15-745 Lecture 2

Tim Callahan
CMU

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 auxilliary / side data structures…

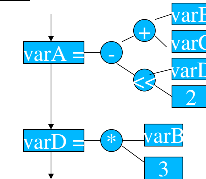
IRs (Intermediate Representations)

- 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

```
A = (B+C)-(D<<2);
D = A*3;
```



IRs (Intermediate Representations)

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

```
A = (B+C)-(D<<2);
D = A*3;
```

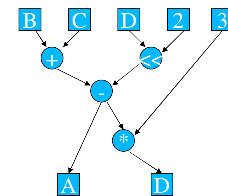
```
T1 = B+C;
T2 = D<<2;
A = T1 – T2;
D = A * 3;
```


IRs (Intermediate Representations)

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

```
A = (B+C)-(D<<2);
D = A*3;
```

```
T1 = B+C;
T2 = D<<2;
A = T1 – T2;
D = A * 3;
```

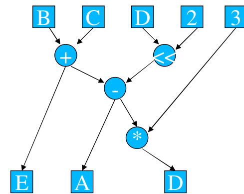


IRs (Intermediate Representations)

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

A = (B+C)-(D<<2);
D = A*3;
E = B + C;

T1 = B+C;
T2 = D<<2;
A = T1 – T2;
D = A * 3;
E =B + C;



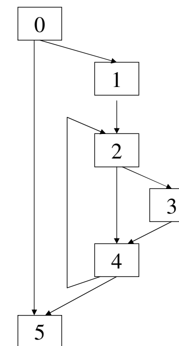
Control Flow Graph (CFG)

- One per procedure
- Special Entry and Exit nodes
- Dominators:
  x dom y if every possible execution path from the entry to y includes x.
- Reflexive: x dom x

Computing Dominators

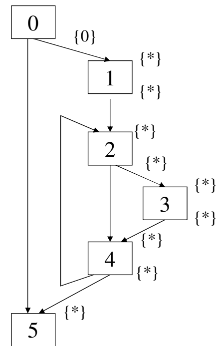
- One per procedure
- Special Entry and Exit nodes
- Dominators:
  x dom y if every possible execution path from the entry to y includes x.
- Reflexive: x dom x

Computing dominators



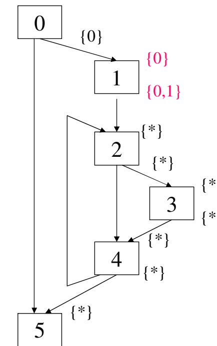
- Initialize:
  - 0{0}, rest {*}
- Meet function: intersect
- Transfer function: add self
- Iterate until no change…

Computing dominators



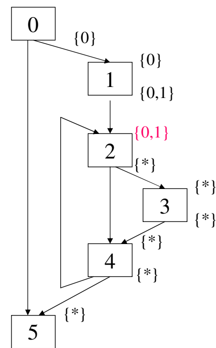
- Initialize:
  - 0{0}, rest {*}
- Meet function: intersect
- Transfer function: add self
- Iterate until no change…

Computing dominators



- Initialize:
  - 0{0}, rest {*}
- Meet function: intersect
- Transfer function: add self
- Iterate until no change…

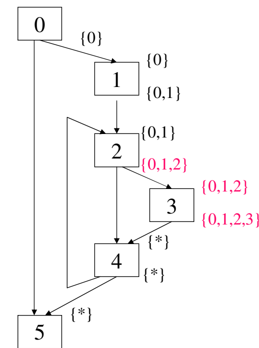
Computing dominators



- Initialize:
  - 0{0}, rest {*}
- Meet function: intersect
- Transfer function: add self
- Iterate until no change…

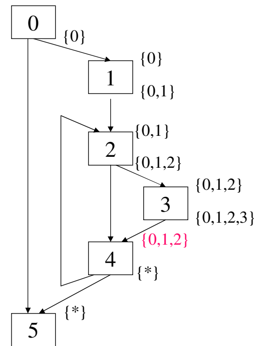
- What if we had initialized to empty sets?

Computing dominators



- Initialize:
  - 0{0}, rest {*}
- Meet function: intersect
- Transfer function: add self
- Iterate until no change…

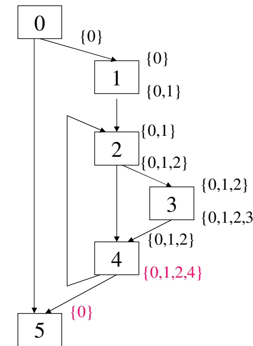
Computing dominators



- Initialize:
  - 0{0}, rest {*}
- Meet function: intersect
- Transfer function: add self
- Iterate until no change…

- 17 -

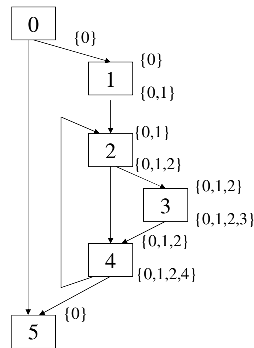
Computing dominators



- Initialize:
  - 0{0}, rest {*}
- Meet function: intersect
- Transfer function: add self
- Iterate until no change…

- 18 -

Computing dominators



- Initialize:
  - 0{0}, rest {*}
- Meet function: intersect
- Transfer function: add self
- Iterate until no change…

- 19 -