Announcements

• HW6: Dataflow Analysis
  – Available soon

• Talk: Sumit Gulwani of Microsoft
  “Data Manipulation using Programming By Examples and Natural Language”
  – 3:00-4:00 in Wu & Chen

• My office hours: 4:00 – 5:15 today
CODE ANALYSIS
Iterative Dataflow Analysis

• Find a solution to those constraints by starting from a rough guess.
• Start with: in[n] = ∅ and out[n] = ∅
• They don’t satisfy the constraints:
  – in[n] ⊇ use[n]
  – in[n] ⊇ out[n] - def[n]
  – out[n] ⊇ in[n'] if n' ∈ succ[n]

• Idea: iteratively re-compute in[n] and out[n] where forced to by the constraints.
  – Each iteration will add variables to the sets in[n] and out[n]
    (i.e. the live variable sets will increase monotonically)
• We stop when in[n] and out[n] satisfy these equations:
  (which are derived from the constraints above)
  – in[n] = use[n] ∪ (out[n] - def[n])
  – out[n] = \bigcup_{n' \in succ[n]} in[n']
A Worklist Algorithm

- Use a FIFO queue of nodes that might need to be updated.

for all $n$, $\text{in}[n] := \emptyset$, $\text{out}[n] := \emptyset$

$w =$ new queue with all nodes

repeat until $w$ is empty

let $n =$ $w$.pop() \hspace{1cm} // pull a node off the queue

old$\_in =$ $\text{in}[n]$ \hspace{1cm} // remember old $\text{in}[n]$

$\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$

$\text{in}[n] := \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$

if ($\text{old}\_in = \neq \text{in}[n]$), \hspace{1cm} // if $\text{in}[n]$ has changed

for all $m$ in $\text{pred}[n]$, $w$.push($m$) \hspace{1cm} // add to worklist

end
OTHER DATAFLOW ANALYSES
Generalizing Dataflow Analyses

• The kind of iterative constraint solving used for liveness analysis applies to other kinds of analyses as well.
  – Reaching definitions analysis
  – Available expressions analysis
  – Alias Analysis
  – Constant Propagation
  – These analyses follow the same 3-step approach as for liveness.

• To see these as an instance of the same kind of algorithm, the next few examples to work over a canonical intermediate instruction representation called quadruples
  – Allows easy definition of def[n] and use[n]
  – A “looser” variant of LLVM’s IR that doesn’t require the “static single assignment” – i.e. it has mutable local variables
A Quadruple sequence is just a control-flow graph (flowgraph) where each node is a quadruple:

<table>
<thead>
<tr>
<th>Quadruple forms n</th>
<th>def[n]</th>
<th>use[n]</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b \text{ op } c )</td>
<td>{a}</td>
<td>{b,c}</td>
<td>arithmetic</td>
</tr>
<tr>
<td>( a = [b] )</td>
<td>{a}</td>
<td>{b}</td>
<td>load</td>
</tr>
<tr>
<td>([a] = b)</td>
<td>\Ø</td>
<td>{b}</td>
<td>store</td>
</tr>
<tr>
<td>( a = f(b_1,\ldots,b_n) )</td>
<td>{a}</td>
<td>{b_1,\ldots,b_n}</td>
<td>call w/return</td>
</tr>
<tr>
<td>( f(b_1,\ldots,b_n) )</td>
<td>\Ø</td>
<td>{b_1,\ldots,b_n}</td>
<td>call no return</td>
</tr>
<tr>
<td>jump L</td>
<td>\Ø</td>
<td>\Ø</td>
<td>jump</td>
</tr>
<tr>
<td>if a goto L1 else L2</td>
<td>\Ø</td>
<td>{a}</td>
<td>branch</td>
</tr>
<tr>
<td>return a</td>
<td>\Ø</td>
<td>{a}</td>
<td>return</td>
</tr>
</tbody>
</table>
REACHING DEFINITIONS
Reaching Definition Analysis

• Question: what uses in a program does a given variable definition reach?

• This analysis is used for constant propagation & copy prop.
  – If only one definition reaches a particular use, can replace use by the definition (for constant propagation).
  – Copy propagation additionally requires that the copied value still has its same value – computed using an available expressions analysis (next)

• Input: Quadruple CFG
• Output: in[n] (resp. out[n]) is the set of nodes defining some variable such that the definition may reach the beginning (resp. end) of node n
Example of Reaching Definitions

- Results of computing reaching definitions on this simple CFG:

```
b = a + 2

out[1]: {1}
in[2]:   {1}

c = b * b

out[2]: {1,2}
in[3]:   {1,2}

b = c + 1

out[3]: {2,3}
in[4]:   {2,3}

return b * a
```
Reaching Definitions Step 1

- Define the sets of interest for the analysis
- Let $\text{defs}[a]$ be the set of *nodes* that define the variable a
- Define $\text{gen}[n]$ and $\text{kill}[n]$ as follows:

<table>
<thead>
<tr>
<th>Quadruple forms n:</th>
<th>gen[n]</th>
<th>kill[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b \text{ op } c$</td>
<td>${n}$</td>
<td>$\text{defs}[a] - {n}$</td>
</tr>
<tr>
<td>$a = \text{load } b$</td>
<td>${n}$</td>
<td>$\text{defs}[a] - {n}$</td>
</tr>
<tr>
<td>$[a] = b$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$a = f(b_1,...,b_n)$</td>
<td>${n}$</td>
<td>$\text{defs}[a] - {n}$</td>
</tr>
<tr>
<td>$f(b_1,...,b_n)$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{jump } L$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{if } a \text{ goto } L1 \text{ else } L2$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$L:$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{return } a$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Reaching Definitions Step 2

• Define the constraints that a reaching definitions solution must satisfy.

• \( \text{out}[n] \supseteq \text{gen}[n] \)
  “The definitions that reach the end of a node at least include the definitions generated by the node”

• \( \text{in}[n] \supseteq \text{out}[n'] \) if \( n' \) is in \( \text{pred}[n] \)
  “The definitions that reach the beginning of a node include those that reach the exit of any predecessor”

• \( \text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n] \)
  “The definitions that come in to a node either reach the end of the node or are killed by it.”
  – Equivalently: \( \text{out}[n] \supseteq \text{in}[n] - \text{kill}[n] \)
Reaching Definitions Step 3

- Convert constraints to iterated update equations:
  - \( \text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \)
  - \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)

- Algorithm: initialize \( \text{in}[n] \) and \( \text{out}[n] \) to \( \emptyset \)
  - Iterate the update equations until a fixed point is reached

- The algorithm terminates because \( \text{in}[n] \) and \( \text{out}[n] \) increase only monotonically
  - At most to a maximum set that includes all variables in the program

- The algorithm is precise because it finds the smallest sets that satisfy the constraints.
AVAILABLE EXPRESSIONS
Available Expressions

• Idea: want to perform common subexpression elimination:
  – \( a = x + 1 \) \( a = x + 1 \)
  
  \[ \ldots \]
  – \( b = x + 1 \) \( b = a \)

• This transformation is safe if \( x+1 \) means computes the same value at both places (i.e. \( x \) hasn’t been assigned).
  – “\( x+1 \)” is an available expression

• Dataflow values:
  – in\( [n] \) = set of nodes whose values are available on entry to \( n \)
  – out\( [n] \) = set of nodes whose values are available on exit of \( n \)
Available Expressions Step 1

- Define the sets of values
- Define \texttt{gen}[n] and \texttt{kill}[n] as follows:

<table>
<thead>
<tr>
<th>Quadruple forms n:</th>
<th>gen[n]</th>
<th>kill[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b \text{ op} c )</td>
<td>{n} - \text{kill}[n]</td>
<td>\text{uses}[a]</td>
</tr>
<tr>
<td>( a = [b] )</td>
<td>{n} - \text{kill}[n]</td>
<td>\text{uses}[a]</td>
</tr>
<tr>
<td>([a] = b)</td>
<td>(\emptyset)</td>
<td>\text{uses}([x])</td>
</tr>
<tr>
<td>jump L</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>if a goto L1 else L2</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>L:</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>( a = f(b_1,\ldots,b_n) )</td>
<td>(\emptyset)</td>
<td>\text{uses}[a] \cup \text{uses}([x])</td>
</tr>
<tr>
<td>( f(b_1,\ldots,b_n) )</td>
<td>(\emptyset)</td>
<td>\text{uses}([x])</td>
</tr>
<tr>
<td>return a</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

Note the need for “may alias” information…

Note that functions are assumed to be impure…
Available Expressions Step 2

• Define the constraints that an available expressions solution must satisfy.

• $\text{out}[n] \supseteq \text{gen}[n]$
  “The expressions made available by $n$ that reach the end of the node”

• $\text{in}[n] \subseteq \text{out}[n']$ if $n'$ is in $\text{pred}[n]$
  “The expressions available at the beginning of a node include those that reach the exit of every predecessor”

• $\text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n]$
  “The expressions available on entry either reach the end of the node or are killed by it.”
  – Equivalently: $\text{out}[n] \supseteq \text{in}[n] - \text{kill}[n]$

Note similarities and differences with constraints for “reaching definitions”.
Available Expressions Step 3

- Convert constraints to iterated update equations:
  
  - \( \text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n'] \)
  
  - \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)

- Algorithm: initialize \( \text{in}[n] \) and \( \text{out}[n] \) to \{set of all nodes\}
  - Iterate the update equations until a fixed point is reached

- The algorithm terminates because \( \text{in}[n] \) and \( \text{out}[n] \) *decrease only monotonically*
  - At most to a minimum of the empty set

- The algorithm is precise because it finds the *largest* sets that satisfy the constraints.
GENERAL DATAFLOW ANALYSIS
Comparing Dataflow Analyses

• Look at the update equations in the inner loop of the analyses

• Liveness: (backward)
  – Let gen[n] = use[n] and kill[n] = def[n]
  – out[n] := $\bigcup_{n' \in \text{succ}[n]} \text{in}[n']$
  – in[n] := gen[n] $\cup$ (out[n] - kill[n])

• Reaching Definitions: (forward)
  – in[n] := $\bigcup_{n' \in \text{pred}[n]} \text{out}[n']$
  – out[n] := gen[n] $\cup$ (in[n] - kill[n])

• Available Expressions: (forward)
  – in[n] := $\bigcap_{n' \in \text{pred}[n]} \text{out}[n']$
  – out[n] := gen[n] $\cup$ (in[n] - kill[n])
Common Features

• All of these analyses have a domain over which they solve constraints.
  – Liveness, the domain is sets of variables
  – Reaching defns., Available exprs. the domain is sets of nodes
• Each analysis has a notion of gen\[n\] and kill\[n\]
  – Used to explain how information propagates across a node.
• Each analysis is propagates information either forward or backward
  – Forward: in\[n\] defined in terms of predecessor nodes’ out[]
  – Backward: out\[n\] defined in terms of successor nodes’ in[]
• Each analysis has a way of aggregating information
  – Liveness & reaching definitions take union (U)
  – Available expressions uses intersection (∩)
  – Union expresses a property that holds for some path (existential)
  – Intersection expresses a property that holds for all paths (universal)
A forward dataflow analysis can be characterized by:

1. A domain of dataflow values $\mathcal{L}$
   - e.g. $\mathcal{L} = \text{the powerset of all variables}$
   - Think of $\ell \in \mathcal{L}$ as a property, then “$x \in \ell$”
     means “$x$ has the property”

2. For each node $n$, a flow function $F_n : \mathcal{L} \to \mathcal{L}$
   - So far we’ve seen $F_n(\ell) = \text{gen}[n] \cup (\ell - \text{kill}[n])$
   - So: $\text{out}[n] = F_n(\text{in}[n])$
   - “If $\ell$ is a property that holds before the node $n$,
     then $F_n(\ell)$ holds after $n$”

3. A combining operator $\sqcap$
   - “If we know either $\ell_1$ or $\ell_2$ holds on entry
     to node $n$, we know at most $\ell_1 \sqcap \ell_2$”

   - $\text{in}[n] := \sqcap_{n' \in \text{pred}[n]} \text{out}[n']$
for all $n$, $in[n] := T$, $out[n] := T$
repeat until no change
   for all $n$
      
      $in[n] := \prod_{n' \in \text{pred}[n]} out[n']$
      
      $out[n] := F_n(in[n])$
   end
end

• Here, $T \in L$ ("top") represents having the "maximum" amount of information.
  – Having “more” information enables more optimizations
  – “Maximum” amount could be inconsistent with the constraints.
  – Iteration refines the answer, eliminating inconsistencies
Structure of $\mathcal{L}$

- The domain has structure that reflects the “amount” of information contained in each dataflow value.
- Some dataflow values are more informative than others:
  - Write $\mathcal{L}_1 \subseteq \mathcal{L}_2$ whenever $\mathcal{L}_2$ provides at least as much information as $\mathcal{L}_1$.
  - The dataflow value $\mathcal{L}_2$ is “better” for enabling optimizations.
- Example 1: for liveness analysis, smaller sets of variables are more informative.
  - Having smaller sets of variables live across an edge means that there are fewer conflicts for register allocation assignments.
  - So: $\mathcal{L}_1 \subseteq \mathcal{L}_2$ if and only if $\mathcal{L}_1 \supseteq \mathcal{L}_2$
- Example 2: for available expressions analysis, larger sets of nodes are more informative.
  - Having a larger set of nodes (equivalently, expressions) available means that there is more opportunity for common subexpression elimination.
  - So: $\mathcal{L}_1 \subseteq \mathcal{L}_2$ if and only if $\mathcal{L}_1 \supseteq \mathcal{L}_2$
\[ \mathcal{L} \] as a Partial Order

- \( \mathcal{L} \) is a partial order defined by the ordering relation \( \sqsubseteq \).
- A partial order is an ordered set.
- Some of the elements might be incomparable.
  - That is, there might be \( \ell_1, \ell_2 \in \mathcal{L} \) such that neither \( \ell_1 \sqsubseteq \ell_2 \) nor \( \ell_2 \sqsubseteq \ell_1 \)

- Properties of a partial order:
  - Reflexivity: \( \ell \sqsubseteq \ell \)
  - Transitivity: \( \ell_1 \sqsubseteq \ell_2 \) and \( \ell_2 \sqsubseteq \ell_3 \) implies \( \ell_1 \sqsubseteq \ell_3 \)
  - Anti-symmetry: \( \ell_1 \sqsubseteq \ell_2 \) and \( \ell_2 \sqsubseteq \ell_1 \) implies \( \ell_1 = \ell_2 \)

- Examples:
  - Integers ordered by \( \leq \)
  - Types ordered by \( < \):
  - Sets ordered by \( \subseteq \) or \( \supseteq \)
Subsets of \{a,b,c\} ordered by \(\subseteq\)

Partial order presented as a Hasse diagram.

Height is 3

order \(\sqsubseteq\) is \(\subseteq\) meet \(\cap\) is \(\land\) join \(\cup\) is \(\lor\)
Meets and Joins

• The combining operator $\sqcap$ is called the “meet” operation.
• It constructs the greatest lower bound:
  – $\ell_1 \sqcap \ell_2 \subseteq \ell_1$ and $\ell_1 \sqcap \ell_2 \subseteq \ell_2$
    “the meet is a lower bound”
  – If $\ell \subseteq \ell_1$ and $\ell \subseteq \ell_2$ then $\ell \subseteq \ell_1 \sqcap \ell_2$
    “there is no greater lower bound”

• Dually, the $\sqcup$ operator is called the “join” operation.
• It constructs the least upper bound:
  – $\ell_1 \subseteq \ell_1 \sqcup \ell_2$ and $\ell_2 \subseteq \ell_1 \sqcup \ell_2$
    “the join is an upper bound”
  – If $\ell_1 \subseteq \ell$ and $\ell_2 \subseteq \ell$ then $\ell_1 \sqcup \ell_2 \subseteq \ell$
    “there is no smaller upper bound”

• A partial order that has all meets and joins is called a lattice.
  – If it has just meets, it’s called a meet semi-lattice.
Another Way to Describe the Algorithm

- Algorithm repeatedly computes (for each node n):
  - $\text{out}[n] := F_n(\text{in}[n])$

- Equivalently: $\text{out}[n] := F_n(\prod_{n' \in \text{pred}[n]} \text{out}[n'])$
  - By definition of in[n]

- We can write this as a simultaneous update of the vector of out[n] values:
  - let $x_n = \text{out}[n]$
  - Let $X = (x_1, x_2, \ldots, x_n)$ it’s a vector of points in $\mathcal{L}$

  - $F(X) = (F_1(\prod_{j \in \text{pred}[1]} \text{out}[j]), F_2(\prod_{j \in \text{pred}[2]} \text{out}[j]), \ldots, F_n(\prod_{j \in \text{pred}[n]} \text{out}[j]))$

- Any solution to the constraints is a fixpoint $X$ of $F$
  - i.e. $F(X) = X$
Iteration Computes Fixpoints

- Let $X_0 = (T, T, \ldots, T)$
- Each loop through the algorithm apply $F$ to the old vector:
  $X_1 = F(X_0)$
  $X_2 = F(X_1)$
  ...
- $F^{k+1}(X) = F(F^k(X))$
- A fixpoint is reached when $F^k(X) = F^{k+1}(X)$
  - That’s when the algorithm stops.

- Wanted: a maximal fixpoint
  - Because that one is more informative/useful for performing optimizations
Monotonicity & Termination

• Each flow function $F_n$ maps lattice elements to lattice elements; to be sensible is should be \textit{monotonic}:

• $F : \mathcal{L} \rightarrow \mathcal{L}$ is \textit{monotonic} iff:
  \[ \mathcal{L}_1 \sqsubseteq \mathcal{L}_2 \text{ implies that } F(\mathcal{L}_1) \sqsubseteq F(\mathcal{L}_2) \]
  – Intuitively: “If you have more information entering a node, then you have more information leaving the node.”

• Monotonicity lifts point-wise to the function: $F : \mathcal{L}^n \rightarrow \mathcal{L}^n$
  – vector $(x_1, x_2, \ldots, x_n) \sqsubseteq (y_1, y_2, \ldots, y_n)$ iff $x_i \sqsubseteq y_i$ for each $i$

• Note that $F$ is consistent: $F(X_0) \sqsubseteq X_0$
  – So each iteration moves at least one step down the lattice (for some component of the vector)
  – $\cdots \sqsubseteq F(F(X_0)) \sqsubseteq F(X_0) \sqsubseteq X_0$

• Therefore, \# steps needed to reach a fixpoint is at most the height $H$ of $\mathcal{L}$ times the number of nodes: $O(Hn)$
Building Lattices?

- Information about individual nodes or variables can be lifted pointwise:
  - If $\mathcal{L}$ is a lattice, then so is $\{ f : X \rightarrow \mathcal{L} \}$ where $f \sqsubseteq g$ if and only if $f(x) \sqsubseteq g(x)$ for all $x \in X$.

- Like types, the dataflow lattices are static approximations to the dynamic behavior:
  - Could pick a lattice based on subtyping:
  - Or other information:

- Points in the lattice are sometimes called dataflow "facts"
See HW6: Dataflow Analysis
## Def / Use for SSA

### Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>def[n]</th>
<th>use[n]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = b op c</code></td>
<td><code>{a}</code></td>
<td><code>{b, c}</code></td>
<td>arithmetic</td>
</tr>
<tr>
<td><code>a = load b</code></td>
<td><code>{a}</code></td>
<td><code>{b}</code></td>
<td>load</td>
</tr>
<tr>
<td><code>store a, b</code></td>
<td><code>Ø</code></td>
<td><code>{b}</code></td>
<td>store</td>
</tr>
<tr>
<td><code>a = alloca t</code></td>
<td><code>{a}</code></td>
<td><code>Ø</code></td>
<td>alloca</td>
</tr>
<tr>
<td><code>a = bitcast b to u</code></td>
<td><code>{a}</code></td>
<td><code>{b}</code></td>
<td>bitcast</td>
</tr>
<tr>
<td><code>a = gep b [c, d, …]</code></td>
<td><code>{a}</code></td>
<td><code>{b, c, d, …}</code></td>
<td>getelementptr</td>
</tr>
<tr>
<td><code>f(b_1, …, b_n)</code></td>
<td><code>{a}</code></td>
<td><code>{b_1, …, b_n}</code></td>
<td>call w/return</td>
</tr>
<tr>
<td><code>f(b_1, …, b_n)</code></td>
<td><code>Ø</code></td>
<td><code>{b_1, …, b_n}</code></td>
<td>void call (no return)</td>
</tr>
</tbody>
</table>

### Terminators

<table>
<thead>
<tr>
<th>Terminator</th>
<th>def[n]</th>
<th>use[n]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>br L</code></td>
<td><code>Ø</code></td>
<td><code>Ø</code></td>
<td>jump</td>
</tr>
<tr>
<td><code>br a L1 L2</code></td>
<td><code>Ø</code></td>
<td><code>{a}</code></td>
<td>conditional branch</td>
</tr>
<tr>
<td><code>return a</code></td>
<td><code>Ø</code></td>
<td><code>{a}</code></td>
<td>return</td>
</tr>
</tbody>
</table>