Modular Interpretive Decompilation of Low-Level Code by Partial Evaluation

Elvira Albert
joint work with
Miguel Gómez-Zamalloa and Germán Puebla

(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.

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

Motivation

Low-level code $\Rightarrow$ 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

\[ p(\text{in1}, \text{in2}) = \text{output} \]

\[
\begin{align*}
\text{static data (in1)} & \\
\text{program } p & \quad \text{partial evaluator(mix)} \\
\text{dynamic data (in2)} & \\
\text{specialized program } p_{\text{in1}} & \quad \text{output}
\end{align*}
\]

\[
[p] \left[ \text{in1, in2} \right] = \left[ \left[ \left[ \text{mix} \right] \left[ p, \text{in1} \right] \right] \right] \left[ \text{in2} \right]
\]
First Futamura Projection

Partial Evaluation and the Interpretive Approach

\[ p(in1,in2) = \text{output} \]

\[
\begin{align*}
\text{bytecode} & \rightarrow \text{interp}(LP) \\
\text{dynamic data (in2)} & \rightarrow \text{specialized program} \ p_{in1} \\
\text{static data (in1)} & \rightarrow \text{partial evaluator (mix)} \\
\text{output} & = \text{programs}
\end{align*}
\]

\[ [[bc\_interp]] [in1,in2] = [[[mix]] [bc\_interp,in1]] [in2] \]
First Futamura Projection

Partial Evaluation and the Interpretive Approach

\[ p(in1, in2) = output \]

\[
[[bc\_interp]] [in1, in2] = [[[[mix]] [bc\_interp, in1]]] [in2]
\]
First Futamura Projection
Partial Evaluation and the Interpretive Approach

\[ p(in1, in2) = \text{output} \]

The diagram illustrates the process:

1. **Input Args (in2)**
2. **Specialized Program** \( p_{in1} \)
3. **Partial Evaluator** (mix)
4. **Bytecode Interpreter** \( \text{interp}(LP) \)
5. **Bytecode Program** \( (in1) \)

\[ [[bc\_interp]][in1, in2] = [[[mix]][bc\_interp, in1]][in2] \]
First Futamura Projection

Partial Evaluation and the Interpretive Approach

\[ p(in1, in2) = output \]

![Diagram showing the process of partial evaluation and the interpretive approach](image)

\[ [[bc\_interp][in1, in2]] = [[[mix][bc\_interp, in1]][in2]] \]
Example 1: Source code

```c
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

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

bytecode

```
0: load(1)   7: store(0)
1: if0eq(11) 8: load(0)
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)
```
Example 1: Source code

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

bytecode interpreter

```prolog
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) :-
    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, _, _),
                         next(PC, PC2),
                         localVar_update(LV, X, I, LV2),
                         S2 = st(PC2, S, LV2).
```

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
```

Decompiled code

```prolog
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

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

bytecode interpreter

```prolog
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,LV2),
  S2 = st(PC2,S,LV2)).
```

Decompiled code

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

Elvira Albert (UCM)  Interprettive Decomp. of Low-Level Code  Beijing, September 2008
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
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 ⇒ 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.
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.
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 ⇒ 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.
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 ⇒ Big-step semantics 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.
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.
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 ⇒ Big-step semantics 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.
- 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 \(\infty\) gains).
- Future work: Special handling for the heap, exploit instrumented decompilation, improve efficiency, applications, etc.
Experimental Evaluation (JOlden benchmarks suite)

![Graph showing the comparison between modular and block-modular approaches in terms of time per number of instructions.]
Intraprocedural Decompilation

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

  $$\text{Inst ::= push(x) | load(v) | store(v) | add | sub | mul | div | rem | neg | if \diamond \!(pc) | if0 \diamond \!(pc) | goto(pc) | return}$$

- State $\equiv \langle PC, \text{OpStack}, \text{LocalVars} \rangle$

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

```prolog
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).
```

```prolog
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

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

$L_{bc}$-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)
```
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
```

$L_{bc}$-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
```

Decompiled code
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
   ↓
exec(st(0, [], [X, Y, 0]), S_f)
```

$L_{bc}$-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
```

Decompiled code
Example 1: Source code

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

Lbc-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
```

Decompiled code

```
main(gcd, [X, Y], Z)
  ↓
exec(st(0, [], [X, Y, 0]), Sf)
  ↓
exec(st(1, [Y], [X, Y, 0]), Sf)
exec(st(2, [Y], [X, Y, 0]), Sf)
exec(st(3, [], [X, 0, 0]), Sf)
exec(st(4, [X], [X, Y, 0]), Sf)
```

Unfolding trees
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
  \downarrow
exec(st(0, [], [X, Y, 0]), S_f)
  \downarrow
exec(st(1, [Y], [X, Y, 0]), S_f)
  \{Y=0\}
exec(st(11, [], [X, 0, 0]), S_f)
  \downarrow
true
```

$L_{bc}$-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)
```
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
    \downarrow
    exec(st(0, [], [X, Y, 0]), S_f)
    \downarrow
    exec(st(1, [Y], [X, Y, 0]), S_f)
    \{ Y=0 \}
    exec(st(11, [], [X, 0, 0]), S_f)
    \downarrow
    true
```

$L_{bc}$-bytecode

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>load(1)</td>
<td>0</td>
</tr>
<tr>
<td>if0eq(11)</td>
<td>1</td>
</tr>
<tr>
<td>load(0)</td>
<td>2</td>
</tr>
<tr>
<td>load(1)</td>
<td>3</td>
</tr>
<tr>
<td>rem</td>
<td>4</td>
</tr>
<tr>
<td>store(2)</td>
<td>5</td>
</tr>
<tr>
<td>load(1)</td>
<td>6</td>
</tr>
<tr>
<td>store(0)</td>
<td>7</td>
</tr>
<tr>
<td>load(2)</td>
<td>8</td>
</tr>
<tr>
<td>store(1)</td>
<td>9</td>
</tr>
<tr>
<td>goto(0)</td>
<td>10</td>
</tr>
<tr>
<td>load(0)</td>
<td>11</td>
</tr>
<tr>
<td>return</td>
<td>12</td>
</tr>
</tbody>
</table>
```

Decompiled code

```
main(gcd, [X, 0], X).
```
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
    ↓
  exec(st(0, [], [X, Y, 0]), S_f)
    ↓
  exec(st(1, [Y], [X, Y, 0]), S_f)
      {y=0}→
      {y≠0}→
exec(st(11, [], [X, Y, 0]), S_f)
exec(st(2, [], [X, Y, 0]), S_f)
    ↓
true
```

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

Decompiled code

```
main(gcd, [X, 0], X).
```
Example 1: Source code

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

Unfolding trees

```plaintext
main(gcd, [X, Y], Z) ↓
  exec(st(0, [], [X, Y, 0]), S_f) ↓
  exec(st(1, [Y], [X, Y, 0]), S_f) ↓
{Y=0} exec(st(11, [], [X, 0, 0]), S_f) << true << exec(st(2, [], [X, Y, 0]), S_f) {Y≠0} exec(st(10, [], [Y, R, R]), S_f)
```

$L_{bc}$-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
```

Decompiled code

```
main(gcd, [X, 0], X).
```
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
  ↓
exec(st(0, [], [X, Y, 0]), S_f)
  ↓
exec(st(1, [Y], [X, Y, 0]), S_f)
  {Y=0}
  {Y≠0}
  ↑
exec(st(11, [], [X, 0, 0]), S_f)
exec(st(2, [], [X, 0, 0]), S_f)
  \{R is X rem Y\}
  ↑
exec(st(10, [], [Y, R, R]), S_f)
  ↑
exec(st(0, [], [Y, R, R]), S_f)
```

Decompiled code

```
main(gcd, [X, 0], X).
```
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
  ↓
exec(st(0, [], [X, Y, 0]), S_f)
  ↓
exec(st(1, [Y], [X, Y, 0]), S_f)
{ Y=0 }  { Y≠0 }
exec(st(11, [], [X, 0, 0]), S_f)
  ↓
exec(st(10, [], [Y, R, R]), S_f)
  ↓
exec(st(0, [], [Y, R, R]), S_f)
```

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
```

Decompiled code

```
main(gcd, [X, 0], X).
main(gcd, [X, Y], Z) :- Y \= 0,
  R is X rem Y, exec1(Y, R, Z).
```
Example 1: Source code

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

Unfolding trees

```prolog
exec(st(0, [], [Y, R, R]), S_f)
```

$L_{bc}$-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
```

Decompiled code

```
main(gcd, [X, 0], X).
main(gcd, [X, Y], Z) :- Y \= 0,
                 R is X \rem Y, exec1(Y, R, Z).
```
Intraprocedural Decompilation

Example 1: Source code

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

Unfolding trees

```
eval(st(0, [], [Y, R, R]), S_f)
  └─ eval(st(1, [R], [Y, R, R]), S_f)
      └─ {R=0}
          └─ eval(st(12, [], [Y, 0, 0]), S_f)
              └─ true
                  └─ eval(st(2, [], [Y, R, R]), S_f)
                      └─ {R ≠ 0}
                          └─ eval(st(10, [], [R, R', Z]), S_f)
                              └─ eval(st(0, [], [R, R', Z]), S_f)
```

$L_{bc}$-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)
```

Decompiled code

```
main(gcd, [X, 0], X).
main(gcd, [X, Y], Z) :- Y \= 0,
  R is X \ modulo \ Y, exec1(Y, R, Z).
```
Example 1: Source code

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

Unfolding trees

```
exec(st(0, [], [Y, R, R]), S_f)
  
exec(st(1, [R], [Y, R, R]), S_f)
  {R=0}
  {R≠0}
  
exec(st(12, [], [Y, 0, 0]), S_f)
exec(st(2, [], [Y, R, R]), S_f)
  {R' is Y rem R}
  
true
exec(st(10, [], [R, R', Z]), S_f)
  
exec(st(0, [], [R, R', Z]), S_f)
```

$L_{bc}$-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
```

 Decompiled code

```
main(gcd, [X, 0], X).
main(gcd, [X, Y], Z) :- Y \= 0, exec_1(Y, 0, Y).
main(gcd, [X, Y], Z) :- R \= 0,
  R' is X rem Y, exec_1(Y, R, Z),
  R' is Y rem R, exec_1(R, R', Z).
```
Example 1: Source code

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

$L_{bc}$-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)
```
**Example 1: Source code**

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

**Unfolding trees**

```
main(gcd, [X, Y], Z)
```

**\( L_{bc} \)-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**
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
  ↓
  exec(st(0, [], [X, Y, 0]), S_f)
```

$L_{bc}$-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
```

Decompiled code
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
  ↓
exec(st(0, [], [X, Y, 0]), S_f)
  ↓
exec(st(1, [Y], [X, Y, 0]), S_f)
```

$L_{bc}$-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

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

Unfolding trees

```
main(gcd, [X, Y], Z)
  ↓
exec(st(0, [], [X, Y, 0]), S_f)
  ↓
exec(st(1, [Y], [X, Y, 0]), S_f)
  {Y=0}
exec(st(11, [], [X, 0, 0]), S_f)
true
```

\[ \mathcal{L}_{bc}\text{-bytecode} \]

```
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
```

Decompiled code

```
```
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
  ↓
  exec(st(0, [], [X, Y, 0]), S_f)
  ↓
  exec(st(1, [Y], [X, Y, 0]), S_f)
  {Y=0}
  exec(st(11, [], [X, 0, 0]), S_f)
  ↓
  true
```

\( L_{bc} \)-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
```

Decompiled code

```
main(gcd, [X, 0], X).
```
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
  ↓
exec(st(0, [], [X, Y, 0]), Sf)
  ↓
exec(st(1, [Y], [X, Y, 0]), Sf)
  {Y=0}
  {Y≠0}
exec(st(11, [], [X, 0, 0]), Sf)
exec(st(2, [], [X, Y, 0]), Sf)
```

Decompiled code

```
main(gcd, [X, 0], X).
```
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
  ↓
exec(st(0, [], [X, Y, 0]), S_f)
  ↓
exec(st(1, [Y], [X, Y, 0]), S_f)
  ↓
exec(st(11, [], [X, 0, 0]), S_f)
  ↓
true
exec(st(2, [], [X, Y, 0]), S_f)
  ↓
  {Y=0}
exec(st(10, [], [Y, R, R]), S_f)
  ↓
exec(st(0, [], [Y, R, R]), S_f)
```

$L_{bc}$-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
```

Decompiled code

```
main(gcd, [X, 0], X).
```
Example 1: Source code

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

Unfolding trees

```
main(gcd, [X, Y], Z)
  ↓
exec(st(0, [], [X, Y, 0]), S_f)
  ↓
exec(st(1, [Y], [X, Y, 0]), S_f)
  {Y=0}
  {Y≠0}
exec(st(11, [], [X, 0, 0]), S_f)
  exec(st(2, [], [X, Y, 0]), S_f)
  {R is X % Y}
  exec(st(10, [], [Y, R, R]), S_f)
  exec(st(0, [], [Y, R, R]), S_f)
```

$L_{bc}$-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
```

Decompiled code

```
main(gcd, [X, 0], X).
```
Example 1: Source code

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

```
Unfolding trees

main(gcd, [X, Y], Z)
  exec(st(0, [], [X, Y, 0]), S_f)
  exec(st(1, [Y], [X, Y, 0]), S_f)
    {Y=0}
    {Y≠0}
exec(st(11, [], [X, 0, 0]), S_f)
exec(st(2, [], [X, Y, 0]), S_f)
  {R is X rem Y}
exec(st(10, [], [Y, R, R]), S_f)
  exec(st(0, [], [Y, R, R]), S_f)
```

```
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
```

Decompiled code

```prolog
main(gcd, [X, 0], X).
main(gcd, [X, Y], Z) :- Y \= 0,
    R is X rem Y, exec1(Y, R, Z).
```
Example 1: Source code

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

Unfolding trees

```
exec(st(0, [], [Y, R, R]), S_f)
```

$L_{bc}$-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).
main(gcd, [X, Y], Z) :- Y \= 0,
    R is X \mod Y, exec1(Y, R, Z).
```
Example 1: Source code

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

Unfolding trees

```
exec(st(0, [], [Y, R, R]), S_f)
  
exec(st(1, [R], [Y, R, R]), S_f)
  \{R=0\}
  
exec(st(12, [], [Y, 0, 0]), S_f)
  \{R\neq 0\}
  
exec(st(2, [], [Y, R, R]), S_f)
  \{R' \text{ is } Y \text{ rem } R\}
  
exec(st(10, [], [R, R', Z]), S_f)
  \{R' \text{ is } Y \text{ rem } R\}
  
exec(st(0, [], [R, R', Z]), S_f)
```

$\mathcal{L}_{bc}$-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)
```

Decompiled code

```
main(gcd, [X, 0], X).
main(gcd, [X, Y], Z) :- Y \neq 0,
    R is X rem Y, exec1(Y, R, Z).
```
Example 1: Source code

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

Unfolding trees

```
exec(st(0, [], [Y, R, R]), S_f)
  \downarrow
exec(st(1, [R], [Y, R, R]), S_f)
  \downarrow
  \{R=0\}
exec(st(12, [], [Y, 0, 0]), S_f)
exec(st(2, [], [Y, R, R]), S_f)
  \downarrow
  \{R\neq0\}
  \uparrow
true
exec(st(10, [], [R, R', Z]), S_f)
exec(st(0, [], [R, R', Z]), S_f)
```

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)
```

Decompiled code

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