# Intermediate Representations

CS 132: Compiler Design

Monday, February 23, 2011

# Types of Intermediate Representations

- ✓ Graphical
  - Trees
  - Graphs

- ✓ Linear
  - E.g., real or idealized assembly code

- ✓ In-Between (*Hybrid*)
  - E.g., graphs of linear basic blocks
  - Combination of linear code + graphical information

# Definition: Basic Block

*A maximal sequence of instructions that is only entered at the first instruction and that may leave the sequence at the last instruction*

# Types of Intermediate Representations

- ✓ High-Level
  - Abstract primitives (invoking a virtual method, creating tuples, ...)
  - Structured control
- ✓ Low-Level
  - Assembly-level and/or hardware specific operations
  - Data representations exposed
- ✓ In-Between
  - Mostly low-level operations, plus a few "powerful" primitives
    - E.g., invoke a method, throw/catch and exception

# Advantages of High-Level/Low-Level Representations

```
val a = (1,3)
val b = (1,3)
```

---

```
a = malloc(8)
*a = 1
*(a+4) = 3
b = malloc(8)
*b = 1
*(b+4) = 3
```

```
val a = (1,3)
val b = (5,7)
```

---

```
a = malloc(8)
*a = 1
*(a+4) = 3
b = malloc(8)
*b = 5
*(b+4) = 7
```

# Linear Representations

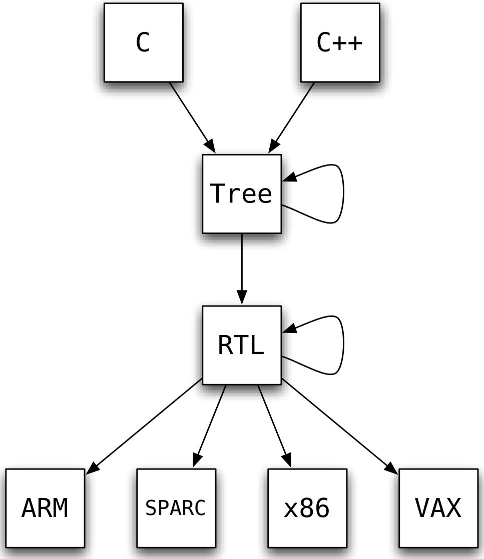
- ✓ Stack code (JVM, CIL, . . .)
- ✓ Three-address code (a.k.a. RTL, Quadruples)
  - Pseudo-assembly: very simple operations
  - Arbitrarily many temporaries
  - Conditional jumps to labels
  - Primitives for call/return
  - Memory load, stores
- ✓ Assembly language (x86, PowerPC, . . .)

```
  i ← 1
  j ← 1
  k ← 0
L1:
  if k >= 100 goto done
  if j >= 20 goto L2
  j ← i
  k ← k + 1
  goto L3
L2:
  j ← k
  k ← k + 2
L3:
  j ← j - 1
  goto L1
```

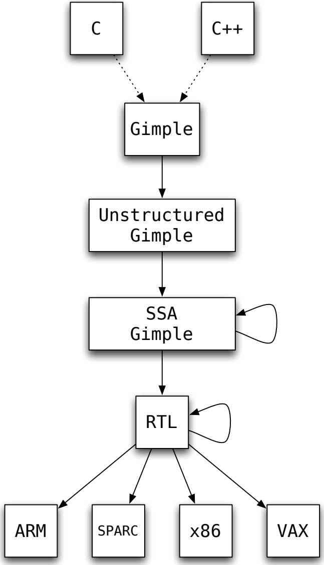
# Graphical Representations

- ✓ Parse trees
- ✓ Abstract syntax trees
  - Source language
  - Intermediate languages
  - Assembly language
- ✓ Directed Acyclic Graphs
  - Express sharing among subexpressions
- ✓ Graphs
  - Control flow: nodes are expressions or basic blocks, arrows if control can pass from one node to the other.
  - Dependence: nodes are expressions, arrows if a value created by the source will be used by the target.
  - ...

# GCC 3.X



# GCC 4.X


C
C++
Gimple
Unstructured Gimple
SSA Gimple
RTL
ARM
SPARC
x86
VAX

# GENERIC and GIMPLE

| GENERIC | High GIMPLE | Low GIMPLE |
| --- | --- | --- |
| `if (foo (a + b,c))` | `t1 = a + b` | `t1 = a + b` |
| `  c = b++ / a` | `t2 = foo (t1, c)` | `t2 = foo (t1, c)` |
| `endif` | `if (t2 != 0)` | `if (t2 != 0) <`**`L1,L2`**`>` |
| `return c` | `  t3 = b` | **`L1:`** |
| | `  b = b + 1` | `t3 = b` |
| | `  c = t3 / a` | `b = b + 1` |
| | `endif` | `c = t3 / a` |
| | `return c` | `goto `**`L3`** |
| | | **`L2:`** |
| | | **`L3:`** |
| | | `return c` |

# Static Single Assignment

Idea: *every variable is written to exactly once in the code*

```
j ← 0
k ← 2
k ← k + 1
j ← j + k
```

# Static Single Assignment

Idea: *every variable is written to exactly once in the code*

| | |
|---|---|
| j ← 0 | j1 ← 0 |
| k ← 2 | k1 ← 2 |
| k ← k + 1 | k2 ← k1 + 1 |
| j ← j + k | j2 ← j1 + k2 |

Advantage: Some optimizations are easier/more efficient
Difficulties?

# Draw the control-flow graph; convert to SSA

```
i ← 1; j ← 1; k ← 1;
while (k < 100) do {
  j ← j - 1
  if (j < 20) then {
     j ← i
     k ← k+1
  } else {
     j ← k
     k ← k+2
  }
}
return;
```

# Phi Functions to the Rescue!

Assume a control-flow node has $n$ incoming arrows.

# PHI FUNCTIONS TO THE RESCUE!

Assume a control-flow node has $n$ incoming arrows.

$$\phi(x_1, \ldots, x_n) := x_i \text{ where we entered via the i-th arrow}$$

# Phi Functions to the Rescue!

Assume a control-flow node has $n$ incoming arrows.

$$\phi(x_1, \ldots, x_n) := x_i \text{ where we entered via the i-th arrow}$$

Construct an SSA Control-Flow graph for the last code sequence.

# Trivially Easy SSA Optimizations

1. We can eliminate the definition $x_i \leftarrow c$ (where $c$ is a constant) by …
2. We can eliminate the definition $x_i \leftarrow y_j$ (where $y_j$ is a variable) by …
3. An assignment $x_i \leftarrow e$ can be eliminated if …

# Implementing SSA

Remaining issues

1. How can we actually implement code written with $\phi$ functions?

# Implementing SSA

Remaining issues

1. How can we actually implement code written with $\phi$ functions?

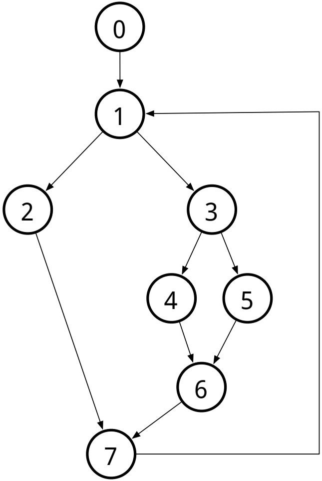
2. How can we place the $\phi$ functions efficiently?

# Dominators

In a CFG, $B_1$ **dominates** $B_2$ if every path from the start of the code to $B_2$ must pass through $B_1$.

# DOMINATORS

In a CFG, $B_1$ **dominates** $B_2$ if every path from the start of the code to $B_2$ must pass through $B_1$.



# Dominators

In a CFG, $B_1$ **dominates** $B_2$ if every path from the start of the code to $B_2$ must pass through $B_1$.

Dataflow equation:

$$\text{dom}(n) =$$

# Dominators

In a CFG, $B_1$ **dominates** $B_2$ if every path from the start of the code to $B_2$ must pass through $B_1$.

Dataflow equation:

$$\mathrm{dom}(n) = \{n\} \cup \Big( \bigcap_{m \in \mathrm{pred}\, n} \mathrm{dom}(m) \Big)$$

# Dominators

In a CFG, $B_1$ **dominates** $B_2$ if every path from the start of the code to $B_2$ must pass through $B_1$.

Dataflow equation:

$$\text{dom}(n) = \{n\} \cup \Big( \bigcap_{m \in \text{pred}\, n} \text{dom}(m) \Big)$$

Initial conditions: $\text{dom}(\textit{entry}) = \{\textit{entry}\}$, all other sets as *large* as possible. Why?

# Dominators

In a CFG, $B_1$ **dominates** $B_2$ if every path from the start of the code to $B_2$ must pass through $B_1$.

Dataflow equation:

$$\mathrm{dom}(n) = \{n\} \cup \Big( \bigcap_{m \in \mathrm{pred}\, n} \mathrm{dom}(m) \Big)$$

Initial conditions: $\mathrm{dom}(\mathit{entry}) = \{\mathit{entry}\}$, all other sets as *large* as possible. Why?

Efficiency improvement: work with **immediate dominators**.

# Using Dominance Frontiers

The **dominance frontier** $\mathrm{DF}(n)$ of a node $n$ is the set of nodes $m$ such that

1. $n$ dominates an immediate predecessor of $m$
2. $n$ does not strictly dominate $m$

# Using Dominance Frontiers

The **dominance frontier** DF$(n)$ of a node $n$ is the set of nodes $m$ such that

1. $n$ dominates an immediate predecessor of $m$
2. $n$ does not strictly dominate $m$

Intuitively, look at all the paths going out of $n$; as soon as you find a node no longer strictly dominated by $n$, you're in the frontier.

# Using Dominance Frontiers

The **dominance frontier** $\text{DF}(n)$ of a node $n$ is the set of nodes $m$ such that

1. $n$ dominates an immediate predecessor of $m$
2. $n$ does not strictly dominate $m$

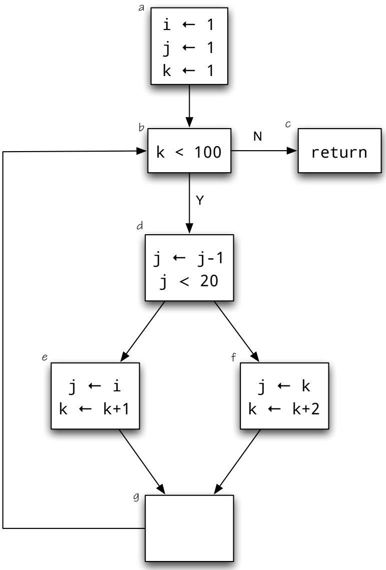
Intuitively, look at all the paths going out of $n$; as soon as you find a node no longer strictly dominated by $n$, you're in the frontier.

A definition of $x$ in node $b$ forces a $\phi$-function for $x$ in every element of $\text{DF}(b)$.

$$x = \phi(x, \ldots, x)$$

These new $\phi$ functions act as new definitions, so we have to iterate this process. When done, go through and rename.

# EXAMPLE



# Optional Pruning

The dominance-frontier method produces "minimal SSA"

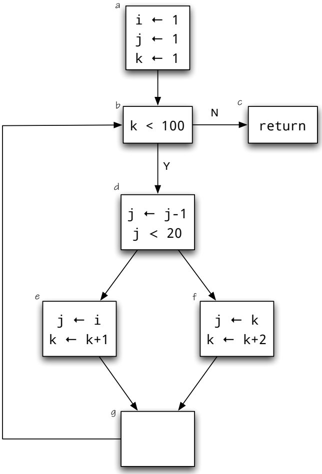
- ✓ $\phi$-functions exactly where

But we could prune further:

- ✓ If a variable is dead in a block, don't insert a $\phi$ function there.
- ✓ Even more simply (but less optimally), only insert $\phi$ functions that are live at the end of *some* basic block.

# A Simpler Algorithm: "Maximal SSA"

- ✓ Put a $\phi$ function for every variable at the start of every block!
- ✓ With care, we can add $\phi$ functions and do all renaming in a single pass.



# Optimizing Maximal SSA

Some $\phi$ functions can be easily removed

- ✓ $v_i \leftarrow \phi(v_i, \ldots, v_i)$ (trivially)
- ✓ $v_i \leftarrow \phi(v_i, v_j, v_i, v_j, v_j)$ (via subsitution)

Theorem: Repeatedly remove such $\phi$ functions. If the graph is *reducible*, the final result will be minimal SSA.

## Other Representations: $\lambda$-calculus based

```
i ← 1; j ← 1; k ← 1;
while (k < 100) do {
  j ← j - 1
  if (j < 20) then {
     j ← i
     k ← k+1
  } else {
     j ← k
     k ← k+2
  }
}
return;
```

# Other Representations: λ-calculus based

```
i ← 1; j ← 1; k ← 1;
while (k < 100) do {
  j ← j - 1
  if (j < 20) then {
     j ← i
     k ← k+1
  } else {
     j ← k
     k ← k+2
  }
}
return;
```

```
let
   f(i,j2,k2) =
     if (k2 < 100) then
       let j3 = j2 - 1
       in if (j3 < 20) then
            let j4 = i
                k3 = k2+1
            in f(i,j4,k3) end
          else
            let j5 = k2
                k4 = k2+2
            in f(i,j5,k4) end
     else
       (i,j2,k2)
in
   f(1,1,1)
```

# Other Representations: Continuation-Passing Style

```
fib n =
 if (n == 0) then
    0
 else if (n == 1) then
    1
 else
    fib (n-1) + fib (n-2)
```

# Other Representations: Continuation-Passing Style

```
fib n =
 if (n == 0) then
    0
 else if (n == 1) then
    1
 else
    fib (n-1) + fib (n-2)
```

```
fib n k =
 if (n == 0) then
    k 0
 else if (n == 1) then
    k 1
 else
    fib (n-1) (\x ->
      fib (n-2) (\ y ->
        k (k+y)))
```