Announcements

First project has been posted. Deadline: March 4.
Project Part 2: Common subexpressions

Consider the tree for the expression

\[ a + a \times (b - c) + (b - c) \times d \]

Both \( a \) and \( b - c \) are common subexpressions (cse)

- compute the same value
- should compute the value once

A simple and general form of code improvement
Directed acyclic graphs

The directed acyclic graph is a useful representation for such expressions

\[ a + a \times (b - c) + (b - c) \times d \]

The *dag* clearly exposes the *ceses*

Aho, Sethi, and Ullman, §5.2, §9.8, …
Directed acyclic graphs

A directed acyclic graph is a tree with sharing

• a tree is a directed acyclic graph where each node has at most one parent
• a dag allows multiple parents for each node
• both a tree and a dag have a distinguished root
• no cycles in the graph!

To find common subexpressions (within a statement)

• build the dag
• generate code from the dag

This should lead to faster evaluation
Directed acyclic graphs

How do we build a \textit{dag} for an expression (single RHS)?

• use construction primitives for building tree
• teach primitives to catch \textit{cse’s} (see (ASU §5.2))
  — \textit{mkleaf()} and \textit{mknod}e()
  — hash on \textit{<op,l,r>}
• unique name for each node — its \textit{value number}

Anywhere that we build a tree, we could build a \textit{dag}

• initialize hash table on each expression
• catch only \textit{cse}s within expression
Directed acyclic graphs

What about assignment?

- complicates cse detection
- each value has a unique node
- add subscripts to variables

While building the dag, an assignment $x \leftarrow \ldots$

- determine “new” node for lhs — a new $x_i$
- kills all nodes built from $x_{i-1}$

Example

$S_1$: \[ a \leftarrow a + b \quad \text{a}_1 \leftarrow \text{a}_0 + b_0 \]
$S_2$: \[ c \leftarrow a + b \quad \text{c}_0 \leftarrow \text{a}_1 + b_0 \]

How to handle more than a single statement?
Directed acyclic graphs

Use a single dag for an entire basic block

A dag for a basic block has labeled nodes

1. leaves are labeled with unique identifier
   — either variable names or constants
   — leaves represent values on entry, e.g., $x_0$

2. interior nodes are labeled with operators

3. nodes have optional identifier labels
   — interior nodes represent computed values
   — identifier label represents assignment
Directed acyclic graphs

Example

<table>
<thead>
<tr>
<th>Code</th>
<th>After Renaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \leftarrow b + c )</td>
<td>( a_1 \leftarrow b_0 + c_0 )</td>
</tr>
<tr>
<td>( b \leftarrow a - d )</td>
<td>( b_1 \leftarrow a_1 - d_0 )</td>
</tr>
<tr>
<td>( c \leftarrow b + c )</td>
<td>( c_1 \leftarrow b_1 + c_0 )</td>
</tr>
<tr>
<td>( d \leftarrow a - d )</td>
<td>( d_1 \leftarrow a_1 - d_0 )</td>
</tr>
</tbody>
</table>
Directed acyclic graphs

Building a dag

\[ \text{node}( \text{id} ) \rightarrow \text{current dag for } \text{id} \]

1. set node(y) to undefined, for each symbol y

2. for each statement \( x \leftarrow y \text{ op } z \), repeat steps 3, 4, and 5

3. if node(y) is undefined,
   create a leaf for y
   set node(y) to the new node
   \textit{do the same for } z

4. if \( \text{op, node(y), node(z)} \) doesn’t exist,
   create it;
   let n be the node found or newly created node

5. delete x from the list of labels for node(x)
   append x to the list of labels for n
   set node(x) to n

Aho, Sethi, and Ullman, Algorithm 9.2, in §9.8
Directed acyclic graphs

Reality

Do compilers really use this stuff?

The dag construction algorithm is fast enough

A compilers that uses quads will (often)

- build a dag to find cses
- convert back to quads for later passes

Are there many cses? Yes!

- they arise in addressing
- array subscript code
- field access in records
- expressions based on loop indices
- access to parameters
General rewrite system:

- Replace **pattern 1** by **pattern 2**: 
  \[
  \text{pattern 1} \rightarrow \text{pattern 2}
  \]
- Apply single rewrite rules, one at a time, until no more rules can be applied.

Examples - come up with a rewrite system to implement the following:

1. simple functions: Syntactic representation of numbers: $\bullet | | | | | \#$ corresponds to integer value “5”. $\$ and $\#$ are left and right markers, respectively. Think of $\bullet$ as the **not yet done** marker.
   
   (a) $f(x) = x - 2$; report error for 0 and 1;
   
   $\$ $\bullet | | | | | | \# \Rightarrow^{*} \$ | | | | | | | | | | | | \#$
   
   (b) $f(x) = 2 \times x$
   
   $\$ $\bullet | | | | | | \# \Rightarrow^{*} \$ | | | | | | | | | | | | | | | | | | \#$

2. Sketch a small proof system with facts (e.g.: $a$, $b$) and inference rules (e.g.: $\frac{a \rightarrow b}{b}$)
**Lambda calculus**

- formalism for studying ways in which functions can be formed, combined, and used for computation

- **computation** is defined as rewriting rules (operational semantics)

- the syntactic notion of computation was developed first; a mathematical semantics followed much later

Examples:

- \( f(x) = x + 2 \)
- \( \lambda x. x + 2 \)
  - different notation
- \( (\lambda x. x + 2) \ 1 \)
  - \( 1 + 2 = 3 \)
  - function application and substitution
- \( (\lambda x. x) \ (\lambda y. y) \)
  - arguments and returned “values” can be functions
- \( \lambda x. xx \)
  - untyped lambda calculus
  - \( f(x) = x(x) \)
Lambda calculus and functional programming

Lambda calculus is the theoretical foundation of pure functional programming (no side effects, *referential transparency*).

Functional programming: functions are *first class citizens*

- can be a return value
- can be passed as arguments
- can be put into a data structure
- value of an expression can be a function

\[((\lambda x.x) \ (\lambda x.1)) \ (\lambda y.y)\]
Next Lecture

• Lambda calculus

• Introduction to functional languages (Scheme) ; Scott: Chapters 10.1 - 10.3