THEORY OF COMPILATION

LECTURE 12

STATIC ANALYSIS
You are here

Source text

Compiler

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Optimization

Code Generation

Executable code
Static Program Analysis

• Can automatically **prove** interesting properties
  ‣ absence of null pointer dereferences, numerical assertions, termination, absence of data races, information flow, ...

• Nice combination of **math** and **system** building
  ‣ combines program semantics, data structures, discrete math, logic, parallelism, decision procedures, ...
Static Program Analysis

- No need to run the program!
  - No concrete input needed!
Plan for Today

- We will learn a generic analysis technique called **Abstract Interpretation**
  - and understand the **guarantees** it provides

- We will apply it to two domains of interest:
  - Numerical
  - Pointers
Abstract Interpretation

- **Interpretation**: run the program on a concrete input and produce concrete output.

- **Abstract Interpretation**: run the program on an abstract input value. The output is an abstraction of the set of reachable states.
Concrete Semantics

• WHILE language
  ‣ Syntax:
    \[ S \rightarrow x := E \mid S ; S \mid \text{skip} \]
    \[ \mid \text{if } E \text{ then } S \text{ else } S \]
    \[ \mid \text{while } E \text{ do } S \]
    \[ E \rightarrow x \mid E \Diamond E \]
    \[ \Diamond \in \{+, -, *, /, =, \neq, <, >, \leq, \geq\} \]
Denotational Semantics

• $\sigma : \text{Var} \rightarrow \mathbb{Z}$
  ▶ A state of the program
    (also called a store)

• $\Sigma$ — the set of all such states

• $\llbracket s \rrbracket : \Sigma \rightarrow \Sigma$
  ▶ $\llbracket s \rrbracket \sigma$ is the state resulting from $\sigma$ after executing the statement $s$
Denotational Semantics

- $\llbracket x := e \rrbracket \sigma = \sigma[x \mapsto \llbracket e \rrbracket \sigma]$
- $\llbracket s_1 ; s_2 \rrbracket \sigma = \llbracket s_2 \rrbracket (\llbracket s_1 \rrbracket \sigma)$
- $\llbracket \text{if } e \text{ then } s_1 \text{ else } s_2 \rrbracket \sigma =$
- $\llbracket \text{while } e \text{ do } s \rrbracket \sigma =$

$S \rightarrow x := E \mid S ; S \mid \text{skip}$
$\mid \text{if } E \text{ then } S \text{ else } S$
$\mid \text{while } E \text{ do } S$

$E \rightarrow x \mid E \Diamond E$
Denotational Semantics

- $[x := e] \sigma = \sigma[x \mapsto [e] \sigma]$
- $[s_1 ; s_2] \sigma = [s_2]([s_1] \sigma)$
- $[\text{if } e \text{ then } s_1 \text{ else } s_2] \sigma = \begin{cases} [s_1] \sigma & \text{if } [e] \sigma = \text{true} \\ [s_2] \sigma & \text{if } [e] \sigma = \text{false} \end{cases}$
- $[\text{while } e \text{ do } s] \sigma =$
Denotational Semantics

- $\llbracket x := e \rrbracket_\sigma = \sigma[x \mapsto \llbracket e \rrbracket_\sigma]$
- $\llbracket s_1 ; s_2 \rrbracket_\sigma = \llbracket s_2 \rrbracket(\llbracket s_1 \rrbracket_\sigma)$
- $\llbracket \text{if } e \text{ then } s_1 \text{ else } s_2 \rrbracket_\sigma = \begin{cases} \llbracket s_1 \rrbracket_\sigma & \text{if } \llbracket e \rrbracket_\sigma = \text{true} \\ \llbracket s_2 \rrbracket_\sigma & \text{if } \llbracket e \rrbracket_\sigma = \text{false} \end{cases}$
- $\llbracket \text{while } e \text{ do } s \rrbracket_\sigma = \begin{cases} \sigma & \text{if } \llbracket e \rrbracket_\sigma = \text{false} \\ \llbracket \text{while } e \text{ do } s \rrbracket_\sigma & \text{if } \llbracket e \rrbracket_\sigma = \text{true} \end{cases}$
Denotational Semantics

- \([x := e] \sigma = \sigma[x \mapsto [e] \sigma]\)

- \([s_1 ; s_2] \sigma = [s_2]([s_1] \sigma)\)

- \([\text{if } e \text{ then } s_1 \text{ else } s_2] \sigma = \begin{cases} [s_1] \sigma & \text{if } [e] \sigma = \text{true} \\ [s_2] \sigma & \text{if } [e] \sigma = \text{false} \end{cases}\)

- \([\text{while } e \text{ do } s] \sigma = \begin{cases} \sigma & \text{if } [e] \sigma = \text{false} \\ ?? & \text{if } [e] \sigma = \text{true} \end{cases}\)
Galois Connection

Concrete State Space

Abstract State Space

sets of stores \( \rightarrow \) descriptions of sets of stores

\( \alpha \)

\( \gamma \)
Galois Connection

Concrete State Space

Abstract State Space

\[ \alpha \]

sets of stores \( \rightarrow \)
descriptions of sets of stores \( \leftarrow \gamma \)
Galois Connection

Concrete State Space

Abstract State Space

sets of stores

descriptions of sets of stores

α

γ

α
Galois Connection

Concrete State Space

Abstract State Space

sets of stores

descriptions of sets of stores
Galois Connection

Concrete State Space

Abstract State Space

sets of stores

γ

α

descriptions of sets of stores

α

γ

α

γ
Galois Connection

• Lattices $C$ and $A$

• Functions $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$

\[ \forall c \in C, \ a \in A. \quad \alpha(c) \sqsubseteq a \iff c \sqsubseteq \gamma(a) \]

• Equivalently,

\[ \alpha(\gamma(a)) \sqsubseteq a \land c \sqsubseteq \gamma(\alpha(c)) \]
Galois Connection

Concrete State Space

Abstract State Space

\[ \alpha(\gamma(a)) \subseteq a \land c \subseteq \gamma(\alpha(c)) \]
Abstract Domain — Example

Concrete State Space

Abstract State Space

\[ C = \mathcal{P}(\mathbb{Z}) \]
Abstraction Function — Example

• \( \alpha : \mathcal{P}(\mathbb{Z}) \rightarrow \{\bot, 0, +, –, \top\} \)

• It is useful to define an auxiliary function \( \beta \)
  
  \[ \beta : \mathbb{Z} \rightarrow \{\bot, 0, +, –, \top\} \]

  \[ \alpha(S) = \bigsqcup \{ \beta(\sigma) \mid \sigma \in S \} \]
Abstract Interpretation

- Back to our program analysis:
  - A will be our domain lattice, also called abstract domain.
  - Every operation in our concrete semantics will have corresponding abstract semantics.

\[
\sigma^# \in A \quad \text{abstract state}
\]

\[
[\sigma]^# : A \rightarrow A \quad \text{abstract transformer}
\]
Abstract Semantics — Example

- WHILE program with $k$ variables $v_1, \ldots, v_k$
  - $\alpha : \mathcal{P}(\mathbb{Z}^k) \rightarrow \{\bot, 0, +, -, \top\}^k$

- $\llbracket x := e \rrbracket_{\sigma} = \sigma[x \mapsto \llbracket e \rrbracket_{\sigma}]$

- $\llbracket s_1 ; s_2 \rrbracket_{\sigma} = \llbracket s_2 \rrbracket_{\sigma}(\llbracket s_1 \rrbracket_{\sigma})$
Abstract Semantics

\[ \sigma \xrightarrow{[s]^\#} \sigma' \]

\[ \alpha \]

\[ \sigma \xrightarrow{[s]} \sigma' \]
Abstract Semantics

\[ \sigma \rightarrow [s] \rightarrow \sigma' \]

\[ \alpha \rightarrow [s] \rightarrow \alpha \]

\[ \sigma \rightarrow \sigma' \]
Abstract Semantics

\[ \sigma \xrightarrow{\sigma'} \]

\[ \alpha \]

\[ \alpha([s](\sigma)) \subseteq [s]^#(\alpha(\sigma)) \]
Abstract Semantics — Example

1: \( x := 5; \)
2: \( y := 7; \)
3: while (i \geq 0) do
   (4: \( y := y + 1; \)
    5: \( i := i - 1 \)
   )
4: \( x := 5 \)
5: \( y := 7 \)
6: \( i \geq 0 \)
Abstract Semantics — Example

1: \( x := 5; \)
2: \( y := 7; \)
3: while \((i \geq 0)\) do
   (4: \( y := y + 1; \)
    5: \( i := i - 1 \)
   )
6: 

\[
\begin{array}{ccc}
x & y & i \\
Z & Z & Z \\
\end{array}
\]

---

\[
\begin{array}{ccc}
x & y & i \\
T & T & T \\
\end{array}
\]

---

\[
\begin{array}{ccc}
x & y & i \\
5 & Z & Z \\
\end{array}
\]

---

\[
\begin{array}{ccc}
x & y & i \\
5 & 7 & Z \\
\end{array}
\]

---

\[
\begin{array}{ccc}
x & y & i \\
5 & 7 & N \\
\end{array}
\]
Abstract Semantics — Example

1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
   (4: y := y + 1;
    5: i := i - 1)
6:
Abstract Semantics — Example

1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
   (x := 5
    y := y + 1;
    i := i - 1
   )
4:   y := y + 1;
5:   i := i - 1
6:   x := 5;
Abstract Semantics — Example

1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
   (4: y := y + 1;
    5: i := i − 1)
6:
Important Point

Note that abstract transformers are defined per **programming language** and **abstract domain**, once and for all, and **not** per program!

Abstract transformers define the new formal abstract semantics of the language.

This means that **any program** in that programming language can be analyzed using the **same transformers**
1: x := 5;
2: y := -1;
3: while (i ≥ 0) do
   (4: y := y + 1;
    5: i := i - 1
   )
6:
Transformer Soundness

1: $x := 5$;
2: $y := -1$;
3: while ($i \geq 0$) do
   (4: $y := y + 1$;
    5: $i := i - 1$
   )
6:
**Transformer Soundness**

1: \( x := 5; \)
2: \( y := -1; \)
3: **while** \( i \geq 0 \) **do**
   4: \( y := y + 1; \)
   5: \( i := i - 1 \)
4: \[ x \quad y \quad i \]
   \[ 5 \quad -1 \quad \mathbb{Z} \]
5: **while** \( i \geq 0 \) **do**
   4: \( y := y + 1; \)
   5: \( i := i - 1 \)
6: \[ x \quad y \quad i \]
   \[ 5 \quad 0 \quad \mathbb{N} \]
Transformer Soundness

1: x := 5;
2: y := -1;
3: while (i ≥ 0) do
   (y := y + 1;
    i := i - 1)
4: y := y + 1
5: i := i - 1
6:
1: \( x := 5; \)
2: \( y := -1; \)
3: \( \textbf{while} (i \geq 0) \textbf{do} \)
   \( (\) 
4: \( y := y + 1; \)
5: \( i := i - 1 \)
   \( ) \)
6: \( x \ y \ i \\
      5 \ -1 \ \mathbb{Z} \\
      + \ -\ + \ 
\)
A sound abstract transformer should always — for every state — produce results that are a superset of what a concrete transformer would produce.
A sound abstract transformer should always — for every state — produce results that are a superset of what a concrete transformer would produce.
How about this?

1: \( x := 5; \)
2: \( y := -1; \)
3: \( \text{while } (i \geq 0) \text{ do} \)
   (\(\)
4: \( y := y + 1; \)
5: \( i := i - 1 \)
  )
6: 

\[
\left[y := y + 1\right] \sigma = \sigma [y \mapsto \top]
\]
How about this?

This transformer is **sound**, but it’s not **precise**.

\[
\llbracket y := y + 1 \rrbracket^\# \sigma^\# = \sigma^\#[y \mapsto \top]
\]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>N</td>
</tr>
</tbody>
</table>

\[
\llbracket\rrbracket
\]

\[
\downarrow
\]

\[
y := y + 1
\]

\[
\llbracket \rrbracket^#
\]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>N</td>
</tr>
</tbody>
</table>

\[
\llbracket\rrbracket
\]

\[
\downarrow
\]

\[
y := y + 1
\]

\[
\llbracket \rrbracket^#
\]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
</tbody>
</table>
Transformer Precision

This abstract state:

represents infinitely many concrete states where y is always 0, including:

If we perform $y := y + 1$ on any of these concrete states, we will always get states where $y$ is always positive, such as:

It would therefore be sound to represent them using this abstract state:
Transformer Precision

This abstract state:

represents infinitely many concrete states where \( y \) is always 0, including:

If we perform \( y := y + 1 \) on any of these concrete states, we will always get states where \( y \) is always positive, such as:

It would therefore be sound to represent them using this abstract state:

However, the abstract transformer produces an abstract state where \( y \) can be any value:
Best Abstract Transformer

• It is easy to be sound and imprecise: always produce $\top$

• A good transformer is both sound and precise. If we lose precision, it needs to be clear why and where:
  ▸ sometimes, computing the most precise transformer (also called the best transformer) is impossible
  ▸ for efficiency reasons, we may compromise for a transformer that is “good enough”
Let’s prove a property!

1: \( x := 5; \)
2: \( y := 7; \)

3: while (\( i \geq 0 \)) do 
   ( 
   4: \( y := y + 1; \)
   5: \( i := i - 1 \)
   6: )

7: assert \( 0 \leq x + y \)
Let’s prove a property!

1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
   (  
   4:   y := y + 1;
   5:   i := i - 1
   6:  )
7: assert 0 ≤ x + y
Let’s prove a property!

1: \( x := 5 \);
2: \( y := 7 \);
3: \( \text{while } (i \geq 0) \text{ do } \)
   (4: \( y := y + 1 \);
   5: \( i := i - 1 \)
   6: )
7: \( \text{assert } 0 \leq x + y \)

\[ [x + y]^{\sigma} = + \]
Coming Up

Recap & More
THEORY OF COMPILATION

LECTURE 12

Complicated
ADVANCED
STATIC ANALYSIS
Exam

★ 20% Compiler Phases
★ 40% Syntax, Semantics, Code generation
★ 10% Optimizations
★ 30% Static Analysis
Reminder:

Galois Connection

\[ \alpha(\gamma(a)) \subseteq a \land c \subseteq \gamma(\alpha(c)) \]
Abstract Domain

Concrete State Space

Abstract State Space

$C = \mathcal{P}(\mathbb{Z})$

$\subseteq$

$\alpha$

$\gamma$

Reminder:
Reminder:

Abstract Semantics

\[ \alpha(\llbracket s \rrbracket(\sigma)) \subseteq \llbracket s \rrbracket^\#(\alpha(\sigma)) \]
1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
   (4: y := y + 1;
    5: i := i - 1
   )
7: assert 0 ≤ x + y

Proving Properties

Reminder:

[[x + y]]^σ = +
Numerical Domains
Intervals Domain

constants

$c$ 

$\mathbb{Z} \cup \{T\}$

signs

$+, -, 0$
Intervals Domain

constants

\( c \in \mathbb{Z} \cup \{ \top \} \)

signs

\(+, -, 0\)

intervals

\([a, b]\) with

\((\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{\infty\})\)
Instead of abstracting variable values using the sign of the value, we will abstract them using an interval.
Intervals Domain

...
1: \( x := 5; \)
2: \( y := 7; \)
3: \( \textbf{while } (i \geq 0) \textbf{ do } \)
   (\quad
4: \quad y := y + 1;
5: \quad i := i - 1
6: \quad)
7: \textbf{assert } 0 \leq y - x
1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
   (4: y := y + 1;
5: i := i − 1
6: )
7: assert 0 ≤ y − x
1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
   (4: y := y + 1;
    5: i := i - 1
   6: )
7: assert 0 ≤ y - x
1: x := 5;  
2: y := 7;  
3: while (i ≥ 0) do  
(  
4:   y := y + 1;  
5:   i := i - 1  
6: )  
7: assert 0 ≤ y - x
Let the iterations begin!

1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
   (4: y := y + 1;
    5: i := i - 1
   )
7: assert 0 ≤ y - x
Cannot Reach a Fixed Point

• With the interval abstraction we could not reach a fixed point.
  ‣ The domain has infinite height.

• What should we do?
  ‣ Introduce a special operator that would replace the “join” operation in our abstract semantics
  ‣ It is a hack to ensure termination, at the expense of precision
Widening

- $\triangledown : A \times A \rightarrow A$
  - $x \sqcup y \subseteq x \triangledown y$
Widening

- $\nabla: A \times A \rightarrow A$
  - $x \sqcup y \subseteq x \nabla y$

**Example — for intervals**

- $x \nabla \bot = \bot \nabla x = x$
- $[a_1, b_1] \nabla [a_2, b_2] = [c, d]$
  
  $$c = \begin{cases} 
  a_1 & \text{if } a_1 \leq a_2 \\
  -\infty & \text{if } a_1 > a_2 
  \end{cases}$$
  $$d = \begin{cases} 
  b_1 & \text{if } b_2 \leq b_1 \\
  \infty & \text{if } b_2 > b_1 
  \end{cases}$$
1: \(x := 5;\)
2: \(y := 7;\)
3: \textbf{while} \(i \geq 0\) \textbf{do} \quad \begin{align*}
4: & \quad y := y + 1; \\
5: & \quad i := i - 1 \\
6: & \end{align*}
7: \textbf{assert} \ 0 \leq y - x

\begin{tabular}{c|c|c|c|c|}
\hline
\textbf{x} & \textbf{y} & \textbf{i} & \\
\hline
[5,5] & [7,7] & \top \\
\hline
\end{tabular}
1: \( x := 5; \)
2: \( y := 7; \)
3: while \( i \geq 0 \) do
   ( 4: \( y := y + 1; \)
   5: \( i := i - 1 \)
   6: )
7: assert \( 0 \leq y - x \)
1: \( x := 5; \)
2: \( y := 7; \)
3: \textbf{while} \( (i \geq 0) \) \textbf{do} \\
   ( \\
4: \quad y := y + 1; \\
5: \quad i := i - 1 \\
6:   ) \)
7: \textbf{assert} \( 0 \leq y - x \)
Abstraction is not an elephant

- Constant, sign, and interval domain cannot track relationships between variable values.

```plaintext
if ( ... ) (  
    x := 1 ; y := -1
)  
else (  
    x := -1 ; y := 1
)

assert 0 ≤ x + y
```
Abstraction is not an elephant

- Constant, sign, and interval domain cannot track relationships between variable values.

```plaintext
if ( ... ) (  
    x := 1 ; y := -1  
)  
else (  
    x := -1 ; y := 1  
)
assert 0 ≤ x + y

if ( ... )  
    x := 1  
else  
    x := -1;
y := -x;
assert 0 ≤ x + y
```
Variable Relations

• A very useful property, in particular for bounds checking

```plaintext
foo(n) {
    a := new Z[n];
    i := 0;
    j := n - 1;
    while (i < j) do (  
        if (a[i] = 0)  
            a[j] := 1;
        i := i + 1;
        j := j - 1;
    )
}

bar(n) {
    a := new Z[n];
    i := 0;
    j := n - 1;
    while (i < j) do (  
        if (a[i] = 0)  
            then i := i + 1;
        j := j - 1;
        a[i + j] := 2;
    )
}
```
A New Domain

Constants

Signs

Intervals
A New Domain

Constants

Signs

Intervals

Octagons
Octagons Domain

- **Octagon** = a set of inequalities of the form

  \[ \pm u_i \leq c \quad \pm u_i \pm u_j \leq c \quad (i \neq j) \]

- **Semantics:**
  intersection of half-planes

  - \( x \leq 1 \)
  - \( x \geq -2 \)
  - \( y \leq 2 \)
  - \( y \geq -1 \)
  - \( -x + y \leq 3 \)
  - \( x + y \leq 2 \)
  - \( -x - y \leq 2 \)
  - \( x - y \leq 1 \)
  - \( y = 2 \)
  - \( y = 2 \)
  - \( x = 1 \)
  - \( x = 1 \)
  - \( y = -1 \)
  - \( y = -1 \)
  - \( x - y = 1 \)
  - \( x - y = 1 \)
  - \( x + y = 3 \)
  - \( x + y = 3 \)
  - \( x = 2 \)
  - \( x = 2 \)
  - \( y = 2 \)
  - \( y = 2 \)
Octagons Domain

- **Octagon** = a set of inequalities of the form

\[ \pm \nu_i \leq C \quad \pm \nu_i \pm \nu_j \leq C \quad (i \neq j) \]

- Semantics: intersection of half-planes

  - \( x \leq 1 \)
  - \( x \geq -2 \)
  - \( y \leq 2 \)
  - \( y \geq -1 \)
  - \( -x + y \leq 3 \)
  - \( x + y \leq 2 \)
  - \( -x - y \leq 2 \)
  - \( x - y \leq 1 \)
Octagons Domain

• There is more than one way to represent a single octagonal region.
  ‣ E.g.

\[
\gamma \left( \left\{ \begin{array}{l} x \leq 5 \\ y \leq 7 \end{array} \right\} \right) = \gamma \left( \left\{ \begin{array}{l} x \leq 5 \\ y \leq 7 \\ x + y \leq 12 \end{array} \right\} \right)
\]
Closure for Octagons

• S is a **closed octagon** iff:

  ‣ For any $i, j, c$, such that $S \Rightarrow v_i + v_j \leq c$, there exists $c' \leq c$ such that $v_i + v_j \leq c' \in S$

  ‣ similarly for $-v_i + v_j \leq c$, $v_i - v_j \leq c$, etc. 

  *and for* $v_i \leq c$

• Canonical representation:

  ‣ Every *(signed)* variable and every pair of *(signed)* variables has exactly one bound (may be $\infty$)
Order Relation on Octagons

- $S_1 \subseteq S_2$ iff whenever $\pm v_i \pm v_j \leq c \in S_2$, there is a $c' \leq c$ such that $\pm v_i \pm v_j \leq c' \in S_1$

(with same signs of course)
Join for Octagons

- $S_1 \sqcup S_2$ can be computed by taking piecewise maximum of bounds of corresponding inequalities

$$(x \leq 5) \land (x + y \leq 10) \sqcup (x \leq 4) \land (x + y \leq 11)$$

$\Downarrow$

$$(x \leq 5) \land (x + y \leq 11)$$
Abstract Transformers

• *It’s complicated*...

• A few basic ones:
  ▸ \( x := c \)
  ▸ \( x < c \)
  ▸ \( x := x + c \)
  ▸ \( x := y + c \)
Abstract Transformers

• *It’s complicated*...

• A few basic ones:
  - $x := c$
  - $x < c$
  - $x := x + c$
  - $x := y + c$

• General assignments — $x := e$
  - Approximate by interval arithmetic on $e$
Polyhedra Domain

constraints are of the following form:

\[ c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \leq c \]

the slope can vary

an abstract state is again a conjunction of constraints:

\[ x - y \geq -20 \land \]
\[ x - 3y \leq 2 \land \]
\[ x + y \geq 5 \]

Order, join, transformers: require solving linear equations.
Polyhedra: Example

- McCarthy’s “91 function”

\[
M(n) = \begin{cases} 
    n - 10 & \text{if } n > 100 \\
    M(M(n + 11)) & \text{if } n \leq 100 
\end{cases}
\]

```python
def m(n):
    c = 1
    while c != 0:
        if n > 100: n -= 10; c -= 1
        else: n += 11; c += 1
    return n
```

```python
def m(n):
    x = y = 0; n_0 = n
    while x <= y:
        if n > 100: n -= 10; x += 1
        else: n += 11; y += 1
    return n
```

Replacing \(c\) by \(1 + y - x\)
Numerical Domains: Summary

**Cartesian**
- Constants
- Signs
- Intervals

**Relational**
- Octagons
- Polyhedra
Pointer Analysis
Simple Example

program

\[
\begin{align*}
x & := 5 && \text{S1} \\
\text{ptr} & := & \&x && \text{S2} \\
*\text{ptr} & := 9 && \text{S3} \\
y & := x && \text{S4}
\end{align*}
\]

data flow

- What are the data dependencies in this program?
- **Problem**: just looking at variable names will not give you the correct information
  - After statement S2, program names “x” and “*ptr” are both expressions that refer to the same memory location.
  - We say that \texttt{ptr points-to} \texttt{x} after statement S2.
- In a C-like language that has pointers, we must know the \textit{points-to relation} to be able to determine dependencies correctly
Program Model

- Extend WHILE with statements that deal with pointers:
  - **address**: \( x := \&y \)
  - **copy**: \( x := y \) (regular assignment)
  - **load**: \( x := *y \)
  - **store**: \( *x := y \)

- For now: no heap, no function calls. Allowed types are \( \mathbb{Z}, \mathbb{Z}^* \) (pointer to number), \( \mathbb{Z}^{**} \), ...
Points-to Relation

• Directed graph:
  ‣ Nodes are program variables (+ special node for null)
  ‣ Edge (a,b) — variable a points-to variable b

• Of course, points-to is different at different program locations
Points-to Relation

• Directed graph:
  ▸ Nodes are program variables (+ special node for null)
  ▸ Edge (a,b) — variable a points-to variable b

• Out-degree may be > 1
  if there are multiple paths

What does x point to here?
Points-to Relation

- As an abstract domain (a lattice):
  - Nodes are fixed per program; can think of it as a power-set domain of possible edges.
  - \( \bot \) is a graph with no edges.
  - \( \sqsubseteq \) is the subgraph relation (edge subset)
  - \( \sqcup \) is obtain by union of edges
Points-to Analysis: Two Flavors

• Flow Sensitive
  ‣ Based on abstract interpretation / dataflow
  ‣ Can examine behavior at different locations

• Flow Insensitive
  ‣ Computes a single points-to relation for the entire program
  ‣ Works by generating constraints and solving them
  ‣ (Andersen’s algorithm / Steengards algorithm)
Points-to: Abstract Semantics

\[ [s] \# G = G' \]
Points-to: Abstract Semantics

\[ [s] \# G = G' \]

\[
\begin{align*}
x &: \& y \\
G' &= G \text{ with } pt'(x) \leftarrow \{y\}
\end{align*}
\]

\[
\begin{align*}
x &: y \\
G' &= G \text{ with } pt'(x) \leftarrow pt(y)
\end{align*}
\]

\[
\begin{align*}
x &: *y \\
G
\end{align*}
\]

\[
\begin{align*}
*x &: y \\
G
\end{align*}
\]
Points-to: Abstract Semantics

\[ [s] \# G = G' \]

- **x := &y**
  - \( G' = G \) with \( \text{pt}'(x) \leftarrow \{y\} \)

- **x := y**
  - \( G' = G \) with \( \text{pt}'(x) \leftarrow \text{pt}(y) \)

- **x := *y**
  - \( G' = G \) with \( \text{pt}'(x) \leftarrow \bigcup \{ \text{pt}(a) \mid a \in \text{pt}(y) \} \)

- ***x := y**
  - \( G' = G \) with \( \text{pt}'(x) \leftarrow \bigcup \{ \text{pt}(a) \mid a \in \text{pt}(y) \} \)
Points-to: Abstract Semantics

\[ [s] \# G = G' \]

\[ x := \& y \]
\[ G' = G \text{ with } pt'(x) \leftarrow \{ y \} \]

\[ x := y \]
\[ G' = G \text{ with } pt'(x) \leftarrow pt(y) \]

\[ x := * y \]
\[ G' = G \text{ with } pt'(x) \leftarrow \bigcup \{ pt(a) \mid a \in pt(y) \} \]

\[ *x := y \]
\[ G' = G \text{ with } pt'(a) \leftarrow pt(y) \text{ for all } a \in pt(x) \]
Points-to: Abstract Semantics

\[ [s] # G = G' \]

- **x := &y**
  \[ G' = G \text{ with } pt'(x) \leftarrow \{y\} \]

- **x := y**
  \[ G' = G \text{ with } pt'(x) \leftarrow pt(y) \]

- **x := *y**
  \[ G' = G \text{ with } pt'(x) \leftarrow U\{pt(a) \mid a \in pt(y)\} \]

- ***x := y**
  \[ G' = G \text{ with } \forall a \in pt(x), pt'(a) \leftarrow pt(y) \]

**strong updates**

**weak update (why?)**
Dynamic Allocation

• What to do with \( x := \text{new Z[...]} \) ?
  ‣ Program can create an unbounded number of objects
  ‣ Need some static naming scheme for dynamically allocated objects

● AbsObj

● Single name for the entire heap
  \( \text{AbsObj} = \{ \text{H} \} \)

● Type based static names
  \( \text{AbsObj} = \{ T \mid T \text{ is a type in the program} \} \)

● Name based on static allocation site
  \( \text{AbsObj} = \{ \mu \mid \text{stmt}(\mu) \text{ is } p := \text{newArray}^{\mu} a \} \)
Dynamic Allocation

• AbsObj — set of abstract object names
  ▸ Single name for the entire heap
    AbsObj = \{ H \}
  ▸ Type-based static names
    AbsObj = \{ T \mid T \text{ is a type in the program} \}
  ▸ Name based on static allocation site
    AbsObj = \{ \mu \mid \text{statement } \mu: p := \text{new } Z[a] \}
Dynamic Allocation: Semantics

• ** Basically:** model every “new” as “address of”

```
1: p := new Z[5];
2: q := new Z[5];
3: if (p = q) then
4:    z := p
5: else
6:    z := q
```

```
1: p := &A1;
2: q := &A2;
3: if (p = q) then
4:    z := p
5: else
6:    z := q
```
Dynamic Allocation: Semantics

- **Basically**: model every “new” as “address of”

1: p := new Z[5];
2: q := new Z[5];
3: if (p = q) then
4:    z := p
5: else
6:    z := q

- **Conservative**: may result in spurious “may point to” entries; but “must not point to” results are always sound.

1: p := &A1;
2: q := &A2;
3: if (p = q) then
4:    z := p
5: else
6:    z := q
Points-to Analysis: Example

1: w1 := &a1;
2: w2 := &a2;
3: q := new Z[5];
4: r := new Z[5];
5: *w1 := r;
6: if (...) then
7: p := w1
8: else
9: p := w2;
10: *p := q

a1, a2, q, r : Z*
w1, w2, p : Z**
Aliasing Analysis

Derived from result of point-to analysis

1: \( p := \text{new } Z[5]; \)
2: \( q := \text{new } Z[5]; \)
3: if \( p = q \) then
4: \( z := p \)
5: else
6: \( z := q \)

\( z \) and \( p \) may not alias
\( q \) and \( p \) may not alias
\( z \) and \( q \) may alias
Example: Pointers + Sign

1: a := 0;
2: b := 0;
3: q := &a;
4: *q := *q + 1
5: assert a + b > 0

Abstract Domain:

Points-to × Signs

⟨ q → a , a b + − ⟩
Example: Aliasing + Available Expressions

1: a := *q + 4 * c;
2: b := 0;
3: *p := 1;
4: b := *q + 4 * c
Example: Aliasing + Available Expressions

1: a := *q + 4 * c;
2: b := 0;
3: *p := 1;
4: b := *q + 4 * c

Optimization is valid:
p and q are not aliased
Recap

• Lexical analysis
  – regular expressions identify tokens (“words”)

• Syntax analysis
  – context-free grammars identify the structure of the program (“sentences”)

• Contextual (semantic) analysis
  – type checking defined via typing judgements
  – can be encoded via attribute grammars

• Syntax directed translation
  – attribute grammars

• Intermediate representation
  – many possible IRs
  – generation of intermediate representation
Journey inside a compiler

```
float initial, rate;
position = initial +
rate * 60
```

```
<FLOAT> <ID,"initial"> <COMMA> <ID,"rate"> <SEMI>
<ID,"position"> <=> <ID,"initial"> <+> <ID,"rate"> <*> <60>
```
Journey inside a compiler

\[(\text{ID,position}) \leftrightarrow (\text{ID,initial}) \leftrightarrow (\text{ID,rate}) \leftrightarrow 60\]

\[
\begin{align*}
\text{AST} &= \text{<ID,position>} \\
& \quad + \text{<ID,initial>} \\
& \quad \times \text{<ID,rate>} \\
& \quad 60
\end{align*}
\]
Journey inside a compiler

Symbol Table

<table>
<thead>
<tr>
<th>symbol</th>
<th>type</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>position</td>
<td>float</td>
</tr>
<tr>
<td>2</td>
<td>initial</td>
<td>float</td>
</tr>
<tr>
<td>3</td>
<td>rate</td>
<td>float</td>
</tr>
</tbody>
</table>

Annotated AST

```
<id,1> = <id,2> + <id,3> * inttofloat 60
```
Journey inside a compiler

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Intermediate Representation

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Journey inside a compiler

Intermediate Representation

\[
\begin{align*}
t1 &= \text{inttofloat}(60) \\
t2 &= \text{id3} \times t1 \\
t3 &= \text{id2} + t2 \\
id1 &= t3
\end{align*}
\]

Optimized

\[
\begin{align*}
t1 &= \text{id3} \times 60.0 \\
id1 &= \text{id2} + t1
\end{align*}
\]
Journey inside a compiler

Optimized

t1 = id3 * 60.0
id1 = id2 + t1

Assembly

LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
You Have Reached
Your Destination