Towards Deductive Compilation: Implementing a Partial Evaluator Via a Software Verification Tool

Reiner Hähnle
(joint work with Richard Bubel and Ran Ji)

Chalmers University of Technology
Department of Computer Science and Engineering

10 March 2011

Seminar "Deduction at Scale" Schloss Ringberg, March 2011

Introduction

Starting Point

Program verification tool (KeY) based on

- Dynamic logic for Java source code
- First-order theorem proving
- Symbolic execution
- Invariant reasoning

Introduction

Starting Point

Program verification tool (KeY) based on

- Dynamic logic for Java source code
- First-order theorem proving
- Symbolic execution
- Invariant reasoning

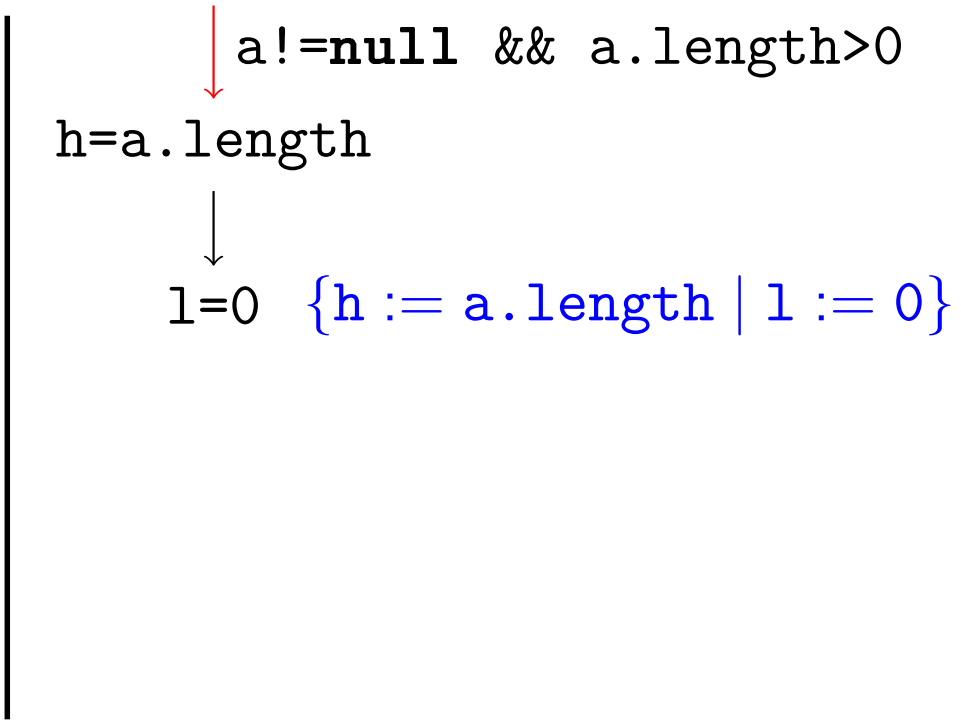
Constructing a specialized program from a verification proof attempt

Overview of Symbolic Execution

```
{a!=null && a.length>0}
h = a.length; ⇐ pc
l = 0;
while (a[(h-l)/2]>0) {
  body
}
rest
```

Overview of Symbolic Execution

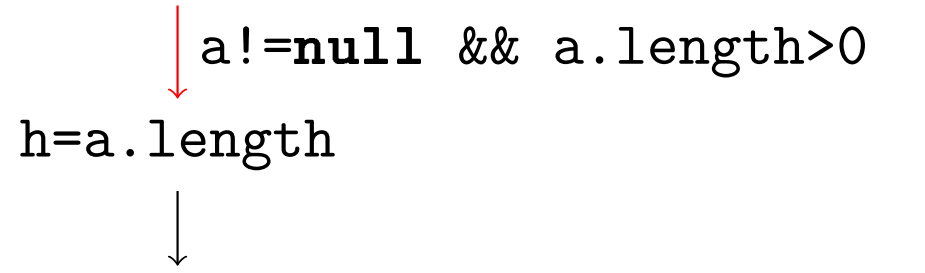
```
{a!=null && a.length>0}
while (a[(h-1)/2]>0) {⇐
  body
}
rest
```



1. Precondition is path condition in SE tree; nodes have symbolic state

Overview of Symbolic Execution

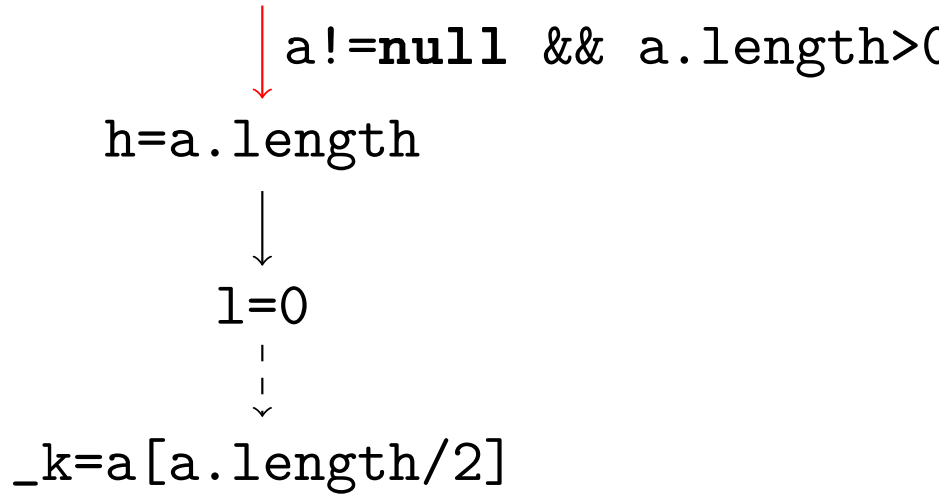
```
{a!=null && a.length>0}
int _i = a.length-0; ⇐
int _j = _i/2;
int _k = a[_j];
boolean _g = (_k>0);
while (_g) {
  body
}
rest
```



1. Precondition is path condition in SE tree; nodes have symbolic state
2. Local program transformation: simple, side-effect free expressions

Overview of Symbolic Execution

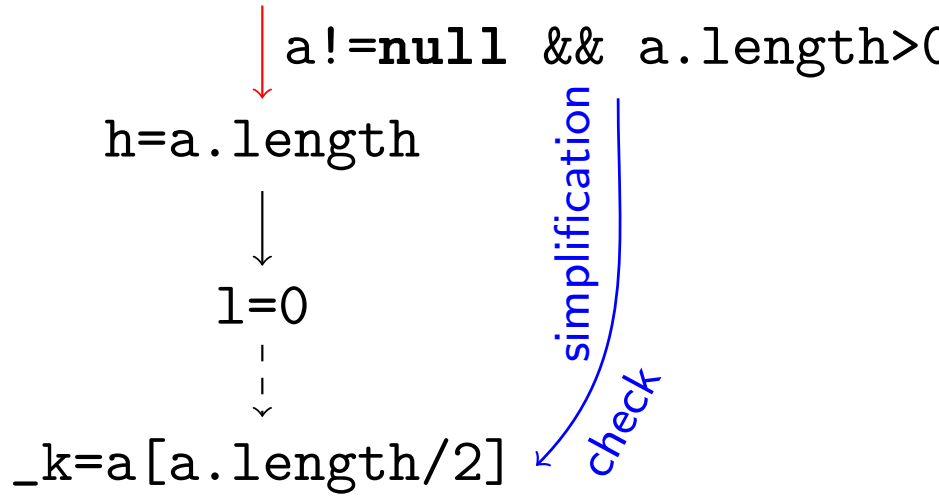
```
{a!=null && a.length>0}
boolean _g = (_k>0); ⇐
while (_g) {
  body
}
rest
```



1. Precondition is path condition in SE tree; nodes have symbolic state
2. Local program transformation: simple, side-effect free expressions

Overview of Symbolic Execution

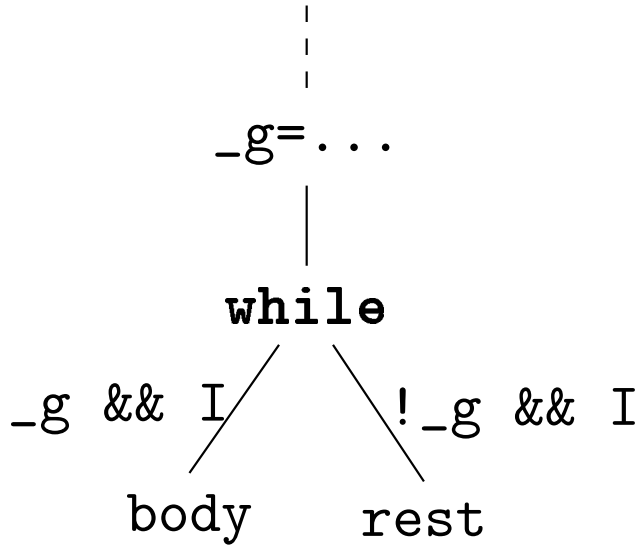
```
{a!=null && a.length>0}
boolean _g = (_k>0); ⇐
while (_g) {
  body
}
rest
```



1. Precondition is path condition in SE tree; nodes have symbolic state
2. Local program transformation: simple, side-effect free expressions
3. First-order reasoning required for simplification, checking bounds

Overview of Symbolic Execution

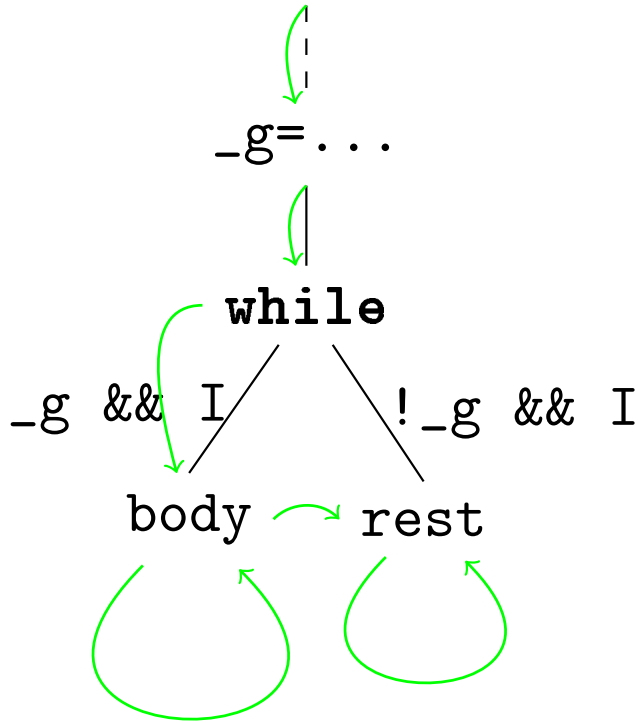
```
{a!=null && a.length>0}
while (_g) {
  body
}
rest
```



1. Precondition is path condition in SE tree; nodes have symbolic state
2. Local program transformation: simple, side-effect free expressions
3. First-order reasoning required for simplification, checking bounds
4. Execute loop under suitable invariant

Overview of Symbolic Execution

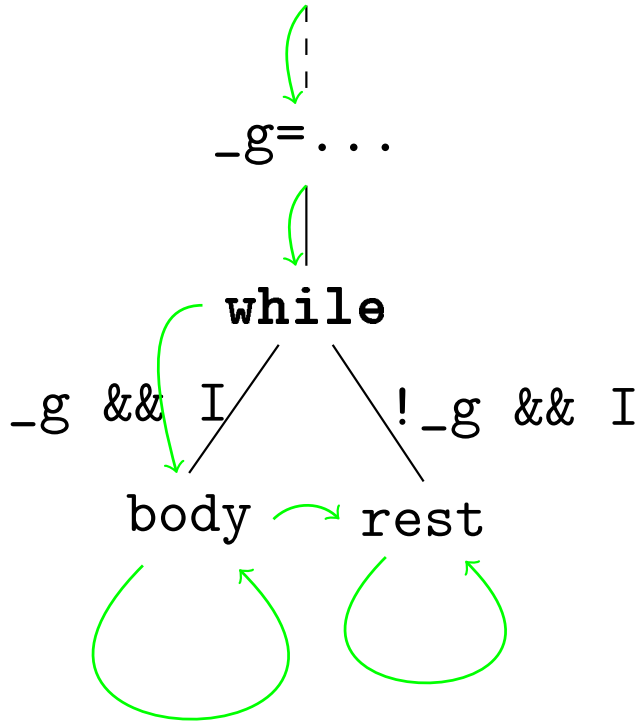
```
{a!=null && a.length>0}
while (_g) {
  body
}
rest
```



1. Precondition is path condition in SE tree; nodes have symbolic state
2. Local program transformation: simple, side-effect free expressions
3. First-order reasoning required for simplification, checking bounds
4. Execute loop under suitable invariant
5. View SE as depth left first AST traversal (inlined first argument of ;)

Overview of Symbolic Execution

```
{a!=null && a.length>0}
while (_g) {
  body
}
rest
```

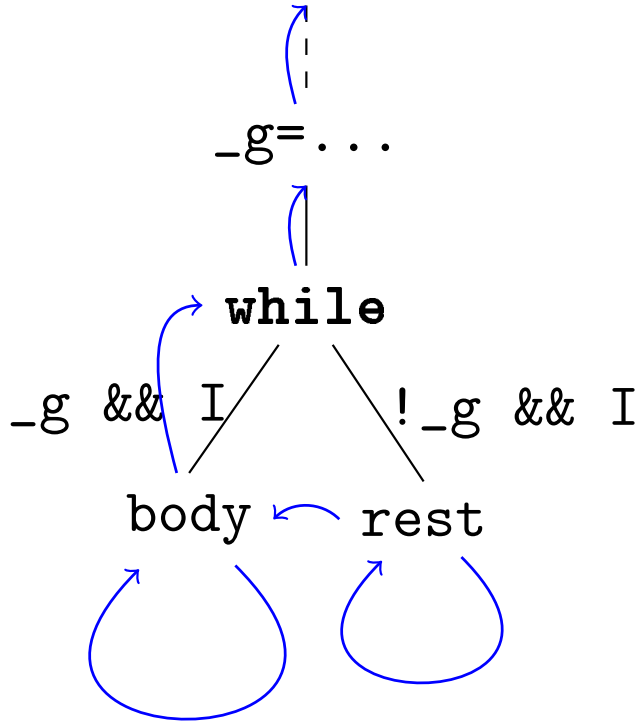


Observations

1. Transformation of complex assignments, symbolic state simplification: single static assignment (SSA) form easily obtainable
2. If strongest postcondition not needed, can use **true** as invariant

Overview of Symbolic Execution

```
{a!=null && a.length>0}
while (_g) {
  body
}
rest
```



Observations

1. Transformation of complex assignments, symbolic state simplification: single static assignment (SSA) form easily obtainable
2. If strongest postcondition not needed, can use **true** as invariant
3. May synthesize specialized program by bottom-up AST traversal:

   Backward Analysis used variables, etc.
   Program Specialisation dead code elimination, condition evaluation

Program Logic Calculus

Calculus

$$\text{ruleName}\ \frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[p_1] \quad \ldots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[p_n]}{\Gamma \Longrightarrow \mathcal{U}[p]}$$

Notation:

- Γ: path conditions (set of formulas)
- $\mathcal{U}$: update (information from the program has been excuted)
- p: Java program (program to be excuted)

Program Logic Calculus

Calculus

$$\text{ruleName}\ \frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[p_1] \quad \ldots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[p_n]}{\Gamma \Longrightarrow \mathcal{U}[p]}$$

Notation:

- Γ: path conditions (set of formulas)
- $\mathcal{U}$: update (information from the program has been excuted)
- p: Java program (program to be excuted)
- rule application from bottom-to-top
- postcondition ignored

Interleaving Symbolic Execution and Partial Evaluation

Proof-Search Space Reduction can be achieved by adding calculus rules performing (or invoking) a basic partial evaluator (FMCO 2009):

- constant propagation
- constant expression evaluation
- dead-code elimination

Interleaving Symbolic Execution and Partial Evaluation

Proof-Search Space Reduction can be achieved by adding calculus rules performing (or invoking) a basic partial evaluator (FMCO 2009):

- constant propagation
- constant expression evaluation
- dead-code elimination

One reason why this is a good idea:

Proof branching during symbolic execution creates new static input values:

$$\frac{\mathcal{U}(\texttt{b}) \Longrightarrow \mathcal{U}[\texttt{ack=true;r}] \quad \mathcal{U}(\neg\texttt{b}) \Longrightarrow \mathcal{U}[\texttt{ack=false;r}]}{\Longrightarrow \mathcal{U}[\texttt{if (b) \{ack=true;\} else \{ack=false;\} r}]}$$

Interleaving Symbolic Execution and Partial Evaluation

Proof-Search Space Reduction can be achieved by adding calculus rules performing (or invoking) a basic partial evaluator (FMCO 2009):

- constant propagation
- constant expression evaluation
- dead-code elimination

One reason why this is a good idea:

Proof branching during symbolic execution creates new static input values:

$$\frac{\mathcal{U}(\texttt{b}) \Longrightarrow \mathcal{U}[\texttt{ack=true;r}] \quad \mathcal{U}(\neg\texttt{b}) \Longrightarrow \mathcal{U}[\texttt{ack=false;r}]}{\Longrightarrow \mathcal{U}[\texttt{if (b) \{ack=true;\} else \{ack=false;\} r}]}$$

Can we extract a specialized program out of a verification proof?

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}]$$

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (Fwd)(Bk)$$

Sequent annotated with

$(Fwd)(Bk)$: program analysis and synthesis results from AST traversal

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (\mathit{Fwd})(\mathit{Bk})$$

Sequent annotated with

$(\mathit{Fwd})(\mathit{Bk})$: program analysis and synthesis results from AST traversal

- *Fwd*: program information maintained in forward analysis

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (\mathit{Fwd})(\mathit{Bk})$$

Sequent annotated with

$(\mathit{Fwd})(\mathit{Bk})$: program analysis and synthesis results from AST traversal

- *Fwd*: program information maintained in forward analysis
  - program variables potentially read in continuation of p

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (\mathit{Fwd})(\mathit{Bk})$$

Sequent annotated with

$(\mathit{Fwd})(\mathit{Bk})$: program analysis and synthesis results from AST traversal

- *Fwd*: program information maintained in forward analysis
  - program variables potentially read in continuation of p
- $\mathit{Bk} = \mathit{sp}, \mathit{use}$: program information synthesized in backward analysis

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (\mathit{Fwd})(\mathit{Bk})$$

Sequent annotated with

$(\mathit{Fwd})(\mathit{Bk})$: program analysis and synthesis results from AST traversal

- *Fwd*: program information maintained in forward analysis
  - program variables potentially read in continuation of p
- $\mathit{Bk} = \mathit{sp}, \mathit{use}$: program information synthesized in backward analysis
  - *sp*: generated specialized program of Java source program p
  - *use*: program variables used in p (or continuation of p)

Program Specialization

Extended Symbolic State Node

$$\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (Fwd)(Bk)$$

Sequent annotated with

$(Fwd)(Bk)$: program analysis and synthesis results from AST traversal

- *Fwd*: program information maintained in forward analysis
  - program variables potentially read in continuation of p
- $Bk = sp, use$: program information synthesized in backward analysis
  - *sp*: generated specialized program of Java source program p
  - *use*: program variables used in p (or continuation of p)
- In general, *Fwd* and *Bk* could contain other information
  - View as specific pre-/postconditions or constraint system

Program Generation Rules

$$\frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[\mathrm{p}_1] \mid (X_1)(sp_1, use_1) \quad \ldots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[\mathrm{p}_n] \mid (X_n)(sp_n, use_n)}{\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (X)(sp, use)}$$

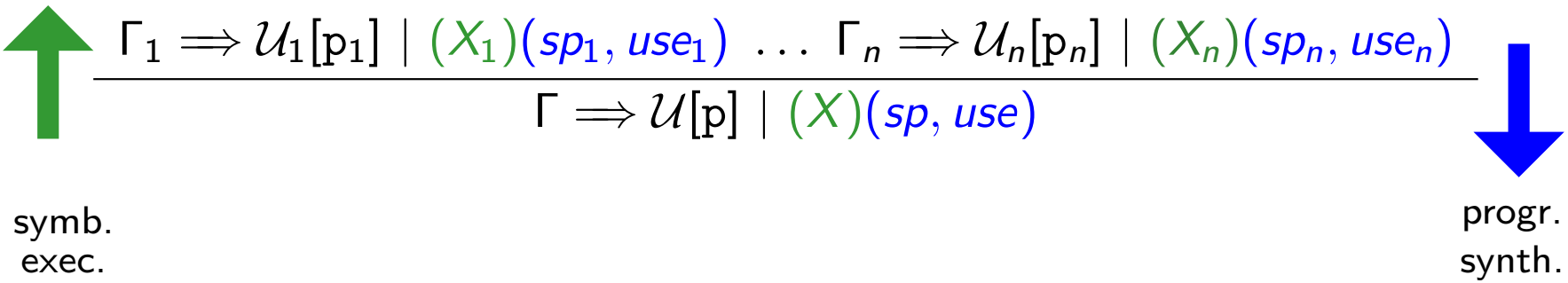
Program Generation Rules

$$\frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[\mathrm{p}_1] \mid (X_1)(sp_1, use_1) \quad \ldots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[\mathrm{p}_n] \mid (X_n)(sp_n, use_n)}{\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}] \mid (X)(sp, use)}$$

symb.
exec.

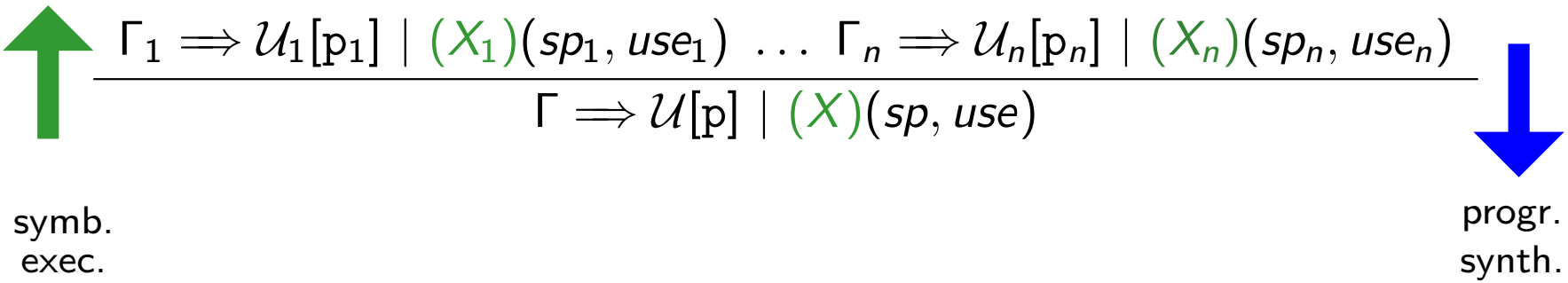
- Java source code executed

Program Generation Rules



- Java source code executed then specialized program synthesized

Program Generation Rules



- Java source code executed then specialized program synthesized
- Establishing rule correctness requires to prove bisimulation property of original and specialized program

Selected Program Generation Rules

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\mathtt{nop}, X)}$$

- 'initiates' backward program synthesis

Selected Program Generation Rules

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\mathtt{nop}, X)}$$

- 'initiates' backward program synthesis

Selected Program Generation Rules

$$\text{emptyBox}\quad \frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\mathtt{nop}, X)}$$

- 'initiates' backward program synthesis
- ensures variables X read in program continuation are in used variables set (e.g., return variable)

Selected Program Generation Rules

emptyBox
$$\frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\texttt{nop}, X)}$$

- 'initiates' backward program synthesis
- ensures variables X read in program continuation are in used variables set (e.g., return variable)

assignment
$$\frac{\Gamma \Longrightarrow \mathcal{U}\{l := r\}[\texttt{rest}] \mid (X)(\overline{rest}, use)}{\Gamma \Longrightarrow \mathcal{U}[\texttt{l = r;rest}] \mid (X) \begin{pmatrix} \texttt{l} = \texttt{r}; \overline{rest}, (use - \{l\} \cup locs(r)) & \text{if } l \in use \\ \overline{rest}, use & \text{otherwise} \end{pmatrix}}$$

Selected Program Generation Rules

$$\text{emptyBox}\quad \frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\mathtt{nop}, X)}$$

- 'initiates' backward program synthesis
- ensures variables X read in program continuation are in used variables set (e.g., return variable)

assignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{l := r\}[\mathtt{rest}] \mid (X)(\overline{rest}, use)}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{l = r;rest}] \mid (X)\begin{pmatrix} \mathtt{l} = \mathtt{r};\overline{rest}, (use - \{l\} \cup locs(r)) & \text{if } l \in use \\ \overline{rest}, use & \text{otherwise} \end{pmatrix}}$$

Selected Program Generation Rules

emptyBox
$$\frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\mathtt{nop}, X)}$$

- 'initiates' backward program synthesis
- ensures variables X read in program continuation are in used variables set (e.g., return variable)

assignment
$$\frac{\Gamma \Longrightarrow \mathcal{U}\{l := r\}[\mathtt{rest}] \mid (X)(\overline{rest}, use)}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{l = r;rest}] \mid (X) \begin{pmatrix} \mathtt{l} = \mathtt{r}; \overline{rest}, (use - \{l\} \cup locs(r)) & \text{if } l \in use \\ \overline{rest}, use & \text{otherwise} \end{pmatrix}}$$

- updates used variable set
- assignment to unused variable deleted

Selected Program Generation Rules

emptyBox
$$\frac{\Gamma \Longrightarrow \mathcal{U} \mid (X)(\_,\_)}{\Gamma \Longrightarrow \mathcal{U}[] \mid (X)(\mathtt{nop}, X)}$$

- 'initiates' backward program synthesis
- ensures variables X read in program continuation are in used variables set (e.g., return variable)

assignment
$$\frac{\Gamma \Longrightarrow \mathcal{U}\{l := r\}[\mathtt{rest}] \mid (X)(\overline{rest}, use)}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{l = r;rest}] \mid (X) \begin{pmatrix} \mathtt{l} = \mathtt{r}; \overline{rest}, (use - \{l\} \cup locs(r)) & \text{if } l \in use \\ \overline{rest}, use & \text{otherwise} \end{pmatrix}}$$

- updates used variable set
- assignment to unused variable deleted

Conditional Rule

conditional

$$\frac{\Gamma, \mathcal{U} b \Longrightarrow \mathcal{U}[\texttt{p;rest}] \mid (X)\ (\overline{p; \mathit{rest}}, \mathit{use}_{p;\mathit{rest}}) \qquad \Gamma, \mathcal{U}(\neg b) \Longrightarrow \mathcal{U}[\texttt{q;rest}] \mid (X)\ (\overline{q; \mathit{rest}}, \mathit{use}_{q;\mathit{rest}})}{\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{p\} else \{q\};rest}] \mid (X)\ \left(\begin{array}{l} \texttt{if (b) } \{\overline{\texttt{p;rest}}\} \\ \texttt{else } \{\overline{\texttt{q;rest}}\} \end{array} , (\mathit{use}_{p;\mathit{rest}} \cup \mathit{use}_{q;\mathit{rest}} \cup \mathit{locs}(b)) \right)}$$

Conditional Rule

conditional

$$\frac{\Gamma, \mathcal{U} b \Longrightarrow \mathcal{U}[\texttt{p;rest}] \mid (X)\ (\overline{p; \mathit{rest}}, \mathit{use}_{p;\mathit{rest}}) \qquad \Gamma, \mathcal{U}(\neg b) \Longrightarrow \mathcal{U}[\texttt{q;rest}] \mid (X)\ (\overline{q; \mathit{rest}}, \mathit{use}_{q;\mathit{rest}})}{\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{p\} else \{q\};rest}] \mid (X) \left(\begin{array}{l} \texttt{if (b) } \overline{\texttt{\{p;rest\}}} \\ \texttt{else } \overline{\texttt{\{q;rest\}}} \end{array} , (\mathit{use}_{p;\mathit{rest}} \cup \mathit{use}_{q;\mathit{rest}} \cup \mathit{locs}(b)) \right)}$$

Conditional Rule

conditional

$$\frac{\Gamma, \mathcal{U} b \Longrightarrow \mathcal{U}[\texttt{p;rest}] \mid (X)\ (\overline{p; \mathit{rest}}, \mathit{use}_{p;\mathit{rest}}) \quad \Gamma, \mathcal{U}(\neg b) \Longrightarrow \mathcal{U}[\texttt{q;rest}] \mid (X)\ (\overline{q; \mathit{rest}}, \mathit{use}_{q;\mathit{rest}})}{\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{p\} else \{q\};rest}] \mid (X) \left(\begin{array}{l} \texttt{if (b)}\ \overline{\{\texttt{p;rest}\}} \\ \texttt{else}\ \overline{\{\texttt{q;rest}\}} \end{array} , (\mathit{use}_{p;\mathit{rest}} \cup \mathit{use}_{q;\mathit{rest}} \cup \mathit{locs}(b)) \right)}$$

Conditional Rule

conditional

$$\frac{\begin{array}{c}\Gamma, \mathcal{U}b \Longrightarrow \mathcal{U}[\texttt{p;rest}] \mid (X)\ (\overline{p; \mathit{rest}}, \mathit{use}_{p;\mathit{rest}}) \\ \Gamma, \mathcal{U}(\neg b) \Longrightarrow \mathcal{U}[\texttt{q;rest}] \mid (X)\ (\overline{q; \mathit{rest}}, \mathit{use}_{q;\mathit{rest}})\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{p\} else \{q\};rest}] \\ \qquad \mid (X)\ \left(\begin{array}{l}\texttt{if (b) } \{\overline{\texttt{p;rest}}\} \\ \texttt{else } \{\overline{\texttt{q;rest}}\}\end{array}\ , (\mathit{use}_{p;\mathit{rest}} \cup \mathit{use}_{q;\mathit{rest}} \cup \mathit{locs}(b)) \right)\end{array}}$$

Generating Specialized Programs containing Loops

loopUnwind

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{ p; while (b) p\} rest}] \mid (X)\ (\ \overline{\texttt{if(b)\{p; while(b)p\} rest}}\ , \mathit{use})}{\Gamma \Longrightarrow \mathcal{U}[\texttt{while (b) \{p\}; rest}] \mid (X)\ (\ \overline{\texttt{if(b)\{p; while(b)p\} rest}}\ , \mathit{use})}$$

Generating Specialized Programs containing Loops

loopUnwind

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{ p; while (b) p\} rest}] \mid (X)\ (\ \overline{\texttt{if(b)\{p; while(b)p\} rest}}\ , \mathit{use})}{\Gamma \Longrightarrow \mathcal{U}[\texttt{while (b) \{p\}; rest}] \mid (X)\ (\ \overline{\texttt{if(b)\{p; while(b)p\} rest}}\ , \mathit{use})}$$

Generating Specialized Programs containing Loops

loopUnwind

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{ p; while (b) p\} rest}] \mid (X)\ (\ \overline{\texttt{if(b)\{p;while(b)p\} rest}}\ , \mathit{use})}{\Gamma \Longrightarrow \mathcal{U}[\texttt{while (b) \{p\}; rest}] \mid (X)\ (\ \overline{\texttt{if(b)\{p;while(b)p\} rest}}\ , \mathit{use})}$$

Generating Specialized Programs containing Loops

loopUnwind

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\texttt{if (b) \{ p; while (b) p\} rest}] \mid (X)\ (\ \overline{\texttt{if(b)\{p;while(b)p\} rest}}\ , \mathit{use})}{\Gamma \Longrightarrow \mathcal{U}[\texttt{while (b) \{p\}; rest}] \mid (X)\ (\ \overline{\texttt{if(b)\{p;while(b)p\} rest}}\ , \mathit{use})}$$

Loop Invariant Rule

loopInvariant

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv} \\ \Gamma, \mathcal{U}\mathcal{V}_a(\mathit{Inv} \wedge b) \Longrightarrow [\mathtt{p}]\mathit{Inv} \\ \Gamma, \mathcal{U}\mathcal{V}_a(\mathit{Inv} \wedge \neg b) \Longrightarrow [\mathtt{rest}]\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

Loop Invariant Rule

$$\text{loopInvariant}\quad \frac{\Gamma \Longrightarrow \mathcal{U} Inv \quad \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge b) \Longrightarrow [\texttt{p}]Inv \quad \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge \neg b) \Longrightarrow [\texttt{rest}]}{\Gamma \Longrightarrow \mathcal{U}[\texttt{while (b) \{p\} rest}]}$$

Since we are not interested in proving correctness, use **`true`** as invariant!

Loop Invariant Rule

loopInvariant

$$\frac{\Gamma \Longrightarrow \mathcal{U} Inv \quad \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge b) \Longrightarrow [\mathtt{p}]Inv \quad \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge \neg b) \Longrightarrow [\mathtt{rest}]}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

Since we are not interested in proving correctness, use **`true`** as invariant!

loopInvariantTrue

$$\frac{\Gamma, \mathcal{U}\mathcal{V}_a b \Longrightarrow [\mathtt{p}] \mid (X \cup use_{rest} \cup locs(b))\ (\overline{p}, use_{body}) \quad \Gamma, \mathcal{U}\mathcal{V}_a \neg b \Longrightarrow [\mathtt{rest}] \mid (X)\ (\overline{rest}, use_{rest})}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}] \mid (X)\ (\ \mathtt{while(b)\{\overline{p}\}\ \overline{rest}}\ , (use_{body} \cup use_{rest} \cup locs(b)))}$$

Loop Invariant Rule

loopInvariant

$$\frac{\Gamma \Longrightarrow \mathcal{U} Inv \quad \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge b) \Longrightarrow [\mathtt{p}]Inv \quad \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge \neg b) \Longrightarrow [\mathtt{rest}]}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

Since we are not interested in proving correctness, use **`true`** as invariant!

loopInvariantTrue

$$\frac{\Gamma, \mathcal{U}\mathcal{V}_a b \Longrightarrow [\mathtt{p}] \mid (X \cup use_{rest} \cup locs(b))\ (\overline{p}, use_{body}) \quad \Gamma, \mathcal{U}\mathcal{V}_a \neg b \Longrightarrow [\mathtt{rest}] \mid (X)\ (\overline{rest}, use_{rest})}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}] \mid (X)\ (\ \mathtt{while(b)\{\overline{p}\}\ \overline{rest}}\ , (use_{body} \cup use_{rest} \cup locs(b)))}$$

Loop Invariant Rule

loopInvariant

$$\frac{\Gamma \Longrightarrow \mathcal{U} Inv \quad \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge b) \Longrightarrow [\mathtt{p}]Inv \quad \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge \neg b) \Longrightarrow [\mathtt{rest}]}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

Since we are not interested in proving correctness, use **`true`** as invariant!

loopInvariantTrue

$$\frac{\begin{array}{l}\Gamma, \mathcal{U}\mathcal{V}_a b \Longrightarrow [\mathtt{p}] \mid (X \cup use_{rest} \cup locs(b))\ (\overline{p}, use_{body}) \\ \Gamma, \mathcal{U}\mathcal{V}_a \neg b \Longrightarrow [\mathtt{rest}] \mid (X)\ (\overline{rest}, use_{rest})\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}] \\ \qquad \mid (X)\ \left(\ \mathtt{while(b)\{\overline{p}\}\ \overline{rest}}\ , (use_{body} \cup use_{rest} \cup locs(b))\right)\end{array}}$$

Loop Invariant Rule

loopInvariant

$$\frac{\Gamma \Longrightarrow \mathcal{U} Inv \quad \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge b) \Longrightarrow [\mathtt{p}]Inv \quad \Gamma, \mathcal{U}\mathcal{V}_a(Inv \wedge \neg b) \Longrightarrow [\mathtt{rest}]}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}]}$$

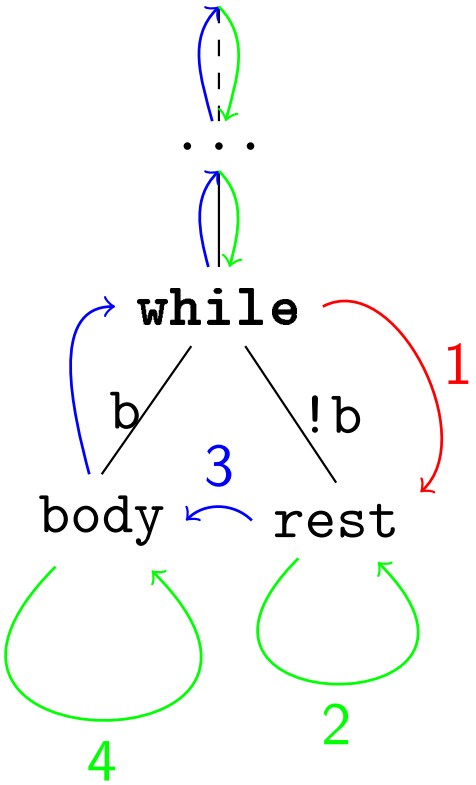
Since we are not interested in proving correctness, use **`true`** as invariant!

loopInvariantTrue

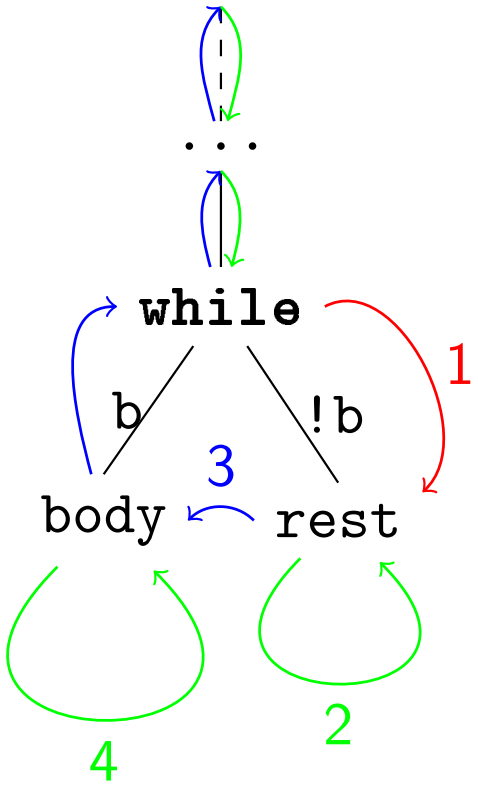
$$\frac{\begin{array}{l}\Gamma, \mathcal{U}\mathcal{V}_a b \Longrightarrow [\mathtt{p}] \mid (X \cup use_{rest} \cup locs(b))\ (\overline{p}, use_{body}) \\ \Gamma, \mathcal{U}\mathcal{V}_a \neg b \Longrightarrow [\mathtt{rest}] \mid (X)\ (\overline{rest}, use_{rest})\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}\ rest}] \\ \qquad \mid (X)\ \left(\ \mathtt{while(b)\{}\overline{\mathtt{p}}\mathtt{\}}\ \overline{\mathtt{rest}}\ , (use_{body} \cup use_{rest} \cup locs(b))\right)\end{array}}$$

In "preserves invariant" branch the program variables used in the continuation of the loop body must be reflected correctly

Work Flow of Synthesizing Loop

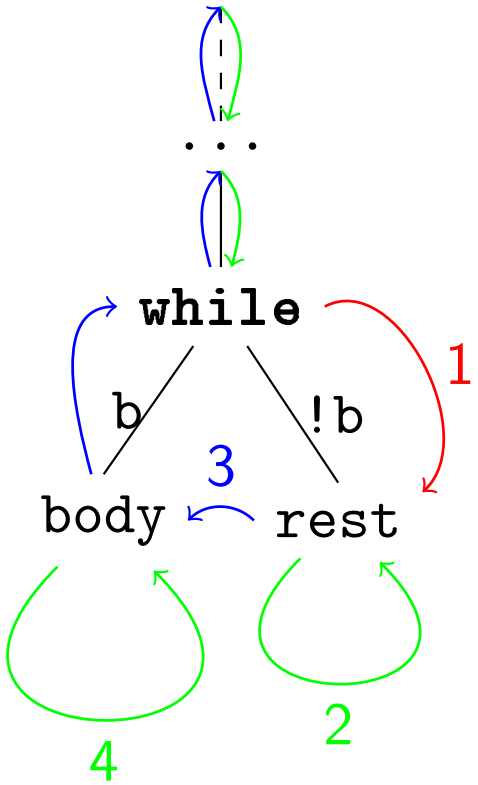


Work Flow of Synthesizing Loop



- Differs from traditional symbolic execution

Work Flow of Synthesizing Loop



- Differs from traditional symbolic execution
- Differs from strict forward/backward static analysis

Example

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Analysis

Example

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Analysis

$$\Longrightarrow [\texttt{i=0;...}] \mid (tot)(sp_0, use_0)$$

Example

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Analysis

$$\frac{\Longrightarrow \{\texttt{i} := 0\}[\texttt{count=n;...}] \mid (\mathit{tot})(\mathit{sp}_1, \mathit{use}_1)}{\Longrightarrow [\texttt{i=0;...}] \mid (\mathit{tot})(\mathit{sp}_0, \mathit{use}_0)}$$

Example

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Analysis

$$\Longrightarrow \{\ldots \,||\, \texttt{count} := \texttt{n}\}[\texttt{tot=0;while(i<=n)}\ldots] \mid (tot)(sp_2, use_2)$$

$$\Longrightarrow \{\texttt{i} := 0\}[\texttt{count=n;}\ldots] \mid (tot)(sp_1, use_1)$$

$$\Longrightarrow [\texttt{i=0;}\ldots] \mid (tot)(sp_0, use_0)$$

Example

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Analysis

$$\Longrightarrow \{\ldots \,||\, \texttt{tot} := 0\}[\texttt{while(i<=n)}\ldots] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\ldots \,||\, \texttt{count} := \texttt{n}\}[\texttt{tot=0;while(i<=n)}\ldots] \mid (tot)(sp_2, use_2)$$

$$\Longrightarrow \{\texttt{i} := 0\}[\texttt{count=n;}\ldots] \mid (tot)(sp_1, use_1)$$

$$\Longrightarrow [\texttt{i=0;}\ldots] \mid (tot)(sp_0, use_0)$$

Example Cont'd: Loop Unwind

$$\Longrightarrow \{\texttt{i} := 0 || \ldots || \texttt{tot} := 0\}[\texttt{while(i<=n)...}] \mid (tot)(sp_3, use_3)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind

$$\frac{\Longrightarrow \{\texttt{i} := 0 || \ldots\}[\texttt{if(i<=n)\{...;if(i>=2 \&\& cpn)...;i++;while...\}}] \mid (tot)(sp_3, use_3)}{\Longrightarrow \{\texttt{i} := 0 || \ldots || \texttt{tot} := 0\}[\texttt{while(i<=n)...}] \mid (tot)(sp_3, use_3)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind

$$\frac{\Longrightarrow \{\texttt{i} := 0 || \ldots\}[\texttt{if(0<=n)\{...;if(0>=2 \&\& cpn)...;i=0+1;while...\}}] \mid (tot)(sp_3, use_3)}{\Longrightarrow \{\texttt{i} := 0 || \ldots\}[\texttt{if(i<=n)\{...;if(i>=2 \&\& cpn)...;i++;while...\}}] \mid (tot)(sp_3, use_3)}$$

$$\Longrightarrow \{\texttt{i} := 0 || \ldots || \texttt{tot} := 0\}[\texttt{while(i<=n)...}] \mid (tot)(sp_3, use_3)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind

$$\Longrightarrow \{\texttt{i} := 0 || \ldots ; \texttt{tot} := 0\}[\texttt{if(0<=n)\{...;tot=0+m;i=1;while...\}}] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\texttt{i} := 0 || \ldots\}[\texttt{if(0<=n)\{...;if(0>=2 \&\& cpn)...;i=0+1;while...\}}] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\texttt{i} := 0 || \ldots\}[\texttt{if(i<=n)\{...;if(i>=2 \&\& cpn)...;i++;while...\}}] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\texttt{i} := 0 || \ldots || \texttt{tot} := 0\}[\texttt{while(i<=n)...}] \mid (tot)(sp_3, use_3)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind

$$\Longrightarrow \{\texttt{i} := 0 || \ldots || \texttt{tot} := 0\}[\texttt{if(0<=n)\{int m=read();tot=m;i=1;while...\}}] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\texttt{i} := 0 || \ldots ; \texttt{tot} := 0\}[\texttt{if(0<=n)\{...;tot=0+m;i=1;while...\}}] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\texttt{i} := 0 || \ldots\}[\texttt{if(0<=n)\{...;if(0>=2 \&\& cpn)...;i=0+1;while...\}}] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\texttt{i} := 0 || \ldots\}[\texttt{if(i<=n)\{...;if(i>=2 \&\& cpn)...;i++;while...\}}] \mid (tot)(sp_3, use_3)$$

$$\Longrightarrow \{\texttt{i} := 0 || \ldots || \texttt{tot} := 0\}[\texttt{while(i<=n)...}] \mid (tot)(sp_3, use_3)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind 2nd Round

$\Longrightarrow \{\ldots\}$[`if(0<=n){int m=read();tot=m;i=1;while...}`] | $(tot)(sp_3, use_3)$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind 2nd Round

$$\frac{\neg(0 <= n) \Longrightarrow \{\ldots\}[] \mid (tot)(nop, tot) \qquad 0 <= n \Longrightarrow \{\ldots\}[\texttt{int m=read();...}] \mid (tot)(sp_4, use_4)}{\Longrightarrow \{\ldots\}[\texttt{if(0<=n)\{int m=read();tot=m;i=1;while...\}}] \mid (tot)(sp_3, use_3)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind 2nd Round

$$\dfrac{\neg(0 <= n) \Longrightarrow \{\ldots\}[] \mid (tot)(nop, tot) \qquad \dfrac{0 <= n \Longrightarrow \{\ldots\}[\texttt{int m=read();...}] \mid (tot)(sp_4, use_4)}{\ldots}}{\Longrightarrow \{\ldots\}[\texttt{if(0<=n)\{int m=read();tot=m;i=1;while...\}}] \mid (tot)(sp_3, use_3)}$$

$$\dfrac{0 <= n \Longrightarrow \{\ldots \| \texttt{m} := \texttt{read()} \| \texttt{tot} := \texttt{m} \| \texttt{i} := 1\}[\texttt{while(i<=n)...}] \mid (tot)(sp_5, use_5)}{\ldots}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind 2nd Round

$$\dfrac{\neg(0 <= n) \Longrightarrow \{\ldots\}[] \mid (tot)(nop, tot) \qquad \dfrac{\dfrac{0 <= n \Longrightarrow \{\ldots\}[\texttt{if(i<=n)...;while...}] \mid (tot)(sp_5, use_5)}{0 <= n \Longrightarrow \{\ldots \| \texttt{m := read()} \| \texttt{tot := m} \| \texttt{i := 1}\}[\texttt{while(i<=n)...}] \mid (tot)(sp_5, use_5)}}{\ldots} \quad 0 <= n \Longrightarrow \{\ldots\}[\texttt{int m=read();...}] \mid (tot)(sp_4, use_4)}{\Longrightarrow \{\ldots\}[\texttt{if(0<=n)\{int m=read();tot=m;i=1;while...\}}] \mid (tot)(sp_3, use_3)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Unwind 2nd Round

$$\dfrac{\neg(0 <= n) \Longrightarrow \{\ldots\}[] \mid (tot)(nop, tot) \qquad \dfrac{\dfrac{\dfrac{1 <= n \Longrightarrow \{\ldots \| \texttt{i} := 2\}[\texttt{while(i<=n)...}] \mid (tot)(sp_6, use_6)}{\ldots}}{0 <= n \Longrightarrow \{\ldots\}[\texttt{if(i<=n)...;while...}] \mid (tot)(sp_5, use_5)}}{0 <= n \Longrightarrow \{\ldots \| \texttt{m} := \texttt{read()} \| \texttt{tot} := \texttt{m} \| \texttt{i} := 1\}[\texttt{while(i<=n)...}] \mid (tot)(sp_5, use_5)}}{\ldots}}{\Longrightarrow \{\ldots\}[\texttt{if(0<=n)\{int m=read();tot=m;i=1;while...\}}] \mid (tot)(sp_3, use_3)}$$

$$0 <= n \Longrightarrow \{\ldots\}[\texttt{int m=read();...}] \mid (tot)(sp_4, use_4)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$1 <= n \Longrightarrow \{\ldots \| \texttt{i} := 2\}[\texttt{while(i<=n)...}] \mid (tot)(sp_6, use_6)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\frac{\ldots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \qquad \ldots, i <= n \Longrightarrow [\texttt{int } \ldots] \mid (tot \cup i \cup tot)(sp_7, use_7)}{1 <= n \Longrightarrow \{\ldots \| \texttt{i} := 2\}[\texttt{while(i<=n)}\ldots] \mid (tot)(sp_6, use_6)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\frac{\ldots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \qquad \frac{\ldots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)...}] \mid (tot \cup i)(sp_8, use_8)}{\ldots, i <= n \Longrightarrow [\texttt{int ...}] \mid (tot \cup i \cup tot)(sp_7, use_7)}}{1 <= n \Longrightarrow \{\ldots \| \texttt{i := 2}\}[\texttt{while(i<=n)...}] \mid (tot)(sp_6, use_6)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \quad \dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}{\dots \Longrightarrow \{\texttt{m} := \texttt{read()}\}[\texttt{if(cpn)}\dots] \mid (tot \cup i)(sp_8, use_8)} \quad \dots, i <= n \Longrightarrow [\texttt{int } \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)}{1 <= n \Longrightarrow \{\dots \| \texttt{i} := 2\}[\texttt{while(i<=n)}\dots] \mid (tot)(sp_6, use_6)}$$

$$\dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots \Longrightarrow \{\dots \| \texttt{tot} := \texttt{tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \quad \dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}}{\dots \Longrightarrow \{\texttt{m} := \texttt{read()}\}[\texttt{if(cpn)}\dots] \mid (tot \cup i)(sp_8, use_8)}}{\dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \quad \dots, i <= n \Longrightarrow [\texttt{int } \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)}}{1 <= n \Longrightarrow \{\dots \| \texttt{i} := 2\}[\texttt{while(i<=n)}\dots] \mid (tot)(sp_6, use_6)}$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \quad \dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}{\dots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)}\dots] \mid (tot \cup i)(sp_8, use_8)}}{\dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \quad \dots, i <= n \Longrightarrow [\texttt{int } \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)}}{\dots \Longrightarrow \{\dots \| \texttt{tot := tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}}{\dots \Longrightarrow \{\dots \| \texttt{i := i+1}\}[] \mid (tot \cup i)(sp_{12}, use_{12})}$$

$$1 <= n \Longrightarrow \{\dots \| \texttt{i := 2}\}[\texttt{while(i<=n)}\dots] \mid (tot)(sp_6, use_6)$$

Original Java Code

```
...
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \quad \dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}{\dots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)}\dots] \mid (tot \cup i)(sp_8, use_8)} \quad \dots, i <= n \Longrightarrow [\texttt{int } \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)}{\dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \quad 1 <= n \Longrightarrow \{\dots \| \texttt{i := 2}\}[\texttt{while(i<=n)}\dots] \mid (tot)(sp_6, use_6)}}{\dots \Longrightarrow \{\dots \| \texttt{tot := tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}}{\dots \Longrightarrow \{\dots \| \texttt{i := i+1}\}[] \mid (tot \cup i)(sp_{12}, use_{12})}$$

Synthesis

- $sp_{12} : nop$ $\qquad$ $use_{12} : \{tot, i\}$

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\dfrac{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \quad \dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}{\dots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)}\dots] \mid (tot \cup i)(sp_8, use_8)}}{\dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \quad \dots, i <= n \Longrightarrow [\texttt{int } \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)}}{1 <= n \Longrightarrow \{\dots \| \texttt{i := 2}\}[\texttt{while(i<=n)}\dots] \mid (tot)(sp_6, use_6)}$$

$$\dfrac{\dfrac{\dots \Longrightarrow \{\dots \| \texttt{i := i+1}\}[] \mid (tot \cup i)(sp_{12}, use_{12})}{\dots \Longrightarrow \{\dots \| \texttt{tot := tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}}{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \quad \dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}$$

Synthesis

- $sp_{12} : nop$ $\qquad use_{12} : \{tot, i\}$
- $sp_{10} : tot = tot + m; i++;$ $\qquad use_{10} : \{tot, i\}$

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \quad \dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}{\dots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)}\dots] \mid (tot \cup i)(sp_8, use_8)} \quad \dots \Longrightarrow \{\dots \| \texttt{tot := tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}{\dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \quad \dots, i <= n \Longrightarrow [\texttt{int } \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)} \quad \dots \Longrightarrow \{\dots \| \texttt{i := i+1}\}[] \mid (tot \cup i)(sp_{12}, use_{12})}{1 <= n \Longrightarrow \{\dots \| \texttt{i := 2}\}[\texttt{while(i<=n)}\dots] \mid (tot)(sp_6, use_6)}$$

Synthesis

- $sp_{12} : nop$ $\qquad use_{12} : \{tot, i\}$
- $sp_{10} : tot = tot + m; i++;$ $\qquad use_{10} : \{tot, i\}$
- $sp_8 : if(cpn)\{tot = tot + m * 0.9; i++;\}$
  $else\{tot = tot + m; i++;\}$ $\qquad use_8 : \{tot, i, cpn\}$

Example Cont'd: Loop Invariant True

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots, cpn \Longrightarrow \dots \mid (tot \cup i)(sp_9, use_9) \quad \dots, \neg cpn \Longrightarrow \{\dots\}[\texttt{tot=tot+m;}\dots] \mid (tot \cup i)(sp_{10}, use_{10})}{\dots \Longrightarrow \{\texttt{m := read()}\}[\texttt{if(cpn)}\dots] \mid (tot \cup i)(sp_8, use_8)}}{\dots, \neg(i <= n) \Longrightarrow [] \mid (tot)(nop, tot) \quad \dots, i <= n \Longrightarrow [\texttt{int } \dots] \mid (tot \cup i \cup tot)(sp_7, use_7)}}{1 <= n \Longrightarrow \{\dots \| \texttt{i := 2}\}[\texttt{while(i<=n)}\dots] \mid (tot)(sp_6, use_6)}$$

$$\dfrac{\dots \Longrightarrow \{\dots \| \texttt{i := i+1}\}[] \mid (tot \cup i)(sp_{12}, use_{12})}{\dots \Longrightarrow \{\dots \| \texttt{tot := tot+m}\}[\texttt{i++;}] \mid (tot \cup i)(sp_{11}, use_{11})}$$

Synthesis

- $sp_{12} : nop$ $\quad use_{12} : \{tot, i\}$
- $sp_{10} : tot = tot + m; i++;$ $\quad use_{10} : \{tot, i\}$
- $sp_8 : if(cpn)\{tot = tot + m * 0.9; i++;\}$
  $else\{tot = tot + m; i++;\}$ $\quad use_8 : \{tot, i, cpn\}$
- $sp_6 : while(i <= n)\{int\ m = read();$
  $if(cpn)\{tot = tot + m * 0.9; i++;\}$
  $else\{tot = tot + m; i++;\}\}$ $\quad use_6 : \{tot, i, cpn\}$

Specialized Program

Specialized Java Code

```
tot = 0;
if(0 <= n) {
  int m = read();
  tot = m;
  if(1 <= n) {
    int m = read();
    tot = tot + m;
    i = 2;
    while(i <= n) {
      int m = read();
      if (cpn) {
        tot = tot + 0.9 * m;
        i++;
      } else {
        tot = tot + m;
        i++;
      }
    }
  }
}
return tot;
```

Specialized Program

Original Java Code

```
i = 0;
count = n;
tot = 0;
while(i <= count) {
 int m = read();
 if(i >=2 && cpn)
  tot = tot + m * 0.9;
 else
  tot = tot + m;
 i++;
}
return tot;
```

Specialized Java Code

```
tot = 0;
if(0 <= n) {
  int m = read();
  tot = m;
  if(1 <= n) {
    int m = read();
    tot = tot + m;
    i = 2;
    while(i <= n) {
      int m = read();
      if (cpn) {
        tot = tot + 0.9 * m;
        i++;
      } else {
        tot = tot + m;
        i++;
      }
    }
  }
}
return tot;
```

Bytecode Compilation

$$\frac{\Gamma \Longrightarrow \{l := r\}[\texttt{rest}]\ (\overline{rest}, use)}{\Gamma \Longrightarrow [\texttt{l=r;rest}]\ \begin{cases} \texttt{iload r}; \texttt{istore l}; \overline{rest}, ((use - \{l\}) \cup \{r\}) & \text{if } l \in use \\ \overline{rest}, use & \text{otherwise} \end{cases}}$$

Realise a rule-based Java bytecode compiler:

- Change the target language from Java source code to Java bytecode
- Single static assignment form: easy to synthesize bytecode
- Compiler correctness: soundness of program logic + local bisimulation
- Some available optimizations (FO reasoning, partial evaluation):
  - dead code elimination (can't reach unexecuted code in closed branches)
  - type inference
  - safety analysis (avoid creation of exception handlers)
  - constant propagation, expression simplification
  - precise usage, binding time analysis of variables

Summary

- New architecture to construct verified compilers:
  Verification + PE + local transformation = (verified) compiler
- Correctness of symbolic execution rules & bisimulation property guarantee correct specialisation/compilation
- Symbolic execution permits in dynamic analysis at compile time
- First-order reasoning, partial evaluation integrated
  - Infeasible path detection + Interleaving partial evaluation:
    $\Rightarrow$ specialized and optimized programs
- Use-definition chains are maintained to eliminate unused assignments
  - Further analyses can be added
- Contracts (variable import/export) computed automatically:
  compositional (can compile methods independently)
- Implementation in KeY verification system ongoing

Related Work, Outlook

Related Work

- Compiler verification
- Hoare's Grand Challenge "The Verifying Compiler"
  $\Rightarrow$ "The Compiling Verifier"
- Rule-based compilation
- Translation validation of optimizing compilers
- Online partial evaluation
- MSR trace-based compilation

Outlook

- Invariants in recursive calls, parallelize independent code
- Room for heuristics:
  - to unwind or not to unwind
  - merge tails (e.g., by computing product)
- Import information from other tools, e.g., invariants
- Add security/safety properties in asserts or postcondition:
  detect violation and patch with inlined monitors, wrappers, etc.