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:
    
    \[
    S \rightarrow x := E \mid S ; S \mid \text{skip} \\
    \mid \text{if } E \text{ then } S \text{ else } S \\
    \mid \text{while } E \text{ do } S \\
    E \rightarrow x \mid \# \mid E \circ E \\
    \circ \in \{+, –, *, /, =, ≠, <, >, ≤, ≥\}
    \]


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 $s$

Semantics of expressions

- $\llbracket x := e \rrbracket \sigma = \sigma[x \mapsto \llbracket e \rrbracket \sigma]$
- $\llbracket x \rrbracket \sigma = \sigma(x)$
- $\llbracket n \rrbracket \sigma = n$
- $\llbracket e_1 \bowtie e_2 \rrbracket \sigma = \llbracket e_1 \rrbracket \sigma \bowtie \llbracket e_2 \rrbracket \sigma$

(For Boolean values: true $= 1$, false $= 0$)

Let's prove a property!

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

Let's prove $\llbracket i = 0 \rrbracket = \llbracket \text{if } i = 0 \text{ then } i := 0 \text{ else } \text{exit} \rrbracket$.

Galois Connection

Concrete state space

Abstract state space

sets of stores

descriptions of sets of stores

α

β

γ

δ
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,

\[ \alpha(\gamma(a)) \sqsubseteq a \land c \sqsubseteq \gamma(\alpha(c)) \]
Concretization Function — Revisited

- $\gamma : \{\bot, 0, +, -, T\} \rightarrow \mathcal{P}(\mathbb{Z})$
- We can use $\beta$ to define $\gamma$ as well
  - $\beta : \mathbb{Z} \rightarrow \{\bot, 0, +, -, T\}$
- $\alpha(S) = \bigsqcup \{\beta(\sigma) | \sigma \in S\}$
- $\gamma(a) = \bigsqcup \{\sigma \in \mathbb{Z} | \beta(\sigma) \sqsubseteq a\}$

Abstract Interpretation

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

Abstract State — Example

- WHILE program with $k$ variables $v_1, \ldots, v_k$
  - $\mathcal{P}(\mathbb{Z}^k)$ $\sigma \in \{\bot, 0, +, -, T\}^k$

Abstract Semantics — Example

- WHILE program with $k$ variables $v_1, \ldots, v_k$
  - $\alpha : \mathcal{P}(\mathbb{Z}^k) \rightarrow \{\bot, 0, +, -, T\}^k$
  - $[x := e] \sigma^i = \sigma \circ [x \mapsto [e] \sigma^i]$
Abstract Semantics — Example

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

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.

Properties of Abstract Semantics

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

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

\[ \alpha([s](\sigma)) \subseteq [s] (\alpha(\sigma)) \]

Transformer Soundness

1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
4:   y := y + 1;
5:   i := i - 1
6: 
A sound abstract transformer should always — for every state — produce results that are a superset of what a concrete transformer would produce.

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

This transformer is sound, but it's not precise.

\[ y := y + 1 \]

This abstract state:

\[ x, y, i \]

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

\[ y := y + 1 \]

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

\[ x, y, i \]

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

\[ x, y, i \]

Best Abstract Transformer

- It is easy to be sound and imprecise: always produce \( T \)
- 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 ≥ 0) do`
   4. `y := y + 1;`
   5. `i := i - 1;`
4. `assert 0 ≤ y - x;`

Numerical Domains

Instead of abstracting variable values using the sign of the value, we will abstract them using an interval.

Intervals Domain

Instead of abstracting variable values using the sign of the value, we will abstract them using an interval.

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

\[
\begin{align*}
[a, b] + [c, d] &= [a + c, b + d] \\
[a, b] \cdot c &= \min\{a \cdot c, b \cdot c\}, \max\{a \cdot c, b \cdot c\} \\
[a, b] \cdot [c, d] &= (a, b) \sqcup (a, c) \sqcup (a, d) \sqcup (a, b) \sqcup (a, b) \sqcup (a, c) \sqcup (a, d) \sqcup (a, b)
\end{align*}
\]
Let the iterations begin!

1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
   4:   y := y + i;
   5:   i := i - 1
7: assert 0 ≤ y - x

x := 5
y := 7
i ≥ 0
y := y + i
i := i - 1

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

x := 5
y := 7
i ≥ 0
y := y + 1
i := i - 1

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

• \( \nabla : A \times A \rightarrow A \) such that:
  ▷ for any ascending chain \( x_0 \subseteq x_1 \subseteq x_2 \subseteq \cdots \)
    \( \quad w_0 = x_0, w_{i+1} = w_i \nabla x_{i+1} \)
    \( \quad w_i \) stabilizes

\[
\begin{array}{cccccccc}
  & w_0 & w_1 & w_2 & w_3 & \cdots & w_{i+1} & w_{i+2} & w_{i+3} \\
\hline
  x_0 & x_1 & x_2 & x_3 & \cdots & x_{i+1} & x_{i+2} & x_{i+3} & \cdots
\end{array}
\]

\[
3k. w_k = w_{k+1} = w_{k+2} = \cdots
\]

Widening

• \( \nabla : A \times A \rightarrow A \)
  ▷ \( w_0 = x_0, w_{i+1} = w_i \nabla x_{i+1} \)
  \( w_i \) stabilizes

• Example — for intervals
  \[
  x \nabla \bot = \bot \nabla x = x
  \]
  \[
  [a_2, b_2] \nabla [a_3, b_3] = [c, d]
  \]
  \[
  c =
  \begin{cases}
    a_2 & \text{if } b_2 \leq a_3 \\
    a_3 & \text{if } b_2 \leq a_3
  \end{cases}
  \]
  \[
  d =
  \begin{cases}
    b_3 & \text{if } b_2 \leq a_3 \\
    b_2 & \text{if } b_2 \leq a_3
  \end{cases}
  \]
1: x := 5;
2: y := 7;
3: while (i ≥ 0) do
4:   y := y + 1;
5:   i := i - 1
6: end
7: assert 0 ≤ y - x

46

Abstraction is not an elephant

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

47

Variable Relations

- A very useful property, in particular for bounds checking

48

A New Domain

Octagons Domain
- **Octagon** = a set of inequalities of the form
  \[ \pm v_i \leq c \pm v_j \leq c \quad (i \neq j) \]

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

\[
\begin{align*}
\begin{array}{c}
\text{Octagons Domain} \\
\text{• Octagon} = \text{a set of inequalities of the form} \\
\text{• There is more than one way to represent a} \\
\text{• Every (signed) variable and every pair of (signed) } \\
\text{variables has exactly one bound (may be } \infty \text{)}
\end{array}
\end{align*}
\]

- **Closure for Octagons**
  - **S** is a **closed octagon** iff:
    - For any \( i, j, c \), such that \( S \models v_i + v_j \leq c \), there exists \( c' \leq c \) such that \( v_i + v_j \leq c' \in S \)
    - similarly for \( -v_i + v_j \leq c \), \( -v_i - v_j \leq c \), and for \( v \leq c \)
- **Canonical representation:**
  - Every (signed) variable and every pair of (signed) variables has exactly one bound (may be \( \infty \))

- **Matrix Representation**
  - **Example:**
    \[
    \begin{bmatrix}
    x & y & 5 & 7 & \leq & 12 \\
    -x & -y & 0 & 0 & \leq & \infty \\
    x & y & 0 & 0 & \leq & 12 \\
    \end{bmatrix}
    \]
    \[
    \begin{bmatrix}
    -x & -y & 0 & 0 & \leq & \infty \\
    x & y & 0 & 0 & \leq & 12 \\
    \end{bmatrix}
    \]

- **Order Relation on Octagons**
  - **\( S_1 \sqsubseteq S_2 \)** iff whenever \( \pm v_i \pm v_j \leq c \in S_2 \), there is a \( c' \leq c \) such that \( \pm w \pm v_j \leq c' \in S_1 \)
  - (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

$$\begin{align*}
\left\{ \begin{array}{l}
x \leq 5 \\
x + y \leq 10
\end{array} \right\} & \sqcup \left\{ \begin{array}{l}
x \leq 4 \\
x + y \leq 11
\end{array} \right\} \\
\downarrow & \\
\left\{ \begin{array}{l}
x \leq 5 \\
x + y \leq 11
\end{array} \right\}
\end{align*}$$

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 $e$

Octagon — Examples

- Assume input array of size $n$, show that array accesses are safe.

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:
    - $x - y \geq -20$
    - $x - 3y \leq 2$
    - $x + y \leq 5$

- 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:
c -= 1
if n > 100: n -= 10
else: n += 11;
c += 2
return n
```

• 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 *ptr points to x after statement S2.
  • In a C-like language that has pointers, we must know the 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 $b$
  - Out-degree may be $> 1$ if there are multiple paths
- Of course, points-to is different at different program locations

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

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

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

Strong updates vs. weak update (why?)

Dynamic Allocation

- What to do with \( x := \text{new } Z[\ldots] \)?
  - Program can create an unbounded number of objects
  - Need some static naming scheme for dynamically allocated objects
- AbsObj
  - Single name for the entire heap
    \( \text{AbsObj} = \{H\} \)
  - Type-based static names
    \( \text{AbsObj} = \{T \mid T \text{ is a type in the program} \} \)
  - Name based on static allocation site
    \( \text{AbsObj} = \{\mu \mid \text{statement } \mu : p := \text{new } Z[a] \} \)

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
```

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

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
```
Aliasing Analysis

Derived from result of point-to analysis

1. p := new Z[5];
2. q := 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

Abstract Domain:
Points-to × Signs

Example: Aliasing + Available Expressions

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
**Journey inside a compiler**

**Lexical Analysis**

- **Syntax Analysis**
- **Sem. Analysis**
- **Inter. Rep.**
- **Code Gen.**

---

**Intermediate Representation**

- \( t_1 = \text{inttofloat}(60) \)
- \( t_2 = \text{id3} \times t_1 \)
- \( t_3 = \text{id2} + t_2 \)
- \( \text{id1} = t_3 \)

---

**Optimized**

- \( t_1 = \text{id3} \times 60.0 \)
- \( \text{id1} = \text{id2} + t_1 \)

---

**Assembly**

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