1. **Theoretical Compilation Lecture 12**

2. **You are here**

3. **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, ...

4. **Static Program Analysis**
   - No need to run the program!
     - No concrete input needed!

5. **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 → x := E | S ; S | skip
  | if E then S else S
  | while E do S
E → x | E ⊕ E
∅ ∈ {+, −, *, /, =, ≠, <, >, ≤, ≥}
```

Denotational Semantics

- σ : Var → ℤ
  - A state of the program (also called a store)
  - Σ — the set of all such states
  - ⟦s⟧ : Σ → Σ
    - ⟦s⟧σ is the state resulting from σ after executing the statement s

Denotational Semantics

- ⟦x := e⟧σ = σ[x ↦ ⟦e⟧σ]

- ⟦s₁ ; s₂⟧σ = ⟦s₂⟧(⟦s₁⟧σ)

- ⟦if E then s₁ else s₂⟧σ = \begin{cases} ⟦s₁⟧σ & \text{if } ⟦e⟧σ = true \\ ⟦s₂⟧σ & \text{if } ⟦e⟧σ = false \end{cases}

- ⟦while E do S⟧σ = \begin{cases} σ & \text{if } ⟦e⟧σ = false \\ ?? & \text{if } ⟦e⟧σ = true \end{cases}

Galois Connection

sets of stores → descriptions of sets of stores

Concrete State Space

Abstract State Space
Galois Connection

- Lattices C and A
- Functions \( \alpha : C \to A \) and \( \gamma : A \to C \)

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

- Equivalently,

\[
\alpha(\gamma(a)) \sqsubseteq a \land c \sqsubseteq \gamma(\alpha(c))
\]
Abstract Semantics — Example

- WHILE program with $k$ variables $v_1, \ldots, v_k$
  - $\alpha : \mathcal{A}(\mathbb{Z}^k) \to \{\bot, 0, +, -, \top\}^k$
- $[x := e] \sigma' = \sigma [x \mapsto [e] \sigma']$
- $[s_1; s_2] \sigma' = [s_2] ([s_1] \sigma')$

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.
A sound abstract transformer should always — for every state — produce results that are a superset of what a concrete transformer would produce.

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

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 >= 0) do
   4: y := y + 1;
   5: i := i - 1
4: assert i >= x + y

Coming Up

THEORY OF COMPILATION
LECTURE 12

Exam

- 20% Compiler Phases
- 40% Syntax, Semantics, Code generation
- 10% Optimizations
- 30% Static Analysis

Reminder:

Galois Connection
Abstract Domain

Concrete State Space

Abstract State Space

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

Abstract Semantics

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

Proving Properties

Numerical Domains

Intervals Domain

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

Intervals: Transformers

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

Intervals: Transformers

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

Intervals: Transformers

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

- $\nabla : A \times A \rightarrow A$
- $x \sqcup y \subseteq x \nabla y$
- Example — for intervals
  - $x \nabla \perp = \perp$
  - $[a_1, b_1] \nabla [a_2, b_2] = [c, d]$
  - $c = \begin{cases} a_1 & \text{if } a_1 \leq a_2 \\ \infty & \text{if } a_1 > a_2 \end{cases}$
  - $d = \begin{cases} b_1 & \text{if } b_1 \leq b_2 \\ \infty & \text{if } b_1 > b_2 \end{cases}$

Try again.

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

Abstraction is not an elephant

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

- A very useful property, in particular for bounds checking

```c
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;
    }
}
```

```c
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

Octagons Domain

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

  Semantics: intersection of half-planes

Octagons Domain

- There is more than one way to represent a single octagonal region.
  - E.g.
    \[ \gamma \left( \begin{array}{c} x \leq 5 \\ y \leq 7 \end{array} \right) = \gamma \left( \begin{array}{c} x \leq 5 \\ y \leq 7 \end{array} \right) \]

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

\[
(r \leq 5) \land (r + y \leq 10) \sqcup (r \leq 4) \land (r + y \leq 11)
\]

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:

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

An abstract state is again a conjunction of constraints:

\[
\begin{align*}
  x - y &\geq -20 \\
  x - 3y &\leq 2 \\
  x + y &\geq 5
\end{align*}
\]

The slope can vary.

Order, join, transformers require solving linear equations.

Polyhedra: Example

- McCarthy’s “91 function” [Manna, Pnueli, McCarthy 1970]

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

```python
def m(n):
    x = y = 0 ; n = 0
    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

Constants
Signs
Intervals
Octagons
Polyhedra

Relational

Pointer Analysis

Simple Example

```
x := 5
ptr := &x
*ptr := 9
y := x
```

- 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 \( \mathbb{Z}, \mathbb{Z}^*, \mathbb{Z}^{**}, \ldots \)

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

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 / Steengard's algorithm)

Points-to: Abstract Semantics

- \(s \models G = G'\)

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
    - \(\text{AbsObj} = \{\} \)
  - Type based static names
    - \(\text{AbsObj} = \{T | T \text{ is a type in the program}\}\)
  - Name based on static allocation site
    - \(\text{AbsObj} = \{\text{stmt(s) is p := new}\}_{\text{stmt(s)}}\)
Dynamic Allocation

- **AbsObj** — set of abstract object names
  - 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"

Points-to Analysis: Example

Aliasing Analysis

Derived from result of point-to analysis

Example: Pointers + Sign

Abstract Domain:

\[
\begin{align*}
\text{Points-to} & \times \text{Signs} \\
\end{align*}
\]
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

\[
\text{float initial, rate; position = initial + rate \times 60}
\]

Token Stream

\[
<\text{ID,position}> \rightarrow <\text{ID,initial}> + <\text{ID,rate}> \times <\text{60}>
\]

Intermediate Representation

Annotated AST

Symbol Table

- position: float
- initial: float
- rate: float

Symbol Table

- position: float
- initial: float
- rate: float

Annnotated AST

Intermediate Representation

60
Journey inside a compiler

Lexical Analysis
Syntax Analysis
Semantics Analysis
Intermediate Representation

$t_1 = \text{inttofloat}(60)$
$t_2 = \text{id}_3 \times t_1$
$t_3 = \text{id}_2 + t_2$
$id_1 = t_3$

Intermediate Representation

$t_1 = \text{inttofloat}(60)$
$t_2 = \text{id}_3 \times t_1$
$t_3 = \text{id}_2 + t_2$
$id_1 = t_3$

Intermediate Representation

$t_1 = \text{inttofloat}(60)$
$t_2 = \text{id}_3 \times t_1$
$t_3 = \text{id}_2 + t_2$
$id_1 = t_3$

Optimized

$t_1 = \text{id}_3 \times 60.0$
$id_1 = \text{id}_2 + t_1$

Optimized

$t_1 = \text{id}_3 \times 60.0$
$id_1 = \text{id}_2 + t_1$

Assembly

LDF $R2, \text{id}_3$
MULF $R2, R2, \#60.0$
LDF $R1, \text{id}_2$
ADDF $R1, R1, R2$
STF $\text{id}_1, R1$

You Have Reached
Your Destination