CS 473: COMPILER DESIGN
Compilation in a Nutshell

Source Code
(Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0 ; }

Abstract Syntax Tree:

Intermediate code:
11:
  %cnd = icmp eq i64 %b, 0
  br i1 %cnd, label %l2, label %l3
12:
  store i64* %a, 1
  br label %l3
13:

Assembly Code
11:
  cmpq %eax, $0
  jeq 12
  jmp 13
12:
  ...

Lexical Analysis
Parsing
Analysis & Transformation
Backend
Compiler Goals

• The most important thing for a compiler to do is to translate source programs into target programs that *correctly* implement them
  – Parsing: does the AST match the program the user wrote?
  – Type-checking: does the user’s program make sense?
  – Translation: does each IR do the same thing as the one before it?
  – Instruction selection: does the instruction for each tile do the same thing as the tree nodes in the tile?
  – Register allocation: do we avoid using the same register for two different variables at the same time?
Compiler Goals

• The most important thing for a compiler to do is to translate source programs into target programs that correctly implement them.

• The second most important thing is to translate source programs into target programs that efficiently implement them:
  – small number of instructions
  – faster instructions instead of slower (e.g., registers instead of memory accesses)
  – use less stack space
  – don’t recompute the same thing repeatedly
  – (without making compilation take too long)
Compilation in a Nutshell

Source Code
(Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0 ; }

Abstract Syntax Tree:

Intermediate code:
l1:  
   %cnd = icmp eq i64 %b, 0  
   br i1 %cnd, label %l2, label %l3  
l2:  
   store i64* %a, 1  
   br label %l3  
l3:

Assembly Code
l1:
   cmpq %eax, $0  
   jeq 12  
   jmp 13  
l2:
   ...

Lexical Analysis
Parsing
Analysis & Transformation
Backend
OTHER DATAFLOW ANALYSES
Generalizing Dataflow Analyses

• Optimization = Analysis + Transformation

• The kind of iterative constraint solving used for liveness analysis applies to other kinds of analyses as well
  – Reaching definitions analysis
  – Available expressions analysis
  – Alias Analysis
  – Constant Propagation
  – These analyses follow the same 3-step approach as for liveness.
  – We can do them at any level of IR!

• To see these as an instance of the same kind of algorithm, the next few examples to work over a canonical intermediate instruction representation called quadruples
  – Allows easy definition of def[n] and use[n]
**Quadruple Statements**

- A quadruple is a generic program instruction: (up to) 1 destination variable, 2 source variables, 1 operator

<table>
<thead>
<tr>
<th>Quadruple forms n:</th>
<th>def[n]</th>
<th>use[n]</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b op c</td>
<td>{a}</td>
<td>{b,c}</td>
<td>arithmetic</td>
</tr>
<tr>
<td>a = load b</td>
<td>{a}</td>
<td>{b}</td>
<td>load</td>
</tr>
<tr>
<td>store b, a</td>
<td>Ø</td>
<td>{b}</td>
<td>store</td>
</tr>
<tr>
<td>a = call f(b₁,…,bₙ)</td>
<td>{a}</td>
<td>{b₁,…,bₙ}</td>
<td>call w/return</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>def[n]</th>
<th>use[n]</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>br L</td>
<td>Ø</td>
<td>Ø</td>
<td>direct jump</td>
</tr>
<tr>
<td>br a, L1, L2</td>
<td>Ø</td>
<td>{a}</td>
<td>branch</td>
</tr>
<tr>
<td>ret a</td>
<td>Ø</td>
<td>{a}</td>
<td>return</td>
</tr>
</tbody>
</table>

- Can be thought of as generic assembly, or tree IR with one op (Binop, Mem, …)
- We’ll describe analyses and transformations on CFGs of quadruples
REACHING DEFINITIONS
Reaching Definition Analysis

• Optimization = Analysis + Transformation

• Question: what uses in a program does a given variable definition reach?

• This analysis is used for constant propagation & copy prop.
  – Constant propagation: if a variable use will always have the same value, replace it with that value
  – Copy propagation: if we have a statement \( x = y \), replace uses of \( x \) with \( y \) (like coalescing in register allocation)

• Input: Quadruple CFG

• Output: \( \text{in}[n] \) (resp. \( \text{out}[n] \)) is the set of nodes that define a variable such that the definition may reach the beginning (resp. end) of node \( n \)
Example of Reaching Definitions

- Results of computing reaching definitions on this simple CFG:

1. $b = a + 2$
   - out[1]: {1}
   - in[2]:  {1}

2. $c = b \times b$
   - out[2]: {1,2}
   - in[3]:  {1,2}

3. $b = c + 1$
   - out[3]: {2,3}
   - in[4]:  {2,3}

4. return $b \times a$
Questions

Top
Reaching Definitions Step 1

- Define the sets of interest for the analysis
- Let $\text{defs}[a]$ be the set of nodes that define the variable $a$
- A node that defines a variable $a$ both generates a definition of $a$ and kills any other definition of $a$

- Define $\text{gen}[n]$ and $\text{kill}[n]$ as follows:

  Quadruple forms $n$:  |  $\text{gen}[n]$  |  $\text{kill}[n]$  \\
  \hline
  $a = b \text{ op } c$  |  $\{n\}$  |  $\text{defs}[a] - \{n\}$  \\
  $a = \text{load } b$  |  $\{n\}$  |  $\text{defs}[a] - \{n\}$  \\
  $\text{store } b, a$  |  $\emptyset$  |  $\emptyset$  \\
  $a = f(b_1,\ldots,b_n)$  |  $\{n\}$  |  $\text{defs}[a] - \{n\}$  \\
  $\text{br } L$  |  $\emptyset$  |  $\emptyset$  \\
  $\text{br } a \ L1 \ L2$  |  $\emptyset$  |  $\emptyset$  \\
  $\text{return } a$  |  $\emptyset$  |  $\emptyset$  \\

- How do $\text{gen}[n]$ and $\text{kill}[n]$ relate to the definitions available before and after $n$?
Reaching Definitions Step 2

- Define the constraints that a reaching definitions solution must satisfy:

- \( \text{out}[n] \supseteq \text{gen}[n] \)
  "The definitions that reach the end of a node at least include the definitions generated by the node"

- \( \text{in}[n] \supseteq \text{out}[n'] \) if \( n' \) is in \( \text{pred}[n] \)
  "The definitions that reach the beginning of a node include those that reach the exit of any predecessor"

- \( \text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n] \)
  "The definitions that come into a node either reach the end of the node or are killed by it"
  - Equivalently: \( \text{out}[n] \supseteq \text{in}[n] - \text{kill}[n] \)
Reaching Definitions Step 3

• Convert constraints to iterated updates:
  
  • $\text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n']$
  
  • $\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$

• Algorithm: initialize $\text{in}[n]$ and $\text{out}[n]$ to $\emptyset$, iterate the updates until a fixed point is reached

• The algorithm terminates because $\text{in}[n]$ and $\text{out}[n]$ increase monotonically, and at most include all definitions in the program

• The algorithm is precise because it finds the smallest sets that satisfy the constraints

• Constant propagation: If a use of $x$ is only reached by one definition of the form $x = c$, then we can replace the use of $x$ with the constant $c$
AVAILABLE EXPRESSIONS
Available Expressions

• Goal: don’t recompute what we’ve already computed (“common subexpression elimination”)
  
  - \[ a = x + 1 \]
  
  - \[ a = x + 1 \]
  
  - \[ b = x + 1 \]
  
  - \[ b = a \]

• This transformation is safe if \( x+1 \) computes the same value at both places (i.e. \( x \) hasn’t been modified in between)
  
  - In this case, we say that \( x+1 \) is an available expression

• Dataflow values:
  
  - \( \text{in}[n] = \) set of nodes whose RHSs are available on entry to \( n \)
  
  - \( \text{out}[n] = \) set of nodes whose RHSs are available on exit of \( n \)
Each statement makes some values available and changes others.

Define \( \text{gen}[n] \) and \( \text{kill}[n] \) as follows:

<table>
<thead>
<tr>
<th>Quadruple forms n:</th>
<th>( \text{gen}[n] )</th>
<th>( \text{kill}[n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b \text{ op } c )</td>
<td>{n} - \text{kill}[n]</td>
<td>\text{uses}[a]</td>
</tr>
<tr>
<td>( a = \text{load } b )</td>
<td>{n} - \text{kill}[n]</td>
<td>\text{uses}[a]</td>
</tr>
<tr>
<td>( \text{store } b, a )</td>
<td>( \emptyset )</td>
<td>( ? )</td>
</tr>
<tr>
<td>( \text{br } L )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{br } a \text{ L1 L2} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( a = f(b_1,...,b_n) )</td>
<td>( \emptyset )</td>
<td>( \text{uses}[a] \cup \text{all loads} )</td>
</tr>
<tr>
<td>( \text{return } a )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

Note: \{n\} - \text{kill}[n] means that, e.g., \( x = x + 1 \) doesn’t make \( x + 1 \) available.
• Each statement makes some values available and changes others

• Define \( \text{gen}[n] \) and \( \text{kill}[n] \) as follows:

<table>
<thead>
<tr>
<th>Quadruple forms n:</th>
<th>( \text{gen}[n] )</th>
<th>( \text{kill}[n] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b \ op \ c )</td>
<td>{n} - \text{kill}[n]</td>
<td>\text{uses}[a]</td>
</tr>
<tr>
<td>( a = \text{load} \ b )</td>
<td>{n} - \text{kill}[n]</td>
<td>\text{uses}[a]</td>
</tr>
<tr>
<td>( \text{store} \ b, a )</td>
<td>( \emptyset )</td>
<td>\text{loads from a?}</td>
</tr>
<tr>
<td>( \text{br L} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \text{br a L1 L2} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( a = f(b_1, \ldots, b_n) )</td>
<td>( \emptyset )</td>
<td>\text{uses}[a] \cup \text{all loads}</td>
</tr>
<tr>
<td>( \text{return } a )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

\[ \text{int } *x = a[1]; \]
\[ *x = 5; \]

We might never mention \( a \), but still load from the memory that \( a \) points to!
Available Expressions Step 1

- Each statement makes some values available and changes others

- Define $\text{gen}[n]$ and $\text{kill}[n]$ as follows:

  - **Quadruple forms n:**
    - $a = b \text{ op } c$
    - $a = \text{load } b$
    - $\text{store } b, a$
    - $\text{br } L$
    - $\text{br } a \ L1 \ L2$
    - $a = f(b_1, \ldots, b_n)$
    - return $a$

    | Quadruple form   | gen[n]       | kill[n]          |
    |------------------|--------------|------------------|
    | $a = b \text{ op } c$ | $\{n\} - \text{kill}[n]$ | $\text{uses}[a]$ |
    | $a = \text{load } b$ | $\{n\} - \text{kill}[n]$ | $\text{uses}[a]$ |
    | $\text{store } b, a$ | $\emptyset$ | $\text{all loads}$ |
    | $\text{br } L$ | $\emptyset$ | $\emptyset$ |
    | $\text{br } a \ L1 \ L2$ | $\emptyset$ | $\emptyset$ |
    | $a = f(b_1, \ldots, b_n)$ | $\emptyset$ | $\text{uses}[a] \cup \text{all loads}$ |
    | return $a$ | $\emptyset$ | $\emptyset$ |

int *x = a[1];
*x = 5;

We might never mention a, but still load from the memory that a points to!
Available Expressions Step 1

- Each statement makes some values available and changes others

- Define \text{gen}[n] and \text{kill}[n] as follows:

  - Quadruple forms \( n \):
    \[
    \begin{array}{ll}
    \text{a = b op c} & \{n\} - \text{kill}[n] \\
    \text{a = load b} & \{n\} - \text{kill}[n] \\
    \text{store b, a} & \emptyset \\
    \text{br L} & \emptyset \\
    \text{br a L1 L2} & \emptyset \\
    \text{a = f(b_1,...,b_n)} & \emptyset \\
    \text{return a} & \emptyset 
    \end{array}
    \]

- \text{gen}[n] \quad \text{kill}[n]
  \begin{align*}
  \text{a = b op c} & \{n\} - \text{kill}[n] & \text{uses}[a] \\
  \text{a = load b} & \{n\} - \text{kill}[n] & \text{uses}[a] \\
  \text{store b, a} & \emptyset & \text{all loads that might alias with a} \\
  \text{br L} & \emptyset & \emptyset \\
  \text{br a L1 L2} & \emptyset & \emptyset \\
  \text{a = f(b_1,...,b_n)} & \emptyset & \text{uses}[a] \cup \text{all loads} \\
  \text{return a} & \emptyset & \emptyset 
  \end{align*}

- \text{int *x = a[1];} \quad \text{We might never mention a, but still load from the memory that a points to!}

- \text{*x = 5;}

Questions

Top
Available Expressions Step 1

• Each statement makes some values available and changes others

• Define \textit{gen}[n] and \textit{kill}[n] as follows:

• Quadruple forms n: \hspace{1cm} \textit{gen}[n] \hspace{1cm} \textit{kill}[n]

  \begin{align*}
  a = b \text{ op } c & \quad \{n\} - \text{kill}[n] \quad \text{uses}[a] \\
  a = \text{load } b & \quad \{n\} - \text{kill}[n] \quad \text{uses}[a] \\
  \text{store } b, a & \quad \emptyset \quad \text{all loads that might alias with } a \\
  \text{br } L & \quad \emptyset \quad \emptyset \\
  \text{br } a \ L1 \ L2 & \quad \emptyset \quad \emptyset \\
  a = f(b_1, \ldots, b_n) & \quad \emptyset \quad \text{uses}[a] \cup \text{all loads} \\
  \text{return } a & \quad \emptyset \quad \emptyset \\
  \end{align*}

• How do \textit{gen}[n] and \textit{kill}[n] relate to the expressions available before and after n?
Available Expressions Step 2

- Define the constraints that an available expressions solution must satisfy.

- $\text{out}[n] \supseteq \text{gen}[n]$
  “The expressions made available by $n$ that reach the end of the node”

- $\text{in}[n] \subseteq \text{out}[n']$ if $n'$ is in $\text{pred}[n]$
  “The expressions available at the beginning of a node include those that reach the exit of every predecessor”
  - We need to know that the expression will definitely be available, not that it might possibly be

- $\text{out}[n] \cup \text{kill}[n] \supseteq \text{in}[n]$
  “The expressions available on entry either reach the end of the node or are killed by it”
  - Equivalently: $\text{out}[n] \supseteq \text{in}[n] - \text{kill}[n]$

Note similarities and differences with constraints for “reaching definitions”.
Available Expressions Step 3

• Convert constraints to iterated updates:
  • \( \text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n'] \)
  • \( \text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n]) \)

• Algorithm: initialize \( \text{in}[n] \) and \( \text{out}[n] \) to \{set of all nodes\}, and iterate the updates until a fixed point is reached
• The algorithm terminates because \( \text{in}[n] \) and \( \text{out}[n] \) decrease only \textit{monotonically}, and can’t get smaller than the empty set
• The algorithm is precise because it finds the \textit{largest} sets that satisfy the constraints

• Common subexpression elimination: If we compute \( e \) when \( e \) is already available from a node \( t = e \), replace \( e \) with \( t \)
Questions

Top
Comparing Dataflow Analyses

- Look at the update equations in the inner loop of the analyses

- **Liveness:** (backward)
  - Let $\text{gen}[n] = \text{use}[n]$ and $\text{kill}[n] = \text{def}[n]$
  - $\text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']$
  - $\text{in}[n] := \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])$

- **Reaching Definitions:** (forward)
  - $\text{in}[n] := \bigcup_{n' \in \text{pred}[n]} \text{out}[n']$
  - $\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$

- **Available Expressions:** (forward)
  - $\text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']$
  - $\text{out}[n] := \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$
Common Features

- All of these analyses have a domain over which they solve constraints
  - Liveness: sets of variables
  - Reaching defns., available exprs.: sets of nodes
- Each analysis has a notion of $\text{gen}[n]$ and $\text{kill}[n]$
  - Used to explain how information propagates across a node
- Each analysis propagates information either forward or backward
  - Forward: $\text{in}[n]$ defined in terms of $\text{out}[n']$ for predecessors $n'$
  - Backward: $\text{out}[n]$ defined in terms of $\text{in}[n']$ for successors $n'$
- Each analysis has a way of combining information
  - Liveness & reaching definitions take union ($\cup$)
  - Available expressions uses intersection ($\cap$)
  - Union expresses a property that holds for some path (existential)
  - Intersection expresses a property that holds for all paths (universal)
Questions

Top
Dataflow Analysis on Basic Blocks

- Sometimes it is helpful to think of the fall-through between sequential instructions as an edge of the control-flow graph too.
- But it can be more efficient to do one basic block at a time!
Dataflow Analysis on Basic Blocks

x = y
z = a + b
Cjump

generates “x = y”

generates “z = a + b”

Jump
c = -z

kills other defs of x

kills other defs of z
Dataflow Analysis on Basic Blocks

\[ x = y \]
\[ z = a + b \]
\[ \text{Cjump} \]
\[ c = -z \]
\[ \text{Jump} \]

generates "\( x = y \)" and "\( z = a + b \)"
kills other defs of \( x \) and \( z \)
Dataflow Analysis on Basic Blocks

generates “x = y” and “x = a + b”
kills other defs of x
Dataflow Analysis on Basic Blocks

In general, the combination of nodes 1 and 2 has:

\[ \text{kill}[n] = \text{kill}[1] \cup \text{kill}[2] \]
\[ \text{gen}[n] = (\text{gen}[1] - \text{kill}[2]) \cup \text{gen}[2] \]