The structure of an abstract interpreter
Static Analysis by Abstract Interpretation

Xavier Rival

East China Normal University

November, 25th. 2013
Design of an abstract interpreter

We follow the abstract interpretation methodology:

1. we set up a concrete semantics
2. we fix an abstraction
3. we derive an abstract interpreter

Focus of this lecture

- Discuss two families of abstract interpreters
  - following the structure of the control flow graph
  - following the structure of the abstract syntax tree
- Analyze an imperative language
  - first, we start with a minimalistic imperative language
  - then, we extend it gradually
Outline

1. A small imperative language
2. Reachable states and abstract interpretation
3. Abstract interpretation of the denotational semantics
4. Static analysis with nested variables scopes
5. Static analysis in presence of procedures and functions
6. Static analysis in presence of branching instructions
7. Conclusion
Programs

For now, we allow only a very limited set of statements:
assignments, conditions, loops

Syntactic elements:
- **Variables** $X$: finite, predefined set of variables
- **Labels** $L$: before and after each statement
- **Values** $V$: $V_{\text{int}} \cup V_{\text{float}} \cup \ldots$
- $e$ ranges over arithmetic expressions
  
  
  $$e ::= c \in V_{\text{int}} \cup V_{\text{float}} \cup \ldots \mid e + e \mid e * e \mid \ldots$$

Syntax

$$
\begin{align*}
  i & ::= x := e ; & \text{assignment} \\
      & | \quad \text{if}(c) \ b \ \text{else} \ b & \text{condition} \\
      & | \quad \text{while}(c) \ b & \text{loop} \\

  b & ::= \{ i ; \ldots ; i ; \} & \text{block}
\end{align*}
$$
States

At one point in the execution, we can observe:

- a control state \( l \in L \);
- a memory state \( m \), mapping each variable into a value

\[
m \in M, \text{ where } M = X \rightarrow V
\]

A program can also crash: we add error state \( \Omega \)

**Definition: states**

\[
S = (L \times M) \cup \{\Omega\}
\]

**Initial states** \( S_I \): each variable may take any value

- \( l_{\text{init}} \): entry point
- \( S_I = \{(l_{\text{init}}, m) \mid m \in M\} \)
A small imperative language

Transition relation (1/2)

Semantics of expressions:

- \([e] : M \rightarrow \mathbb{V} \cup \Omega\) (or \(P(\mathbb{V} \cup \{\Omega\})\)) if **non determinism**
- it should be defined by induction over the syntax of expressions...

A program execution step is a transition \(s_0 \rightarrow s_1\)

Definition of \(\rightarrow\):

- case of \(\ell_0 : x = e; \ell_1\)
  - if \([e](m) \neq \Omega\), then \((\ell_0, m) \rightarrow (\ell_1, m[x \leftarrow [e](m)])\)
  - if \([e](m) = \Omega\), then \((\ell_0, m) \rightarrow \Omega\)

- case of \(\ell_0 : \text{if}(c)\{\ell_1 : \text{b}_t \ell_2\} \text{ else}\{\ell_3 : \text{b}_f \ell_4\} \ell_5\)
  - if \([e](m) = \text{true}\), then \((\ell_0, m) \rightarrow (\ell_1, m)\)
  - if \([e](m) = \text{false}\), then \((\ell_0, m) \rightarrow (\ell_3, m)\)
  - if \([e](m) = \Omega\), then \((\ell_0, m) \rightarrow \Omega\)
  - \((\ell_2, m) \rightarrow (\ell_5, m)\)
  - \((\ell_4, m) \rightarrow (\ell_5, m)\)
Definition of $\rightarrow$ (continued)

- case of $l_0 : \textbf{while}(c)\{l_1 : b_t \ l_2 \ l_3$
  - if $\lbrack e \rbrack(m) = \textbf{true}$, then $(l_0, m) \rightarrow (l_1, m)$
  - $(l_2, m) \rightarrow (l_1, m)$
  - if $\lbrack e \rbrack(m) = \textbf{false}$, then $(l_0, m) \rightarrow (l_3, m)$
  - $(l_2, m) \rightarrow (l_3, m)$
  - if $\lbrack e \rbrack(m) = \Omega$, then $(l_0, m) \rightarrow \Omega$
  - $(l_2, m) \rightarrow \Omega$

- case of $\{l_0 : i_0; l_1 : \ldots; l_{n-1}i_{n-1}; l_n\}$
  - trivial...
Static analysis problem

- verify the absence of run-time errors i.e., that $\Omega$ is not reachable
- OR verify another safety property, e.g., that $x$ is positive at all times

In all cases, we would like to compute an over-approximation of the reachable states

Definition: reachable states $\llbracket P \rrbracket^R$

$$\llbracket P \rrbracket^R = \{ s_n \mid \exists s_0 \in S_I, s_1, s_2, \ldots, s_{n-1} \in S, \forall i, s_i \rightarrow s_{i+1} \}$$
$$= \text{lfp}_{S_I} F_P$$

where

$$F_P : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$$

$$X \mapsto X \cup \{ s' \mid \exists s \in X, s \rightarrow s' \}$$
Outline

1. A small imperative language
2. Reachable states and abstract interpretation
3. Abstract interpretation of the denotational semantics
4. Static analysis with nested variables scopes
5. Static analysis in presence of procedures and functions
6. Static analysis in presence of branching instructions
7. Conclusion
Abstraction (1/3)

Assumption: an abstract domain $\mathcal{D}_{\text{M}}$ for memory states

- abstract lattice $\mathcal{D}_{\text{M}}$ with order $\subseteq_{\text{M}}$
- galois connection

$$(\mathcal{P}(\text{M}), \subseteq) \xleftrightarrow{\alpha_{\text{M}}} (\mathcal{D}_{\text{M}} \wedge_{\text{M}}, \subseteq_{\text{M}})$$

Notes:

- in this lecture, we assume a Galois connection (more on this in another lecture)
- we will use interval constraints as examples
- in the examples, we do not detail the definition of $\mathcal{D}_{\text{M}}$ (there are three lectures on numerical abstract domains)
Abstraction (2/3)

We now need to define the abstraction $\alpha(S)$ of $S \subseteq \mathbb{S}$

What to do with labels?

- **ignore labels: flow insensitive abstraction**
  \[
  \alpha(S) = \alpha_M(\{m \in M \mid \exists \ell \in \mathbb{L}, (\ell, m) \in S\})
  \]
  - advantage: very cheap, one abstract value for the whole program
  - disadvantage: very imprecise

- **keep labels: flow sensitive abstraction**
  \[
  \alpha(S) : \mathbb{L} \rightarrow \mathbb{D}_M^\#
  \]
  \[
  \ell \mapsto \alpha_M(\{m \in M \mid (\ell, m) \in S\})
  \]
  - advantage: very precise
  - disadvantage: more costly

We use the context-sensitive abstraction
Abstraction (3/3)

Flow sensitive abstract, galois connection

We define:

- the **domain** $D^# = L \rightarrow D^#_M$
- the **order relation** $\subseteq^#$ by $X^# \subseteq^# Y^# \iff \forall l \in L, \ X^#(l) \subseteq^#_M Y^#(l)$
- the **abstraction**
  $$\alpha : \mathcal{P}(S) \longrightarrow D^#$$
  $$l \longmapsto \alpha_M(\{m \in M | (l, m) \in S\})$$
- the **concretization**
  $$\gamma : D^# \longrightarrow \mathcal{P}(S)$$
  $$X^# \longmapsto \{ (l, m) \in S | m \in \gamma_M(X^#(l)) \}$$

Then, we have a Galois-connection $(\mathcal{P}(S), \subseteq) \xleftarrow{\gamma} (D^#, \subseteq^#)$
Abstract interpretation

How to achieve soundness

- we search for an abstract semantics \([P]^{\#}\) such that \([P] \subseteq \gamma([P]^{\#})\)
- to do this, we search to apply a fixpoint approximation theorem
  - the concrete domain is a complete lattice
  - \(F_P\) is monotonic
  - we need to define \(F_P^{\#}\) such that
    \[
    \forall X^{\#} \in \mathbb{D}^{\#}, \ F_P \circ \gamma(X^{\#}) \subseteq \gamma \circ F_P^{\#}(X^{\#})
    \]

Since \(F_P(X) = X \cup \{s' \mid \exists s \in X, \ s \to s'\}\), we need:

- an over-approximation \(\cup^{\#}\) of \(\cup\) in the abstract
  \[
  \forall X_0^{\#}, X_1^{\#} \in \mathbb{D}^{\#}, \ \gamma(X_0^{\#}) \cup \gamma(X_1^{\#}) \subseteq \gamma(X_0^{\#} \cup^{\#} X_1^{\#})
  \]
- an over-approximation \(\text{post} : \mathbb{D}^{\#} \to \mathbb{D}^{\#}\) of \(\to\) in the abstract
  \[
  \forall X^{\#} \in \mathbb{D}^{\#}, \ \forall s_0 \in \gamma(X^{\#}), \ \forall s_1 \in S, \ s_0 \to s_1 \implies s_1 \in \gamma(\text{post}(X^{\#}))
  \]
Abstract interpretation

Then:

\[ F_P \circ \gamma(X^\#) = \gamma(X^\#) \cup \{ s' \mid \exists s \in \gamma(X^\#), s \rightarrow s' \} \]
\[ \subseteq \gamma(X^\#) \cup \gamma(post(X^\#)) \]
\[ \subseteq \gamma(X^\# \sqcup^\# post(X^\#)) \]

Thus, we simply need to let \( F_P^\# \) be defined by

**Abstract semantic function**

\[ F_P^\# : D^\# \rightarrow D^\# \]
\[ X^\# \rightarrow X^\# \sqcup^\# post(X^\#) \]

... but we still need to define \( \sqcup^\#, post \)
Approximation of join

We want to approximate:

$$\gamma(X_0^\#) \cup \gamma(X_1^\#) = \{(l, m) \in S \mid m \in \gamma_M(X_0^\#)(l) \cup \gamma_M(X_1^\#)(l)\}$$

Thus, we simply let

- $\sqcup_M$ denote a sound over-approximation of union in $D_M^\#$:
  $$\forall m_0^\#, m_1^\# \in D_M^\#, \gamma_M(m_0^\#) \cup \gamma_M(m_1^\#) \subseteq \gamma_M(m_0^\# \sqcup_M m_1^\#)$$

- $\sqcup^\#$ is then defined by: $(X_0^\# \sqcup^\# X_1^\#) : l \mapsto X_0^\#(l) \sqcup_M X_1^\#(l)$

This definition provides a sound $\sqcup^\#$.
Abstraction of transitions

- similarly, we search for a definition of \( \text{post} \), that satisfies the soundness condition
- to do that, we fix \( X^\# \in D^\# \), and study \( \text{post}(X^\#) \)

\[
\forall s_0 \in \gamma(X^\#), \forall s_1 \in S, \quad s_0 \rightarrow s_1 \implies s_1 \in \gamma(\text{post}(X^\#))
\]

i.e.,

\[
\forall (l_0, m_0) \in \gamma(X^\#), \forall (l_1, m_1) \in S, \quad (l_0, m_0) \rightarrow (l_1, m_1) \implies (l_1, m_1) \in \gamma(\text{post}(X^\#))
\]

i.e.,

\[
\forall l_0, l_1 \in L, \forall m_0 \in \gamma_M(X^\#(l_0)), \forall m_1 \in M, \quad (l_0, m_0) \rightarrow (l_1, m_1) \implies m_1 \in \gamma_M(\text{post}(X^\#)(l_1))
\]

Thus we need a set of sound transfer functions \( (\delta_{l_0,l_1}^\#) \):

**Sound transfer functions**

\[
\forall m^\# \in D_{M}^\#, \forall m_0 \in m^\#, \forall m_1 \in \gamma_M(m^\#), \quad (l_0, m_0) \rightarrow (l_1, m_1) \implies m_1 \in \gamma_M(\delta_{l_0,l_1}^\#(m^\#))
\]

This ensure soundness of \( \text{post} : X^\# \mapsto (l_1 \mapsto \sqcup \{ \delta_{l_0,l_1}^\#(X^\#(l_0)) \mid l_0 \in L \} ) \)
Transfer functions

**Assignment** \( l_0 : x = e; \ l_1 \)

Then, \( \delta^{#}_{l_0, l_1} : m^{#} \mapsto assign(x, e, m^{#}) \) where

**Assignment transfer function**

\[ \forall m \in \gamma_M(m^{#}), m[x \leftarrow \llbracket e \rrbracket(m)] \in \gamma_M(assign(x, e, m^{#})) \]

**Condition test** \( l_0 : \text{if}(c)\{l_1 : b_t \ l_2 \} \ \text{else}\{l_3 : b_f \ l_4 \} \ l_5 \)

Then,

\[ \delta^{#}_{l_0, l_1} : m^{#} \mapsto test(c, m^{#}) \]
\[ \delta^{#}_{l_0, l_3} : m^{#} \mapsto test(\neg c, m^{#}) \]
\[ \delta^{#}_{l_2, l_5} : m^{#} \mapsto m^{#} \]
\[ \delta^{#}_{l_2, l_5} : m^{#} \mapsto m^{#} \]

where

**Assignment transfer function**

\[ \forall m \in \gamma_M(m^{#}), \llbracket c \rrbracket(m) = \text{true} \implies m \in \gamma_M(test(c, m^{#})) \]
Fixpoint approximation

- abstract domain $D^\#_M$ may not be of finite height, so \textit{widening} is needed to enforce termination of abstract iterates
- widening operator for $D^\#$ is defined from a widening for $D^\#_M$ in the same way as $\sqcup^\#$

\textbf{Widening}

$\nabla^\#_M$ should satisfy the conditions below:

- \textbf{soundness}: $\gamma^\#_M(m^\#_0) \cup \gamma^\#_M(m^\#_1) \subseteq \gamma^\#_M(m^\#_0 \nabla^\#_M m^\#_1)$

- \textbf{termination}: for all sequence $(m^\#_i)_{i \in \mathbb{N}}$, the sequence $(y^\#_i)_{i \in \mathbb{N}}$ defined by $y^\#_0 = m^\#_0$ and $y^\#_{n+1} = y^\#_n \nabla^\#_M m^\#_{n+1}$ is ultimately stationary

\textbf{Notes:}

- this is our first abstract interpreter, not a very clever or efficient one
- it does a lot of unnecessary work; should do chaotic iterations instead
- we are going to introduce another one in a few slides
Abstract interpretation of the denotational semantics

Outline

1. A small imperative language
2. Reachable states and abstract interpretation
3. Abstract interpretation of the denotational semantics
4. Static analysis with nested variables scopes
5. Static analysis in presence of procedures and functions
6. Static analysis in presence of branching instructions
7. Conclusion
Abstract interpretation of the denotational semantics

Denotational semantics

We wish to:

- avoid the issues with iteration strategies order
- use an intuitive, efficient way to compute the abstract semantics

Definition: denotational semantics

\[
\llbracket P \rrbracket_D : \mathcal{P}(S) \longrightarrow \mathcal{P}(S) \\
X \longmapsto \{ s' \in S | \exists s \in X, s \rightarrow^* s' \}
\]

We can give an alternate, lower level definition:

Given \( \ell_0 : P; \ell_1 \) (\( \ell_0, \ell_1 \) are the initial and final labels of \( P \)):

\[
\llbracket P \rrbracket_D(S) : \mathcal{P}(M) \longrightarrow \mathcal{P}(M) \times \mathcal{B} \\
X \longmapsto (\{ m_1 \in M | \exists m_0 \in X, (\ell_0, m_0) \rightarrow^* (\ell_1, m_1) \}, b) \\
\text{where} \quad \left\{ \begin{array}{l} 
    b = \text{true} \quad \text{if } \exists m_0 \in M, (\ell_0, m_0) \rightarrow^* \Omega \\
    b = \text{false} \quad \text{otherwise}
\end{array} \right.
\]
for short, we eliminate the second component of $\llbracket P \rrbracket_D$ (errors)
i.e., we abstract away the errors

We call sound abstract semantics any function $\llbracket P \rrbracket^\#_D$ such that:

$$\forall X \in S, \quad \alpha_M(\llbracket P \rrbracket_D(X)) \sqsubseteq^\# \llbracket P \rrbracket^\#_D(\alpha_M(X))$$

- this definition could be extended so as to take into account the second component of $\llbracket P \rrbracket_D$
- in practice: $\llbracket P \rrbracket^\#_D$ raises alarms whenever its computation reaches a case where $\Omega$ would be reached
Abstract semantics

Computation by induction over the syntax of programs:

\[
\begin{align*}
[x = e;]^\#_D(m^#) & = \text{assign}(x, e, m^#) \\
[\text{if}(c)\{b_t\} \text{else}\{b_f\}]^\#_D(m^#) & = [b_t]^\#_D(\text{test}(c, m^#)) \sqcup_M [b_f]^\#_D(\text{test}(\neg c, m^#)) \\
[\text{while}(c)\{b\}]^\#_D(m^#) & = \text{test}(\neg c, n^#) \\
& \text{where } n^# = \text{lfp}^#_M F^# \\
& \text{and } F^# : z^# \mapsto [b]^\#_D(\text{test}(c, z^#)) \\
[i_0; \ldots; i_{n-1};]^\#_D(m^#) & = [i_{n-1}]^\#_D \circ \ldots \circ [i_0]^\#_D(m^#)
\end{align*}
\]

Then, under the already mentioned soundness conditions for \( \sqcup_M, \text{assign}, \text{test}, \) we have the following soundness theorem:

\[
\forall X \in \mathcal{S}, \ \alpha_M([P]^D(X)) \sqsubseteq^# [P]^\#_D(\alpha_M(X))
\]

More on \( \text{lfp}^# \) on the next slide...
Computation of abstract post-fixpoints

How to compute $\text{lfp}^\#_{m^\#} F^\#$?

- **Standard technique**: compute the sequence

$$
\begin{cases}
    m_0^\# = m^\# \\
    m_n^\# = m_n^\# \sqcup_M F^\#(m_n^\#) \\
    m_{n+1}^\# = m_n^\# \ominus_M F^\#(m_n^\#)
\end{cases}
$$

until it converges (finitely many iterations), and produce $\lim m_n^\#$ but

- it is not very precise
- there are many ways to make this more precise

- **Unrolling iterations**:

  - no widening for the $k$ first iterations

$$
\begin{cases}
    m_0^\# = m^\# \\
    m_n^\# = m_n^\# \sqcup_M F^\#(m_n^\#) & \text{if } n < k \\
    m_n^\# = m_n^\# \ominus_M F^\#(m_n^\#) & \text{if } n \geq k
\end{cases}
$$
Computation of abstract post-fixpoints

- **Alternate join and widening:**
  - do some join iterations in the widening sequence
  - do a join when a new branch is visited...

- **Decreasing iterations**
Outline

1. A small imperative language
2. Reachable states and abstract interpretation
3. Abstract interpretation of the denotational semantics
4. **Static analysis with nested variables scopes**
5. Static analysis in presence of procedures and functions
6. Static analysis in presence of branching instructions
7. Conclusion
A language with local variables

- The language considered so far is very minimalistic
- We will now extend it stage by stage
- First, we add local variables, with scopes

Extended syntax

Same syntax, except that a set of variables is defined at the beginning of each block:

\[
i ::= \ldots \text{ unchanged}
\]

\[
b ::= \{ t x; \ldots t x; i; \ldots i; \} \text{ block}
\]

- initial values of variables: unspecified (could be any)
- block end: the local variables are destroyed
Extension of the operational semantics

- set of variables \( X \) is supposed infinite, countable
- if \( P \) is a program, and \( \ell \) a label in \( P \), the variables at \( \ell \) correspond to the variables of all the blocks \( \ell \) is in (there might be nested scopes)
- thus a memory state should now be a partial map \( m : X \rightarrow V \), i.e., not defined for all variables (actually only finitely many ones)

**transition corresponding to block**

\( \ell_0 : \{ t_0 \ x_0; \ldots \ t_{n-1} \ x_{n-1}; \ell_1 : i_0; \ldots ; \ell_k : i_{k-1}; \ell_{k+1} \}; \ell_{k+2} \)

- **entry:** new local variables are added
  \[
  (\ell_0, m_0) \rightarrow (\ell_1, m_1)
  \]
  where \( m_1 \in \text{addvars}(\{x_0, \ldots, x_{n-1}\}, m_0) \)
  \[
  = \{m_0[x_0 \mapsto v_0, \ldots, x_{n-1} \mapsto v_{n-1}] \mid \forall i, v_i \text{ has type } t_i\}
  \]

- **exit:** local variables are removed
  \[
  (\ell_{k+1}, m_{k+1}) \rightarrow (\ell_{k+2}, m_{k+2})
  \]
  where \( m_{k+2} = \text{remvars}(\{x_0, \ldots, x_{n-1}\}, m_{k+1}) \)
  \[
  = \lambda x \cdot \begin{cases} 
  m_{k+1}(x) & \text{if } x \not\in \{x_0, \ldots, x_{n-1}\} \\
  \text{undefined} & \text{otherwise}
  \end{cases}
  \]
Static analysis with nested variables scopes

Static analysis operations

- The extension of the **denotational semantics** is trivial
- We consider block $b$ defined as $\{t_0 \ x_0; \ldots; t_{n-1} \ x_{n-1}; i_0; \ldots; i_{k-1}; \}$

Then, the abstract semantics is defined by:

$$\llbracket P \rrbracket_D^\#(m^\#) = rem(\{x_0, \ldots, x_{n-1}\},\llbracket i_{k-1} \rrbracket_D \circ \ldots \circ \llbracket i_0 \rrbracket_D(\text{add}(\{t_0 \ x_0, \ldots, t_{n-1} \ x_{n-1}, m^\#))))$$

where $\text{add}$, $\text{rem}$ are sound abstract transfer functions:

**Sound abstract transfer functions**

- $\text{add}$ over-approximates the effect of $\text{addvars}$

$$\forall m \in \gamma_M(m^\#), \ \text{addvars}(\{x_0, \ldots, x_{n-1}\}, m) \subseteq \gamma_M(\text{add}(\{t_0 \ x_0, \ldots, t_{n-1} \ x_{n-1}\}, m^\#))$$

- $\text{rem}$ over-approximates the effect of $\text{remvars}$

$$\forall m \in \gamma_M(m^\#), \ \text{remvars}(\{x_0, \ldots, x_{n-1}\}, m) \in \gamma_M(\text{rem}(\{x_0, \ldots, x_{n-1}\}, m^\#))$$
Other operations on variables

- **Global** and **static** variables:
  should be created at the beginning of the analysis, removed at the end

- **Initialized** variables:
  \( t \ x = e \); is decomposed into \( t \ x; \ x = e \)

- **Volatile** variables:
  a read of a volatile variable is assumed to return **any value**
Outline

1. A small imperative language
2. Reachable states and abstract interpretation
3. Abstract interpretation of the denotational semantics
4. Static analysis with nested variables scopes
5. Static analysis in presence of procedures and functions
6. Static analysis in presence of branching instructions
7. Conclusion
A language with procedures

- We show a minimal extension with parameter-less procedures first, discuss functions later
- A program is made of a set of procedures, with one main procedure

**Extended syntax**

\[
\begin{align*}
P & ::= p \ldots p \\
p & ::= \text{name()}\{b\} \\
i & ::= \ldots \\
| & ::= \text{name();} \\
b & ::= \ldots
\end{align*}
\]

program: set of procedures
procedure body
other instructions unchanged
procedure call
unchanged
Static analysis in presence of procedures and functions

Extension of the operational semantics

Call string

A call string is a possibly empty, finite sequence
\[ \kappa = (\text{name}_0, \ell_0) \cdot \ldots \cdot (\text{name}_n, \ell_n) \]

describing a calling context, where:

- \( \ell_i \) stands for the label right after a call (return point)
- \( \text{name}_i \) stands for the name of a procedure

**States**: a state is a triple \((\kappa, \ell, m)\)

**Initial states**: states of the for \((\kappa_\epsilon, \ell_0, m)\) where:

- \( \kappa_\epsilon \) stands for the empty call-string
- \( \ell_0 \) is the entry label of procedure main

**Procedure call and return**: \( \ell_0 : \text{name}() ; \ell_1 : \ldots \text{name}() \{ \ell_b ; \ldots ; \ell_e \} \)

- **call**: \((\kappa, \ell_0, m) \rightarrow ((\text{name}, \ell_1) \cdot \kappa, \ell_b, m)\)
- **return**: \(((\text{name}, \ell_1) \cdot \kappa, \ell_e, m) \rightarrow (\kappa, \ell_1, m)\)

This defines the **classical abstract interpreter extension**

it is a fully context sensitive analysis
Analysis of a procedure call / a procedure exit

We assume $P$ contains a procedure `name() { b }`

- The extension of the denotational semantics is trivial

$$\llbracket name() \rrbracket_D(X) = \llbracket b \rrbracket_D(X)$$

- This defines the extension of the denotational abstract interpreter

$$\llbracket name() \rrbracket^\#_D(m^\#) = \llbracket b \rrbracket^\#_D(m^\#)$$

it is also a fully context sensitive analysis

- in practice, call string may need be propagated to report context information
- this interpreter is very simple, but does not handle recursion
Procedures with parameters, functions

We considered only a **very limited** set of cases so far

- **Procedures with parameters:**
  - parameters are treated like local variables, with initialization

- **Functions with a return value:**
  - the return value is propagated through a **fictitious variable**

- **Function calls inside sub-expressions**, such as $y = f() + g(x)$
  - the analysis needs to be aware of the execution order
    - if there is an ambiguity, **this issue should be reported**
  - the best solution is to **flatten** the expressions, e.g., as

    $y = f() + g(x) \quad \leadsto \quad \{ 
    \begin{align*}
    \text{int } t_0 &= f(); \\
    \text{int } t_1 &= g(x); \\
    y &= t_0 + t_1;
    \end{align*}
    \}$
Alternative approaches to the analysis of procedures

- **Non-context sensitive or partially context sensitive** analyses:
  - abstraction of call-strings **beyond a given length** $k$
    - which could be 0
  - usually much **less precise**, but **cheaper** than full context sensitive
  - this technique **can deal with recursion** (global fixpoint)

- **Analyses computing procedure summaries**:
  - the body $b$ of procedure $\text{name}()$ is described by a **summary**, which over-approximates $[b]$
  - this technique **can deal with recursion** (global fixpoint)
Static analysis in presence of branching instructions

Outline

1. A small imperative language
2. Reachable states and abstract interpretation
3. Abstract interpretation of the denotational semantics
4. Static analysis with nested variables scopes
5. Static analysis in presence of procedures and functions
6. Static analysis in presence of branching instructions
7. Conclusion
A language with goto

- Real programming languages feature non-structured branching, `break` (in `switch`, loops...), `continue` (in loops), `goto`...
- This is a serious issue for the denotational abstract interpreter, as it follows the structure of the code
- We study only `goto` (the other cases are similar)

Extended syntax

\[
i ::= \ldots \quad \text{other instructions unchanged} \\
| \quad \textcolor{red}{\textbf{goto}} \; \text{lab}; \quad \text{non conditional branching} \\
| \quad \text{lab}; \quad \text{branching label} \\
b ::= \ldots \quad \text{unchanged}
\]

- First, we assume **only forward branchings**
- We consider **backward branchings** afterwards
Concrete denotational semantics

- The extension of the **operational semantics** is **trivial**
  
  For \( l_0 : \text{goto } \text{lab}; l_1; \ldots; \text{lab}_{n-1} : \text{lab}; l_n \), we have \((l_0, m) \rightarrow (l_n, m)\)

- For the **denotational semantics** the extension is more involved
  
  The usual trick is to use **continuations**:

  
  \[
  \llbracket P \rrbracket_D : \mathcal{P}(S) \times \mathcal{P}(S) \longrightarrow \mathcal{P}(S) \times \mathcal{P}(S)
  \]

  In configuration \((X_0, X_1)\),
  
  - \(X_0\) denotes states waiting for the current continuation
  - \(X_1\) denotes states waiting for the continuation at \text{lab}
    
    i.e., the **branching executions**

  Then:

  \[
  \begin{cases}
    \llbracket \text{goto } \text{lab} \rrbracket_D(X_0, X_1) = (\emptyset, X_0 \cup X_1) \\
    \llbracket \text{lab :} \rrbracket_D(X_0, X_1) = (X_0 \cup X_1, \emptyset)
  \end{cases}
  \]

- This generalizes to programs with **several branching labels**
  
  i.e., the second component is a function from labels into \(\mathcal{P}(S)\)
Abstraction

- We simply follow the structure of the denotational semantics:
  - an abstract state should be a **pair** \((m_0^\#, m_1^\#)\)
  - the abstract interpretation of most statements is unchanged
    \[
    [[x = e;]]^D_D(m_0^\#, m_1^\#) = (\text{assign}(x, e, m_0^\#), m_1^\#)
    \]
  - the branching statements:
    \[
    [[\text{goto} \ lab]]^D_D(m_0^\#, m_1^\#) = \bot, m_0^\# \sqcup M m_1^\#
    \]
    \[
    [[\text{lab}:]]^D_D(m_0^\#, m_1^\#) = (m_0^\# \sqcup M m_1^\#, \bot)
    \]

- **Case of backward gotos:**
  - the denotational semantics of a block is a **least-fixpoint**
  - the abstract semantics also needs a least-fixpoint
Outline

1. A small imperative language
2. Reachable states and abstract interpretation
3. Abstract interpretation of the denotational semantics
4. Static analysis with nested variables scopes
5. Static analysis in presence of procedures and functions
6. Static analysis in presence of branching instructions
7. Conclusion
Two abstract interpreters, one abstract domain interface

- The first abstract interpreter follows the structure of the control flow graph
- The second one follows the structure of the AST (Astrée implements this)
- Both require an abstract domain be provided, to represent sets of memory states

- No assumption is made on the inner structure of $\mathbb{D}_M^\#$
- We only made assumptions on the operators it should provide and their soundness conditions
Abstract domain interface

- **least element** $\bot$
- **greatest element** $\top$: $\gamma_M(\top) = S$
- **sound inclusion check** $\sqsubseteq^M$: $m_0^\# \sqsubseteq^M m_1^\# \implies \gamma_M(m_0^\#) \subseteq \gamma_M(m_1^\#)$
- **sound join operator** $\sqcup_M$: $\gamma_M(m_0^\#) \sqcup \gamma_M(m_1^\#) \subseteq \gamma(m_0^\# \cup_M m_1^\#)$
- **widening operator** $\nabla_M$
- **sound assignment operator** $\text{assign}$
- **sound test operator** $\text{test}$
- operator to **add variables** $\text{add}$
- operator to **remove variables** $\text{rem}$ (projection)

(all soundness conditions given in the previous slides)

Design of these operators: **lectures on abstract domains**