Dataflow Analysis (Chapter 17)

• What is dataflow analysis?
• How is a particular analysis defined?
• Look at a few specific analyses
• Once you have analyzed some particular aspect of the data flow of a program, what can you do that you couldn’t before?
  – Register Allocation
  – Constant propagation
  – Dead code elimination
  – Common sub-expression elimination
  – ...

Optimization

• What is the "best" translation of a given program to a given machine code?
• There are very few "optimal" translations known
• What we really want to do is more like anti-pessimism
  – We don’t want to generate obviously bad code
• What that means is that if you have a transformation that almost always makes code better, then we want to perform that transformation.
• Most transformations have preconditions
Scope of Optimizations

• **Intraprocedural global optimizations**
  – Limited to a single procedure at a time
  – Global in that they consider the entire procedure at once

• **Interprocedural optimizations are also done**
  – But are more complicated

• **Local optimizations are also done**
  – But have lower “yield”
  – These are sometimes called “peephole” optimizations
Generic Optimization Recipe

• All of the optimizations that I mentioned
  – Register allocation
  – Constant propagation
  – Dead code elimination
• Common sub-expression elimination

• Follow the same recipe
  – Perform dataflow analysis of some kind
    • Identify opportunities to optimize
    • Gather evidence that the transformation is correct
      – Program behaviour will be preserved
  – Transform the program to make it faster somehow
So what is dataflow analysis?

- As the name says, it is an analysis of how data will flow during the execution of a program
  - Where are values produced?
  - Where are values consumed?
  - What values will flow along which edges of the control flow graph?

- Different analyses concern themselves with different aspects of how data is produced, communicated, and consumed
Register Allocation

• We want to decide which registers should hold what Temps
• Why is this a form of “optimization”?
• What information do we need to do this?
  – Liveness
Constant propagation

- We want to use constant values in place of Temps that just hold constant values
- Why is this a form of “optimization”?  
- What information do we need to do this?  
  - What values reach a particular instruction?  
  - Are those values constant?
Dead code elimination

• We want to remove code that computes values that don’t affect the program in the future
  – i.e., values that are dead when they are created
• Why is this a form of “optimization”?  
• What information do we need to do this?  
  – Liveness
Common sub-expression elimination

- We want to avoid re-computing values that we have already computed
- Why is this a form of “optimization”? 
- What information do we need to do this?  
  - What expressions have already been computed? 
  - Have they been computed on every possible path to this instruction? 
  - Have any of the values that they depend on changed?
Which program representation?

- For register allocation, we used final code augmented with use/def information
- Is this always a good choice? Maybe not – it doesn’t capture meaning
- For many optimizations, we need to know the meaning of the code in order to ensure that behaviour is preserved
- What representation is all of:
  - simple
  - portable
  - meaningful
Quadruples: Simplified IR Trees

• Take linearized IR trees
• Restrict them to have either a single MEM or BINOP
• The instructions then look like:
  – a <- b binop c
  – Where a, b, and c are either constants or Temps
• Some machine architectures have instructions that take exactly this form
  – MIPS
  – ARM
Simplified IR Trees

\[ a \leftarrow b \text{ binop } c \]
\[ a \leftarrow b \]
\[ M[a] \leftarrow b \]
\[ a \leftarrow M[b] \]
\[ a \leftarrow f(a_1,a_2,\ldots,a_n) \]
\[ f(a_1,a_2,\ldots,a_n) \]

\[ \text{goto } L \]
\[ L: \]
\[ \text{if } a \text{ relop } b \text{ goto } L1 \]
\[ \text{else goto } L2 \]
In fact, with a bit of work

• We can perform our analyses on unrestricted IR
• The key is that each linearized IR statement defines at most one temp
• The primary wrinkle is that you have to search the whole IR tree on the “right hand side” of each statement to find the temps that are used by the instruction
“May” vs “Must” analyses

• May the value from a particular definition of \( a \) still be in \( a \) when we reach some statement \( s \)?
  – Is it possible that ...

• Must the value from a particular definition of \( a \) still be in \( a \) when we reach some statement \( s \)?
  – Is it guaranteed that ...

• This question of “may” vs “must” gives rise to completely different analyses

• Is the liveness analysis we used for register allocation a “may” or a “must” analysis?
Reaching Definitions

- Given some unambiguous definition of a Temp $a$
- Is it possible that that definition reaches a particular statement $s$?

```
a ← XXX
```

- "May" or "must"?
Ambiguous vs Unambiguous Definitions

• Some statements unambiguously define a Location
  \( a \leftarrow XXX \)
  – If this statement is executed, \( a \) gets defined

• Other statements “might” define a “Location”
  \( M[p] \leftarrow XXX \)
  – If there are any Locations that are held in memory, this might change them
  – Unless we can prove that \( p \neq \) address of the Location

• We have arranged our use of Temps so that they can only be unambiguously defined
  – There’s no way to alias Temps, so focus on them.
Back to Reaching Definitions

- A definition $d$ of a Temp $a$ is said to “reach” a statement $s$ in the program if
  - There exists some path from $d$ to $s$ in the flowgraph that does not contain a definition of $a$

\[
d: a \leftarrow XXX
\]
\[
a \leftarrow YYY \quad \quad \quad \quad a \leftarrow ZZZ
\]
\[
s \text{ (that reads } a\text{)}
\]

- Here, $d$ DOES NOT reach $s$
Back to Reaching Definitions

- A definition \( d \) of a Temp \( a \) is said to “reach” a statement \( s \) in the program if
  - There exists some path from \( d \) to \( s \) in the flowgraph that does not contain a definition of \( a \)

\[
\text{d: } a \leftarrow XXX
\]

\[
\text{a } \leftarrow YYY
\]

\[
\text{s (that reads } a)\]

- Here, \( d \) DOES reach \( s \)
**Exercise**: To the right of each line, write the line numbers of lines with definitions that (may) reach the end of that line.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>a ← 5</td>
</tr>
<tr>
<td>2:</td>
<td>c ← 1</td>
</tr>
<tr>
<td>3:</td>
<td>if c &gt; a goto 6</td>
</tr>
<tr>
<td>4:</td>
<td>c ← c + c</td>
</tr>
<tr>
<td>5:</td>
<td>goto 3</td>
</tr>
<tr>
<td>6:</td>
<td>a ← c − a</td>
</tr>
<tr>
<td>7:</td>
<td>c ← 0</td>
</tr>
</tbody>
</table>

Definitions that may reach the end of the line
“Gen” and “Kill” sets

It is useful to think about various dataflow analyses in terms of “gen” and “kill” sets. Let's tabulate “gen” and “kill” sets for **Reaching Definitions**.

<table>
<thead>
<tr>
<th>Statement $s$</th>
<th>$\text{gen}[s]$</th>
<th>$\text{kill}[s]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d: t \leftarrow b \oplus c$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d: t \leftarrow M[b]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$M[a] \leftarrow b$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if $a &lt; b$ L1 else L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>goto L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f(a_1, a_2, \ldots)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d: t \leftarrow f(a_1, a_2, \ldots)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
“Gen” and “Kill” sets

It is useful to think about various dataflow analyses in terms of “gen” and “kill” sets. Let's tabulate “gen” and “kill” sets for Reaching Definitions.

<table>
<thead>
<tr>
<th>Statement s</th>
<th>gen[s]</th>
<th>kill[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{let } d : t \leftarrow b \oplus c )</td>
<td>{d}</td>
<td>(\text{defs}(t) - {d})</td>
</tr>
<tr>
<td>( \text{let } d : t \leftarrow M[b] )</td>
<td>{d}</td>
<td>(\text{defs}(t) - {d})</td>
</tr>
<tr>
<td>( \text{let } M[a] \leftarrow b )</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>( \text{if } a &lt; b \text{ then } L1 \text{ else } L2 )</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>goto L</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>L:</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>( \text{let } f(a_1,a_2,...) )</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>( \text{let } d : t \leftarrow f(a_1,a_2,...) )</td>
<td>{d}</td>
<td>(\text{defs}(t) - {d})</td>
</tr>
</tbody>
</table>
Dataflow Equations

Typically, we “define” the dataflow analysis by writing down dataflow equations derived from the flowgraph (e.g., as we did for liveness analysis).

Let's quickly recap our notation / terminology.
Flow Graph Terminology

\begin{align*}
\text{pred}[n] &= \{ p1, p2 \} \\
\text{in}[n] &= \{ x \mid x \text{ is “interesting” just before } n \} \\
\text{succ}[n] &= \{ s1, s2 \} \\
\text{out}[n] &= \{ x \mid x \text{ is “interesting” just after } n \}
\end{align*}

What is “interesting” depends on the analysis of course.
Dataflow Analysis

What is “interesting” depends on the analysis of course.

E.g., Liveness:

\[ \text{in}[n] = \{ x \mid \text{Temp } x \text{ is live just before } n \} \]
\[ \text{out}[n] = \{ x \mid \text{Temp } x \text{ is live just after } n \} \]

E.g., Reaching definitions:

\[ \text{in}[n] = \{ d \mid \text{Definition } d \text{ reaches the point just before } n \} \]
\[ \text{out}[n] = \{ d \mid \text{Definition } d \text{ reaches the point just after } n \} \]
Back\textsuperscript{2} to Reaching Definitions

- A definition \( d \) of a Temp \( a \) is said to “reach” a statement \( s \) in the program if
  - There exists some path from \( d \) to \( s \) in the flowgraph that does not contain a definition of \( a \)

\[
\text{d: } a \leftarrow \text{XXX} \\
\quad \text{a } \leftarrow \text{YYY} \\
\quad \text{a } \leftarrow \text{ZZZ} \\
\quad \text{s (that reads a)}
\]

- Here, \( d \) DOES NOT reach \( s \)
A definition $d$ of a Temp $a$ is said to “reach” a statement $s$ in the program if

- There exists some path from $d$ to $s$ in the flowgraph that does not contain a definition of $a$

Here, $d$ DOES reach $s$
Reaching Definitions

• What are the equations?
Reaching Definitions

- What are the equations?

\[
\begin{align*}
in[n] &= \bigcup_{p \text{ in pred}[n]} \text{out}[p] \\
\text{out}[n] &= \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\end{align*}
\]
in[n] = use[n] ∪ (out[n] – def[n])

out[n] = \bigcup_{s \text{ in succ}[n]} \text{in}[s]
Reaching Definitions vs Liveness

\[ \text{in}[n] = \bigcup_{p \text{ in pred}[n]} \text{out}[p] \]

\[ \text{out}[n] = \text{gen}[n] \bigcup (\text{in}[n] - \text{kill}[n]) \]

\[ \text{in}[n] = \text{use}[n] \bigcup (\text{out}[n] - \text{def}[n]) \]

\[ \text{out}[n] = \bigcup_{s \text{ in succ}[n]} \text{in}[s] \]
Forwards vs Backwards Analyses

• We sometimes classify analyses as forwards vs backwards depending on how information flows in the flow graph

• Which is liveness?
  – Forwards or Backwards?

• Which is reaching definitions?
  – Forwards or Backwards?
Liveness using “gen” and “kill”

• We did liveness using “use” and “def”
  – I think that was more intuitive than using “gen” and “kill”
  – But we can trivially reformulate in “gen” and “kill” terms
• “use” generates liveness (= “gen”)
• “def” kills liveness (= “kill”)

\[
in[n] = \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])
\]

\[
\text{out}[n] = \bigcup_{s \text{ in succ}[n]} \text{in}[s]
\]
What can we use Reaching Definitions for?

- Reaching definitions tells us what definitions might reach a particular instruction
- Why do we care?
- Constant propagation
  - We can replace the use of a Temp in an instruction with the constant that defines it if:
    - Only one definition of the Temp reaches the instruction
    - That definition uses a constant
  - Or, slightly more generally
    - All the definitions that reach the instruction are constant and use the same constant
Dead code elimination

• We want to remove code that computes values that don’t affect the program in the future
  – i.e., values that are dead when they are created

• Why is this a form of “optimization”?

• What information do we need to do this?
  – Liveness
Common sub-expression elimination

- We want to avoid re-computing values that we have already computed
- Why is this a form of “optimization”? 
- What information do we need to do this?
  - What expressions have already been computed?
  - Have they been computed on every possible path to this instruction?
  - Have any of the values that they depend on changed?
Available Expressions

• An expression $x \text{ OP } y$ is “available” at a statement $s$ in the flow graph if, on every path to statement $s$, $x \text{ OP } y$ is computed at least once, and there are no definitions of $x$ or $y$ since the most recent occurrence of $x \text{ OP } y$ on that path.

• This is a “must” analysis
Available Expressions

\[
a \leftarrow x + y
\]

\[
s : c \leftarrow x + y
\]

\[
a \leftarrow x + y
\]
Available Expressions

\[ a \leftarrow x + y \]

\[ s : c \leftarrow x + y \]
Available Expressions

\[
a \leftarrow x + y
\]

\[
a \leftarrow x + y
\]

\[
x \leftarrow 3
\]

\[
s : c \leftarrow x + y
\]
Available Expressions

- An expression $x \text{ OP } y$ is “available” at a statement $s$ in the flow graph if, on every path to statement $s$, $x \text{ OP } y$ is computed at least once, and there are no definitions of $x$ or $y$ since the most recent occurrence of $x \text{ OP } y$ on that path.
- $\text{gen}[s: t \leftarrow x \text{ OP } y] = \{x \text{ OP } y\} - \text{kill}[s]
- \text{kill}[s: t \leftarrow *] = \text{expressions including } t
Available Expressions

• What are the data flow equations?
Available Expressions

- **What are the data flow equations?**

\[
in[n] = \bigcap_{p \in \text{pred}[n]} \text{out}[p]
\]

\[
\text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\]
Computing Available Expressions

The book claims that because the equations use $\cap$ rather than $\cup$ we must use a different way of computing the solution to the equations.

The book is wrong about this, but the approach that the book takes works:

Still iterative (i.e., recompute / loop until no more changes).

But starts from “full” rather than empty sets.

Rationale: “intersections makes sets smaller not bigger”. (this is part of why I don’t like dataflow equations)
Reaching Expressions

Reaching expressions is another analysis we'll need to do CSE optimizations.

“Reaching” is similar to “available” but it is a “may” rather than “must” analysis.

i.e., Roughly speaking:
   Reaching means “may be available”
   Available means “must be available”

Exercise: tabulate the gen/kill sets.
Exercise: write DF equations

Note: in practice it is advantageous to just compute these “as needed” rather than do a “global” analysis. (Why?)
Common Subexpression Elimination

Naïve idea

\[ a \leftarrow b + c \]
\[ \ldots \quad /\!/ \text{no defs of } b \text{ or } c \text{ here} \]
\[ d \leftarrow b + c \]

Since we already computed \( b + c \). We wish to eliminate its second computation:

\[ a \leftarrow b + c \]
\[ \ldots \quad /\!/ \text{no defs of } b \text{ or } c \text{ here} \]
\[ d \leftarrow a \]

But... some complications
- what if \( a \) may have been redefined?
- what if there are multiple paths to reach the second computation.
Common Subexpression Elimination

\[ t \leftarrow a \oplus b \quad t' \leftarrow a \oplus b \quad t'' \leftarrow a \oplus b \]

\[ \ldots \quad \ldots \quad \ldots \]

no definitions of \( a \) or \( b \)

\[ n: \quad r \leftarrow a \oplus b \]

\( a \oplus b \) is “available”

Available expression analysis will tell us that \( r \leftarrow a \oplus b \) is a “redundant” computation.

i.e., we know that if we reach this instruction we already computed \( a \oplus b \) and its value is still the same.

**Q:** How do we transform the program to eliminate the redundant computation?
Common Subexpression Elimination

Introduce a new temp z and use it as follows:

\[
\begin{align*}
    t & \leftarrow a \oplus b \\
    z & \leftarrow t
\end{align*}
\]

\[
\begin{align*}
    t' & \leftarrow a \oplus b \\
    z & \leftarrow t'
\end{align*}
\]

\[
\begin{align*}
    t'' & \leftarrow a \oplus b \\
    z & \leftarrow t''
\end{align*}
\]

Note: Available expressions tells us that we can perform the optimization at node \( n \).

Q: How do we know were to find the nodes \( t \leftarrow a \oplus b, t'' \leftarrow a \oplus b, \ldots \)

A: Search for “reaching expressions \( a \oplus b \)” backwards in the flowgraph from \( n \).