Principles of Program Analysis

Lecture 1
Harry Xu
Spring 2013
An Imperfect World

• Software has bugs
  – The northeast blackout of 2003, affected 10 million people in Ontario and 45 million in eight U.S. states (caused by a race condition)
  – The explosion of the Ariane 5, valued at $500 million, 45 seconds after its lift-off (due to an 16-bit integer overflow)

• Software is slow
  – the conversion of a single date field from a SOAP data source to a Java object can require as many as 268 method calls and the generation of 70 objects
Program Analysis

• Discovering facts about programs
• A wide variety of applications
  – Finding bugs (e.g., model checking, testing, etc.)
  – Optimizing performance (e.g., compiler optimizations, bloat detection, etc.)
  – Detecting security vulnerabilities (e.g., detecting violations of security policies, etc.)
  – Improving software maintainability and understandability (e.g., reverse-engineering of UML diagrams, software visualization, etc.)
Static v.s. Dynamic Analysis

• Static analysis
  – Attempt to understand certain program properties without running a program
  – Make over-conservative claims

• Dynamic analysis
  – Need to run user *instrumented* code
  – Add overhead to running time and memory consumption
This Class

- Focus on *static program analysis* in this class
- We will discuss
  - Both principles and practices
  - Both classical program analysis algorithms and the state-of-the-art research
- We will cover five major topics
  - Dataflow analysis
  - Abstract interpretation
  - Constraint-based analysis
  - Type and effect system
  - Scalable interprocedural analysis
This Class

• We will spend two weeks on each topic
  – Discuss analysis principles in the first week (via lectures)
  – Discuss state-or-the-art research in the second week (via student presentations)

• Homework for each topic
  – A project that implements program analysis algorithms in Java
  – Paper critiques

• Students volunteer to present papers
  – 15 slots
  – Bonus credits!
Projects

• Two students form a group
• Based on the soot program analysis framework (http://www.sable.mcgill.ca/soot/)
• The first project
  – Implement a “hello-world” version of an intra-procedural analysis that prints out all heap load/store operations
  – Due Friday April 10
Course Pre-Reqs and Grading

• Office hour: Thursday 2—4pm, DBH 3212
• Reader: Taesu Kim
• Prerequisites: Java programming experience
• Grading
  – Paper critiques (20%)
  – Projects (40%)
  – In-class final (40%)
Static Analysis

- Key property: safe approximation
  - A larger set of possibilities than what will ever happen during any execution of the program
A Simple Example

read(x);
if(x>0) y = 1;
else {y = 2; S}; //S does not write y
z = y;
A Simple Example

read(x);
if(x>0) y = 1;
else {y = 2;  //S does not write z
z = y;

• Which of the following statements about z are valid from the perspective of a static analysis?
A Simple Example

read(x);
if(x>0) y = 1;
else {y = 2; S};  //S does not write z
z = y;

• Which of the following statements about z are valid from the perspective of a static analysis?
  – The value of z is 1
A Simple Example

read(x);
if(x>0) y = 1;
else {y = 2; S}; //S does not write z
z = y;

- Which of the following statements about z are valid from the perspective of a static analysis?
  - The value of z is 1  
    ✗
A Simple Example

read(x);
if(x>0) y = 1;
else {y = 2; S};  //S does not write z
z = y;

• Which of the following statements about z are valid from the perspective of a static analysis?
  – The value of z is 1  ❌
  – The value of z is 2
A Simple Example

read(x);
if(x>0) y = 1;
else {y = 2; S};   //S does not write z
z = y;

• Which of the following statements about z are valid from the perspective of a static analysis?
  – The value of z is 1
  – The value of z is 2
A Simple Example

read(x);
if(x>0) y = 1;
else {y = 2; S};   //S does not write z
z = y;

• Which of the following statements about z are valid from the perspective of a static analysis?
  – The value of z is 1  
  – The value of z is 2  
  – The value of z is in the set \{1, 2\}
A Simple Example

read(x);
if(x>0) y = 1;
else {y = 2; S};  //S does not write z
z = y;

• Which of the following statements about z are valid from the perspective of a static analysis?
  – The value of z is 1  ❌
  – The value of z is 2  ❌
  – The value of z is in the set {1, 2}  √
A Simple Example

```cpp
read(x);
if(x>0) y = 1;
else {y = 2; S}; //S does not write z
z = y;
```

• Which of the following statements about z are valid from the perspective of a static analysis?

  – The value of z is 1
  – The value of z is 2
  – The value of z is in the set \{1, 2\}
  – The value of z is in the set \{1, 2, 34, 128\}
A Simple Example

read(x);
if(x>0) y = 1;
else {y = 2; S};  //S does not write z
z = y;

• Which of the following statements about z are valid from the perspective of a static analysis?
  – The value of z is 1
  – The value of z is 2
  – The value of z is in the set {1, 2}
  – The value of z is in the set {1, 2, 34, 128}
A Simple Example

```c
read(x);
if(x>0) y = 1;
else {y = 2; S};  //S does not write z
z = y;
```

- Which of the following statements about $z$ are valid from the perspective of a static analysis?
  - The value of $z$ is 1
    - ✗
  - The value of $z$ is 2
    - ✗
  - The value of $z$ is in the set $\{1, 2\}$
    - ✓
  - The value of $z$ is in the set $\{1, 2, 34, 128\}$
    - ✓
  - The value of $z$ depends on the value of $x$; when $x > 0$, $z$ is 1; otherwise $z$ is 2
    - ✗
A Simple Example

read(x);
if(x>0) y = 1;
else {y = 2; S};  //S does not write z
z = y;

• Which of the following statements about z are valid from the perspective of a static analysis?
  – The value of z is 1
  – The value of z is 2
  – The value of z is in the set {1, 2}
  – The value of z is in the set {1, 2, 34, 128}
  – The value of z depends on the value of x; when x > 0, z is 1; otherwise z is 2

The Nature of Approximations

The exact world

Over-approximation

Under-approximation

Slogans: Err on the safe side!
Trade precision for efficiency!
Setting the Stage

• Formalism
  – A simple imperative language
  – Operational semantics
  – Lattice theory
  – Fixedpoint computation

• A simple reaching-definition analysis used throughout the quarter
A while Language

\[
\begin{align*}
 a & \in \text{AEExp} \quad \text{arithmetic expressions} \\
 b & \in \text{BEExp} \quad \text{boolean expressions} \\
 S & \in \text{ Stmt} \quad \text{statements} \\
 x, y & \in \text{Var} \quad \text{variables} \\
 n & \in \text{Num} \quad \text{numerals} \\
 \ell & \in \text{Lab} \quad \text{labels} \\
op_a & \in \text{Op}_a \quad \text{arithmetic operators} \\
op_b & \in \text{Op}_b \quad \text{boolean operators} \\
op_r & \in \text{Op}_r \quad \text{relational operators}
\end{align*}
\]

\[
\begin{align*}
 a & ::= x \mid n \mid a_1 \, op_a \, a_2 \\
 b & ::= \text{true} \mid \text{false} \mid \text{not} \, b \mid b_1 \, op_b \, b_2 \mid a_1 \, op_r \, a_2 \\
 S & ::= [x := a]^{\ell} \mid [\text{skip}]^{\ell} \mid S_1 ; S_2 \mid \\
 & \quad \text{if} [b]^{\ell} \text{ then } S_1 \text{ else } S_2 \mid \text{while} [b]^{\ell} \text{ do } S
\end{align*}
\]
An Example Program

\[ y := x \]
\[ z := 1 \]

while \[ y > 1 \]
do
  \[ z := z \times y \]
  \[ y := y - 1 \]

\[ y := 0 \]

Computes the factorial of the number in \( x \) and leaves the result in \( z \)
Formal Semantics

• Why useful
  – Formally define what a program does exactly
  – Prove the correctness of an language implementation or a program analysis

• Three major kinds of semantics
  – Denotational semantics
  – Operational semantics
  – Axiomatic semantics
Denotational Semantics

• Concerned about the conceptual meaning of a program
• Each phrase is interpreted as a denotation
• The meaning of a program reduces to the meaning of the sequence of commands
An Denotational Semantics Example

Syntactic Domains

N : Numeral  -- nonnegative numerals
D : Digit    -- decimal digits

Abstract Production Rules

Numeral ::= Digit | Numeral Digit
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Semantic Domain

Number = \{0, 1, 2, 3, 4, ...\}  -- natural numbers

Semantic Functions

value : Numeral \rightarrow Number
digit : Digit \rightarrow Number

Semantic Equations

\[ value [N D] = plus (times(10, value [N]), digit [D]) \]
\[ value [D] = digit [D] \]
\[ digit [0] = 0 \quad digit [3] = 3 \quad digit [6] = 6 \quad digit [8] = 8 \]
\[ digit [2] = 2 \quad digit [5] = 5 \]
Denotational Semantics

\[
\text{value } [1023] = \text{plus}(\text{times}(10, \text{value}[102]), \text{digit}[3]) \\
= \text{plus}(\text{times}(10, \text{plus}(\text{times}(10, \\
\text{value}[10], \text{digit}[2]))), \text{digit}[3]) \\
= \text{plus}(\text{times}(10, \text{plus}(\text{times}(10, \\
\text{plus}(\text{times}(10, \text{plus}(\text{times}(10, \text{digit}[1]), \text{digit}[0]))), \\
\text{digit}[2]))), \text{digit}[3]) \\
= 1023
\]

Two language constructs are semantically equivalent if they share the same denotation.
Axiomatic Semantics

• Based on mathematical logic (e.g., Hoare logic)
  – Used to reason about the correctness of a program

• Hoare triple
  – \{P\} C \{Q\}
  – P and Q are assertions (i.e., formulae in predicate logic) and C is a command
  – P is the precondition and Q is the postcondition
  – When P is met, C establishes Q

• Example: \{x + 1 = 43\} y:= x+1 \{y = 43\}
Operational Semantics

• The execution of a program is described directly

• Structural (small-step) operational semantics
  – Formally define how the individual steps of a computation take place

• Big-step operational semantics
  – How the overall results of an execution are obtained
Operational Semantics

• More commonly used in formally reasoning about a program analysis algorithm
  – The algorithm is sound if it appropriately abstracts the concrete operational semantics of the program
Operational Semantics

A state is a mapping from variables to integers:

$$\sigma \in \text{State} = \text{Var} \rightarrow \mathbb{Z}$$

The semantics of arithmetic and boolean expressions

$$A : \text{AExp} \rightarrow (\text{State} \rightarrow \mathbb{Z}) \quad (\text{no errors allowed})$$

$$B : \text{BExp} \rightarrow (\text{State} \rightarrow \text{T}) \quad (\text{no errors allowed})$$

The transitions of the semantics are of the form

$$\langle S, \sigma \rangle \rightarrow \sigma' \quad \text{and} \quad \langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$$
Transitions

\[\langle [x := a]^l, \sigma \rangle \rightarrow \sigma[x \mapsto A[a] \sigma]\]
\[\langle [\text{skip}]^l, \sigma \rangle \rightarrow \sigma\]

\[\frac{\langle S_1, \sigma \rangle \rightarrow \langle S'_1, \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S'_1; S_2, \sigma' \rangle}\]

\[\frac{\langle S_1, \sigma \rangle \rightarrow \sigma'}{\langle S_1; S_2, \sigma \rangle \rightarrow \langle S_2, \sigma' \rangle}\]

\[\langle \text{if } [b]^l \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle \quad \text{if } B[b] \sigma = \text{true}\]
\[\langle \text{if } [b]^l \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle \quad \text{if } B[b] \sigma = \text{false}\]

\[\langle \text{while } [b]^l \text{ do } S, \sigma \rangle \rightarrow \langle (S; \text{while } [b]^l \text{ do } S), \sigma \rangle \quad \text{if } B[b] \sigma = \text{true}\]
\[\langle \text{while } [b]^l \text{ do } S, \sigma \rangle \rightarrow \sigma \quad \text{if } B[b] \sigma = \text{false}\]
Example Derivation Sequence

\[
\langle [y := x]^1; [z := 1]^2; \text{while } [y > 1]^3 \text{ do } ([z := z \cdot y]^4; [y := y - 1]^5); [y := 0]^6, \sigma_{300} \rangle \\
\quad \rightarrow \langle [z := 1]^2; \text{while } [y > 1]^3 \text{ do } ([z := z \cdot y]^4; [y := y - 1]^5); [y := 0]^6, \sigma_{330} \rangle \\
\quad \rightarrow \langle \text{while } [y > 1]^3 \text{ do } ([z := z \cdot y]^4; [y := y - 1]^5); [y := 0]^6, \sigma_{331} \rangle \\
\quad \rightarrow \langle [z := z \cdot y]^4; [y := y - 1]^5; \\
\quad \quad \quad \text{while } [y > 1]^3 \text{ do } ([z := z \cdot y]^4; [y := y - 1]^5); [y := 0]^6, \sigma_{331} \rangle \\
\quad \rightarrow \langle [y := y - 1]^5; \text{while } [y > 1]^3 \text{ do } ([z := z \cdot y]^4; [y := y - 1]^5); [y := 0]^6, \sigma_{333} \rangle \\
\quad \rightarrow \langle \text{while } [y > 1]^3 \text{ do } ([z := z \cdot y]^4; [y := y - 1]^5); [y := 0]^6, \sigma_{323} \rangle \\
\quad \rightarrow \langle [z := z \cdot y]^4; [y := y - 1]^5; \\
\quad \quad \quad \text{while } [y > 1]^3 \text{ do } ([z := z \cdot y]^4; [y := y - 1]^5); [y := 0]^6, \sigma_{323} \rangle \\
\quad \rightarrow \langle [y := y - 1]^5; \text{while } [y > 1]^3 \text{ do } ([z := z \cdot y]^4; [y := y - 1]^5); [y := 0]^6, \sigma_{326} \rangle \\
\quad \rightarrow \langle \text{while } [y > 1]^3 \text{ do } ([z := z \cdot y]^4; [y := y - 1]^5); [y := 0]^6, \sigma_{316} \rangle \\
\quad \rightarrow \langle [y := 0]^6, \sigma_{316} \rangle \\
\rightarrow \sigma_{306}
\]
Lattice Theory

• A lattice is a partially ordered set \((L, \leq)\)
• Any two elements have a supremum (i.e., least upper bound) and an infimum (i.e., greatest lower bound)
• For any two elements \(a\) and \(b\) in \(L\), \(a\) and \(b\) have a join: \(a \lor b\) (supremum)
• For any two elements \(a\) and \(b\) in \(L\), \(a\) and \(b\) have a meet: \(a \land b\) (infimum)
An Example Lattice

• A lattice of partitions of a four-element set \{1, 2, 3, 4\}
• Ordered by the relation “is refinement of”
• \( a \lor b = \) a coarser-grained partition than both \( a \) and \( b \)
• \( a \land b = \) a finer-grained partition than both \( a \) and \( b \)
General Properties

• Commutative laws
  – a \& b = b \& a \quad a \lor b = b \lor a

• Associative laws
  – a \lor (b \lor c) = (a \lor b) \lor c \quad a \&(b \& c) = (a \& b) \& c

• Absorption laws
  – a \lor (a \& b) = a \quad a \& (a \lor b) = a

• Idempotent laws
  – a \lor a = a \quad a \& a = a
More about Lattice

- The least element $\bot$ (i.e., unknown) and the greatest element $\top$ (i.e., everything)
  - $\top \land a = a$  $\top \lor a = \top$
  - $\bot \land a = \bot$  $\bot \lor a = a$

- Semi-lattice
  - A join-semi-lattice only has a join for any non-empty finite subset
  - A meet-semi-lattice only has a meet for any non-empty finite subset

- Real-world examples
  - Types in Java
Fixedpoint Computation

A fixedpoint equation has the form
\[ f(x) = x \]

Its solutions are called the fixed points of \( f \) because if \( x_p \) is a solution then
\[
  x_p = f(x_p) = f(f(x_p)) = f(f(f(x_p))) = \ldots
\]

In program analysis, we look for both such \( x_p \) and function \( f \) that can eventually reach a fixedpoint
Tarski’s Fixedpoint Theorem

Let \( L = (L, \sqsubseteq) \) be a complete lattice and let \( f : L \rightarrow L \) be a monotone function.

The greatest fixed point \( \text{gfp}(f) \) satisfy:

\[
\text{gfp}(f) = \bigcup \{ l \mid l \sqsubseteq f(l) \} \in \{ l \mid f(l) = l \}
\]

The least fixed point \( \text{lfp}(f) \) satisfy:

\[
\text{lfp}(f) = \bigcap \{ l \mid f(l) \sqsubseteq l \} \in \{ l \mid f(l) = l \}
\]
Dataflow Analysis

Harry Xu
CS 253/INF 212
Spring 2013
Acknowledgements

Many slides in this file were taken from the chapter 2 slides available at

We thank the authors of the book *Principles of Program Analysis* for providing their slides.
Dataflow analysis

• A class of static analyses that aim to understand how data flows in the program

• Typical examples
  – Available expression analysis
  – Reaching definition analysis
  – Live variable analysis
  – Constant propagation
Analysis Scope

• Intraprocedural analysis
  – Focusing on each individual function
  – Do not track dataflow across function boundary

• Interprocedural analysis
  – Analyze the whole program
  – Way more expensive
Control flow graph

**Example:** \( z := 1 \); while \( x > 0 \) do \( (z := z \times y); x := x - 1 \)

- \( init(\cdots) = 1 \)
- \( final(\cdots) = \{2\} \)
- \( labels(\cdots) = \{1, 2, 3, 4\} \)
- \( flow(\cdots) = \{(1, 2), (2, 3), (3, 4), (4, 2)\} \)
- \( flow^R(\cdots) = \{(2, 1), (2, 4), (3, 2), (4, 3)\} \)
Intraprocedural Dataflow Analyses

• Classical analyses
  – Available expression analysis
  – Reaching definition analysis
  – Live variable analysis
Available Expression Analysis

The aim of the *Available Expressions Analysis* is to determine

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

**Example:**

```
[x:= a+b]¹; [y:=a*b]²; while [y> a+b]³ do ([a:=a+1]⁴; [x:= a+b]⁵)
```

The analysis enables a transformation into

```
[x:= a+b]¹; [y:=a*b]²; while [y>x]³ do ([a:=a+1]⁴; [x:= a+b]⁵)
```
Basic Idea

\[ N = X_1 \cap X_2 \]

\[ x := a \]

\[ X = (N \setminus \{\text{expressions with an } x\}) \cup \{\text{subexpressions of } a \text{ without an } x\} \]
Analysis Algorithm

\textit{kill} and \textit{gen} functions

\begin{align*}
\text{kill}_{AE}([x \leftarrow a]^\ell) &= \{a' \in \text{AExp}_\star \mid x \in \text{FV}(a')\} \\
\text{kill}_{AE}([\text{skip}]^\ell) &= \emptyset \\
\text{kill}_{AE}([b]^\ell) &= \emptyset \\
\text{gen}_{AE}([x \leftarrow a]^\ell) &= \{a' \in \text{AExp}(a) \mid x \notin \text{FV}(a')\} \\
\text{gen}_{AE}([\text{skip}]^\ell) &= \emptyset \\
\text{gen}_{AE}([b]^\ell) &= \text{AExp}(b)
\end{align*}

data flow equations: \text{AE}^= 

\begin{align*}
\text{AE}_{\text{entry}}(\ell) &= \begin{cases} 
\emptyset & \text{if } \ell = \text{init}(S_\star) \\
\emptyset \cap \{\text{AE}_{\text{exit}}(\ell') \mid (\ell', \ell) \in \text{flow}(S_\star)\} & \text{otherwise}
\end{cases} \\
\text{AE}_{\text{exit}}(\ell) &= (\text{AE}_{\text{entry}}(\ell) \setminus \text{kill}_{AE}(B^\ell)) \cup \text{gen}_{AE}(B^\ell) \\
&\text{where } B^\ell \in \text{blocks}(S_\star)
\end{align*}
Analysis Example

\[ x := a + b \]
\[ y := a \times b \]
\[ \text{while } y > a + b \text{ do } \]
\[ a := a + 1 \]
\[ x := a + b \]

*kill* and *gen* functions:

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{kill}_{AE}(\ell) )</th>
<th>( \text{gen}_{AE}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \emptyset )</td>
<td>( {a + b} )</td>
</tr>
<tr>
<td>2</td>
<td>( \emptyset )</td>
<td>( {a \times b} )</td>
</tr>
<tr>
<td>3</td>
<td>( \emptyset )</td>
<td>( {a + b} )</td>
</tr>
<tr>
<td>4</td>
<td>( {a + b, a \times b, a + 1} )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>5</td>
<td>( \emptyset )</td>
<td>( {a + b} )</td>
</tr>
</tbody>
</table>
Example (Cond)

\[ x := a + b \] \(^1\); \[ y := a \times b \] \(^2\); while \( y > a + b \) \(^3\) do (\[ a := a + 1 \] \(^4\); \[ x := a + b \] \(^5\))

Equations:

\[
\begin{align*}
\text{AE}_{\text{entry}}(1) &= \emptyset \\
\text{AE}_{\text{entry}}(2) &= \text{AE}_{\text{exit}}(1) \\
\text{AE}_{\text{entry}}(3) &= \text{AE}_{\text{exit}}(2) \cap \text{AE}_{\text{exit}}(5) \\
\text{AE}_{\text{entry}}(4) &= \text{AE}_{\text{exit}}(3) \\
\text{AE}_{\text{entry}}(5) &= \text{AE}_{\text{exit}}(4)
\end{align*}
\]

\[
\begin{align*}
\text{AE}_{\text{exit}}(1) &= \text{AE}_{\text{entry}}(1) \cup \{a+b\} \\
\text{AE}_{\text{exit}}(2) &= \text{AE}_{\text{entry}}(2) \cup \{a \times b\} \\
\text{AE}_{\text{exit}}(3) &= \text{AE}_{\text{entry}}(3) \cup \{a+b\} \\
\text{AE}_{\text{exit}}(4) &= \text{AE}_{\text{entry}}(4) \setminus \{a+b, a \times b, a+1\} \\
\text{AE}_{\text{exit}}(5) &= \text{AE}_{\text{entry}}(5) \cup \{a+b\}
\end{align*}
\]
Example (Cond)

\[ x := a + b \] \(^1\); \[ y := a \times b \] \(^2\); while \[ y > a + b \] \(^3\) do (\[ a := a + 1 \] \(^4\); \[ x := a + b \] \(^5\))

Largest solution:

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{AE}_{entry}(\ell) )</th>
<th>( \text{AE}_{exit}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \emptyset )</td>
<td>( { a + b } )</td>
</tr>
<tr>
<td>2</td>
<td>( { a + b } )</td>
<td>( { a + b, a \times b } )</td>
</tr>
<tr>
<td>3</td>
<td>( { a + b } )</td>
<td>( { a + b } )</td>
</tr>
<tr>
<td>4</td>
<td>( { a + b } )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>5</td>
<td>( \emptyset )</td>
<td>( { a + b } )</td>
</tr>
</tbody>
</table>
Reaching Definition Analysis

The aim of the Reaching Definitions Analysis is to determine

For each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.

Example:  

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

useful for definition-use chains and use-definition chains
Basic Idea

\[ N = X_1 \cup X_2 \]

\[ [x := \alpha]^l \]

\[ X = (N \setminus \{(x, ?), (x, 1), \ldots\}) \]

\[ \cup \{(x, \ell)\} \]

\[ \text{kill} \]

\[ \text{gen} \]
Analysis Algorithm

$\textit{kill}$ and $\textit{gen}$ functions

$$
\begin{align*}
\text{kill}_{RD}([x := a]^\ell) & = \{(x, ?)\} \\
& \cup \{(x, \ell') \mid B^{\ell'} \text{ is an assignment to } x \text{ in } S_*\} \\
\text{kill}_{RD}([\text{skip}]^\ell) & = \emptyset \\
\text{kill}_{RD}([b]^\ell) & = \emptyset \\
\text{gen}_{RD}([x := a]^\ell) & = \{(x, \ell)\} \\
\text{gen}_{RD}([\text{skip}]^\ell) & = \emptyset \\
\text{gen}_{RD}([b]^\ell) & = \emptyset
\end{align*}
$$

Data flow equations: $RD^=$

$$
\begin{align*}
RD_{entry}(\ell) & = \begin{cases} \\
\{ (x, ?) \mid x \in FV(S_*) \} \\
\cup RD_{exit}(\ell') \mid (\ell', \ell) \in \text{flow}(S_*) \end{cases} & \text{if } \ell = \text{init}(S_*) \\
\text{ otherwise}
\end{align*}
$$

$$
RD_{exit}(\ell) = (RD_{entry}(\ell) \setminus \text{kill}_{RD}(B^\ell)) \cup \text{gen}_{RD}(B^\ell)
$$

where $B^\ell \in blocks(S_*)$
Analysis Example

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

*kill* and *gen* functions:

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{kill}_{RD}(\ell) )</th>
<th>( \text{gen}_{RD}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{ (x, ?), (x, 1), (x, 5) }</td>
<td>{ (x, 1) }</td>
</tr>
<tr>
<td>2</td>
<td>{ (y, ?), (y, 2), (y, 4) }</td>
<td>{ (y, 2) }</td>
</tr>
<tr>
<td>3</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>4</td>
<td>{ (y, ?), (y, 2), (y, 4) }</td>
<td>{ (y, 4) }</td>
</tr>
<tr>
<td>5</td>
<td>{ (x, ?), (x, 1), (x, 5) }</td>
<td>{ (x, 5) }</td>
</tr>
</tbody>
</table>
Example (Cond)

\[ x:=5 \] \(^1\); \[ y:=1 \] \(^2\); while \[ x>1 \] \(^3\) do \([ y:=x*y ] \) \(^4\); \[ x:=x-1 \] \(^5\)

Equations:

\[ \text{RD}_{\text{entry}}(1) = \{(x, ?), (y, ?)\} \]
\[ \text{RD}_{\text{entry}}(2) = \text{RD}_{\text{exit}}(1) \]
\[ \text{RD}_{\text{entry}}(3) = \text{RD}_{\text{exit}}(2) \cup \text{RD}_{\text{exit}}(5) \]
\[ \text{RD}_{\text{entry}}(4) = \text{RD}_{\text{exit}}(3) \]
\[ \text{RD}_{\text{entry}}(5) = \text{RD}_{\text{exit}}(4) \]

\[ \text{RD}_{\text{exit}}(1) = (\text{RD}_{\text{entry}}(1) \setminus \{(x, ?), (x, 1), (x, 5)\}) \cup \{(x, 1)\} \]
\[ \text{RD}_{\text{exit}}(2) = (\text{RD}_{\text{entry}}(2) \setminus \{(y, ?), (y, 2), (y, 4)\}) \cup \{(y, 2)\} \]
\[ \text{RD}_{\text{exit}}(3) = \text{RD}_{\text{entry}}(3) \]
\[ \text{RD}_{\text{exit}}(4) = (\text{RD}_{\text{entry}}(4) \setminus \{(y, ?), (y, 2), (y, 4)\}) \cup \{(y, 4)\} \]
\[ \text{RD}_{\text{exit}}(5) = (\text{RD}_{\text{entry}}(5) \setminus \{(x, ?), (x, 1), (x, 5)\}) \cup \{(x, 5)\} \]
Example (Cond)

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

Smallest solution:

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{RD}_{\text{entry}}(\ell) )</th>
<th>( \text{RD}_{\text{exit}}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{(x, ?), (y, ?)}</td>
<td>{(y, ?), (x, 1)}</td>
</tr>
<tr>
<td>2</td>
<td>{(y, ?), (x, 1)}</td>
<td>{(x, 1), (y, 2)}</td>
</tr>
<tr>
<td>3</td>
<td>{(x, 1), (y, 2), (y, 4), (x, 5)}</td>
<td>{(x, 1), (y, 2), (y, 4), (x, 5)}</td>
</tr>
<tr>
<td>4</td>
<td>{(x, 1), (y, 2), (y, 4), (x, 5)}</td>
<td>{(x, 1), (y, 4), (x, 5)}</td>
</tr>
<tr>
<td>5</td>
<td>{(x, 1), (y, 4), (x, 5)}</td>
<td>{(y, 4), (x, 5)}</td>
</tr>
</tbody>
</table>
Live Variable Analysis

A variable is *live* at the exit from a label if there is a path from the label to a use of the variable that does not re-define the variable.

The aim of the *Live Variables Analysis* is to determine for each program point, which variables may be live at the exit from the point.

Example:

```
point of interest

```

The analysis enables a transformation into

```
```
Basic Idea

\[ N = (X \setminus \{x\}) \cup \{\text{all variables of } a\} \]

\[ x := a \]

\[ X = N_1 \cup N_2 \]
Analysis Algorithm

**kill and gen functions**

\[
\begin{align*}
\text{kill}_{\text{LV}}([x := a]^\ell) &= \{x\} \\
\text{kill}_{\text{LV}}([\text{skip}]^\ell) &= \emptyset \\
\text{kill}_{\text{LV}}([b]^\ell) &= \emptyset \\
\text{gen}_{\text{LV}}([x := a]^\ell) &= \text{FV}(a) \\
\text{gen}_{\text{LV}}([\text{skip}]^\ell) &= \emptyset \\
\text{gen}_{\text{LV}}([b]^\ell) &= \text{FV}(b)
\end{align*}
\]

**data flow equations:**

\[
\begin{align*}
\text{LV}_{\text{exit}}(\ell) &= \begin{cases} \\
\emptyset & \text{if } \ell \in \text{final}(S_*), \\
\bigcup \{\text{LV}_{\text{entry}}(\ell') | (\ell', \ell) \in \text{flow}^R(S_*)\} & \text{otherwise}
\end{cases} \\
\text{LV}_{\text{entry}}(\ell) &= (\text{LV}_{\text{exit}}(\ell) \setminus \text{kill}_{\text{LV}}(B^\ell)) \cup \text{gen}_{\text{LV}}(B^\ell) \\
&\text{where } B^\ell \in \text{blocks}(S_*)
\end{align*}
\]
Example

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

*kill* and *gen* functions:

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{kill}_{LV}(\ell) )</th>
<th>( \text{gen}_{LV}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>2</td>
<td>{y}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>3</td>
<td>{x}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>4</td>
<td>\emptyset</td>
<td>{x, y}</td>
</tr>
<tr>
<td>5</td>
<td>{z}</td>
<td>{y}</td>
</tr>
<tr>
<td>6</td>
<td>{z}</td>
<td>{y}</td>
</tr>
<tr>
<td>7</td>
<td>{x}</td>
<td>{z}</td>
</tr>
</tbody>
</table>
Example (Cond)

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

Equations:

\[
\begin{align*}
\text{LV}_{\text{entry}}(1) &= \text{LV}_{\text{exit}}(1) \setminus \{x\} & \text{LV}_{\text{exit}}(1) &= \text{LV}_{\text{entry}}(2) \\
\text{LV}_{\text{entry}}(2) &= \text{LV}_{\text{exit}}(2) \setminus \{y\} & \text{LV}_{\text{exit}}(2) &= \text{LV}_{\text{entry}}(3) \\
\text{LV}_{\text{entry}}(3) &= \text{LV}_{\text{exit}}(3) \setminus \{x\} & \text{LV}_{\text{exit}}(3) &= \text{LV}_{\text{entry}}(4) \\
\text{LV}_{\text{entry}}(4) &= \text{LV}_{\text{exit}}(4) \cup \{x, y\} & \text{LV}_{\text{exit}}(4) &= \text{LV}_{\text{entry}}(5) \cup \text{LV}_{\text{entry}}(6) \\
\text{LV}_{\text{entry}}(5) &= (\text{LV}_{\text{exit}}(5) \setminus \{z\}) \cup \{y\} & \text{LV}_{\text{exit}}(5) &= \text{LV}_{\text{entry}}(7) \\
\text{LV}_{\text{entry}}(6) &= (\text{LV}_{\text{exit}}(6) \setminus \{z\}) \cup \{y\} & \text{LV}_{\text{exit}}(6) &= \text{LV}_{\text{entry}}(7) \\
\text{LV}_{\text{entry}}(7) &= \{z\} & \text{LV}_{\text{exit}}(7) &= \emptyset
\end{align*}
\]
Example (Cond)

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

Smallest solution:

<table>
<thead>
<tr>
<th>(\ell)</th>
<th>(LV_{entry}(\ell))</th>
<th>(LV_{exit}(\ell))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>2</td>
<td>(\emptyset)</td>
<td>{(y)}</td>
</tr>
<tr>
<td>3</td>
<td>{(y)}</td>
<td>{(x, y)}</td>
</tr>
<tr>
<td>4</td>
<td>{(x, y)}</td>
<td>{(y)}</td>
</tr>
<tr>
<td>5</td>
<td>{(y)}</td>
<td>{(z)}</td>
</tr>
<tr>
<td>6</td>
<td>{(y)}</td>
<td>{(z)}</td>
</tr>
<tr>
<td>7</td>
<td>{(z)}</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
Extracting Similarities

A common pattern exists in these analyses

\[
\text{Analysis}_\circ(\ell) = \begin{cases} 
\bigcup \{ \text{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F \} & \text{if } \ell \in E \\
\text{otherwise}
\end{cases}
\]

\[
\text{Analysis}_\bullet(\ell) = f_\ell(\text{Analysis}_\circ(\ell))
\]

where

- $\bigcup$ is $\cap$ or $\cup$ (and $\bigcap$ is $\cup$ or $\cap$),
- $F$ is either $\text{flow}(S_\star)$ or $\text{flow}^R(S_\star)$,
- $E$ is $\{\text{init}(S_\star)\}$ or $\text{final}(S_\star)$,
- $\iota$ specifies the initial or final analysis information, and
- $f_\ell$ is the transfer function associated with $B^\ell \in \text{blocks}(S_\star)$.
Forward v.s. Backward

- The *forward analyses* have $F$ to be $\text{flow}(S_\star)$ and then $\text{Analysis}_\circ$ concerns entry conditions and $\text{Analysis}_\bullet$ concerns exit conditions; the equation system presupposes that $S_\star$ has isolated entries.

- The *backward analyses* have $F$ to be $\text{flow}^R(S_\star)$ and then $\text{Analysis}_\circ$ concerns exit conditions and $\text{Analysis}_\bullet$ concerns entry conditions; the equation system presupposes that $S_\star$ has isolated exits.
Union or Intersection

- When $\bigcap$ is $\bigcup$ we require the **greatest sets** that solve the equations and we are able to detect properties satisfied by *all execution paths* reaching (or leaving) the entry (or exit) of a label; the analysis is called a **must**-analysis.

- When $\bigcup$ is $\bigcap$ we require the **smallest sets** that solve the equations and we are able to detect properties satisfied by *at least one execution path* to (or from) the entry (or exit) of a label; the analysis is called a **may**-analysis.
Property Space

$L$ is a complete lattice used to represent the data flow information (data flow facts)

$L$ is the combination operation: $P(L) \rightarrow L$, used to Combine information from different paths
Transfer Function

The set of transfer functions, $\mathcal{F}$, is a set of monotone functions over $L$, meaning that

$$l \subseteq l' \implies f_\ell(l) \subseteq f_\ell(l')$$

and furthermore they fulfil the following conditions:

- $\mathcal{F}$ contains all the transfer functions $f_\ell : L \to L$ in question (for $\ell \in \text{Lab}_\ast$)

- $\mathcal{F}$ contains the identity function

- $\mathcal{F}$ is closed under composition of functions
Frameworks

A Monotone Framework consists of:

- a complete lattice, $L$, that satisfies the Ascending Chain Condition; we write $\sqcup$ for the least upper bound operator

- a set $F$ of monotone functions from $L$ to $L$ that contains the identity function and that is closed under function composition

A Distributive Framework is a Monotone Framework where all functions $f$ in $F$ are required to be distributive:

$$f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$$
Framework Instances

An *instance* of a Framework consists of:

- the complete lattice, $L$, of the framework
- the space of functions, $\mathcal{F}$, of the framework
- a finite flow, $F$ (typically $\text{flow}(S_*)$ or $\text{flow}^R(S_*)$)
- a finite set of *extremal labels*, $E$ (typically $\{\text{init}(S_*)\}$ or $\text{final}(S_*)$
- an *extremal value*, $i \in L$, for the extremal labels
- a mapping, $f$, from the labels $\text{Lab}_*$ to transfer functions in $\mathcal{F}$
Equations and Constraints

Equations of the Instance:

\[
\text{Analysis}_o(\ell) = \bigcup \{ \text{Analysis}_i(\ell') \mid (\ell', \ell) \in F \} \cup i_E^\ell
\]

where \( i_E^\ell = \begin{cases} 
\top & \text{if } \ell \in E \\
\bot & \text{if } \ell \notin E
\end{cases} \)

\[
\text{Analysis}_i(\ell) = f_\ell(\text{Analysis}_o(\ell))
\]

Constraints of the Instance:

\[
\text{Analysis}_o(\ell) \supseteq \bigcup \{ \text{Analysis}_i(\ell') \mid (\ell', \ell) \in F \} \cup i_E^\ell
\]

where \( i_E^\ell = \begin{cases} 
\top & \text{if } \ell \in E \\
\bot & \text{if } \ell \notin E
\end{cases} \)

\[
\text{Analysis}_i(\ell) \supseteq f_\ell(\text{Analysis}_o(\ell))
\]
# Examples Revisited

<table>
<thead>
<tr>
<th></th>
<th>Available Expressions</th>
<th>Reaching Definitions</th>
<th>Very Busy Expressions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>$\mathcal{P}(\text{AExp}_*)$</td>
<td>$\mathcal{P}(\text{Var}<em>* \times \text{Lab}</em>*)$</td>
<td>$\mathcal{P}(\text{AExp}_*)$</td>
<td>$\mathcal{P}(\text{Var}_*)$</td>
</tr>
<tr>
<td>$\sqsubseteq$</td>
<td>$\sqsubseteq$</td>
<td>$\sqsubseteq$</td>
<td>$\sqsubseteq$</td>
<td>$\sqsubseteq$</td>
</tr>
<tr>
<td>$\sqcup$</td>
<td>$\cap$</td>
<td>$\cap$</td>
<td>$\cap$</td>
<td>$\cap$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\text{AExp}_*$</td>
<td>$\emptyset$</td>
<td>$\text{AExp}_*$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\ell$</td>
<td>$\emptyset$</td>
<td>${x, ?,</td>
<td>x \in \text{FV}(S_*)}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$E$</td>
<td>${\text{init}(S_*)}$</td>
<td>${\text{init}(S_*)}$</td>
<td>$\text{final}(S_*)$</td>
<td>$\text{final}(S_*)$</td>
</tr>
<tr>
<td>$F$</td>
<td>$\text{flow}(S_*)$</td>
<td>$\text{flow}(S_*)$</td>
<td>$\text{flow}^R(S_*)$</td>
<td>$\text{flow}^R(S_*)$</td>
</tr>
<tr>
<td>$\mathcal{F}$</td>
<td>${f : L \rightarrow L \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_\ell$</td>
<td>$f_\ell(l) = (l \setminus \text{kill}(B^\ell)) \cup \text{gen}(B^\ell)$ where $B^\ell \in \text{blocks}(S_*)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bit-Vector Frameworks

A *Bit Vector Framework* has

- $L = \mathcal{P}(D)$ for $D$ finite
- $\mathcal{F} = \{ f \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g\}$

**Examples:**

- Available Expressions
- Live Variables
- Reaching Definitions
- Very Busy Expressions
Bit-Vector Frameworks are Monotone and Distributive

\[ f(l_1 \cup l_2) = \begin{cases} 
  f(l_1 \cup l_2) \\
  f(l_1 \cap l_2) \\
  ((l_1 \setminus l_k) \cup (l_2 \setminus l_k)) \cup l_g \\
  ((l_1 \setminus l_k) \cap (l_2 \setminus l_k)) \cup l_g 
\end{cases} \]

Monotonicity can be proved in a similar manner
Example: Constant Propagation

• Determine, for each program point, whether or not a variable has a constant value whenever execution reaches the point

Example:

\[ [x := 6]^1; [y := 3]^2; \text{while } [x > y]^3 \text{ do } ([x := x - 1]^4; [z := y \times y]^6) \]

The analysis enables a transformation into

\[ [x := 6]^1; [y := 3]^2; \text{while } [x > 3]^3 \text{ do } ([x := x - 1]^4; [z := 9]^6) \]
Now You Tell Me

• How to define a lattice $L$?

• How to define transfer functions?

• Is constant propagation a monotone framework?

• Is it a distributive framework?
Solving the Equation

- Many different approaches
- The least fixed-point solution
  - Always decidable
  - A worklist-based algorithm for monotone frameworks
Algorithm

• Idea: iterate until stabilization

Worklist Algorithm

Input: An instance \((L, \mathcal{F}, F, E, \nu, f.\) of a Monotone Framework

Output: The MFP Solution: \(MFP_0, MFP_1\)

Data structures:

• Analysis: the current analysis result for block entries (or exits)

• The worklist \(W\): a list of pairs \((\ell, \ell')\) indicating that the current analysis result has changed at the entry (or exit) to the block \(\ell\) and hence the entry (or exit) information must be recomputed for \(\ell'\)
Algorithm (Cond.)

Step 1  Initialisation (of \( W \) and Analysis)
\[
W := \text{nil};
\]
for all \((\ell, \ell')\) in \( F \) do \( W := \text{cons}((\ell, \ell'), W)\);
for all \( \ell \) in \( F \) or \( E \) do
  if \( \ell \in E \) then Analysis[\( \ell \)] := \( \tau \) else Analysis[\( \ell \)] := \( \bot_L \);

Step 2  Iteration (updating \( W \) and Analysis)
while \( W \neq \text{nil} \) do
  \( \ell := \text{fst}(\text{head}(W)); \ell' = \text{snd}(\text{head}(W)); W := \text{tail}(W); \)
  if \( f_\ell(\text{Analysis}[\ell]) \nsubseteq \text{Analysis}[\ell'] \) then
    Analysis[\( \ell' \)] := Analysis[\( \ell' \)] \( \sqcup f_\ell(\text{Analysis}[\ell]) \);
  for all \( \ell'' \) with \((\ell', \ell'')\) in \( F \) do \( W := \text{cons}((\ell', \ell''), W) \);

Step 3  Presenting the result (\( MFP_\circ \) and \( MFP_\bullet \))
for all \( \ell \) in \( F \) or \( E \) do
  \( MFP_\circ(\ell) := \text{Analysis}[\ell]; \)
  \( MFP_\bullet(\ell) := f_\ell(\text{Analysis}[\ell]) \)