Administration

• Minijava project
  – Phase 5 (Flow graph and liveness)
    • due Wednesday March 28th (tomorrow)
    • 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
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.

• $\text{gen}[s: t \leftarrow x \text{ OP } y] = \{x \text{ OP } y\} - \text{kill}[s]$

• $\text{kill}[s: t \leftarrow \ast] = \text{expressions including } t$
Reaching Expressions

- What are the equations?
- gen & kill same as for Available Expressions
  \[ \text{gen}[n: t <- x \text{ OP } y] = \{x \text{ OP } y\} - \text{kill}[n] \]
  \[ \text{kill}[n: t <- \ast] = \text{expressions using } t, \text{ including this one} \]
- in & out same as for Reaching Definitions
  \[ \text{in}[n] = \bigcup_{p \in \text{pred}[n]} \text{out}[p] \]
  \[ \text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \]
Common Subexpression Elimination

Naïve idea

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

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

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

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

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

\[
\text{... } \quad \text{... } \quad \text{... }
\]

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

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

\[ a \oplus b \text{ is “available”} \]

Available expression analysis will tell us that \( a \oplus b \) 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:

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

... no definitions of $a$ or $b$

n: \[
\begin{align*}
r & \leftarrow z
\end{align*}
\]

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

Q: How do we know were to find the nodes \\
\[
t \leftarrow a \oplus b , t'' \leftarrow a \oplus b, ...
\]

A: Use Reaching Expressions for $a \oplus b$
Common subexpression in action

• run viravail sample/common.java
• run iravail sample/common.java
• run ircse sample/common.java

• run iravail sample/commonmem.java
• run ircse sample/commonmem.java
Copy Propagation

- Like Constant Propagation, but instead of a constant we propagate the value in a Temp
- If we have \( d: t \leftarrow z \) and \( e: y \leftarrow t \oplus x \)
- We could replace the use of \( t \) in \( e \) with the Temp that it is a copy of from \( d \).
  - i.e., replace \( e: y \leftarrow t \oplus x \) with \( e: y \leftarrow z \oplus x \)
- What are the conditions?
  - \( d \) reaches \( e \)
  - no other definition of \( t \) reaches \( e \)
  - there is no definition of \( z \) on the path from \( d \) to \( e \)
    - this one is new – constants don’t have definitions
Copy Propagation

- Why is this a good idea?
- Can make some Temps dead
- But more importantly, it can enable other optimizations like CSE
- Consider
  
  \[
  \begin{align*}
  a & \leftarrow y + z \\
  u & \leftarrow y \\
  b & \leftarrow u + z
  \end{align*}
  \]

- Is there a CSE opportunity here?
Other optimizations

- Programs spend a lot of their time in loops
- Loop optimizations:
  - Hoisting loop-invariant computations
  - Induction variable elimination
  - Strength reduction
  - Array bounds checking
  - Unrolling loops
Finding loops

- At the source language level, it is trivial
- At the IR level?
Finding loops

• At the source language level, it is trivial
• At the IR level?
• Tricky
  – All we have are jumps and cjumps
Hoisting Loop-invariant computations

```plaintext
i = 0;
k = f(...);
while (i < 10) {
    j = 5 * k;
    i = g(...j...);
}
```
Induction variables

```java
i = 0;
while (i < 10) {
    j = 5 * i;
    f(... j ...);
    i = i + 1;
}
```
Strength reduction

```c
i = 0;
while (i < 10) {
    j = 5 * i;
    f(... j ...);
    i = i + 1;
}
```
Array bounds checks

```java
i = 0;
j = 0;
while (i < a.length) {
    j = j + a[i];
i = i + 1;
}
```
Loop unrolling

```java
i = 0;
j = 0;
while (i < a.length) {
    j = j + a[i];
i = i + 1;
}
```