Lecture 04 – Numerical Abstractions

PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav
Previously...

- Trace semantics
- Collecting semantics
- Lattices
- Galois connection
- Least fixed point
Galois Connection

\[ \alpha(C) \subseteq A \iff C \subseteq \gamma(A) \]
How do we figure out the abstract effect $[S]^#$ of a statement $S$?
Sound Abstract Transformer

\[ \alpha \circ [S] \circ \gamma (A) \equiv [S]^\#(A) \]
Soundness of Induced Analysis

\[ \alpha(lfp(G)) \sqsubseteq lfp(\alpha \circ G \circ \gamma) \sqsubseteq lfp(F) \]
Trivial Example

\[ x = 42; \]
\[ \text{while}(?) x++; \]

is \( x = 17 \) possible in \( \text{lfp}(G) \)? is it in \( \gamma(\text{lfp}(F)) \)?
Today

- A few more words about Lattices, Galois Connections, and friends
- Numerical Abstractions
- parity
- signs
- constant
- interval
- octagon
- polyhedra
Complete Lattices

- A complete lattice \((L, \sqsubseteq)\) is a poset such that all subsets have least upper bounds as well as greatest lower bounds.

- In particular:
  - \(\bot = \bigvee \emptyset = \bigwedge L\) is the least element (bottom).
  - \(\top = \bigvee L = \bigwedge \emptyset\) is the greatest element (top).

- Why do we care? (roughly speaking):
  - Use a lattice to represent properties of a program.
  - Join operation \(\sqcup\) handle information reaching a program point from multiple sources.
  - Meet operation \(\sqcap\) handle restrictions (e.g., conditions).
Galois Connection

- Connect two lattices
  - \((C, \sqsubseteq_c)\) representing “concrete” information
  - \((A, \sqsubseteq_a)\) representing abstract information

- Using two functions
  - \(\alpha: C \rightarrow A\) abstraction function
  - \(\gamma: A \rightarrow C\) concretization function

- such that
  - \(\alpha(C) \sqsubseteq_a A \iff C \sqsubseteq_c \gamma(A)\)

- Alternatively
  - \(\alpha\) and \(\gamma\) are order-preserving (monotone)
    - \(\forall a \in A \\alpha(\gamma(a)) \sqsubseteq_a a\)
    - \(\forall c \in C \ c \sqsubseteq_c \gamma(\alpha(c))\)
Galois Connection

- Why do we care? (roughly)
  - captures intuition: values in one lattice used to represent values in another lattice in a conservative manner
  - In a Galois Connection \( \alpha \) and \( \gamma \) determine one another, enough to define one, and can compute the other
    - \( \alpha(c) = \sqcap\{ a \mid c \sqsubseteq \gamma(a) \} \)
    - \( \gamma(a) = \sqcup\{ c \mid \alpha(c) \sqsubseteq a \} \)
  - global soundness theorem allows us to extend “local soundness” of individual operations to soundness of LFP computation
“Analysis Algorithm”

$\begin{align*}
a &= \perp \\
\text{while } a \sqsubseteq f(a) \text{ do } a = f(a)
\end{align*}$

- Complete lattice $(A, \sqsubseteq_a, \perp, T, \sqcup, \sqcap)$
- $\llbracket S \rrbracket^\#(a)$ the abstract transformer for each statement
- $a_1 \sqcup a_2$ join operation
- $a_1 \sqsubseteq a_2$ check for convergence (fixed point)
Parity Abstraction

1: while (x != 1) do {
2:   if (x % 2 == 0) {
3:     x := x / 2;
4:   } else {
5:     x := x * 3 + 1;
6:     assert (x % 2 == 0);
7:   }
8: }

Parity Abstraction

\[ \alpha(C) \sqsubseteq A \iff C \sqsubseteq \gamma(A) \]

program traces

abstract state \( x \mapsto E \)

\( \gamma(A) \)

\( A \)

\( C \)

\( \alpha(C) \)
Parity Abstraction

\[ \alpha(C) \sqsubseteq A \iff C \sqsubseteq \gamma(A) \]
Some traces of the example program

x = 3
1:x!=1 -> 2:xmod2!=0 -> 5:x=x*3+1 (10) -> 6:assert 10 mod2==0 -> 1:x!=1
-> 2:xmod2==0 -> 3:x=x/2 (5) -> 1:x!=1 -> 2:xmod2!=0 -> 5:x=x*3+1 (16)
-> 6:assert 16 mod2==0 -> 1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (8) -> 1:x!=1
-> 2:xmod2==0 -> 3:x=x/2 (4) -> 1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (2) ->
1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (1)

x = 5
1:x!=1 -> 2:xmod2!=0 -> 5:x=x*3+1 (16) -> 6:assert 16 mod2==0 -> 1:x!=1
-> 2:xmod2==0 -> 3:x=x/2 (8) -> 1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (4) ->
1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (2) -> 1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (1)

x = 7
1:x!=1 -> 2:xmod2!=0 -> 5:x=x*3+1 (22) -> 6:assert 22 mod2==0 -> 1:x!=1
-> 2:xmod2==0 -> 3:x=x/2 (11) -> 1:x!=1 -> 2:xmod2!=0 -> 5:x=x*3+1
(34) -> 6:assert 34 mod2==0 -> 1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (17) ->
1:x!=1 -> 2:xmod2!=0 -> 5:x=x*3+1 (52) -> 6:assert 52 mod2==0 -> 1:x!=1
-> 2:xmod2==0 -> 3:x=x/2 (26) -> 1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (13) ->
1:x!=1 -> 2:xmod2!=0 -> 5:x=x*3+1 (40) -> 6:assert 40 mod2==0 -> 1:x!=1
-> 2:xmod2==0 -> 3:x=x/2 (20) -> 1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (10) ->
1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (5) -> 1:x!=1 -> 2:xmod2!=0 -> 5:x=x*3+1
(16) -> 6:assert 16 mod2==0 -> 1:x!=1 -> 2:xmod2==0 -> 3:x=x/2 (8) -> ...

16
Some traces of the example program

\[ x = 9 \]

1: \( x = 1 \) -> 2: \( x \mod 2 \neq 0 \) -> 5: \( x = x \times 3 + 1 \) (28) -> 6: assert 28 \ mod 2 == 0 ->
1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (14) -> 1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) ->
3: \( x = x/2 \) (7) -> 1: \( x = 1 \) -> 2: \( x \mod 2 != 0 \) -> 5: \( x = x \times 3 + 1 \) (22) -> 6: assert 22
mod2 == 0 -> 1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (11) -> 1: \( x = 1 \) ->
2: \( x \mod 2 != 0 \) -> 5: \( x = x \times 3 + 1 \) (34) -> 6: assert 34 \ mod 2 == 0 -> 1: \( x = 1 \) ->
2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (17) -> 1: \( x = 1 \) -> 2: \( x \mod 2 != 0 \) -> 5: \( x = x \times 3 + 1 \) (52) ->
6: assert 52 \ mod 2 == 0 -> 1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (26) ->
1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (13) -> 1: \( x = 1 \) -> 2: \( x \mod 2 != 0 \) ->
5: \( x = x \times 3 + 1 \) (40) -> 6: assert 40 \ mod 2 == 0 -> 1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) ->
3: \( x = x/2 \) (20) -> 1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (10) -> 1: \( x = 1 \) ->
2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (5) -> 1: \( x = 1 \) -> 2: \( x \mod 2 != 0 \) -> 5: \( x = x \times 3 + 1 \) (16) ->
6: assert 16 \ mod 2 == 0 -> ...

\[ x = 11 \]

1: \( x = 1 \) -> 2: \( x \mod 2 != 0 \) -> 5: \( x = x \times 3 + 1 \) (34) -> 6: assert 34 \ mod 2 == 0 ->
1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (17) -> 1: \( x = 1 \) -> 2: \( x \mod 2 != 0 \) ->
5: \( x = x \times 3 + 1 \) (52) -> 6: assert 52 \ mod 2 == 0 -> 1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) ->
3: \( x = x/2 \) (26) -> 1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (13) -> 1: \( x = 1 \) ->
2: \( x \mod 2 == 0 \) -> 5: \( x = x \times 3 + 1 \) (40) -> 6: assert 40 \ mod 2 == 0 -> 1: \( x = 1 \) ->
2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (20) -> 1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (10) ->
1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) -> 3: \( x = x/2 \) (5) -> 1: \( x = 1 \) -> 2: \( x \mod 2 == 0 \) ->
5: \( x = x \times 3 + 1 \) (16) -> 6: assert 16 \ mod 2 == 0 -> ...
Collecting Semantics (label 6)

1: x! = 1 -> 2: x mod 2 != 0 -> 5: x = x * 3 + 1 (10) -> 6: assert 10 mod 2 == 0

1: x! = 1 -> 2: x mod 2 != 0 -> 5: x = x * 3 + 1 (10) -> 6: assert 10 mod 2 == 0 -> 1: x! = 1 -> 2: x mod 2 == 0 -> 3: x = x / 2 (5) -> 1: x! = 1 -> 2: x mod 2 != 0 -> 5: x = x * 3 + 1 (16) -> 6: assert 16 mod 2 == 0

1: x! = 1 -> 2: x mod 2 != 0 -> 5: x = x * 3 + 1 (16) -> 6: assert 16 mod 2 == 0

1: x! = 1 -> 2: x mod 2 != 0 -> 5: x = x * 3 + 1 (22) -> 6: assert 22 mod 2 == 0

1: x! = 1 -> 2: x mod 2 != 0 -> 5: x = x * 3 + 1 (22) -> 6: assert 22 mod 2 == 0 -> 1: x! = 1 -> 2: x mod 2 == 0 -> 3: x = x / 2 (11) -> 1: x! = 1 -> 2: x mod 2 != 0 -> 5: x = x * 3 + 1 (34) -> 6: assert 34 mod 2 == 0

1: x! = 1 -> 2: x mod 2 != 0 -> 5: x = x * 3 + 1 (22) -> 6: assert 22 mod 2 == 0 -> 1: x! = 1 -> 2: x mod 2 == 0 -> 3: x = x / 2 (11) -> 1: x! = 1 -> 2: x mod 2 != 0 -> 5: x = x * 3 + 1 (34) -> 6: assert 34 mod 2 == 0 -> 1: x! = 1 -> 2: x mod 2 == 0 -> 3: x = x / 2 (17) -> 1: x! = 1 -> 2: x mod 2 != 0 -> 5: x = x * 3 + 1 (52) -> 6: assert 52 mod 2 == 0

...
From Set of Traces to Set of States

1: \( x! = 1 \rightarrow 2: x \mod 2! = 0 \rightarrow 5: x = x \times 3 + 1 \) (10) \( \rightarrow 6: \text{assert } 10 \mod 2 = 0 \)

6: \( x \mapsto 10 \)

1: \( x! = 1 \rightarrow 2: x \mod 2! = 0 \rightarrow 5: x = x \times 3 + 1 \) (10) \( \rightarrow 6: \text{assert } 10 \mod 2 = 0 \rightarrow 1: x! = 1 \rightarrow 2: x \mod 2 = 0 \rightarrow 3: x = x/2 \) (5) \( \rightarrow 1: x! = 1 \rightarrow 2: x \mod 2! = 0 \rightarrow 5: x = x \times 3 + 1 \) (16) \( \rightarrow 6: \text{assert } 16 \mod 2 = 0 \)

6: \( x \mapsto 16 \)

1: \( x! = 1 \rightarrow 2: x \mod 2! = 0 \rightarrow 5: x = x \times 3 + 1 \) (16) \( \rightarrow 6: \text{assert } 16 \mod 2 = 0 \)

6: \( x \mapsto 16 \)

1: \( x! = 1 \rightarrow 2: x \mod 2! = 0 \rightarrow 5: x = x \times 3 + 1 \) (22) \( \rightarrow 6: \text{assert } 22 \mod 2 = 0 \)

6: \( x \mapsto 22 \)

1: \( x! = 1 \rightarrow 2: x \mod 2! = 0 \rightarrow 5: x = x \times 3 + 1 \) (22) \( \rightarrow 6: \text{assert } 22 \mod 2 = 0 \rightarrow 1: x! = 1 \rightarrow 2: x \mod 2 = 0 \rightarrow 3: x = x/2 \) (11) \( \rightarrow 1: x! = 1 \rightarrow 2: x \mod 2! = 0 \rightarrow 5: x = x \times 3 + 1 \) (34) \( \rightarrow 6: \text{assert } 34 \mod 2 = 0 \)

6: \( x \mapsto 34 \)

1: \( x! = 1 \rightarrow 2: x \mod 2! = 0 \rightarrow 5: x = x \times 3 + 1 \) (22) \( \rightarrow 6: \text{assert } 22 \mod 2 = 0 \rightarrow 1: x! = 1 \rightarrow 2: x \mod 2 = 0 \rightarrow 3: x = x/2 \) (11) \( \rightarrow 1: x! = 1 \rightarrow 2: x \mod 2! = 0 \rightarrow 5: x = x \times 3 + 1 \) (34) \( \rightarrow 6: \text{assert } 34 \mod 2 = 0 \rightarrow 1: x! = 1 \rightarrow 2: x \mod 2 = 0 \rightarrow 3: x = x/2 \) (17) \( \rightarrow 1: x! = 1 \rightarrow 2: x \mod 2! = 0 \rightarrow 5: x = x \times 3 + 1 \) (52) \( \rightarrow 6: \text{assert } 52 \mod 2 = 0 \)

6: \( x \mapsto 52 \)

...
Set of States

- still unbounded
- can abstract it using parity
- cannot compute it through the concrete semantics
- need to compute directly in the abstract

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6: x</td>
<td>10</td>
</tr>
<tr>
<td>6: x</td>
<td>16</td>
</tr>
<tr>
<td>6: x</td>
<td>16</td>
</tr>
<tr>
<td>6: x</td>
<td>16</td>
</tr>
<tr>
<td>6: x</td>
<td>22</td>
</tr>
<tr>
<td>6: x</td>
<td>34</td>
</tr>
<tr>
<td>6: x</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parity Abstraction

- concrete state: \( \text{Var} \rightarrow \mathbb{Z} \)
- abstract state: \( \text{Var} \rightarrow \{ \bot, E, O, T \} \)
  - even / odd
  - \( \bot \) non-initialized (bottom)
  - \( T \) either even or odd (top)
- Transformers:
  - \( \left[ x = x / 2 \right] \)(\( \sigma \)) = ?
  - \( \left[ x = x \times 3 + 1 \right] \)(\( \sigma \)) =
    - \( x \) is even, then \( \sigma[x \leftarrow O] \)
    - \( x \) is odd, then \( \sigma[x \leftarrow E] \)
Parity Abstraction

1: while (x != 1) do {  // x\rightarrow\{E, O\}
2:   if (x \% 2) == 0 {  // x\rightarrow\{E\}
3:       x := x / 2;  // x\rightarrow\{E, O\}
4:   } else {  // x\rightarrow\{O\}
5:       x := x * 3 + 1;  // x\rightarrow\{E\}
6:       assert (x \%2 ==0);  // x\rightarrow\{E\}
7:   }
8: }
Where does the Galois Connection help me?

- Establish Galois connection
- Show each individual transformer is sound
- Show each individual transformer is monotonic

- soundness of the analysis is guaranteed

- global soundness theorem
Sign Abstraction

- concrete state: Var → Z
- abstract state: Var → {⊥, 0, +, -, T}
  - zero, positive, negative
  - ⊥ non-initialized (bottom)
  - T (top)
Example

```c
main(int i) {
    int x=3,y=1;
    do {
        y = y + 1;
    } while(--i > 0)
    assert 0 < x + y
}
```
Sign and Parity

(slide from Patrick Cousot)
Constant Abstraction

\[ \begin{align*}
\rho &\in \mathbb{Z} \\
\text{State} &\equiv (\text{Var} \rightarrow \mathbb{Z}^T)_{\bot} \\
\end{align*} \]

(infinite lattice, finite height)

\[ \begin{align*}
\mathbb{T} &\quad \text{Variable not a constant} \\
-\infty &\quad \ldots \\
-1 &\quad \theta &\quad 1 &\quad \ldots \\
\infty &
\end{align*} \]
Constant Abstraction

- $L = ((\operatorname{Var} \rightarrow \mathbb{Z}^\top) \downarrow, \sqsubseteq)$
- $\sigma_1 \sqsubseteq \sigma_2$ iff $\forall v: \sigma_1(v) \sqsubseteq \sigma_2(v)$
- $\sqsubseteq$ ordering in the $\mathbb{Z}^\top \downarrow$ lattice

Examples:
- $[x \mapsto \bot, y \mapsto 42, z \mapsto \bot] \sqsubseteq [x \mapsto \bot, y \mapsto 42, z \mapsto 73]$
- $[x \mapsto \bot, y \mapsto 42, z \mapsto 73] \sqsubseteq [x \mapsto \bot, y \mapsto 42, z \mapsto \top]$
Constant Abstraction

\[ A: \text{AExp} \rightarrow (\text{State} \rightarrow \mathbb{Z}) \]

\[
A[x] \sigma = \begin{cases} 
\bot & \text{if } \sigma = \bot \\
\sigma(x) & \text{otherwise}
\end{cases}
\]

\[
A[n] \sigma = \begin{cases} 
\bot & \text{if } \sigma = \bot \\
n & \text{otherwise}
\end{cases}
\]

\[
A[a_1 \text{ op } a_2] \sigma = A[a_1] \sigma \text{ op } A[a_2] \sigma
\]

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| \{x := a\} \text{lab} | \begin{cases} 
\bot & \text{if } \sigma = \bot \\
\sigma[x \mapsto A[a] \sigma] & \text{otherwise}
\end{cases} |
| \{\text{skip}\} \text{lab} | \sigma                        |
| \{b\} \text{lab}       | \sigma                        |
Example

\[x := 42]\;
\[y := 73]\;
\text{(if } [?]\text{ then )} \quad [z := x + y];
\text{else )} \quad [z := 12];
\text{else) } \quad [w := z];

\[X \mapsto \bot, y \mapsto \bot, z \mapsto \bot, w \mapsto \bot]\]
\[X \mapsto 42, y \mapsto \bot, z \mapsto \bot, w \mapsto \bot]\]
\[X \mapsto 42, y \mapsto 73, z \mapsto \bot, w \mapsto \bot]\]
\[X \mapsto 42, y \mapsto 73, z \mapsto 115, w \mapsto \bot]\]
\[X \mapsto 42, y \mapsto 73, z \mapsto 12, w \mapsto \bot]\]
\[X \mapsto 42, y \mapsto 73, z \mapsto 12, w \mapsto 12]\]
\[X \mapsto 42, y \mapsto 73, z \mapsto \bot, w \mapsto 12]\]
Constant Propagation is Non Distributive

- Consider the transformer $f = \llbracket [y=x*x] \rrbracket^*$
- Consider two states $\sigma_1, \sigma_2$
  - $\sigma_1(x) = 1$
  - $\sigma_2(x) = -1$

$(\sigma_1 \sqcup \sigma_2)(x) = \top$

$f(\sigma_1 \sqcup \sigma_2)$ maps $y$ to $\top$

$f(\sigma_1)$ maps $y$ to $1$

$f(\sigma_2)$ maps $y$ to $1$

$f(\sigma_1) \sqcup f(\sigma_2)$ maps $y$ to $1$

$f(\sigma_1 \sqcup \sigma_2) \neq f(\sigma_1) \sqcup f(\sigma_2)$
Intervals Abstraction

\[ y \mapsto [3, 6] \]

\[ x \mapsto [1, 4] \]
Interval Lattice

(infinite lattice, infinite height)
Example

int x = 0;
if (?) x++;
if (?) x++;

[a1,a2] ∪ [b1,b2] = [min(a1,b1), max(a2,b2)]
Example

```c
int x = 0;
while(?) x++;
```

What now?
Widening

- Idea: replace join operator with a more conservative operator that will guarantee convergence
- a.k.a. acceleration

- Complete lattice \((A, \sqsubseteq)\)
- A function \(\nabla: A \times A \rightarrow A\) is a widening operator iff
  - for every two elements \(a_1, a_2 \in A\), \(a_1 \sqcup a_2 \sqsubseteq a_1 \nabla a_2\)
  - and for every increasing chain \(x_0, x_1, \ldots \in A\) the increasing chain \(y_0 = x_0, y_{n+1} = y_n \nabla x_{n+1}\) is finite.
An Algorithm for Computing Over-Approximation of $\text{lfp}$

$\text{lfp}(f) \subseteq \bigtriangleup_{n \in \mathbb{N}} f^n(\bot)$

$l = \bot$
while $f(l) \neq l$ do $l = f(l)$

- guarantee convergence even in infinite-height lattices with widening
Widening

- useful also in the finite case

```c
int x = 0;
while(x<10000) x++;
```
Cartesian vs. Relational Abstractions

- **Cartesian** (also called independent-attribute) abstraction abstracts each variable separately
  - set of points abstracted by a point of sets
    \{ (1,2), (3,4), (5,6) \} => \{1,3,5\},(2,4,6)\}
  - losing relationship between variables
  - e.g., intervals, constants, signs, parity

- **Relational abstraction** tracks relationships between variables
Octagon Abstraction

- abstract state is an intersection of linear inequalities of the form $\pm x \pm y \leq c$

- captures relationships common in programs (array access)
Example

proc incr (x:int) returns (y:int)
begin
  y = x+1;
end

var i:int;
begin
  i = 0;
  while (i<=10) do
    i = incr(i);
  done;
end
Result with Octagon

proc incr (x : int) returns (y : int) { 
/* [x>=0; -x+10>=0] */
    y = x + 1;
/* [x>=0; -x+10>=0; -x+y-1>=0; x+y-1>=0; y-1>=0; -x-y+21>=0; x-y+1>=0; -y+11>=0] */
}
begin
/* top */
i = 0; /* [i>=0; -i+11>=0] */
while i <= 10 do
    /* [i>=0; -i+10>=0] */
    i = incr(i); /* [i-1>=0; -i+11>=0] */
done; /* [i-11>=0; -i+11>=0] */
end
Polyhedral Abstraction

- abstract state is an intersection of linear inequalities of the form $a_1 x_2 + a_2 x_2 + \ldots + a_n x_n \leq c$

- represent a set of points by their convex hull

McCarthy 91 function

proc MC (n : int) returns (r : int) var t1 : int, t2 : int;
begin
   /* top */
   if n > 100 then
      /* [n-101>=0] */
      r = n - 10; /* [-n+r+10=0; n-101>=0] */
   else
      /* [-n+100>=0] */
      t1 = n + 11; /* [-n+t1-11=0; -n+100>=0] */
      t2 = MC(t1); /* [-n+t1-11=0; -n+100>=0; 
                      -n+t2-1>=0; t2-91>=0] */
      r = MC(t2); /* [-n+t1-11=0; -n+100>=0; 
                     -n+t2-1>=0; t2-91>=0; r-t2+10>=0; 
                     r-91>=0] */
   endif; /* [-n+r+10>=0; r-91>=0] */
end

var a : int, b : int;
begin /* top */
   b = MC(a); /* [-a+b+10>=0; b-91>=0] */
end
Operations on Polyhedra
Recap

- Cartesian
  - parity – finite
  - signs – finite
  - constant – infinite lattice, finite height
  - interval – infinite height

- Relational
  - octagon – infinite
  - polyhedra – infinite
Back to a bit of dataflow analysis...
Recap

- Represent properties of a program using a lattice \((L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)\)

- A continuous function \(f: L \rightarrow L\)
  - Monotone function when \(L\) satisfies ACC implies continuous

- Kleene’s fixedpoint theorem
  - \(\text{lfp}(f) = \bigsqcup_{n \in \mathbb{N}} f^n(\bot)\)

- A constructive method for computing the lfp
Some required notation

blocks : Stmt → P(Blocks)
blocks([x := a]^{lab}) = {[x := a]^{lab}}
blocks([skip]^{lab}) = {[skip]^{lab}}
blocks(S_1; S_2) = blocks(S_1) \cup blocks(S_2)
blocks(if [b]^{lab} then S_1 else S_2) = {[b]^{lab}} \cup blocks(S_1) \cup blocks(S_2)
blocks(while [b]^{lab} do S) = {[b]^{lab}} \cup blocks(S)

FV: (BExp \cup AExp) → Var
Variables used in an expression

AExp(a) = all non-unit expressions in the arithmetic expression a
similarly AExp(b) for a boolean expression b
Available Expressions Analysis

\[ \text{x := a+b}^1; \]
\[ \text{y := a*b}^2; \]
while \[ y > a+b \]^3 (  
\[ \text{a := a + 1}^4; \]
\[ \text{x := a + b}^5 \] )

(a+b) always available at label 3

For each program point, which expressions must have already been computed, and not later modified, on all paths to the program point.
Available Expressions Analysis

- **Property space**
  - \( \text{in}_{AE}, \text{out}_{AE}: \text{Lab} \rightarrow \varnothing (\text{AExp}) \)
  - Mapping a label to set of arithmetic expressions available at that label

- **Dataflow equations**
  - Flow equations – how to join incoming dataflow facts
  - Effect equations - given an input set of expressions \( S \), what is the effect of a statement
Available Expressions Analysis

- \( \text{in}_{AE}(\text{lab}) = \)
  - \( \emptyset \) when lab is the initial label
  - \( \cap \{ \text{out}_{AE}(\text{lab'}) | \text{lab'} \in \text{pred(lab)} \} \) otherwise
- \( \text{out}_{AE}(\text{lab}) = \ldots \)

<table>
<thead>
<tr>
<th>Block</th>
<th>out (lab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([x := a]^{lab})</td>
<td>(\text{in(lab)} \setminus { a' \in \text{AExp}</td>
</tr>
<tr>
<td>([\text{skip}]^{lab})</td>
<td>(\text{in(lab)})</td>
</tr>
<tr>
<td>([b]^{lab})</td>
<td>(\text{in(lab)} \cup \text{AExp(b)})</td>
</tr>
</tbody>
</table>

From now on going to drop the AE subscript when clear from context
Transfer Functions

1: \( x = a + b \)
2: \( y := a \times b \)
3: \( y > a + b \)
4: \( a = a + 1 \)
5: \( x = a + b \)

\[
\begin{align*}
\text{out(1)} &= \text{in(1)} \cup \{ a+b \} \\
\text{out(2)} &= \text{in(2)} \cup \{ a \times b \} \\
\text{out(3)} &= \text{in(3)} \cup \{ a + b \} \\
\text{out(4)} &= \text{in(4)} \setminus \{ a+b, a \times b, a+1 \} \\
\text{out(5)} &= \text{in(5)} \cup \{ a+b \}
\end{align*}
\]

\[
in(1) = \emptyset \\
in(2) = \text{out}(1) \\
in(3) = \text{out}(2) \cap \text{out}(5) \\
in(4) = \text{out}(3) \\
in(5) = \text{out}(4)
\]

\[
[x := a+b]^1; \\
[y := a \times b]^2; \\
\text{while } [y > a+b]^3 \text{ (}
\quad [a := a + 1]^4; \\
\quad [x := a + b]^5
\text{ )}
\]
Solution

1: \( x = a + b \)

in(1) = \( \emptyset \)

2: \( y := a \times b \)

in(2) = out(1) = \{ a + b \}

out(2) = \{ a+b, a*b \} \quad \text{in(3) = \{ a + b \}}

3: \( y > a + b \)

out(2) = \{ a+b, a*b \} \quad \text{in(3) = \{ a + b \}}

4: \( a = a + 1 \)

in(4) = out(3) = \{ a + b \}

out(4) = \emptyset

5: \( x = a + b \)

out(5) = \{ a+b \}
Kill/Gen

<table>
<thead>
<tr>
<th>Block</th>
<th>out (lab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([x := a])_{lab} ]</td>
<td>in(lab) ( \setminus { a' \in \text{AExp} \mid x \in \text{FV}(a') } \cup { a' \in \text{AExp}(a) \mid x \notin \text{FV}(a') } )</td>
</tr>
<tr>
<td>([\text{skip}]_{lab})</td>
<td>in(lab)</td>
</tr>
<tr>
<td>([b]_{lab})</td>
<td>in(lab) \union \text{AExp}(b)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>([x := a])_{lab} ]</td>
<td>{ a' \in \text{AExp} \mid x \in \text{FV}(a') }</td>
<td>{ a' \in \text{AExp}(a) \mid x \notin \text{FV}(a') }</td>
</tr>
<tr>
<td>([\text{skip}]_{lab})</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>([b]_{lab})</td>
<td>(\emptyset)</td>
<td>\text{AExp}(b)</td>
</tr>
</tbody>
</table>

\[\text{out}(\text{lab}) = \text{in}(\text{lab}) \setminus \text{kill}(B_{lab}) \cup \text{gen}(B_{lab})\]

\[B_{lab} = \text{block at label lab}\]
Why solution with largest sets?

\[
\text{in}(1) = \emptyset
\]

1: \( z = x + y \)

\[
\text{out}(1) = \text{in}(1) \cup \{ x+y \}
\]

\[
\text{in}(2) = \text{out}(1) \cap \text{out}(3)
\]

2: true

\[
\text{out}(2) = \text{in}(2)
\]

\[
\text{in}(3) = \text{out}(2)
\]

3: skip

\[
\text{out}(3) = \text{in}(3)
\]

\[
\text{in}(1) = \emptyset
\]

\[
\text{in}(2) = \text{out}(1) \cap \text{out}(3)
\]

\[
\text{in}(3) = \text{out}(2)
\]

\[
[z := x+y]^1;\\
\text{while [true]}^2 (\\
\quad [\text{skip}]^3;\\
\quad )
\]

After simplification: \( \text{in}(2) = \text{in}(2) \cap \{ x+y \} \)

Solutions: \( \{ x+y \} \) or \( \emptyset \)
### Reaching Definitions Revisited

<table>
<thead>
<tr>
<th>Block</th>
<th>out (lab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([x := a]_{lab})</td>
<td>(\text{in(lab)} \setminus {(x, l)</td>
</tr>
<tr>
<td>([\text{skip}]_{lab})</td>
<td>(\text{in(lab)})</td>
</tr>
<tr>
<td>([b]_{lab})</td>
<td>(\text{in(lab)})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>([x := a]_{lab})</td>
<td>({(x, l)</td>
<td>l \in \text{Lab}})</td>
</tr>
<tr>
<td>([\text{skip}]_{lab})</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>([b]_{lab})</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

For each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.
Why solution with smallest sets?

\[ \text{in}(1) = \{ (x,?), (y,?), (z,?) \} \]

1: \( z = x+y \)
\[
\text{out}(1) = (\text{in}(1) \setminus \{ (z,?) \}) \cup \{ (z,1) \}
\]
\[ \text{in}(2) = \text{out}(1) \cup \text{out}(3) \]

2: \text{true}
\[
\text{out}(2) = \text{in}(2)
\]
\[ \text{in}(3) = \text{out}(2) \]

3: \text{skip}
\[
\text{out}(3) = \text{in}(3)
\]

\[ \text{in}(1) = \{ (x,?), (y,?), (z,?) \} \]
\[ \text{in}(2) = \text{out}(1) \cup \text{out}(3) \]
\[ \text{in}(3) = \text{out}(2) \]

After simplification: \( \text{in}(2) = \text{in}(2) \cup \{ (x,?), (y,?), (z,1) \} \)

Many solutions: any superset of \( \{ (x,?), (y,?), (z,1) \} \)
Live Variables

[ x :=2]¹;
[y:=4]²;
[x:=1]³;
(if [y>x]⁴ then [z:=y]⁵
else [z:=y*y]⁶);
[x:=z]⁷

For each program point, which variables may be live at the exit from the point.
Live Variables

\[
\begin{align*}
[x:=2] &; \\
[y:=4] &; \\
[x:=1] &; \\
\text{(if } [y>x] \text{ then } [z:=y] &; \\
\text{else } [z:=y*y] &; \\
[x:=z] & \\
\end{align*}
\]
Live Variables

\[ x := 2 \]
\[ y := 4 \]
\[ x := 1 \]
(if \[ y > x \] then \[ z := y \] else \[ z := y \times y \])
\[ x := z \]

Block | kill | gen
--- | --- | ---
\[ x := a \] | \{ x \} | \{ FV(a) \}
\[ \text{skip} \] | \Ø | \Ø
\[ b \] | \Ø | FV(b)

1: x := 2
2: y := 4
3: x := 1
4: y > x
5: z := y
6: z := y \times y
7: x := z
Live Variables: solution

\[ x := 2 \]
\[ y := 4 \]
\[ x := 1 \]
(if \( y > x \) then \[ z := y \]
else \[ z := y \times y \])
\[ x := z \]

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ x := a ]</td>
<td>{ x }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>[ skip ]</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>[ b ]</td>
<td>Ø</td>
<td>FV(b)</td>
</tr>
</tbody>
</table>

Diagram:

- \( x := 2 \)
- \( y := 4 \)
- \( x := 1 \)
- (if \( y > x \) then \( z := y \)
else \( z := y \times y \))
- \( x := z \)
Why solution with smallest set?

After simplification: \( \text{in}(1) = \text{in}(1) \cup \{x\} \)

Many solutions: any superset of \( \{x\} \)
Monotone Frameworks

\[ \text{In}(\text{lab}) = \begin{cases} \text{Initial} & \text{when lab }\in\text{ Entry labels} \\ \sqcup \{ \text{out}(\text{lab}') \mid (\text{lab}', \text{lab}) \in \text{CFG edges} \} & \text{otherwise} \end{cases} \]

\[ \text{out}(\text{lab}) = f_{\text{lab}}(\text{in}(\text{lab})) \]

- $\sqcup$ is $\cup$ or $\cap$
- CFG edges go either forward or backwards
- Entry labels are either initial program labels or final program labels (when going backwards)
- Initial is an initial state (or final when going backwards)
- $f_{\text{lab}}$ is the transfer function associated with the blocks $B^{\text{lab}}$
Forward vs. Backward Analyses

1: \( x := 2 \) 
\( \{ (x,?), (y,?), (z,?) \} \)

2: \( y := 4 \) 
\( \{ (x,1), (y,?), (z,?) \} \)

4: \( y > x \) 
\( \{ (x,1), (y,2), (z,?) \} \)

5: \( z := y \)

6: \( z = y \cdot y \)

7: \( x := z \)

\( \emptyset \)
Must vs. May Analyses

- When $\cap$ is $\cap$ - must analysis
  - Want largest sets that solve the equation system
  - Properties hold on all paths reaching a label (exiting a label, for backwards)

- When $\cup$ is $\cup$ - may analysis
  - Want smallest sets that solve the equation system
  - Properties hold at least on one path reaching a label (existing a label, for backwards)
Example: Reaching Definition

- $L = \emptyset (\text{Var} \times \text{Lab})$ is partially ordered by $\subseteq$
- $\bot$ is $\bigcup$
- $L$ satisfies the Ascending Chain Condition because $\text{Var} \times \text{Lab}$ is finite (for a given program)
Example: Available Expressions

- $L = \varnothing(AExp)$ is partially ordered by $\supseteq$
- $\sqcup$ is $\cap$
- $L$ satisfies the Ascending Chain Condition because $AExp$ is finite (for a given program)
## Analyses Summary

<table>
<thead>
<tr>
<th></th>
<th>Reaching Definitions</th>
<th>Available Expressions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>$\mathcal{G}(\text{Var} \times \text{Lab})$</td>
<td>$\mathcal{G}(\text{AExp})$</td>
<td>$\mathcal{G}(\text{Var})$</td>
</tr>
<tr>
<td>$\sqsubseteq$</td>
<td>$\sqsubseteq$</td>
<td>$\sqsupseteq$</td>
<td>$\sqsubseteq$</td>
</tr>
<tr>
<td>$\sqcup$</td>
<td>$\sqcup$</td>
<td>$\sqcap$</td>
<td>$\sqcup$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\emptyset$</td>
<td>$\text{AExp}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Initial</td>
<td>${(x,?) \mid x \in \text{Var}}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Entry labels</td>
<td>${\text{init}}$</td>
<td>${\text{init}}$</td>
<td>final</td>
</tr>
<tr>
<td>Direction</td>
<td>Forward</td>
<td>Forward</td>
<td>Backward</td>
</tr>
<tr>
<td>F</td>
<td>${f : \text{L} \rightarrow \text{L} \mid \exists k, g : f(\text{val}) = (\text{val} \setminus k) \cup g}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_{lab}$</td>
<td>$f_{lab}(\text{val}) = (\text{val} \setminus \text{kill}) \cup \text{gen}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analyses as Monotone Frameworks

- Property space
  - Powerset
  - Clearly a complete lattice

- Transformers
  - Kill/gen form
  - Monotone functions (let’s show it)
Monotonicity of Kill/Gen transformers

- Have to show that $x \subseteq x'$ implies $f(x) \subseteq f(x')$
- Assume $x \subseteq x'$, then for kill set $k$ and gen set $g$
  $(x \setminus k) \cup g \subseteq (x' \setminus k) \cup g$

- Technically, since we want to show it for all functions in $F$, we also have to show that the set is closed under function composition
Distributivity of Kill/Gen transformers

- Have to show that \( f(x \sqcup y) \sqsubseteq f(x) \sqcup f(y) \)
- \( f(x \sqcup y) = ((x \sqcup y) \setminus k) \cup g \)
  = \( ((x \setminus k) \sqcup (y \setminus k)) \cup g \)
  = \( (((x \setminus k) \cup g) \sqcup ((y \setminus k) \cup g)) \)
  = \( f(x) \sqcup f(y) \)

- Used distributivity of \( \sqcup \) and \( \cup \)
  - Works regardless of whether \( \sqcup \) is \( \cup \) or \( \cap \)