Agenda

• Dataflow analysis: a framework and algorithm for many common compiler analyses
• Initial example: dataflow analysis for common subexpression elimination
• Other analysis problems that work in the same framework
• Some of these are the same optimizations we’ve seen, but more formally and with details
Common Subexpression Elimination

- Goal: use dataflow analysis to find common subexpressions
- Idea: calculate *available expressions* at beginning of each basic block
- Avoid re-evaluation of an available expression – use a copy operation
  - Simple inside a single block; more complex dataflow analysis used across blocks
“Available” and Other Terms

• An expression $e$ is **defined** at point $p$ in the CFG if its value is computed at $p$
  – Sometimes called *definition site*

• An expression $e$ is **killed** at point $p$ if one of its operands is defined at $p$
  – Sometimes called *kill site*

• An expression $e$ is **available** at point $p$ if every path leading to $p$ contains a prior definition of $e$ and $e$ is not killed between that definition and $p$
Available Expression Sets

• To compute available expressions, for each block $b$, define
  – $\text{AVAIL}(b)$ – the set of expressions available on entry to $b$
  – $\text{NKILL}(b)$ – the set of expressions not killed in $b$
    • i.e., all expressions in the program except for those killed in $b$
  – $\text{DEF}(b)$ – the set of expressions defined in $b$ and not subsequently killed in $b$
Computing Available Expressions

- \( \text{AVAIL}(b) \) is the set
  \[
  \text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))
  \]
  - \( \text{preds}(b) \) is the set of \( b \)'s predecessors in the CFG
  - The set of expressions available on entry to \( b \) is the set of expressions that were available at the end of every predecessor basic block \( x \)
  - The expressions available on exit from block \( b \) are those defined in \( b \) or available on entry to \( b \) and not killed in \( b \)

- This gives a system of simultaneous equations – a dataflow problem
Computing Available Expressions

• Big Picture
  – Build control-flow graph
  – Calculate initial local data – DEF(b) and NKILL(b)
    • This only needs to be done once for each block b and depends only on the statements in b
  – Iteratively calculate AVAIL(b) by repeatedly evaluating equations until nothing changes
    • Another fixed-point algorithm
Computing DEF and NKILL (1)

• For each block $b$ with operations $o_1, o_2, \ldots, o_k$
  
  $\text{KILLED} = \emptyset$  // killed variables, not expressions
  
  $\text{DEF}(b) = \emptyset$
  
  for $i = k$ to $1$  // note: working back to front
    
    assume $o_i$ is “$x = y + z$”
    
    if ($y \notin \text{KILLED}$ and $z \notin \text{KILLED}$)
      
      add “$y + z$” to $\text{DEF}(b)$
      
      add $x$ to $\text{KILLED}$
      
    ...

    ...

UW CSE 401 Winter 2017
Computing DEF and NKILL (2)

• After computing DEF and KILLED for a block $b$, compute set of all expressions in the program not killed in $b$

  $\text{NKILL}(b) = \{ \text{all expressions} \}$

  for each expression $e$

    for each variable $v \in e$

      if $v \in \text{KILLED}$ then

        $\text{NKILL}(b) = \text{NKILL}(b) - e$
Example: Compute DEF and NKILL

```
j = 2 * a
k = 2 * b
DEF = \{ 2*a, 2*b \}
NKILL = exprs w/o j or k

x = a + b
b = c + d
m = 5 * n
DEF = \{ 5*n, c+d \}
NKILL = exprs w/o m, x, b

h = 2 * a
c = 5 * n
DEF = \{ 5*n \}
NKILL = exprs w/o c

DEF = \{ 2*a \}
NKILL = exprs w/o h
```
Computing Available Expressions

Once DEF(b) and NKILL(b) are computed for all blocks b

\[
\text{Worklist} = \{ \text{all blocks } b_i \} \\
\text{while } (\text{Worklist} \neq \emptyset) \\
\text{remove a block } b \text{ from Worklist} \\
\text{recompute } \text{AVAIL}(b) \\
\text{if } \text{AVAIL}(b) \text{ changed} \\
\text{Worklist} = \text{Worklist} \cup \text{successors}(b)
\]
Example: Find Available Expressions

\[ \text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x))) \]

- **DEF = \{ 2*a, 2*b \}**
- **NKILL = exprs w/o j or k**

- **DEF = \{ 5*n \}**
- **NKILL = exprs w/o c**

- **DEF = \{ 2*a \}**
- **NKILL = exprs w/o h**

- **DEF = \{ 5*n, c+d \}**
- **NKILL = exprs w/o m, x, b**

- **DEF = \{ 5*n, c+d \}**
- **NKILL = exprs w/o m, x, b**
Example: Find Available Expressions

\[ \text{AVAIL}(b) = \cap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x))) \]

AVAIL = \{ \}
DEF = \{ 2*a, 2*b \}
NKILL = exprs w/o j or k

DEF = \{ 5*n, c+d \}
NKILL = exprs w/o m, x, b

DEF = \{ 5*n \}
NKILL = exprs w/o c

DEF = \{ 2*a \}
NKILL = exprs w/o h

DEF = \{ 5*n \}
NKILL = exprs w/o c

DEF = \{ 2*a \}
NKILL = exprs w/o h
Example: Find Available Expressions

\[
AVAIL(b) = \cap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (AVAIL(x) \cap NKILL(x)))
\]

DEF = \{ 5*n, c+d \}
NKILL = exprs w/o m, x, b

DEF = \{ 5*n \}
NKILL = exprs w/o c

DEF = \{ 2*a \}
NKILL = exprs w/o h

AVAIL = \{ \}
DEF = \{ 2*a, 2*b \}
NKILL = exprs w/o j or k
Example: Find Available Expressions

\[
AVAIL(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))
\]

<table>
<thead>
<tr>
<th>AVAIL</th>
<th>DEF</th>
<th>NKILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ 2<em>a, 2</em>b }</td>
<td>exprs w/o j or k</td>
</tr>
<tr>
<td>{ 2<em>a, 2</em>b }</td>
<td>{ 5*n, c+d }</td>
<td>exprs w/o m, x, b</td>
</tr>
<tr>
<td>x = a + b</td>
<td>b = c + d</td>
<td></td>
</tr>
<tr>
<td>m = 5 * n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c = 5 * n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h = 2 * a</td>
<td>{ 5*n }</td>
<td>exprs w/o c</td>
</tr>
<tr>
<td>AVAIL = { 5*n }</td>
<td>DEF = { 2*a }</td>
<td>NKILL = exprs w/o h</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- = in worklist

- = processing
Example: Find Available Expressions

AVAIL(b) = \(\cap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x))))\)

\[
\begin{align*}
\text{AVAIL} &= \{ \} \\
\text{DEF} &= \{ 2\times a, 2\times b \} \\
\text{NKILL} &= \text{exprs w/o } j \text{ or } k
\end{align*}
\]

\[
\begin{align*}
\text{AVAIL} &= \{ 2\times a, 2\times b \} \\
\text{DEF} &= \{ 5\times n \} \\
\text{NKILL} &= \text{exprs w/o } c
\end{align*}
\]

\[
\begin{align*}
\text{AVAIL} &= \{ 5\times n \} \\
\text{DEF} &= \{ 2\times a \} \\
\text{NKILL} &= \text{exprs w/o } h
\end{align*}
\]
Example: Find Available Expressions

AVAIL(b) = \( \cap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x))) \)

- \( AVAIL = \{ \} \)
- \( DEF = \{ 2*a, 2*b \} \)
- \( NKILL = \text{exprs w/o j or k} \)

- \( AVAIL = \{ 2*a, 2*b \} \)
- \( DEF = \{ 5*n \} \)
- \( NKILL = \text{exprs w/o c} \)

- \( AVAIL = \{ 2*a, 2*b \} \)
- \( DEF = \{ 5*n, c+d \} \)
- \( NKILL = \text{exprs w/o m, x, b} \)

- \( AVAIL = \{ 5*n, 2*a \} \)
- \( DEF = \{ 2*a \} \)
- \( NKILL = \text{exprs w/o h} \)

\( j = 2 * a \)
\( k = 2 * b \)
\( x = a + b \)
\( b = c + d \)
\( m = 5 * n \)
\( c = 5 * n \)
\( h = 2 * a \)
Example: Find Available Expressions

\[ \text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup \text{AVAIL}(x) \cap \text{NKILL}(x)) \]

- **AVAIL**: \{ 2*a, 2*b \}
- **DEF**: \{ 5*n, c+d \}
- **NKILL**: exprs w/o m, x, b

- **AVAIL**: \{ 2*a, 2*b \}
  - **DEF**: \{ 2*a, 2*b \}
  - **NKILL**: exprs w/o j or k

- **AVAIL**: \{ 2*a, 2*b \}
  - **DEF**: \{ 5*n \}
  - **NKILL**: exprs w/o c

- **AVAIL**: \{ 5*n, 2*a \}
  - **DEF**: \{ 2*a \}
  - **NKILL**: exprs w/o h

And the common subexpression is???
Example: Find Available Expressions

$$AVAIL(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (AVAIL(x) \cap \text{NKILL}(x)))$$

- j = 2 * a
- k = 2 * b
- \(x = a + b\)
- b = c + d
- m = 5 * n
- h = 2 * a

- AVAIL = \{ \}
- DEF = \{ 2*a, 2*b \}
- NKILL = exprs w/o j or k

- AVAIL = \{ 2*a, 2*b \}
- DEF = \{ 5*n \}
- NKILL = exprs w/o c

- AVAIL = \{ 2*a, 2*b \}
- DEF = \{ 5*n, 2*a \}
- NKILL = exprs w/o h

= in worklist

= processing
Dataflow analysis

• Available expressions are an example of a dataflow analysis problem
• Many similar problems can be expressed in a similar framework
• Only the first part of the story – once we’ve discovered facts, we then need to use them to improve code
Characterizing Dataflow Analysis

• All of these algorithms involve sets of facts about each basic block b
  
  \[ \text{IN}(b) \] – facts true on entry to b
  
  \[ \text{OUT}(b) \] – facts true on exit from b
  
  \[ \text{GEN}(b) \] – facts created and not killed in b
  
  \[ \text{KILL}(b) \] – facts killed in b

• These are related by the equation
  
  \[ \text{OUT}(b) = \text{GEN}(b) \cup (\text{IN}(b) - \text{KILL}(b)) \]
  
  – Solve this iteratively for all blocks
  
  – Sometimes information propagates forward; sometimes backward
Example: Live Variable Analysis

- A variable $v$ is \textit{live} at point $p$ iff there is \textit{any} path from $p$ to a use of $v$ along which $v$ is not redefined
- Some uses:
  - Register allocation – only live variables need a register
  - Eliminating useless stores – if variable not live at store, then stored variable will never be used
  - Detecting uses of uninitialized variables – if live at declaration (before initialization) then it might be used uninitialized
  - Improve SSA construction – only need $\Phi$-function for variables that are live in a block (later)
Liveness Analysis Sets

- For each block \( b \), define
  - \( \text{use}[b] = \) variable used in \( b \) before any \( \text{def} \)
  - \( \text{def}[b] = \) variable defined in \( b \) & not killed
  - \( \text{in}[b] = \) variables live on entry to \( b \)
  - \( \text{out}[b] = \) variables live on exit from \( b \)
Equations for Live Variables

• Given the preceding definitions, we have
  \[
  \text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])
  \]
  \[
  \text{out}[b] = \bigcup_{s \in \text{succ}[b]} \text{in}[s]
  \]

• Algorithm
  – Set \text{in}[b] = \text{out}[b] = \emptyset
  – Update in, out until no change
Example (1 stmt per block)

- Code
  
  ```
  a := 0
  L:  b := a+1
  c := c+b
  a := b*2
  if a < N goto L
  return c
  ```

\[
in[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])
\]
\[
\text{out}[b] = \bigcup_{s \in \text{succ}[b]} \text{in}[s]
\]
## Calculation

\[
in[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])
\]

\[
\text{out}[b] = \bigcup_{s \in \text{succ}[b]} \text{in}[s]
\]

<table>
<thead>
<tr>
<th>block</th>
<th>use</th>
<th>def</th>
<th>out</th>
<th>in</th>
<th>out</th>
<th>in</th>
<th>out</th>
<th>in</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1: \(a := 0\)

2: \(b := a + 1\)

3: \(c := c + b\)

4: \(a := b + 2\)

5: \(a < N\)

6: return \(c\)
Calculation

<table>
<thead>
<tr>
<th>block</th>
<th>use</th>
<th>def</th>
<th>out</th>
<th>in</th>
<th>out</th>
<th>in</th>
<th>out</th>
<th>in</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>c</td>
<td>--</td>
<td>--</td>
<td>c</td>
<td>--</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>--</td>
<td>c</td>
<td>a,c</td>
<td>a,c</td>
<td>a,c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>a</td>
<td>a,c</td>
<td>b,c</td>
<td>a,c</td>
<td>b,c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b,c</td>
<td>c</td>
<td>b,c</td>
<td>b,c</td>
<td>b,c</td>
<td>b,c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>b</td>
<td>b,c</td>
<td>a,c</td>
<td>b,c</td>
<td>a,c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td>a</td>
<td>a,c</td>
<td>c</td>
<td>a,c</td>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1: \( a := 0 \)

2: \( b := a + 1 \)

3: \( c := c + b \)

4: \( a := b + 2 \)

5: \( a < N \)

6: return \( c \)

\[
\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])
\]

\[
\text{out}[b] = \bigcup_{s \in \text{succ}[b]} \text{in}[s]
\]
Equations for Live Variables v2

• Many problems have more than one formulation. For example, Live Variables...

• Sets
  
  – \( \text{USED}(b) \) – variables used in \( b \) before being defined in \( b \)
  
  – \( \text{NOTDEF}(b) \) – variables not defined in \( b \)
  
  – \( \text{LIVE}(b) \) – variables live on \textit{exit} from \( b \)

• Equation

\[
\text{LIVE}(b) = \bigcup_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{LIVE}(s) \cap \text{NOTDEF}(s))
\]
Efficiency of Dataflow Analysis

• The algorithms eventually terminate, but the expected time needed can be reduced by picking a good order to visit nodes in the CFG
  – Forward problems – reverse postorder
  – Backward problems – postorder
Example: Reaching Definitions

• A definition $d$ of some variable $v$ reaches operation $i$ iff $i$ reads the value of $v$ and there is a path from $d$ to $i$ that does not define $v$

• Uses
  – Find all of the possible definition points for a variable in an expression
Equations for Reaching Definitions

• Sets
  – DEFOUT(b) – set of definitions in b that reach the end of b (i.e., not subsequently redefined in b)
  – SURVIVED(b) – set of all definitions not obscured by a definition in b
  – REACHES(b) – set of definitions that reach b

• Equation

\[
\text{REACHES}(b) = \bigcup_{p \in \text{preds}(b)} \text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p))
\]
Example: Very Busy Expressions

• An expression $e$ is considered *very busy* at some point $p$ if $e$ is evaluated and used along every path that leaves $p$, and evaluating $e$ at $p$ would produce the same result as evaluating it at the original locations

• Uses
  – Code hoisting – move $e$ to $p$ (reduces code size; no effect on execution time)
Equations for Very Busy Expressions

• Sets
  – USED(b) – expressions used in b before they are killed
  – KILLED(b) – expressions redefined in b before they are used
  – VERYBUSY(b) – expressions very busy on exit from b

• Equation
  \[ \text{VERYBUSY}(b) = \bigcap_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{VERYBUSY}(s) - \text{KILLED}(s)) \]
Using Dataflow Information

• A few examples of possible transformations...
Classic Common-Subexpression Elimination (CSE)

- In a statement $s: t := x \text{ op } y$, if $x \text{ op } y$ is available at $s$ then it need not be recomputed.
- Analysis: compute *reaching expressions* i.e., statements $n: v := x \text{ op } y$ such that the path from $n$ to $s$ does not compute $x \text{ op } y$ or define $x$ or $y$. 
Classic CSE Transformation

• If \( x \) \text{ op } \( y \) is defined at \( n \) and reaches \( s \)
  – Create new temporary \( w \)
  – Rewrite \( n: v := x \text{ op } y \) as
    \begin{align*}
    n &: w := x \text{ op } y \\
    n' &: v := w
    \end{align*}
  – Modify statement \( s \) to be
    \( s: t := w \)
  – (Rely on copy propagation to remove extra assignments if not really needed)
Revisiting Example (w/slight addition)

\[
\begin{align*}
  j &= 2 \ast a \\
  k &= 2 \ast b \\
  x &= a + b \\
  b &= c + d \\
  m &= 5 \ast n \\
  h &= 2 \ast a \\
  i &= 5 \ast n \\
  c &= 5 \ast n \\
  \text{AVAIL} &= \{ 2*a, 2*b \} \\
  \text{AVAIL} &= \{ 5*n, 2*a \} \\
  \text{AVAIL} &= \{ 2*a, 2*b \} \\
  \text{AVAIL} &= \{ \} \\
\end{align*}
\]
Revisiting Example (w/slight addition)

AVAIL = { 2*a, 2*b }

x = a + b
b = c + d
t2 = 5 * n
m = t2

h = t1
i = t2

AVAIL = { }

AVAIL = { 2*a, 2*b }

AVAIL = { 2*a, 2*b }

AVAIL = { 5*n, 2*a }

AVAIL = { 5*n, 2*a }

---

AVAIL = { 2*a, 2*b }

x = a + b
b = c + d
t2 = 5 * n
m = t2

h = t1
i = t2

AVAIL = { 2*a, 2*b }

AVAIL = { 5*n, 2*a }

---
Then Apply Very Busy...

\[
\begin{align*}
\text{AVAIL} &= \{ \text{2*a, 2*b} \} \\
\text{t1} &= 2 \times a \\
j &= \text{t1} \\
k &= 2 \times b \\
t2 &= 5 \times n \\
x &= a + b \\
b &= c + d \\
t2 &= 5 \times n \\
m &= t2 \\
h &= \text{t1} \\
i &= \text{t2} \\
t2 &= 5 \times n \\
c &= t2 \\
\text{AVAIL} &= \{ \text{2*a, 2*b} \} \\
\text{AVAIL} &= \{ \text{5*n, 2*a} \} \\
\end{align*}
\]
Constant Propagation

• Suppose we have
  – Statement d: t := c, where c is constant
  – Statement n that uses t

• If d reaches n and no other definitions of t reach n, then rewrite n to use c instead of t
Copy Propagation

• Similar to constant propagation
• Setup:
  – Statement d: t := z
  – Statement n uses t
• If d reaches n and no other definition of t reaches n, and there is no definition of z on any path from d to n, then rewrite n to use z instead of t
  – Recall that this can help remove dead assignments
Copy Propagation Tradeoffs

• Downside is that this can increase the lifetime of variable z and increase need for registers or memory traffic

• But it can expose other optimizations, e.g.,

  a := y + z
  u := y
  c := u + z       // copy propagation makes this y + z

  — After copy propagation we can recognize the common subexpression
Dead Code Elimination

• If we have an instruction
  \[ s: a := b \text{ op } c \]
and a is not live-out after s, then s can be eliminated
  – Provided it has no implicit side effects that are visible (output, exceptions, etc.)
  • If b or c are function calls, they have to be assumed to have unknown side effects unless the compiler can prove otherwise
Dataflow...

• General framework for discovering facts about programs
  – Although not the only possible story
• And then: facts open opportunities for code improvement
• Next time: SSA (static single assignment) form – transform program to a new form where each variable has only one single definition
  – Can make many optimizations/analysis more efficient