Abstract Interpretation

Semantics and applications to verification

Xavier Rival

École Normale Supérieure

April 10th, 2026

Program of this lecture

Towards a realistic abstract interpreter

Last class ended with a brief overview of a **simplistic static analyzer**

Today:

- **more general soundness proof**:
  using γ, and requiring no monotonicity in the abstract level
- **more general abstract domain**:
  signs is good for introduction only, we want to see constants, intervals...
- **extended language** with **expressions** and **conditions**
  i.e., not only three address arithmetic
- **more general abstract iteration technique**:
  convergence guaranteed even with **infinite height domain**

Outline

About soundness relations

Several formalisms available:

- **abstraction function** $\alpha : C \to A$, returns the **best** approximation
- **concretization function** $\gamma : A \to C$, returns the meaning of an abstract element
- **Galois connection** $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$

Limitations of our previous abstract interpreter:

- uses the best abstraction function α all the time
- tries to establish equality $[\![P]\!]^\sharp \circ \alpha = \alpha \circ [\![P]\!]$ but fails... indeed, some operators may only compute an over-approximation
- proves $\alpha \circ [\![P]\!] \sqsubseteq [\![P]\!]^\sharp \circ \alpha$
  at the cost of proving monotonicity of $[\![P]\!]^\sharp$

Alternate approach

Use γ only and prove $[\![P]\!] \circ \gamma \subseteq \gamma \circ [\![P]\!]^\sharp$

Comparing soundness frameworks

We have seen **several ways to express soundness**:

1. $\alpha([\![P]\!]) = [\![P]\!]^\sharp$
2. $\alpha([\![P]\!]) \sqsubseteq [\![P]\!]^\sharp$
3. $[\![P]\!] \subseteq \gamma([\![P]\!]^\sharp)$
4. $[\![P]\!] \vdash [\![P]\!]^\sharp$

Some are stronger than others:

- the first is the strongest (it implies the others when applicable)
- the fourth is the weakest
- the second and third are equivalent in a Galois connection setup

Some are more general:

- the first two require an α, the third a γ
- the fourth requires very little: it does not even require α or γ to exist!

The choice of the framework to use is a balance in general...

A language with expressions

We now consider the denotational semantics of our **imperative language**:

- **variables** $\mathbb{X}$: finite, predefined set of variables
- **values** $\mathbb{V}$: $\mathbb{V}_{\text{int}} \cup \mathbb{V}_{\text{float}} \cup \ldots$
- **expressions** are allowed (not just three address instructions)
- **conditions** are **simplified** compared to initial language

Syntax

e	::=	$v \ (v \in \mathbb{V}) \mid \mathrm{x} \ (\mathrm{x} \in \mathbb{X}) \mid \mathrm{e} + \mathrm{e} \mid \mathrm{e} * \mathrm{e} \mid \ldots$	expressions
c	::=	$\mathrm{x} < v \mid \mathrm{x} = v \mid \ldots$	basic conditions
P	::=	$\mathrm{x} := \mathrm{e}$	assignment
	$\mid$	input(x)	non deterministic value input
	$\mid$	if(c) P else P	condition
	$\mid$	while(c) P	loop
	$\mid$	$P; P$	block, program($\mathbb{P}$)

Semantics of expressions and conditions (refresher)

We have defined a few lectures ago:

- a **semantics for expressions**, defined **by induction over the syntax**:

$$\llbracket \mathsf{e} \rrbracket : \mathbb{M} \longrightarrow \mathbb{V} \uplus \{\Omega\}$$

$$\begin{array}{rcl} \llbracket v \rrbracket(m) & = & v \\ \llbracket \mathrm{x} \rrbracket(m) & = & m(\mathrm{x}) \\ \llbracket \mathsf{e}_0 + \mathsf{e}_1 \rrbracket(m) & = & \llbracket \mathsf{e}_0 \rrbracket(m) \underline{+} \llbracket \mathsf{e}_1 \rrbracket(m) \\ \llbracket \mathsf{e}_0 \,/\, \mathsf{e}_1 \rrbracket(m) & = & \begin{cases} \Omega & \text{if } \llbracket \mathsf{e}_1 \rrbracket(m) = 0 \\ \llbracket \mathsf{e}_0 \rrbracket(m) \,\underline{/}\, \llbracket \mathsf{e}_1 \rrbracket(m) & \text{otherwise} \end{cases} \end{array}$$

- a **semantics for conditions**, following the same principle:

$$\llbracket \mathsf{c} \rrbracket : \mathbb{M} \longrightarrow \mathbb{V}_{\mathrm{bool}} \uplus \{\Omega\}$$

Semantics of satements

We have also defined:

Denotational semantics of programs

We use the **denotational semantics** $[\![P]\!]_{\mathcal{D}} : \mathcal{P}(\mathbb{M}) \longrightarrow \mathcal{P}(\mathbb{M})$ by:

$$
\begin{array}{rcl}
[\![\mathrm{x} := \mathrm{e}]\!]_{\mathcal{D}}(\mathcal{M}) & = & \{m[\mathrm{x} \leftarrow [\![\mathrm{e}]\!](m)] \mid m \in \mathcal{M} \wedge [\![\mathrm{e}]\!](m) \neq \Omega\} \\
[\![\mathrm{input}(\mathrm{x})]\!]_{\mathcal{D}}(\mathcal{M}) & = & \{m[\mathrm{x} \leftarrow v] \mid v \in \mathbb{V} \wedge m \in \mathcal{M}\} \\
[\![\mathrm{if}(\mathrm{c})\ P_0\ \mathrm{else}\ P_1]\!]_{\mathcal{D}}(\mathcal{M}) & = & \quad [\![P_0]\!]_{\mathcal{D}}(\{m \in \mathcal{M} \mid [\![\mathrm{c}]\!](m) = \mathtt{TRUE}\}) \\
 & & \cup\ [\![P_1]\!]_{\mathcal{D}}(\{m \in \mathcal{M} \mid [\![\mathrm{c}]\!](m) = \mathtt{FALSE}\}) \\
[\![\mathrm{while}(\mathrm{c})\ P]\!]_{\mathcal{D}}(\mathcal{M}) & = & \{m \in \mathrm{lfp}\ F_{\mathcal{D}} \mid [\![\mathrm{c}]\!](m) = \mathtt{FALSE}\} \\
\multicolumn{3}{l}{\mathrm{where}\ F_{\mathcal{D}} : \mathcal{M}' \longmapsto \mathcal{M} \cup [\![P]\!]_{\mathcal{D}}(\{m \in \mathcal{M}' \mid [\![\mathrm{c}]\!](m) = \mathtt{TRUE}\})} \\
[\![P_0; P_1]\!]_{\mathcal{D}}(\mathcal{M}) & = & [\![P_1]\!]_{\mathcal{D}} \circ [\![P_0]\!]_{\mathcal{D}}(\mathcal{M})
\end{array}
$$

- As before, we seek for **an abstract interpretation of $[\![P]\!]_{\mathcal{D}}$**
- We first need to set up **the abstraction relation**

Example

Example program:

$$
\begin{array}{l}
\text{input}(\mathtt{x});\\
\mathtt{x} = 3 - \mathtt{x};\\
\text{if}(\mathtt{x} \geq 1)\{\\
\qquad \mathtt{y} = 8 - 2 * \mathtt{x};\\
\}\text{else}\{\\
\qquad \mathtt{y} = 1;\\
\}
\end{array}
$$

Analysis with the lattice of signs: $\mathtt{x} \mapsto \top; \mathtt{y} \mapsto \top$

Can we use another abstract domain instead, such as **intervals** ?

- intuitively, $\mathtt{x} \leq 3$
- thus, either $1 \leq \mathtt{x} \leq 3$ and $2 \leq \mathtt{y} \leq 6$ or $\mathtt{x} < 1$ and $\mathtt{y} = 1$
- in any case, $1 \leq \mathtt{y} \leq 6$

Galois-connection based non-relational abstraction

We compose two abstractions:

- **non relational abstraction:** the values a variable may take is abstracted separately from the other variables
- **parameter value abstraction:** an **abstract value** describes a set of concrete values (not necessarily the lattice of sign anymore) defined by $(\mathcal{P}(\mathbb{V}), \subseteq) \underset{\alpha_{\mathcal{V}}}{\overset{\gamma_{\mathcal{V}}}{\leftrightarrows}} (D^\sharp_{\mathcal{V}}, \sqsubseteq)$

Definitions are quite similar:

Abstraction

- **concrete domain:** $(\mathcal{P}(\mathbb{X} \rightarrow \mathbb{V}), \subseteq) = (\mathcal{P}(\mathbb{M}), \subseteq)$
- **abstract domain:** $(D^\sharp, \sqsubseteq)$ ($D^\sharp = \mathbb{X} \rightarrow D^\sharp_{\mathcal{V}}$ and $\sqsubseteq$ is pointwise)
- **Galois connection** $(\mathcal{P}(\mathbb{M}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D^\sharp, \sqsubseteq)$, defined by

$$\begin{array}{rlcl} \alpha : & \mathcal{M} & \longmapsto & \lambda(\mathrm{x} \in \mathbb{X}) \cdot \alpha_{\mathcal{V}}(\{m(\mathrm{x}) \mid m \in \mathcal{M}\}) \\ \gamma : & M^\sharp & \longmapsto & \{m : \mathbb{X} \rightarrow \mathbb{V} \mid \forall \mathrm{x},\ m(\mathrm{x}) \in \gamma_{\mathcal{V}}(M^\sharp(\mathrm{x}))\} \end{array}$$

Or a more general abstraction, using only γ

As before, we compose two abstractions:

- **non relational abstraction:** the values a variable may take is abstracted separately from the other variables, as before, but we consider only concretization
- **parameter value abstraction:** an **abstract value** describes a set of concrete values defined by a monotone concretization function $\gamma_{\mathcal{V}} : (D^\sharp_{\mathcal{V}}, \sqsubseteq) \to (\mathcal{P}(\mathbb{V}), \subseteq)$

Abstraction relation based on concretization only

- **concrete domain:** $(\mathcal{P}(\mathbb{X} \to \mathbb{V}), \subseteq)$
- **abstract domain:** $(D^\sharp, \sqsubseteq)$ ($D^\sharp = \mathbb{X} \to D^\sharp_{\mathcal{V}}$ and $\sqsubseteq$ is pointwise)
- **concretization function** $\gamma : (D^\sharp, \sqsubseteq) \longrightarrow (\mathcal{P}(\mathbb{M}), \subseteq)$, defined by

$$\gamma : \quad M^\sharp \quad \longmapsto \quad \{m : \mathbb{X} \to \mathbb{V} \mid \forall \mathrm{x},\ m(\mathrm{x}) \in \gamma_{\mathcal{V}}(M^\sharp(\mathrm{x}))\}$$

$\Rightarrow$ likewise, our proof will use only γ though it works even when α is defined

Abstract semantics of sequences (revised)

We search **for an abstract semantics** $[\![P]\!]^\sharp : D^\sharp \rightarrow D^\sharp$ such that:

$$[\![P]\!] \circ \gamma \subseteq \gamma \circ [\![P]\!]^\sharp$$

We still aim for a **proof by induction over the syntax of programs**

Sequences / composition forced us to require **monotonicity** last time:

- we assume $[\![P_0]\!] \circ \gamma \subseteq \gamma \circ [\![P_0]\!]^\sharp$
- we assume $[\![P_1]\!] \circ \gamma \subseteq \gamma \circ [\![P_1]\!]^\sharp$
- since $[\![P_0; P_1]\!] = [\![P_1]\!] \circ [\![P_0]\!]$, we search for something similar in the abstract level

$$\begin{array}{rcll} [\![P_1]\!] \circ [\![P_0]\!] \circ \gamma & \subseteq & [\![P_1]\!] \circ \gamma \circ [\![P_0]\!]^\sharp & \text{(by induction)} \\ & \subseteq & \gamma \circ [\![P_1]\!]^\sharp \circ [\![P_0]\!]^\sharp & \text{(by induction)} \end{array}$$

No more requirement that $[\![P]\!]^\sharp$ be monotone (much better!)

Abstract semantics of expressions

Analysis of an expression

- semantics of expressions $[\![\mathrm{e}]\!] : \mathbb{M} \longrightarrow \mathbb{V} \uplus \{\Omega\}$
- thus, the abstract semantics **should evaluate it into an abstract value**:

$$[\![\mathrm{e}]\!]^\sharp : D^\sharp \longrightarrow D^\sharp_{\mathcal{V}}$$

Since we use the concrete semantics as a guide, we need:

- **abstraction for constants:**
  i.e., a function $\phi_{\mathcal{V}} : \mathbb{V} \rightarrow D^\sharp_{\mathcal{V}}$ such that $\forall v \in \mathbb{V},\ v \in \gamma_{\mathcal{V}}(\phi_{\mathcal{V}}(v))$
  note: if $\alpha_{\mathcal{V}}$ exists, then we may take $v \longmapsto \alpha_{\mathcal{V}}(\{v\})$ note: if it is too hard to compute, we may take something coarser
- **abstract operators:**
  i.e., for each binary operator $\oplus$, an abstract operator $\oplus^\sharp$ such that:

$$\forall v_0^\sharp, v_1^\sharp \in D^\sharp_{\mathcal{V}},\ \{v_0 \oplus v_1 \mid \forall i,\ v_i \in \gamma_{\mathcal{V}}(v_i^\sharp)\} \subseteq \gamma_{\mathcal{V}}(v_0^\sharp \oplus^\sharp v_1^\sharp)$$

Abstract semantics of expressions

Analysis of expressions: definition

We define $[\![\mathrm{e}]\!]^\sharp : D^\sharp \longrightarrow D^\sharp_{\mathcal{V}}$ by:

$$\begin{array}{rcl} [\![v]\!]^\sharp(M^\sharp) & = & \phi_{\mathcal{V}}(v) \\ [\![\mathtt{x}]\!]^\sharp(M^\sharp) & = & M^\sharp(\mathtt{x}) \\ [\![\mathrm{e}_0 \oplus \mathrm{e}_1]\!]^\sharp(M^\sharp) & = & [\![\mathrm{e}_0]\!]^\sharp(M^\sharp) \oplus^\sharp [\![\mathrm{e}_1]\!]^\sharp(M^\sharp) \end{array}$$

Analysis of expressions: soundness

For all expression e and for all abstract memory state $M^\sharp \in D^\sharp$, we have:

$$\forall m \in \gamma(M^\sharp),\ [\![\mathrm{e}]\!](m) \text{ returns no error } \Longrightarrow [\![\mathrm{e}]\!](m) \in \gamma_{\mathcal{V}}([\![\mathrm{e}]\!]^\sharp(M^\sharp))$$

Proof:

- basic **induction over the syntax**
- relies on the soundness of each operation

Analysis of an assignment

We now rely on the abstract semantics of expressions:

$$[\![\mathrm{x} = \mathrm{e}]\!]^\sharp(M^\sharp) = M^\sharp[\mathrm{x} \leftarrow [\![\mathrm{e}]\!]^\sharp(M^\sharp)]$$

- soundness proof is very similar
- but now, is given in terms of γ

Example, based on the **interval abstract domain**; analysis of $x = 3 * y + z * x$.
If $M^\sharp : x \mapsto [-2, 3], y \mapsto [0, 1]$, and $z \mapsto [-1, 4]$ then:

$$\begin{array}{rcl} [\![\mathrm{x} = 3 * y + z * x]\!]^\sharp(M^\sharp)(x) & = & 3 * [0, 1] + [-1, 4] * [-2, 3] \\ & = & [0, 3] + [-8, 12] \\ & = & [-8, 15] \end{array}$$

Abstract semantics of conditions

Analysis of a condition

- the semantics $[\![\mathtt{c}]\!] : \mathbb{M} \longrightarrow \mathbb{V}_{\mathrm{bool}}$ of a condition evaluates it into a boolean value (or an error)
- **but** the semantics relies on its functional inverse:
  e.g., $\{m \in \mathbb{M} \mid [\![\mathtt{c}]\!](m) = \mathtt{TRUE}\}$ or $\{m \in \mathbb{M} \mid [\![\mathtt{c}]\!](m) = \mathtt{FALSE}\}$
- thus, the abstract semantics **should tell which memories satisfy a condition**:

$$[\![\mathtt{c}]\!]^\sharp : \mathbb{V}_{\mathrm{bool}} \times D^\sharp \longrightarrow D^\sharp$$

$$\forall b \in \mathbb{V}_{\mathrm{bool}},\ \forall m \in \gamma(M^\sharp),\ [\![\mathtt{c}]\!](m) = b \Longrightarrow m \in \gamma([\![\mathtt{c}]\!]^\sharp(b, M^\sharp))$$

- we assume that the abstract domain provides such a function
  $[\![\mathtt{c}]\!]^\sharp : \mathbb{V}_{\mathrm{bool}} \times D^\sharp \longrightarrow D^\sharp$
- this is also called a **backward abstract semantics**
  intuitively: it goes from outputs to arguments
- we will implement some when considering specific abstract domains

Analysis of a condition statement

Abstraction of concrete union:

- we assume a **sound abstract union operation** $\mathbf{join}^\sharp_{\mathcal{V}}$ over the value abstract domain:

$$\forall v_0^\sharp, v_1^\sharp,\ \gamma_{\mathcal{V}}(v_0^\sharp) \cup \gamma_{\mathcal{V}}(v_1^\sharp) \subseteq \gamma_{\mathcal{V}}(\mathbf{join}^\sharp_{\mathcal{V}}(v_0^\sharp, v_1^\sharp))$$

  it may be $\sqcup_{\mathcal{V}}$ if it exists, but $\mathbf{join}^\sharp_{\mathcal{V}}$ may also over-approximate it
- we let $\mathbf{join}^\sharp$ be the pointwise extension of $\mathbf{join}^\sharp_{\mathcal{V}}$
- it is also sound: $\forall M_0^\sharp, M_1^\sharp,\ \gamma(M_0^\sharp) \cup \gamma(M_1^\sharp) \subseteq \gamma(\mathbf{join}^\sharp(M_0^\sharp, M_1^\sharp))$

We derive:

$$\begin{array}{l}[\![\mathrm{if}(\mathrm{c})\ P_0\ \mathrm{else}\ P_1]\!]^\sharp(M^\sharp) = \\ \qquad \mathbf{join}^\sharp([\![P_0]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{TRUE}, M^\sharp)), [\![P_1]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{FALSE}, M^\sharp)))\end{array}$$

Proof of soundness:

- similar as in the previous course
- relies on the soundness of $[\![\mathrm{c}]\!]^\sharp$, $[\![P_0]\!]^\sharp$, $[\![P_1]\!]^\sharp$ and $\mathbf{join}^\sharp$

Example condition statement analysis

We can now show the interval analysis of the example program:

$$
\begin{array}{l}
\mathsf{input}(\mathtt{x});\\
\qquad\qquad \mathtt{x} \mapsto [0, +\infty[, \mathtt{y} \mapsto]-\infty, +\infty[\\
\mathtt{x} = 3 - \mathtt{x};\\
\qquad\qquad \mathtt{x} \mapsto]-\infty, 3], \mathtt{y} \mapsto]-\infty, +\infty[\\
\mathsf{if}(\mathtt{x} \geq 1)\{\\
\qquad\qquad \mathtt{x} \mapsto [1, 3], \mathtt{y} \mapsto]-\infty, +\infty[\\
\qquad \mathtt{y} = 8 - 2 * \mathtt{x};\\
\qquad\qquad \mathtt{x} \mapsto [1, 3], \mathtt{y} \mapsto [2, 6]\\
\}\mathsf{else}\{\\
\qquad\qquad \mathtt{x} \mapsto]-\infty, 1], \mathtt{y} \mapsto]-\infty, +\infty[\\
\qquad \mathtt{y} = 1;\\
\qquad\qquad \mathtt{x} \mapsto]-\infty, 1], \mathtt{y} \mapsto [1, 1]\\
\}\\
\qquad\qquad \mathtt{x} \mapsto]-\infty, 3], \mathtt{y} \mapsto [1, 6]
\end{array}
$$

Another example

$$
\begin{array}{l}
\text{input}(\mathrm{x}); \\
\qquad\quad \mathrm{x} \mapsto]-\infty,+\infty[, \mathrm{y} \mapsto]-\infty,+\infty[\\
\mathrm{x} = \mathrm{y} * \mathrm{y} + 1; \\
\qquad\quad \mathrm{x} \mapsto [1,+\infty[, \mathrm{y} \mapsto]-\infty,+\infty[\\
\text{if}(\mathrm{x} \leq 0)\{ \\
\qquad\quad ? \\
\qquad \mathrm{x} = -1; \\
\qquad\quad ? \\
\}\text{else}\{ \\
\qquad\quad ? \\
\} \\
\qquad\quad ?
\end{array}
$$

Questions:

- fill the "?"...
- what is happening ? how to make the analysis precise ?

Reduction in the non relational abstraction

Consider the following two abstract elements:

$$\mathtt{x} \in \bot, \mathtt{y} \in [1, 10]$$

and

$$\mathtt{x} \in \bot, \mathtt{y} \in \bot$$

- **their concretisations are equal to $\emptyset$**
- in fact, applying $\gamma \circ \alpha$ to the former returns the latter

Reduction

The **optimal reduction function** is defined by $\gamma \circ \alpha$ and returns an optimal abstract element, with the same concretisation.

While optimal reduction is not computable in general, it is in this case.

Fixpoint approximation

Again, quite similar to the previous course:

- statement while(c) P, with abstract pre-condition $M^\sharp$
- we assume sound $[\![\mathrm{c}]\!]^\sharp$ and $[\![P]\!]^\sharp$ are defined

Fixpoint approximation (instead of fixpoint transfer)

We assume $(C, \subseteq)$ and $(A, \sqsubseteq)$ are complete lattices, with a concretization function $\gamma : (A, \sqsubseteq) \rightarrow (C, \subseteq)$, two functions $f : C \rightarrow C$ and $f^\sharp : A \rightarrow A$, and two elements $c_0 \in C, a_0 \in A$ such that:

- f is continuous
- $f \circ \gamma \subseteq \gamma \circ f^\sharp$
- $c_0 \subseteq \gamma(a_0)$

We let $a_\infty = \sqcup\{(f^\sharp)^n(a_0) \mid n \in \mathbb{N}\}$. Then:

- f **has a least-fixpoint** (by Kleene's fixpoint theorem)
- $\mathrm{lfp}_{c_0} f \subseteq \gamma(a_\infty)$

Fixpoint approximation: proof

Existence of the concrete fixpoint:
First, we remark that $\mathsf{lfp}_{c_0} f$ exists, following Kleene's fixpoint theorem. Moreover:

$$\mathsf{lfp}_{c_0} f = \bigcup_{n \in \mathbb{N}} f^n(c_0)$$

Approximation of the fixpoint:
First, $a_\infty = \sqcup\{(f^\sharp)^n(a_0) \mid n \in \mathbb{N}\}$ exists since we assume A is a complete lattice (note that we have not addressed how to compute it quite yet!). We prove by induction over n that $f^n(c_0) \subseteq \gamma((f^\sharp)^n(a_0))$:

- since we assumed $c_0 \subseteq \gamma(a_0)$, the property holds at rank 0;
- if we assume $f^n(c_0) \subseteq \gamma((f^\sharp)^n(a_0))$, then $f(f^n(c_0)) \subseteq f(\gamma((f^\sharp)^n(a_0)))$ since f is monotone, which implies $f^{n+1}(c_0) \subseteq \gamma((f^\sharp)^{n+1}(a_0))$ since $f \circ \gamma \subseteq \gamma \circ f^\sharp$.

The fixpoint approximation property follows from property of least upper-bounds and from the monotonicity of γ.

Analysis of a loop

Again, quite similar to the previous course:

- statement while(c) P, with abstract pre-condition $M^\sharp$
- we assume $[\![\mathrm{c}]\!]^\sharp$ and $[\![P]\!]^\sharp$ sound abstract semantics for the condition and the loop body
- we assume the abstract domain is a finite height lattice
  this ensures that we can compute $a_\infty = \sqcup\{(f^\sharp)^n(a_0) \mid n \in \mathbb{N}\}$
  (exercise), but **intervals do not satisfy this condition**

Computation of abstract iterates:

$[\![\mathrm{while(c)}\, P]\!]^\sharp(M^\sharp) = [\![\mathrm{c}]\!]^\sharp(\mathtt{FALSE}, M_n^\sharp)$

where $\begin{cases} I_0^\sharp &= M^\sharp \\ I_{k+1}^\sharp &= [\![P]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{TRUE}, I_k^\sharp)) \end{cases}$ $\quad \begin{array}{rcl} M_0^\sharp &=& M^\sharp \\ M_{k+1}^\sharp &=& \mathbf{join}^\sharp(M_k^\sharp, I_{k+1}^\sharp) \end{array}$

and $M_{n+1}^\sharp = M_n^\sharp$

Static analysis

We can now summarize the definition of our static analysis:

Definition

$$
\begin{array}{rcl}
[\![P_0; P_1]\!]^\sharp(M^\sharp) & = & [\![P_1]\!]^\sharp \circ [\![P_0]\!]^\sharp(M^\sharp) \\
[\![\mathrm{x} = \mathrm{e}]\!]^\sharp(M^\sharp) & = & M^\sharp[\mathrm{x} \leftarrow [\![\mathrm{e}]\!]^\sharp(M^\sharp)] \\
[\![\mathrm{input}()]\!]^\sharp(M^\sharp) & = & M^\sharp[\mathrm{x} \leftarrow \top] \\
[\![\mathrm{if}(\mathrm{c})\, P_0 \text{ else } P_1]\!]^\sharp(M^\sharp) & = & \mathbf{join}^\sharp([\![P_0]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{TRUE}, M^\sharp)), \\
 & & \quad [\![P_1]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{FALSE}, M^\sharp))) \\
[\![\mathrm{while}(\mathrm{c})\, P]\!]^\sharp(M^\sharp) & = & \lim_n M_n^\sharp
\end{array}
$$

where $I_0^\sharp = M_0^\sharp = M^\sharp$, $I_{k+1}^\sharp = [\![P]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{TRUE}, I_k^\sharp))$
and $M_{k+1}^\sharp = \mathbf{join}^\sharp(M_k^\sharp, I_{k+1}^\sharp)$

And, by induction over the syntax, we can prove:

Soundness

For all program P, $\forall M^\sharp \in D^\sharp,\ [\![P]\!] \circ \gamma(M^\sharp) \subseteq \gamma \circ [\![P]\!]^\sharp(M^\sharp)$

Outline

Limitations related to abstract iteration

We need a finite height lattice:

- otherwise the computation of lfp $F^\sharp$ **may not converge** as was the case when we discussed **WLP calculus**
- **consequence 1**: so far, the **abstract domain of intervals** is out...
- **consequence 2**: if the number of variables **is not fixed** or **bounded**, we cannot prove convergence at this point

Even when the abstract domain $D^\sharp_{\mathcal{V}}$ is of finite height, **this height may be huge**: then abstract computations are very costly!

We now need a more general abstract iteration technique

Intuition from **search for an unknown inductive property**:

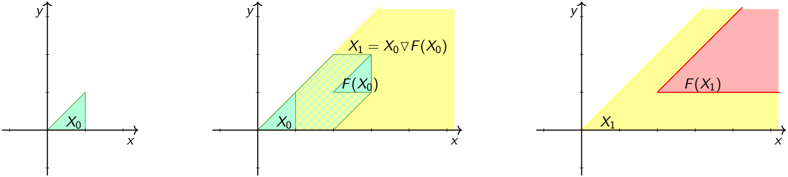
1. look at the base case and following cases
2. try to **generalize them**

Widening iteration: search for inductive abstract properties

Computing invariants about infinite executions with widening $\triangledown$

- **Widening** $\triangledown$ over-approximates $\cup$: **soundness guarantee**
- **Widening** $\triangledown$ guarantees the **termination of the analyses**
- Typical choice of $\triangledown$: **remove unstable constraints**

Example: iteration of the translation $(2, 1)$, with **octagonal polyhedra** (i.e., convex polyhedra the axes of which are either at a 0° or 45° angle)



initial — iteration 1 — iteration 2: stable !

- Initially: **3 constraints**
- After one iteration: **2 constraints**, then stable

Widening operator

Widening operator: Definition

A **widening operator** over an abstract domain $D^\sharp$ is a binary operator $\triangledown$ such that:

- $\forall M_0^\sharp, M_1^\sharp,\ \gamma(M_0^\sharp) \cup \gamma(M_1^\sharp) \subseteq \gamma(M_0^\sharp \triangledown M_1^\sharp)$
- if $(N_k^\sharp)_{k\in\mathbb{N}}$ is a sequence of elements of $D^\sharp$ the **sequence $(M_k^\sharp)_{k\in\mathbb{N}}$ defined below is stationary:**

$$\begin{array}{rcl} M_0^\sharp & = & N_0^\sharp \\ M_{k+1}^\sharp & = & M_k^\sharp \triangledown N_{k+1}^\sharp \end{array}$$

- **Intuition**:
  point 1 expresses **over-approximation** of concrete union
  point 2 enforces **termination**
- **Alternate definitions** exist:
  e.g., using $\sqsubseteq$ instead of $\subseteq$ over concretizations

Widening operator in a finite height domain

Theorem

We assume that $(D^\sharp, \sqsubseteq)$ is a **finite height domain** and that $\sqcup$ **is the least upper bound over** $D^\sharp$.
Then $\sqcup$ **defines a widening over** $D^\sharp$.

Proof:

1. since $M_0^\sharp \sqsubseteq M_0^\sharp \sqcup M_1^\sharp$, we have $\gamma(M_0^\sharp) \sqsubseteq \gamma(M_0^\sharp \sqcup M_1^\sharp)$
2. a sequence of iterates $(M_k^\sharp)_{k\in\mathbb{N}}$ is an **increasing chain**, so if every increasing chain is finite, it will eventually stabilize

Applications:

- obvious widening operators for the lattices of constants, signs...
- abstract iteration algorithms are also the same

A widening operator in an infinite height domain

We consider the **value lattice of semi intervals with left bound** 0:

- $D^\sharp_{\mathcal{V}} = \{\bot\} \uplus \mathbb{Z}_+ \uplus \{+\infty\}$; $\gamma_{\mathcal{V}}(v) = \{0, 1, \ldots, v\}$
- $\forall v^\sharp,\ \bot \sqsubseteq v^\sharp$ and if $v^\sharp_0 \leq v^\sharp_1$, then $v^\sharp_0 \sqsubseteq v^\sharp_1$

We define **the widening operator** below:

Widening operator

$$
\begin{array}{rcl}
\bot \triangledown v^\sharp & = & v^\sharp \\
v^\sharp \triangledown \bot & = & v^\sharp \\
v^\sharp_0 \triangledown v^\sharp_1 & = & \left\{ \begin{array}{ll} v^\sharp_0 & \text{if } v^\sharp_0 \geq v^\sharp_1 \\ +\infty & \text{if } v^\sharp_0 < v^\sharp_1 \end{array} \right.
\end{array}
$$

Examples: $[0, 8] \triangledown [0, 6] = [0, 8]$ $\quad$ $[0, 8] \triangledown [0, 9] = [0, +\infty[$

Widening for intervals

Exercise: generalize this definition for both bounds

Fixpoint approximation using a widening operator

Theorem: widening based fixpoint approximation

We assume $(C, \subseteq)$ is a complete lattice and that $(A, \sqsubseteq)$ is an abstract domain with a concretization function $\gamma : A \to C$ and a widening operator $\triangledown$. Moreover, we assume that:

- f is continuous (so it has a least fixpoint $\mathrm{lfp}\, f = \bigcup_{n \in \mathbb{N}} f^n(\emptyset)$)
- $f \circ \gamma \subseteq \gamma \circ f^\sharp$

We let the sequence $(M^\sharp_k)_{k \in \mathbb{N}}$ be defined by:

$$\begin{array}{rcl} M^\sharp_0 & = & \bot \\ M^\sharp_{k+1} & = & M^\sharp_k \triangledown f^\sharp(M^\sharp_k) \end{array}$$

Then:

1. $(M^\sharp_k)_{k \in \mathbb{N}}$ **is stationary** and we write $M^\sharp_{\mathrm{lim}}$ for its limit
2. $\mathrm{lfp}\, f \subseteq \gamma(M^\sharp_{\mathrm{lim}})$

Fixpoint approximation using a widening operator, proof

We assume all the assumptions of the theorem, and prove the two points:

1. **Sequence convergence**: We let $\begin{cases} N^\sharp_0 & = & \bot \\ N^\sharp_{k+1} & = & f^\sharp(M^\sharp_k) \end{cases}$
   Then, convergence follows directly from the definition of widening.
   There exists a rank K from which all iterates are stable.
2. **Soundness of the limit**:
   We prove by induction over k that $\forall l \geq k,\ f^k(\emptyset) \subseteq \gamma(M^\sharp_l)$:
   - the result clearly holds for $k = 0$;
   - if the result holds at rank k and $l \geq k$ then:
     $$\begin{array}{rcll} f^{k+1}(\emptyset) & = & f(f^k(\emptyset)) & \\ & \subseteq & f(\gamma(M^\sharp_l)) & \text{by induction} \\ & \subseteq & \gamma(f^\sharp(M^\sharp_l)) & \text{since } f \circ \gamma \subseteq \gamma \circ f^\sharp \\ & \subseteq & \gamma(M^\sharp_l \triangledown f^\sharp(M^\sharp_l)) & \text{by definition of } \triangledown \\ & = & \gamma(M^\sharp_{l+1}) & \end{array}$$

   When $(M^\sharp_k)_{k\in\mathbb{N}}$ converges, $\forall l \geq K,\ M^\sharp_l = M^\sharp_K = M^\sharp_\infty$, thus
   $\forall k,\ f^k(\emptyset) \subseteq \gamma(M^\sharp_\infty)$ thus $\mathsf{lfp}\, f \subseteq \gamma(M^\sharp_\infty)$

Example widening iteration

```
int x = 0;
while(TRUE){
    if(x < 10 000){
        x = x + 1;
    } else {
        x = −x;
    }
}
```

Example widening iteration

int $\mathtt{x} = 0$;

$\mathtt{x} \in [0, 0]$

while(TRUE){

if($\mathtt{x} < 10\,000$){

$\mathtt{x} = \mathtt{x} + 1$;

} else {

$\mathtt{x} = -\mathtt{x}$;

}

}

Example widening iteration

int $\mathtt{x} = 0$;

$\mathtt{x} \in [0, 0]$

while(TRUE){

$\mathtt{x} \in [0, 0]$

if($\mathtt{x} < 10\,000$){

$\mathtt{x} = \mathtt{x} + 1$;

} else {

$\mathtt{x} = -\mathtt{x}$;

}

}

Entry into the loop

Example widening iteration

int $\mathtt{x} = 0$;
$\mathtt{x} \in [0, 0]$
while($\mathtt{TRUE}$){
$\mathtt{x} \in [0, 0]$
if($\mathtt{x} < 10\,000$){
$\mathtt{x} \in [0, 0]$
$\mathtt{x} = \mathtt{x} + 1$;

} else {
$\mathtt{x} \in \emptyset$
$\mathtt{x} = -\mathtt{x}$;

}

}

Only the "true" branch may be taken

Example widening iteration

int $\mathtt{x} = 0$;
$\mathtt{x} \in [0, 0]$
while($\mathtt{TRUE}$){
$\mathtt{x} \in [0, 0]$
if($\mathtt{x} < 10\,000$){
$\mathtt{x} \in [0, 0]$
$\mathtt{x} = \mathtt{x} + 1$;
$\mathtt{x} \in [1, 1]$
} else {
$\mathtt{x} \in \emptyset$
$\mathtt{x} = -\mathtt{x}$;
$\mathtt{x} \in \emptyset$
}

}

Incrementation

Example widening iteration

$\textbf{int}\ \mathtt{x} = 0;$
$\quad\quad \mathtt{x} \in [0, 0]$
$\text{while}(\mathtt{TRUE})\{$
$\quad\quad \mathtt{x} \in [0, 0]$
$\quad \text{if}(\mathtt{x} < 10\,000)\{$
$\quad\quad \mathtt{x} \in [0, 0]$
$\quad\quad \mathtt{x} = \mathtt{x} + 1;$
$\quad\quad \mathtt{x} \in [1, 1]$
$\quad \}\ \text{else}\ \{$
$\quad\quad \mathtt{x} \in \emptyset$
$\quad\quad \mathtt{x} = -\mathtt{x};$
$\quad\quad \mathtt{x} \in \emptyset$
$\quad \}$
$\quad\quad \mathtt{x} \in [1, 1]$
$\}$

Abstract union at the end of the condition

Example widening iteration

```
int x = 0;
                x ∈ [0, 0]
while(TRUE){
                x ∈ [0, +∞[
    if(x < 10 000){
                x ∈ [0, 0]
        x = x + 1;
                x ∈ [1, 1]
    } else {
                x ∈ ∅
        x = −x;
                x ∈ ∅
    }
                x ∈ [1, 1]
}
```

Widening at loop head

Example widening iteration

$\textbf{int}\ \mathtt{x} = 0;$
$\qquad\qquad \mathtt{x} \in [0, 0]$
$\text{while}(\mathtt{TRUE})\{$
$\qquad\qquad \mathtt{x} \in [0, +\infty[$
$\qquad \text{if}(\mathtt{x} < 10\,000)\{$
$\qquad\qquad \mathtt{x} \in [0, 9999]$
$\qquad\qquad \mathtt{x} = \mathtt{x} + 1;$
$\qquad\qquad \mathtt{x} \in [1, 1]$
$\qquad \}\ \text{else}\ \{$
$\qquad\qquad \mathtt{x} \in [10000, +\infty[$
$\qquad\qquad \mathtt{x} = -\mathtt{x};$
$\qquad\qquad \mathtt{x} \in \emptyset$
$\qquad \}$
$\qquad\qquad \mathtt{x} \in [1, 1]$
$\}$

Now both branches may be taken

Example widening iteration

```
int x = 0;
            x ∈ [0, 0]
while(TRUE){
            x ∈ [0, +∞[
    if(x < 10 000){
            x ∈ [0, 9999]
        x = x + 1;
            x ∈ [1, 10000]
    } else {
            x ∈ [10000, +∞[
        x = −x;
            x ∈] − ∞, −10000]
    }
            x ∈ [1, 1]
}
```

Numerical assignments

Example widening iteration

$\textbf{int}\ \mathtt{x} = 0;$

$\mathtt{x} \in [0, 0]$

$\text{while}(\mathtt{TRUE})\{$

$\mathtt{x} \in [0, +\infty[$

$\text{if}(\mathtt{x} < 10\,000)\{$

$\mathtt{x} \in [0, 9999]$

$\mathtt{x} = \mathtt{x} + 1;$

$\mathtt{x} \in [1, 10000]$

$\}\ \text{else}\ \{$

$\mathtt{x} \in [10000, +\infty[$

$\mathtt{x} = -\mathtt{x};$

$\mathtt{x} \in]-\infty, -10000]$

$\}$

$\mathtt{x} \in]-\infty, 10000]$

$\}$

Abstract union at the end of the condition

Example widening iteration

```
int x = 0;
            x ∈ [0, 0]
while(TRUE){
            x ∈] − ∞, +∞[
    if(x < 10 000){
            x ∈ [0, 9999]
        x = x + 1;
            x ∈ [1, 10000]
    } else {
            x ∈ [10000, +∞[
        x = −x;
            x ∈] − ∞, −10000]
    }
            x ∈] − ∞, 10000]
}
```

Widening at loop head

Example widening iteration

$$
\begin{array}{l}
\textbf{int}\ \mathtt{x} = 0; \\
\qquad\qquad \mathtt{x} \in [0, 0] \\
\text{while}(\mathtt{TRUE})\{ \\
\qquad\qquad \mathtt{x} \in]-\infty, +\infty[\\
\quad \text{if}(\mathtt{x} < 10\,000)\{ \\
\qquad\qquad \mathtt{x} \in]-\infty, 9999] \\
\qquad \mathtt{x} = \mathtt{x} + 1; \\
\qquad\qquad \mathtt{x} \in [1, 10000] \\
\quad \}\ \text{else}\ \{ \\
\qquad\qquad \mathtt{x} \in [10000, +\infty[\\
\qquad \mathtt{x} = -\mathtt{x}; \\
\qquad\qquad \mathtt{x} \in]-\infty, -10000] \\
\quad \} \\
\qquad\qquad \mathtt{x} \in]-\infty, 10000] \\
\}
\end{array}
$$

Both branches may be taken

Example widening iteration

$$
\begin{array}{l}
\textbf{int}\ \mathtt{x} = 0; \\
\qquad\qquad\qquad \color{blue}{\mathtt{x} \in [0, 0]} \\
\mathsf{while}(\mathtt{TRUE})\{ \\
\qquad\qquad\qquad \color{blue}{\mathtt{x} \in]-\infty, +\infty[} \\
\qquad \mathsf{if}(\mathtt{x} < 10\,000)\{ \\
\qquad\qquad\qquad \color{blue}{\mathtt{x} \in]-\infty, 9999]} \\
\qquad\qquad \mathtt{x} = \mathtt{x} + 1; \\
\qquad\qquad\qquad \color{blue}{\mathtt{x} \in]-\infty, 10000]} \\
\qquad \}\ \mathsf{else}\ \{ \\
\qquad\qquad\qquad \color{blue}{\mathtt{x} \in [10000, +\infty[} \\
\qquad\qquad \mathtt{x} = -\mathtt{x}; \\
\qquad\qquad\qquad \color{blue}{\mathtt{x} \in]-\infty, -10000]} \\
\qquad \} \\
\qquad\qquad\qquad \color{blue}{\mathtt{x} \in]-\infty, 10000]} \\
\}
\end{array}
$$

Numerical assignments

Example widening iteration

$$
\begin{array}{l}
\textbf{int } \mathtt{x} = 0; \\
\qquad\qquad \mathtt{x} \in [0, 0] \\
\text{while}(\mathtt{TRUE})\{ \\
\qquad\qquad \mathtt{x} \in]-\infty, +\infty[\\
\quad \text{if}(\mathtt{x} < 10\,000)\{ \\
\qquad\qquad \mathtt{x} \in]-\infty, 9999] \\
\qquad \mathtt{x} = \mathtt{x} + 1; \\
\qquad\qquad \mathtt{x} \in]-\infty, 10000] \\
\quad \}\ \text{else}\ \{ \\
\qquad\qquad \mathtt{x} \in [10000, +\infty[\\
\qquad \mathtt{x} = -\mathtt{x}; \\
\qquad\qquad \mathtt{x} \in]-\infty, -10000] \\
\quad \} \\
\qquad\qquad \mathtt{x} \in]-\infty, 10000] \\
\}
\end{array}
$$

Stable! No information at loop head,
but still, some interesting information inside the loop

Loop unrolling

From the example, we observe that **intervals widening is imprecise**:

- quickly **goes to** $-\infty$ **or** $+\infty$
- **ignores possible stable bounds**

Can we do better ?

Yes, we can... many techniques improve standard widening

Loop unrolling: postpone widening

We fix an index l, and postpone widening until after l

$$\begin{array}{rcll} M_0^\sharp & = & \bot & \\ M_{k+1}^\sharp & = & \mathbf{join}^\sharp(M_k^\sharp, f^\sharp(M_k^\sharp)) & \text{if } k < l \\ M_{k+1}^\sharp & = & M_k^\sharp \triangledown f^\sharp(M_k^\sharp) & \text{otherwise} \end{array}$$

- Typically, k is set to 1 or 2...
- **Proof** of a new fixpoint approximation theorem: very similar

Widening with threshold

Now, let us improve the widening itself:

- the standard ∇ operator of intervals goes straight to ∞
- we can **slow down the process**

Threshold widening

Let $\mathcal{T}$ be a **finite set of integers**, called **thresholds**. We let the **threshold widening** be defined by:

$$\begin{array}{rcl} \bot \nabla v^\sharp & = & v^\sharp \\ v^\sharp \nabla \bot & = & v^\sharp \\ v_0^\sharp \nabla v_1^\sharp & = & \left\{ \begin{array}{ll} v_0^\sharp & \text{if } v_0^\sharp \geq v_1^\sharp \\ \min\{v^\sharp \in \mathcal{T} \mid \forall i,\ v_i^\sharp \leq v^\sharp\} & \text{if } \{v^\sharp \in \mathcal{T} \mid \forall i,\ v_i^\sharp \leq v^\sharp\} \neq \emptyset \\ +\infty & \text{otherwise} \end{array} \right. \end{array}$$

- **Proof** of the widening property: exercise
- **Example** with $\mathcal{L} = \{10\}$:

  $[0, 8] \nabla [0, 9] = [0, 10] \qquad [0, 8] \nabla [0, 15] = [0, +\infty[$

Techniques related to iterations

No widening after visiting a branch for the first time:

- loop unrolling **postpones** widening for a **finite number of times**
- there are **finitely many branches** in any block of code
  branch: condition block entry or inner loop entry

Principle

Mark program branches and **apply widening** only **when no new branch was visited during the previous iteration**

Iteration from a fixpoint approximant:

- **observation**: if $f \circ \gamma \subseteq \gamma \circ f^\sharp$ and $\mathsf{lfp}\, f \subseteq \gamma(M^\sharp)$, then:
  $\mathsf{lfp}\, f = f(\mathsf{lfp}\, f) \subseteq f \circ \gamma(M^\sharp) \subseteq \gamma \circ f^\sharp(M^\sharp)$
- so $f^\sharp(M^\sharp)$ **also approximates** $\mathsf{lfp}\, f$, and may be better

Principle

After an abstract invariant is found, perform additional iterations

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

```
int x = 0;
while(TRUE){
    if(x < 10 000){        9999 will be a threshold value at loop head
        x = x + 1;
    } else {
        x = -x;
    }
}
```

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

```
int x = 0;
            x ∈ [0, 0]
while(TRUE){

    if(x < 10 000){      9999 will be a threshold value at loop head

        x = x + 1;

    } else {

        x = −x;

    }

}
```

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

```
int x = 0;
            x ∈ [0, 0]
while(TRUE){
            x ∈ [0, 0]
    if(x < 10 000){      9999 will be a threshold value at loop head

        x = x + 1;

    } else {

        x = −x;

    }

}
```

Entering the loop

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

```
int x = 0;
            x ∈ [0, 0]
while(TRUE){
            x ∈ [0, 0]
    if(x < 10 000){        9999 will be a threshold value at loop head
            x ∈ [0, 0]
        x = x + 1;

    } else {
            x ∈ ∅
        x = −x;

    }

}
```

Only true branch possible

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

```
int x = 0;
                x ∈ [0, 0]
while(TRUE){
                x ∈ [0, 0]
    if(x < 10 000){      9999 will be a threshold value at loop head
                x ∈ [0, 0]
        x = x + 1;
                x ∈ [1, 1]
    } else {
                x ∈ ∅
        x = −x;
                x ∈ ∅
    }

}
```

Incrementation of interval

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $\mathtt{x} = 0$;
    $\mathtt{x} \in [0, 0]$
while(TRUE){
    $\mathtt{x} \in [0, 0]$
  if($\mathtt{x} < 10\,000$){   9999 will be a threshold value at loop head
    $\mathtt{x} \in [0, 0]$
    $\mathtt{x} = \mathtt{x} + 1$;
    $\mathtt{x} \in [1, 1]$
  } else {
    $\mathtt{x} \in \emptyset$
    $\mathtt{x} = -\mathtt{x}$;
    $\mathtt{x} \in \emptyset$
  }
    $\mathtt{x} \in [1, 1]$
}

Propagation

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $\mathtt{x} = 0$;
    $\mathtt{x} \in [0, 0]$
while(TRUE){
    $\mathtt{x} \in [0, 1]$
  if($\mathtt{x} < 10\,000$){   9999 will be a threshold value at loop head
    $\mathtt{x} \in [0, 0]$
    $\mathtt{x} = \mathtt{x} + 1$;
    $\mathtt{x} \in [1, 1]$
  } else {
    $\mathtt{x} \in \emptyset$
    $\mathtt{x} = -\mathtt{x}$;
    $\mathtt{x} \in \emptyset$
  }
    $\mathtt{x} \in [1, 1]$
}

Join at loop head

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $\mathtt{x} = 0$;
    $\mathtt{x} \in [0, 0]$
while(TRUE){
    $\mathtt{x} \in [0, 1]$
  if($\mathtt{x} < 10\,000$){   9999 will be a threshold value at loop head
    $\mathtt{x} \in [0, 1]$
    $\mathtt{x} = \mathtt{x} + 1$;
    $\mathtt{x} \in [1, 1]$
  } else {
    $\mathtt{x} \in \emptyset$
    $\mathtt{x} = -\mathtt{x}$;
    $\mathtt{x} \in \emptyset$
  }
    $\mathtt{x} \in [1, 1]$
}

Still only the true branch may be taken

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

```
int x = 0;
                x ∈ [0, 0]
while(TRUE){
                x ∈ [0, 1]
    if(x < 10 000){        9999 will be a threshold value at loop head
                x ∈ [0, 1]
        x = x + 1;
                x ∈ [1, 2]
    } else {
                x ∈ ∅
        x = −x;
                x ∈ ∅
    }
                x ∈ [1, 1]
}
```

Incrementation of interval

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

```
int x = 0;
            x ∈ [0, 0]
while(TRUE){
            x ∈ [0, 1]
    if(x < 10 000){        9999 will be a threshold value at loop head
            x ∈ [0, 1]
        x = x + 1;
            x ∈ [1, 2]
    } else {
            x ∈ ∅
        x = −x;
            x ∈ ∅
    }
            x ∈ [1, 2]
}
```

Propagation

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

```
int x = 0;
            x ∈ [0, 0]
while(TRUE){
            x ∈ [0, 9999]    instead of [0, +∞[
    if(x < 10 000){       9999 will be a threshold value at loop head
            x ∈ [0, 1]
        x = x + 1;
            x ∈ [1, 2]
    } else {
            x ∈ ∅
        x = −x;
            x ∈ ∅
    }
            x ∈ [1, 2]
}
```

Widening at the loop head, + threshold

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

```
int x = 0;
            x ∈ [0, 0]
while(TRUE){
            x ∈ [0, 9999]    instead of [0, +∞[
    if(x < 10 000){     9999 will be a threshold value at loop head
            x ∈ [0, 9999]
        x = x + 1;
            x ∈ [1, 2]
    } else {
            x ∈ ∅
        x = −x;
            x ∈ ∅
    }
            x ∈ [1, 2]
}
```

Still only the true branch may be taken

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $\mathtt{x} = 0$;
    $\mathtt{x} \in [0, 0]$
while(TRUE){
    $\mathtt{x} \in [0, 9999]$ instead of $[0, +\infty[$
  if($\mathtt{x} < 10\,000$){ 9999 will be a threshold value at loop head
    $\mathtt{x} \in [0, 9999]$
   $\mathtt{x} = \mathtt{x} + 1$;
    $\mathtt{x} \in [1, 10000]$
  } else {
    $\mathtt{x} \in \emptyset$
   $\mathtt{x} = -\mathtt{x}$;
    $\mathtt{x} \in \emptyset$
  }
    $\mathtt{x} \in [1, 2]$
}

Numerical assignments

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $\mathtt{x} = 0;$
$\quad\quad \mathtt{x} \in [0, 0]$
while($\mathtt{TRUE}$){
$\quad\quad \mathtt{x} \in [0, 9999]$ instead of $[0, +\infty[$
$\quad$ if($\mathtt{x} < 10\,000$){ 9999 will be a threshold value at loop head
$\quad\quad \mathtt{x} \in [0, 9999]$
$\quad\quad \mathtt{x} = \mathtt{x} + 1;$
$\quad\quad \mathtt{x} \in [1, 10000]$
$\quad$ } else {
$\quad\quad \mathtt{x} \in \emptyset$
$\quad\quad \mathtt{x} = -\mathtt{x};$
$\quad\quad \mathtt{x} \in \emptyset$
$\quad$ }
$\quad\quad \mathtt{x} \in [1, 10000]$
}

Join at the end of the loop

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $\mathtt{x} = 0$;
    $\mathtt{x} \in [0, 0]$
while(TRUE){
    $\mathtt{x} \in [0, 10000]$ instead of $]-\infty, +\infty[$
  if($\mathtt{x} < 10\,000$){ 9999 will be a threshold value at loop head
    $\mathtt{x} \in [0, 9999]$
   $\mathtt{x} = \mathtt{x} + 1$;
    $\mathtt{x} \in [1, 10000]$
  } else {
    $\mathtt{x} \in \emptyset$
   $\mathtt{x} = -\mathtt{x}$;
    $\mathtt{x} \in \emptyset$
  }
    $\mathtt{x} \in [1, 10000]$
}

Join after widening

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $\mathtt{x} = 0;$

$\quad\quad \mathtt{x} \in [0, 0]$

while(TRUE){

$\quad\quad \mathtt{x} \in [0, 10000]$ instead of $]-\infty, +\infty[$

$\quad$ if$(\mathtt{x} < 10\,000)\{$ 9999 will be a threshold value at loop head

$\quad\quad \mathtt{x} \in [0, 9999]$

$\quad\quad \mathtt{x} = \mathtt{x} + 1;$

$\quad\quad \mathtt{x} \in [1, 10000]$

$\quad$ } else {

$\quad\quad \mathtt{x} \in [10000, 10000]$ instead of $[10000, +\infty[$

$\quad\quad \mathtt{x} = -\mathtt{x};$

$\quad\quad \mathtt{x} \in \emptyset$

$\quad$ }

$\quad\quad \mathtt{x} \in [1, 10000]$

}

True branch stable, false branch visited for the first time

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $\mathtt{x} = 0$;
$\quad\quad \mathtt{x} \in [0, 0]$
while(TRUE){
$\quad\quad \mathtt{x} \in [0, 10000]$ instead of $]-\infty, +\infty[$
$\quad$ if($\mathtt{x} < 10\,000$){ $\quad$ 9999 will be a threshold value at loop head
$\quad\quad \mathtt{x} \in [0, 9999]$
$\quad\quad \mathtt{x} = \mathtt{x} + 1$;
$\quad\quad \mathtt{x} \in [1, 10000]$
$\quad$ } else {
$\quad\quad \mathtt{x} \in [10000, 10000]$ instead of $[10000, +\infty[$
$\quad\quad \mathtt{x} = -\mathtt{x}$;
$\quad\quad \mathtt{x} \in [-10000, -10000]$
$\quad$ }
$\quad\quad \mathtt{x} \in [1, 10000]$
}

True branch stable, false branch visited for the first time

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $x = 0$;
  $x \in [0, 0]$
while(TRUE){
  $x \in [0, 10000]$ instead of $]-\infty, +\infty[$
  if($x < 10\,000$){ 9999 will be a threshold value at loop head
    $x \in [0, 9999]$
    $x = x + 1$;
    $x \in [1, 10000]$
  } else {
    $x \in [10000, 10000]$ instead of $[10000, +\infty[$
    $x = -x$;
    $x \in [-10000, -10000]$
  }
  $x \in [-10000, 10000]$
}

Join at the end of the loop

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

```
int x = 0;
            x ∈ [0, 0]
while(TRUE){
            x ∈ [−10000, 10000]    instead of ] − ∞, +∞[
    if(x < 10 000){      9999 will be a threshold value at loop head
            x ∈ [0, 9999]
        x = x + 1;
            x ∈ [1, 10000]
    } else {
            x ∈ [10000, 10000]    instead of [10000, +∞[
        x = −x;
            x ∈ [−10000, −10000]
    }
            x ∈ [−10000, 10000]
}
```

Join again: no widening after visiting a new branch

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $\mathtt{x} = 0$;
        $\mathtt{x} \in [0, 0]$
while(TRUE){
        $\mathtt{x} \in [-10000, 10000]$ instead of $]-\infty, +\infty[$
    if($\mathtt{x} < 10\,000$){   9999 will be a threshold value at loop head
        $\mathtt{x} \in [-10000, 9999]$
        $\mathtt{x} = \mathtt{x} + 1$;
        $\mathtt{x} \in [1, 10000]$
    } else {
        $\mathtt{x} \in [10000, 10000]$ instead of $[10000, +\infty[$
        $\mathtt{x} = -\mathtt{x}$;
        $\mathtt{x} \in [-10000, -10000]$
    }
        $\mathtt{x} \in [-10000, 10000]$
}

Branches

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $x = 0$;
 $x \in [0, 0]$
while(TRUE){
 $x \in [-10000, 10000]$ instead of $]-\infty, +\infty[$
 if($x < 10\,000$){ 9999 will be a threshold value at loop head
 $x \in [-10000, 9999]$
 $x = x + 1$;
 $x \in [-9999, 10000]$
 } else {
 $x \in [10000, 10000]$ instead of $[10000, +\infty[$
 $x = -x$;
 $x \in [-10000, -10000]$
 }
 $x \in [-10000, 10000]$
}

Incrementation of interval in true branch; false branch stable

Example widening iteration, more precise

Classical techniques:

- add test values and neighbors as thresholds
- alternate join and widening
- no widening after visiting a new branch

int $\mathrm{x} = 0$;
    $\mathrm{x} \in [0, 0]$
while(TRUE){
    $\mathrm{x} \in [-10000, 10000]$ instead of $]-\infty, +\infty[$
  if$(\mathrm{x} < 10\,000)${   9999 will be a threshold value at loop head
    $\mathrm{x} \in [-10000, 9999]$
    $\mathrm{x} = \mathrm{x} + 1$;
    $\mathrm{x} \in [-9999, 10000]$
  } else {
    $\mathrm{x} \in [10000, 10000]$ instead of $[10000, +\infty[$
    $\mathrm{x} = -\mathrm{x}$;
    $\mathrm{x} \in [-10000, -10000]$
  }
    $\mathrm{x} \in [-10000, 10000]$
}

Everything is stable; exact ranges inferred

Widening and monotonicity

Remarks about the widening over intervals:

- it is **monotone** in its second argument,
- but it is **not monotone in its first argument**!

In fact, interesting widenings **are not monotone in their first argument**:

Let $(D^\sharp, \sqsubseteq)$ be an infinite height domain, with a widening $\triangledown$ that is stable ($\forall v^\sharp,\ v^\sharp \triangledown v^\sharp = v^\sharp$) and such that $\forall v_0^\sharp, v_1^\sharp,\ \forall i,\ v_i^\sharp \sqsubseteq v_0^\sharp \triangledown v_1^\sharp$. Then, **$\triangledown$ is not monotone in its first argument** (proof: Patrick Cousot).

Proof: we assume it is, let $w_0^\sharp \sqsubset w_1^\sharp \sqsubset \ldots$ be an infinite chain over $D^\sharp$ and define $v_0^\sharp = w_0^\sharp$, $v_{k+1}^\sharp = v_k^\sharp \triangledown w_{k+1}^\sharp$; we prove by induction that $v_k^\sharp = w_k^\sharp$:

- clear at rank 0
- we assume that $v_k^\sharp = w_k^\sharp$: then $v_{k+1}^\sharp = v_k^\sharp \triangledown w_{k+1}^\sharp$, so $w_{k+1}^\sharp \sqsubseteq v_{k+1}^\sharp$; moreover, $v_{k+1}^\sharp = v_k^\sharp \triangledown w_{k+1}^\sharp = w_k^\sharp \triangledown w_{k+1}^\sharp \sqsubseteq w_{k+1}^\sharp \triangledown w_{k+1}^\sharp = w_{k+1}^\sharp$

This contradicts the widening definition: the sequence should be stationary.

Outline

Summary

This lecture:

- **abstraction** and its formalization
- **computation of an abstract semantics** in a very simplified case

Next lectures:

- **construction** of a few **non trivial abstractions**
- **more general** ways to **compute sound abstract properties**