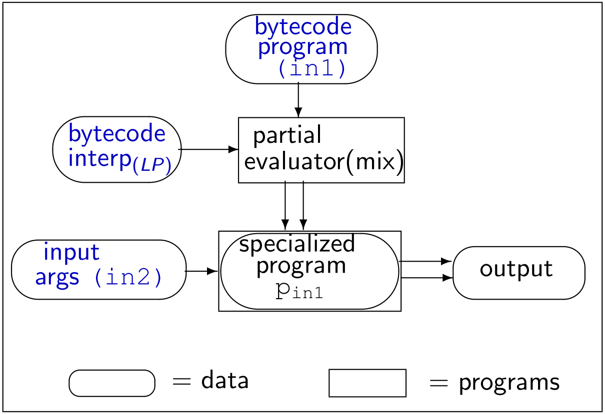


$[\![\texttt{bc\_interp}]\!]\ [\texttt{in1}, \texttt{in2}] = [\![\ [\![\texttt{mix}]\!]\ [\texttt{bc\_interp}, \texttt{in1}]\ ]\!]\ [\texttt{in2}]$

First Futamura Projection

Partial Evaluation and the Interpretive Approach

$\texttt{p}(\texttt{in1}, \texttt{in2}) = \texttt{output}$

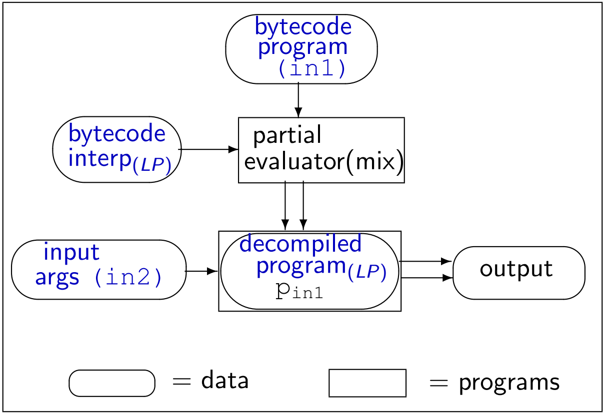


$$[\![\texttt{bc\_interp}]\!]\ [\texttt{in1}, \texttt{in2}]\ =\ [\![\ [\![\texttt{mix}]\!]\ [\texttt{bc\_interp}, \texttt{in1}]\ ]\!]\ [\texttt{in2}]$$

First Futamura Projection

Partial Evaluation and the Interpretive Approach

$\mathtt{p}(\mathtt{in1}, \mathtt{in2}) = \mathtt{output}$



$[\![\mathtt{bc\_interp}]\!]\ [\mathtt{in1}, \mathtt{in2}] \ = \ [\![\ [\![\mathtt{mix}]\!]\ [\mathtt{bc\_interp}, \mathtt{in1}]\ ]\!]\ [\mathtt{in2}]$

Example 1: Source code

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

Example 1: Source code

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

bytecode

```
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:return
```

Example 1: Source code

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

bytecode

```
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:return
```

bytecode interpreter

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

execute(S1,Sf) :-
    S1 = st(PC,_,_)),
    bytecode(PC,Inst,_),
    step(Inst,S1,S2),
    execute(S2,Sf).
......

step(push(X),S1,S2) :-
    S1 = st(PC,S,L)),
    next(PC,PC2),
    S2 = st(PC2,[X|S],L)).

step(store(X),S1,S2) :-
    S1 = st(PC,[I|S],LV)),
    next(PC,PC2),
    localVar_update(LV,X,I,
    S2 = st(PC2,S,LV2)).
.............
```

Example 1: Source code

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

bytecode

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

bytecode interpreter

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

execute(S1,Sf) :-
    S1 = st(PC,_,_)),
    bytecode(PC,Inst,_),
    step(Inst,S1,S2),
    execute(S2,Sf).
......

step(push(X),S1,S2) :-
    S1 = st(PC,S,L)),
    next(PC,PC2),
    S2 = st(PC2,[X|S],L)).

step(store(X),S1,S2) :-
    S1 = st(PC,[I|S],LV)),
    next(PC,PC2),
    localVar_update(LV,X,I,
    S2 = st(PC2,S,LV2)).
.............
```

Decompiled code

```
main(gcd,[X,0],X).
main(gcd,[X,Y],Z) :- Y \= 0,
    R is X rem Y, exec_1(Y,R,Z).

exec_1(Y,0,Y).
exec_1(Y,R,Z) :- R \= 0,
    R' is Y rem R, exec_1(R,R',Z).
```

Contributions in Interpretive Decompilation

Advantages w.r.t. dedicated (de-)compilers:

- flexibility: interpreter easier to modify;
- more reliable: easier to trust that the semantics preserved;
- easier to maintain: new changes easily reflected in interpreter;
- easier to implement: provided a partial evaluator is available.

Contributions in Interpretive Decompilation

Advantages w.r.t. dedicated (de-)compilers:

- flexibility: interpreter easier to modify;
- more reliable: easier to trust that the semantics preserved;
- easier to maintain: new changes easily reflected in interpreter;
- easier to implement: provided a partial evaluator is available.

Only proofs-of-concept in interpretive decompilation:

- e.g. in [PADL'07] we decompile a subset of Java Bytecode to Prolog.
- Open issues we have answered in this work:
  - Scalability: first *modular* decompilation scheme by PE
  - Structure preservation: of the original program
  - Quality: equivalent to hand-written decompilers

Conclusions and Future Work

- We have provided mechanisms to positively answer these issues:
  - Method optimality: Code for each method is decompiled only once $\Rightarrow$ Big-step interpreter and PE annotations.
  - Block optimality: Code for each instruction is emitted and evaluated at most once $\Rightarrow$ PE annotations and pre-analysis.

Conclusions and Future Work

- We have provided mechanisms to positively answer these issues:
  - Method optimality: Code for each method is decompiled only once $\Rightarrow$ Big-step interpreter and PE annotations.
  - Block optimality: Code for each instruction is emitted and evaluated at most once $\Rightarrow$ PE annotations and pre-analysis.
- Implemented an interpretive decompiler of Java Bytecode to Prolog.

Conclusions and Future Work

- We have provided mechanisms to positively answer these issues:
  - Method optimality: Code for each method is decompiled only once ⇒ Big-step interpreter and PE annotations.
  - Block optimality: Code for each instruction is emitted and evaluated at most once ⇒ PE annotations and pre-analysis.
- Implemented an interpretive decompiler of Java Bytecode to Prolog.
- Future work: Special handling for the heap, exploit instrumented decompilation, improve efficiency, applications, etc.

Question to SCAM Audience

- Are we happy with hand-written decompilers or we would like more flexible approaches?

Contributions

Contribution 1

- *Modular* decompilation: decompile a method at a time
- First *modular* decompilation scheme by PE:
  - compositional treatment to method invocation $\Rightarrow$ consider a *big-step* interpreter;
  - "residualize" calls in decompiled program, we automatically generate program annotations for this purpose;

Contributions

Contribution 1

- *Modular* decompilation: decompile a method at a time
- First *modular* decompilation scheme by PE:
  - compositional treatment to method invocation ⇒ consider a *big-step* interpreter;
  - "residualize" calls in decompiled program, we automatically generate program annotations for this purpose;

Proposition (modular optimality)

We decompile the code corresponding to each method in P_{bc} exactly once.

Decompilation of Low-level Code

Contribution 2

- Is possible to obtain by interpretive decompilation programs whose quality is equivalent to dedicated decompilers?
- Idea: since decompilers first build a *CFG* for the method, study how a similar notion can be used for controlling PE of the interpreter
- Block-level decompilation produce a rule for each block in the CFG.

Decompilation of Low-level Code

Contribution 2

- Is possible to obtain by interpretive decompilation programs whose quality is equivalent to dedicated decompilers?
- Idea: since decompilers first build a *CFG* for the method, study how a similar notion can be used for controlling PE of the interpreter
- Block-level decompilation produce a rule for each block in the CFG.

Proposition (block optimality)

1. residual code for each bytecode instruction emitted once;
2. each bytecode instruction evaluated at most once;

Conclusions and Future Work

- Open issues: scalability, structure preservation, quality ...
- We have provided mechanisms to positively answer these issues:
  - Method optimality: Code for each method is decompiled only once $\Rightarrow$ Big-step semantics and PE annotations.
  - Block optimality: Code for each instruction is emitted and evaluated at most once $\Rightarrow$ PE annotations and pre-analysis.

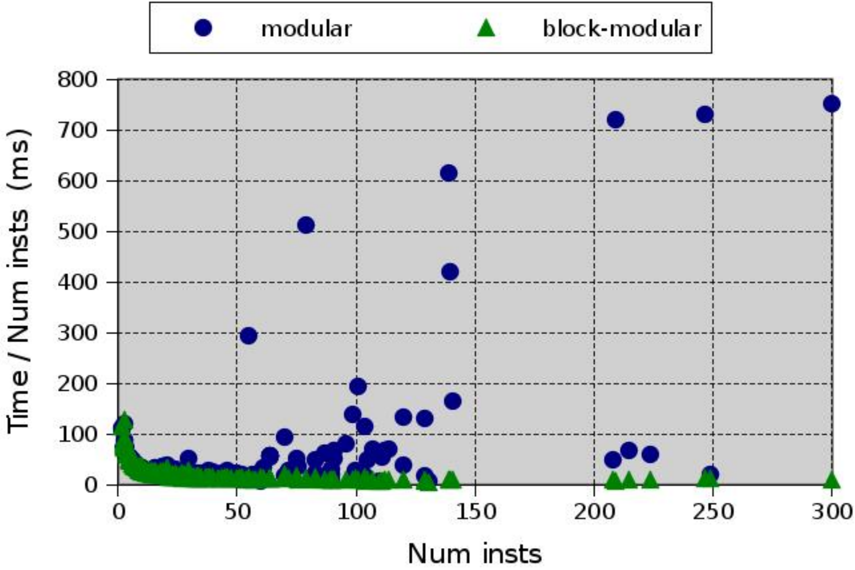
Conclusions and Future Work

- Open issues: scalability, structure preservation, quality ...
- We have provided mechanisms to positively answer these issues:
  - Method optimality: Code for each method is decompiled only once $\Rightarrow$ Big-step semantics and PE annotations.
  - Block optimality: Code for each instruction is emitted and evaluated at most once $\Rightarrow$ PE annotations and pre-analysis.
- Implemented an interpretive decompiler of Java Bytecode to Prolog.
- Average improvements: 10 times faster decompilations and 5 times smaller decompiled program sizes (even we get ∞ gains).

Conclusions and Future Work

- Open issues: scalability, structure preservation, quality ...
- We have provided mechanisms to positively answer these issues:
  - Method optimality: Code for each method is decompiled only once $\Rightarrow$ Big-step semantics and PE annotations.
  - Block optimality: Code for each instruction is emitted and evaluated at most once $\Rightarrow$ PE annotations and pre-analysis.
- Implemented an interpretive decompiler of Java Bytecode to Prolog.
- Average improvements: 10 times faster decompilations and 5 times smaller decompiled program sizes (even we get ∞ gains).
- Future work: Special handling for the heap, exploit instrumented decompilation, improve efficiency, applications, etc.

Experimental Evaluation (JOlden benchmarks suite)



Intraprocedural Decompilation

- We consider the $\mathcal{L}_{bc}$-bytecode language ($\mathcal{L}_{bc} \subset$ Java bytecode).

$$\begin{array}{rl} \mathit{Inst} ::= & \mathsf{push(x)} \mid \mathsf{load(v)} \mid \mathsf{store(v)} \mid \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{rem} \mid \mathsf{neg} \mid \\ & \mathsf{if} \diamond \mathsf{(pc)} \mid \mathsf{if0} \diamond \mathsf{(pc)} \mid \mathsf{goto(pc)} \mid \mathsf{return} \end{array}$$

- State $\equiv \langle PC, OpStack, LocalVars \rangle$

The $\mathcal{L}_{bc}$-bytecode interpreter

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

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

```
step(push(X),S1,S2) :-
   S1 = st(PC,S,L)),
   next(PC,PC2),
   S2 = st(PC2,[X|S],L)).

step(store(X),S1,S2) :-
   S1 = st(PC,[I|S],LV)),
   next(PC,PC2),
   localVar_update(LV,X,I,LV2),
   S2 = st(PC2,S,LV2)).

step(goto(PC),S1,S2) :-
   S1 = st(_,S,LV)),
   S2 = st(PC,S,LV)).
```

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees

$\mathtt{main}(\mathtt{gcd}, [\mathtt{X}, \mathtt{Y}], \mathtt{Z})$

Decompiled code

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Unfolding trees

$$\mathtt{main}(\mathtt{gcd}, [\mathtt{X}, \mathtt{Y}], \mathtt{Z})$$

$$\downarrow$$

$$\mathtt{exec}(\mathtt{st}(\mathtt{0}, [], [\mathtt{X}, \mathtt{Y}, \mathtt{0}]), \mathtt{S_f})$$

Decompiled code

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees

$$\mathtt{main}(\mathtt{gcd}, [\mathtt{X}, \mathtt{Y}], \mathtt{Z})$$
$$\downarrow$$
$$\mathtt{exec}(\mathtt{st}(\mathtt{0}, [], [\mathtt{X}, \mathtt{Y}, \mathtt{0}]), \mathtt{S_f})$$
$$\downarrow$$
$$\mathtt{exec}(\mathtt{st}(\mathtt{1}, [\mathtt{Y}], [\mathtt{X}, \mathtt{Y}, \mathtt{0}]), \mathtt{S_f})$$

Decompiled code

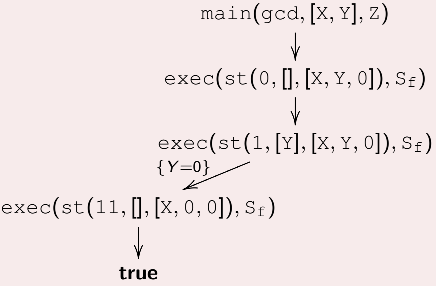
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees

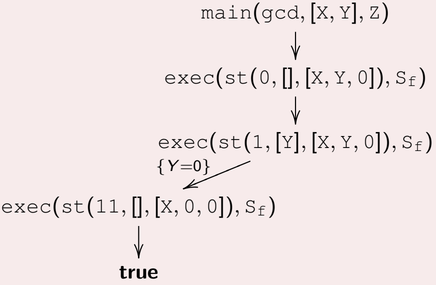


Decompiled code

Example 1: Source code

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

Unfolding trees



$\mathcal{L}_{bc}$-bytecode

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

Decompiled code

```
main(gcd,[X,0],X).
```

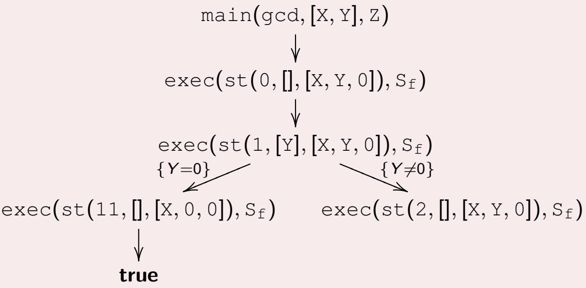
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
```

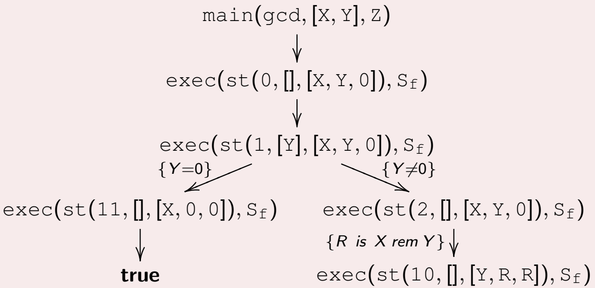
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
```

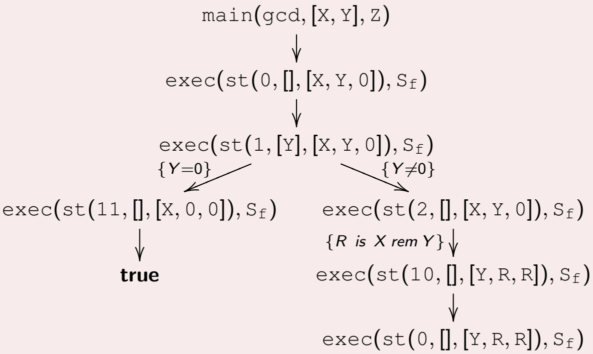
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
```

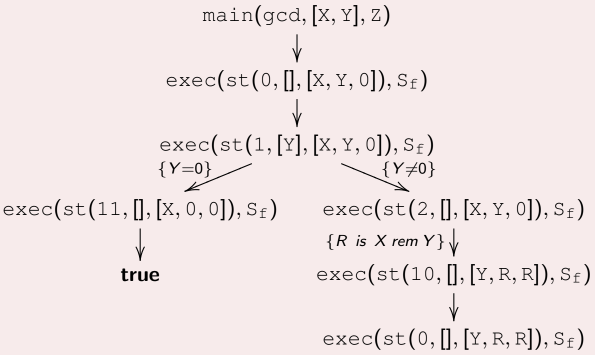
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
main(gcd,[X,Y],Z) :- Y \= 0,
    R is X rem Y, exec_1(Y,R,Z).
```

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Unfolding trees

$$\texttt{exec}(\texttt{st}(0, [], [\texttt{Y}, \texttt{R}, \texttt{R}]), \texttt{S}_\texttt{f})$$

Decompiled code

```
main(gcd,[X,0],X).
main(gcd,[X,Y],Z) :- Y \= 0,
   R is X rem Y, exec_1(Y,R,Z).
```

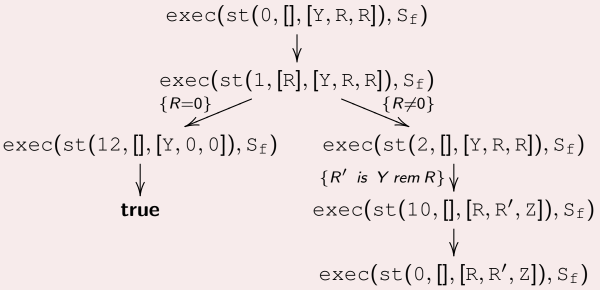
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
main(gcd,[X,Y],Z) :- Y \= 0,
    R is X rem Y, exec_1(Y,R,Z).
```

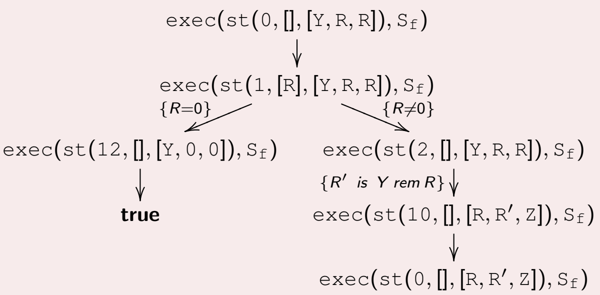
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
main(gcd,[X,Y],Z) :- Y \= 0,
    R is X rem Y, exec_1(Y,R,Z).

exec_1(Y,0,Y).
exec_1(Y,R,Z) :- R \= 0,
    R' is Y rem R, exec_1(R,R',Z).
```

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Unfolding trees

$\mathtt{main}(\mathtt{gcd}, [\mathtt{X}, \mathtt{Y}], \mathtt{Z})$

Decompiled code

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Unfolding trees

$$\mathtt{main}(\mathtt{gcd}, [\mathtt{X}, \mathtt{Y}], \mathtt{Z})$$

$$\downarrow$$

$$\mathtt{exec}(\mathtt{st}(0, [], [\mathtt{X}, \mathtt{Y}, 0]), \mathtt{S_f})$$

Decompiled code

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees

$$\mathtt{main}(\mathtt{gcd}, [\mathtt{X}, \mathtt{Y}], \mathtt{Z})$$
$$\downarrow$$
$$\mathtt{exec}(\mathtt{st}(0, [], [\mathtt{X}, \mathtt{Y}, 0]), \mathtt{S_f})$$
$$\downarrow$$
$$\mathtt{exec}(\mathtt{st}(1, [\mathtt{Y}], [\mathtt{X}, \mathtt{Y}, 0]), \mathtt{S_f})$$

Decompiled code

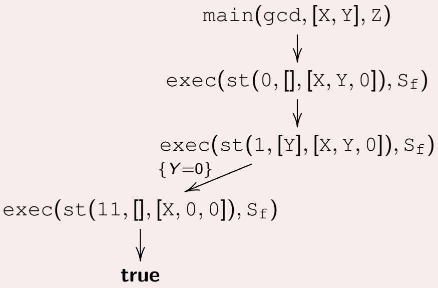
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees



Decompiled code

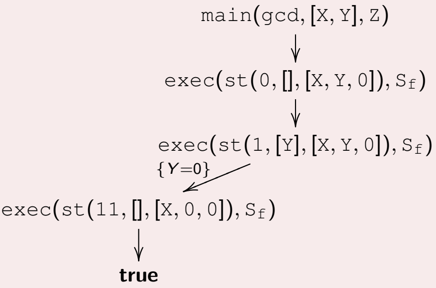
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
```

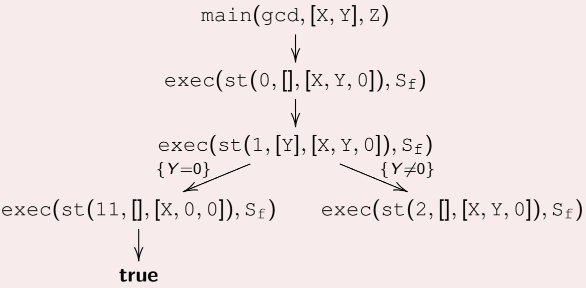
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
```

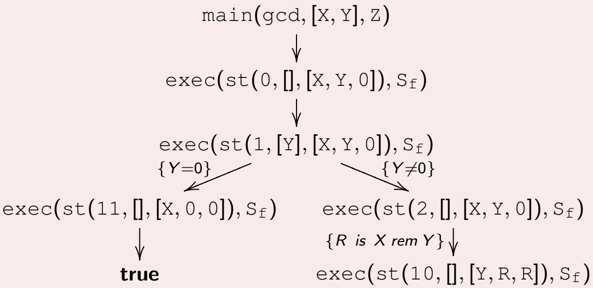
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
```

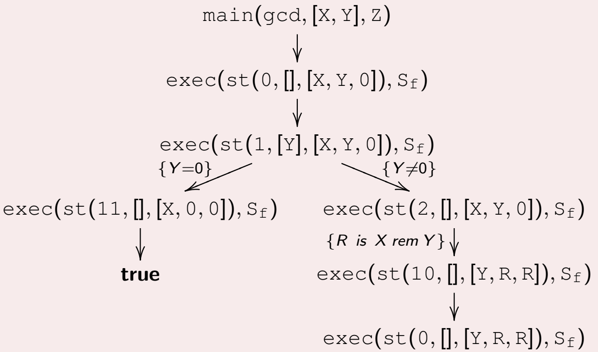
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
```

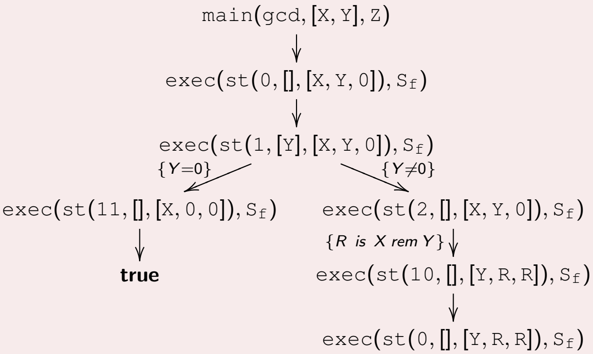
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).
main(gcd,[X,Y],Z) :- Y \= 0,
    R is X rem Y, exec_1(Y,R,Z).
```

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

```
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:return
```

Unfolding trees

$$\mathtt{exec}(\mathtt{st}(0, [], [\mathtt{Y}, \mathtt{R}, \mathtt{R}]), \mathtt{S}_\mathtt{f})$$

Decompiled code

```
main(gcd,[X,0],X).
main(gcd,[X,Y],Z) :- Y \= 0,
    R is X rem Y, exec_1(Y,R,Z).
```

Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees

$\mathtt{exec}(\mathtt{st}(0, [], [\mathtt{Y}, \mathtt{R}, \mathtt{R}]), \mathtt{S_f})$

↓

$\mathtt{exec}(\mathtt{st}(1, [\mathtt{R}], [\mathtt{Y}, \mathtt{R}, \mathtt{R}]), \mathtt{S_f})$

$\{R{=}0\}$ ↙ ↘ $\{R{\neq}0\}$

$\mathtt{exec}(\mathtt{st}(12, [], [\mathtt{Y}, 0, 0]), \mathtt{S_f})$ ↓ **true**

$\mathtt{exec}(\mathtt{st}(2, [], [\mathtt{Y}, \mathtt{R}, \mathtt{R}]), \mathtt{S_f})$

$\{R' \text{ is } Y \text{ rem } R\}$ ↓

$\mathtt{exec}(\mathtt{st}(10, [], [\mathtt{R}, \mathtt{R'}, \mathtt{Z}]), \mathtt{S_f})$

↓

$\mathtt{exec}(\mathtt{st}(0, [], [\mathtt{R}, \mathtt{R'}, \mathtt{Z}]), \mathtt{S_f})$

Decompiled code

```
main(gcd,[X,0],X).
main(gcd,[X,Y],Z) :- Y \= 0,
    R is X rem Y, exec_1(Y,R,Z).
```

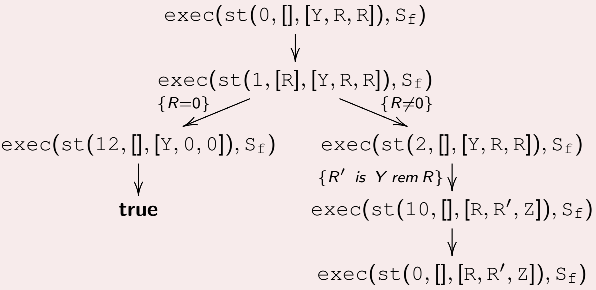
Example 1: Source code

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

$\mathcal{L}_{bc}$-bytecode

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

Unfolding trees



Decompiled code

```
main(gcd,[X,0],X).                exec_1(Y,0,Y).
main(gcd,[X,Y],Z) :- Y \= 0,      exec_1(Y,R,Z) :- R \= 0,
   R is X rem Y, exec_1(Y,R,Z).      R′ is Y rem R, exec_1(R,R′,Z).
```