CS 473: COMPILER DESIGN
Multiple IRs

• Goal: get program closer to machine code without losing the information needed to do analysis and optimizations

• In practice, multiple intermediate representations might be used (for different purposes)

- AST
- HIR
- MIR
- x86
- Java Bytecode
- ARM

Optimization
Multiple IRs

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
- In practice, multiple intermediate representations might be used (for different purposes)
start:
[b]
Jmp start
done:
IR 2: Block-Based IR

- **Goal:** linear sequence of statements
  - No expression sequences, no calls inside other expressions, Cjmp fall-through
- **Same expressions and statements as tree IR, but a program is a list of statements instead of a tree**
- **Statements grouped into basic blocks with jumps at the end**

```java
if b then a[i] := 0

Cjmp

b
Move
Mem
0

label2:

Cjmp (not b) label2
Move (Mem (fp + 1)) 0
```

```java
label2:
```
Homework 3 Questions

Top
\[ p.z.x \quad m = \{x : \text{int}, y : \text{int}, z : m\} \]
Does p.z.x mean (p.z).x or p.(z.x)?

(p.z).x

p.(z.x)
Tree IR: Arrays and Records

\[ p.z.x \quad m = \{x : \text{int}, y : \text{int}, z : m\} \]
\[
p.z.x \quad m = \{x : \text{int}, y : \text{int}, z : m\}
\]
p.z.x  \quad m = \{x : \text{int}, y : \text{int}, z : m\}
Homework 3 Questions

Top
write(" ")

```
Call
  g
  args
```

```
Call
  g
  [args]
  sl
```

```
Call
  write
  [" "]
  sl
```
write(" ")

Tree IR: Function Calls

- Call
  - write
  - [" "]
  - sl

- String
  - "123"

- String
  - a
  - 123
write(" ")
Homework 3 Questions

Top
IR 2: Block-Based IR

- Translate in 3 steps:
  1. Linearize: get rid of Seq nodes, convert to list of statements
  2. Group: group into blocks of straight-line code ("basic blocks")
  3. Schedule: set up fall-throughs for Cjmps

```
Cjmp (not b) label2
Move (Mem (fp + 1)) 0

label2:
```
1. Linearize: get rid of Seq nodes, convert to list of statements

In Tiger: \( y := (x := x + 1; x) \)
In C: \( y = (x = x + 1); \)
same as \( (x := x + 1; y := x) \)

- Rewrite trees to move Seqs to top

```
  Move
  \( y \)
  Seq
    Move \( x \)
      Move \( x \)
        Plus \( x \)
          x
          1
  Seq
    Move \( x \)
      Plus \( x \)
        x
        1
    Move \( y \)
      y
      x
```
Block IR: Expression Sequences

- Rewrite trees to move $\text{Seqs}$ to top
- Move $\text{Seqs}$ up until they’re not below any non-$\text{Seq}$ node
(x := x + 1; x) + y
\[ y + (x := x + 1; x) \]
$x + (x := x + 1; x)$

$\{x = 3\}$
\[ x + (x := x + 1; x) \iff (\text{temp} := x; x := x + 1; \text{temp} + x) \]
\{ x = 3 \}
\( x + (x := x + 1; x) \quad \iff \quad (\text{temp} := x; x := x + 1; \text{temp} + x) \)

\( \{x = 3\} \)
Block IR: Evaluation Order

\[(s; e_1) + e_2 \implies (s; e_1 + e_2)\]

\[e_1 + (s; e_2) \implies (\text{temp} = e_1; s; \text{temp} + e_2)\]

\[e_1 + (s; e_2) \implies (s; e_1 + e_2) \text{ only when } s \text{ and } e_1 \text{ commute}\]

- A statement *commutes* with an expression when running the statement doesn’t change the behavior of the expression.
- In general, we can’t always tell whether two things commute, so we need to be *conservative*.
Questions

Top
(s; e₁) + e₂ => (s; e₁ + e₂)

e₁ + (s; e₂) => (temp = e₁; s; temp + e₂)

e₁ + (s; e₂) => (s; e₁ + e₂) only when s and e₁ commute

- Move Seqs up until they’re not below any non-Seq node
- In general, recognizing patterns in graphs is tricky, but in this case we’re only looking for Seq nodes
1. Recurse through tree looking for Seq nodes
2. Each time we find a Seq node \((s; e)\), change it to \(e\) and add \(s\) to a list of statements
3. When we move up past a node with multiple subexpressions (like BinOp), add new temp and statement to preserve evaluation order (unless statement and expression commute)
4. Once we reach the root, put the whole list of statements before it
1. Recurse through tree looking for Seq nodes

statements:
1. Recurse through tree looking for Seq nodes

```
Plus
x
Seq
Move
x
Plus
x
1
```

statements:
2. Each time we find a Seq node \((s; e)\), change it to \(e\) and add \(s\) to a list of statements

statements:
3. When we move up past a node with multiple subexpressions, add new temp and statement to preserve evaluation order (unless statement and expression commute)

statements:
3. When we move up past a node with multiple subexpressions, add new temp and statement to preserve evaluation order (unless statement and expression commute)

statements:
4. Once we reach the root, put the whole list of statements before it.
f(x) + g(y)

f(x)  =>  (temp = f(x); temp)
Block IR: Function Calls

f(x) + g(y)  =>  (temp1 = f(x); temp2 = g(y); temp1 + temp2)

f(x)  =>  (temp = f(x); temp)

(z := f(x) + g(y); z) =>
(z := (...; ...; temp1 + temp2); z) =>
(...; ...; z := temp1 + temp2; z)
Block IR: Linearize

\[ x := (s; e) \Rightarrow (s; x := e) \]

\[ (s; e_1) + e_2 \Rightarrow (s; e_1 + e_2) \]

\[ e_1 + (s; e_2) \Rightarrow (temp = e_1; s; temp + e_2) \]

\[ e_1 + (s; e_2) \Rightarrow (s; e_1 + e_2) \text{ only when } s \text{ and } e_1 \text{ commute} \]

\[ f(x) \Rightarrow (temp = f(x); temp) \]

- Move \text{Seq}s up until they’re not below any non-\text{Seq} node
- When the whole program is one big \text{Seq}, we don’t need trees anymore – it’s just an assembly-style list of statements
Block IR: Linearize

- Move\(\text{Seqs}\) up until they’re not below any non-\(\text{Seq}\) node
- When the whole program is one big \(\text{Seq}\), we don’t need trees anymore – it’s just an assembly-style list of statements

Tree representation:

```
Seq
  Move
    old_x
    x
  Seq
    Move
      x
      Plus
        x
        1
    Plus
      old_x
      x
```

Transformed statements:

\[
\text{Move}(\text{old}_x, \ x) \\
\text{Move}(\ x, \ x + 1) \\
\text{old}_x + \ x
\]
Questions

Top
2. Group: group into blocks of straight-line code ("basic blocks")

- A *basic block* has the form:
  
  label:
  ...
  // straight-line code, no jumps or labels
  ...
  Jump/Cjump

- This lets us deal with code in sections instead of individual statements, and is useful for *control flow* analysis
2. Group: group into blocks of straight-line code ("basic blocks")

- A basic block has the form:

```plaintext
label:
...
// straight-line code, no jumps or labels
...
Jump/Cjump
```

```
stmt1
stmt2
...
stmtn
```
2. **Group**: group into blocks of straight-line code ("basic blocks")

- A *basic block* has the form:

  ```
  label:
  ...
  // straight-line code, no jumps or labels
  ...
  Jump/Cjump
  ```

  ```
  new_label1:
  stmt1
  stmt2
  ...
  stmt_n
  ```
2. Group: group into blocks of straight-line code (“basic blocks”)

- A basic block has the form:

```plaintext
label:
... // straight-line code, no jumps or labels
...
Jump/Cjump

new_label1:
stmt1
stmt2
Jump/Cjump
...
```
2. Group: group into blocks of straight-line code ("basic blocks")

- A *basic block* has the form:

```
label:
... // straight-line code, no jumps or labels
...
Jump/Cjump
```

```
new_label1:
stmt1
stmt2
Jump/Cjump
```
2. Group: group into blocks of straight-line code ("basic blocks")

- A basic block has the form:
  
  ```
  label:
  ...
  // straight-line code, no jumps or labels
  ...
  Jump/Cjump
  ```

  ```
  new_label1:
  stmt1
  stmt2
  start:  
  ...
  ```
2. Group: group into blocks of straight-line code ("basic blocks")

• A basic block has the form:

```plaintext
label:
...
// straight-line code, no jumps or labels
...
Jump/Cjump

new_label1:
stmt1
stmt2
Jump start

start:
...
```


2. Group: group into blocks of straight-line code ("basic blocks")

- A *basic block* has the form:
  
  `label:`
  
  `...  // straight-line code, no jumps or labels`
  
  `...`

- **Algorithm:** go through code statement by statement
- If first statement isn’t a label, insert a new label
- If we reach a jump, end block and start new block
- If we reach a label, add jump to that label and start new block
Questions

Top
3. Schedule: set up fall-throughs for Cjumps

- A basic block has the form:

```
label:
...
// straight-line code, no jumps or labels
...
Jump/Cjump
```

- So we can rearrange the order of the basic blocks without changing the program’s behavior at all!
3. Schedule: set up fall-throughs for Cjmps

label:
... // straight-line code, no jumps or labels
...
Cjump LT a 0 ltrue lfalse

ltrue:
Move(x, 1)

lfalse:
Move(x, 0)
3. Schedule: set up fall-throughs for Cjumps

label:
...
// straight-line code, no jumps or labels
...
Cjump LT a 0 ltrue lfalse

lfalse:
Move(x, 0)

ltrue:
Move(x, 1)
3. Schedule: set up fall-throughs for Cjumps

label:
...  // straight-line code, no jumps or labels
...
Cjump LT a 0 ltrue

lfalse:
Move(x, 0)

ltrue:
Move(x, 1)
3. Schedule: set up fall-throughs for Cjmps

label:
...
// straight-line code, no jumps or labels
...
Jump label2

label1:
code1

label2:
code2
3. Schedule: set up fall-throughs for Cjumps

```
label:
...
  // straight-line code, no jumps or labels
...
Jump label2

label2:
code2

label1:
code1
```
3. Schedule: set up fall-throughs for Cjumps

label:
...
  // straight-line code, no jumps or labels
...

label2:
code2

label1:
code1
Questions

Top
3. Schedule: set up fall-throughs for Cjumps

- A basic block has the form:
  
```plaintext
label:
... // straight-line code, no jumps or labels
...
Jump/Cjump
```

- So we can rearrange the order of the basic blocks without changing the program’s behavior at all!

- First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block
• First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block
First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block.

```
Cjump l3 l4
```
Block IR: Scheduling

- First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block
- Second pass: fix any \texttt{Cjump}s that still aren’t followed by their false label

\begin{align*}
l1: & \quad \text{Cjump LT a 0 l3 l4} & \quad \text{Cjump GE a 0 l4 l3} \\
l3: & \quad \phantom{\text{Cjump LT a 0 l3 l4}} \\
\end{align*}
Block IR: Scheduling

• First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block

• Second pass: fix any \( \text{Cjump} \)s that still aren’t followed by their false label

\[
\begin{align*}
\text{l1:} & \quad \text{Cjump LT a 0 l4 l5} \\
\text{l3:} & \quad \text{} \\
\text{l6:} & \quad \text{Jump l5} \\
\text{l3:} & \quad \text{}
\end{align*}
\]

• This might add extra code, but we need it in order to get rid of the false labels!
Block IR: Scheduling

- First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block.
- Second pass: fix any Cjumps that still aren’t followed by their false label.
  - This might add extra code, but we need it in order to get rid of the false labels!
- Third pass: get rid of all the false labels of Cjumps, and any Jumps that go straight to the next block.

- Now we can just *fall through* to the next block when a basic block ends with a non-jump statement or false Cjump.
Questions

Top