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

Motivation

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 $\rightarrow$ 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

"No silver bullet" — Concertation is essential!

Tools: Decisive Factor for Effectivity & Reach of Methods

Established Software Production Tools

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

Tools: Decisive Factor for Effectivity & Reach of Methods

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

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

Type Systems

"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

Type Systems

"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

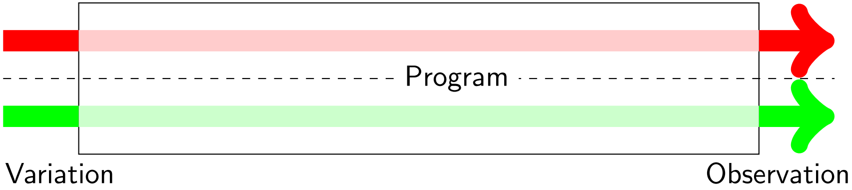
Extended Type Systems

...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

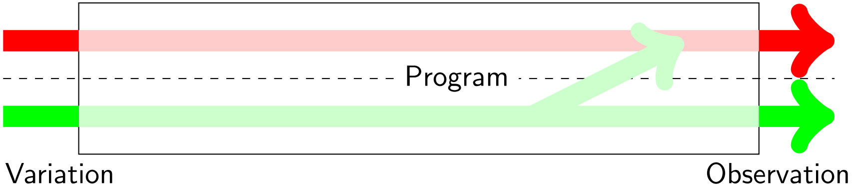
A Type System for Security Analysis

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



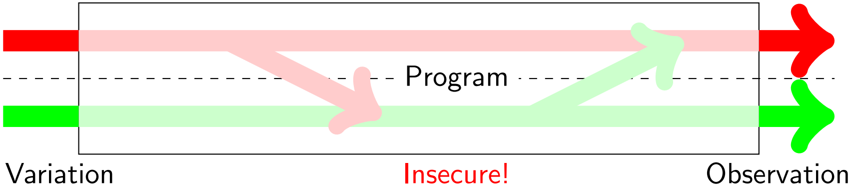
A Type System for Security Analysis

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



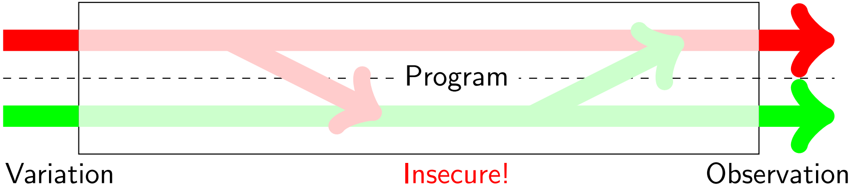
A Type System for Security Analysis

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

  `if (secret) public = 0 else public = 1;` insecure
- Type checking in polynomial time
- Approximation (false positives): wrongly classified as insecure

  `if (secret) public = 0 else public = 0;` sicher
- Implemented for Spark, problematic for JAVA (aliasing)

Abstract Interpretation

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 ∞ 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 ∞ 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

$\boxed{\texttt{gSum=0, index=}\geq}$

```
gSum = 0;
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 ∞ 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

```
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 ∞ 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

```
gSum = 0;
while (index>0) {   gSum=0, index=>
  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 ∞ 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

```
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 ∞ 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

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

Abstract Interpretation

All configurations at each program location with abstract values

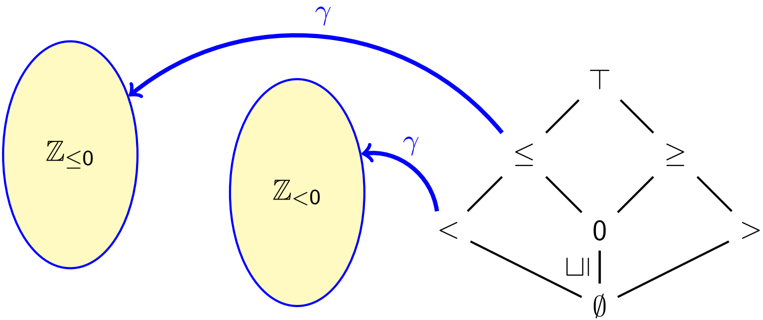
- Due to Cousot & Cousot, 1977ff
- Commercial use, e.g. in A3XX software
- Termination when no ∞ ascending chains in abstract domain
- Abstract interpretation of all operators in target program
- Soundness: in abstract domain reachable states approximate all reachable states in concrete computations

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

Abstract Interpretation: Value Domains as Lattices

All configurations at each program location with abstract values

All configurations at each program location with abstract values

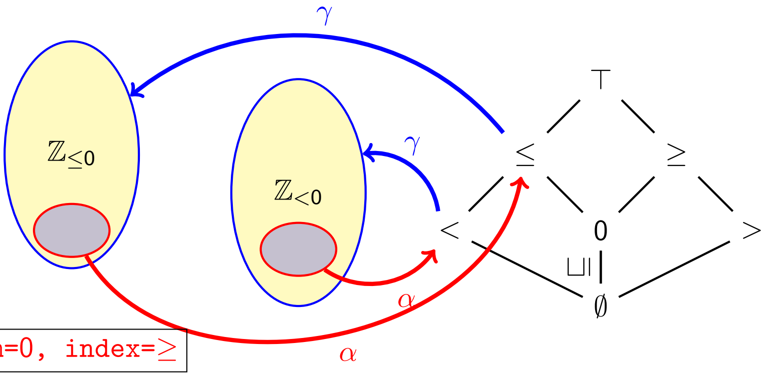


$\boxed{\texttt{gSum=0, index=}\geq}$

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

Abstract Interpretation: Value Domains as Lattices

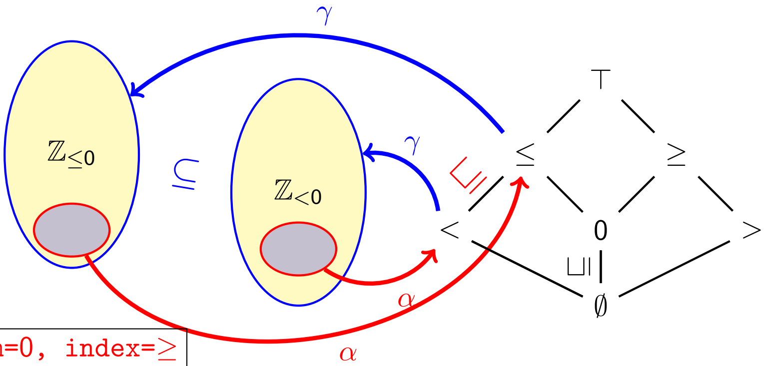
All configurations at each program location with abstract values



gSum=0, index=$\geq$

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

All configurations at each program location with abstract values



gSum=0, index=$\geq$

```
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:
$\mathcal{U} \ = \ \{\mathtt{gSum} := g_0 + i_0 \mid\mid \mathtt{index} := i_0 - 1\}$

Formulas Partial correctness of program p in start state $\mathcal{U}$ wrt. post condition: $\mathcal{U}[\mathrm{p}]\mathrm{post}$

Example $\mathtt{index} \geq 0 \mathrel{->} [\mathtt{Gauss}](\mathtt{index} \doteq 0 \ \&\ \mathtt{gSum} \geq 0)$

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:
$\mathcal{U} \;=\; \{\mathtt{gSum} := g_0 + i_0 \;||\; \mathtt{index} := i_0 - 1\}$

Formulas Partial correctness of program p in start state $\mathcal{U}$ wrt. post condition: $\mathcal{U}[\mathrm{p}]\mathrm{post}$

Example $\mathtt{index} \geq 0 \;\texttt{->}\; [\mathtt{Gauss}](\mathtt{index} \doteq 0 \;\&\; \mathtt{gSum} \geq 0)$

Symbolic Execution in Program Logic

Partial Correctness Formulas

- Pre $->$ $[\mathtt{P}]$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:

  $$s \models [\mathtt{P}]\psi \quad \text{iff} \quad \text{for all } t\text{: if } s[\![P]\!]t \text{ then } t \models \psi$$

Symbolic Execution in Program Logic

Partial Correctness Formulas

- Pre -> $[\mathtt{P}]$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:
  $s \models [\mathtt{P}]\psi$ iff for all t: if $s[\![P]\!]t$ then $t \models \psi$

Updates (generalized explicit substitutions)

- $\mathcal{U} = \{\mathtt{v}_1 := t_1 \mid\mid \cdots \mid\mid \mathtt{v}_n := t_n\}$, $\mathtt{v}_i$ program variables, t_i f.-o. terms
- $\mathcal{U}\varphi$ when φ f.-o. formula: like substitution
- $\mathcal{U}\varphi$ when φ program formula: delay application
  - $\mathcal{U}$ accumulate in front of program formulas during symbolic execution
  - Updates simplified constantly (rewrite system with normal form)
- Extensions for object access, arrays, conditional terms

Symbolic Execution in Program Logic

Partial Correctness Formulas

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

$$s \models [\mathtt{P}]\psi \quad \text{iff} \quad \text{for all } t\text{: if } s[\![P]\!]t \text{ then } t \models \psi$$

$$\text{assignment} \quad \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{x} := \mathtt{e}\}[\mathtt{rest}]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x=e;\ rest}]\phi, \Delta}$$

where $\mathtt{e}$ is a simple expression (no side effects, method calls)

Symbolic Execution in Program Logic Cont'd

Symbolic Execution of Conditional Statement

$$\text{if}\ \frac{\Gamma, \mathcal{U}\mathtt{b} \Longrightarrow \mathcal{U}[\mathtt{p;\ \ rest}]\phi, \Delta \qquad \Gamma, \mathcal{U}!\mathtt{b} \Longrightarrow \mathcal{U}[\mathtt{q;\ \ rest}]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{\textbf{if}\ (b)\ \{\ p\ \}\ \textbf{else}\ \{\ q\ \};\ rest}]\phi, \Delta}$$

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

Symbolic Execution in Program Logic Cont'd

Symbolic Execution of Conditional Statement

$$\text{if}\ \frac{\Gamma, \mathcal{U}\mathtt{b} \Longrightarrow \mathcal{U}[\mathtt{p;\ rest}]\phi, \Delta \qquad \Gamma, \mathcal{U}!\mathtt{b} \Longrightarrow \mathcal{U}[\mathtt{q;\ rest}]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{\textbf{if}\ (b)\ \{\ p\ \}\ \textbf{else}\ \{\ q\ \};\ rest}]\phi, \Delta}$$

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

Symbolic Execution of Loop Statement

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{\textbf{if}\ (b)\ \{\ p;\ \textbf{while}\ (b)\ p\};\ rest}]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{\textbf{while}\ (b)\ \{p\};\ rest}]\phi, \Delta}$$

Without fixed loop bound: no finite symbolic execution

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

Type Systems, Abstract Interpretation, Program Logic

Extended Type Systems

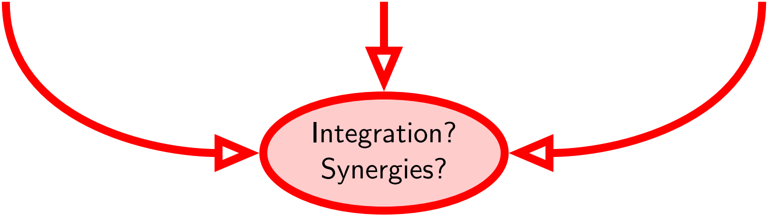
- fully automatic
- efficient
- low expressivity
- depend on target language
- mostly prototypes

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



How are these three techniques related?

Type Systems, Abstract Interpretation, Symbolic Execution

How are these three techniques related?

1. View a calculus for type inference as abstract interpretation

```
x = (x % 2 * y)* z - 327;
```

↓Abstraction

```
x = (x, y, z);
```

Complex assignment

concrete/symbolic execution

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

Type Systems, Abstract Interpretation, Symbolic Execution

How are these three techniques related?

1. View a calculus for type inference as abstract interpretation

```
x = (x % 2 * y)* z - 327;
```

↓Abstraction

```
x = (x, y, z);
```

Complex assignment concrete/symbolic execution

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

Type Systems, Abstract Interpretation, Symbolic Execution

How are these three techniques related?

1. View a calculus for type inference as abstract interpretation

```
x = (x % 2 * y)* z - 327;
```

$\Downarrow$ Abstraction

```
x = (x, y, z);
```

Complex assignment concrete/symbolic execution

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

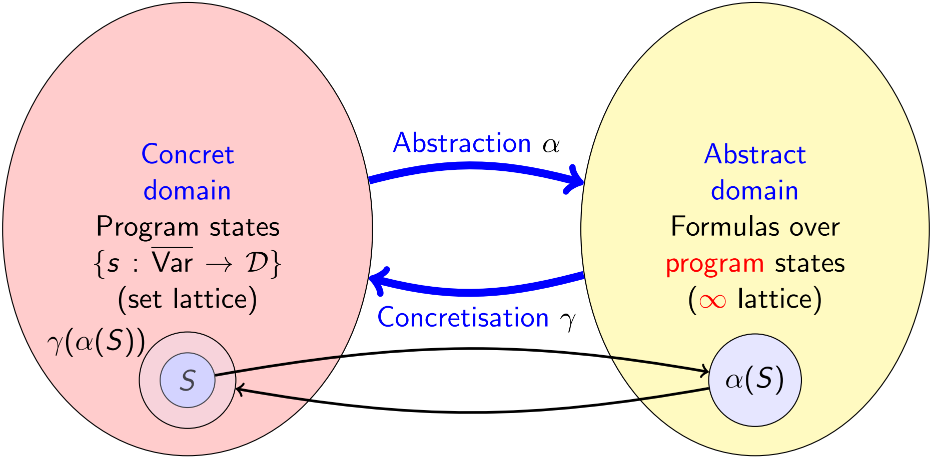
- Abstract interpretation compatible with symbolic execution in program logic
- Conceptual integration of all three techniques possible!

Realisation

Realisation in reverse order:

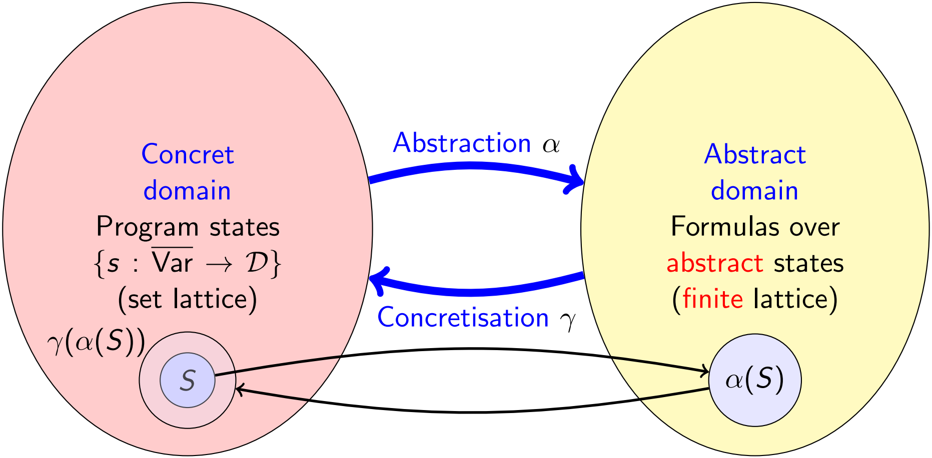
1. Implementation of abstraction in program logic
   - logic signature for abstract domains and abstraction function
   - calculus for abstraction steps and execution of abstract programs
   - soundness conditions
2. Model (information flow) type system as abstract domain

Abstraction and Symbolic Execution



Symbolic execution as infinite, abstract Domain

Abstraction and Symbolic Execution



Symbolic execution in finite, abstract Domain

Logic Representation of Abstract Domains

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 χ_a

Logic Representation of Abstract Domains

Signature (first-order) for abstract Domain $\mathcal{A}$

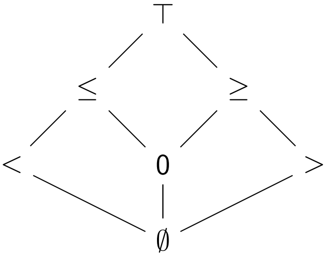
- for each $a \in \mathcal{A}$ and $z \in \mathbb{Z}$ there is a constant $\gamma_{a,z}$
- for each $a \in \mathcal{A}$ there is a unary predicate χ_a

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 χ_a are characteristic sets of the abstract values"

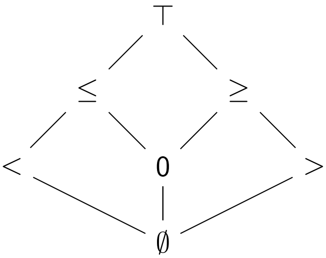
Example: Abstract Domain

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



Example: Abstract Domain

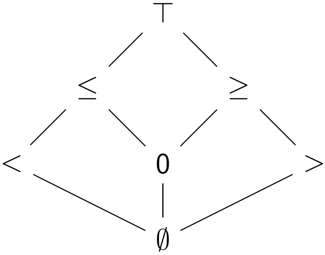
```
gSum = 0;
while (index>0) {
  index = index - 1;
  gSum  = gSum + index;
}
```



$a \in \mathcal{A}$	$\chi_a(x)$	$\gamma_{a,\mathbb{Z}}$	axioms for the new symbols
$<$	$\chi_<$	$\gamma_{<,1}, \ldots$	$\forall x.(\chi_<(x) \leftrightarrow x < 0),\ \chi_<(\gamma_{<,\mathbb{Z}})$

Example: Abstract Domain

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



$a \in \mathcal{A}$	$\chi_a(x)$	$\gamma_{a,\mathbb{Z}}$	axioms for the new symbols
$<$	$\chi_<$	$\gamma_{<,1}, \ldots$	$\forall x.(\chi_<(x) \leftrightarrow x < 0)$, $\chi_<(\gamma_{<,\mathbb{Z}})$
0	χ_0	$\gamma_{0,1}, \ldots$	$\forall x.(\chi_0(x) \leftrightarrow x \doteq 0)$, $\chi_0(\gamma_{0,\mathbb{Z}})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Abstract Deduction in Program Logic

Abstraction of target program

Replace

$$\Gamma \Longrightarrow [\mathtt{p}]\varphi$$

with

$$\alpha(\Gamma) \Longrightarrow [\alpha(\mathtt{p})]\alpha(\varphi)$$

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

Abstraction of updates

Replace

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

with

$$\Gamma \Longrightarrow \alpha(\mathcal{U})[\mathrm{p}]\varphi$$

- Theory of abstraction of updates is sufficient

Abstraction of updates

Replace

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

with

$$\Gamma \Longrightarrow \alpha(\mathcal{U})[\mathrm{p}]\varphi$$

- Theory of abstraction of updates is sufficient

Decisive insight: abstraction $\sim$ logic weakening (implication)

Weakening of Updates

$$\{x := 3 \parallel y := 3 - 5\} \quad \rightsquigarrow$$

Replace an update $\mathcal{U}$

Weakening of Updates

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

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

Replace an update $\mathcal{U}$
with
weaker, i.e., more abstract update $\mathcal{U}'$

Weakening of Updates

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

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

Replace an update $\mathcal{U}$
with
weaker, i.e., more abstract update $\mathcal{U}'$
notion of weakening: replace integer expressions with new constants $\gamma_{a,\mathbb{Z}}$

Weakening of Updates: Calculus Rule

$$\text{weakenUpdate}\ \frac{\{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}) \qquad \Longrightarrow \{\mathcal{U}'\}\varphi}{\Longrightarrow \{\mathcal{U}\}\varphi}$$

- right premiss: weakened update (implies conclusion)
- left premiss: soundness condition for weakening step
- $\bar{\mathtt{x}} := (\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n})$ assignable program variables in $\mathcal{U}, \mathcal{U}'$
- $\bar{c} := (c_1, \ldots, c_n)$ Skolem constants
  "value of $\bar{\mathtt{x}}$ 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 $\bar{\mathtt{x}}$ are contained in the abstracted $\bar{\gamma}$

Example: Start of Symbolic Execution in Program Logic

$$\Longrightarrow \texttt{idx} \geq \texttt{0} \;\texttt{->}\; [\texttt{gSum=0;W}](\texttt{idx} \doteq \texttt{0} \;\&\; \texttt{gSum} \geq \texttt{0})$$

Example: Start of Symbolic Execution in Program Logic

$$\frac{\texttt{idx} \geq 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{while (idx>0) \{R\} }](\texttt{idx} \doteq 0 \;\&\; \texttt{gSum} \geq 0)}{\Longrightarrow \texttt{idx} \geq 0 \;\texttt{->}\; [\texttt{gSum=0;W}](\texttt{idx} \doteq 0 \;\&\; \texttt{gSum} \geq 0)}$$

Unwinding the loop?

Loop invariant?

$$\frac{\texttt{idx} \geq 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{while (idx>0) \{R\} }](\texttt{idx} \doteq 0 \ \& \ \texttt{gSum} \geq 0)}{\Longrightarrow \texttt{idx} \geq 0 \ \texttt{->} \ [\texttt{gSum=0;W}](\texttt{idx} \doteq 0 \ \& \ \texttt{gSum} \geq 0)}$$

Example: Start of Symbolic Execution in Program Logic

Unwinding the loop?

Loop invariant

$$\frac{\texttt{idx} \geq 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{while (idx>0) \{R\} }](\texttt{idx} \doteq 0 \ \&\ \texttt{gSum} \geq 0)}{\Longrightarrow \texttt{idx} \geq 0 \ \texttt{->}\ [\texttt{gSum=0;W}](\texttt{idx} \doteq 0 \ \&\ \texttt{gSum} \geq 0)}$$

Usual Rule for Introduction of Loop Invariant

$$\frac{\begin{array}{c}\Gamma \Longrightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta \\ g, \mathit{Inv} \Longrightarrow [\mathrm{p}]\mathit{Inv} \\ !\, g, \mathit{Inv} \Longrightarrow [\mathtt{rest}]\varphi\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while}\ \mathtt{(}g\mathtt{)}\ \mathtt{\{p\};}\ \mathtt{rest}]\varphi, \Delta}
\quad
\begin{array}{l}\text{(initially valid)} \\ \text{(invariant)} \\ \text{(continue)}\end{array}$$

Usual Rule for Introduction of Loop Invariant

$$
\frac{\begin{array}{ll} \Gamma \Longrightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta & \text{(initially valid)} \\ g, \mathit{Inv} \Longrightarrow [\mathtt{p}]\mathit{Inv} & \text{(invariant)} \\ !\,g, \mathit{Inv} \Longrightarrow [\mathtt{rest}]\varphi & \text{(continue)} \end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while}\ (g)\ \{\mathtt{p}\};\ \mathtt{rest}]\varphi, \Delta}
$$

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 Γ, Δ

Usual Rule for Introduction of Loop Invariant

$$\frac{\begin{array}{ll} \Gamma \Longrightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta & \text{(initially valid)} \\ g, \mathit{Inv} \Longrightarrow [\mathtt{p}]\mathit{Inv} & \text{(invariant)} \\ !\,g, \mathit{Inv} \Longrightarrow [\mathtt{rest}]\varphi & \text{(continue)} \end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};\ rest}]\varphi, \Delta}$$

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 Γ, Δ

Idea

- Approximation of reachable states using update instead of formula
- Preserves proof context (updates scope their local variables)
- Computation of update that charactarizes loop invariant: fixpoint algorithm based on incremental abstraction of update

Loop Rule with Updates as Invariant

$$\frac{}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; \ldots}]\varphi, \Delta} \quad \begin{array}{l}\text{(initially valid)}\\ \text{(invariant)}\\ \text{(continue)}\end{array}$$

Loop Rule with Updates as Invariant

$$\frac{\Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; \ldots}]\varphi, \Delta}$$

(initially valid)
(invariant)
(continue)

- Approximate loop post-states with abstract update $\mathcal{U}'$ (replaces loop invariant)

Loop Rule with Updates as Invariant

$$\frac{\begin{array}{c}\Gamma, \{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{\mathtt{c}}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{\mathtt{c}}), \Delta \\ \\ \Gamma, \{\mathcal{U}'\}\, !\, g \Longrightarrow \{\mathcal{U}'\}[...]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while}\ \ (g)\ \ \{\mathtt{p}\};\ \ \ldots]\varphi, \Delta}$$

(initially valid)
(invariant)
(continue)

- Soundness: $\mathcal{U}'$ is logically weaker than $\mathcal{U}$
- Approximate loop post-states with abstract update $\mathcal{U}'$ (replaces loop invariant)

Loop Rule with Updates as Invariant

$$\frac{\begin{array}{cl}\Gamma, \{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{\mathsf{c}}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{\mathsf{c}}), \Delta & \text{(initially valid)} \\ \Gamma, \{\mathcal{U}'\}g,\ \{\mathcal{U}'\}[p](\bar{\mathtt{x}} \doteq \bar{\mathsf{c}}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{\mathsf{c}}), \Delta & \text{(invariant)} \\ \Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[...]\varphi, \Delta & \text{(continue)}\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};\ \ldots}]\varphi, \Delta}$$

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

Example: Application of Loop Rule with Updates

$$\frac{\texttt{idx} \geq 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{while (idx>0)\{R\} }]\mathsf{Post}}{\Longrightarrow \texttt{idx} \geq 0 \texttt{ -> } [\texttt{gSum=0;W}]\underbrace{(\texttt{idx} \doteq 0 \ \&\ \texttt{gSum} \geq 0)}_{\mathsf{Post}}}$$

Example: Application of Loop Rule with Updates

$$\text{(initially valid)}\ \mathtt{idx} \geq 0, \{\mathtt{gSum} := 0\}(\mathtt{idx} \doteq \mathtt{c}_1 \ \&\ \mathtt{gSum} \doteq \mathtt{c}_2) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\mathtt{idx} \doteq \mathtt{c}_1 \ \&\ \mathtt{gSum} \doteq \mathtt{c}_2)$$

$$\dfrac{\dfrac{\text{(initially valid)}}{\quad} \qquad \mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ (idx>0)\{R\}}\ ]\mathrm{Post}}{\Longrightarrow \mathtt{idx} \geq 0 \ \mathtt{->}\ [\mathtt{gSum=0;W}]\underbrace{(\mathtt{idx} \doteq 0 \ \&\ \mathtt{gSum} \geq 0)}_{\mathrm{Post}}}$$

Example: Application of Loop Rule with Updates

$$\text{(initially valid)}\ \mathtt{idx} \geq 0, \{\mathtt{gSum} := 0\}(\mathtt{idx} \doteq \mathtt{c}_1\ \&\ \mathtt{gSum} \doteq \mathtt{c}_2) \Longrightarrow \exists\bar{\gamma}.\{\mathcal{U}'\}(\mathtt{idx} \doteq \mathtt{c}_1\ \&\ \mathtt{gSum} \doteq \mathtt{c}_2)$$

$$\text{(invariant)}\ \{\mathcal{U}'\}\mathtt{idx} > 0,\ \{\mathcal{U}'\}[\mathtt{R}](\bar{\mathtt{x}} \doteq \bar{\mathtt{c}}) \Longrightarrow \exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{\mathtt{c}})$$

$$\cfrac{\cfrac{\text{(initially valid)}}{} \qquad\qquad \cfrac{\text{(invariant)}}{} \qquad\qquad \mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ \ (idx>0)\{R\}}\ \ ]\mathit{Post}}{\Longrightarrow \mathtt{idx} \geq 0\ \texttt{->}\ [\mathtt{gSum=0;W}]\underbrace{(\mathtt{idx} \doteq 0\ \&\ \mathtt{gSum} \geq 0)}_{\text{Post}}}$$

Example: Application of Loop Rule with Updates

(initially valid) $\mathtt{idx} \geq 0, \{\mathtt{gSum} := 0\}(\mathtt{idx} \doteq \mathtt{c}_1 \; \& \; \mathtt{gSum} \doteq \mathtt{c}_2) \Longrightarrow$
$\exists \bar{\gamma}. \{\mathcal{U}'\}(\mathtt{idx} \doteq \mathtt{c}_1 \; \& \; \mathtt{gSum} \doteq \mathtt{c}_2)$

(invariant) $\{\mathcal{U}'\}\mathtt{idx} > 0, \; \{\mathcal{U}'\}[\mathtt{R}](\bar{\mathtt{x}} \doteq \bar{\mathtt{c}}) \Longrightarrow \exists \bar{\gamma}. \{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{\mathtt{c}})$

(continue) $\{\mathcal{U}'\} \, ! \, \mathtt{idx} > 0 \Longrightarrow \{\mathcal{U}'\}[\;]\mathrm{Post}$

$$\frac{\begin{array}{ccc} \overline{\text{(initially valid)}} & \overline{\text{(invariant)}} & \overline{\text{(continue)}} \\ \end{array}}{\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ \ (idx>0)\{R\}}\ \ ]\mathrm{Post}}$$

$$\overline{\Longrightarrow \mathtt{idx} \geq 0 \; \texttt{->} \; [\mathtt{gSum=0;W}] \underbrace{(\mathtt{idx} \doteq 0 \; \& \; \mathtt{gSum} \geq 0)}_{\mathrm{Post}}}$$

Example: Application of Loop Rule with Updates

(initially valid) $\mathtt{idx} \geq 0, \{\mathtt{gSum} := 0\}(\mathtt{idx} \doteq \mathtt{c}_1 \ \& \ \mathtt{gSum} \doteq \mathtt{c}_2) \Longrightarrow$
$\exists \bar{\gamma}.\{\mathcal{U}'\}(\mathtt{idx} \doteq \mathtt{c}_1 \ \& \ \mathtt{gSum} \doteq \mathtt{c}_2)$

(invariant) $\{\mathcal{U}'\}\mathtt{idx} > 0, \ \{\mathcal{U}'\}[\mathtt{R}](\bar{\mathtt{x}} \doteq \bar{\mathtt{c}}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{\mathtt{c}})$

(continue) $\{\mathcal{U}'\}\,!\,\mathtt{idx} > 0 \Longrightarrow \{\mathcal{U}'\}[\ ]\mathrm{Post}$

How to obtain $\mathcal{U}'$?

$$\frac{\dfrac{\text{(initially valid)}}{} \qquad \dfrac{\text{(invariant)}}{} \qquad \dfrac{\text{(continue)}}{}}{\dfrac{\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ \ (idx>0)\{R\}}\ \ ]\mathrm{Post}}{\Longrightarrow \mathtt{idx} \geq 0 \ \texttt{->}\ [\mathtt{gSum=0;W}]\underbrace{(\mathtt{idx} \doteq 0 \ \& \ \mathtt{gSum} \geq 0)}_{\mathrm{Post}}}}$$

Fixpoint approximation
in side computation

Computing Invariants

$$\frac{\texttt{idx} \geq 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{while (idx>0)\{R\}}]\mathsf{Post}}{}$$

1. Start: initial proof obligation for the loop

Computing Invariants

Termination (ignored)

$$\frac{\texttt{idx} \geq 0, \{\texttt{gSum} := 0\}(\texttt{idx} > 0) \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{R;W}]\mathsf{Post}}{\texttt{idx} \geq 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{while (idx>0)\{R\}}]\mathsf{Post}}$$

1. Start: initial proof obligation for the loop
2. Unwind the loop `W` once

Computing Invariants

$$\texttt{idx} > 0 \Longrightarrow \{\texttt{gSum} := \texttt{idx} - 1 \| \texttt{idx} := \texttt{idx} - 1\}[\texttt{W}]\mathsf{Post}$$

$$\vdots$$

$$\texttt{idx} > 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{R;W}]\mathsf{Post}$$

Termination (ignored)

$$\texttt{idx} \geq 0, \{\texttt{gSum} := 0\}(\texttt{idx} > 0) \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{R;W}]\mathsf{Post}$$

$$\texttt{idx} \geq 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{while (idx>0)\{R\}}]\mathsf{Post}$$

1. Start: initial proof obligation for the loop
2. Unwind the loop `W` once and symbolically execute body `R`

Computing Invariants

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

$$\vdots$$

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

Termination (ignored)

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

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

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

Computing Invariants

$$\texttt{idx} > 0 \Longrightarrow \{\texttt{gSum} := \texttt{idx} - 1 \| \texttt{idx} := \texttt{idx} - 1\}[\texttt{W}]\mathsf{Post}$$

$$\vdots$$

$$\texttt{idx} > 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{R;W}]\mathsf{Post}$$

Termination (ignored)

$$\texttt{idx} \geq 0, \{\texttt{gSum} := 0\}(\texttt{idx} > 0) \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{R;W}]\mathsf{Post}$$

$$\texttt{idx} \geq 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{while (idx>0)\{R\}}]\mathsf{Post}$$

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

$$\mathtt{idx} > 0 \Longrightarrow \{\mathtt{gSum} := \gamma_{\geq,1} \| \mathtt{idx} := \gamma_{\geq,2}\}[\mathtt{W}]\mathrm{Post}$$

$$\mathtt{idx} > 0 \Longrightarrow \{\mathtt{gSum} := \mathtt{idx} - 1 \| \mathtt{idx} := \mathtt{idx} - 1\}[\mathtt{W}]\mathrm{Post}$$

$$\vdots$$

$$\mathtt{idx} > 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{R;W}]\mathrm{Post}$$

Termination (ignored)

$$\mathtt{idx} \geq 0, \{\mathtt{gSum} := 0\}(\mathtt{idx} > 0) \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{R;W}]\mathrm{Post}$$

$$\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ (idx>0)\{R\}}]\mathrm{Post}$$

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
   3. abstraction of conflicting subterms using $\sqcup$-minimal value

Computing Invariants

$$\mathtt{idx} > 0 \Longrightarrow \{\mathtt{gSum} := \gamma_{\geq,1} \| \mathtt{idx} := \gamma_{\geq,2}\}[\mathtt{W}]\mathsf{Post}$$

$$\mathtt{idx} > 0 \Longrightarrow \{\mathtt{gSum} := \mathtt{idx} - 1 \| \mathtt{idx} := \mathtt{idx} - 1\}[\mathtt{W}]\mathsf{Post}$$

$$\vdots$$

$$\mathtt{idx} > 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{R;W}]\mathsf{Post}$$

Termination (ignored)

$$\mathtt{idx} \geq 0, \{\mathtt{gSum} := 0\}(\mathtt{idx} > 0) \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{R;W}]\mathsf{Post}$$

$$\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ (idx>0)\{R\}}]\mathsf{Post}$$

---

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 ⇒ automated theorem prover

Computing Invariants Cont'd

$$\dfrac{\begin{array}{c}\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := \gamma_{\geq,1} \| \mathtt{idx} := \gamma_{\geq,2}\}[\mathtt{W}]\mathsf{Post} \\ \vdots \end{array}}{\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ (idx>0)\{R\}}]\mathsf{Post}}$$

1. No fixpoint reached yet $\rightarrow$ unwind loop once more

Computing Invariants Cont'd

Termination (ignored)

$$\frac{\begin{array}{c}\mathtt{idx} \geq 0, \{\mathtt{gSum} := 0\}(\mathtt{idx} > 0) \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{R;W}]\mathsf{Post} \\ \mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := \gamma_{\geq,1} \| \mathtt{idx} := \gamma_{\geq,2}\}[\mathtt{W}]\mathsf{Post} \\ \vdots \end{array}}{\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ \ (idx>0)\{R\}}]\mathsf{Post}}$$

1. No fixpoint reached yet $\rightarrow$ unwind loop once more
2. Symbolically execute loop body

Computing Invariants Cont'd

$$\ldots \Longrightarrow \{\mathtt{gSum} := \gamma_{\geq,3} \| \mathtt{idx} := \gamma_{\geq,4}\}[\mathtt{W}]\mathrm{Post}$$

$$\vdots$$

Termination (ignored)

$$\mathtt{idx} \geq 0, \{\mathtt{gSum} := 0\}(\mathtt{idx} > 0) \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{R;W}]\mathrm{Post}$$

$$\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := \gamma_{\geq,1} \| \mathtt{idx} := \gamma_{\geq,2}\}[\mathtt{W}]\mathrm{Post}$$

$$\vdots$$

$$\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ (idx>0)\{R\}}]\mathrm{Post}$$

---

1. No fixpoint reached yet $\rightarrow$ unwind loop once more
2. Symbolically execute loop body
3. Compute abstract update

found invariant update $\ldots \Longrightarrow \{\mathtt{gSum} := \gamma_{\geq,3} \| \mathtt{idx} := \gamma_{\geq,4}\}[\mathtt{W}]\mathrm{Post}$

$\vdots$

Termination (ignored) $\mathtt{idx} \geq 0, \{\mathtt{gSum} := 0\}(\mathtt{idx} > 0) \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{R;W}]\mathrm{Post}$

$\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := \gamma_{\geq,1} \| \mathtt{idx} := \gamma_{\geq,2}\}[\mathtt{W}]\mathrm{Post}$

$\vdots$

$\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ (idx>0)\{R\}}]\mathrm{Post}$

---

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

(initially valid) $\mathtt{idx} \geq 0, \{\mathtt{gSum} := 0\}(\mathtt{idx} \doteq \mathtt{c}_1 \ \& \ \mathtt{gSum} \doteq \mathtt{c}_2) \Longrightarrow$
$\exists \bar{\gamma}. \{\mathcal{U}'\}(\mathtt{idx} \doteq \mathtt{c}_1 \ \& \ \mathtt{gSum} \doteq \mathtt{c}_2)$

(invariant) $\{\mathcal{U}'\}\mathtt{idx} > 0, \ \{\mathcal{U}'\}[\mathtt{R}](\bar{\mathtt{x}} \doteq \bar{\mathtt{c}}) \Longrightarrow \exists \bar{\gamma}. \{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{\mathtt{c}})$

(continue) $\{\mathcal{U}'\} \, ! \, \mathtt{idx} > 0 \Longrightarrow \{\mathcal{U}'\}[\ ]\mathrm{Post}$

How to obtain $\mathcal{U}'$?

$$\frac{\overline{\text{(initially valid)}} \qquad \overline{\text{(invariant)}} \qquad \overline{\text{(continue)}}}{\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ \ (idx>0)\{R\}}\ \ ]\mathrm{Post}}$$

$$\Longrightarrow \mathtt{idx} \geq 0 \ \mathtt{->} \ [\mathtt{gSum=0;W}]\underbrace{(\mathtt{idx} \doteq 0 \ \& \ \mathtt{gSum} \geq 0)}_{\mathrm{Post}}$$

Example: Application of Loop Rule with Updates

(initially valid) $\texttt{idx} \geq 0, \{\texttt{gSum} := 0\}(\texttt{idx} \doteq \texttt{c}_1 \ \& \ \texttt{gSum} \doteq \texttt{c}_2) \Longrightarrow$
$\exists \bar{\gamma}.\{\mathcal{U}'\}(\texttt{idx} \doteq \texttt{c}_1 \ \& \ \texttt{gSum} \doteq \texttt{c}_2)$

(invariant) $\{\mathcal{U}'\}\texttt{idx} > 0, \ \{\mathcal{U}'\}[\texttt{R}](\bar{\texttt{x}} \doteq \bar{\texttt{c}}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\texttt{x}} \doteq \bar{\texttt{c}})$

(continue) $\{\mathcal{U}'\} \,!\, \texttt{idx} > 0 \Longrightarrow \{\mathcal{U}'\}[\ ]\mathrm{Post}$

How to obtain $\mathcal{U}'$?

$\texttt{gSum} := \gamma_{\geq,3} \| \texttt{idx} := \gamma_{\geq,4}$

$$\frac{\text{(initially valid)} \qquad \text{(invariant)} \qquad \text{(continue)}}{\texttt{idx} \geq 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{while (idx>0)\{R\}}\ ]\mathrm{Post}}$$

$$\Longrightarrow \texttt{idx} \geq 0 \texttt{ -> } [\texttt{gSum=0;W}]\underbrace{(\texttt{idx} \doteq 0 \ \& \ \texttt{gSum} \geq 0)}_{\mathrm{Post}}$$

Example: Final Step

$$\ast \quad \ast$$

$$\vdots \quad \vdots$$

$$\dfrac{\dfrac{(\text{init.}) \quad (\text{inv.}) \qquad\qquad \color{red}{(\text{continue})}}{\texttt{idx} \geq 0 \Longrightarrow \{\texttt{gSum} := 0\}[\texttt{while (idx>0)\{R\}}\ ]\text{Post}}}{\Longrightarrow \texttt{idx} \geq 0 \texttt{ -> [gSum=0;W]} \underbrace{(\texttt{idx} \doteq 0 \ \& \ \texttt{gSum} \geq 0)}_{\text{Post}}}$$

Example: Final Step

$$
\dfrac{
\dfrac{
\begin{array}{c} * \\ \vdots \\ (\text{init.}) \end{array}
\begin{array}{c} * \\ \vdots \\ (\text{inv.}) \end{array}
\quad
\begin{array}{r}
\{\color{red}{\mathtt{gSum} := \gamma_{\geq,3} \| \mathtt{idx} := \gamma_{\geq,4}}\}\, !\, \mathtt{idx} > 0 \Longrightarrow \\
\{\color{red}{\mathtt{gSum} := \gamma_{\geq,3} \| \mathtt{idx} := \gamma_{\geq,4}}\}[\ ](\mathtt{idx} \doteq 0 \ \&\ \mathtt{gSum} \geq 0)
\end{array}
}{
\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ \ (idx>0)\{R\}}\ ]\mathrm{Post}
}
}{
\Longrightarrow \mathtt{idx} \geq 0 \ \mathtt{->}\ [\mathtt{gSum=0;W}]\underbrace{(\mathtt{idx} \doteq 0 \ \&\ \mathtt{gSum} \geq 0)}_{\mathrm{Post}}
}
$$

$$
\dfrac{
\dfrac{\ast}{\dfrac{\vdots}{\text{(init.)}}}
\qquad
\dfrac{\ast}{\dfrac{\vdots}{\text{(inv.)}}}
\qquad
\dfrac{
\dfrac{
\dfrac{!\,\gamma_{\geq,4} > 0 \Longrightarrow (\gamma_{\geq,4} \doteq 0 \;\&\; \gamma_{\geq,3} \geq 0)}{\vdots}
}{
\begin{array}{l}
\{\mathtt{gSum} := \gamma_{\geq,3} \| \mathtt{idx} := \gamma_{\geq,4}\}\,!\,\mathtt{idx} > 0 \Longrightarrow \\
\qquad \{\mathtt{gSum} := \gamma_{\geq,3} \| \mathtt{idx} := \gamma_{\geq,4}\}[\;](\mathtt{idx} \doteq 0 \;\&\; \mathtt{gSum} \geq 0)
\end{array}
}
}{
\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ (idx>0)\{R\}}\ ]\mathrm{Post}
}
}{
\Longrightarrow \mathtt{idx} \geq 0 \;\mathtt{->}\; [\mathtt{gSum=0;W}]\underbrace{(\mathtt{idx} \doteq 0 \;\&\; \mathtt{gSum} \geq 0)}_{\mathrm{Post}}
}
$$

Example: Final Step

$$*$$

$$\vdots \text{(axioms for } \gamma_{\geq,\mathbb{Z}})$$

$$! \, \gamma_{\geq,4} > 0 \Longrightarrow (\gamma_{\geq,4} \doteq 0 \ \& \ \gamma_{\geq,3} \geq 0)$$

$$* \quad *$$

$$\vdots$$

$$\vdots \quad \vdots$$

$$\{\mathtt{gSum} := \gamma_{\geq,3} \| \mathtt{idx} := \gamma_{\geq,4}\} \, ! \, \mathtt{idx} > 0 \Longrightarrow$$
$$\{\mathtt{gSum} := \gamma_{\geq,3} \| \mathtt{idx} := \gamma_{\geq,4}\}[\,](\mathtt{idx} \doteq 0 \ \& \ \mathtt{gSum} \geq 0)$$

(init.)(inv.)

$$\mathtt{idx} \geq 0 \Longrightarrow \{\mathtt{gSum} := 0\}[\mathtt{while\ \ (idx>0)\{R\}}\ ]\mathrm{Post}$$

$$\Longrightarrow \mathtt{idx} \geq 0 \ \texttt{->} \ [\mathtt{gSum=0;W}] \underbrace{(\mathtt{idx} \doteq 0 \ \& \ \mathtt{gSum} \geq 0)}_{\text{Post}}$$

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

$\mathtt{x} \mapsto 3$

$\mathtt{y} \mapsto 5$

Signature (minimalist version)

- program variables $\mathrm{PV} := \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots\}$
- $\ldots$

Semantics

State s

$$\begin{array}{l|l} \mathtt{x} \mapsto 3 & \mathtt{x}^{\mathtt{dep}} \mapsto \{\mathtt{x}\} \\ \mathtt{y} \mapsto 5 & \mathtt{y}^{\mathtt{dep}} \mapsto \{\mathtt{z}\} \end{array}$$

Signature (minimalist version)

- program variables PV $:= \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots\}$

$+$ PV$^{dep} := \{\mathtt{x}^{\mathtt{dep}} | \mathtt{x} \in \mathsf{PV}\}$
- ...

Semantics

State s

$$\begin{array}{l|l} \mathtt{x} \mapsto 3 & \mathtt{x}^{\mathtt{dep}} \mapsto \{\mathtt{x}\} \\ \mathtt{y} \mapsto 5 & \mathtt{y}^{\mathtt{dep}} \mapsto \{\mathtt{z}\} \end{array}$$

Signature (minimalist version)

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

A Logic for Analysing Information Flow

Semantics

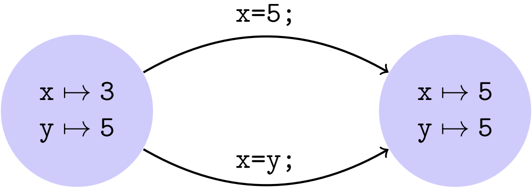
State s

$$\begin{array}{l|l} \mathtt{x} \mapsto 3 & \mathtt{x}^{\mathtt{dep}} \mapsto \{\mathtt{x}\} \\ \mathtt{y} \mapsto 5 & \mathtt{y}^{\mathtt{dep}} \mapsto \{\mathtt{z}\} \end{array}$$

Signature (minimalist version)

- program variables $\mathsf{PV} := \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots\}$
- \+ $\mathsf{PV}^{dep} := \{\mathtt{x}^{\mathtt{dep}} | \mathtt{x} \in \mathsf{PV}\}$
- \+ interpreted functions for set theory fragment $\dot{\{\}}, \{\dot{\mathtt{x}}\}$ f.a. $\mathtt{x} \in \mathsf{PV}, \dot{\cup}$
- \+ interpreted predicate $\dot{\subseteq}$
- $\ldots$

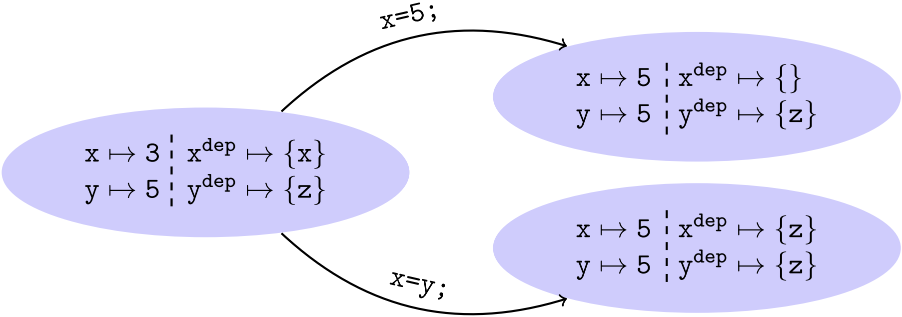
Symbolic Execution and Variable Dependencies



$$\text{assignment}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{x} := \mathtt{e}\}[\mathtt{rest}]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x=e;\ rest}]\phi, \Delta}$$

Symbolic execution of assignment:
post-state differentiates only values, no dependencies!

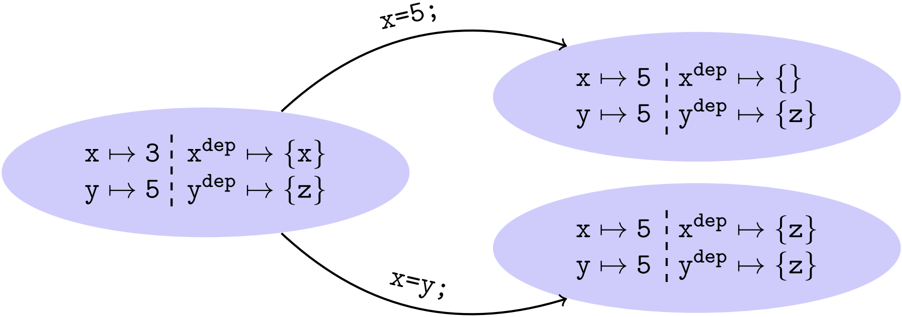
Symbolic Execution and Variable Dependencies



$$\text{assignment}^{\mathtt{dep}} \quad \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{x} := \mathtt{e} \parallel \mathtt{x}^{\mathtt{dep}} := \mathtt{dep}(\mathtt{e})\}[\mathtt{rest}]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x=e;}\ \ \mathtt{rest}]\phi, \Delta}$$

Tracking dependencies in separate variables $\mathtt{v}^{\mathtt{dep}}$

Symbolic Execution and Variable Dependencies



$$\text{assignment}^{dep} \quad \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{x} := \mathtt{e} \parallel \color{red}{\mathtt{x}^{dep} := \mathrm{dep}(\mathtt{e})}\}[\mathtt{rest}]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x=e;}\ \ \mathtt{rest}]\phi, \Delta}$$

Tracking dependencies in separate variables $\mathtt{v}^{dep}$

More rules need to be changed: if, unwindLoop (implicit dependencies)

Example: Proving Non-Interference

```
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

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

$\mathtt{l1}, \mathtt{l2}$ public variables, $\mathtt{h}$ secret variable

Non-interference: (e.g., l1 does not depend on h)

Proof obligation to be proven in program logic:

$$\mathtt{l1}^{\mathrm{dep}} \doteq \{\mathtt{l1}\},\ \mathtt{l2}^{\mathrm{dep}} \doteq \{\mathtt{l2}\},\ \mathtt{h}^{\mathrm{dep}} \doteq \{\mathtt{h}\} \Longrightarrow \\ [\mathtt{l1=0;l2=0;}\ldots]\big(\mathtt{l1}^{\mathrm{dep}}\ \dot{\subseteq} \{\mathtt{l1}\} \,\dot{\cup} \{\mathtt{l2}\}\big)$$

Proof is fully automatic; this cannot be shown with type analysis!

Example: a Suitable Abstract Domain

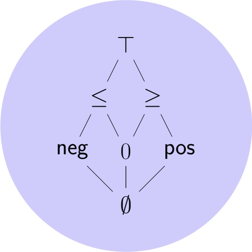
State

$\texttt{l1} \mapsto 0$

$\texttt{l2} \mapsto 0$

$\ldots$

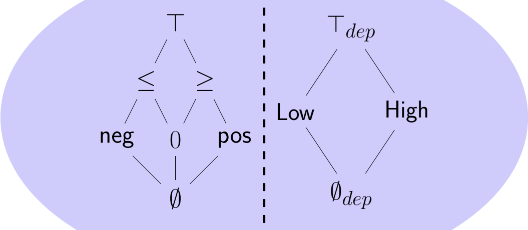
Abstract Domain



State

$\texttt{l1} \mapsto 0$	$\texttt{l1}^{\texttt{dep}} \mapsto \{\}$
$\texttt{l2} \mapsto 0$	$\texttt{l2}^{\texttt{dep}} \mapsto \{\}$
$\dots$	$\dots$

Abstract Domain



$$\gamma(\emptyset) = \emptyset, \gamma(\mathsf{Low}) = 2^{\{\texttt{l1},\texttt{l2}\}}, \gamma(\mathsf{High}) = 2^{\{\texttt{h}\}}, \gamma(\top_{dep}) = 2^{\mathsf{PV}}$$

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 $\sim$ 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.)

Summary:

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

Summary: Logic Solves Everything

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., `l = h-h`)

  proof of soundness requires trace semantics
- More complex abstract domains
- Integration with abstract interpretation tools
- Implementation (prototype ready), evaluation