Theoretical Guide to Compilation

Lecture 12

You are here

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 → 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 = \text{true} \\ ⟦S2⟧σ & \text{if } ⟦E⟧σ = \text{false} \end{cases}

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

Galois Connection

Concrete State Space

Abstract State Space

sets of stores

-descriptions of sets of stores
Galois Connection

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

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

- Equivalently,

$$\alpha(\gamma(a)) \sqsubseteq a \land c \sqsubseteq \gamma(\alpha(c))$$

Concrete State Space

Abstract State Space

$\alpha(\gamma(a)) \sqsubseteq a \land c \sqsubseteq \gamma(\alpha(c))$

Abstract Domain — Example

Abstraction Function — Example

- $\alpha : \mathcal{P}(\mathbb{Z}) \to \{\bot, 0, +, -\}$
- It is useful to define an auxiliary function $\beta$

$$\beta : \mathbb{Z} \to \{\bot, 0, +, -\}$$

$$\alpha(S) = \sqcup \{\beta(\sigma) \mid \sigma \in S\}$$

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.

$$\sigma^A \in A \quad \square \quad \text{abstract state}$$

$$[S]^A : A \to A \quad \square \quad \text{abstract transformer}$$
Abstract Semantics — Example

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

- $[x := e] \sigma' = \alpha[x := \llbracket e \rrbracket] \alpha$

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

- $[v \geq e] \sigma' = \sigma' \cap (v \geq \llbracket e \rrbracket) \sigma' \in (+, 0) \cap (+, \top) : \top$

Abstract Semantics

\[
\begin{array}{ccc}
\sigma & \xrightarrow{[s]} & \sigma' \\
\alpha & \downarrow & \alpha \\
\sigma & \xrightarrow{[s]} & \sigma'
\end{array}
\]

Abstract Semantics

\[
\begin{array}{ccc}
\sigma & \xrightarrow{[s]} & \sigma' \cap \alpha([s](\sigma)) \\
\alpha & \downarrow & \alpha \\
\sigma & \xrightarrow{[s]} & \sigma'
\end{array}
\]

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.
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:

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!

Let's prove a property!

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

Let's prove a property!

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

Let’s prove a property!

Let’s prove a property!

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

Let’s prove a property!

Let’s prove a property!

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

Let’s prove a property!

Let’s prove a property!

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

Let’s prove a property!

Let’s prove a property!

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

Let’s prove a property!

Let’s prove a property!

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

Intervals: Transformers

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

Intervals: Transformers

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

Intervals: Transformers

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

Intervals: Transformers

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

Intervals: Transformers

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

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

• ∇ : A × A → A such that:
  • for any ascending chain x₀ ⊆ x₁ ⊆ x₂ ⊆ ... ,
    w₀ = x₀, w₁ = w₀ ∨ x₁; w₀ stabilizes

[ ... ]

Example — for intervals

• x ∨ ⊥ = ⊥ ∨ x = x
• [a₂, b₂] ∨ [a₁, b₁] = [c, d]
  c = a₁ if a₂ ≤ a₁
  d = b₁ if b₂ ≥ b₁
  b₁ = a₂ if a₂ ≥ a₁
  b₂ = b₁ if b₁ ≤ b₂
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

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.\[
  \gamma\left(\begin{cases}
  x \leq 5 \\
  y \leq 7
  \end{cases}\right) = \gamma\left(\begin{cases}
  x \leq 5 \\
  y \leq 7 \\
  x + y \leq 12
  \end{cases}\right)
  \]

Closure for Octagons

- S is a closed octagon iff:
  - For any \(i, j, c\), such that \(S = v_i + v_j \leq c\), 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 \(v_i \leq c\)

  - Canonical representation:
    - Every (signed) variable and every pair of (signed) variables has exactly one bound (may be \(\infty\))

Matrix Representation

- Example:
  \[
  \begin{pmatrix}
  x & y & z \\
  x & y & z \\
  x & y & z
  \end{pmatrix}
  \]

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

- \(S_1 \cup S_2\) can be computed by taking piecewise maximum of bounds of corresponding inequalities
  \[
  \begin{cases}
  x \leq 5 \\
  x + y \leq 10
  \end{cases}
  \cup
  \begin{cases}
  x \leq 4 \\
  x + y \leq 11
  \end{cases}
  \]
  \[
  \begin{cases}
  x \leq 5 \\
  x + y \leq 11
  \end{cases}
  \]
Join for Octagons

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

\[
\begin{align*}
\{ x \leq 2 \} & \sqcup \{ y \leq 7 \} = \{ x \leq 2 \} \sqcup \{ x \leq 1 \} & \{ y \leq 7 \} \\
\{ x \leq 5 \} & \sqcup \{ y \leq 7 \} = \{ x \leq 5 \} \sqcup \{ y \leq 11 \} \\
\{ x \leq 5 \} & \sqcup \{ y \leq 11 \} = \{ x \leq 5 \} \sqcup \{ x \leq 12 \} \\
\end{align*}
\]

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 $e$

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:

$x = y + 20$
$x = 3y + 2$
$x + y \leq 5$

Order, join, transformers:

Require solving linear equations.

Polyhedra: Example

- McCarthy’s “91 function”

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

\[
def m(n):
  c = 1
  while c != 0:
    c -= 1
    if n > 100: n -= 10
    else: n += 11; c += 2
  return n
\]

Numerical Domains: Summary
**Pointer Analysis**

### Simple Example

```
x := 5  S1
ptr := &x  S2
*ptr := 9  S3
y := x  S4
```

- 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.

### 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 $b$

  ![Points-to Relation Diagram]

- Of course, points-to is different at different program locations

  - Out-degree may be > 1 if there are multiple paths
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.
  - $\perp$ is a graph with no edges.
  - $\sqsubseteq$ is the subgraph relation (edge subset)
  - $\sqcup$ is obtained 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

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

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

\[
\begin{align*}
x := *y
G &\Rightarrow G' \\
G' = G &\text{ with } pt'(x) \rightarrow U\{pt(a) | a \in pt(y)\} \\
\end{align*}
\]

\[
\begin{align*}
*x := y
G &\Rightarrow G' \\
G' = G &\text{ with } pt'(a) \rightarrow pt(y) \text{ for all } a \in pt(x) \\
\end{align*}
\]

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
  - Type-based static names
    - AbsObj = $\{H\}$
    - Name based on static allocation site
      - AbsObj = $\{p : \text{stmt(p) is p := newZ[a]}\}$

Dynamic Allocation

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

- **Basically**: model every "new" as "address of"

```plaintext
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

```plaintext
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

```plaintext
z and p may not alias
q and p may not alias
z and q may alias
```

Example: Pointers + Sign

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

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

Lexical Analysis
Syntax Analysis
Sem. Analysis
Inter. Rep.
Code Gen.

Journey inside a compiler

Lexical Analysis
Syntax Analysis
Sem. Analysis
Inter. Rep.
Code Gen.

Journey inside a compiler

Lexical Analysis
Syntax Analysis
Sem. Analysis
Inter. Rep.
Code Gen.

Journey inside a compiler

Lexical Analysis
Syntax Analysis
Sem. Analysis
Inter. Rep.
Code Gen.

Journey inside a compiler

Lexical Analysis
Syntax Analysis
Sem. Analysis
Inter. Rep.
Code Gen.
Journey inside a compiler

Intermediate Representation

\[\begin{align*}
    t_1 &= \text{inttofloat}(46) \\
    t_2 &= \text{id3} \times t_1 \\
    t_3 &= \text{id2} + t_2 \\
    \text{id1} &= t_3
\end{align*}\]

Optimized

\[\begin{align*}
    t_1 &= \text{id3} + 60.0 \\
    \text{id1} &= \text{id2} + t_1
\end{align*}\]

Journey inside a compiler

Optimized

\[\begin{align*}
    t_1 &= \text{id3} + 60.0 \\
    \text{id1} &= \text{id2} + t_1
\end{align*}\]

Assembly

\[\begin{align*}
    \text{LDF} & \, R2, \, \text{id3} \\
    \text{MULF} & \, R2, \, R2, \, 60.0 \\
    \text{LDF} & \, R1, \, \text{id2} \\
    \text{ADDF} & \, R1, \, R1, \, R2 \\
    \text{STF} & \, \text{id1}, \, R1
\end{align*}\]

You Have Reached Your Destination