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 $\gamma$, 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

1. Another Soundness Relation

2. Revisiting Abstract Iteration

3. Conclusion
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** $(C, \subseteq) \leftrightarrow (A, \sqsubseteq)$

Limitations of our previous abstract interpreter:
- uses the best abstraction function $\alpha$ all the time
- tries to establish equality $\lceil P \rceil^\# \circ \alpha = \alpha \circ \lceil P \rceil$ but fails...
  
  indeed, some operators may only compute an over-approximation
- proves $\alpha \circ \lceil P \rceil \subseteq \lceil P \rceil^\# \circ \alpha$
  
  at the cost of proving monotonicity of $\lceil P \rceil^\#$

Alternate approach

**Use $\gamma$ only and prove** $\lceil P \rceil \circ \gamma \subseteq \gamma \circ \lceil P \rceil^\#$
A language with expressions

We now consider the denotational semantics of our imperative language:

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

**Syntax**

$$
e ::= v \ (v \in V) \ | \ x \ (x \in X) \ | \ e + e \ | \ e * e \ | \ldots$$  
expressions

$$
c ::= x < v \ | \ x = v \ | \ldots$$  
basic conditions

$$
i ::= x := e \ |
input(x) \ |
if(c) \ b \ else \ b \ |
while(c) \ b$$  
animation

$$
b ::= \{i; \ldots; i;\}$$  
block, program($P$)
Semantics of expressions and conditions (refresher)

We have defined a few lectures ago:

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

  \[
  \begin{align*}
  \llbracket e \rrbracket : M &\longrightarrow \mathbb{V} \cup \{\Omega\} \\
  \llbracket v \rrbracket (m) & = v \\
  \llbracket x \rrbracket (m) & = m(x) \\
  \llbracket e_0 + e_1 \rrbracket (m) & = \llbracket e_0 \rrbracket (m) \pm \llbracket e_1 \rrbracket (m) \\
  \llbracket e_0 / e_1 \rrbracket (m) & = \begin{cases} 
  \Omega & \text{if } \llbracket e_1 \rrbracket (m) = 0 \\
  \llbracket e_0 \rrbracket (m) \big/ \llbracket e_1 \rrbracket (m) & \text{otherwise}
  \end{cases}
  \end{align*}
  \]

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

  \[
  \llbracket c \rrbracket : M \longrightarrow \mathbb{V}_{\text{bool}} \cup \{\Omega\}
  \]
Semantics of statements (refresher)

We have also defined:

**Denotational semantics of programs**

We use the denotational semantics $[i]_D : \mathcal{P}(M) \to \mathcal{P}(M)$ by:

\[
[x := e]_D(M) = \{ m[x \leftarrow \llbracket e \rrbracket(m)] \mid m \in M \}
\]

\[
[\text{input}(x)]_D(M) = \{ m[x \leftarrow v] \mid v \in \mathcal{V} \land m \in M \}
\]

\[
[\text{if}(c) \ b_0 \ \text{else} \ b_1]_D(M) = [b_0]_D(\{m \in M \mid \llbracket c \rrbracket(m) = \text{TRUE}\}) \cup [b_1]_D(\{m \in M \mid \llbracket c \rrbracket(m) = \text{FALSE}\})
\]

\[
[\text{while}(c) \ b]_D(M) = \{ m \in \text{lfp } F_D \mid \llbracket c \rrbracket(m) = \text{FALSE}\}
\]

where $F_D : M' \mapsto M \cup [b]_D(\{m \in M' \mid \llbracket c \rrbracket(m) = \text{TRUE}\})$

\[
[i_0; i_1]_D(M) = [i_1]_D \circ [i_0]_D(M)
\]

- As before, we seek for an abstract interpretation of $[i]_D$
- 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

\[(\mathcal{P}(\mathbb{Z}), \subseteq) \xrightarrow{\alpha_{\mathcal{V}}} (D_{\mathcal{V}}^\#, \sqsubseteq)\]

Definitions are quite similar:

**Abstraction**

- **concrete domain:** \((\mathcal{P}(X \rightarrow \mathbb{Z}), \subseteq)\)
- **abstract domain:** \((D^\#, \sqsubseteq)\), where \(D^\# = X \rightarrow D_{\mathcal{V}}^\#\) and \(\sqsubseteq\) is the pointwise ordering
- **Galois connection** \((\mathcal{P}(\mathbb{Z}), \subseteq) \xrightarrow{\gamma} (D^\#, \sqsubseteq)\), defined by

\[
\alpha : \mathcal{M} \mapsto (\alpha_{\mathcal{V}}(\{\sigma_0 \mid \sigma \in \mathcal{M}\}), \ldots, \alpha_{\mathcal{V}}(\{\sigma_{n-1} \mid \sigma \in \mathcal{M}\}))
\]

\[
\gamma : M^\# \mapsto \{\sigma \in \mathbb{Z}^n \mid \forall i, \sigma_i \in \gamma_{\mathcal{V}}(M_i^\#)\}
\]
Abstract semantics of sequences (revised)

We search for an abstract semantics \([P]^{#} : D^{#} \rightarrow D^{#}\) such that:

\[ [P] \circ \gamma \subseteq \gamma \circ [P]^{#} \]

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

Sequences / composition forced us to require monotonicity last time:
- we assume \([P_0] \circ \gamma \subseteq \gamma \circ [P_0]^{#}\)
- we assume \([P_1] \circ \gamma \subseteq \gamma \circ [P_1]^{#}\)
- since \([P_0; P_1] = [P_1] \circ [P_0]\), we search for something similar in the abstract level
  \[ [P_1] \circ [P_0] \circ \gamma \subseteq [P_1] \circ \gamma \circ [P_0]^{#} \]
  \[ \subseteq \gamma \circ [P_1]^{#} \circ [P_0]^{#} \] (by induction)

No more requirement that \([P]^{#}\) be monotone (much better!)
Abstract semantics of expressions

Analysis of an expression

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

$$[e]^{\#} : D^{\#} \rightarrow D^{\#}_{\mathbb{V}}$$

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

- **abstraction for constants:**
  i.e., a function $\phi_{\mathbb{V}} : \mathbb{V} \rightarrow D^{\#}_{\mathbb{V}}$ such that $\forall v \in \mathbb{V}, v \in \gamma_{\mathbb{V}}(\phi_{\mathbb{V}}(v))$
  note: if $\alpha_{\mathbb{V}}$ exists, then we may take $v \mapsto \alpha_{\mathbb{V}}(\{v\})$
  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 $\oplus^{\#}$ such that:

$$\forall v_0^{\#}, v_1^{\#} \in D^{\#}_{\mathbb{V}}, \{v_0 \oplus v_1 \mid \forall i, v_i \in \gamma_{\mathbb{V}}(v_i^{\#})\} \subseteq \gamma_{\mathbb{V}}(v_0^{\#} \oplus^{\#} v_1^{\#})$$
Abstract semantics of expressions

Analysis of expressions: definition

We define $\llbracket e \rrbracket^\#: D^\# \rightarrow D^V_\#$ by:

$$
\llbracket v \rrbracket^\#(M^\#) = \phi_V(v) \\
\llbracket x \rrbracket^\#(M^\#) = M^\#(x) \\
\llbracket e_0 \oplus e_1 \rrbracket^\#(M^\#) = \llbracket e_0 \rrbracket^\#(M^\#) \oplus^\# \llbracket e_1 \rrbracket^\#(M^\#)
$$

Analysis of expressions: soundness

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

$$
\forall m \in \gamma(M^\#), \llbracket e \rrbracket(m) \text{ returns no error } \implies \llbracket e \rrbracket(m) \in \gamma_V(\llbracket e \rrbracket^\#(M^\#))
$$

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:

\[
[x = e]^\#(M^#) = M^#[x \leftarrow [e]^\#(M^#)]
\]

- soundness proof is very similar
- but now, is given in terms of $\gamma$
Abstract semantics of conditions

Analysis of a condition

- the semantics \([c] : M \rightarrow \mathbb{V}_{\text{bool}}\) of a condition evaluates it into a boolean value (or an error)

- **but** the semantics relies on its functional inverse: e.g., \(\{ m \in M | [c](m) = \text{TRUE} \}\) or \(\{ m \in M | [c](m) = \text{FALSE} \}\)

- thus, the abstract semantics **should tell which memories satisfy a condition**:

\[
[c]^{\#} : \mathbb{V}_{\text{bool}} \times D^{\#} \rightarrow D^{\#}
\]

\[\forall b \in \mathbb{V}_{\text{bool}}, \forall m \in \gamma(M^{\#}), [c](m) = b \implies m \in \gamma([c]^{\#}(b, M^{\#}))\]

- we assume that the abstract domain provides such a function

\[
[c]^{\#} : \mathbb{V}_{\text{bool}} \times D^{\#} \rightarrow D^{\#}
\]

- 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** $\text{join}^\#$, over the value abstract domain:

$$\forall v_0^#, v_1^#, \gamma(v_0^#) \cup \gamma(v_1^#) \subseteq \gamma(\text{join}^\#(v_0^#, v_1^#))$$

it may be $\sqsubseteq_\gamma$ if it exists, but could over-approximate it
- we let $\text{join}^#$ be the pointwise extension of $\text{join}^\#_\gamma$
- it is also sound: $\forall M_0^#, M_1^#, \gamma(M_0^#) \cup \gamma(M_1^#) \subseteq \gamma(\text{join}^#(M_0^#, M_1^#))$

We derive:

$$\llbracket \text{if}(c) \ P_0 \ \text{else} \ P_1 \rrbracket^#(M^#) =$$

$$\text{join}^#(\llbracket P_0 \rrbracket^#(\llbracket c \rrbracket^#(\text{TRUE}, M^#)), \llbracket P_1 \rrbracket^#(\llbracket c \rrbracket^#(\text{FALSE}, M^#)))$$

**Proof of soundness:**
- similar as in the previous course
- relies on the soundness of $\llbracket c \rrbracket^#$, $\llbracket P_0 \rrbracket^#$, $\llbracket P_1 \rrbracket^#$ and $\text{join}^#$
Another Soundness Relation

Analysis of a loop

Again, quite similar to the previous course:

- statement \(\textbf{while}(c) \ P\), with abstract pre-condition \(M^\#\)
- we assume \(\llbracket c \rrbracket^\#\) and \(\llbracket P \rrbracket^\#\) sound abstract semantics for the condition and the loop body
- we derive, using a \textit{new version of the fixpoint transfer theorem} (exercise):

\[
\llbracket \textbf{while}(c) \ P \rrbracket^\#(M^\#) = \llbracket c \rrbracket^\#(\text{FALSE}, \text{lfp}_{M^\#} F^\#) \\
\]

where \(F^\# : M^\#_0 \mapsto \text{join}^\#(M^\#_0, \llbracket P \rrbracket^\#(\llbracket c \rrbracket^\#(\text{TRUE}, M^\#_0)))\)

Computation of abstract iterates:

\[
\begin{align*}
M^\#_0 &= M^\# \\
M^\#_{n+1} &= \text{join}^\#(M^\#_n, \llbracket P \rrbracket^\#(\llbracket c \rrbracket^\#(\text{TRUE}, M^\#_n)))
\end{align*}
\]

Exit condition: when successive iterates are equal
Static analysis

We can now summarize the definition of our static analysis:

**Definition**

\[
\begin{align*}
[P_0; P_1](M) &= [P_1] \circ [P_0](M) \\
[x = e](M) &= M[x \leftarrow [e](M)] \\
[input()](M) &= M[x \leftarrow \top] \\
[\text{if}(c) P_0 \text{ else } P_1](M) &= \text{join}([P_0](\text{if}(c)(\text{TRUE}, M)), [P_1](\text{if}(c)(\text{FALSE}, M))) \\
[\text{while}(c) P](M) &= \text{if}(c)(\text{FALSE}, \text{lfp} M F) \\
\text{where } F : M_0 \rightarrow \text{join}(M_0, [P](\text{if}(c)(\text{TRUE}, M_0)))
\end{align*}
\]

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

**Soundness**

For all program \( P \), \( \forall M \in D, [P] \circ \gamma(M) \subseteq \gamma \circ [P](M) \)
Outline

1. Another Soundness Relation
2. Revisiting Abstract Iteration
3. Conclusion
Limitations related to abstract iteration

We need a finite height lattice:
- otherwise the computation of $\text{Ifp } F^\#$ 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 $D^\#$ 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 $\nabla$
- **Widening** $\nabla$ over-approximates $\cup$: **soundness guarantee**
- **Widening** $\nabla$ guarantees the **termination of the analyses**
- Typical choice of $\nabla$: **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 $0^\circ$ or $45^\circ$ angle)

- Initially: 3 constraints
- After one iteration: 2 constraints, then stable
Widening operator: Definition

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

- $\forall M^0_0, M^0_1, \gamma(M^0_0) \cup \gamma(M^0_1) \subseteq \gamma(M^0_0 \nabla M^0_1)$
- if $(N^k_k)_{k \in \mathbb{N}}$ is a sequence of elements of $D^\#$ the sequence $(M^k_k)_{k \in \mathbb{N}}$ defined below is stationary:

\[
\begin{align*}
M^0_0 &= N^0_0 \\
M^k_{k+1} &= M^k_k \nabla N^k_{k+1}
\end{align*}
\]

- **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
Theorem

We assume that \((D\#, \subseteq)\) is a finite height domain and that \(\sqcup\) is the least upper bound over \(D\#\). Then \(\sqcup\) defines a widening over \(D\#\).

Proof:

1. since \(M_0\# \subseteq M_0\# \sqcup M_1\#\), we have \(\gamma(M_0\#) \subseteq \gamma(M_0\# \sqcup M_1\#)\)

2. a sequence of iterates \((M_k\#)_{k \in \mathbb{N}}\) 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:

- \( D^\#_V = \{\perp\} \uplus \mathbb{Z}_+^* \uplus \{+\infty\} \); \( \gamma_V(v) = \{0, 1, \ldots, v\} \)
- \( \forall v^\#, \perp \sqsubseteq v^\# \) and if \( v_0^\# \leq v_1^\# \), then \( v_0^\# \sqsubseteq v_1^\# \)

We define the widening operator below:

### Widening operator

\[
\begin{align*}
\perp \triangledown v^\# &= v^\# \\
v^\# \triangledown \perp &= v^\# \\
v_0^\# \triangledown v_1^\# &= \begin{cases} v_0^\# & \text{if } v_0^\# \geq v_1^\# \\ +\infty & \text{if } v_0^\# < v_1^\# \end{cases}
\end{align*}
\]

#### 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 \(\gamma : A \rightarrow C\) and a widening operator \(\nabla\). Moreover, we assume that:

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

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

\[
\begin{align*}
M_0^\# & = \bot \\
M_k^\# & = M_k^\# \nabla f^\#(M_k^\#)
\end{align*}
\]

Then:

1. \((M_k^\#)_{k \in \mathbb{N}}\) is stationary and we write \(M^\#_{\text{lim}}\) for its limit
2. \(\text{lfp} f \subseteq \gamma(M^\#_{\text{lim}})\)
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
   \[
   \begin{align*}
   N_0^\# &= \bot \\
   N_k^\# &= f^\#(M_k^\#)
   \end{align*}
   \]
   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 $\forall l \geq k$, $f^k(\emptyset) \subseteq \gamma(M_i^\#)$:
   - the result clearly holds for $k = 0$;
   - if the result holds at rank $k$ and $l \geq k$ then:
     \[
     f^{k+1}(\emptyset) = f(f^k(\emptyset)) \\
     \subseteq f(\gamma(M_i^\#)) \quad \text{by induction} \\
     \subseteq \gamma(f^\#(M_i^\#)) \quad \text{since } f \circ \gamma \subseteq \gamma \circ f^\# \\
     \subseteq \gamma(M_i^\# \triangledown f^\#(M_i^\#)) \quad \text{by definition of } \triangledown \\
     = \gamma(M_{i+1}^\#)
     \]
   When $(M_k^\#)_{k \in \mathbb{N}}$ converges, $\forall l \geq K$, $M_l^\# = M_K^\# = M_\infty^\#$, thus
   $\forall k$, $f^k(\emptyset) \subseteq \gamma(M_\infty^\#)$ thus $\text{lfp } f \subseteq \gamma(M_\infty^\#)$
Example widening iteration

```c
int x = 0;

while(TRUE){
    if(x < 10000){
        x = x + 1;
    } else {
        x = -x;
    }
}
```
Example widening iteration

```c
int x = 0;
    x ∈ [0, 0]
while(TRUE){
    if(x < 10000){
        x = x + 1;
    } else {
        x = −x;
    }
}
```
Example widening iteration

```c
int x = 0;
    x ∈ [0, 0]
while(TRUE){
    x ∈ [0, 0]
    if(x < 10000){
        x = x + 1;
    } else {
        x = -x;
    }
}
```

Entry into the loop
Example widening iteration

```c
int x = 0;
    x ∈ [0, 0]
while(TRUE){
    x ∈ [0, 0]
    if(x < 10000){
        x ∈ [0, 0]
        x = x + 1;
    }
    else {
        x ∈ ∅
        x = −x;
    }
}

Only the “true” branch may be taken
```
Example widening iteration

```c
int x = 0;
x ∈ [0, 0]
while(TRUE){
x ∈ [0, 0]
if(x < 10000){
x ∈ [0, 0]
x = x + 1;
x ∈ [1, 1]
}
else {
x ∈ ∅
x = -x;
x ∈ ∅
}
}
```

Incrementation
Example widening iteration

```c
int x = 0;
    x ∈ [0, 0]
while(TRUE){
    x ∈ [0, 0]
    if(x < 10000){
        x ∈ [0, 0]
        x = x + 1;
        x ∈ [1, 1]
    } else {
        x ∈ ∅
        x = −x;
        x ∈ ∅
    }
}
    x ∈ [1, 1]
}
```

Abstract union at the end of the condition
Revisiting Abstract Iteration

Example widening iteration

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

Widening at loop head
Example widening iteration

```c
int x = 0;
    x ∈ [0, 0]
while(TRUE){
    x ∈ [0, +∞[
    if(x < 10000){
        x ∈ [0, 9999]
        x = x + 1;
        x ∈ [1, 1]
    } else {
        x ∈ [10000, +∞[
        x = -x;
        x ∈ ∅
    }
} x ∈ [1, 1]
```
Example widening iteration

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

Numerical assignments
Example widening iteration

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

Abstract union at the end of the condition
```
Example widening iteration

```c
int x = 0;
x ∈ [0, 0]
while(TRUE){
x ∈ ]−∞, +∞[
if(x < 10000){
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

```c
int x = 0;
    x ∈ [0, 0]
while(TRUE){
    x ∈ ]−∞, +∞[
    if(x < 10000){
        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

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

Numerical assignments
Example widening iteration

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

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

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 $l$

$$
\begin{align*}
M_0^\# &= \bot \\
M_{k+1}^\# &= \text{join}^\#(M_k^\#, f^\#(M_k^\#)) \quad \text{if } k < l \\
M_{k+1}^\# &= M_k^\# \nabla f^\#(M_k^\#) \quad \text{otherwise}
\end{align*}
$$

- 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 \( \nabla \) operator of intervals goes straight to \( \infty \)
- 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:

\[
\begin{align*}
\bot \nabla \nu^\# &= \nu^# \\
\nu^# \nabla \bot &= \nu^#
\end{align*}
\]

\[
\nu_0^# \nabla \nu_1^# = \begin{cases} 
\nu_0^# & \text{if } \nu_0^# \geq \nu_1^# \\
\min\{\nu^# \in \mathcal{T} \mid \forall i, \nu_i^# \leq \nu^#\} & \text{if } \{\nu^# \in \mathcal{T} \mid \forall i, \nu_i^# \leq \nu^#\} \neq \emptyset \\
+\infty & \text{otherwise}
\end{cases}
\]

- **Proof** of the widening property: exercise
- **Example** with \( \mathcal{L} = \{10\} \):
  \[
  [0, 8] \nabla [0, 9] = [0, 10] \quad [0, 8] \nabla [0, 15] = [0, +\infty[\]
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 \( f \circ \gamma \subseteq \gamma \circ f^\# \) and \( \text{lfp } f \subseteq \gamma(M^\#) \), then:
  \[ \text{lfp } f = f(\text{lfp } f) \subseteq f \circ \gamma(M^\#) \subseteq \gamma \circ f^\#(M^\#) \]
- so \( f^\#(M^\#) \) also approximates \( \text{lfp } f \), 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

```c
int x = 0;

while (TRUE) {
    if (x < 10000) {
        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

```plaintext
int x = 0;
x ∈ [0, 0]
while(TRUE){
    if(x < 10000){
        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

```c
int x = 0;
x ∈ [0, 0]
while(TRUE){
x ∈ [0, 0]
if(x < 10000){
    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

```plaintext
int x = 0;
    x ∈ [0, 0]
while(TRUE){
    x ∈ [0, 0]
    if(x < 10000){
        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

```c
int x = 0;
    x ∈ [0, 0]
while (TRUE) {
    x ∈ [0, 0]
    if (x < 10000) {
        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

```c
int x = 0;
x ∈ [0, 0]
while(TRUE){
x ∈ [0, 0]
if(x < 10000){
x ∈ [0, 0]
x = x + 1;
x ∈ [1, 1]
} else {
x ∈ ∅
x = −x;
x ∈ ∅
}
x ∈ [1, 1]
}
```

Propagation
Revisiting Abstract Iteration

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

```plaintext
int x = 0;
    x ∈ [0, 0]
while (TRUE) {
    x ∈ [0, 1]
    if (x < 10000) {
        9999 will be a threshold value at loop head
        x ∈ [0, 0]
        x = x + 1;
        x ∈ [1, 1]
    } else {
        x ∈ ∅
        x = -x;
        x ∈ ∅
    }
} else {
    x ∈ [1, 1]
}
```

Join at loop head

Xavier Rival
Abstract Interpretation: Introduction
April 6th, 2018
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

```c
int x = 0;
    x ∈ [0, 0]
while(TRUE){
    x ∈ [0, 1]
    if(x < 10000){
        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
Revisiting Abstract Iteration

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

```c
int x = 0;
x ∈ [0, 0]
while(TRUE){
x ∈ [0, 1]
if(x < 10000){
x ∈ [0, 1]
x = x + 1;
x ∈ [1, 2]
} else {
x ∈ ∅
x = −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

```c
int x = 0;
    x ∈ [0, 0]
while (TRUE) {
    x ∈ [0, 1]
    if (x < 10000) {
        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

```c
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 ∈ ∅
    }
} else {
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;

while (TRUE) {
    x ∈ [0, 0]
    if (x < 10000) {
        x ∈ [0, 9999]
        9999 will be a threshold value at loop head
        x = x + 1;
        x ∈ [1, 2]
    } else {
        x ∈ ∅
        x = −x;
        x ∈ ∅
    }
    x ∈ [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

```java
int x = 0;
    x ∈ [0, 0]
while(TRUE){
    x ∈ [0, 9999]  instead of [0, +∞]  
    if(x < 10000){
        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

```c
int x = 0;
x ∈ [0, 0]
while(TRUE){
    x ∈ [0, 9999] instead of [0, +∞]
    if(x < 10000){
        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, 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

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

```plaintext
int x = 0;
x ∈ [0, 0]
while (TRUE) {
    x ∈ [0, 10000] instead of ] −∞, +∞[
    if (x < 10000) {
        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 ∈ ∅
    }
}  
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

```c
int x = 0;
    x ∈ [0, 0]
while(TRUE){
    x ∈ [0, 10000] instead of ] − ∞, +∞[
    if(x < 10000){
        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

```python
int x = 0;
x ∈ [0, 0]
while (TRUE) {
    x ∈ [0, 10000] instead of ] −∞, +∞[
    if (x < 10000) {
        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]
    }
}
```

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

```c
int x = 0;
x ∈ [0, 0]
while (TRUE) {
    x ∈ [−10000, 10000] instead of ] − ∞, +∞[
    if (x < 10000) {
        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

```c
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

```plaintext
int x = 0;
x ∈ [0, 0]
while(TRUE){
x ∈ [−10000, 10000] instead of ] − ∞, +∞[
    if(x < 10000){
        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

```plaintext
int x = 0;
  x ∈ [0, 0]
while (TRUE) {
  x ∈ [−10000, 10000]  instead of ] − ∞, +∞[
  if (x < 10000) {
    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
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 \((D^\#, \sqsubseteq)\) be an infinite height domain, with a widening \(\nabla\) that is stable \((\forall v^\#, v^\# \nabla v^\# = v^\#)\) and such that \(\forall v_0^\#, v_1^\#, \forall i, v_i^\# \sqsubseteq v_0^\# \nabla v_1^\#.\) Then, \(\nabla\) is **not monotone in its first argument** (proof: Patrick Cousot).

**Proof:** we assume it is, let \(w_0^\# \sqsubseteq w_1^\# \sqsubseteq \ldots\) be an infinite chain over \(D^\#\) and define \(v_0^\# = w_0^\#, v_{k+1}^\# = v_k^\# \nabla w_{k+1}^\#\); we prove by induction that \(v_k^\# = w_k^\#:\)

- clear at rank 0
- we assume that \(v_k^\# = w_k^\#:\) then \(v_{k+1}^\# = v_k^\# \nabla w_{k+1}^\#,\) so \(w_{k+1}^\# \sqsubseteq v_{k+1}^\#;\)
  - moreover, \(v_{k+1}^\# = v_k^\# \nabla w_{k+1}^\# = w_k^\# \nabla w_{k+1}^\# \sqsubseteq w_{k+1}^\# \nabla w_{k+1}^\# = w_{k+1}^\#\)

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

1. Another Soundness Relation
2. Revisiting Abstract Iteration
3. Conclusion
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...