My Background

- UC Berkeley PhD - compiler for reconfigurable computing
  - one-person compiler team - not recommended
  - IR: SUIF plus home-brew graph
- SRC Computers - compiler for Pentium + Xilinx FPGA
  - IR: Commercial front end plus home-brew graph
- CMU (Phoenix) - compiler for spatial computing
  - IR: SUIF plus home-brew graph: Pegasus

Outline

- IRs
- Graphs
- Control flow
  - Dominators
- Local data flow
- Global data flow
  - Reaching Definitions
  - Liveness
- Task 0

IRs (Intermediate Representations)

- Typically starts similar to source language, and eventually gets "lowered" to something close to assembly.
- Contains all semantic information necessary to execute the program
- Even hello_world() is messier than you think...
- Always remember what is your essential IR, versus what are the auxiliary / side data structures...
IRs (Intermediate Representations)

- Starting point: AST (abstract syntax tree)
  - typically retains artifacts from source language
  - see SUIF hierarchy for example

--

IRs (Intermediate Representations)

- Dismantle high-level control structures to get control flow graph
  - are we losing something by throwing away this information?
- Basic blocks contain lists of statements:
  - LHS is a variable
  - RHS is an expression tree

\[
\begin{align*}
A & = (B+C)-(D<<2); \\
D & = A*3;
\end{align*}
\]

--

IRs (Intermediate Representations)

- Even lower: tuples: one operation
  - Introduce "compiler temporaries", or "pseudoregisters"

\[
\begin{align*}
A & = (B+C)-(D<<2); \\
D & = A*3;
\end{align*}
\]

--

IRs (Intermediate Representations)

- But you can go in the other direction too - build up the DAG for each basic block:

\[
\begin{align*}
A & = (B+C)-(D<<2); \\
D & = A*3; \\
T1 & = B+C; \\
T2 & = D<<2; \\
A & = T1 - T2; \\
D & = A \ast 3;
\end{align*}
\]
IRs (Intermediate Representations)

- But you can go in the other direction too – build up the DAG for each basic block:

\[
\begin{align*}
A &= (B+C)-(D<<2); \\
D &= A*3; \\
E &= B + C;
\end{align*}
\]

Control Flow Graph (CFG)

- One per procedure
- Special Entry and Exit nodes
- Dominators:
  \[ x \text{ dom } y \text{ if every possible execution path from the entry to } y \text{ includes } x. \]
- Reflexive: \( x \text{ dom } x \)

Computing Dominators

- One per procedure
- Special Entry and Exit nodes
- Dominators:
  \[ x \text{ dom } y \text{ if every possible execution path from the entry to } y \text{ includes } x. \]
- Reflexive: \( x \text{ dom } x \)

Computing dominators

- Initialize: \( 0 \{0\}, \text{ rest } * \)
- Meet function: intersect
- Transfer function: add self
- Iterate until no change…
Computing dominators

- Initialize:
  - $\emptyset \{ 0 \}$, rest $\{ \ast \}$
  - Meet function: intersect
  - Transfer function: add self
  - Iterate until no change…

- What if we had initialized to empty sets?
Computing dominators

- Initialize:
  - Start with \{0\}, rest \{*\}
  - Meet function: intersect
  - Transfer function: add self
  - Iterate until no change...

- Diagram:
  - Nodes labeled with sets of values
  - Edges show relationships between nodes

- Iterates to:
  - \{0,1\}
  - \{0,1,2\}
  - \{0,1,2,3\}
  - \{0,1,2,4\}