CS153: Compilers
Lecture 17: Control Flow Graph and Data Flow Analysis

Stephen Chong
https://www.seas.harvard.edu/courses/cs153
Announcements

• Project 5 out
  • Due Tuesday Nov 13 (14 days)

• Project 6 out
  • Due Tuesday Nov 20 (21 days)

• Project 7 will be released today
  • Due Thursday Nov 29 (30 days)
Today

• Control Flow Graphs
  • Basic Blocks

• Dataflow Analysis
  • Available Expressions
Optimizations So Far

• We’ve look only at local optimizations
  • Limited to “pure” expressions
  • Avoid variable capture by having unique variable names
Next Few Lectures

• Imperative Representations
  • Like MIPS assembly at the instruction level.
    • except we assume an infinite # of temps
    • and abstract away details of the calling convention
  • But with a bit more structure.

• Organized into a Control-Flow graph
  • nodes: labeled basic blocks of instructions
    • single-entry, single-exit
    • i.e., no jumps, branching, or labels inside block
  • edges: jumps/branches to basic blocks

• Dataflow analysis
  • computing information to answer questions about data flowing through the graph.
Control-Flow Graphs

• Graphical representation of a program
• Edges in graph represent control flow: how execution traverses a program
• Nodes represent statements

```plaintext
x := 0;
y := 0;
while (n > 0) {
    if (n % 2 = 0) {
        x := x + n;
y := y + 1;
    }
    else {
        y := y + n;
x := x + 1;
    }
n := n - 1;
}print(x);
```

```
x := 0
y := 0
n > 0
n%2=0
x:=x+n
y:=y+n
y:=y+1
x:=x+1
n:=n-1
print(x)
```
Basic Blocks

• We will require that nodes of a control flow graph are **basic blocks**
  • Sequences of statements such that:
    • Can be entered only at beginning of block
    • Can be exited only at end of block
      ‣ Exit by branching, by unconditional jump to another block, or by returning from function
  • Basic blocks simplify representation and analysis
Basic Blocks

- Basic block: single entry, single exit

```plaintext
x := 0;
y := 0;
while (n > 0) {
    if (n % 2 == 0) {
        x := x + n;
        y := y + 1;
    } else {
        y := y + n;
        x := x + 1;
    }
    n := n - 1;
}
print(x);
```
type operand =
  | Int of int | Var of var | Label of label

type block =
  | Return of operand
  | Jump of label
  | Branch of operand * test * operand * label * label
  | Move of var * operand * block
  | Load of var * int * operand * block
  | Store of var * int * operand * block
  | Assign of var * primop * (operand list) * block
  | Call of var * operand * (operand list) * block

type proc = { vars : var list, prologue: label, epilogue: label, blocks : (label * block) list }
Differences with Monadic Form

• Essentially MIPS assembly with infinite number of registers

• No lambdas, so easy to translate to MIPS modulo register allocation and assignment.
  • Monadic form requires extra pass to eliminate lambdas and make closures explicit (closure conversion, lambda lifting)

• Unlike Monadic Form, variables are mutable

• Return constructor is function return, not monadic return
Let’s Revisit Optimizations

• Folding
  • \( t := 3 + 4 \) becomes \( t := 7 \)

• Constant propagation
  • \( t := 7; \ B; \ u := t + 3; \ B' \)
    becomes \( t := 7; \ B; \ u := 7 + 3; \ B' \)
  • Problem! \( B \) might assign a fresh value to \( t \)

• Copy propagation
  • \( t := u; \ B; \ v := t + 3; \ B' \)
    becomes \( t := u; \ B; \ v := u + 3; \ B' \)
  • Problem! \( B \) might assign a fresh value to \( t \) or \( u \)
Let’s Revisit Optimizations

- **Dead code elimination**
  - \( x := e; B; \text{jump } L \) becomes \( B; \text{jump } L \)
  - Problem! Block \( L \) might use \( x \)
- \( x := e_1; B_1; x := e_2; B_2 \) becomes \( B_1; x := e_2; B_2 \)
  (\( x \) not used in \( B_1 \))

- **Common sub-expression elimination**
  - \( x := y + z; B_1; w := y + z; B_2 \) becomes \( x := y + z; B_1; w := x; B_2 \)
  - Problem: \( B_1 \) might change \( x, y, \) or \( z \)
Optimization in Imperative Settings

- Optimization on a functional representation:
  - Only had to worry about variable capture.
  - Could avoid this by renaming variables so that they were unique.
  - Then: \( \text{let } x = p(v_1, \ldots, v_n) \text{ in } e \equiv e[x \mapsto p(v_1, \ldots, v_n)] \)

- Optimization in an imperative representation:
  - Have to worry about intervening updates
    - for defined variable, similar to variable capture.
    - but must also worry about free variables.
  - \( x := p(v_1, \ldots, v_n); B \equiv B[x \mapsto p(v_1, \ldots, v_n)] \) only when \( B \) doesn't modify \( x \) or modify any of the \( v_i \).

- On the other hand, graph representation makes it possible to be more precise about the scope of a variable.
Dataflow Analysis

• To handle intervening updates we will compute **analysis facts** for each **program point**
  • There is a “program point” immediately before and after each instruction

• Analysis facts are facts about variables, expressions, etc.
  • Which facts we are interested in will depend on the particular optimization or analysis we are concerned with

• Given that some facts $D$ hold at a program point before instruction $S$, after $S$ executes some facts $D'$ will hold
  • How $S$ transforms $D$ into $D'$ is called the transfer function for $S$

• This kind of analysis is called dataflow analysis
  • Because given a control-flow graph, we are computing facts about data/variables and propagating these facts over the control flow graph
Available Expressions

• An expression \( e \) is available at program point \( p \) if on all paths from the entry to \( p \), expression \( e \) is computed at least once, and there are no intervening assignment to \( x \) or to the free variables of \( e \).

• If \( e \) is available at \( p \), we do not need to re-compute \( e \);
  • (i.e., for common sub-expression elimination)

• How do we compute the available expressions at each program point?
Available Expressions Example

1. $\emptyset$
2. $\emptyset$
3. $\{a+b\}$
4. $\{a+b, a*b\}$
5. $\{a+b, a*b\}$
6. $\{a+b, a*b\}$
7. $\{a+b, a*b\}$
8. $\emptyset$
9. $\{a+b\}$
10. $\{a+b\}$
11. $\{a+b\}$
12. $\{a+b\}$

(Numbers indicate the order that the facts are computed in this example.)
More Formally

- Suppose $D$ is a set of expressions that are available at program point $p$
- Suppose $p$ is immediately before “$x := e_1; B$”
- Then the statement “$x := e_1$”
  - generates the available expression $e_1$, and
  - kills any available expression $e_2$ in $D$ such that $x$ is in variables($e_2$)
- So the available expressions for $B$ are:
  $$(D \cup \{e_1\}) - \{ e_2 \mid x \in \text{variables}(e_2) \}$$
Gen and Kill Sets

- Can describe this analysis by the set of available expressions that each statement generates and kills!

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x:=v;</code></td>
<td><code>{ v }</code></td>
<td>`{e</td>
</tr>
<tr>
<td><code>x:=v_1 op v_2;</code></td>
<td><code>{v_1 op v_2}</code></td>
<td>`{e</td>
</tr>
<tr>
<td><code>x:=(v+i);</code></td>
<td><code>{}</code></td>
<td>`{e</td>
</tr>
<tr>
<td><code>(v+i):=x;</code></td>
<td><code>{}</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td><code>jump L;</code></td>
<td><code>{}</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td><code>return v;</code></td>
<td><code>{}</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td>if <code>v_1 op v_2 goto L_1 else goto L_2;</code></td>
<td><code>{}</code></td>
<td><code>{}</code></td>
</tr>
<tr>
<td><code>x:=v(v_1,...v_n);</code></td>
<td><code>{}</code></td>
<td>`{e</td>
</tr>
</tbody>
</table>
Available Expressions Example

What is the effect of each statement on the facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := a + b</td>
<td>a+b</td>
<td></td>
</tr>
<tr>
<td>y := a * b</td>
<td>a*b</td>
<td></td>
</tr>
<tr>
<td>y &gt; a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td>a+1</td>
<td>a+1</td>
</tr>
</tbody>
</table>

entry

- x := a + b;
- y := a * b;

y > a

- y > a
- a := a + 1;

exit

- x := a + b
Aliasing

• We don’t track expressions involving memory (loads & stores).
  • We can tell whether variables names are equal.
  • We cannot (in general) tell whether two variables will have the same value.
• If we track \(*x\) as an available expression, and then see \(*y := e'\), don’t know whether to kill \(*x\)
  • Don’t know whether \(x\)’s value will be the same as \(y\)’s value.
Because a function call may access memory, and may have side effects, we can’t consider them to be available expressions.
Flowing Through the Graph

• How to propagate available expression facts over control flow graph?
• Given available expressions $D_{\text{in}}[L]$ that flow into block labeled $L$, compute $D_{\text{out}}[L]$ that flow out
  • Composition of transfer functions of statements in $L$'s block
• For each block $L$, we can define:
  • $\text{succ}[L] = \text{the blocks $L$ might jump to}$
  • $\text{pred}[L] = \text{the blocks that might jump to $L$}$
• We can then flow $D_{\text{out}}[L]$ to all of the blocks in $\text{succ}[L]$
  • They'll compute new $D_{\text{out}}$'s and flow them to their successors and so on
• How should we combine facts from predecessors?
  • e.g., if $\text{pred}[L] = \{L_1, L_2, L_3\}$, how do we combine $D_{\text{out}}[L_1], D_{\text{out}}[L_2], D_{\text{out}}[L_3]$ to get $D_{\text{in}}[L]$?
  • Union or intersection?
Algorithm Sketch

- initialize $D_{in}[\mathcal{L}]$ to the empty set.
- initialize $D_{out}[\mathcal{L}]$ to the available expressions that flow out of block $\mathcal{L}$, assuming $D_{in}[\mathcal{L}]$ are the set flowing in.
- loop until no change {
  - for each $\mathcal{L}$:
    - $ln := \cap\{D_{out}[\mathcal{L}'] | \mathcal{L}' \in pred[\mathcal{L}] \}$
    - if $ln \neq D_{in}[\mathcal{L}]$ then {
      - $D_{in}[\mathcal{L}] := ln$
      - $D_{out}[\mathcal{L}] := \text{flow } D_{in}[\mathcal{L}] \text{ through } \mathcal{L}'s \text{ block.}$
    }
  }
}
Termination and Speed

- We know the available expressions dataflow analysis will terminate!
  - Each time through the loop each $D_{\text{in}}[L]$ and $D_{\text{out}}[L]$ either stay the same or increase
  - If all $D_{\text{in}}[L]$ and $D_{\text{out}}[L]$ stay the same, we stop
  - There's a finite number of assignments in the program and finite blocks, so a finite number of times we can increase $D_{\text{in}}[L]$ and $D_{\text{out}}[L]$
- In general, if set of facts form a lattice, transfer functions monotonic, then termination guaranteed
- There are a number of tricks used to speed up the analysis:
  - Can keep a work queue that holds only those blocks that have changed
  - Pre-compute transfer function for a block (i.e., composition of transfer functions of statements in block)