Common-Subexpression elimination

Goal: remove redundant computations.

Example: statement `a[i] := a[i]+1` translates to:

```
t1 := 4*i
t2 := a+t1
t3 := M[t2]
t4 := t3+1
t5 := 4*i
t6 := a+t5
M[t6] := t4
```

Potential for optimization:

- Term `4*i` is computed twice.
- `a+t1` and `a+t5` are equal, since `t1`=`t5`.

Available-Assignments Analysis

Instruction i	$gen[i]$	$kill[i]$
$\texttt{LABEL}\ l$	$\emptyset$	$\emptyset$
$x := y$	$\emptyset$	$assg(x)$
$x := k$	$\{x := k\}$	$assg(x)$
$x := \textbf{unop}\ y$ where $x \neq y$	$\{x := \textbf{unop}\ y\}$	$assg(x)$
$x := \textbf{unop}\ x$	$\emptyset$	$assg(x)$
$x := \textbf{unop}\ k$	$\{x := \textbf{unop}\ k\}$	$assg(x)$
$x := y\ \textbf{binop}\ z$ where $x \neq y$ and $x \neq z$	$\{x := y\ \textbf{binop}\ z\}$	$assg(x)$
$x := y\ \textbf{binop}\ z$ where $x = y$ or $x = z$	$\emptyset$	$assg(x)$
$x := y\ \textbf{binop}\ k$ where $x \neq y$	$\{x := y\ \textbf{binop}\ k\}$	$assg(x)$
$x := x\ \textbf{binop}\ k$	$\emptyset$	$assg(x)$
$x := M[y]$ where $x \neq y$	$\{x := M[y]\}$	$assg(x)$
$x := M[x]$	$\emptyset$	$assg(x)$
$x := M[k]$	$\{x := M[k]\}$	$assg(x)$
$M[x] := \mathrm{y}$	$\emptyset$	$loads$
$M[k] := \mathrm{y}$	$\emptyset$	$loads$
$\texttt{GOTO}\ l$	$\emptyset$	$\emptyset$
$\texttt{IF}\ x\ \textbf{relop}\ y\ \texttt{THEN}\ l_t\ \texttt{ELSE}\ l_f$	$\emptyset$	$\emptyset$
$x := \texttt{CALL}\ f(args)$	$\emptyset$	$assg(x)$

$assg(x)$: Assignements that use x on the left or right-hand sides,

$loads$: Statements of form $y := M[\cdot]$.

Example for Available Assignments

1: $i := 0$
2: $a := n * 3$
3: IF $i < a$ THEN *loop* ELSE *end*
4: LABEL *loop*
5: $b := i * 4$
6: $c := p + b$
7: $d := M[c]$
8: $e := d * 2$
9: $f := i * 4$
10: $g := p + f$
11: $M[g] := e$
12: $i := i + 1$
13: $a := n * 3$
14: IF $i < a$ THEN *loop* ELSE *end*
15: LABEL *end*

i	$pred[i]$	$gen[i]$	$kill[i]$
1		1	1, 5, 9, 12
2	1	2	2
3	2		
4	3, 14		
5	4	5	5, 6
6	5	6	6, 7
7	6	7	7, 8
8	7	8	8
9	8	9	9, 10
10	9	10	10
11	10		7
12	11		1, 5, 9, 12
13	12	2	2
14	13		
15	3, 14		

Note: Assignment 2 and assignment 13 both represented by 2.

Fix-Point Iteration

$$out[i] = gen[i] \cup (in[i] \setminus kill[i]) \quad (1)$$
$$in[i] = \bigcap_{j \in pred[i]} out[j] \quad (2)$$

Initialized to the set of all assignments, except for $in[1] = \emptyset$.

	Initialisation		Iteration 1		Iteration 2	
i	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$
1		1, 2, 5, 6, 7, 8, 9, 10		1		1
2	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1	1, 2	1	1, 2
3	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2	1, 2	1, 2	1, 2
4	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2	1, 2	2	2
5	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2	1, 2, 5	2	2, 5
6	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5	1, 2, 5, 6	2, 5	2, 5, 6
7	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6	1, 2, 5, 6, 7	2, 5, 6	2, 5, 6, 7
8	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7	1, 2, 5, 6, 7, 8	2, 5, 6, 7	2, 5, 6, 7, 8
9	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8	1, 2, 5, 6, 7, 8, 9	2, 5, 6, 7, 8	2, 5, 6, 7, 8, 9
10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9	1, 2, 5, 6, 7, 8, 9, 10	2, 5, 6, 7, 8, 9	2, 5, 6, 7, 8, 9, 10
11	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 8, 9, 10	2, 5, 6, 7, 8, 9, 10	2, 5, 6, 8, 9, 10
12	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 8, 9, 10	2, 6, 8, 10	2, 5, 6, 8, 9, 10	2, 6, 8, 10
13	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	2, 6, 8, 10	2, 6, 8, 10	2, 6, 8, 10	2, 6, 8, 10
14	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	2, 6, 8, 10	2, 6, 8, 10	2, 6, 8, 10	2, 6, 8, 10
15	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	2	2	2	2

Iteration 3 = iteration 2.

Used in Common-Subexpression Elimination

The computation 5: $b{:=}i{*}4$ is available at 9: $f{:=}i{*}4$.

1: $i := 0$
2: $a := n * 3$
3: IF $i < a$ THEN *loop* ELSE *end*
4: LABEL *loop*
5: $b := i * 4$
6: $c := p + b$
7: $d := M[c]$
8: $e := d * 2$
9: $f := i * 4$
10: $g := p + f$
11: $M[g] := e$
12: $i := i + 1$
13: $a := n * 3$
14: IF $i < a$ THEN *loop* ELSE *end*
15: LABEL *end*

Used in Common-Subexpression Elimination

The computation 5: $b{:=}i{*}4$ is available at 9: $f{:=}i{*}4$.

1: $i := 0$
2: $a := n * 3$
3: IF $i < a$ THEN *loop* ELSE *end*
4: LABEL *loop*
5: $b := i * 4$
6: $c := p + b$
7: $d := M[c]$
8: $e := d * 2$
9: $f := b$
10: $g := p + f$
11: $M[g] := e$
12: $i := i + 1$
13: $a := a$
14: IF $i < a$ THEN *loop* ELSE *end*
15: LABEL *end*

Used in Common-Subexpression Elimination

The computation 5: $b{:=}i{*}4$ is available at 9: $f{:=}i{*}4$.

1: $i := 0$
2: $a := n * 3$
3: IF $i < a$ THEN *loop* ELSE *end*
4: LABEL *loop*
5: $b := i * 4$
6: $c := p + b$
7: $d := M[c]$
8: $e := d * 2$
9: $f := b$
10: $g := p + f$ $\leftarrow$ will not be eliminated
11: $M[g] := e$
12: $i := i + 1$
13: $a := a$
14: IF $i < a$ THEN *loop* ELSE *end*
15: LABEL *end*

1. Data-Flow Analysis
   - Common-Subexpression Elimination
   - Jump-to-Jump Elimination
   - Index-Checking Elimination
2. Loop Optimizations
   - Hoisting Loop-Invariant Computation
   - Prefetching
3. Function Calls
   - Inlining
   - Tail-Call Optimization
4. Specialization

Jump-to-Jump Elimination

Avoid successive jumps: `[... GOTO` l_1`, ..., LABEL` l_1`, GOTO` l_2`...]`.

instruction	*gen*	*kill*
`LABEL` l	$\{l\}$	$\emptyset$
`GOTO` l	$\emptyset$	$\emptyset$
`IF` c `THEN` l_1 `ELSE` l_2	$\emptyset$	$\emptyset$
any other	$\emptyset$	the set of all labels

$$in[i] = \begin{cases} gen[i] \setminus kill[i] & \text{if } out[i] \text{ is empty} \\ out[i] \setminus kill[i] & \text{if } out[i] \text{ is non-empty} \end{cases} \tag{3}$$

$$out[i] = \bigcap_{j \in succ[i]} in[j] \tag{4}$$

A jump i: `GOTO` l can be replaced with i: `GOTO` l', if $l' \in in[i]$.

1. Data-Flow Analysis
   - Common-Subexpression Elimination
   - Jump-to-Jump Elimination
   - Index-Checking Elimination
2. Loop Optimizations
   - Hoisting Loop-Invariant Computation
   - Prefetching
3. Function Calls
   - Inlining
   - Tail-Call Optimization
4. Specialization

Index-Check Elimination

Checks if `i` is within the array-size bounds when used in `a[i]`.

Idea: use `IF-THEN-ELSE` to check bounds and analyze whether the condition reduces statically to *true* or *false*.

Example: If a's lowest/highest index is 0/10, translate
`for i:=0 to 9 do a[i]:=0;` to:

```
i := 0
LABEL for1
IF i <= 9 THEN for2 ELSE for3
LABEL for2
IF i < 0 THEN error ELSE ok1
LABEL ok1
IF i > 10 THEN error ELSE ok2
LABEL ok2
t := i * 4
t := a + t
M[t] := 0
i := i + 1
GOTO for1
LABEL for3
```

Inequalities

Collect inequalities of the form $p \leq q$ and $p < q$, where p and q are either variables or constants.

In order to ensure a finite number of inequalities, use an universe Q of inequalities derived from program's condition(al)s. For example:

- $when(x < 10) = \{x < 10\}$, $whennot(x < 10) = \{10 \leq x\}$.
- $when(x = y) = \{x \leq y,\ y \leq x\}$, $whennot(x = y) = \emptyset$.

Our example program provides the following universe:

$$Q = \{i \leq 9,\ 9 < i,\ i < 0,\ 0 \leq i, 10 < i,\ i \leq 10\}$$

Fixpoint-iteration computes $in[i] =$ the set of inequalities (from Q) that are true (hold) at the beginning of instruction i.

Equations for Inequalities

$$
in[i] = \begin{cases}
\bigcap_{j \in pred[i]} in[j] & \text{if } pred[i] \text{ has more than one element} \\
in[pred[i]] \cup when(c) & \text{if } pred[i] \text{ is } \mathtt{IF}\ c\ \mathtt{THEN}\ i\ \mathtt{ELSE}\ j \\
in[pred[i]] \cup whennot(c) & \text{if } pred[i] \text{ is } \mathtt{IF}\ c\ \mathtt{THEN}\ j\ \mathtt{ELSE}\ i \\
(in[pred[i]] \setminus conds(Q,x)) \cup equal(Q,x,p) & \text{if } pred[i] \text{ is of the form } x := p \\
in[pred[i]] \setminus upper(Q,x) & \text{if } pred[i] \text{ is of the form } x := x + k \text{ where } k \geq 0 \\
in[pred[i]] \setminus lower(Q,x) & \text{if } pred[i] \text{ is of the form } x := x - k \text{ where } k \geq 0 \\
in[pred[i]] \setminus conds(Q,x) & \text{if } pred[i] \text{ is of a form } x := e \text{ not covered above} \\
in[pred[i]] & otherwise
\end{cases}
$$

$conds(Q, x)$: inequalities in x

$upper(Q, x)$: inequalities of form $x < p$ or $x \leq p$

$lower(Q, x)$: inequalities of form $p < x$ or $p \leq x$

$equal(Q, x, p)$: inequalities from Q, which are consequences of $x = p$

Limitations of Data-Flow Analysis

Can never be exact:

- many analysis problems are undecidable, i.e., one cannot solve them accurately.
- Tradeoff between efficient computation and precision.
- Use conservative approximation: optimize only when you are sure the assumptions hold.

Hoisting Loop-Invariant Computation

A term is loop invariant if it is computed inside a loop but has the same value at each iteration.

Solution: Unroll the loop once and do common-subexpression elimination.

The loop-invariant term is now computed in the unrolled part and reused in the subsequent loop. Example:

```
while (c) {                  if (c) {
  body              ⟶          body;
}                              while (c) {
                                 body
                               }
                             }
```

```
do                           body;
  body              ⟶        while (c) {
while (c)                      body
                             }
```

Disadvantage: Code size.