CS 473: COMPILER DESIGN
Midterm 2 Questions

Top
Dataflow Analysis: Common Features

- All of these analyses have a *domain* over which they solve constraints
  - Liveness: sets of variables
  - Reaching defns., available exprs.: 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 propagates information either *forward* or *backward*
  - Forward: in[n] defined in terms of out[n’] for predecessors n’
  - Backward: out[n] defined in terms of in[n’] for successors n’
- Each analysis has a way of combining information
  - Liveness & reaching definitions take union (∪)
  - Available expressions uses intersection (∩)
  - Union expresses a property that holds for *some* path (existential)
  - Intersection expresses a property that holds for *all* paths (universal)
Dataflow Analysis: Example

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

Iteration 1:

\[
\text{in}[n] = \emptyset \\
\text{out}[n] = \text{gen}[n]
\]

I.e., label each node with the definitions it generates
Dataflow Analysis: Example

\[
\begin{align*}
\text{in}[n] & := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \\
\text{out}[n] & := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\end{align*}
\]

Iteration 2:

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

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

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

Iteration 2:

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

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

Iteration 2:

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

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

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

Iteration 2:

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

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

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

Iteration 2:

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

\[
\begin{align*}
in[n] & := \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \\
\text{out}[n] & := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\end{align*}
\]

Iteration 3:

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

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

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

Iteration 3:

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

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

Iteration 3:

Apply the equations

1. Data flows from out to in
2. Data flows through \( n \) if it is not in \( \text{kill}[n] \)
Dataflow Analysis: Example

\[
\text{in}\[n\] := \bigcup_{n' \in \text{pred}\[n\]} \text{out}\[n'\]\\
\text{out}\[n\] := \text{gen}\[n\] \cup (\text{in}\[n\] \setminus \text{kill}\[n\])
\]

Iteration \(k\):

Apply the equations

1. Data flows from out to in
2. Data flows through \(n\) if it is not in \(\text{kill}\[n\]\)

3 is another definition of \(t\), so it is in \(\text{kill}[6]\) and won’t flow through 6

1 is another definition of \(i\), so it is in \(\text{kill}[5]\) and won’t flow through 5
Questions

Top
OPTIMIZING LOOPS
Loop Analysis

Loop Nesting Tree:

Natural Loops
Now that we have the loop nesting tree, we can optimize loops!

1. Start from the bottom of the nesting tree
2. Look for ways to improve performance of the loop
3. Move up the tree until all loops have been optimized
4. Apply non-loop optimizations
5. Repeat from step 1

Stop when we reach a fixed point, or feel like we’ve done enough
Loop-Invariant Code Motion

• Suppose we have a loop:

\[
\begin{align*}
\text{x < 10} \\
\text{y = a + b} \\
\text{z = x + y} \\
\text{x = x + 1}
\end{align*}
\]

\[
\begin{align*}
\text{y = a + b} \\
\text{x < 10} \\
\text{z = x + y} \\
\text{x = x + 1}
\end{align*}
\]

• When is this correct? When a and b don’t change in the loop.
Loop-Invariant Code Motion

• Find statements in a loop body that don’t change over iterations (the value is *invariant*) and move them before the loop (also called *hoisting*)

• Safe to hoist $y = a + b$ when either:
  – $a$ and $b$ are constants/defined outside the loop
  – $a$ and $b$ are defined inside the loop, but have the same value in every iteration

• Not safe if:
  – there are multiple definitions of $y$ in the loop
  – $y$ isn’t always defined in the loop
  – we use the old value of $y$ in the loop before defining it

• We can check all these conditions with dataflow analyses like the ones we’ve already seen
Questions

Top
• Suppose we have a loop:

\[
\begin{align*}
\text{x} &< 10 \\
y &\equiv x \times n \\
z &\equiv x + y \\
x &\equiv x + 1
\end{align*}
\]

• When is this correct? We need to know that \(x\) is the *induction variable* (basically, the iterator) for the loop.
Loop Induction Variables

• Many loops are for loops (or for-loop-like): an iterator variable is incremented/decremented by the same amount in every iteration.

• Often, other instructions in a for loop will use the value of the iterator in a regular way:

\[
\text{for}(i = 0; i < 5; i++)\
\begin{align*}
& a[i + 2] = 10; \\
& j = 4 \times i + 8 \\
& \text{if } (i \geq 5) \text{ goto done} \\
& \text{store}(10, a + j) \\
& i = i + 1
\end{align*}
\]

• We call \( i \) a \textit{basic induction variable}, and \( j \) a \textit{derived induction variable} in the \textit{family} of \( i \).

• In general, a derived induction variable will be some linear function of the base: \( j = a \times i + b \), where \( a \) and \( b \) are constants.
i = 0
if (i >= 5) goto done
j = 4 * i + 8
store(10, a + j)
i = i + 1

• We call i a *basic induction variable*, and j a *derived induction variable* in the *family* of i

• In general, a derived induction variable will be some linear function of the base: $j = a * i + b$, where $a$ and $b$ are constants

• How do we identify basic and derived induction variables in a loop?
  – A basic induction variable is only modified by $i = i + c$ (or $i = i - c$)
  – A derived induction variable is modified by $j = k + c$ or $j = k * c$, where $k$ is itself an induction variable (either basic or derived)
Questions

Top
Strength Reduction

• Once we know all the basic and derived induction variables, we can turn the multiplications in derived induction variables into additions

```plaintext
i = 0
if (i >= 5) goto done
j = 4 * i + 8
store(10, a + j)
i = i + 1
j' = j' + 4 // increment of i (1) * coefficient of i in j (4)
```
Strength Reduction

• Once we know all the basic and derived induction variables, we can turn the multiplications in derived induction variables into additions

\[
i = 0
\]
\[
j' = 8 \quad // 4 \times 0 + 8
\]
\[
\text{if } (i >= 5) \text{ goto done}
\]
\[
j = 4 \times i + 8
\]
\[
\text{store}(10, a + j)
\]
\[
i = i + 1
\]
\[
j' = j' + 4 \quad // \text{increment of } i \ (1) \times \text{coefficient of } i \text{ in } j \ (4)
\]
Strength Reduction

• Once we know all the basic and derived induction variables, we can turn the multiplications in derived induction variables into additions

\[
i = 0 \\
j' = 8 \quad // \quad 4 \times 0 + 8 \\
\text{if } (i \geq 5) \text{ goto done} \\
j = j' \\
\text{store}(10, a + j) \\
i = i + 1 \\
j' = j' + 4 \quad // \text{ increment of } i \ (1) \times \text{ coefficient of } i \text{ in } j \ (4)\]
Strength Reduction

• Once we know all the basic and derived induction variables, we can turn the multiplications in derived induction variables into additions

\[
i = 0 \\
j' = 8 \quad // \ 4 \times 0 + 8 \\
\text{if } (i \geq 5) \text{ goto done} \\
j = j' \\
\text{store}(10, a + j) \quad \text{store}(10, a + j') \\
i = i + 1 \\
j' = j' + 4 \quad // \text{ increment of } i \ (1) \times \text{ coefficient of } i \text{ in } j \ (4)
\]

• Now we won’t emit any multiplication instructions!

• But we can go further: now we don’t need i anymore!
i = 0
j' = 8       // 4 * 0 + 8
if (i >= 5) goto done
store(10, a + j')
i = i + 1
j' = j' + 4   // increment of i (1) * coefficient of i in j (4)
i = 0
j' = 8       // 4 * 0 + 8
if (i >= 5) goto done       if (j' >= 28) goto done       // 4 * 5 + 8 = 28
store(10, a + j')
i = i + 1
j' = j' + 4       // increment of i (1) * coefficient of i in j (4)
Induction Variable Elimination

\[ i = 0 \]
\[ j' = 8 \quad \text{// } 4 * 0 + 8 \]

if (i >= 5) goto done \quad \text{// } 4 * 5 + 8 = 28

store(10, a + j')

\[ i = i + 1 \]
\[ j' = j' + 4 \quad \text{// increment of } i \text{ (1) * coefficient of } i \text{ in } j \text{ (4)} \]

- Only valid if we don’t use the value of \( i \) afterwards!
Induction Variable Elimination

\[
i = 0 \\
\text{if } (i \geq 5) \text{ goto done} \\
j = 4 \times i + 8 \\
\text{store}(10, a + j) \\
i = i + 1 \\
\]

became

\[
j' = 8 \\
\text{if } (j' \geq 28) \text{ goto done} \\
\text{store}(10, a + j') \\
j' = j' + 4 \\
\]

• Combined \(i\) and \(j\) into a single variable without the multiplication
Loop Unrolling

• Suppose we have a loop:

- x < 10
- y = z * x
- z = x + y
- x = x + 1

- x < 10
- y = z * x
- z = x + y
- y = z * (x + 1)
- z = (x + 1) + y
- x = x + 2

• When is this correct? When the number of iterations is always even.
• We can unroll 4, 8, ... times, as appropriate
• Suppose we have a loop:

\[
\begin{align*}
&x < 9 \\
y &= z \times x \\
z &= x + y \\
x &= x + 1
\end{align*}
\]

• We might need to add an epilogue that finishes off the odd/leftover iterations:

\[
\begin{align*}
x < 8 \\
\ldots \\
x &= x + 2 \\
y &= z \times x \\
z &= x + y \\
x &= x + 1
\end{align*}
\]
Questions

Top
Loop Optimizations

• Instructions in loops are executed many more times than instructions outside of loops, so it’s worth taking extra effort to optimize them

• We can speed up loops by paying attention to what changes, and finding things that are either unchanged (invariant code motion) or changed in regular ways (induction variables, strength reduction)

• What’s actually efficient depends on the target processor!
  – For instance, if the processor can do multiplication as efficiently as addition, strength reduction is a lot of work for nothing

• We have to watch out for corner cases to make sure our optimizations don’t change the behavior of the code