Abstract Interpretation

Semantics and applications to verification

Xavier Rival

École Normale Supérieure

April 6th, 2018

Program of this lecture

Towards a more realistic abstract interpreter

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**
  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** $\alpha : C \rightarrow A$, returns the **best** approximation
- **concretization function** $\gamma : A \rightarrow C$, returns the meaning of an abstract element
- **Galois connection** $(C, \subseteq) \xleftarrow[\alpha]{\gamma} (A, \sqsubseteq)$

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

A language with expressions

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

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

Syntax

$\mathtt{e}$	$::=$	$v \ \ (v \in \mathbb{V}) \mid \mathtt{x} \ \ (\mathtt{x} \in \mathbb{X}) \mid \mathtt{e} + \mathtt{e} \mid \mathtt{e} * \mathtt{e} \mid \ldots$	expressions
$\mathtt{c}$	$::=$	$\mathtt{x} < v \mid \mathtt{x} = v \mid \ldots$	basic conditions
$\mathtt{i}$	$::=$	$\mathtt{x} := \mathtt{e}$	assignment
	$\mid$	$\textbf{input}(\mathtt{x})$	random value input
	$\mid$	$\textbf{if}(\mathtt{c})\ \mathtt{b}\ \textbf{else}\ \mathtt{b}$	condition
	$\mid$	$\textbf{while}(\mathtt{c})\ \mathtt{b}$	loop
$\mathtt{b}$	$::=$	$\{\mathtt{i}; \ldots ; \mathtt{i};\}$	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**:

$$
\begin{array}{l}
[\![\mathsf{e}]\!] : \mathbb{M} \longrightarrow \mathbb{V} \uplus \{\Omega\} \\
\begin{array}{rcl}
[\![v]\!](m) & = & v \\
[\![\mathrm{x}]\!](m) & = & m(\mathrm{x}) \\
[\![\mathsf{e}_0 + \mathsf{e}_1]\!](m) & = & [\![\mathsf{e}_0]\!](m) \underline{+} [\![\mathsf{e}_1]\!](m) \\
[\![\mathsf{e}_0 \,/\, \mathsf{e}_1]\!](m) & = & \left\{ \begin{array}{ll} \Omega & \text{if } [\![\mathsf{e}_1]\!](m) = 0 \\ [\![\mathsf{e}_0]\!](m) \,\underline{/}\, [\![\mathsf{e}_1]\!](m) & \text{otherwise} \end{array} \right.
\end{array}
\end{array}
$$

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

$$
[\![\mathsf{c}]\!] : \mathbb{M} \longrightarrow \mathbb{V}_{\mathrm{bool}} \uplus \{\Omega\}
$$

Semantics of satements (refresher)

We have also defined:

Denotational semantics of programs

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

$$
\begin{array}{rcl}
[\![\mathtt{x} := \mathtt{e}]\!]_{\mathcal{D}}(\mathcal{M}) & = & \{m[\mathtt{x} \leftarrow [\![\mathtt{e}]\!](m)] \mid m \in \mathcal{M}\} \\
[\![\mathbf{input}(\mathtt{x})]\!]_{\mathcal{D}}(\mathcal{M}) & = & \{m[\mathtt{x} \leftarrow v] \mid v \in \mathbb{V} \wedge m \in \mathcal{M}\} \\
[\![\mathbf{if}(\mathtt{c})\ \mathtt{b}_0\ \mathbf{else}\ \mathtt{b}_1]\!]_{\mathcal{D}}(\mathcal{M}) & = & [\![\mathtt{b}_0]\!]_{\mathcal{D}}(\{m \in \mathcal{M} \mid [\![\mathtt{c}]\!](m) = \mathtt{TRUE}\}) \\
 & & \cup\ [\![\mathtt{b}_1]\!]_{\mathcal{D}}(\{m \in \mathcal{M} \mid [\![\mathtt{c}]\!](m) = \mathtt{FALSE}\}) \\
[\![\mathbf{while}(\mathtt{c})\ \mathtt{b}]\!]_{\mathcal{D}}(\mathcal{M}) & = & \{m \in \mathbf{lfp}\, F_{\mathcal{D}} \mid [\![\mathtt{c}]\!](m) = \mathtt{FALSE}\} \\
\multicolumn{3}{l}{\text{where } F_{\mathcal{D}} : \mathcal{M}' \longmapsto \mathcal{M} \cup [\![\mathtt{b}]\!]_{\mathcal{D}}(\{m \in \mathcal{M}' \mid [\![\mathtt{c}]\!](m) = \mathtt{TRUE}\})} \\
[\![\mathtt{i}_0; \mathtt{i}_1]\!]_{\mathcal{D}}(\mathcal{M}) & = & [\![\mathtt{i}_1]\!]_{\mathcal{D}} \circ [\![\mathtt{i}_0]\!]_{\mathcal{D}}(\mathcal{M})
\end{array}
$$

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

Towards a more general 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 $(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha_{\mathcal{V}}}{\overset{\gamma_{\mathcal{V}}}{\leftrightarrows}} (D^\sharp_{\mathcal{V}}, \sqsubseteq)$

Definitions are quite similar:

Abstraction

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

$$\begin{array}{rrcl} \alpha : & \mathcal{M} & \longmapsto & (\alpha_{\mathcal{V}}(\{\sigma_0 \mid \sigma \in \mathcal{M}\}), \ldots, \alpha_{\mathcal{V}}(\{\sigma_{n-1} \mid \sigma \in \mathcal{M}\})) \\ \gamma : & M^\sharp & \longmapsto & \{\sigma \in \mathbb{Z}^n \mid \forall i,\ \sigma_i \in \gamma_{\mathcal{V}}(M^\sharp_i)\} \end{array}$$

Abstract semantics of sequences (revised)

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

- the semantics $[\![e]\!] : \mathbb{M} \longrightarrow \mathbb{V}$ of an expression evaluates it into a value
- thus, the abstract semantics **should evaluate it into an abstract value**:

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

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

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

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

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 $[\![\mathtt{c}]\!]^\sharp : \mathbb{V}_{\mathrm{bool}} \times D^\sharp \longrightarrow D^\sharp$
- we will implement some when considering specific domains

We will see more general principles soon

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 could 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:

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

Proof of soundness:

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

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 derive, using a **new version of the fixpoint transfer theorem** (exercise):

$$[\![\textbf{while}(\mathrm{c})\ P]\!]^\sharp(M^\sharp) = [\![\mathrm{c}]\!]^\sharp(\mathtt{FALSE}, \mathbf{lfp}_{M^\sharp}\ F^\sharp)$$
$$\text{where } F^\sharp : M_0^\sharp \longmapsto \mathbf{join}^\sharp(M_0^\sharp, [\![P]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{TRUE}, M_0^\sharp)))$$

Computation of abstract iterates:

$$\left\{ \begin{array}{rcl} M_0^\sharp & = & M^\sharp \\ M_{n+1}^\sharp & = & \mathbf{join}^\sharp(M_n^\sharp, [\![P]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{TRUE}, M_n^\sharp))) \end{array} \right.$$

Exit condition: when successive iterates are equal

Static analysis

We can now summarize the definition of our static analysis:

Definition

$$
\begin{array}{rcl}
[\![P_0; P_1]\!]^\sharp(M^\sharp) & = & [\![P_1]\!]^\sharp \circ [\![P_0]\!]^\sharp(M^\sharp) \\
[\![\mathrm{x} = \mathrm{e}]\!]^\sharp(M^\sharp) & = & M^\sharp[\mathrm{x} \leftarrow [\![\mathrm{e}]\!]^\sharp(M^\sharp)] \\
[\![\mathbf{input}()]\!]^\sharp(M^\sharp) & = & M^\sharp[\mathrm{x} \leftarrow \top] \\
[\![\mathbf{if}(\mathrm{c})\, P_0 \,\mathbf{else}\, P_1]\!]^\sharp(M^\sharp) & = & \mathbf{join}^\sharp([\![P_0]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{TRUE}, M^\sharp)), \\
 & & \quad [\![P_1]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{FALSE}, M^\sharp))) \\
[\![\mathbf{while}(\mathrm{c})\, P]\!]^\sharp(M^\sharp) & = & [\![\mathrm{c}]\!]^\sharp(\mathtt{FALSE}, \mathbf{lfp}_{M^\sharp}\, F^\sharp) \\
\multicolumn{3}{c}{\text{where } F^\sharp : M_0^\sharp \longmapsto \mathbf{join}^\sharp(M_0^\sharp, [\![P]\!]^\sharp([\![\mathrm{c}]\!]^\sharp(\mathtt{TRUE}, M_0^\sharp)))}
\end{array}
$$

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**:

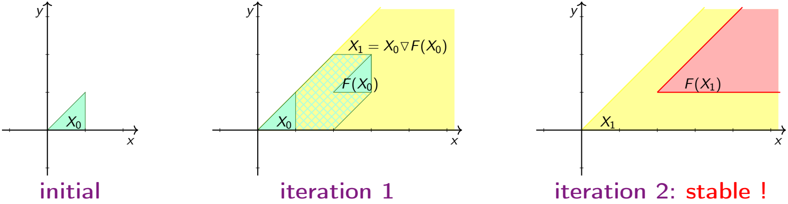
1. look at the base case and following cases
2. try to **generalize them**

Widening iteration: search for inductive abstract properties

Computing invariants about infinite executions with widening ∇

- **Widening** ∇ over-approximates $\cup$: **soundness guarantee**
- **Widening** ∇ guarantees the **termination of the analyses**
- Typical choice of ∇: **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)



- 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**:

- $D^\sharp_\mathcal{V} = \{\bot\} \uplus \mathbb{Z}^\star_+ \uplus \{+\infty\}$; $\gamma_\mathcal{V}(v) = \{0, 1, \ldots, v\}$
- $\forall v^\sharp,\ \bot \sqsubseteq v^\sharp$ 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

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

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 $\mathbf{lfp}\, f = \bigcup_{n \in \mathbb{N}} f^n(\emptyset)$)
- $f \circ \gamma \subseteq \gamma \circ f^\sharp$

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

$$\begin{array}{rcl} M_0^\sharp & = & \bot \\ M_{k+1}^\sharp & = & M_k^\sharp \triangledown f^\sharp(M_k^\sharp) \end{array}$$

Then:

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

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 $\begin{cases} N_0^\sharp & = & \bot \\ N_{k+1}^\sharp & = & f^\sharp(M_k^\sharp) \end{cases}$
   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 $\forall l \geq k,\ f^k(\emptyset) \subseteq \gamma(M_l^\sharp)$:
   - the result clearly holds for $k = 0$;
   - if the result holds at rank k and $l \geq k$ then:

$$\begin{array}{rcll} f^{k+1}(\emptyset) & = & f(f^k(\emptyset)) & \\ & \subseteq & f(\gamma(M_l^\sharp)) & \text{by induction} \\ & \subseteq & \gamma(f^\sharp(M_l^\sharp)) & \text{since } f \circ \gamma \subseteq \gamma \circ f^\sharp \\ & \subseteq & \gamma(M_l^\sharp \triangledown f^\sharp(M_l^\sharp)) & \text{by definition of } \triangledown \\ & = & \gamma(M_{l+1}^\sharp) & \end{array}$$

   When $(M_k^\sharp)_{k\in\mathbb{N}}$ converges, $\forall l \geq K,\ M_l^\sharp = M_K^\sharp = M_\infty^\sharp$, thus
   $\forall k,\ f^k(\emptyset) \subseteq \gamma(M_\infty^\sharp)$ thus $\mathbf{lfp}\, f \subseteq \gamma(M_\infty^\sharp)$

Example widening iteration

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

Example widening iteration

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

Example widening iteration

```
int x = 0;
            x ∈ [0, 0]
while(TRUE){
            x ∈ [0, 0]
    if(x < 10 000){

        x = x + 1;

    } else {

        x = −x;

    }

}
```

Entry into the loop

Example widening iteration

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

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

$\quad\}$

$\}$

Only the "true" branch may be taken

Example widening iteration

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

Incrementation

Example widening iteration

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

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

Now both branches may be taken

Example widening iteration

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

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

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

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

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

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

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

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

$\quad\}\ \textbf{else}\ \{$

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

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

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

$\quad\}$

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

$\}$

Numerical assignments

Example widening iteration

int $\mathtt{x} = 0;$

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

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

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

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

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

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

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

$\}$ **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

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

Both branches may be taken

Example widening iteration

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

Numerical assignments

Example widening iteration

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

$\quad\quad x \in [0, 0]$

$\textbf{while}(\texttt{TRUE})\{$

$\quad\quad x \in]-\infty, +\infty[$

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

$\quad\quad x \in]-\infty, 9999]$

$\quad\quad x = x + 1;$

$\quad\quad x \in]-\infty, 10000]$

$\quad \}\ \textbf{else}\ \{$

$\quad\quad x \in [10000, +\infty[$

$\quad\quad x = -x;$

$\quad\quad x \in]-\infty, -10000]$

$\quad \}$

$\quad\quad x \in]-\infty, 10000]$

$\}$

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**

Post-fixpoint iteration:

- **observation**: if $f \circ \gamma \subseteq \gamma \circ f^\sharp$ and $\mathbf{lfp}\, f \subseteq \gamma(M^\sharp)$, then:
  $\mathbf{lfp}\, f = f(\mathbf{lfp}\, f) \subseteq f \circ \gamma(M^\sharp) \subseteq \gamma \circ f^\sharp(M^\sharp)$
- so $f^\sharp(M^\sharp)$ **also approximates 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;
        $x \in [0, 0]$
while(TRUE){
        $x \in [0, 0]$
    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 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 ∈ ∅
    }
            x ∈ [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 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, 0]
        x = x + 1;
            x ∈ [1, 1]
    } else {
            x ∈ ∅
        x = −x;
            x ∈ ∅
    }
            x ∈ [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 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, 1]
    } else {
                x ∈ ∅
        x = −x;
                x ∈ ∅
    }
                x ∈ [1, 1]
}
```

Still only the true branch is 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, 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 $\mathtt{x} = 0;$
$\qquad \mathtt{x} \in [0, 0]$
while$(\mathtt{TRUE})\{$
$\qquad \mathtt{x} \in [0, 9999]$ instead of $[0, +\infty[$
$\quad$ **if**$(\mathtt{x} < 10\,000)\{$ 9999 will be a threshold value at loop head
$\qquad \mathtt{x} \in [0, 1]$
$\qquad \mathtt{x} = \mathtt{x} + 1;$
$\qquad \mathtt{x} \in [1, 2]$
$\quad \}$ **else** $\{$
$\qquad \mathtt{x} \in \emptyset$
$\qquad \mathtt{x} = -\mathtt{x};$
$\qquad \mathtt{x} \in \emptyset$
$\quad \}$
$\qquad \mathtt{x} \in [1, 2]$
$\}$

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

Now both branches are 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, 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, 10000]
    } else {
            x ∈ ∅
        x = −x;
            x ∈ ∅
    }
            x ∈ [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 $\mathtt{x} = 0$;
    $\mathtt{x} \in [0, 0]$
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** {
    $\mathtt{x} \in \emptyset$
   $\mathtt{x} = -\mathtt{x}$;
    $\mathtt{x} \in \emptyset$
  }
    $\mathtt{x} \in [1, 10000]$
}

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$(\mathtt{TRUE})\{$
        $\mathtt{x} \in [0, 10000]$ 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 $\mathtt{x} = 0;$

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

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

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

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

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

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

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

$\quad \}$ **else** $\{$

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

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

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

$\quad \}$

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

$\}$

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

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

Update on projects...