An Improvement: Basic Blocks

- No need to compute this one stmt at a time
- For straight line code:
  - In[s1; s2] = in[s1]
  - Out[s1; s2] = out[s2]
- Can we combine the gen and kill sets into one set per BB?

\[ \text{Gen}[BB] = \{2, 3, 4, 5\} \]
\[ \text{Kill}[BB] = \{1, 8, 11\} \]
\[ \text{Gen}[s1; s2] = (\text{Gen}[s1] - \text{Kill}[s2]) + \text{Gen}[s2] \]
\[ \text{Kill}[s1; s2] = (\text{Kill}[s1] - \text{Gen}[s2]) + \text{Kill}[s2] \]
Def-use chains are valuable too

- Def-use is IS symmetric to use-def.
- Iff d is in u’s UD chain, then u is in d’s DU chain.
- Can compute DU chains from UD chains - same basic info, a set of \(<u,d>\) pairs, just packaged differently.

Pairs for \(x\):

Use-def lists:
- Def-use lists:
  - \(d.B3: u.B5\)

What the ... is a Lattice?

- Represents values: for one item, or vector of all (often boolean to powerset)
- Has a defined top and bot
- According to ASU:
  - Top is least info: \(\top \land X = X\)
  - Bot is end: \(\bot \land X = \bot\)
  - Init in\([b]\) with top, out\([b]\) with \(F_b(\top)\).

Better Constant Propagation

- What about: \(x \leftarrow 1\) if \((y > z)\) \(x \leftarrow 1\)
  - \(a \leftarrow x\)
- Use a better lattice
  - Meet: \(a \land \top \rightarrow a\)
  - \(a \land \bot \rightarrow \bot\)
  - \(c \land c \rightarrow c\)
  - \(c \land d \ (\text{if } c \neq d) \rightarrow \bot\)
- Init all vars to: bot or top?
Available Expressions

- X+Y is "available" at statement S if
  - x+y is computed along every path from the start to S AND
  - neither x nor y is modified after the last evaluation of x+y

```
a <- b+c
b <- a-d
c <- b+c
d <- a-d
```

Computing Available Expressions

- Forward or backward?
- Values?
- Lattice?
- gen[b] =
  - kill[b] =
  - in[b] =
  - out[b] =
  - initialization?

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- gen[b] = if b evals expr e and doesn’t define variables used in e
- kill[b] = if b assigns to x, then all exprs using x are killed.
- out[b] = in[b] - kill[b] ∪ gen[b]
- in[b] = what to do at a join point?
- initialization?

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- gen[b] = if b evals expr e and doesn’t define variables used in e
- kill[b] = if b assigns to x, exprs(x) are killed
  - out[b] = (in[b] - kill[b]) ∪ gen[b]
- in[b] = An expr is avail only if avail on ALL edges, so: in[b] = ∩ over all p ∈ pred(b), out[p]
- Initialization
  - All nodes except entry are set to ALL avail
  - Entry is set to NONE avail
### Constructing Gen & Kill

<table>
<thead>
<tr>
<th>Stmt s</th>
<th>gen[s]</th>
<th>kill[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t \leftarrow x \ op \ y )</td>
<td>{ x \ op \ y } -kill[s]</td>
<td>{ exprs containing ( t ) }</td>
</tr>
<tr>
<td>( t \leftarrow M[a] )</td>
<td>{ ( M[a] ) } -kill[s]</td>
<td>{ exprs containing ( t ) }</td>
</tr>
<tr>
<td>( M[a] \leftarrow b )</td>
<td>{ }</td>
<td>{ for all ( x, M[x] ) }</td>
</tr>
<tr>
<td>( f(a, \ldots) )</td>
<td>{ ( M[x] ) for all ( x ) }</td>
<td>{ exprs containing ( t ) for all ( x, M[x] ) }</td>
</tr>
<tr>
<td>( t \leftarrow f(a,\ldots) )</td>
<td>{ }</td>
<td>{ exprs containing ( t ) for all ( x, M[x] ) }</td>
</tr>
</tbody>
</table>

### Example

Entry

1. \( c \leftarrow a+b \)
2. \( d \leftarrow a\times c \)
3. \( e \leftarrow d*d \)
4. \( i \leftarrow 1 \)
5. \( f[i] \leftarrow a+b \)
6. \( c \leftarrow e^2 \)
7. \( br c>d \)
8. \( g[i] \leftarrow a\times c \)
9. \( g[i] \leftarrow d*d \)
10. \( i \leftarrow i+1 \)
11. \( br i>10 \)
12. Exit

Gen={a+b,a*c,d*d} Kill={c>d,c*2,i>10,i+1}

Gen={a+b,c*d} Kill={c^2,M[x],a*c}

Gen={d*d} Kill={M[x]}
Example

\[ \text{In} = \{ \} \]
\[ \text{Out} = \{a+b, a*c, d*d\} \]
\[ \text{Gen} = \{a+b, a*c, d*d\} \]
\[ \text{Kill} = \{c>d\} \]

\[ \text{In} = \{ \} \]
\[ \text{Out} = \{a+b, a*c, d*d\} \]
\[ \text{Gen} = \{a+b, a*c\} \]
\[ \text{Kill} = \{d*d, c>e\} \]

\[ \text{In} = \{ \} \]
\[ \text{Out} = \{a+b, a*c, d*d\} \]
\[ \text{Gen} = \{a+b\} \]
\[ \text{Kill} = \{a*c, d*d\} \]

\[ \text{In} = \{ \} \]
\[ \text{Out} = \{a+b, a*c, d*d\} \]
\[ \text{Gen} = \{a+b\} \]
\[ \text{Kill} = \{a*c, d*d\} \]

\[ \text{In} = \{ \} \]
\[ \text{Out} = \{a+b, a*c, d*d\} \]
\[ \text{Gen} = \{a+b\} \]
\[ \text{Kill} = \{a*c, d*d\} \]

\[ \text{In} = \{ \} \]
\[ \text{Out} = \{a+b, a*c, d*d\} \]
\[ \text{Gen} = \{a+b\} \]
\[ \text{Kill} = \{a*c, d*d\} \]
Example

\[W=\{3\}\]

In={}  
out={}  
in={}  
out={a+b, a*c, d*d}

\[\text{c} \leftarrow \text{a+b}\]
\[\text{d} \leftarrow \text{a*c}\]
\[\text{e} \leftarrow \text{d*d}\]
\[\text{i} \leftarrow 1\]

\[\text{f}[\text{i}] \leftarrow \text{a+b}\]
\[\text{g}[\text{i}] \leftarrow \text{c*d}\]
\[\text{i} \leftarrow \text{i+1}\]

Gen={a+b, a*c, d*d}  
Kill={c*d, i+1}

CSE

• Calculate Available expressions
• For every stmt in program
  If expression, \( x \ \text{op} \ y \), is available {
    Compute reaching expressions for \( x \ \text{op} \ y \) at this stmt
    foreach stmt in RE of the form \( t \leftarrow x \ \text{op} \ y \)
    rewrite at: \( t' \leftarrow x \ \text{op} \ y \)
      \( t \leftarrow t' \)
  }
  replace \( x \ \text{op} \ y \) in stmt with \( t' \)

Calculating RE

• Could be dataflow problem, but not needed enough, so …
• To find RE for \( x \ \text{op} \ y \) at stmt \( S \)
  - traverse cfg backward from \( S \) until
    • reach \( t \leftarrow x + y \) (& put into RE)
    • reach definition of \( x \) or \( y \)
Example

```
Example
Entry
i <- 1
f[i] <- t1 ;
```

```
t1 <- a+b
```

```
c <- c*2
```

```
br c>d
```

```
g[i] <- a*c
g[i] <- t2 ;
```

```
t2 <- d*d
```

```
i <- i+1
```

```
Exit
```

```
Later in course we look at bidirectional dataflow
```

Dataflow Summary

```
Dataflow Summary
Union
intersection
```

```
Forward
Reaching defs
Available exprs
```

```
Backward
Live variables
```

```
Later in course we look at bidirectional dataflow
```

Dataflow Framework

- Lattice
- Universe of values
- Meet operator
- Basic attributes (e.g., gen, kill)
- Traversal order
- Transfer function

Dataflow Framework

- Another formulation: "Meet over Paths" (MOP)
  - To find in[B],
    - Enumerate all paths from entry to B to get P
    - Foreach path \( p_x \) in P, \( p_x = \{ \text{entry} \rightarrow b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b \} \),
      calculate sum of transfer functions:
      \[ s_x = F_{b_1}(\ldots F_{b_j}(F_{b_i}(\text{out}[\text{entry}]))) \ldots ) \]
    - Then do one big Meet over all the \( s_x \) values
    - Not practical; more of theoretical interest....
Finally...why not put the values on the edges?

\[ \text{in}[n] = \bigwedge_{p \in \text{pred}(n)} \text{out}[p] \]

\[ \text{out}[n] = F_n(\text{in}[n]) \]

\[ \text{e}[n \rightarrow s] = \bigwedge_{p \in \text{pred}(n)} F_n, s(e[p \rightarrow n]) \]

Muchnick’s example: smart const prop

\[ x = a + b; \quad a=1, b=2 \rightarrow a=2, b=1 \]

My example (again, const prop)

\[ \text{If } (i=10) \]

\[ ? \quad Y \quad N \quad ? \]