CSE P 501 – Compilers

Dataflow Analysis
Hal Perkins
Winter 2016
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 optimizations we’ve seen, but more formally and with details
The Story So Far…

• Redundant expression elimination
  – Local Value Numbering
  – Superlocal Value Numbering
    • Extends VN to EBBs
    • SSA-like namespace
  – Dominator VN Technique (DVNT)

• All of these propagate along forward edges

• None are global
  – In particular, can’t handle back edges (loops)
Dominator Value Numbering

- Most sophisticated algorithm so far
- Still misses some opportunities
- Can’t handle loops
Available Expressions

• Goal: use dataflow analysis to find common subexpressions whose range spans basic blocks

• Idea: calculate available expressions at beginning of each basic block

• Avoid re-evaluation of an available expression – use a copy operation
“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
  
  – AVAIL\((b)\) – the set of expressions available on entry to \( b \)
  
  – NKILL\((b)\) – the set of expressions not killed in \( b \)
    
    • i.e., all expressions in the program except for those killed in \( b \)
  
  – DEF\((b)\) – the set of expressions defined in \( b \) and not subsequently killed in \( b \)
Computing Available Expressions

• 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
Name Space Issues

- In previous value-numbering algorithms, we used a SSA-like renaming to keep track of versions.
- In global dataflow problems, we use the original namespace:
  - we require \( a+b \) have the same value along all paths to its use
  - If \( a \) or \( b \) is updated along any path to its use, then \( a+b \) has the “wrong” value
  - so original names are exactly what we want
- The KILL information captures when a value is no longer available.
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 $o_1, o_2, \ldots, o_k$

  KILLED = $\emptyset$ // killed variables, not expressions
  DEF(b) = $\emptyset$

  for $i = k$ to 1 // note: working back to front

    assume $o_i$ is “x = y + z”

    if ($y \notin$ KILLED and $z \notin$ KILLED)

      add “y + z” to DEF(b)

      add x to KILLED

  ...


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 $b$

\[
NKILL(b) = \{ \text{all expressions} \}
\]
for each expression $e$

for each variable $v \in e$

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

\[
NKILL(b) = NKILL(b) - e
\]
Example: Compute DEF and NKILL

- Compute
  - $j = 2 \times a$
  - $k = 2 \times b$
- $x = a + b$
- $b = c + d$
- $m = 5 \times n$
- $h = 2 \times a$
- 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
Computing Available Expressions

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

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

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

\[
\begin{align*}
  j &= 2 \times a \\
k &= 2 \times b \\
x &= a + b \\
b &= c + d \\
m &= 5 \times n \\
c &= 5 \times n \\
h &= 2 \times a
\end{align*}
\]

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

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

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

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

\[= \text{in worklist}\]
\[= \text{processing}\]
Example: Find Available Expressions

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

---

\[
\begin{align*}
  j &= 2 \times a \\
  k &= 2 \times b \\
  x &= a + b \\
  b &= c + d \\
  m &= 5 \times n \\
  c &= 5 \times n \\
  h &= 2 \times a \\
  h &= 2 \times a
\end{align*}
\]

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

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

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

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

\[=\text{in worklist}\]
\[=\text{processing}\]
Example: Find Available Expressions

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

\[
\begin{align*}
  j &= 2 \times a \\
  k &= 2 \times b
\end{align*}
\]

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

\[
\begin{align*}
  x &= a + b \\
  b &= c + d \\
  m &= 5 \times n
\end{align*}
\]

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

\[
\begin{align*}
  c &= 5 \times n
\end{align*}
\]

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

\[
\begin{align*}
  h &= 2 \times a
\end{align*}
\]

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

\[
\begin{align*}
  m &= 5 \times n
\end{align*}
\]

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

[Diagram with nodes and edges labeled with equations and conditions]

\[
\begin{align*}
  = \text{in worklist} \\
  = \text{processing}
\end{align*}
\]
Example: Find Available Expressions

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

\[
\begin{align*}
  j &= 2 \ast a \\
  k &= 2 \ast b \\
  x &= a + b \\
  b &= c + d \\
  m &= 5 \ast n \\
  c &= 5 \ast n \\
  h &= 2 \ast a
\end{align*}
\]

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

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

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

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

Diagram:

- Green square = in worklist
- Yellow square = processing
Example: Find Available Expressions

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

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

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

- \text{AVAIL} = \{ \} \\
  \text{DEF} = \{ 2*a \} \\
  \text{NKILL} = \text{exprs w/o k}

= in worklist

= processing
Example: Find Available Expressions

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

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

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

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

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

- Green = in worklist
- Yellow = processing
Example: Find Available Expressions

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

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

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

\[ j = 2 \times a \]
\[ k = 2 \times b \]

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

\[ \text{AVAIL} = \{ 2a, 2b \} \]
\[ \text{DEF} = \{ 5n, c+d \} \]
\[ \text{NKILL} = \text{exprs w/o } m, x, b \]

\[ x = a + b \]
\[ b = c + d \]
\[ m = 5 \times n \]

\[ c = 5 \times n \]

\[ \text{AVAIL} = \{ 5n \} \]
\[ \text{DEF} = \{ 5n \} \]
\[ \text{NKILL} = \text{exprs w/o } c \]

\[ h = 2 \times a \]

\[ \text{AVAIL} = \{ 5n, 2a \} \]
\[ \text{DEF} = \{ 2a \} \]
\[ \text{NKILL} = \text{exprs w/o } h \]

\[ \text{AVAIL} = \{ 2a, 2b \} \]
\[ \text{DEF} = \{ 5n \} \]
\[ \text{NKILL} = \text{exprs w/o } c \]

\[ \text{AVAIL} = \{ 2a, 2b \} \]
\[ \text{DEF} = \{ 5n \} \]
\[ \text{NKILL} = \text{exprs w/o } c \]
Comparing Algorithms

- LVN – Local Value Numbering
- SVN – Superlocal Value Numbering
- DVN – DominatoT-based Value Numbering
- GRE – Global Redundancy Elimination
Comparing Algorithms (2)

- LVN $\Rightarrow$ SVN $\Rightarrow$ DVN form a strict hierarchy – later algorithms find a superset of previous information.
- Global RE finds a somewhat different set:
  - Discovers $e+f$ in $F$ (computed in both $D$ and $E$).
  - Misses identical values if they have different names (e.g., $a+b$ and $c+d$ when $a=c$ and $b=d$).
- Value Numbering catches this.
Scope of Analysis

• Larger context (EBBs, regions, global, interprocedural) sometimes helps
  – More opportunities for optimizations

• But not always
  – Introduces uncertainties about flow of control
  – Usually only allows weaker analysis
  – Sometimes has unwanted side effects
    • Can create additional pressure on registers, for example
Code Replication

• Sometimes replicating code increases opportunities – modify the code to create larger regions with simple control flow

• Two examples
  – Cloning
  – Inline substitution
Cloning

• Idea: duplicate blocks with multiple predecessors

• Tradeoff
  – More local optimization possibilities – larger blocks, fewer branches
  – But: larger code size, may slow down if it interacts badly with cache
Original VN Example

A
m = a + b
n = a + b

B
p = c + d
r = c + d

C
q = a + b
r = c + d

D
e = b + 18
s = a + b
u = e + f

E
e = a + 17
t = c + d
u = e + f

F
v = a + b
w = c + d
x = e + f

G
y = a + b
z = c + d
Example with cloning

\[
\begin{align*}
m &= a + b \\
n &= a + b \\
p &= c + d \\
r &= c + d \\
y &= a + b \\
z &= c + d \\
e &= b + 18 \\
s &= a + b \\
u &= e + f \\
v &= a + b \\
w &= c + d \\
x &= e + f \\
y &= a + b \\
z &= c + d \\
e &= a + 17 \\
t &= c + d \\
u &= e + f \\
v &= a + b \\
w &= c + d \\
x &= e + f \\
y &= a + b \\
z &= c + d
\end{align*}
\]
Inline Substitution

- Problem: an optimizer has to treat a procedure call as if it (could have) modified all globally reachable data
  - Plus there is the basic expense of calling the procedure
- Inline Substitution: replace each call site with a copy of the called function body
Inline Substitution Issues

• Pro
  – More effective optimization – better local context and don’t need to invalidate local assumptions
  – Eliminate overhead of normal function call
• Con
  – Potential code bloat
  – Need to manage recompilation when either caller or callee changes
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
  - IN(b) – facts true on entry to b
  - OUT(b) – facts true on exit from b
  - GEN(b) – facts created and not killed in b
  - 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
Dataflow Analysis (1)

• A collection of techniques for compile-time reasoning about run-time values
• Almost always involves building a graph
  – Trivial for basic blocks
  – Control-flow graph or derivative for global problems
  – Call graph or derivative for whole-program problems
Dataflow Analysis (2)

- Usually formulated as a set of *simultaneous equations* (dataflow problem)
  - Sets attached to nodes and edges
  - Need a lattice (or semilattice) to describe values
    - In particular, has an appropriate operator to combine values and an appropriate “bottom” or minimal value
Dataflow Analysis (3)

• Desired solution is usually a *meet over all paths* (MOP) solution
  – “What is true on every path from entry”
  – “What can happen on any path from entry”
  – Usually relates to safety of optimization
Dataflow Analysis (4)

• Limitations
  – Precision – “up to symbolic execution”
    • Assumes all paths taken
  – Sometimes cannot afford to compute full solution
  – Arrays – classic analysis treats each array as a single fact
  – Pointers – difficult, expensive to analyze
    • Imprecision rapidly adds up
    • But gotta do it to effectively optimize things like C/C++

• For scalar values we can quickly solve simple problems
Example: Live Variable Analysis

• A variable $v$ is \textit{live} at point $p$ iff there is 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
  - $\text{use}[b] = \text{variable used in } b \text{ before any } \text{def}$
  - $\text{def}[b] = \text{variable defined in } b \text{ & not killed}$
  - $\text{in}[b] = \text{variables live on entry to } b$
  - $\text{out}[b] = \text{variables live on exit from } b$
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

\[
\begin{align*}
in[b] &= \text{use}[b] \cup (\text{out}[b] - \text{def}[b]) \\
\text{out}[b] &= \bigcup_{s \in \text{succ}[b]} \text{in}[s]
\end{align*}
\]
Calculation

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

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 \( c \)

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

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

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

• Sets
  – USED(b) – variables used in b before being defined in b
  – NOTDEF(b) – variables not defined in b
  – LIVE(b) – variables live on 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
  – \text{DEFOUT}(b) – set of definitions in b that reach the end of b (i.e., not subsequently redefined in b)
  – \text{SURVIVED}(b) – set of all definitions not obscured by a definition in b
  – \text{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
  \[ \text{VERYBUSY}(b) = \bigcap_{s \in \text{succ}(b)} \text{USED}(s) \cup \big( \text{VERYBUSY}(s) - \text{KILLED}(s) \big) \]
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
    \[
    n: w := x \text{ op } y
    \]
    \[
    n': v := w
    \]
  – Modify statement \( s \) to be
    \[
    s: t := w
    \]
  – (Rely on copy propagation to remove extra assignments that are not really needed)
Revisiting Example (w/slight addition)

\[
\begin{align*}
  j &= 2 \times a \\
  k &= 2 \times b \\
  x &= a + b \\
  b &= c + d \\
  m &= 5 \times n \\
  h &= 2 \times a \\
  i &= 5 \times n \\
  c &= 5 \times n
\end{align*}
\]

AVAIL = \{ \}

AVAIL = \{ 2*a, 2*b \}

AVAIL = \{ 5*n, 2*a \}
Revisiting Example (w/slight addition)

AVAIL = \{ 2a, 2b \}

\begin{align*}
x &= a + b \\
b &= c + d \\
t2 &= 5 \times n \\
m &= t2
\end{align*}

AVAIL = \{ 2a, 2b \}

\begin{align*}
t1 &= 2 \times a \\
j &= t1 \\
k &= 2 \times b
\end{align*}

AVAIL = \{ \}

\begin{align*}
t2 &= 5 \times n \\
c &= t2
\end{align*}

AVAIL = \{ 2a, 2b \}

\begin{align*}
h &= t1 \\
i &= t2
\end{align*}

AVAIL = \{ 5n, 2a \}
Then Apply Very Busy...

\[
\begin{align*}
t_1 &= 2 \times a \\
j &= t_1 \\
k &= 2 \times b \\
t_2 &= 5 \times n
\end{align*}
\]

\[
\begin{align*}
x &= a + b \\
b &= c + d \\
t_2 &= 5 \times n \\
m &= t_2
\end{align*}
\]

AVAIL = \{ 2*a, 2*b \}

AVAIL = \{ 5*n, 2*a \}
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.,

\[
\begin{align*}
a & := y + z \\
u & := y \\
c & := u + z & \text{// copy propagation makes this } y + z
\end{align*}
\]

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

- If we have an instruction
  \[ s: \text{a := b op c} \]
  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
Aliases

- A variable or memory location may have multiple names or *aliases*
  - Call-by-reference parameters
  - Variables whose address is taken (&x)
  - Expressions that dereference pointers (p.x, *p)
  - Expressions involving subscripts (a[i])
  - Variables in nested scopes
Aliases vs Optimizations

• Example:

    p.x := 5;  q.x := 7;  a := p.x;

    – Does reaching definition analysis show that the definition of p.x reaches a?
    – (Or: do p and q refer to the same variable/object?)
    – (Or: can p and q refer to the same thing?)
Aliases vs Optimizations

• Example
  void f(int *p, int *q) {
    *p = 1; *q = 2;
    return *p;
  }

– How do we account for the possibility that \( p \) and \( q \) might refer to the same thing?

– Safe approximation: since it’s possible, assume it is true (but rules out a lot)
  • C programmers can use “restrict” to indicate no other pointer is an alias for this one
Types and Aliases (1)

• In Java, ML, MiniJava, and others, if two
  variables have incompatible types they cannot
  be names for the same location
  – Also helps that programmer cannot create
    arbitrary pointers to storage in these languages
Types and Aliases (2)

• Strategy: Divide memory locations into *alias classes* based on type information (every type, array, record field is a class)

• Implication: need to propagate type information from the semantics pass to optimizer
  – Not normally true of a minimally typed IR

• Items in different alias classes cannot refer to each other
Aliases and Flow Analysis

• Idea: Base alias classes on points where a value is created
  – Every new/malloc and each local or global variable whose address is taken is an alias class
  – Pointers can refer to values in multiple alias classes (so each memory reference is to a set of alias classes)
  – Use to calculate “may alias” information (e.g., p “may alias” q at program point s)
Using “may-alias” information

• Treat each alias class as a “variable” in dataflow analysis problems

• Example: framework for available expressions
  – Given statement  \( s: M[a] := b, \)
    
    \[
    \text{gen}[s] = \{ \}
    \]
    
    \[
    \text{kill}[s] = \{ M[x] \mid a \text{ may alias } x \text{ at } s \}
    \]
May-Alias Analysis

• Without alias analysis, #2 kills M[t] since x and t might be related
• If analysis determines that “x may-alias t” is false, M[t] is still available at #3; can eliminate the common subexpression and use copy propagation

• Code
  1:  u := M[t]
  2:  M[x] := r
  3:  w := M[t]
  4:  b := u+w
Where are we now?

- Dataflow analysis is the core of classical optimizations
  - Although not the only possible story
- Still to explore:
  - Discovering and optimizing loops
  - SSA – Static Single Assignment form