Administration

- Minijava project
  - Phase 5 (Flow graph and liveness)
    - due Wednesday March 28th
    - There is no contest 😞
    - but there is a correctness oracle 😊
The Plan

• Dataflow analysis
• Optimization
• Garbage collection
  – A few words on techniques
  – Integration with the compiler
• Other compiler-like things
• JIT
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”)

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

\[
\text{out}[n] = \bigcup_{s \text{ in succ}[n]} \text{in}[s]
\]
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 an unambiguous definition of \( a \)

\[
d: a \leftarrow XXX
\]

\[
a \leftarrow YYY \quad \quad \quad \quad \quad \quad a \leftarrow ZZZ
\]

\( s \) (that reads \( a \))
Reaching Definitions vs Liveness

\[ \text{in}[n] = \bigcup_{p \text{ in } \text{pred}[n]} \text{out}[p] \]  
\[ \text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \]  
\[ \text{in}[n] = \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n]) \]  
\[ \text{out}[n] = \bigcup_{s \text{ in } \text{succ}[n]} \text{in}[s] \]

- Reaching Definitions
- Liveness
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?
Show the compiler

- Look at Reaching Definitions for const.java
  run virreach sample/const.java
  run irreach sample/const.java
  run irreach sample/nonconst.java
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
What can we use Reaching Definitions for?

- Look at the compiler again
  
  run virconst sample/const.java
  run virconst sample/debug.java
  run virconst sample/nonconst.java
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
Dead code elimination

- Look at the compiler again
  - run virdead sample/const.java
  - run virdead sample/debug.java
  - run virconstdead sample/const.java
  - run virconstdead sample/debug.java
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

\[
\begin{align*}
a & \leftarrow x + y \\
\diamond & \\
\longrightarrow & \\
\longrightarrow & \\
s & \coloneqq c \leftarrow x + y
\end{align*}
\]
Available Expressions

\[ a \leftarrow x + y \]

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

\[ a \leftarrow x + y \]
\[ x \leftarrow 3 \]
\[ s : c \leftarrow x + y \]
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.
- $\text{gen}[s: t <- x \text{ OP } y] = \{x \text{ OP } y\} - \text{kill}[s]$
- $\text{kill}[s: t <- *] = \text{expressions including } t$
Available Expressions

- What are the data flow equations?
Available Expressions

- What are the data flow equations?

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

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

Because the equations use $\cap$ rather than $\cup$ we must use a different way of computing the solution to the equations.

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”.

Dataflow & Optimization (Chapter 17)
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”
Reaching Expressions

• What are the equations?

• gen & kill same as for Available Expressions
  
gen[n: t <- x OP y] = {x OP y} – kill[n]
  
kill[n: t <- *] = expressions using t, including this one

• in & out same as for Reaching Definitions

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

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