Lectures 16, 17: Dataflow Analysis

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Programming Languages

Based on slides by Jeff Foster
Abstract syntax trees

- **ASTs are abstract**
  - They don’t contain all information in the program
    - E.g., spacing, comments, brackets, parentheses
  - Any ambiguity is resolved
    - E.g., \( a + b + c \) produces the same AST as \( (a + b) + c \)

- but not great for analysis
  - An AST has many similar forms
    - E.g., for, while, repeat..until, ...
    - E.g., if, switch, ...
  - AST expressions might be complex, nested
    - E.g., \( (10 \times x) + (y > 3?5 \times z : z) \)

- We want a simpler representation for analysis
  - ...at least for dataflow analysis
Control-flow graph (CFG)

- A directed graph, where:
  - Each node represents a statement
  - Each edge represents control flow (i.e. what happens after what)
- Statements may be
  - Assignments \( x := y \ op \ z \) or \( x := \ op \ y \)
  - Copy statements \( x := y \)
  - Branches \( \text{goto} \ L \) or \( \text{if} \ x \ \text{relop} \ y \ \text{goto} \ L \)
  - etc.
Control-flow graph example

\[
x := a + b
\]
\[
y := a \times b
\]
\[
y > a
\]
\[
a := a + 1
\]
\[
x := a + b
\]
Kinds of CFGs

- We usually don’t include declarations (e.g., \texttt{int x})
  - Some CFG implementations do
- We may add special, unique “enter” and “exit” nodes
- We can group “straight-line” code into basic blocks
  - Straight-line: without branches, simple instructions one after the other
Control-flow graph with basic blocks

- Can lead to more efficient implementations
- But, is more complicated
  - We will use single-statement blocks here
Control-flow graph with entry/exit

```
entry
→
x := a + b
→
y := a * b
→
y > a
→
a := a + 1
→
x := a + b
→
exit
```

Pratikakis (CSD)
CFG versus AST

- CFGs are simpler than ASTs
  - Fewer forms, less redundancy, simpler expressions
  - Capture flow of control better, easier to see execution paths
- But, AST is a more faithful representation
  - CFGs introduce temporary variables
  - CFGs lose the block-structure of the program
- AST benefits
  - Easier for reporting errors and other compiler messages
  - Easier to explain to the programmer
  - Easier to unparse and produce code closer to the original
Dataflow analysis

- A framework for proving facts about programs
- Reasons about lots of little facts
- Little or no interaction between different facts
  - Works best on properties about how the program computes
- Based on all paths through the program control-flow
  - Including infeasible paths
Available expressions

- An expression $e$ is available at a program point $p$ if:
  - $e$ is computed on every path leading to $p$, and
  - the value of $e$ has not changed since it was last computed

- Used in compiler optimization
  - If an expression is available don’t recompute its value
  - Instead, save it in a register the first time, and use that
  - …if possible
Is expression $e$ available?

Possible facts:
- $a + b$ is available
- $a \times b$ is available
- $a + 1$ is available
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></td>
</tr>
<tr>
<td>$y := a \times b$</td>
<td>$a \times b$</td>
<td></td>
</tr>
<tr>
<td>$a := a + 1$</td>
<td></td>
<td>$a + 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a + b$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a \times b$</td>
</tr>
</tbody>
</table>
Terminology

- A **joint point** is a program point where two branches meet
- Available expressions is a **forward must** problem
  - *Forward* means the facts flow from “in” to “out” at every node, follow the edge arrows
  - *Must* means at every joint point, the property must hold on *all* paths joined
- There are also **backward** and **may** problems
  - *Backward* means the facts flow from “out” to “in” at every node, backwards on the edges
  - *May* means at every joint point, the property must hold on *any* of the joined paths
- All combinations:
  - Forward may, backward must, etc.
Dataflow equations

- If $s$ is a statement
  - $\text{succ}(s)$ is the set of all immediate successor statements of $s$
  - $\text{pred}(s)$ is the set of all immediate predecessor statements of $s$
  - $\text{In}(s)$ is the set of facts at the program point just before $s$
  - $\text{Out}(s)$ is the set of facts at the program point just after $s$

- Forward must:
  - $\text{In}(s) = \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$
  - $\text{Out}(s) = \text{Gen}(s) \cup (\text{In}(s) \setminus \text{Kill}(s))$
Live variables

- A variable $x$ is *live* at a program point $p$ if:
  - $x$ will be used on some execution path starting at $p$
  - before $x$ is overwritten

- Compiler optimization
  - If a variable is not live, there’s no need to keep it in a register
  - If a variable is dead at an assignment, we can eliminate the assignment
Dataflow equations

- Liveness is a *backward may* problem
  - To decide if a variable is live at a program point $p$, we need to look at the paths starting at $p$
  - The variable is live if it is used on *any* future program point

- Backward may:
  - $Out(s) = \bigcup_{s' \in succ(s)} In(s')$
  - $In(s) = Gen(s) \cup (Out(s) \setminus Kill(s))$
Gen and kill

- All possible facts:
  - $a$ is live
  - $b$ is live
  - $x$ is live
  - $y$ is live

- 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>$y := a \times b$</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>
An expression $e$ is very busy at a program point $p$ if:
- On every path from $p$, expression $e$ is evaluated before its value is changed

Compiler optimization
- The compiler can lift very busy expression computation

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

- A *definition* of a variable $x$ is an assignment to $x$
- A definition of a variable $x$ *reaches* a program point $p$ if:
  - There is no intervening assignment to $x$ between the definition and $p$
- Also called “def-use” information
- What kind of problem?
  - Forward or backward?
  - May or must?
Dominators

- A program point $p$ dominates another program point $p'$ if:
  - $p$ occurs in all paths from the start of the program to $p'$

- What kind of problem?
  - Forward or backward?
  - May or must?
## Space of dataflow analyses

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

- Most dataflow analyses can be classified this way
  - A few cannot: e.g., bidirectional analyses
- Lots of literature on dataflow analysis
So far

- ASTs are very *abstract*, not ideal for program analysis
- Control-flow graph is an alternative representation of the program
  - Captures flow of control, all execution paths
  - Better represents computation steps
  - But, not as close to the original source
- Dataflow analysis: computes a solution to dataflow equations for a program property
  - Depending on property: forward/backward, may/must analysis
  - Worklist algorithm, computes solution per program point
- Examples: available expressions, liveness, very busy expressions, etc.
Formalizing it

• Some algebra background
• Formalization of dataflow analysis
• Properties of dataflow algorithms
  ▶ Termination
  ▶ Solving algorithms
  ▶ Fixpoints
  ▶ Accuracy
• Implementation issues
Partial orders

A partial order is a pair \((P, \leq)\) of a set \(P\) and a relation \(\leq\) such that:

- \((\leq) \subseteq (P \times P)\): The relation \(\leq\) is defined only over elements of \(P\)
- \(\leq\) is reflexive: \(x \leq x\), for all \(x \in P\)
- \(\leq\) is anti-symmetric: if \(x \leq y\) and \(y \leq x\) then \(y = x\)
- \(\leq\) is transitive: if \(x \leq y\) and \(y \leq z\) then \(x \leq z\)
Lattices

- A partial order is a lattice if \( \sqcap \) and \( \sqcup \) are defined such that:
  - \( \sqcap \) 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 \( x \sqcap y \leq z \)
Lattices (cont’d)

- A finite partial order is a lattice if meet and join exist for every pair of elements.
- A lattice has unique elements $\top$ (top) and $\bot$ (bottom) such that:
  - $x \sqcap \bot = \bot$
  - $x \sqcap \top = x$
  - $x \sqcup \bot = x$
  - $x \sqcup \top = \top$
- In a lattice
  - $x \leq y$ if and only if $x \sqcap y = x$
  - $x \leq y$ if and only if $x \sqcup y = y$
- A partial order $P$ is a complete lattice if meet and join are defined on any set $S \subseteq P$. 

Typically, sets of dataflow facts form a lattice

- Top element is $\top = \{a + b, a \ast b, a + 1\}$
- Bottom element is $\bot = \emptyset$
Forward-must dataflow algorithm

Forward-Must($CFG$)
for all statements $s \in CFG$
    $Out(s) := \top$
$W := \{\text{all statements}\}$
while $W \neq \emptyset$
    take $s$ from $W$
    $In(s) := \bigcap_{s' \in \text{pred}(s)} Out(s')$
    $tmp := Gen(s) \cup (In(s) \setminus Kill(s))$
    if $tmp \neq Out(s)$ then
        $Out(s) := tmp$
        $W := W \cup \text{succ}(s)$
    end if
end while
Monotonicity

- A function $f$ on a partial order is *monotonic* if
  \[ x \leq y \Rightarrow f(x) \leq f(y) \]

- Easy to check that operations to compute $In$ and $Out$ are monotonic
  \[ In(s) := \bigcap_{s' \in \text{pred}(s)} Out(s') \]
  \[ tmp := Gen(s) \cup (In(s) \setminus Kill(s)) \]

- Putting these together
  \[ tmp := f_s \left( \bigcap_{s' \in \text{pred}(s)} Out(s') \right) \]
Useful lattices

- \((2^S, \subseteq)\) forms a lattice for any set \(S\)
  - \(2^S\) is the powerset of \(S\): the set of all subsets
- If \((S, \leq)\) is a lattice, so is \((S, \geq)\)
  - I.e., we can flip a lattice upside-down and still have a lattice
- The lattice for constant propagation is:

```
Top

1  2  3  ...

Bottom
```
Termination

- The algorithm terminates because
  - The lattice has finite height
  - The operations to compute $In$ and $Out$ are monotonic
  - On every iteration:
    - We reduce the size of the worklist or
    - We move the set of facts at a statement down the lattice
Forward dataflow

\textbf{Forward}(CFG)

for all statements \( s \in CFG \)

\( \text{Out}(s) := \top \)

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

while \( W \neq \emptyset \)

\begin{align*}
& \text{take } s \text{ from } W \\
& \text{tmp} := f_s \left( \bigcap_{s' \in \text{pred}(s)} \text{Out}(s') \right) \\
& \text{if } \text{tmp} \neq \text{Out}(s) \text{ then} \\
& \quad \text{Out}(s) := \text{tmp} \\
& \quad W := W \cup \text{succ}(s) \\
& \text{end if} \\
& \text{end while}
\end{align*}
Lattices for known analyses

- Available expressions
  - \( P = \{ \text{sets of expressions} \} \)
  - \( S_1 \cap S_2 = S_1 \cap S_2 \)
  - \( \top = \{ \text{all expressions} \} \)

- Reaching definitions
  - \( P = \{ \text{all assignment statements} \} \)
  - \( S_1 \cap S_2 = S_1 \cup S_2 \)
  - \( \top = \emptyset \)
Fixpoints

- We always start with $\top$
  - Every expression is available/no definitions reach this point
  - The most optimistic assumption
  - The strongest hypothesis possible: true at the fewest number of states
- Revise as we encounter contradictions
  - Always move down the lattice (using $\sqcap$)
- Result: greatest fixpoint
Forward vs. backward dataflow

**Forward\((CFG)\)**

for all statements \(s \in CFG\)

\[\text{Out}(s) := \top\]

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

while \(W \neq \emptyset\)

take \(s\) from \(W\)

\[tmp := f_s\left(\bigcap_{s' \in \text{pred}(s)} \text{Out}(s')\right)\]

if \(tmp \neq \text{Out}(s)\) then

\[\text{Out}(s) := tmp\]

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

end if

end while

**Backward\((CFG)\)**

for all statements \(s \in CFG\)

\[\text{In}(s) := \top\]

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

while \(W \neq \emptyset\)

take \(s\) from \(W\)

\[tmp := f_s\left(\bigcap_{s' \in \text{succ}(s)} \text{In}(s')\right)\]

if \(tmp \neq \text{In}(s)\) then

\[\text{In}(s) := tmp\]

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

end if

end while
Termination revisited

- How many times can we apply the step:
  - \( \text{tmp} := f_s \left( \prod_{s' \in \text{pred}(s)} \text{Out}(s') \right) \)
  - \( \text{if tmp} \neq \text{Out}(s) \text{ then ...} \)

- Claim: \( \text{Out}(s) \) only shrinks
  - Proof: \( \text{Out}(s) \) starts as \( \top \)
    - ★ so it must be \( \text{tmp} = \top \) after the first step
  - Assume \( \text{Out}(s) \) shrinks for all predecessors \( s' \) of \( s \)
  - Then \( \prod_{s' \in \text{pred}(s)} \text{Out}(s') \) also shrinks
  - Since \( f_s \) is monotonic, \( f_s \left( \prod_{s' \in \text{pred}(s)} \text{Out}(s') \right) \) shrinks
Termination revisited (cont’d)

- A *descending chain* in a lattice is a sequence
  - \( x_0 \sqsubseteq x_1 \sqsubseteq \ldots \)

- The *height* of a lattice is the length of the longest descending chain in the lattice

- Then, dataflow must terminate in \( O(nk) \) time, where
  - \( n \) is the number of statements in a program
  - \( k \) is the height of the lattice
  - ...assuming the meet operation takes \( O(1) \) time
Least vs. greatest fixpoint

- Usually in dataflow we start with $\top$, move down using $\sqcap$
  - To do this, we need a *meet semilattice with top*
    - complete meet semilattice: meet defined for all elements
    - finite height ensures termination
  - We compute the greatest fixpoint: the solution highest in the lattice

- In other settings (e.g, denotational semantics) we start with $\bot$, move up using $\sqcup$
  - Computes the least fixpoint
Distributive dataflow problems

- By monotonicity we have $f(x \sqcap y) \leq f(x) \sqcap f(y)$
- A function $f$ is *distributive* if $f(x \sqcap y) = f(x) \sqcap f(y)$
- When using distributive functions, joins lose no information:

\[
\begin{align*}
  k(h(f(\top) \sqcap g(\top))) &= \\
  k(h(f(\top)) \sqcap h(g(\top))) &= \\
  k(h(f(\top))) \sqcap k(h(g(\top)))
\end{align*}
\]
Accuracy

- Ideally, we want the *meet over all paths* (MOP) solution
  - Assume $f_s$ is the transfer function of statement $s$
  - Assume $p$ is a path $s_1, \ldots, s_n$
  - We define $f_p = f_n; \ldots; f_1$
  - Let $\text{path}(s)$ be the set of paths from the entry to $s$
  - Then
    \[
    MOP(s) = \bigcap_{p \in \text{path}(s)} f_p(\top)
    \]

- If a dataflow problem is distributive then algorithm produces the MOP solution
What problems are distributive?

- Analyses of *how* the program computes
  - Live variables
  - Available expressions
  - Reaching definitions
  - Very busy expressions

- All Gen/Kill problems are distributive

- Analyses of *what* the program computes are not distributive
  - Constant propagation
Implementation issues

- Dataflow facts are assertions of what is true at every program point
- We represent the set of facts as a bit-vector
  - Order all possible facts
  - The $i$-th bit represents the $i$-th fact
  - Intersection is bitwise and
  - Union is bitwise or
- “Only” a constant factor speedup
  - But very useful in practice!
Basic blocks

- A basic block is a sequence of statements such that
  - No statement except the last is a branch
  - There are no branches to any statement in the block except the first

- Practically, when implementing dataflow
  - Compute Gen/Kill for each basic block
    - By composing the transfer functions of statements
  - Store In/Out sets only for each basic block
  - Typical basic block is around 5 statements
CFG visiting order - acyclic

- Assume forward dataflow
  - Let $G = (V, E)$ be the control-flow graph
  - and $k$ be the height of the lattice
- If $G$ is acyclic, visit it in topological order
  - For every edge, visit the head node before the tail node
- Running time is $O(|E|)$
  - Regardless of the lattice size
CFG visiting order - cycles

- If $G$ has cycles, visit in reverse postorder
  - Order of depth-first search
- Let $Q$ be the max number of back-edges on a path without cycles
  - Depth of loop nesting
  - Back edge goes from descendant node to ancestor node in DFS tree
- Then if $\forall x. f(x) \leq x$ (sufficient, not necessary)
  - Running time is $O((Q + 1)|E|)$
    * depends on definition of $\top$: $f$ shrinks the fact set
Flow-sensitivity

- Dataflow analysis is *flow-sensitive*
  - The answer produced depends on the order of statements in the program
  - We keep track of facts *per program point*
- Alternative: *flow-insensitive* analysis
  - Analysis result does not depend on the statement order
  - Standard example: types
    - A variable has the same type before and after any statement
Dataflow analysis and functions

- What happens at function calls?
  - Lots of possible solutions in the literature
- Usually, analyze one function at a time
  - Called *intraprocedural* analysis
  - When analyzing multiple functions together called *interprocedural*
    - Special case: *whole-program* analysis

- Consequences of intraprocedural analysis
  - Call to function kills all dataflow facts
  - Depending on language, we may be able to save some: e.g., called function cannot affect caller’s local variables
Dataflow analysis and pointers

- Dataflow is good at analyzing local variables
  - What about values in the heap?
  - Not modeled in traditional dataflow
- In practice, when $\star x := e$
  - Assume it can write anywhere
  - All dataflow facts killed!
  - Better: assume it can write all variables whose address is taken
- In general: it’s hard to analyze pointers
Analysis terminology

- **Must vs. May**
  - Definition depends on which answer is imprecise: yes/maybe, or no/maybe result
  - Not always followed in the literature
- **Forward vs. Backward**
- **Flow-sensitive vs. flow-insensitive**
- **Distributive vs. non-distributive**
- **Intraprocedural vs. interprocedural vs. whole-program**
Dataflow analysis used in practice

- Moore’s law: Hardware advances double computing power every 18 months
- Proebsting’s law: Compiler advances double computing power every 18 years
  - Costs less than making chips, but not very much worth the trouble for optimization

- Useful for other things:
  - bug-finding: memory leaks, security vulnerabilities, etc.
  - support for high-level language-features
  - program understanding
  - …