Program Analysis

In computer science, program analysis is the process of automatically analysing the behavior of computer programs.
### Program Analysis

In computer science, program analysis is the process of automatically analysing the behavior of computer programs.

Two main approaches in program analysis are **static** program analysis and **dynamic** program analysis.

Main applications of program analysis are **program correctness** (including security) and **program optimization**.

### Basic Techniques

Techniques related to program analysis include:
- control-flow and data-flow analysis
- constraint-based analysis
- abstract interpretation
- type and effect systems

A technique that is applied for certain kinds of program analysis is **program slicing**.
Basic Techniques

Techniques related to program analysis include:
- control-flow and data-flow analysis
- constraint-based analysis
- abstract interpretation
- type and effect systems

A technique that is applied for certain kinds of program analysis is program slicing.

Related fields include performance analysis and program verification.

Data Flow Analysis

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program.

A program’s control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.
Data Flow Analysis

Definition

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program.

A program’s control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.

The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

Basic Techniques

Basic Approaches

A simple way to perform data-flow analysis of programs is to set up data-flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fixpoint.

The above general approach was developed by Gary Kildall while teaching at the Naval Postgraduate School.
An expression $e$ is available at program point $p$ if
- $e$ is computed on every path to $p$, and
- the value of $e$ has not changed since the last time $e$ was computed on every path to $p$

Optimization
- If an expression is available, need not be recomputed
  - (At least, if it is still in a register somewhere)

---

### Instrumentation Granularity

- Is expression $e$ available?
- Facts:
  - $a + b$ is available
  - $a * b$ is available
  - $a + 1$ is available

The following data flow examples are in courtesy of Prof. Mike Hicks

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := a + b$</td>
<td>$a + b$</td>
<td></td>
</tr>
<tr>
<td>$y := a * b$</td>
<td>$a * b$</td>
<td></td>
</tr>
<tr>
<td>$a := a + 1$</td>
<td>$a + 1, a * b$</td>
<td></td>
</tr>
<tr>
<td>$x := a + b$</td>
<td>$a + b$</td>
<td></td>
</tr>
</tbody>
</table>
Computing Available Expressions

entry

\( x := a + b \)

\( y := a \times b \)

\( y > a \)

\( a := a + 1 \)

\( x := a + b \)

exit

Tuesday, September 20, 2011
Computing Available Expressions

∅  
entry

{x := a + b}  

{a + b}  

{a + b, a * b}  

{a + b, a * b}  

y > a  

∅  

{x := a + b}  

∅  

{x := a + b}  

entry

∅  

{x := a + b}  

∅  

{x := a + b}  

entry

∅  

{x := a + b}  

∅  

{x := a + b}
Computing Available Expressions

Terminology

- A join point is a program point where two branches meet
- Available expressions is a forward must problem
  - Forward = Data flow from in to out
  - Must = Property must hold on all paths to the join
- Dataflow analysis requires facts that summarize all paths to a join point
  - Symbolic execution analyzes each path separately

Outline

1. Program Analysis
2. Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions
3. Theory Behind
4. Sensitivity
5. Summary

Data Flow Equations

- Let \( s \) be a statement
  - \( \text{succ}(s) = \{ \text{immediate successor statements of } s \} \)
  - \( \text{pred}(s) = \{ \text{immediate predecessor statements of } s\} \)
  - \( \text{In}(s) = \text{program point just before executing } s \)
  - \( \text{Out}(s) = \text{program point just after executing } s \)
- \( \text{In}(s) = \bigcap_{s' \in \text{pred}(s)} \text{Out}(s') \)
- \( \text{Out}(s) = \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s)) \)
A variable $v$ is live at program point $p$ if
- $v$ will be used on some execution path originating from $p$...
- before $v$ is overwritten

Optimization
- If a variable is not live, no need to keep it in a register
- If variable is dead at assignment, can eliminate assignment
Gen and Kill

- What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a + b</td>
<td>a, b</td>
<td>x</td>
</tr>
<tr>
<td>y := a * b</td>
<td>a, b</td>
<td>y</td>
</tr>
<tr>
<td>y &gt; a</td>
<td>a, y</td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

Computing Live Variables

- What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a + b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y := a * b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y &gt; a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x := a + b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computing Live Variables

- \{a, b\}
- \{x, a, b\}
- \{x, y, a, b\}
- \{y, a, b\}
- \{x, y, a, b\}

A definition of a variable \(v\) is an assignment to \(v\)
- A definition of variable \(v\) reaches point \(p\) if
  - There is no intervening assignment to \(v\)
  - Also called def-use information
- What kind of problem?
  - Forward or backward?
  - May or must?

Reaching Definitions

A definition of a variable \(v\) is an assignment to \(v\)
- A definition of variable \(v\) reaches point \(p\) if
  - There is no intervening assignment to \(v\)
- Also called def-use information
- What kind of problem?
  - Forward or backward?
  - May or must?
Reaching Definitions

\[ y := x \]
\[ z := x \]
\[ \text{while } y > 0 \text{ do}\]
\[ z := z \ast y \]
\[ y := y - 1 \]
\[ y := 0 \]

\[ \text{while } y > 0 \text{ do}\]
\[ z := z + y \]
\[ y := y - 1 \]
\[ y := 0 \]
**Very Busy Expressions**

- An expression $e$ is very busy at point $p$ if
  - On every path from $p$, expression $e$ is evaluated before the value of $e$ is changed

- Optimization
  - Can hoist very busy expression computation

- What kind of problem?
  - Forward or backward?
  - May or must?

**Space of Data Flow Analyses**

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live variables</td>
<td>Very busy expressions</td>
</tr>
</tbody>
</table>

- Most data flow analyses can be classified this way
  - A few don’t fit: bidirectional analysis
  - Lots of literature on data flow analysis
Data Flow Facts and Lattices

- Typically, data flow facts form a lattice
  - Example: Available expressions

\[
\begin{array}{ccc}
& & a+b, a*b, a+1 \\
a+b, a*b & & a+b, a+1 \\
a*b & & a+b \\
(a) & & (none) \\
\end{array}
\]

A partial order is a lattice if \( \sqcap \) and \( \sqcup \) are defined on any set:

- \( \sqcap \) is the meet or greatest lower bound operation:
  - \( x \sqcap y \leq x \) and \( x \sqcap y \leq y \)
  - If \( z \leq x \) and \( z \leq y \), then \( z \leq x \sqcap y \)
- \( \sqcup \) is the join or least upper bound operation:
  - \( x \leq x \sqcup y \) and \( y \leq x \sqcup y \)
  - If \( x \leq z \) and \( y \leq z \), then \( x \sqcup y \leq z \)

Partial Orders

- A partial order is a pair \((P, \leq)\) such that
  - \( \leq \subseteq P \times P \)
  - \( \leq \) is reflexive: \( x \leq x \)
  - \( \leq \) is antisymmetric: \( x \leq y \) and \( y \leq x \Rightarrow x = y \)
  - \( \leq \) is transitive: \( x \leq y \) and \( y \leq z \Rightarrow x \leq z \)

Lattices

Forward Must Data Flow Algorithm

\[
\begin{align*}
\text{Out}(s) & = \text{Top for all statements } s \\
\end{align*}
\]
Forward Must Data Flow Algorithm

Out(s) = Top for all statements s
//Slight acceleration: Could set Out(s) = Gen(s) ∪ (Top - Kill(s))

W := { all statements } (worklist)

repeat
Take s from W
In(s) := \bigcap_{s' \in \text{pred}(s)} Out(s')
temp := Gen(s) ∪ (In(s) - Kill(s))
if (temp ≠ Out(s)) {
Out(s) := temp
W := W ∩ succ(s)
}
until W = ∅
Forward Must Data Flow Algorithm

Out(s) = Top for all statements s
//Slight acceleration: Could set Out(s) = Gen(s) ∪ (Top - Kill(s))
W := { all statements } (worklist)
repeat
    Take s from W
    In(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')
temp := Gen(s) ∪ (In(s) - Kill(s))
if (temp ≠ Out(s)) {
    Out(s) := temp
    W := W \cap \text{succ}(s)
}
until W = ∅

Out(s) = Top for all statements s
//Slight acceleration: Could set Out(s) = Gen(s) ∪ (Top - Kill(s))
W := { all statements } (worklist)
repeat
    Take s from W
    In(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')
temp := Gen(s) ∪ (In(s) - Kill(s))
if (temp ≠ Out(s)) {
    Out(s) := temp
    W := W \cap \text{succ}(s)
}
until W = ∅
**Forward vs. Backward**

**Out(s) = Top for all statements s**

---

//Slight acceleration: Could set Out(s) = Gen(s) \cup (Top - Kill(s))

W := \{ all statements \} \ (worklist)

repeat

Take s from W

In(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')

temp := \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s))

if (temp \neq \text{Out}(s)) {

Out(s) := temp

W := W \cap \text{succ}(s)

}

until W = \emptyset

---

**In(s) = Top for all s**

---

W := \{ all statements \} \ (worklist)

repeat

Take s from W

Out(s) := f_s(s' \in \text{pred}(s)) \text{Out}(s')

temp := f_s(s' \in \text{succ}(s)) \text{In}(s')

if (temp \neq \text{Out}(s)) {

Out(s) := temp

W := W \cup \text{succ}(s)

}

until W = \emptyset

---
Flow-Sensitivity

- Data flow analysis is flow-sensitive
  - The order of statements is taken into account
  - I.e., we keep track of facts per program point
- Alternative: Flow-insensitive analysis
  - Analysis the same regardless of statement order
  - Standard example: types

```
 /* x : int */ x := ... /* x : int */
```

Path-Sensitivity

A path-sensitive analysis computes different pieces of analysis information dependent on the predicates at conditional branch instructions.

For instance, if a branch contains a condition $x > 0$, then on the fall-through path, the analysis would assume that $x \leq 0$ and on the target of the branch it would assume that indeed $x > 0$ holds.
Context-Sensitivity

Context-sensitive Analysis
A context-sensitive analysis is an inter-procedural analysis that considers the calling context when analyzing the target of a function call. In particular, using context information one can jump back to the original call site, whereas without that information, the analysis information has to be propagated back to all possible call sites, potentially losing precision.

Terminology Review

- Must vs. May
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive

Data Flow Analysis and Functions

- What happens at a function call?
  - Lots of proposed solutions in data flow analysis literature
  - In practice, only analyze one procedure at a time
- Consequences
  - Call to function kills all data flow facts
  - May be able to improve depending on language, e.g., function call may not affect locals
More Terminology

- An analysis that models only a single function at a time is intraprocedural.
- An analysis that takes multiple functions into account is interprocedural.

Data Flow Analysis and The Heap

- Data Flow is good at analyzing local variables.
  - But what about values stored in the heap?
  - Not modeled in traditional data flow.
- In practice: 
  - $x := e$
  - Assume all data flow facts killed (!)
  - Or, assume write through $x$ may affect any variable whose address has been taken.
- In general, hard to analyze pointers.

Outline

1. Program Analysis
2. Data Flow Analysis
   - Available Expressions
   - Liveness Analysis
   - Reaching Definitions
   - Very Busy Expressions
3. Theory Behind
4. Sensitivity
5. Summary

Data Flow Analysis and Optimization

- Moore’s Law: Hardware advances double computing power every 18 months.

Proebsting’s Law: Compiler advances double computing power every 18 years.
Moore's Law: Hardware advances double computing power every 18 months.

Proebsting’s Law: Compiler advances double computing power every 18 years.

Available Expressions
- Liveness Analysis
- Reaching Definitions
- Very Busy Expressions

The theory behind the program analysis
- Sensitivity

References
- Program analysis and understanding by Prof. Mike Hicks at UMD
- Principles of Program Analysis
  http://www2.imm.dtu.dk/riis/PPA/ppasup2004.html