Machine-Independent Optimizations

Compiler Design

CSE 504

1. Example
2. Dataflow Analysis
3. Common Optimizations

Last modified: Mon Apr 27 2015 at 16:23:06 EDT
Version: 1.3 15:28:44 2015/01/25
Compiled at 12:44 on 2015/05/01

A Fragment of Quicksort

```
// a is an array
// a's indices range from m to n
i = m-1;   j = n;  v = a[n];
while(1) {
   do i = i + 1; while (a[i]<v);
   do j = j - 1; while (a[j]>v);
   if (i >= j) break;
   x = a[i]; a[i] = a[j]; a[j] = x;
}
x = a[i]; a[i] = a[n]; a[n] = x;
```

Rearranges a such that elements in `a[m..j]` are all less than any element in `a[i+1..n]`.

Three-Address Code for the Fragment

```
(1)  i = m-1
(2)  j = n
(3)  t1 = 4*n
(4)  v = a[t1]

(5)  i = i+1
(6)  t2 = 4*i
(7)  t3 = a[t2]
(8)  if t3<v goto (5)

(9)  j = j-1
(10) t4 = 4*j
(11) t5 = a[t4]
(12) if t5>v goto (9)

(13) if (i>=j) goto (23)
```

```
(14) t6 = 4*i
(15) x = a[t6]
(16) t7 = 4*i
(17) t8 = 4*j
(18) t9 = a[t8]
(19) a[t7] = t9
(20) t10 = 4*j
(21) a[t10] = x
(22) goto (5)

(23) t11 = 4*i
(24) x = a[t11]
(25) t12 = 4*i
(26) t13 = 4*n
(27) t14 = a[t13]
(28) a[t12] = t14
(29) t15 = 4*n
(30) a[t15] = x
```

Common Subexpression Elimination — 1

```
B3:   (9)   j = j-1
      (10)  t4 = 4*j
      (11)  t5 = a[t4]
      (12)  if t5>v goto (9)

B4:   (13)  if (i>=j) goto (23)

B5:   (14)  t6 = 4*i
      (15)  x = a[t6]
      (16)  t7 = 4*i
      (17)  t8 = 4*j
      (18)  t9 = a[t8]
      (19)  a[t7] = t9
      (20)  t10 = 4*j
      (21)  a[t10] = x
      (22)  goto (5)
```

Common Subexpression Elimination — 1

```
B3:   (9)   j = j-1
      (10)  t4 = 4*j
      (11)  t5 = a[t4]
      (12)  if t5>v goto (9)

B4:   (13)  if (i>=j) goto (23)

B5:   (14)  t6 = 4*i
      (15)  x = a[t6]
      (16)  t7 = 4*i
      (17)  t8 = 4*j
      (18)  t9 = a[t8]
      (19)  a[t7] = t9
      (20)  t10 = 4*j
      (21)  a[t10] = x
      (22)  goto (5)
```

Common Subexpression Elimination — 1

```
B3:   (9)   j = j-1
      (10)  t4 = 4*j
      (11)  t5 = a[t4]
      (12)  if t5>v goto (9)

B4:   (13)  if (i>=j) goto (23)

B5:   (14)  t6 = 4*i
      (15)  x = a[t6]
      (16)  t7 = 4*i
      (17)  t8 = 4*j
      (18)  t9 = a[t8]
      (19)  a[t7] = t9
      (20)  t10 = 4*j
      (21)  a[t10] = x
      (22)  goto (5)
```

Common Subexpression Elimination — 1

B_3:	(9)	j = j-1	
	(10)	t4 = 4*j	
	(11)	t5 = a[t4]	
	(12)	if t5>v goto (9)	
B_4:	(13)	if (i>=j) goto (23)	
B_5:	(14)	t6 = 4*i	
	(15)	x = a[t6]	
	(16)	~~t7 = 4*i~~	
	(17)	t8 = 4*j	
	(18)	t9 = a[t8]	
	(19)	~~a[t7] = t9~~	a[t6] = t9
	(20)	~~t10 = 4*j~~	
	(21)	~~a[t10] = x~~	a[t8] = x
	(22)	goto (5)	

Common Subexpression Elimination — 2

```
B3:   (9)    j = j-1
      (10)   t4 = 4*j
      (11)   t5 = a[t4]
      (12)   if t5>v goto (9)

B4:   (13)   if (i>=j) goto (23)

B5:   (14)    t6 = 4*i
      (15)    x = a[t6]
      (17)    t8 = 4*j
      (18)    t9 = a[t8]
      (19')   a[t6] = t9
      (21')   a[t8] = x
      (22)    goto (5)
```

Common Subexpression Elimination — 2

```
B3:   (9)    j = j-1
      (10)   t4 = 4*j
      (11)   t5 = a[t4]
      (12)   if t5>v goto (9)
B4:   (13)   if (i>=j) goto (23)

B5:   (14)    t6 = 4*i
      (15)    x = a[t6]
      (17)    t8 = 4*j
      (18)    t9 = a[t8]
      (19')   a[t6] = t9
      (21')   a[t8] = x
      (22)    goto (5)
```

Common Subexpression Elimination — 2

B_3:	(9)	j = j-1
	(10)	t4 = 4*j
	(11)	t5 = a[t4]
	(12)	if t5>v goto (9)
B_4:	(13)	if (i>=j) goto (23)
B_5:	(14)	t6 = 4*i
	(15)	x = a[t6]
	(17)	~~t8 = 4*j~~
	(18)	~~t9 = a[t8]~~ t9 = a[t4]
	(19')	a[t6] = t9
	(21')	~~a[t8] = x~~ a[t4] = x
	(22)	goto (5)

Common Subexpression Elimination — 2

B_3:	(9)	`j = j-1`
	(10)	`t4 = 4*j`
	(11)	`t5 = a[t4]`
	(12)	`if t5>v goto (9)`
B_4:	(13)	`if (i>=j) goto (23)`
B_5:	(14)	`t6 = 4*i`
	(15)	`x = a[t6]`
	(17)	~~`t8 = 4*j`~~
	(18)	~~`t9 = a[t8]`~~ `t9 = a[t4]`
	(19')	`a[t6] = t9`
	(21')	~~`a[t8] = x`~~ `a[t4] = x`
	(22)	`goto (5)`

Common Subexpression Elimination — 2

B_3:	(9)	j = j-1
	(10)	t4 = 4*j
	(11)	t5 = a[t4]
	(12)	if t5>v goto (9)
B_4:	(13)	if (i>=j) goto (23)
B_5:	(14)	t6 = 4*i
	(15)	x = a[t6]
	(17)	~~t8 = 4*j~~
	(18)	~~t9 = a[t8]~~ ~~t9 = a[t4]~~
	(19')	~~a[t6] = t9~~ a[t6] = t5
	(21')	~~a[t8] = x~~ a[t4] = x
	(22)	goto (5)

Common Subexpression Elimination — 3

B_2:

```
(5)   i = i+1
(6)   t2 = 4*i
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
```

B_3:

```
(9)    j = j-1
(10)   t4 = 4*j
(11)   t5 = a[t4]
(12)   if t5>v goto (9)
```

B_4:

```
(13)   if (i>=j) goto (23)
```

B_5:

```
(14)    t6 = 4*i
(15)    x = a[t6]
(19')   a[t6] = t5
(21')   a[t4] = x
(22)    goto (5)
```

Common Subexpression Elimination — 3

```
B2:   (5)   i = i+1
      (6)   t2 = 4*i
      (7)   t3 = a[t2]
      (8)   if t3<v goto (5)

B3:   (9)    j = j-1
      (10)   t4 = 4*j
      (11)   t5 = a[t4]
      (12)   if t5>v goto (9)

B4:   (13)   if (i>=j) goto (23)

B5:   (14)    t6 = 4*i
      (15)    x = a[t6]
      (19')   a[t6] = t5
      (21')   a[t4] = x
      (22)    goto (5)
```

Common Subexpression Elimination — 3

B_2:

```
(5)   i = i+1
(6)   t2 = 4*i
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
```

B_3:

```
(9)    j = j-1
(10)   t4 = 4*j
(11)   t5 = a[t4]
(12)   if t5>v goto (9)
```

B_4:

```
(13)   if (i>=j) goto (23)
```

B_5:

(14) ~~t6 = 4*i~~

(15) ~~x = a[t6]~~ x = t3

(19') ~~a[t6] = t5~~ a[t2] = t5

```
(21')  a[t4] = x
(22)   goto (5)
```

Copy Propagation

B_2:

```
(5)   i = i+1
(6)   t2 = 4*i
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
```

B_3:

```
(9)    j = j-1
(10)   t4 = 4*j
(11)   t5 = a[t4]
(12)   if t5>v goto (9)
```

B_4:

```
(13)   if (i>=j) goto (23)
```

B_5:

```
(15')   x = t3
(19')   a[t2] = t5
(21')   a[t4] = x
(22)    goto (5)
```

Copy Propagation

B_2:

```
(5)   i = i+1
(6)   t2 = 4*i
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
```

B_3:

```
(9)    j = j-1
(10)   t4 = 4*j
(11)   t5 = a[t4]
(12)   if t5>v goto (9)
```

B_4:

```
(13)   if (i>=j) goto (23)
```

B_5:

```
(15')   x = t3
(19')   a[t2] = t5
(21')   ~~a[t4] = x~~  a[t4] = t3
(22)    goto (5)
```

Dead Code Elimination

B_2:

```
(5)   i = i+1
(6)   t2 = 4*i
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
```

B_3:

```
(9)    j = j-1
(10)   t4 = 4*j
(11)   t5 = a[t4]
(12)   if t5>v goto (9)
```

B_4:

```
(13)   if (i>=j) goto (23)
```

B_5:

```
(15')   x = t3
(19')   a[t2] = t5
(21')   a[t4] = t3
(22)    goto (5)
```

B_6:

```
(24')   x = t3
(27')   t14 = a[t1]
(28')   a[t2] = t14
(30')   a[t1]=t3
```

Dead Code Elimination

B_2:

```
(5)  i = i+1
(6)  t2 = 4*i
(7)  t3 = a[t2]
(8)  if t3<v goto (5)
```

B_3:

```
(9)   j = j-1
(10)  t4 = 4*j
(11)  t5 = a[t4]
(12)  if t5>v goto (9)
```

B_4:

```
(13)  if (i>=j) goto (23)
```

B_5:

(15') ~~x = t3~~
(19') a[t2] = t5
(21') a[t4] = t3
(22) goto (5)

B_6:

(24') ~~x = t3~~
(27') t14 = a[t1]
(28') a[t2] = t14
(30') a[t1]=t3

Induction Variables and Strength Reduction

B_2:

```
(5)   i = i+1
(6)   t2 = 4*i
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
```

B_3:

```
(9)    j = j-1
(10)   t4 = 4*j
(11)   t5 = a[t4]
(12)   if t5>v goto (9)
```

B_4:

```
(13)   if (i>=j) goto (23)
```

B_5:

```
(19')   a[t2] = t5
(21')   a[t4] = t3
(22)    goto (5)
```

B_6:

```
(27')   t14 = a[t1]
(28')   a[t2] = t14
(30')   a[t1]=t3
```

Induction Variables and Strength Reduction

B_2:

```
(5)  i = i+1
(6)  t2 = 4*i
(7)  t3 = a[t2]
(8)  if t3<v goto (5)
```

B_3:

```
(9)   j = j-1
(10)  t4 = 4*j
(11)  t5 = a[t4]
(12)  if t5>v goto (9)
```

B_4:

```
(13)  if (i>=j) goto (23)
```

B_5:

```
(19')  a[t2] = t5
(21')  a[t4] = t3
(22)   goto (5)
```

B_6:

```
(27')  t14 = a[t1]
(28')  a[t2] = t14
(30')  a[t1]=t3
```

Induction Variables and Strength Reduction

B_2:

```
(5)   i = i+1
(6)   ~~t2 = 4*i~~  t2 = t2+4
(7)   t3 = a[t2]
(8)   if t3<v goto (5)
```

B_3:

```
(9)    j = j-1
(10)   ~~t4 = 4*j~~  t4 = t4-4
(11)   t5 = a[t4]
(12)   if t5>v goto (9)
```

B_4:

```
(13)   if (i>=j) goto (23)
```

B_5:

```
(19')   a[t2] = t5
(21')   a[t4] = t3
(22)    goto (5)
```

B_6:

```
(27')   t14 = a[t1]
(28')   a[t2] = t14
(30')   a[t1]=t3
```

Induction Variables and Strength Reduction

B_2:

```
(5)  i = i+1
(6)  ~~t2 = 4*i~~  t2 = t2+4
(7)  t3 = a[t2]
(8)  if t3<v goto (5)
```

B_3:

```
(9)   j = j-1
(10)  ~~t4 = 4*j~~  t4 = t4-4
(11)  t5 = a[t4]
(12)  if t5>v goto (9)
```

B_4:

```
(13)  if ( i>=j ) goto (23)
```

B_5:

```
(19')  a[t2] = t5
(21')  a[t4] = t3
(22)   goto (5)
```

B_6:

```
(27')  t14 = a[t1]
(28')  a[t2] = t14
(30')  a[t1]=t3
```

Induction Variables and Strength Reduction

B_2:

(5) i = i+1
(6) ~~t2 = 4*i~~ t2 = t2+4
(7) t3 = a[t2]
(8) if t3<v goto (5)

B_3:

(9) j = j-1
(10) ~~t4 = 4*j~~ t4 = t4-4
(11) t5 = a[t4]
(12) if t5>v goto (9)

B_4:

(13) if (~~i>=j~~ t2>=t4) goto (23)

B_5:

(19') a[t2] = t5
(21') a[t4] = t3
(22) goto (5)

B_6:

(27') t14 = a[t1]
(28') a[t2] = t14
(30') a[t1]=t3

Final Steps

B_2:

```
(5)    i = i+1
(6')   t2 = t2+4
(7)    t3 = a[t2]
(8)    if t3<v goto (5)
```

B_3:

```
(9)     j = j-1
(10')   t4 = t4-4
(11)    t5 = a[t4]
(12)    if t5>v goto (9)
```

B_4:

```
(13')   if (t2>=t4) goto (23)
```

B_5:

```
(19')   a[t2] = t5
(21')   a[t4] = t3
(22)    goto (5)
```

B_6:

```
(27')   t14 = a[t1]
(28')   a[t2] = t14
(30')   a[t1]=t3
```

Final Steps

B_2:

(5)	~~i = i+1~~	*dead code*
(6')	t2 = t2+4	
(7)	t3 = a[t2]	
(8)	if t3<v goto (5)	

B_3:

(9)	~~j = j-1~~	*dead code*
(10')	t4 = t4-4	
(11)	t5 = a[t4]	
(12)	if t5>v goto (9)	

B_4:

(13')	if (t2>=t4) goto (23)

B_5:

(19')	a[t2] = t5
(21')	a[t4] = t3
(22)	goto (5)

B_6:

(27')	t14 = a[t1]
(28')	a[t2] = t14
(30')	a[t1]=t3

End Result

```
B1:   (1)    i = m-1
      (3)    t1 = n << 2
      (4)    v = a[t1]
      (4a)   t2 = i << 2
      (4b)   t4 = t1

B2:   (6')   t2 = t2+4
      (7)    t3 = a[t2]
      (8)    if t3<v goto (5)

B3:   (10')   t4 = t4-4
      (11)    t5 = a[t4]
      (12)    if t5>v goto (9)

B4:   (13')   if (t2>=t4) goto (23)

B5:   (19')   a[t2] = t5
      (21')   a[t4] = t3
      (22)    goto (5)

B6:   (27')   t14 = a[t1]
      (28')   a[t2] = t14
      (30')   a[t1]=t3
```

Code Motion (via Another Example)

```
for (i=0; i<n; i++)
   for (j = 0; j < n; j ++)
       c[i][j] = 0.0;
```

Code Motion (via Another Example)

```
for (i=0; i<n; i++)
   for (j = 0; j < n; j ++)
       c[i][j] = 0.0;
```

```
(1)    i = 0
(2)    if (i >= n) goto (12)
(3)    j = 0
(4)    if (j >= n) goto (10)
(5)    t1 = i*n
(6)    t2 = c+t1
(7)    t2[j] = 0.0
(8)    j = j+1
(9)    goto (4)
(10)   i = i+1
(11)   goto (2)
(12)   ...
```

Code Motion (via Another Example)

```
for (i=0; i<n; i++)
   for (j = 0; j < n; j ++)
       c[i][j] = 0.0;
```

```
(1)   i = 0
(2)   if (i >= n) goto (12)
(3)   j = 0
(4)   if (j >= n) goto (10)
(5)   t1 = i*n
(6)   t2 = c+t1
(7)   t2[j] = 0.0
(8)   j = j+1
(9)   goto (4)
(10)  i = i+1
(11)  goto (2)
(12)  ...
```

Code Motion (via Another Example)

```
for (i=0; i<n; i++)
   for (j = 0; j < n; j ++)
       c[i][j] = 0.0;
```

```
(1)   i = 0
(2)   if (i >= n) goto (12)
(3)   j = 0
(4)   if (j >= n) goto (10)
(5)   t1 = i*n
(6)   t2 = c+t1
(7)   t2[j] = 0.0
(8)   j = j+1
(9)   goto (4)
(10)  i = i+1
(11)  goto (2)
(12)  ...
```

$\Rightarrow$

```
(1)   i = 0
(1a)  t2 = c-n
(2)   if (i >= n) goto (12)
(3)   j = 0
(5')  t2 = t2+n
(6')  if (j >= n) goto (10)
(7)   t2[j] = 0.0
(8)   j = j+1
(9)   goto (6')
(10)  i = i+1
(11)  goto (2)
(12)  ...
```

Reaching Definitions

- An assignment of the form $x = e$ for some expression e is said to define x.
- A definition at statement s_1 *reaches* another statement s_2 if:
  - there is some control flow path from s_1 to s_2, such that
  - there is no other definition of x on the path from s_1 to s_2.
- Let $In(s)$ be the set of all definitions that reach s.
- Let $Out(s)$ be the set of all definitions that reach all the immediate successors of s.
- Then $Out(s) = gen(s) \cup (In(s) - kill(s))$, where
  - $gen(s)$ is the set of definitions generated by s, and
  - $kill(s)$ is the set of definitions with the same lhs variables as those in s.
- $In(s) = \bigcup_{t \in pred(s)} Out(t)$

Reaching Definitions vs. Live Variables

- **Live Variables:** In and Out are the smallest sets such that

$$\begin{aligned} In(s) &= use(s) \cup (Out(s) - def(s)) \\ Out(s) &= \bigcup_{t \in succ(s)} In(t) \end{aligned}$$

- **Reaching Definitions:** In and Out are the smallest sets such that

$$\begin{aligned} In(s) &= \bigcup_{t \in pred(s)} Out(t) \\ Out(s) &= gen(s) \cup (In(s) - kill(s)) \end{aligned}$$

- The form of equations is identical, and they can be computed using the same procedure, except:
  - Live Variables are best computed backwards through the flow graph (information goes from successors to predecessors).
  - Reaching Definitions are best computed forwards through the flow graph (information goes from predecessors to successors).

Available Expressions

- An expression e is *available* at statement s if, for *every path* that reaches s_1, there is *some* statement s' where e is evaluated.
- Let $In(s)$ be the set of all expressions available immediately before s is evaluated.
- Let $Out(s)$ be the set of all expressions available immediately after s is evaluated.
- Then $Out(s) = gen(s) \cup (In(s) - kill(s))$, where
  - $gen(s)$ is the set of all expressions evaluated in s, and
  - $kill(s)$ is the set of all expressions that use the lhs variables defined in s.
- $In(s) = \bigcap_{t \in pred(s)} Out(t)$
- In and Out are the greatest sets that satisfy the above equations.

Data-Flow Analysis Framework

The 3 data-flow problems discussed so far (liveness, reaching definitions, and available expressions) can be seen as instances of a general *data-flow analysis* framework, specified by:

- Direction of data flow (forwards or backwards)
- A *semilattice* $(V, \wedge)$:
  - V is a non-empty set that contains a special element $\top$.
  - "$\wedge$" is a binary operator, called *meet*, that is closed over V and is *associative*, *commutative*, and *idempotent* (i.e. $x \wedge x = x$, for all $x \in V$)
  - For all $x \in V$, $x \wedge \top = x$.
- A family of *transfer functions* from $V \rightarrow V$ such that
  - F contains the identity function
  - F is closed w.r.t. to composition. I.e., if $f_1, f_2 \in F$, then f_3 defined as $f_3(x) = f_2(f_1(x))$ is also in F.

DFA Frameworks (contd.)

- **Liveness Analysis:**
  - Backwards analysis.
  - $V = \mathcal{P}(X)$, where X is the set of variables in the program;
  - Meet operator is set union.
  - $\top$ in V is the empty set.
  - Transfer functions are of the form $f(x) = G \cup (x - K)$ for constants G and K.
- **Available Expression Analysis:**
  - Forwards analysis.
  - $V = \mathcal{P}(E)$, where E is the set of expressions in the program.
  - Meet operator is set intersection.
  - $\top$ in V is the universal set (E)
  - Transfer functions are of the form $f(x) = G \cup (x - K)$ for constants G and K.

Specialized DFA Frameworks

Define $x \leq y$ iff $x \wedge y = x$. It can be shown that "$\leq$" is a partial order.

(Here, "$\wedge$" is the meet operator)

- **Monotone Frameworks:**
  - Every f in F is monotone: $x \leq y \Rightarrow f(x) \leq f(y)$.
  - Alternatively: $f(x \wedge y) \leq f(x) \wedge f(y)$
- **Distributive Frameworks:**
  - $f(x \wedge y) = f(x) \wedge f(y)$
  - Note: distributivity implies monotonicity

Generic DFA Algorithm

Given below for a Forward analysis:

1. $Out(entry) = v_{entry}$
2. $Out(b) = \top$, for each $b \neq entry$
3. While no change to any Out:
   - For each $b \neq entry$
     - $In(b) = \wedge_{a \in predecessor(b)} Out(a)$
     - $Out(b) = f_b(In(b))$

- Computes the Maximum Fixed Point (MFP) for monotone frameworks.
- If the framework is monotone and the height of the semilattice is finite, then the algorithm terminates.
- If and when the algorithm terminates, it computes a solution to the data-flow equations.

MOP and MFP solutions

- Ideal: The property computed over all feasible execution paths:

$$Ideal(B) = \bigwedge_{P:\text{feasible execution path}} f_P(v_{entry})$$

- MOP: (Meet over all paths) The property computed over all possible paths in the control flow graph:

$$MOP(B) = \bigwedge_{P:\text{control flow path}} f_P(v_{entry})$$

  - Note that $MOP(x) \leq Ideal(x)$ from monotonicty.
  - $MFP(x) \leq MOP(x)$ in general.
  - $MFP(x) = MOP(x)$ if the framework is distributive.

Constant Propagation

In terms of the DFA framework, constant propagation is a *forward* analysis:

- V: Assignment to each variable in the program, one of the following: a specific constant c, not a constant "NAC" or undefined "$UNDEF$".
- $\wedge$: For each element in the assignment, the meet is defined as follows:
  - $UNDEF \wedge v = v$; $NAC \wedge v = NAC$.
  - $c \wedge c = c$
  - $c_1 \wedge c_2 = NAC$ if $c_1 \neq c_2$

  Then $m_1 \wedge m_2 = m_3$ if $m_3(v) = m_1(v) \wedge m_2(v)$ for each v.
- Transfer functions, as in next slide

Transfer Function for Constant Propagation

If s is not an assignment, then f_s is the identity function.

If s is an assignment of the form $x = rhs$, then $f_s(m) = m'$ such that

- $m'(v) = m(v)$ for all $v \neq x$
- $m'(x) =$ as follows.
  - RHS is a single constant c, then $m'(x) = c$.
  - RHS is a single variable y, then $m'(x) = m(y)$.
  - RHS is an expression of the form $y \oplus z$, then

$$m'(x) = \begin{cases} m(y) \oplus m(z) & \text{if } m(y) \text{ and } m(z) \text{ are constants} \\ NAC & \text{if either } m(y) \text{ or } m(z) \text{ is } NAC \\ UNDEF & \text{otherwise} \end{cases}$$

Constant Propagation Optimization

- Compute m using the DFA algorithm for constant propagation.
- If $l: \quad t \leftarrow e$ is an assignment (in the intermediate code):
  - If $e = s$, and $m(s) = c$, then replace the assignment with $l: \quad t \leftarrow c$
  - If $e = s_1 \oplus s_2$, $m(s_1) = c_1$ and $m(s_2) = c_2$, then replace assignment with $l: \quad t \leftarrow c$, where $c = c_1 \oplus c_2$.

Copy Propagation

- Perform reaching definitions analysis
- If $l_1 : \quad t \leftarrow z$ and $l_2 : \quad y \leftarrow t \oplus x$ are two statements such that
  - l_1 reaches l_2
  - No other definition of t reaches l_2
  - There is no definition of z along any path from l_1 ot l_2

  Then replace $l_2 : \quad y \leftarrow t \oplus x$ with $l_2 : \quad y \leftarrow z \oplus x$

Common Subexpression Elimination

- Perform available expressions and reaching definitions analyses.
- If $l_1 : \quad t \leftarrow x \oplus y$ is a statement where expression $x \oplus y$ is available, and
  - Find statements of the form $l_2 \quad z \leftarrow x \oplus y$ such that l_2 reaches l_1, and x, y are not redefined on any path from l_2 to l_1.
  - Generate a new temporary name w.
  - Replace l_2 with the following two statements:

$$\begin{array}{ll} l_2 : & w \leftarrow x \oplus y \\ l_{2'} : & z \leftarrow w \end{array}$$

  Replace l_1 with $l_1 : \quad t \leftarrow w$.
- Copy propagation can later remove the extra assignments (or better, coalesce the temporaries during register allocation).

Loops

A loop in a control flow graph is a set of nodes L such that

- L has a unique *header* node h
- There is a path from h to every node in L
- From every node in L there is a path to h
- There is no edge in any node outside L to any node in $L - \{h\}$.

Corollary: A loop may have multiple exits, but have a single entry (h).

Loop Invariants

A definition $l_1 : \quad t \leftarrow a_1 \oplus a_2$ inside loop L is invariant in L if *one of the following conditions hold:*

- a_i's are constants
- all definitions of a_1 and a_2 that reach l_1 are outside L
- only one definition for each a_i reaches l_1 and that definition is loop invariant in L.

We can hoist an invariant assignment $l_1 : \quad t \leftarrow a_1 \oplus a_2$ out of loop L if *all of the following conditions hold:*

- l_1 *dominates* all loop exits at which t is live-out
- There is only one definition of t in L