Loops

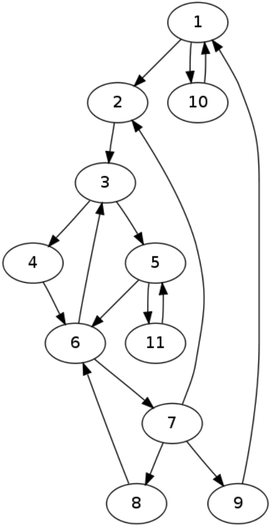
Definition

A back edge is a CFG edge whose target dominates its source.

Definition

A natural loop for back edge $t \rightarrow h$ is a subgraph containing t and h, and all nodes from which t can be reached without passing through h.

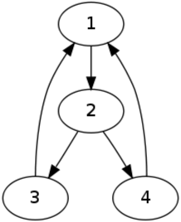
Example



Loops

Definition

The loop for a header h is the union of all natural loops for back edges whose target is h.



Property

Two loops with different headers $h_1 \neq h_2$ are either

- disjoint ($\text{loop}(h_1) \cap \text{loop}(h_2) = \{\}$), or
- nested within each other ($\text{loop}(h_1) \subset \text{loop}(h_2)$).

Loops

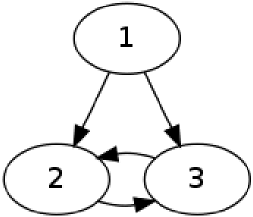
Definition

A subgraph of a graph is strongly connected if there is a path in the subgraph from every node to every other node.

Property

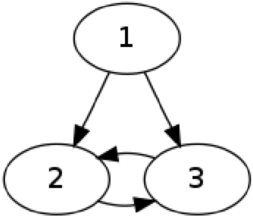
Every loop is a strongly connected subgraph. (Why?)

Example



Is $\{2, 3\}$ a strongly connected subgraph?
Is $\{2, 3\}$ a loop?

Example



Is $\{2, 3\}$ a strongly connected subgraph?
Is $\{2, 3\}$ a loop?

Definition

A CFG is reducible if every strongly connected subgraph contains a unique node (the header) that dominates all nodes in the subgraph.

Loop-invariant computations

Definition

A definition $c = a\ op\ b$ is loop-invariant if a and b

1. are constant,
2. have all their reaching definitions outside the loop, OR
3. have only one reaching definition (why?) which is loop-invariant.

Loop-invariant code motion

```
read i;
x = 1;
y = 2;
t = 2;
while(i<10) {
    t = y - x;
    i = i + t;
}
print t;
```

Loop-invariant code motion

```
read i;
x = 1;
y = 2;
t = 2;
t = y - x;
while(i<10) {
    i = i + t;
}
print t;
```

Loop-invariant code motion

```
read i;
if (cond) {
   x = 1;
} else {
   x = 2;
}
y = 3;
t = 2;
while(i<10) {
    t = y - x;
    i = i + t;
}
print t;
```

Loop-invariant code motion

```
read i;
if (cond) {
   x = 1;
} else {
   x = 2;
}
y = 3;
t = 2;
t = y - x;
while(i<10) {
    i = i + t;
}
print t;
```

Loop-invariant code motion

It is safe to move a computation $\ell : c = a\ op\ b$ to just before the header of the loop if

1. it is loop-invariant,
2. it has no side-effects,
3. c is not live immediately before the loop header,
4. ℓ is the only definition of c in the loop, and
5. ℓ dominates all exits from the loop at which c is live.

Note: 3 and 4 imply 5.

On Side effects

```
while(false) {
    i = 2 / 0;
}
```

Side Effects: Guard using a condition

```
while(cond) {
  i = 2 / 0;
  body;
}
```

```
if(cond) {
  i = 2 / 0;
  do {
    body;
  } while(cond)
}
```

Loop inversion

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

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

Loop inversion

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

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

```
L1:
if(!c) goto L2;
body;
goto L1;
L2:
```

```
if(!c) goto L2;
L1:
body;
if(c) goto L1;
L2:
```

Induction Variables

```
for(i = 0; i < 100; i++) {
  A[i] = 2*i;
}
```

```
i = 0;
L1:
if (i >= 100) goto L2;
t1 = i * 4;
t2 = t1 + A;
t3 = 2 * i;
*t2 = t3;
i = i + 1;
goto L1;
L2:
```

Induction Variables

Definition

Variable `i` is a basic induction variable if all its definitions in the loop are of the form `i = i + c`, where `c` is loop-invariant.

Definition

Variable `j` is a derived induction variable in the family of `i` if `i` is a basic induction variable, and `j = c*i + d` at every use of `j` in the loop, where `c` and `d` are loop-invariant.

Identifying Derived Induction Variables

IF

- `i` is a basic induction variable,
- there is only one definition of `k`, AND
- it has the form `k=i*c` or `k=i+c`, where `c` is loop-invariant

THEN `k` is a derived induction variable in the family of `i`.

Identifying Derived Induction Variables

IF

- `i` is a basic induction variable,
- there is only one definition of `k`, AND
- it has the form `k=i*c` or `k=i+c`, where `c` is loop-invariant

THEN `k` is a derived induction variable in the family of `i`.

IF

- `j` is a derived induction variable in the family of `i`,
- there is only one definition of `k`,
- it has the form `k=j*c` or `k=j+c`, where `c` is loop-invariant, AND
- there is no def of `i` on any path from the def of `j` to the def of `k`

THEN `k` is a derived induction variable in the family of `i`.

What the second condition really means

```
//a, b, c, d, e, f are loop invariant
while (cond){
     ........
     i = i + a;
     ......
     j = c * i + d;
     ....../Code Block A
     k = e * j + f;
}
```

i: Basic Induction Variable

j: Derived Induction Variable in the family of i

k: Derived Induction Variable in the family of i as long as there is no definition of i in Code Block A

Strength Reduction of Derived Induction Variables

```
i = i + a;  // < i, 1, a >
j = c * i + d ; // < i, c, d >
k = e * j + f ; // < j, e, f >
```

Since k is a derived Induction Variable in the family of i, we can write the tuple in terms of i

```
k =  e * j + f
  = e * ( ci + d) + f
  = eci + ed + f
  = ec * i + (ed + f) ; // < i , ec , ed + f >
```

Initialize k to eci + ed + f. Every time i is incremented by a, increment k by e*c*a (constant)

Strength Reduction of Derived Induction Variables

Assume `j` is a DIV in the family of `i`, such that `j = c*i + d`.

1. After each definition `i = i + e`, insert `j' = j' + c*e`.
2. Replace definition of `j` with `j = j'`.
3. Insert `j' = c*i + d` immediately before loop header.

Do copy propagation afterwards.

Strength Reduction of Derived Induction Variables

```
i = 0;
L1:
if (i >= 100) goto L2;
t1 = i * 4;
t2 = t1 + A;
t3 = 2 * i;
*t2 = t3;
i = i + 1;
goto L1;
L2:
```

```
i = 0;
t1' = i * 4;
t2' = i * 4 + A;
t3' = i * 2;
L1:
if (i >= 100) goto L2;
t1 = t1'
t2 = t2'
t3 = t3'
*t2 = t3;
i = i + 1;
t1' = t1' + 4;
t2' = t2' + 4;
t3' = t3' + 2;
goto L1;
L2:
```

Copy Propagation after Strength Reduction

```
i = 0;
t1' = i * 4;
t2' = i * 4 + A;
t3' = i * 2;
L1:
if (i >= 100) goto L2;
t1 = t1'
t2 = t2'
t3 = t3'
*t2 = t3;
i = i + 1;
t1' = t1' + 4;
t2' = t2' + 4;
t3' = t3' + 2;
goto L1;
L2:
```

```
i = 0;
t1' = i * 4;
t2' = i * 4 + A;
t3' = i * 2;
L1:
if(i >= 100) goto L2;
*t2' = t3';
i = i + 1;
t1' = t1' + 4;
t2' = t2' + 4;
t3' = t3' + 2;
goto L1;
L2:
```

Useless Induction Variables

Definition

An induction variable is useless if

- it is dead at the loop exits, AND
- it is used only in its own definition.

Definition

An induction variable is almost useless if

- it is dead at the loop exits,
- it is used only in its own definition and in comparisons with loop constants, AND
- some other variable in the same family is not useless.

Useless Induction Variables

```
i = 0;
t1' = i * 4;
t2' = i * 4 + A;
t3' = i * 2;
L1:
if(i >= 100) goto L2;
*t2' = t3';
i = i + 1;
t1' = t1' + 4;
t2' = t2' + 4;
t3' = t3' + 2;
goto L1;
L2:
```

```
i = 0;
t2' = i * 4 + A;
t3' = i * 2;
L1:
if(i >= 100) goto L2;
*t2' = t3';
i = i + 1;
t2' = t2' + 4;
t3' = t3' + 2;
goto L1;
L2:
```

Almost Useless Induction Variables

```
i = 0;
t1' = i * 4;
t2' = i * 4 + A;
t3' = i * 2;
L1:
if(i >= 100) goto L2;
*t2' = t3';
i = i + 1;
t2' = t2' + 4;
t3' = t3' + 2;
goto L1;
L2:
```

```
i = 0;
t2' = i * 4 + A;
t3' = i * 2;
L1:
if(t3' >= 200) goto L2;
*t2' = t3';
t2' = t2' + 4;
t3' = t3' + 2;
goto L1;
L2:
```

Induction Variables

```
for(i = 0; i < 100; i++) {
  A[i] = 2*i;
}
```

```
i = 0;
L1:
if (i >= 100) goto L2;
t1 = i * 4;
t2 = t1 + A;
t3 = 2 * i;
*t2 = t3;
i = i + 1;
goto L1;
L2:
```

```
t2 = A;
t3 = 0;
L1:
if (t3 >= 200) goto L2;
*t2 = t3;
t2 = t2 + 4;
t3 = t3 + 2;
goto L1;
L2:
```

Loop unrolling

```
while( i < c ) {
  body;
  i = i + 1;
}
```

```
while( i < c ) {
  body;
  i = i + 1;
  if( i >= c ) break;
  body;
  i = i + 1;
}
```

Loop unrolling

```
while( i < c ) {
  body;
  i = i + 1;
}
```

```
while( i < c ) {
  body;
  i = i + 1;
  if( i >= c ) break;
  body;
  i = i + 1;
}
```

- Increases code size
- More chance for optimizations within basic block

Loop unrolling

```
while( i < c ) {
  body;
  i = i + 1;
  if( i >= c ) break;
  body;
  i = i + 1;
}
```

```
while( i < c-1 ) {
  body;
  body;
  i = i + 2;
}
while( i < c ) {
  body;
  i = i + 1;
}
```