Abstract Interpretation
&
Symbolic Execution

Reiner Hähnle

joint work with: Richard Bubel (Chalmers), Benjamin Weiss (KIT)

Högre seminariet i logik
Institutionen för filosofi, lingvistik och vetenskapsteori
Göteborgs universitet

Chalmers University of Technology, Gothenburg, Sweden

19 November 2010
The “usual” (i.e., well-known) software challenges

- Software is ubiquitous
  50% of productivity gain in industrial societies (EC FP7 ICT)
- Size and complexity of software systems
  Software in modern cars in GB-range
- Growing safety/security demands, certification
e-documents, health, transport, finances, connectedness
- Increasing effort for validation and maintenance of software
  The more successful a piece of software, the longer it is in use
- Re-use & adaptation is the norm
  Libraries, system integration
- Short production cycles, rapidly changing requirements
  Innovation through software
Some Suggested Solutions

Software processes
- Optimization of effort
- Empirical studies, agile methods

Requirements analysis
- Optimization of the intended result
- Customer in-the-loop, use-case \( \rightarrow \) task

Abstract modeling
- Reduction of complexity, code generation
- Architectural models, MDD, SW product lines

Validation
- Inspection, testing
- Model-based test generation

Formalisation
- Mechanisation, tools
- Static analysis, formal verification
Some Suggested Solutions

Software processes
Optimization of effort
Empirical studies, agile methods

Requirements analysis
Optimization of the intended result
Customer in-the-loop, use-case → task

Abstract modeling
Reduction of complexity, code generation
Architectural models, MDD, SW product lines

Validation
Inspection, testing
Model-based test generation

Formalisation
Mechanisation, tools
Static analysis, formal verification

“No silver bullet” — Concertation is essential!
Established Software Production Tools

- IDEs, editors, metrics
- Compilation, type checking
- Interactive debugging
- Management and automated execution of test suites
Established Software Production Tools

- IDEs, editors, metrics
- Compilation, type checking
- Interactive debugging
- Management and automated execution of test suites

Formalisation: Prerequisite for further Automation

Formal semantics of modeling & programming languages:

- Eliminating ambiguity
- Formal guarantees of software properties and test coverage
- Basis for automated verification and glass box test generation
- Proof of properties via certification objects
- Cooperation/combination of heterogeneous technologies
What is Static Analysis?

Analysis of Software Properties at Compile Time

Local and with limitations since many years in compilers

The earlier errors are detected the cheaper it is to fix them!

detection of potential run-time errors
detection of mismatches between code and specification
proof of error-freeness (wrt. a formal model)

Optimization of performance (memory leaks, dead code, evaluation)

Automation saves effort
What is Static Analysis?

Analysis of Software Properties at Compile Time

- Local and with limitations since many years in compilers
- The earlier errors are detected the cheaper it is to fix them!
  - detection of potential run-time errors
  - detection of mismatches between code and specification
  - proof of error-freeness (wrt. a formal model)
- Optimization of performance (memory leaks, dead code, evaluation)
- Automation saves effort
What is Static Analysis?

Analysis of Software Properties at Compile Time

- Local and with limitations since many years in compilers
- The earlier errors are detected the cheaper it is to fix them!
  - detection of potential run-time errors
  - detection of mismatches between code and specification
  - proof of error-freeness (wrt. a formal model)
- Optimization of performance (memory leaks, dead code, evaluation)
- Automation saves effort

Static Analysis: Representative Approaches

- Extended type systems
- Data/control flow analysis, code slicing
- Abstract interpretation
- Model checking
- Formalisation in program logic, deduction, symbolic execution
What is Static Analysis?

Analysis of Software Properties at Compile Time

- Local and with limitations since many years in compilers
- The earlier errors are detected the cheaper it is to fix them!
  - detection of potential run-time errors
  - detection of mismatches between code and specification
  - proof of error-freeness (wrt. a formal model)
- Optimization of performance (memory leaks, dead code, evaluation)
- Automation saves effort

Static Analysis: Representative Approaches

- Extended type systems
- Data/control flow analysis, code slicing
- Abstract interpretation
- Model checking
- Formalisation in program logic, deduction, symbolic execution
“Conventional” Type Systems

...used in modern programming language guarantee at compile time that the values of all variables at runtime conform to their declared type.

- important security and safety property
- can prevent attacks and crashes
- programmer is responsible for annotation of variables with types
“Conventional” Type Systems

...used in modern programming language guarantee at compile time that the values of all variables at runtime conform to their declared type.

- important security and safety property
- can prevent attacks and crashes
- programmer is responsible for annotation of variables with types

Extended Type Systems

...attempt to guarantee properties beyond type conformance:

- a reference type variable never has value null when accessed
- the value of a certain variable does/does not depend on another
Non-interference: Public variables don’t depend on secret ones
Non-interference: **Public** variables don’t depend on **secret** ones
Non-interference: Public variables don’t depend on secret ones
A Type System for Security Analysis

Non-interference: Public variables don’t depend on secret ones

Typical State-of-Art Type System

- Hunt-Sands, POPL 2006, On Flow-Sensitive Security Types
- Catches indirect information flow
  \[
  \text{if (secret) public = 0 else public = 1;}
  \]
- Type checking in polynomial time
- Approximation (false positives): wrongly classified as insecure
  \[
  \text{if (secret) public = 0 else public = 0;}
  \]
- Implemented for Spark, problematic for Java (aliasing)
All configurations at each program location with abstract values
Abstract Interpretation

All configurations at each program location with abstract values

- Due to Cousot & Cousot, 1977ff
- Commercial use, e.g. in A3XX software
- Termination when no $\infty$ ascending chains in abstract domain
- Abstract interpretation of all operators in target program
- Soundness: in abstract domain reachable states approximate all reachable states in concrete computations
Abstract Interpretation

- All configurations at each program location with abstract values
- Due to Cousot & Cousot, 1977ff
- Commercial use, e.g. in A3XX software
- Termination when no infinite ascending chains in abstract domain
- Abstract interpretation of all operators in target program
- Soundness: in abstract domain reachable states approximate all reachable states in concrete computations

```c
gSum=0, index=\geq

int gSum = 0;
while (index > 0) {
    index = index - 1;
    gSum = gSum + index;
}
```
Abstract Interpretation

- Due to Cousot & Cousot, 1977ff
- Commercial use, e.g. in A3XX software
- Termination when no $\infty$ ascending chains in abstract domain
- Abstract interpretation of all operators in target program
- Soundness: in abstract domain reachable states approximate all reachable states in concrete computations

```c
int gSum = 0; // gSum=0, index>=
while (index > 0) {
    index = index - 1;
    gSum = gSum + index;
}
```
Abstract Interpretation

- All configurations at each program location with abstract values

- Due to Cousot & Cousot, 1977ff
- Commercial use, e.g. in A3XX software
- Termination when no $\infty$ ascending chains in abstract domain
- Abstract interpretation of all operators in target program
- Soundness: in abstract domain reachable states approximate all reachable states in concrete computations

```c
int gSum = 0;
while (index > 0) {
    index = index - 1;
    gSum = gSum + index;
}
gSum = 0, index = 0
```
Abstract Interpretation

All configurations at each program location with abstract values

- Due to Cousot & Cousot, 1977ff
- Commercial use, e.g. in A3XX software
- Termination when no $\infty$ ascending chains in abstract domain
- Abstract interpretation of all operators in target program
- Soundness: in abstract domain reachable states approximate all reachable states in concrete computations

```c
int gSum = 0;
while (index>0) {
    index = index - 1;  // gSum=0, index>=
gSum = gSum + index;
}  // gSum=0, index=0
```
Abstract Interpretation

All configurations at each program location with abstract values

- Due to Cousot & Cousot, 1977ff
- Commercial use, e.g. in A3XX software
- Termination when no $\infty$ ascending chains in abstract domain
- Abstract interpretation of all operators in target program
- Soundness: in abstract domain reachable states approximate all reachable states in concrete computations

```plaintext
gSum = 0;
while (index > 0) {
    index = index - 1;
    gSum = gSum + index;  \[ gSum=\geq, \ index=\geq \]
} \[ gSum=0, \ index=0 \]
```
Abstract Interpretation

- All configurations at each program location with abstract values
- Due to Cousot & Cousot, 1977ff
- Commercial use, e.g. in A3XX software
- Termination when no \( \infty \) ascending chains in abstract domain
- Abstract interpretation of all operators in target program
- Soundness: in abstract domain reachable states approximate all reachable states in concrete computations

```java
int gSum = 0;
while (index > 0) {
    index = index - 1;
    gSum = gSum + index;
}
gSum = \geq, index = \geq
```
Abstract Interpretation: Value Domains as Lattices

All configurations at each program location with abstract values
Abstract Interpretation: Value Domains as Lattices

All configurations at each program location with abstract values

\[ \mathbb{Z}_{\leq 0} \leftarrow \gamma \leftarrow \mathbb{Z}_{< 0} \]

\[ \leq \quad \geq \]

\[ < \quad \gamma \quad \gamma \quad > \]

\[ gSum = 0, \text{index} \geq 0 \]

```plaintext
gSum = 0;
while (index > 0) {
    index = index - 1;
    gSum = gSum + index;
}
```
All configurations at each program location with abstract values

```
gSum = 0;
while (index > 0) {
    index = index - 1;
    gSum = gSum + index;
}
```
Abstract Interpretation: Value Domains as Lattices

All configurations at each program location with abstract values

\[ \mathbb{Z}_{\leq 0} \subset \mathbb{Z}_{< 0} \]

\[ gSum = 0, \ index = \geq \]

```c
int gSum = 0;
while (index > 0) {
    index = index - 1;
    gSum = gSum + index;
}
```
Symbolic Execution

Execution of one program run with symbolic values

- Due to King 1969ff
- Characterisation of program semantics with calculus: Hoare 1969
- Symbolic execution as verification: Burstall 1974
- Symbolic execution in program logic: Stephan et al., 1986

Characterisation of Soundness in Program Logic

Modeling as used in KeY-System, Hähnle et al., 1998ff

- **Update** Representation of symbolic configuration of program variables (state) with explicit substitutions:
  \[ \mathcal{U} = \{ g\text{Sum} := g_0 + i_0 \mid \text{index} := i_0 - 1 \} \]

- **Formulas** Partial correctness of program \( p \) in start state \( \mathcal{U} \) wrt. post condition: \( \mathcal{U}[p]^{\text{post}} \)

- **Example** \( \text{index} \geq 0 \rightarrow [\text{Gauss}](\text{index} \div 0 \& \text{gSum} \geq 0) \)
Symbolic Execution

All configurations at each program location with symbolic values

- Due to King 1969ff
- Characterisation of program semantics with calculus: Hoare 1969
- Symbolic execution as verification: Burstall 1974
- Symbolic execution in program logic: Stephan et al., 1986

Characterisation of Soundness in Program Logic

Modeling as used in KeY-System, Hähnle et al., 1998ff

**Update** Representation of symbolic configuration of program variables (state) with explicit substitutions:

\[ U = \{ \text{gSum := } g_0 + i_0 \mid \| \text{index := } i_0 - 1 \} \]

**Formulas** Partial correctness of program \( p \) in start state \( U \) wrt. post condition: \( U[p]_\text{post} \)

**Example** \( \text{index} \geq 0 \rightarrow [\text{Gauss}](\text{index} \div 0 \& \text{gSum} \geq 0) \)
## Partial Correctness Formulas

- Pre $\rightarrow$ [P]Post corresponds to Hoare triple $\{\text{Pre}\}P\{\text{Post}\}$
- Dynamic logic: multi-modal logic with operator for each program $P$
- Kripke semantics over program states:

$$s \models [P]\psi \iff \text{for all } t: \text{if } s[[P]]t \text{ then } t \models \psi$$
Partial Correctness Formulas

- Pre $\rightarrow$ [P]Post corresponds to Hoare triple $\{\text{Pre}\}P\{\text{Post}\}$
- Dynamic logic: multi-modal logic with operator for each program $P$
- Kripke semantics over program states:
  \[ s \models [P]\psi \iff \text{for all } t: \text{if } s[\lceil P\rfloor t \text{ then } t \models \psi } \]

Updates (generalized explicit substitutions)

- $U = \{v_1 := t_1 || \cdots || v_n := t_n\}$, $v_i$ program variables, $t_i$ f.-o. terms
- $U\varphi$ when $\varphi$ f.-o. formula: like substitution
- $U\varphi$ when $\varphi$ program formula: delay application
  - $U$ accumulate in front of program formulas during symbolic execution
  - Updates simplified constantly (rewrite system with normal form)
- Extensions for object access, arrays, conditional terms
**Partial Correctness Formulas**

- **Pre** $\rightarrow$ $[P]Post$ corresponds to Hoare triple $\{Pre\}P\{Post\}$
- Dynamic logic: multi-modal logic with operator for each program $P$
- Kripke semantics over program states:
  $$s \models [P]\psi \text{ iff for all } t: \text{ if } s[\![P]\!]t \text{ then } t \models \psi$$

**Assignment**

\[
\Gamma \Rightarrow U\{x := e\}[\text{rest}]\phi, \Delta \\
\Gamma \Rightarrow U[x=e; \text{ rest}]\phi, \Delta
\]

where $e$ is a simple expression (no side effects, method calls)
Symbolic Execution of Conditional Statement

\[
\frac{\Gamma, U_b \Rightarrow U[p; \text{rest}]\phi, \Delta \quad \Gamma, U!b \Rightarrow U[q; \text{rest}]\phi, \Delta}{\Gamma \Rightarrow U[\text{if } (b) \{ p \} \text{ else } \{ q \}; \text{rest}]\phi, \Delta}
\]

Splitting symbolic execution in different branches
path conditions \((U_b, U!b)\) essential for precise modeling
Symbolic Execution in Program Logic Cont’d

Symbolic Execution of Conditional Statement

\[
\Gamma, \mathcal{U}b \Rightarrow \mathcal{U}[p; \text{rest}]\phi, \Delta \\
\Gamma, \mathcal{U}!b \Rightarrow \mathcal{U}[q; \text{rest}]\phi, \Delta \\
\Gamma \Rightarrow \mathcal{U}[\text{if } (b) \{ p \} \text{ else } \{ q \}; \text{rest}]\phi, \Delta
\]

Splitting symbolic execution in different branches
path conditions \((\mathcal{U}b, \mathcal{U}!b)\) essential for precise modeling

Symbolic Execution of Loop Statement

\[
\text{unwindLoop} \\
\Gamma \Rightarrow \mathcal{U}[\text{if } (b) \{ p; \text{while } (b) p \}; \text{rest}]\phi, \Delta \\
\Gamma \Rightarrow \mathcal{U}[\text{while } (b) \{p\}; \text{rest}]\phi, \Delta
\]

Without fixed loop bound: no finite symbolic execution
Extended Type Systems
- fully automatic
- efficient
- low expressivity
- depend on target language
- mostly prototypes

Abstract Interpretation
- fully automatic
- terminating
- approximation
- fixed degree of precision
- commercial usage

Program Logic
- interactive or annotations
- high expressivity
- logic-based specification
- real-world case studies
Type Systems, Abstract Interpretation, Program Logic

Extended Type Systems
- fully automatic
- efficient
- low expressivity
- depend on target language
- mostly prototypes

Abstract Interpretation
- fully automatic
- terminating
- approximation
- fixed degree of precision
- commercial usage

Program Logic
- interactive or annotations
- high expressivity
- logic-based specification
- real-world case studies

Integration? Synergies?
How are these three techniques related?

1. View a calculus for type inference as abstract interpretation

   Complex assignment
   concrete/symbolic execution
   
   \[ x = (x \mod 2 \times y) \times z - 327; \]

   Hunt-Sands type system as book-keeping of variable dependencies

2. View symbolic execution as abstract interpretation

   Infinite abstract domain: logic formulas over memory configurations

   Consequence

   Abstract interpretation compatible with symbolic execution in program logic

   Conceptual integration of all three techniques possible!
How are these three techniques related?

1. View a calculus for type inference as abstract interpretation

\[
x = (x \% 2 \ast y) \ast z - 327;
\]

\[
\downarrow \text{Abstraction}
\]

\[
x = (x, y, z);
\]

Complex assignment
concrete/symbolic execution

Hunt-Sands type system as bookkeeping of variable dependencies
How are these three techniques related?

1. View a calculus for type inference as abstract interpretation

   \[ x = (x \mod 2 \times y) \times z - 327; \]
   \[ x = (x, y, z); \]

   Complex assignment
concrete/symbolic execution
   \[ \downarrow \text{Abstraction} \]
   Hunt-Sands type system as book-keeping of variable dependencies

2. View symbolic execution as abstract interpretation

   Infinite abstract domain: logic formulas over memory configurations
How are these three techniques related?

1. View a calculus for type inference as abstract interpretation

\[
\begin{align*}
    x &= (x \% 2 \times y) \times z - 327; \\
         &\downarrow \text{Abstraction} \\
    x &= (x, y, z);
\end{align*}
\]
Complex assignment
concrete/symbolic execution
Hunt-Sands type system as bookkeeping of variable dependencies

2. View symbolic execution as abstract interpretation

Infinite abstract domain: logic formulas over memory configurations

Consequence

- Abstract interpretation compatible with symbolic execution in program logic
- Conceptual integration of all three techniques possible!
Realisation in reverse order:

1. Implementation of abstraction in program logic
   - logic signature for abstract domains and abstraction function
   - calculus for abstraction steps and execution of abstract programs
   - soundness conditions

2. Model (information flow) type system as abstract domain
Abstraction and Symbolic Execution

Concret domain
Program states
\{s : \text{Var} \rightarrow \mathcal{D}\}
(set lattice)
\(\gamma(\alpha(S))\)

Abstract domain
Formulas over program states
(\(\infty\) lattice)
\(\alpha(S)\)

Abstraction \(\alpha\)
Concretisation \(\gamma\)

Symbolic execution as infinite, abstract Domain
Abstraction and Symbolic Execution

Concret domain
Program states
\( \{s : \text{Var} \rightarrow D \} \)
(set lattice)
\( \gamma(\alpha(S)) \)

\( S \)

Abstract domain
Formulas over abstract states
(finite lattice)
\( \alpha(S) \)

Abstraction \( \alpha \)
Concretisation \( \gamma \)

Symbolic execution in finite, abstract Domain
Signature (first-order) for abstract Domain $\mathcal{A}$

- for each $a \in \mathcal{A}$ and $z \in \mathbb{Z}$ there is a constant $\gamma_{a,z}$
- for each $a \in \mathcal{A}$ there is a unary predicate $\chi_a$

"the $\gamma_{a,z}$ represent the abstract values"

"the $\chi_a$ are characteristic sets of the abstract values"
Signature (first-order) for abstract Domain $\mathcal{A}$

- for each $a \in \mathcal{A}$ and $z \in \mathbb{Z}$ there is a constant $\gamma_{a,z}$
- for each $a \in \mathcal{A}$ there is a unary predicate $\chi_a$

We admit only interpretations $I$ with the following properties:

- for each $a \in \mathcal{A}$ and each $z \in \mathbb{Z}$: $I(\gamma_{a,z}) \in \gamma(a)$
  “the $\gamma_{a,z}$ represent the abstract values”
- for each $a \in \mathcal{A}$: $I(\chi_a) = \gamma(a)$
  “the $\chi_a$ are characteristic sets of the abstract values”
Example: Abstract Domain

gSum = 0;
while (index>0) {
    index = index - 1;
    gSum = gSum + index;
}

∅

\[ a \in A \forall x. (\chi < (\gamma <, Z)) \]
Example: Abstract Domain


gSum = 0;
while (index>0) {
    index = index - 1;
    gSum = gSum + index;
}

\[
\begin{array}{c|c|c|c}
  a \in \mathcal{A} & \chi_a(x) & \gamma_{a,\mathbb{Z}} & \text{axioms for the new symbols} \\
  \langle & \chi_< & \gamma_<,1,\ldots & \forall x. (\chi_< (x) \iff x < 0), \chi_< (\gamma_<,\mathbb{Z}) \\
\end{array}
\]
Example: Abstract Domain

```c
int gSum = 0;
while (index > 0) {
    index = index - 1;
    gSum = gSum + index;
}
```

![Diagram](image)

<table>
<thead>
<tr>
<th>$a \in A$</th>
<th>$\chi_a(x)$</th>
<th>$\gamma_{a,\mathbb{Z}}$</th>
<th>axioms for the new symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;$</td>
<td>$\chi&lt;$</td>
<td>$\gamma_{&lt;,\mathbb{Z}}$</td>
<td>$\forall x. (\chi_&lt;(x) \leftrightarrow x &lt; 0), \chi_&lt;(\gamma_{&lt;,\mathbb{Z}})$</td>
</tr>
<tr>
<td>0</td>
<td>$\chi_0$</td>
<td>$\gamma_{0,\mathbb{Z}}$</td>
<td>$\forall x. (\chi_0(x) \leftrightarrow x \neq 0), \chi_0(\gamma_{0,\mathbb{Z}})$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

$a \in A$  $\chi_a(x)$  $\gamma_{a,\mathbb{Z}}$
Abstraction of target program

Replace

\[ \Gamma \Rightarrow [p] \varphi \]

with

\[ \alpha(\Gamma) \Rightarrow [\alpha(p)] \alpha(\varphi) \]

- formal semantics for complete target language (e.g., JAVA)
- rules for abstract execution of arbitrary programs
Abstraction of updates

Replace

\[ \Gamma \Rightarrow U[p] \varphi \]

with

\[ \Gamma \Rightarrow \alpha(U)[p] \varphi \]

- Theory of abstraction of updates is sufficient
Abstract Deduction in Program Logic

Abstraction of updates

Replace

\[ \Gamma \Rightarrow U[p] \varphi \]

with

\[ \Gamma \Rightarrow \alpha(U)[p] \varphi \]

- Theory of abstraction of updates is sufficient

Decisive insight: abstraction \(\approx\) logic weakening (implication)
Weakening of Updates

\[ \{ x := 3 \| y := 3 - 5 \} \leadsto \]

Replace an update $U$
Weakening of Updates

\[
\{x := 3 \parallel y := 3 - 5\} \leadsto \{x := \gamma_{>,1} \parallel y := \gamma_{<,2}\}
\]

where \(\gamma_{>,1}, \gamma_{<,2}\) new symbols

Replace an update \(\mathcal{U}\) with

\textit{weaker, i.e., more abstract} update \(\mathcal{U}'\)
Weakening of Updates

\{ x := 3 \parallel y := 3 - 5 \} \leadsto \{ x := \gamma_{>,1} \parallel y := \gamma_{<,2} \}

where $\gamma_{>,1}, \gamma_{<,2}$ new symbols

Replace an update $U$ with weaker, i.e., more abstract update $U'$

notion of weakening: replace integer expressions with new constants $\gamma_{a,\mathbb{Z}}$
Weakening of Updates: Calculus Rule

\[ \text{weakenUpdate} \quad \{\mathcal{U}\}(\bar{x} \doteq \bar{c}) \Rightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{x} \doteq \bar{c}) \Rightarrow \{\mathcal{U}'\}\varphi \Rightarrow \{\mathcal{U}\}\varphi \]

- right premiss: weakened update (implies conclusion)
- left premiss: soundness condition for weakening step
- \(\bar{x} \coloneqq (x_1, \ldots, x_n)\) assignable program variables in \(\mathcal{U}, \mathcal{U}'\)
- \(\bar{c} \coloneqq (c_1, \ldots, c_n)\) Skolem constants
  “value of \(\bar{x}\) under \(\mathcal{U}\) in concrete domain”
- \(\bar{\gamma}\) are the abstract symbols introduced in \(\mathcal{U}'\)
- \(\exists \bar{\gamma}.\psi\) abbreviates \(\exists \bar{y}.(\chi_{\bar{a}}(\bar{y}) \& \psi[\bar{\gamma}/\bar{y}])\)

The reachable concrete values \(\bar{c}\) of \(\bar{x}\) are contained in the abstracted \(\bar{\gamma}\)
\[ \Rightarrow \text{idx} \geq 0 \rightarrow [\text{gSum}=0; \text{W}](\text{idx} \div 0 \& \text{gSum} \geq 0) \]
\[
\text{idx} \geq 0 \implies \{\text{gSum := 0}\}[\text{while (idx>0) \{R\} }] (\text{idx \div 0 \& gSum \geq 0})
\]

\[
\implies \text{idx} \geq 0 \rightarrow [\text{gSum=0; W}] (\text{idx \div 0 \& gSum \geq 0})
\]
Example: Start of Symbolic Execution in Program Logic

Unwinding the loop?

Loop invariant?

\[ \text{id}x \geq 0 \Rightarrow \{ \text{gSum} := 0 \} \] \{ \text{while (id}x > 0) \} \{ R \} \{ \text{id}x \div 0 \& \text{gSum} \geq 0 \} \]

\[ \Rightarrow \text{id}x \geq 0 \rightarrow \{ \text{gSum}=0; \text{W} \}(\text{id}x \div 0 \& \text{gSum} \geq 0) \]
Example: Start of Symbolic Execution in Program Logic

Unwinding the loop?

\[ \text{Unwinding the loop?} \]

\[ \text{Loop invariant} \]

\[ \text{idx} \geq 0 \implies \{ \text{gSum} := 0 \}[\text{while (idx}>0) \{ \text{R} \} ](\text{idx} \div 0 \& \text{gSum} \geq 0) \]

\[ \implies \text{idx} \geq 0 \rightarrow [\text{gSum}=0; W](\text{idx} \div 0 \& \text{gSum} \geq 0) \]
Usual Rule for Introduction of Loop Invariant

\[ \Gamma \Rightarrow \{U\} Inv, \Delta \] (initially valid)

\[ g, Inv \Rightarrow [p] Inv \] (invariant)

\[ !g, Inv \Rightarrow [rest] \varphi \] (continue)

\[ \Gamma \Rightarrow \{U\}[while (g) \{p\}; rest] \varphi, \Delta \]
Usual Rule for Introduction of Loop Invariant

\[
\begin{align*}
\Gamma &\Rightarrow \{ U \} \text{Inv}, \Delta \quad \text{(initially valid)} \\
g, \text{Inv} &\Rightarrow [p] \text{Inv} \quad \text{(invariant)} \\
!g, \text{Inv} &\Rightarrow [\text{rest}] \varphi \quad \text{(continue)} \\
\hline
\Gamma &\Rightarrow \{ U \} [\text{while } (g) \{ p \}; \text{rest}] \varphi, \Delta
\end{align*}
\]

Observations

- Loop invariant soundly approximates program states reachable after loop execution
- Loop invariant must generally be provided by user
- Premisses (invariant), (continue) may not use proof context \( \Gamma, \Delta \)
Usual Rule for Introduction of Loop Invariant

\[ \Gamma \Rightarrow \{ U \} Inv, \Delta \quad \text{(initially valid)} \]
\[ g, Inv \Rightarrow [p] Inv \quad \text{(invariant)} \]
\[ ! g, Inv \Rightarrow [\text{rest}] \varphi \quad \text{(continue)} \]
\[ \Gamma \Rightarrow \{ U \} [\text{while } (g) \{ p \}; \text{rest}] \varphi, \Delta \]

Observations

- Loop invariant soundly approximates program states reachable after loop execution
- Loop invariant must generally be provided by user
- Premisses (invariant), (continue) may not use proof context \( \Gamma, \Delta \)

Idea

- Approximation of reachable states using update instead of formula
- Preserves proof context (updates scope their local variables)
- Computation of update that characterizes loop invariant: fixpoint algorithm based on incremental abstraction of update
\[ \Gamma \Rightarrow \{U\}[\text{while}\ (g)\ \{p\};\ \ldots]\varphi,\Delta \]

(initially valid)
(invariant)
(continue)
Loop Rule with Updates as Invariant

\[ \Gamma, \{U\} \not\!g \Rightarrow \{U\}[\ldots]\varphi, \Delta \]

\[ \Gamma \Rightarrow \{U\}[\text{while (g) \{p\}; \ldots]}\varphi, \Delta \]

- Approximate loop post-states with **abstract** update \( U' \)
  (replaces loop invariant)

(initially valid)
(invariant)
(continue)
Loop Rule with Updates as Invariant

\[ \Gamma, \{U\}(\bar{x} \dot{=} \bar{c}) \Rightarrow \exists \bar{\gamma}.\{U'\}(\bar{x} \dot{=} \bar{c}), \Delta \]  

(initially valid)

\( \text{invariant} \)

\( \text{continue} \)

\[ \Gamma, \{U'\}!g \Rightarrow \{U'\}[...]\varphi, \Delta \]

\[ \Gamma \Rightarrow \{U\}[\text{while (g) \{p\}; \ldots}]\varphi, \Delta \]

- **Soundness:** \( U' \) is logically weaker than \( U \)

- **Approximate loop post-states with abstract update \( U' \)**
  (replaces loop invariant)
Loop Rule with Updates as Invariant

\[ \Gamma, \{ U \} (\bar{x} \stackrel{\Delta}{\rightarrow} \bar{c}) \implies \exists \bar{\gamma}. \{ U' \} (\bar{x} \stackrel{\Delta}{\rightarrow} \bar{c}), \Delta \]
\[ \Gamma, \{ U' \} g, \{ U' \}[p] (\bar{x} \stackrel{\Delta}{\rightarrow} \bar{c}) \implies \exists \bar{\gamma}. \{ U' \} (\bar{x} \stackrel{\Delta}{\rightarrow} \bar{c}), \Delta \]
\[ \Gamma, \{ U' \} ! g \implies \{ U' \}[...] \varphi, \Delta \]
\[ \Gamma \implies \{ U \}[\text{while (g) \{p\}; \ldots}] \varphi, \Delta \] (initially valid)

(invariant)
(continue)

- Soundness: \( U' \) is logically weaker than \( U \)
- Whenever loop body executed with start state in \( U' \), then at most states in \( U' \) are reachable (invariant property)
- Approximate loop post-states with \textbf{abstract} update \( U' \) (replaces loop invariant)
Example: Application of Loop Rule with Updates

\[
\begin{align*}
\text{idx} \geq 0 & \implies \{ \text{gSum := 0} \}\left[ \text{while (idx>0)} \{ \text{R} \} \right] \text{Post} \\
& \implies \text{idx} \geq 0 \rightarrow [\text{gSum=}0;W](\text{idx} \div 0 \& \text{gSum} \geq 0) \tag{Post}
\end{align*}
\]
Example: Application of Loop Rule with Updates

(initially valid) $\text{idx} \geq 0, \{g\text{Sum} := 0\}(\text{idx} \div c_1 & g\text{Sum} \div c_2) \Rightarrow \exists \bar{\gamma}.\{U\}'(\text{idx} \div c_1 & g\text{Sum} \div c_2)$

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]
Example: Application of Loop Rule with Updates

(initially valid) \( \text{idx} \geq 0, \{ \text{gSum} := 0 \} (\text{idx} \div c_1 \& \text{gSum} \div c_2) \Rightarrow \exists \bar{\gamma}.\{U'\} (\text{idx} \div c_1 \& \text{gSum} \div c_2) \)

(invariant) \( \{U'\} \text{idx} > 0, \{U'\}[R](\bar{x} \div \bar{c}) \Rightarrow \exists \bar{\gamma}.\{U'\}(\bar{x} \div \bar{c}) \)

\[
\begin{align*}
(\text{initially valid}) \quad \text{idx} \geq 0 & \Rightarrow \{\text{gSum} := 0\}[\text{while} \ (\text{idx} > 0)\{R\} \ ]\text{Post} \\
\quad \Rightarrow \text{idx} \geq 0 & \rightarrow [\text{gSum}=0;W](\text{idx} \div 0 \& \text{gSum} \geq 0) \\
\end{align*}
\]
(initially valid) $\text{idx} \geq 0, \{\text{gSum} := 0\} (\text{idx} \div c_1 \& \text{gSum} \div c_2) \Rightarrow \exists \gamma.\{U\}' (\text{idx} \div c_1 \& \text{gSum} \div c_2)$

(invariant) $\{U\}' \text{idx} > 0, \{U\}'[R](\overline{x} \div \overline{c}) \Rightarrow \exists \gamma.\{U\}' (\overline{x} \div \overline{c})$

(continue) $\{U\}' ! \text{idx} > 0 \Rightarrow \{U\}'[\ ] Post$

\[
\begin{align*}
\text{(initially valid)} & \quad \text{(invariant)} & \quad \text{(continue)} \\
\text{idx} \geq 0 & \Rightarrow \{\text{gSum} := 0\}[\text{while (idx>0)}\{R\} ] Post & \\
\Rightarrow \text{idx} \geq 0 & \rightarrow [\text{gSum=0;W}] (\text{idx} \div 0 \& \text{gSum} \geq 0) \\
& \quad \text{Post}
\end{align*}
\]
Example: Application of Loop Rule with Updates

(initially valid) \( \text{idx} \geq 0, \{ \text{gSum} := 0 \}(\text{idx} \div c_1 \& \text{gSum} \div c_2) \Rightarrow \exists \gamma.\{U'\}(\text{idx} \div c_1 \& \text{gSum} \div c_2) \)

(invariant) \( \{U'\}\text{idx} > 0, \{U'\}[R](\bar{x} \div \bar{c}) \Rightarrow \exists \gamma.\{U'\}(\bar{x} \div \bar{c}) \)

(continue) \( \{U'\} \text{!idx} > 0 \Rightarrow \{U'\}[ ]\text{Post} \)

How to obtain \( U' \)?

(formula)

\( \text{idx} \geq 0 \Rightarrow \{ \text{gSum} := 0 \}[\text{while (idx}>0)\{R\} ]\text{Post} \)

\( \Rightarrow \text{idx} \geq 0 \rightarrow [\text{gSum}=0;W](\text{idx} \div 0 \& \text{gSum} \geq 0) \)

Post
Fixpoint approximation in side computation
Start: initial proof obligation for the loop

\[ \text{idx} \geq 0 \implies \{ \text{gSum} := 0 \} \\text{while (idx}\!>\!0\} \{ R \} \\text{Post} \]
Termination (ignored)

\[ \text{idx} \geq 0, \{ \text{gSum} := 0 \} (\text{idx} > 0) \implies \{ \text{gSum} := 0 \} [R;W]\text{Post} \]

\[ \text{idx} \geq 0 \implies \{ \text{gSum} := 0 \} [\text{while (idx} > 0) \{ R \} ] \text{Post} \]

1. Start: initial proof obligation for the loop
2. **Unwind** the loop \( W \) once
Computing Invariants

\[
\text{idx} > 0 \implies \{\text{gSum} := \text{idx} - 1 \parallel \text{idx} := \text{idx} - 1\}\text{[W]Post}
\]

\[
\vdash

\text{idx} > 0 \implies \{\text{gSum} := 0\}\text{[R;W]Post}
\]

Termination (ignored)

\[
\text{idx} \geq 0, \{\text{gSum} := 0\}(\text{idx} > 0) \implies \{\text{gSum} := 0\}\text{[R;W]Post}
\]

\[
\text{idx} \geq 0 \implies \{\text{gSum} := 0\}\text{{while (idx>0)}[R]}\text{[Post}
\]

1. **Start:** initial proof obligation for the loop

2. **Unwind** the loop \( W \) once and symbolically execute body \( R \)
Computing Invariants

\[ \text{idx} > 0 \Rightarrow \{ \text{gSum} := \text{idx} - 1 \| \text{idx} := \text{idx} - 1 \}\] [W] Post

\[ \vdash \]

\[ \text{idx} > 0 \Rightarrow \{ \text{gSum} := 0 \}\] [R; W] Post

**Termination (ignored)**

\[ \text{idx} \geq 0, \{ \text{gSum} := 0 \}(\text{idx} > 0) \Rightarrow \{ \text{gSum} := 0 \}\] [R; W] Post

\[ \text{idx} \geq 0 \Rightarrow \{ \text{gSum} := 0 \}\] [while (\text{idx} > 0) {R}] Post

---

1. **Start**: initial proof obligation for the loop
2. **Unwind** the loop \( W \) once and symbolically execute body \( R \)
3. **Compute** suitable abstraction of update in post-state
   - **compare** current state with state before most recent unwinding
Computing Invariants

\[
\text{idx} > 0 \implies \{gSum := \text{idx} - 1 || \text{idx} := \text{idx} - 1\}[W]\text{Post}
\]

\[
\vdash
\text{idx} > 0 \implies \{gSum := 0\}[R;W]\text{Post}
\]

**Termination** (ignored)

\[
\text{idx} \geq 0, \{gSum := 0\}(\text{idx} > 0) \implies \{gSum := 0\}[R;W]\text{Post}
\]

\[
\text{idx} \geq 0 \implies \{gSum := 0\}[\text{while (idx} > 0)\{R\}]\text{Post}
\]

1. **Start**: initial proof obligation for the loop
2. **Unwind** the loop $W$ once and symbolically execute body $R$
3. **Compute suitable abstraction of update in post-state**
   1. compare current state with state before most recent unwinding
   2. compare right-hand sides of updates for identical variables
Computing Invariants

\[ \text{id}x > 0 \implies \{ \text{gSum} := \gamma \geq, 1 \| \text{id}x := \gamma \geq, 2 \} \{ \text{W} \} \{ \text{Post} \} \]

\[ \text{id}x > 0 \implies \{ \text{gSum} := \text{id}x - 1 \| \text{id}x := \text{id}x - 1 \} \{ \text{W} \} \{ \text{Post} \} \]
\[ \vdots \]

\[ \text{id}x > 0 \implies \{ \text{gSum} := 0 \} \{ \text{R;} \text{W} \} \{ \text{Post} \} \]

\[ \text{id}x \geq 0, \{ \text{gSum} := 0 \} (\text{id}x > 0) \implies \{ \text{gSum} := 0 \} \{ \text{R;} \text{W} \} \{ \text{Post} \} \]

\[ \text{id}x \geq 0 \implies \{ \text{gSum} := 0 \} \{ \text{W} \} \{ \text{while (id}x > 0) \{ \text{R} \} \} \{ \text{Post} \} \]

1. **Start**: initial proof obligation for the loop
2. **Unwind** the loop \( \text{W} \) once and symbolically execute body \( \text{R} \)
3. **Compute suitable abstraction of update in post-state**
   1. compare current state with state before most recent unwinding
   2. compare right-hand sides of updates for identical variables
   3. abstraction of conflicting subterms using \( \sqcup \)-minimal value
Computing Invariants

\[ idx > 0 \implies \{ gSum := \gamma_{\geq,1} \parallel idx := \gamma_{\geq,2} \}[W]Post \]

\[ idx > 0 \implies \{ gSum := idx - 1 \parallel idx := idx - 1 \}[W]Post \]
\[ \vdots \]
\[ idx > 0 \implies \{ gSum := 0 \}[R;W]Post \]

Termination (ignored)

\[ idx \geq 0, \{ gSum := 0 \}(idx > 0) \implies \{ gSum := 0 \}[R;W]Post \]
\[ idx \geq 0 \implies \{ gSum := 0 \}[\text{while (idx>0)}\{R\}]Post \]

1. Start: initial proof obligation for the loop
2. Unwind the loop \( W \) once and symbolically execute body \( R \)
3. Compute suitable abstraction of update in post-state
4. Test for fixpoint (no new states are reachable post-state)
   can be expressed as logic formula \( \implies \) automated theorem prover
idx ≥ 0 ⇒ \{gSum := γ_{≥,1} \parallel idx := γ_{≥,2}\}[W]Post

\vdots

idx ≥ 0 ⇒ \{gSum := 0\}[while (idx>0){R}]Post

1 No fixpoint reached yet → unwind loop once more
Termination (ignored)

\[ \begin{align*}
\text{idx} \geq 0, \{ \text{gSum} := 0 \} (\text{idx} > 0) & \implies \{ \text{gSum} := 0 \} [R; W] \text{Post} \\
\text{idx} \geq 0 & \implies \{ \text{gSum} := \gamma_{\geq,1} \| \text{idx} := \gamma_{\geq,2} \} [W] \text{Post} \\
\vdots \text{Post} \\
\text{idx} \geq 0 & \implies \{ \text{gSum} := 0 \} [\text{while (idx>0)} \{ R \}] \text{Post}
\end{align*} \]

1. **No fixpoint reached yet → unwind loop once more**
2. **Symbolically execute loop body**
Computing Invariants Cont’d

\[ \ldots \implies \{ gSum := \gamma_{\geq,3} \parallel idx := \gamma_{\geq,4} \} [W] Post \]

\[ \vdots \]

\[ idx \geq 0, \{ gSum := 0 \}(idx > 0) \implies \{ gSum := 0 \}[R;W] Post \]

\[ idx \geq 0 \implies \{ gSum := \gamma_{\geq,1} \parallel idx := \gamma_{\geq,2} \} [W] Post \]

\[ \vdots \]

\[ idx \geq 0 \implies \{ gSum := 0 \}[while (idx>0){R}] Post \]

1. **No** fixpoint reached yet \(\rightarrow\) unwind loop once more
2. Symbolically execute loop body
3. Compute abstract update
Computing Invariants Cont’d

found invariant update

\[ \ldots \implies \{ gSum := \gamma_{\geq,3} \parallel idx := \gamma_{\geq,4} \} W \] Post

Termination (ignored)

\[ idx \geq 0, \{ gSum := 0 \}(idx > 0) \implies \{ gSum := 0 \}[R;W] Post \]

\[ idx \geq 0 \implies \{ gSum := \gamma_{\geq,1} \parallel idx := \gamma_{\geq,2} \} W \] Post

\[ idx \geq 0 \implies \{ gSum := 0 \}[\text{while (idx>0)}\{R\}] Post \]

1. **No fixpoint reached yet \(\rightarrow\) unwind loop once more**
2. **Symbolically execute loop body**
3. **Compute abstract update**
4. **Test fixpoint**
Example: Application of Loop Rule with Updates

\[(\text{initially valid}) \quad \text{idx} \geq 0, \{g\text{Sum} := 0\}(\text{idx} \div c_1 \& g\text{Sum} \div c_2) \Rightarrow \exists \bar{\gamma}.\{U'\}(\text{idx} \div c_1 \& g\text{Sum} \div c_2)\]

\[(\text{invariant}) \quad \{U'\}\text{idx} > 0, \{U'\}[R](\bar{x} \div \bar{c}) \Rightarrow \exists \bar{\gamma}.\{U'\}(\bar{x} \div \bar{c})\]

\[(\text{continue}) \quad \{U'\} \neg \text{idx} > 0 \Rightarrow \{U'\}[ \text{Post}]\]

How to obtain \(U'\)?

\[
\begin{align*}
\text{(initially valid)} & \quad \text{(invariant)} & \quad \text{(continue)} \\
\text{idx} \geq 0 & \Rightarrow \{g\text{Sum} := 0\}[\text{while (idx>0)\{R\}} \quad \text{Post} \\
\Rightarrow \text{idx} \geq 0 & \rightarrow [g\text{Sum}=0;W](\text{idx} \div 0 \& g\text{Sum} \geq 0) \quad \text{Post}
\end{align*}
\]
Example: Application of Loop Rule with Updates

(initially valid) \( \text{idx} \geq 0, \{ \text{gSum} := 0 \} (\text{idx} \div c_1 \& \text{gSum} \div c_2) \Rightarrow \exists \gamma.\{ \mathcal{U}' \}(\text{idx} \div c_1 \& \text{gSum} \div c_2) \)

(invariant) \( \{ \mathcal{U}' \} \text{idx} > 0, \{ \mathcal{U}' \}[R](\bar{x} \div \bar{c}) \Rightarrow \exists \gamma.\{ \mathcal{U}' \}(\bar{x} \div \bar{c}) \)

(continue) \( \{ \mathcal{U}' \} ! \text{idx} > 0 \Rightarrow \{ \mathcal{U}' \}[\ ] \text{Post} \)

How to obtain \( \mathcal{U}' \)?

\( \text{gSum} := \gamma_{\geq,3} \parallel \text{idx} := \gamma_{\geq,4} \)

(continuously valid) \( \text{idx} \geq 0 \Rightarrow \{ \text{gSum} := 0 \}[\text{while} (\text{idx} > 0) \{ \text{R} \} ] \text{Post} \)

\( \Rightarrow \text{idx} \geq 0 \rightarrow [\text{gSum}=0;W](\text{idx} \div 0 \& \text{gSum} \geq 0) \)

Post
Example: Final Step

\[ \text{init.}(\text{inv.}) \]

\[ \begin{align*}
\ast & \ast \\
\vdots & \vdots \\
\text{(init.)(inv.)} & \text{(continue)} \\
\text{id}x \geq 0 \iff \{g\text{Sum} := 0\}[\text{while (idx>0)}\{R\} ]\text{Post} \\
\implies \text{id}x \geq 0 \rightarrow [g\text{Sum}=0; W](\text{id}x \div 0 \& g\text{Sum} \geq 0) \tag{Post}
\end{align*} \]
Example: Final Step

\[ \begin{align*}
\text{(init.)}(\text{inv.}) & \quad 
\{ \text{gSum} := \gamma_{\geq,3} \parallel \text{idx} := \gamma_{\geq,4} \} \quad \text{!idx} > 0 \implies \\
& \quad \{ \text{gSum} := \gamma_{\geq,3} \parallel \text{idx} := \gamma_{\geq,4} \} [ ] (\text{idx} \div 0 \& \text{gSum} \geq 0) \\
\implies idx \geq 0 \implies \{ \text{gSum} := 0 \} [ \text{while (idx>0)} \{ \text{R} \} \] Post \\
& \quad \implies idx \geq 0 \implies \{ g\text{Sum}=0; W \} (\text{idx} \div 0 \& g\text{Sum} \geq 0) \\
& \quad \quad \text{Post}
\end{align*} \]
Example: Final Step

\[ \forall \gamma \geq 4 > 0 \Rightarrow (\gamma \geq 4 \div 0 \land \gamma \geq 3 \geq 0) \]

\[ \vdots \]

\[ \{gSum := \gamma \geq 3 \parallel idx := \gamma \geq 4\} !idx > 0 \Rightarrow \{gSum := \gamma \geq 3 \parallel idx := \gamma \geq 4\}[ ](idx \div 0 \land gSum \geq 0) \]

\[ \text{Post} \]

\[ \Rightarrow idx \geq 0 \Rightarrow [gSum=0;W](idx \div 0 \land gSum \geq 0) \]

\[ \text{Post} \]
Example: Final Step

\[ \ast \]

\[ \vdash \text{(axioms for } \gamma_{\geq, \mathbb{Z}}) \]

\[ \vdash \gamma_{\geq, 4} > 0 \implies (\gamma_{\geq, 4} \div 0 \land \gamma_{\geq, 3} \geq 0) \]

\[ \vdash \]

\[ \{ \text{gSum} := \gamma_{\geq, 3} \parallel \text{idx} := \gamma_{\geq, 4} \} \text{!idx} > 0 \implies \{ \text{gSum} := \gamma_{\geq, 3} \parallel \text{idx} := \gamma_{\geq, 4} \}[ ](\text{idx} \div 0 \land \text{gSum} \geq 0) \]

\[ \text{idx} \geq 0 \implies \{ \text{gSum} := 0 \}[\text{while (idx}>0)\{R\}] \] Post

\[ \Rightarrow \text{idx} \geq 0 \rightarrow [\text{gSum}=0;W](\text{idx} \div 0 \land \text{gSum} \geq 0) \]

Post
Where We Are Now

1. Implementation of abstraction in program logic ✅
   - logic signature for abstract domains and abstraction function
   - calculus for abstraction steps and execution of abstract programs
   - soundness conditions

2. Model (information flow) type system as abstract domain
Semantics

State $s$

$x \mapsto 3$

$y \mapsto 5$

Signature (minimalist version)

- program variables $PV := \{x, y, z, \ldots\}$
- $\ldots$
A Logic for Analysing Information Flow

Semantics

State $s$

- $x \mapsto 3 \quad x^{\text{dep}} \mapsto \{x\}$
- $y \mapsto 5 \quad y^{\text{dep}} \mapsto \{z\}$

Signature (minimalist version)

- program variables $\text{PV} := \{x, y, z, \ldots\}$
- $\text{PV}^{\text{dep}} := \{x^{\text{dep}} | x \in \text{PV}\}$
- $\ldots$
Semantics

State $s$

$\begin{align*}
x & \mapsto 3 \\
x^{\text{dep}} & \mapsto \{x\} \\
y & \mapsto 5 \\
y^{\text{dep}} & \mapsto \{z\}
\end{align*}$

Signature (minimalist version)

- program variables $PV := \{x, y, z, \ldots\}$
- $PV^{\text{dep}} := \{x^{\text{dep}} | x \in PV\}$
- interpreted functions for set theory fragment $\emptyset, \{x\}$ f.a. $x \in PV, \cup$
- $\ldots$
Semantics

State $s$

\[
\begin{align*}
x & \mapsto 3 \quad x^{\text{dep}} & \mapsto \{x\} \\
y & \mapsto 5 \quad y^{\text{dep}} & \mapsto \{z\}
\end{align*}
\]

Signature (minimalist version)

- program variables $PV := \{x, y, z, \ldots\}$
- $PV^{\text{dep}} := \{x^{\text{dep}} | x \in PV\}$
- interpreted functions for set theory fragment $\emptyset, \{x\}$ f.a. $x \in PV, \cup$
- interpreted predicate $\subseteq$
- \ldots
Symbolic Execution of Assignment:

post-state differentiates only values, no dependencies!

More rules need to be changed: if, unwindLoop (implicit dependencies)
Symbolic Execution and Variable Dependencies

- Assignment $x \mapsto 3 \mid x^\text{dep} \mapsto \{x\} \quad y \mapsto 5 \mid y^\text{dep} \mapsto \{z\}$

- Assignment $x \mapsto 5 \mid x^\text{dep} \mapsto \{\}$ \quad $y \mapsto 5 \mid y^\text{dep} \mapsto \{z\}$

- Assignment $x \mapsto 5 \mid x^\text{dep} \mapsto \{z\} \quad y \mapsto 5 \mid y^\text{dep} \mapsto \{z\}$

Assignment $^\text{dep}$

$\Gamma \Rightarrow U\{x := e \parallel x^\text{dep} := \text{dep}(e)\}[\text{rest}]\phi, \Delta$

$\Gamma \Rightarrow U[x=e; \text{rest}]\phi, \Delta$

Tracking dependencies in separate variables $v^\text{dep}$
Symbolic Execution and Variable Dependencies

Assignment dependence:

\[ \Gamma \Rightarrow \mathcal{U}\{x := e \parallel x^{\text{dep}} := \text{dep}(e)\}[\text{rest}] \phi, \Delta \]

\[ \Gamma \Rightarrow \mathcal{U}[x=e; \text{rest}] \phi, \Delta \]

Tracking dependencies in separate variables \(n^{\text{dep}}\)

More rules need to be changed: if, unwindLoop (implicit dependencies)
Example: Proving Non-Interference

```plaintext
l1=0; l2=0;
while (h<0) {
    l2=l2+1;
    h=h+1;
}
if (l2<0) {l1=1;}
```

l1, l2 public variables, h secret variable
Example: Proving Non-Interference

```java
l1=0; l2=0;
while (h<0) {
    l2=l2+1;
    h=h+1;
}
if (l2<0) {l1=1;}
```

11, 12 public variables, h secret variable

**Non-interference:** (e.g., 11 does not depend on h)

Proof obligation to be proven in program logic:

\[
11^{\text{dep}} \doteq \{11\}, \quad 12^{\text{dep}} \doteq \{12\}, \quad h^{\text{dep}} \doteq \{h\} \implies \left[ 11=0; 12=0; \ldots \right] (11^{\text{dep}} \subseteq \{11\} \cup \{12\})
\]

Proof is fully automatic; this cannot be shown with type analysis!
Example: a Suitable Abstract Domain

State

\[ l_1 \mapsto 0 \]
\[ l_2 \mapsto 0 \]
\[ \ldots \]

Abstract Domain

\[ \emptyset \]
\[ \neg 0 \]
\[ \text{pos} \]
\[ \leq \]
\[ \geq \]
\[ \top \]

\[ \gamma(\emptyset) = \emptyset, \gamma(\text{Low}) = 2 \{ l_1, l_2 \}, \gamma(\text{High}) = 2 \{ h \}, \gamma(\top) = 2 \text{PV} \]
Example: a Suitable Abstract Domain

\[
\gamma(\emptyset) = \emptyset, \quad \gamma(\text{Low}) = 2^{\{11, 12\}}, \quad \gamma(\text{High}) = 2^h, \quad \gamma(T_{dep}) = 2^{PV}
\]
Summary

1. Integration of a natural notion of abstraction into program logic
2. Modeling of type inference as abstract interpretation
Summary

1. Integration of a natural notion of abstraction into program logic
2. Modeling of type inference as abstract interpretation

Important Points

- Abstraction of symbolic state expressions ("updates"), not of arbitrary programs: **scales up to real programming languages**
- Abstraction ∼ logic weakening
- Variable-wise abstraction on demand during symbolic execution
- Fixpoint algorithm automatically computes candidate for loop invariant
- Suitable for state-of-art type systems as used in security analysis
- Abstract symbolic execution stronger than type inference
- Soundness of calculus rules has been proven (FMCO 2008 Post-Proc.)
<table>
<thead>
<tr>
<th>Extended Type Systems</th>
<th>Abstract Interpretation</th>
<th>Program Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>- fully automatic</td>
<td>- fully automatic</td>
<td>- interactive or annotations</td>
</tr>
<tr>
<td>- efficient</td>
<td>- terminating</td>
<td>- high expressivity</td>
</tr>
<tr>
<td>- low</td>
<td>- approximation</td>
<td>- logic-based specification</td>
</tr>
<tr>
<td>- expressivity</td>
<td>- fixed degree of precision</td>
<td>- real-world case studies</td>
</tr>
<tr>
<td>- depend on target language</td>
<td>- commercial usage</td>
<td></td>
</tr>
<tr>
<td>- mostly prototypes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Type Systems
- fully automatic
- efficient
- low expressivity
- depend on target language
- mostly prototypes

Abstract Interpretation
- fully automatic
- terminating
- approximation
- fixed degree of precision
- commercial usage

Program Logic
- interactive or annotations
- high expressivity
- logic-based specification
- real-world case studies

Program Logic + Abstraction
- fully automatic, terminating
- high expressivity
- logic-based specification
- real-world case studies
Future Work

- In progress: extension to sequential Java
  - extend abstraction to arrays, object types
  - reuse existing logic-based symbolic execution for Java in KeY
- Increased precision of the analysis (e.g., 1 = h−h
  - proof of soundness requires trace semantics
- More complex abstract domains
- Integration with abstract interpretation tools
- Implementation (prototype ready), evaluation