COMP 181

Lecture 18

Program analysis

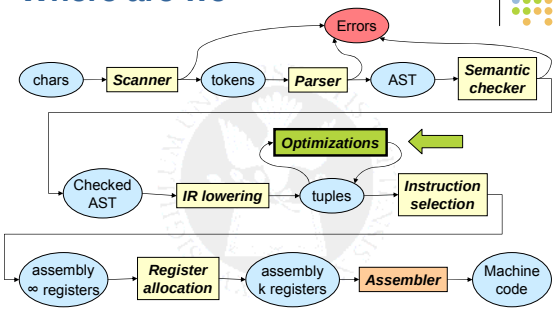
November 9, 2006

Prelude

- 100 yard dash
  - What causes muscles to hurt (and then fail)?
  - What chemical builds up?
  - Lactic acid
- Where does lactic acid come from?
  - Anaerobic metabolism – without oxygen
  - Partially burned carbs leave lactic acid behind
- Turns out, not exactly true
  - Body can use lactic acid as a fuel
  - Heart muscle actually prefers it
  - Key: intense interval training builds up mitochondria, which can burn lactic acid efficiently.

measuring VO2 max

Where are we



Back end

- Instruction selection
  - Mostly a local property
  - Build DAG of straight-line code, use pattern matching to identify instructions
- Optimization and register allocation
  - Can be formulated as local problems
  - More effective as ***global*** passes
    - Larger scope increases knowledge, improves results
    - "Global": typically means an entire procedure
  - Must take into account control flow

Optimization safety

- Safety of code transformations often require verification of code properties
  - Information not obvious, explicit in the code
  - Requires systematic analysis
- Example: dead code elimination

Dead code

- Example:

```
x = y + 1;
y = 2 * z;
x = y + z;
z = 1;
z = x;
```

Which statements can be safely removed?

- Conditions:
  - Computations whose value is never used
  - Obvious for straight-line code
  - What about control flow?

Dead code

- With control-flow:

Which statements are can be removed?

```
x = y + 1;
y = 2 * z;
if (c) x = y + z;
z = 1;
z = x;
```

- Which statements are dead code?
  - What if "c" is false?
  - Dead only on some paths through the code

Dead code

- With more control-flow:

Which statements are can be removed?

```
while (p) {
  x = y + 1;
  y = 2 * z;
  if (c) x = y + z;
  z = 1;
}
z = x;
```

- Now which statements are dead code?

Dead code

- With more control-flow:

Which statements are can be removed?

```
while (p) {
  x = y + 1;
  y = 2 * z;
  if (c) x = y + z;
  z = 1;
}
z = x;
```

- Statement "x = y+1" not dead
- What about "z = 1"?

Low-level IR

- Most optimizations performed in low-level IR
  - Labels and jumps
  - No explicit loops
- Even harder to see possible paths

```
label1:
jumpifnot p label2
x = y + 1
y = 2 * z
jumpifnot c label3
x = y + z
label3:
z = 1
jump label1
label2:
z = x
```


Optimizations and control flow

- Required information
  - Not obvious from program
    - Dead code example: are there any possible paths that make use of the value?
  - Must verify conditions at compile-time
  - Must characterize all possible dynamic behavior
- Control flow makes it hard to extract information
  - High-level: different kinds of control structures
  - Low-level: control-flow hard to infer
- Need a unifying data structure

Control flow graph

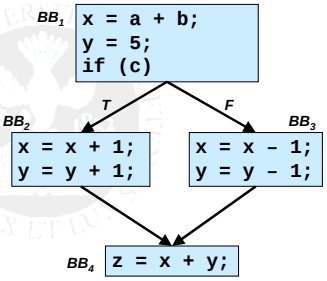
- ***Control flow graph*** (CFG):
  a graph representation of the program
  - Includes both computation and control flow
  - Easy to check control flow properties
  - Provides a framework for global optimizations and other compiler passes
- Nodes are ***basic blocks***
  - Consecutive sequences of non-branching statements
- Edges represent control flow
  - From jump to a label
  - Each block may have multiple incoming/outgoing edges

CFG Example

Program

```
x = a + b;
y = 5;
if (c) {
  x = x + 1;
  y = y + 1;
} else {
  x = x - 1;
  y = y - 1;
}
z = x + y;
```

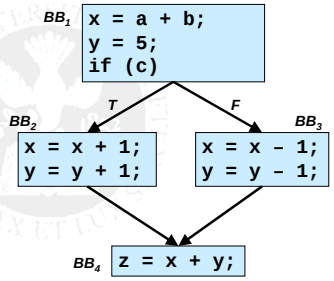
Control flow graph



Multiple program executions

- CFG models all program executions
- An actual execution is a path through the graph
- Multiple paths: multiple possible executions
  - How many?

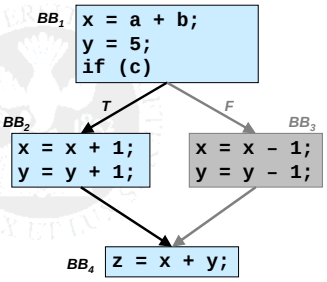
Control flow graph



Execution 1

- CFG models all program executions
- Execution 1:
  - c is true
  - Program executes BB_1, BB_2, and BB_4

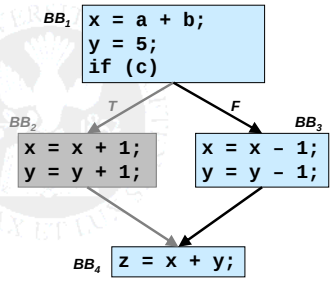
Control flow graph



Execution 2

- CFG models all program executions
- Execution 2:
  - c is false
  - Program executes BB_1, BB_3, and BB_4

Control flow graph



Basic blocks

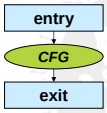
- Idea:
  - Once execution enters the sequence, all statements (or instructions) are executed
  - Single-entry, single-exit region
- Details
  - Starts with a label, ends with one or more branches
  - Edges may be labeled with predicates
  - May include special categories of edges
    - Exception jumps
    - Fall-through edges
    - Computed jumps (jump tables)

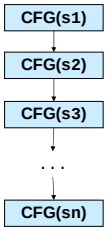
Building the CFG

- Can construct CFG in either high or low-level IR
  - High-level IR:
    - Each high-level control structure has an associated CFG pattern (if, while, for, etc.)
    - May still have to contend with unstructured control flow
  - Low-level IR:
    - Group together non-control-flow instructions
    - Analyze jump and label instructions
- How do the high-level structures look in the CFG?

High-level CFG

- Define construction of CFG recursively on AST
- Start with special entry and exit



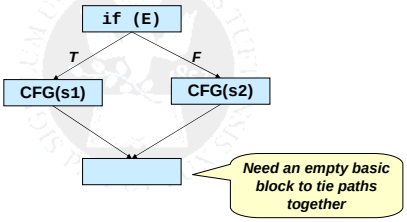


- Block statement:

`CFG( s1; s2; s3; …; sn)`

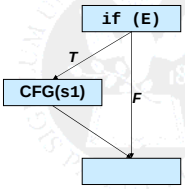
CFG for if-then-else

`CFG( if (E) s1 else s2 )`



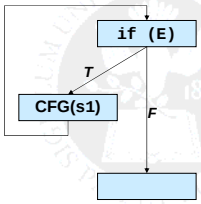
CFG for if-then

`CFG( if (E) s1 )`



CFG for while

`CFG( while (E) s1 )`



What are the potential complications here?

CFG for low-level IR

- More typical compiler design
- Two passes
  - First, group instructions into basic blocks
  - Second, analyze jumps and labels
- How to identify a basic blocks?
  - Non-branching instructions
    Control cannot flow out of a basic block without a jump
  - Non-label instruction
    Control cannot enter the middle of a block without a label

Basic blocks

- Basic block starts:
  - At a label
  - After a jump
- Basic block ends:
  - At a jump
  - Before a label

```
label1:
jumpifnot p label2
x = y + 1
y = 2 * z
jumpifnot c label3
x = y + z
label3:
z = 1
jump label1
label2:
z = x
```

Basic blocks

- Basic block start:
  - At a label
  - After a jump
- Basic block end:
  - At a jump
  - Before a label
- Note: order matters

```
label1:
jumpifnot p label2
```

```
x = y + 1
y = 2 * z
jumpifnot c label3
```

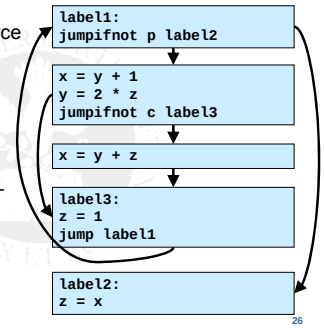
```
x = y + z
```

```
label3:
z = 1
jump label1
```

```
label2:
z = x
```


Add edges

- Unconditional jump
  - Add edge from source of jump to the block containing the label
- Conditional jump
  - 2 successors
  - One may be the fall-through block
- Fall-through



Two CFGs

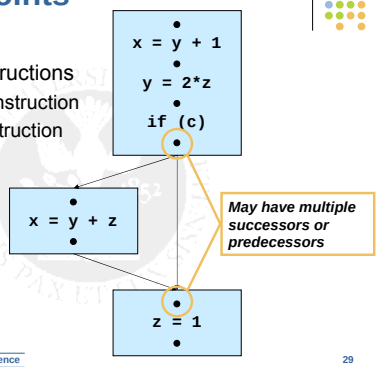
- From the high-level
  - Break down the complex constructs
  - Stop at sequences of non-control-flow statements
  - Requires special handling of break, continue, goto
- From the low-level
  - Start with lowered IR – tuples, or 3-address ops
  - Build up the graph
  - More general algorithm
- Should lead to roughly the same graph

Using the CFG

- Uniform representation for program behavior
  - Shows all possible program behavior
  - Each execution represented as a path
  - Can reason about potential behavior
    - Which paths can happen, which can't
  - Possible paths imply possible values of variables
- Example: liveness information
- Idea:
  - Define program points in CFG
  - Describe how information flows between points

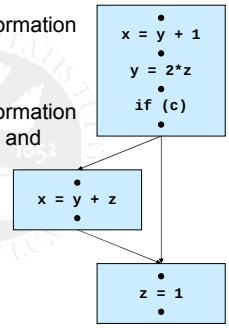
Program points

- In between instructions
  - Before each instruction
  - After each instruction



Flow of information

- ***Question 1***: how does information flow across instructions?
- ***Question 2***: how does information flow between predecessor and successor blocks?



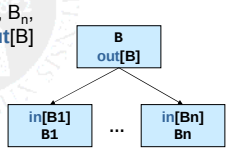
Live variables analysis

- At each program point:
  Which variables contain values computed earlier and needed later
- For instruction I:
  - in[I] : live variables at program point before I
  - out[I] : live variables at program point after I
- For a basic block B:
  - in[B] : live variables at beginning of B
  - out[B] : live variables at end of B
- Note: in[I] = in[B] for first instruction of B
  out[I] = out[B] for last instruction of B

Computing liveness

- ***Answer question 1***: for each instruction I, what is relation between in[I] and out[I]?

- ***Answer question 2***: for each basic block B, with successors $B_1, \ldots, B_n$, what is relationship between out[B] and in[B_1] … in[B_n]



Part 1: Analyze instructions

- Live variables across instructions
- Examples:

in[I] = {y,z}	in[I] = {y,z,t}	in[I] = {x,t}
`x = y + z`	`x = y + z`	`x = x + 1`
out[I] = {x,z}	out[I] = {x,t}	out[I] = {x,t}

- Is there a general rule?
  Yes: knowing the variables live after I, we can compute variables live before I

Liveness across instructions

- How is liveness determined?
  - All variables that I uses are live before I
    *Called the **uses** of I*
  - All variables live after I are also live before I, unless I writes to them
    *Called the **defs** of I*

in[I] = {b}
`a = b + 2`

in[I] = {y,z}
`x = 5`
out[I] = {x,y,z}

- Mathematically:

in[I] = (out[i] – def[I]) ∪ use[I]

Example

- Single basic block
  - Live1 = in[B] = in[I1]
  - Live2 = out[I1] = in[I2]
  - Live3 = out[I2] = in[I3]
  - Live4 = out[I3] = out[B]
- Relation between live sets
  - Live1 = (Live2 – {x}) ∪ {y}
  - Live2 = (Live3 – {y}) ∪ {z}
  - Live3 = (Live4 – {}) ∪ {d}

Live1
`x = y+1`
Live2
`y = 2*z`
Live3
`if (d)`
Live4

Flow of information

- Equation:

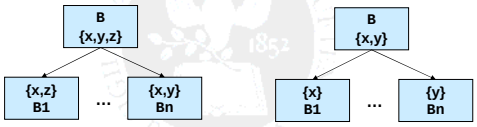
in[I] = (out[i] – def[I]) ∪ use[I]

- Notice: information flows ***backwards***
  - Need out[] sets to compute in[] sets
  - Propagate information up
- Many problems are ***forward***
  Common sub-expressions, constant propagation, others

Live1
`x = y+1`
Live2
`y = 2*z`
Live3
`if (d)`
Live4

Part 2: Analyze control flow

- ***Question 2***: for each basic block B, with successors B_1, …, B_n, what is relationship between **out**[B] and **in**[B_1] … **in**[B_n]
- Examples:



- What's the general rule?

Control flow

- Rule: A variable is live at end of block B if it is live at the beginning of *any* of the successors
  - Characterizes all possible executions
  - ***Conservative***: some paths may not actually happen
- Mathematically:

$$\mathbf{out}[B] = \bigcup_{B' \in succ(B)} \mathbf{in}[B']$$

- Again: information flows backwards

System of equations

- Put parts together:

$$\mathbf{in}[I] = (\mathbf{out}[i] - \mathbf{def}[I]) \cup \mathbf{use}[I]$$
$$\mathbf{out}[B] = \bigcup_{B' \in succ(B)} \mathbf{in}[B']$$

Often called a system of ***Dataflow Equations***

- Defines a system of equations (or constraints)
  - Consider equation instances for each instruction and each basic block
  - What happens with loops?
    - Circular dependences in the constraints
    - Is that a problem?

Solving the problem

- Iterative solution:
  - Start with empty sets of live variables
  - Iteratively apply constraints
  - Stop when we reach a fixed point

For all instructions **in**[I] = **out**[i] = ∅
Repeat
  For each instruction I
    in[I] = (**out**[i] – **def**[I]) ∪ **use**[I]
  For each basic block B
    $\mathbf{out}[B] = \bigcup_{B' \in succ(B)} \mathbf{in}[B']$
Until no new changes in sets

Next time

- More on optimizations
- With liveness information we can perform register allocation

Notes

- CFG stuff a little slow
- Show more examples of difficult control flow
- Show translation of high to low level?
- Liveness
  - Starts out a bit abstract
  - Missing a definition of liveness?
  - Need a concrete example at the end
  - This computed live ranges – it isn't good for dead-code removal