More Loop Unrolling and Vectorization

Loop Unrolling Review

```
     li r0 <- 0
     syscall IO.in_int
     li r2 <- 0
     li r3 <- 1
L1:  ble r1 r0 L2
     add r2 <- r2 r0
     add r0 <- r0 r3
     jmp L1
L2:  mov r1 <- r2
     syscall IO.out_int
```

Loop Unrolling Review

```
    li r0 <- 0
    syscall IO.in_int
    li r2 <- 0
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3
    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```

Goal: unroll this loop, without duplicating `ble`.

Unrolled loop runs for a multiple of the unrolling factor.

- `r0`, `r1`, and number of iterations determine if we have extra iterations

Data-Flow Analysis for Affine Expressions

Similar to constant propagation.

Direction: Forward

Values: (for each variable)

- Unknown (⊤)
- Affine expression $(c_0 + c_1 x_1 + c_2 x_2 + \cdots)$
- Not affine expression (⊥)

Meet operator:

- Let $v[x]$ be the data-flow value for variable x.
- Usual rules for ⊤.
- If $v_1[x] = v_2[x]$:
  - $(v_1 \wedge v_2)[x] = v_1[x]$
- Otherwise,
  - $(v_1 \wedge v_2)[x] = \bot$

Data-Flow Analysis for Affine Expressions

Statement	Transfer Function
`la` x <- c	$f_s(v)[x] = c$
`li` x <- c	$f_s(v)[x] = c$
`ld` x <- $y[c]$	$f_s(v)[x] = v\big[y[c]\big]$
`mov` x <- y	$f_s(v)[x] = v[y]$
`add` x <- y z	$f_s(v)[x] = v[y] + v[z]$
`mul` x <- y z	$f_s(v)[x] = v[y] \cdot v[z]$ (if $v[y] = c$ or $v[z] = c$)
`div` x <- y z	$f_s(v)[x] = v[y]/v[z]$ (if $v[z] = c$ and $v[z] \neq 0$)

Data-Flow Analysis for Affine Expressions

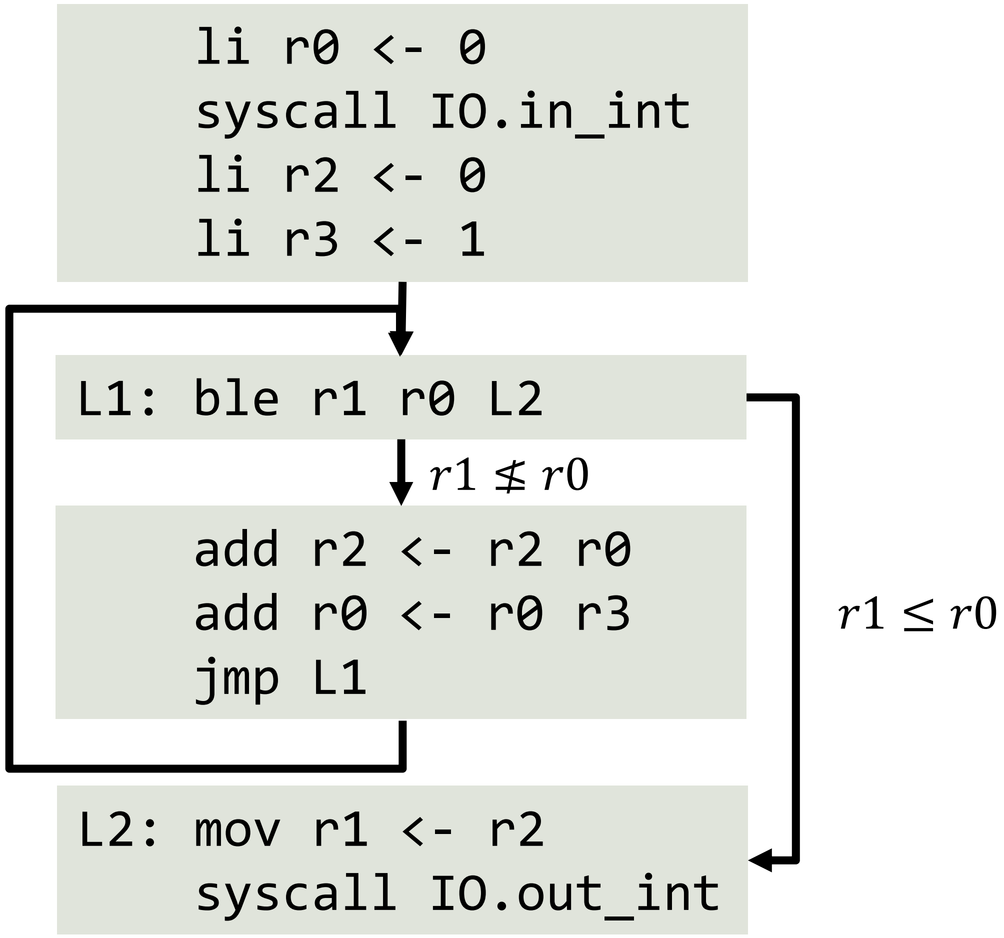
Statement	Transfer Function
`la` x `<-` c	$f_s(v)[x] = c$
`li` x `<-` c	$f_s(v)[x] = c$
`ld` x `<-` $y[c]$	$f_s(v)[x] = v[y[c]]$
`mov` x `<-` y	$f_s(v)[x] = v[y]$
`add` x `<-` y z	$f_s(v)[x] = v[y] + v[z]$
`mul` x `<-` y z	$f_s(v)[x] = v[y] \cdot v[z]$ (if $v[y] = c$ or $v[z] = c$)
`div` x `<-` y z	$f_s(v)[x] = v[y]/v[z]$ (if $v[z] = c$ and $v[z] \neq 0$)

$$\left.\begin{matrix} v[y] = \bot \\ \textit{or} \\ v[z] = \bot \end{matrix}\right\} \Rightarrow f_s(v)[x] = \bot$$

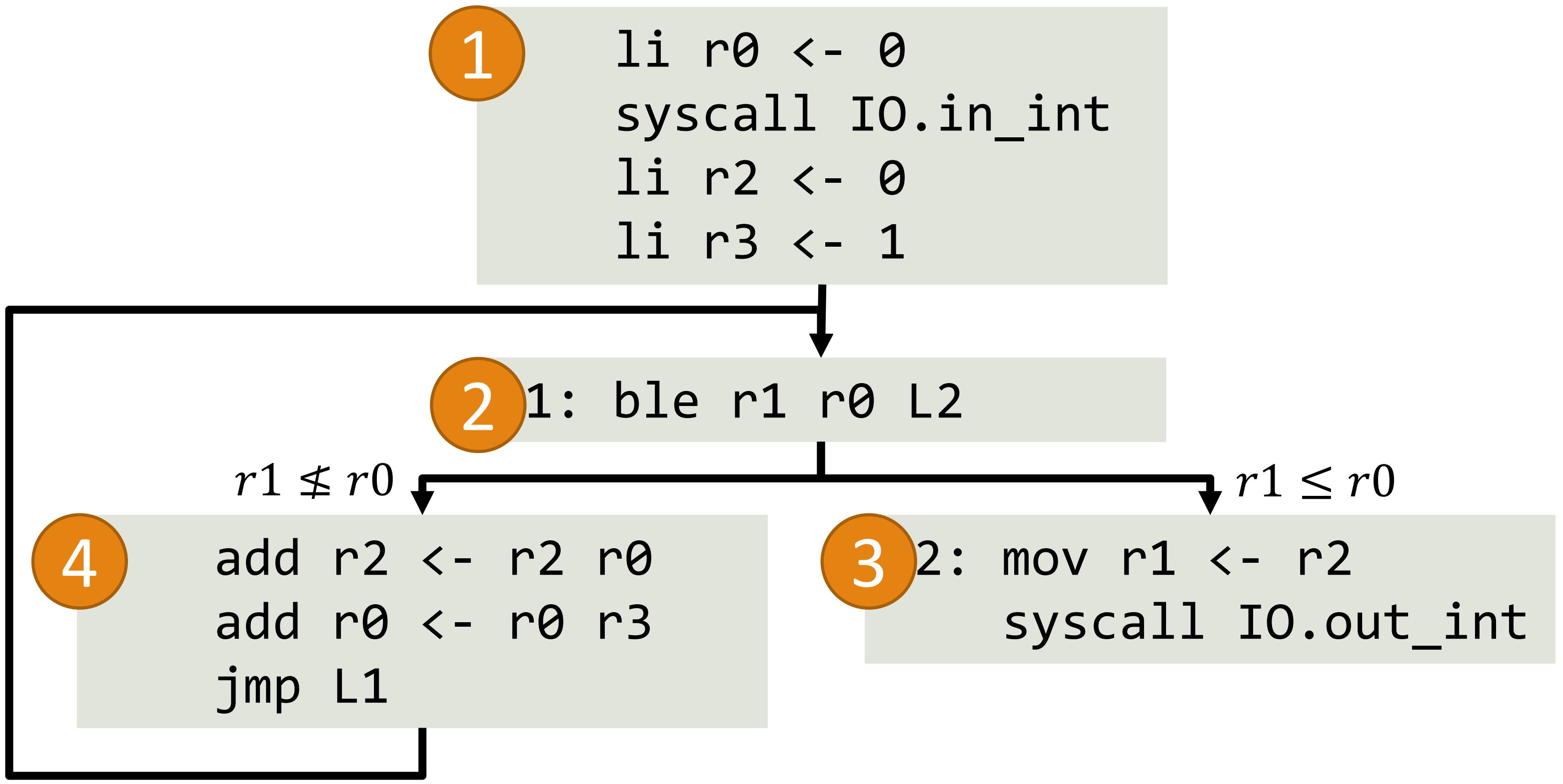
Loop Example

```
    li r0 <- 0
    syscall IO.in_int
    li r2 <- 0
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3
    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```

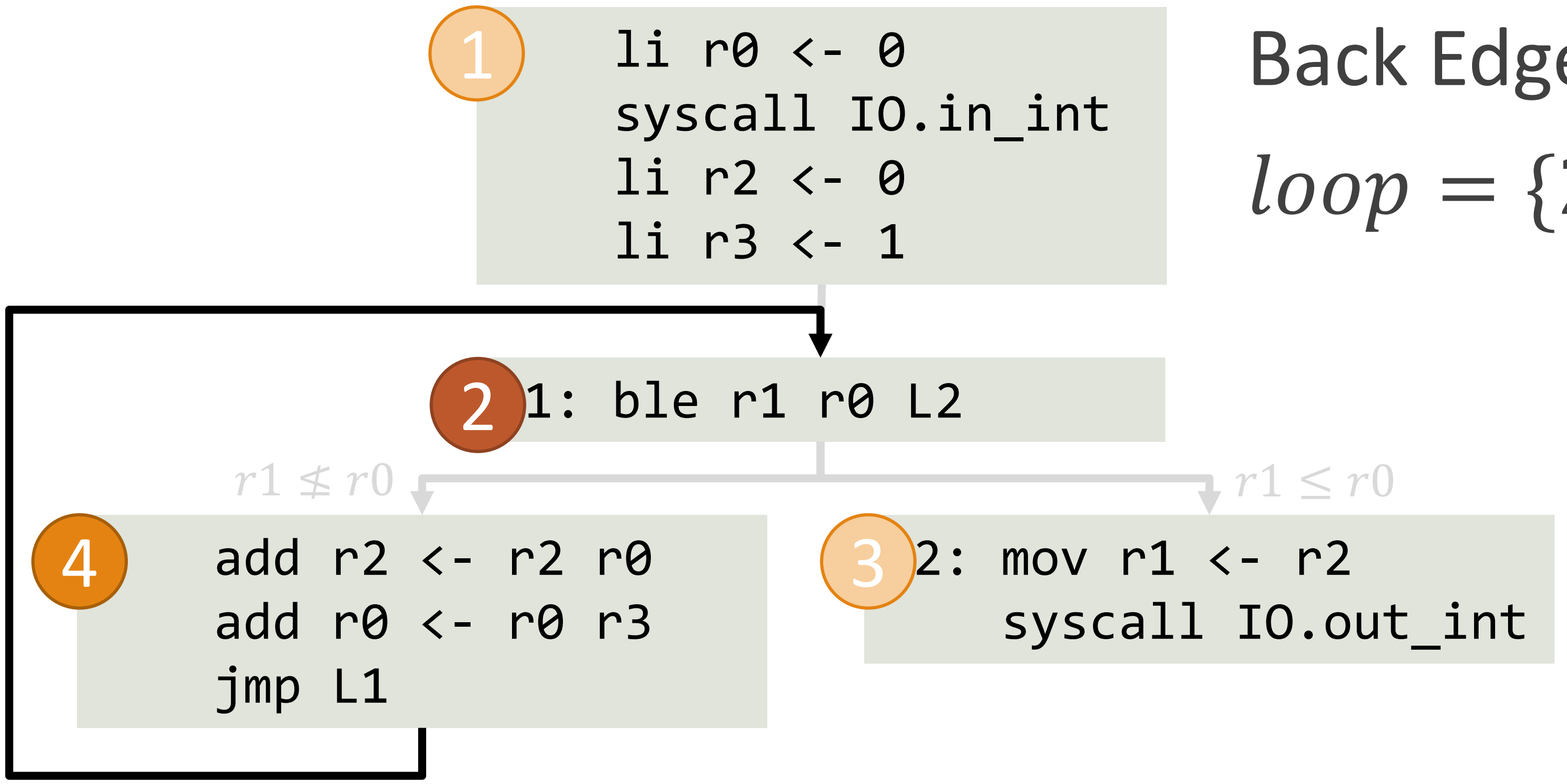
Loop Example (CFG)



Loop Example (DFS Tree)



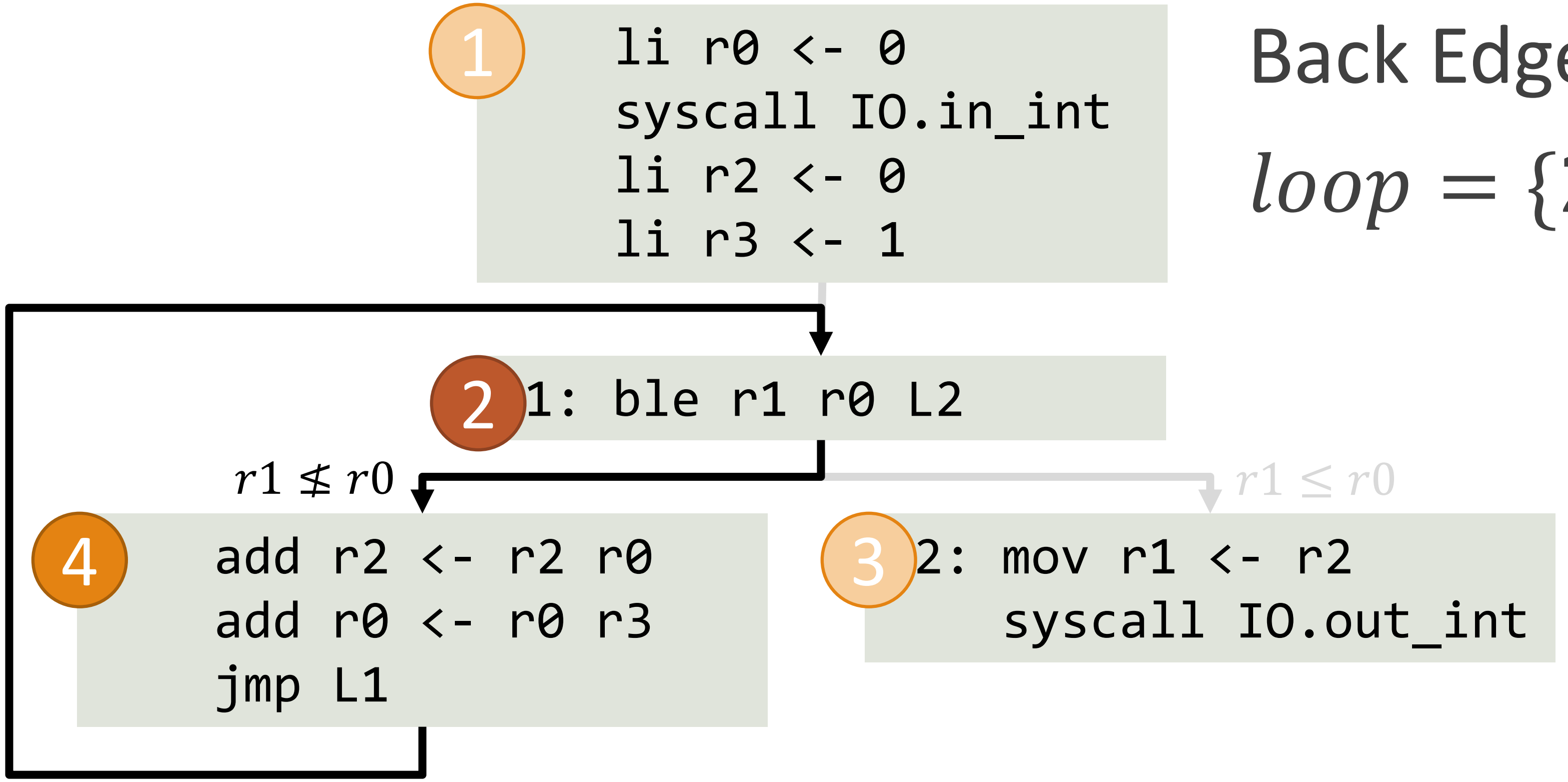
Loop Example (Loop Detection)



Back Edge: $4 \rightarrow 2$

$loop = \{2, 4\}$

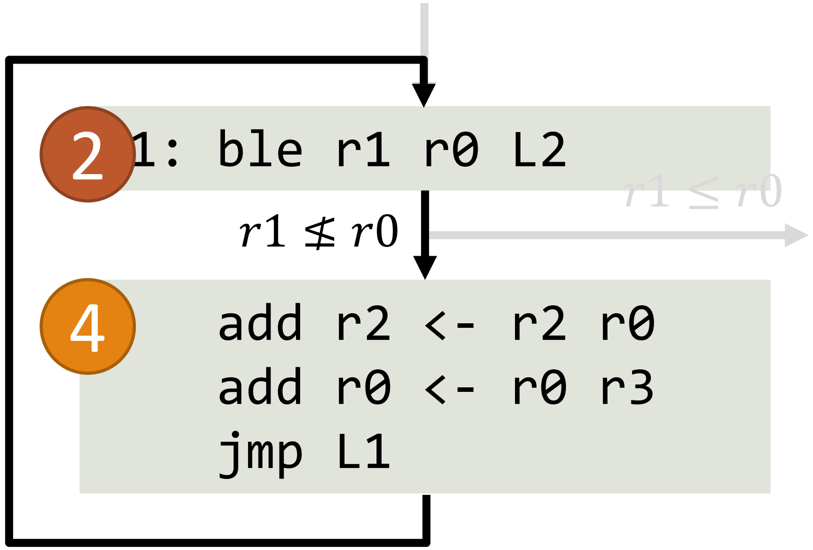
Loop Example (Loop Detection)



Back Edge: $4 \rightarrow 2$

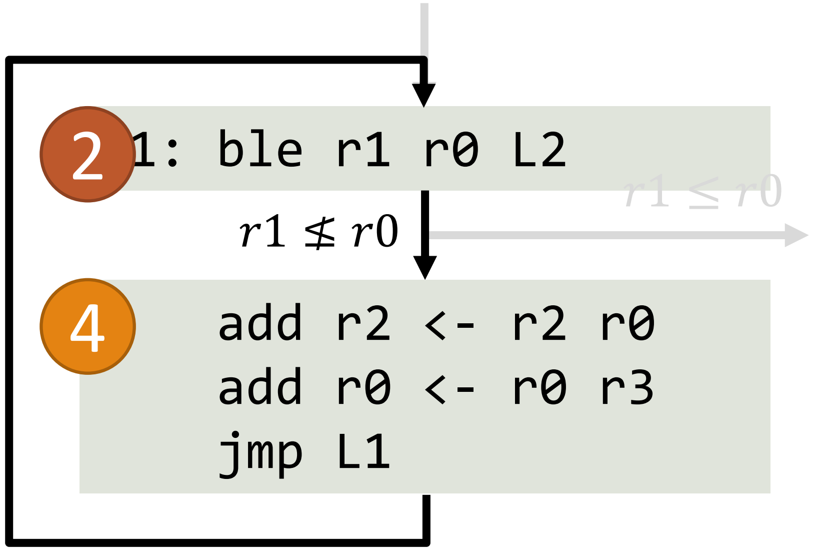
$loop = \{2, 4\}$

Loop Example (Data-Flow Analysis)



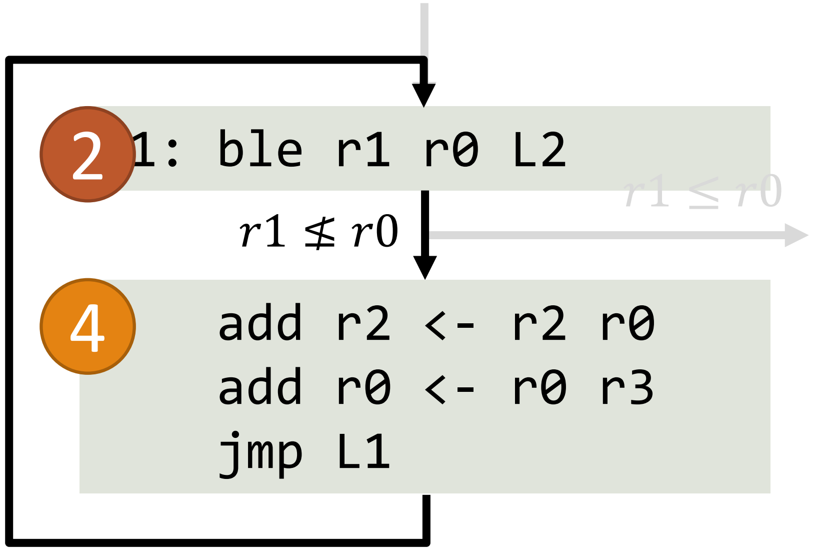
Var	$f_{B_2}(v)$	$f_{B_4}(v)$
r0		
r1		
r2		
r3		

Loop Example (Data-Flow Analysis)



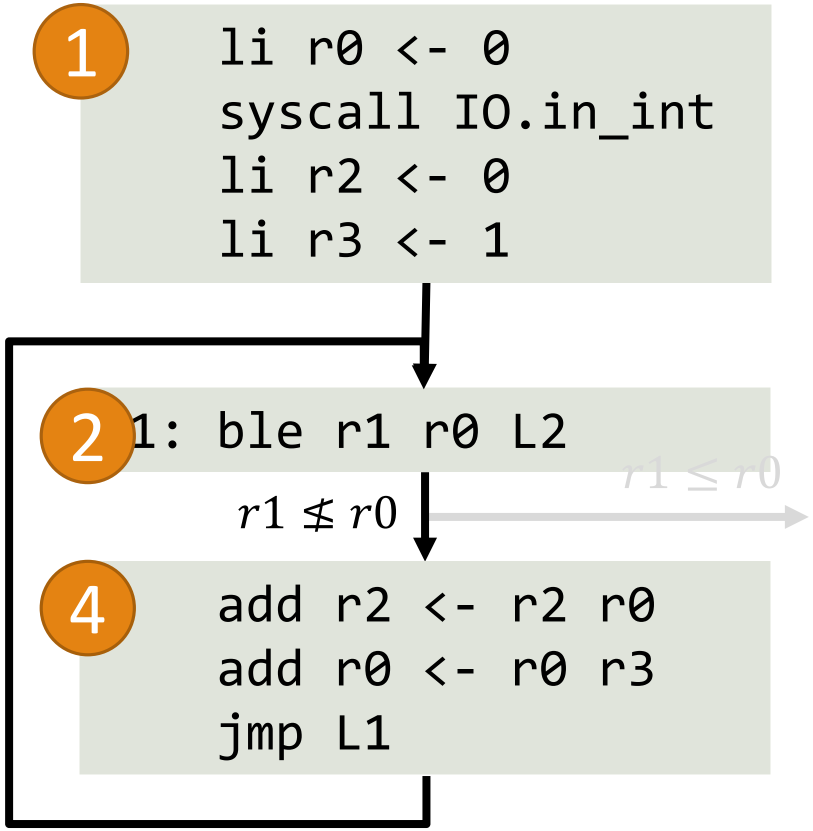
Var	$f_{B_2}(v)$	$f_{B_4}(v)$
r0	$v[r0]$	
r1	$v[r1]$	
r2	$v[r2]$	
r3	$v[r3]$	

Loop Example (Data-Flow Analysis)



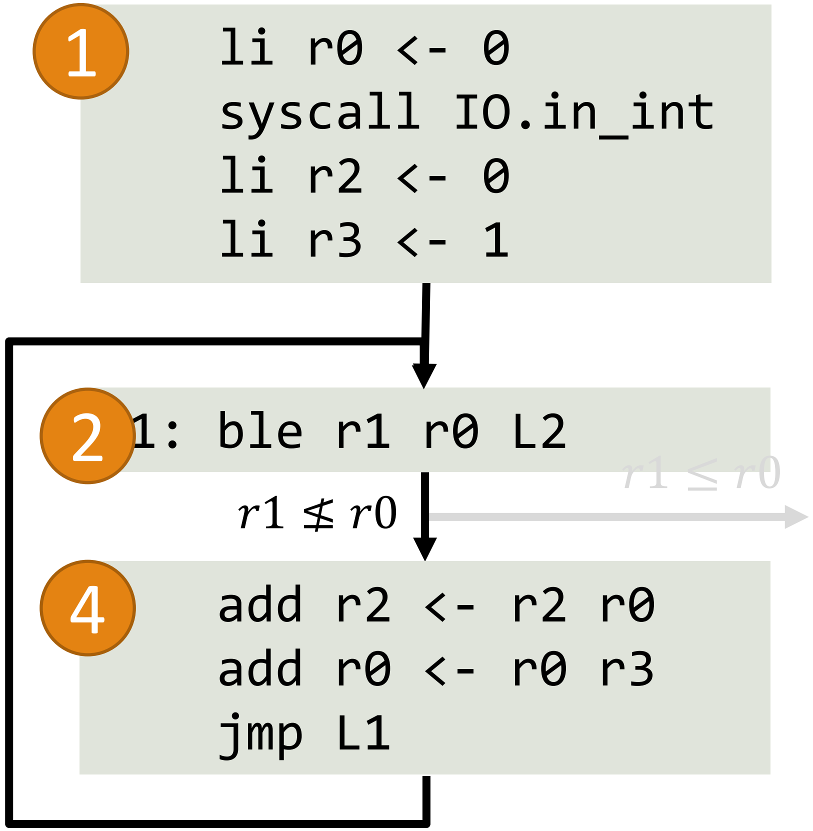
Var	$f_{B_2}(v)$	$f_{B_4}(v)$
r0	$v[r0]$	$v[r0] + v[r3]$
r1	$v[r1]$	$v[r1]$
r2	$v[r2]$	$v[r2] + v[r0]$
r3	$v[r3]$	$v[r3]$

Loop Example (Data-Flow Analysis)



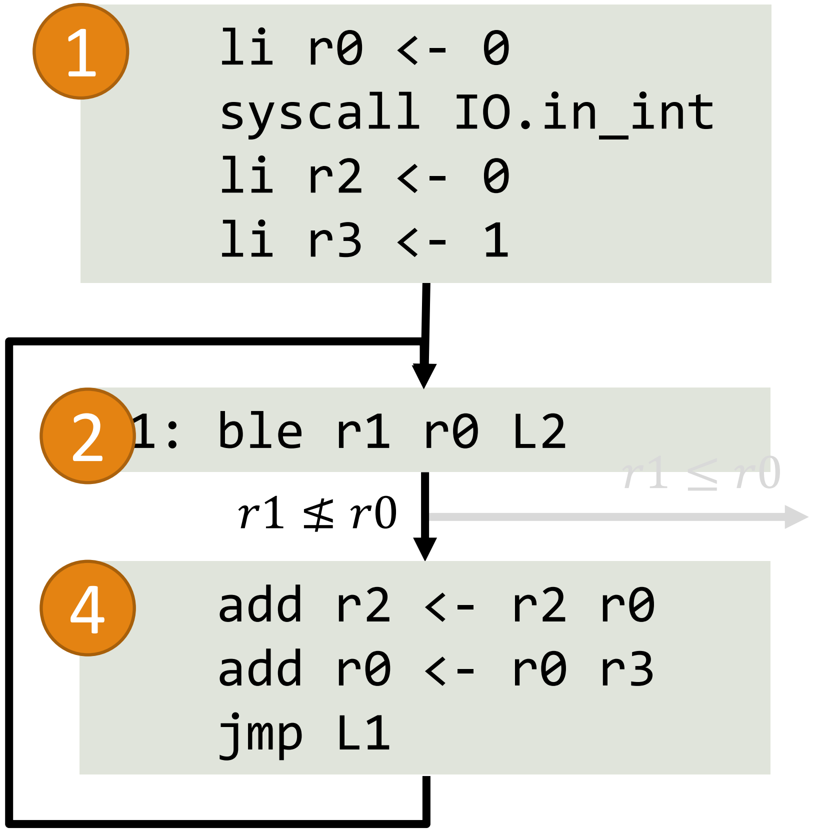
Var	IN[B2]	OUT[B4]
r0		
r1		
r2		
r3		

Loop Example (Data-Flow Analysis)



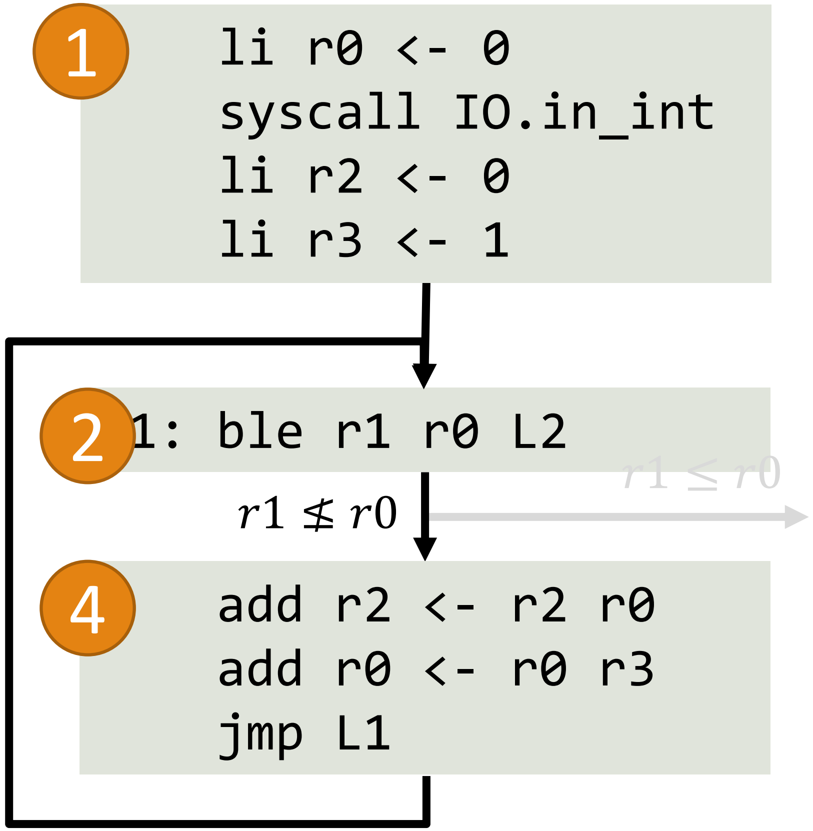
Var	IN[B2]	OUT[B4]
r0	$0 \wedge \top = 0$	1
r1	$\bot \wedge \top = \bot$	$\bot$
r2	$0 \wedge \top = 0$	0
r3	$1 \wedge \top = 1$	1

Loop Example (Data-Flow Analysis)



Var	IN[B2]	OUT[B4]
r0	$0 \wedge 1 = \bot$	$\bot$
r1	$\bot \wedge \bot = \bot$	$\bot$
r2	$0 \wedge 0 = 0$	$\bot$
r3	$1 \wedge 1 = 1$	1

Loop Example (Data-Flow Analysis)



Var		OUT[B4]
r0	$0 \wedge \bot = \bot$	$\bot$
r1	$\bot \wedge \bot = \bot$	$\bot$
r2	$0 \wedge \bot = \bot$	$\bot$
r3	$1 \wedge 1 = 1$	1

Total failure!

Iterated Transfer Functions

Track data-flow values as functions of number of iterations.

- After 1 iteration:
  $f_{B_4}^{1}(v_0)[r0] = v_0[r0] + v_0[r3] = v_0[r0] + 1$
- After 2 iterations:
  $f_{B_4}^{2}(v_0)[r0] = (v_0[r0] + 1) + 1 = v_0[r0] + 2$
- After i iterations:
  $f_{B_4}^{i}(v_0)[r0] = v_0[r0] + i$

Handling Iteration

Symbolic constants:

- If $f(v)[x] = v[x]$, $f^i(v_0)[x] = v_0[x]$

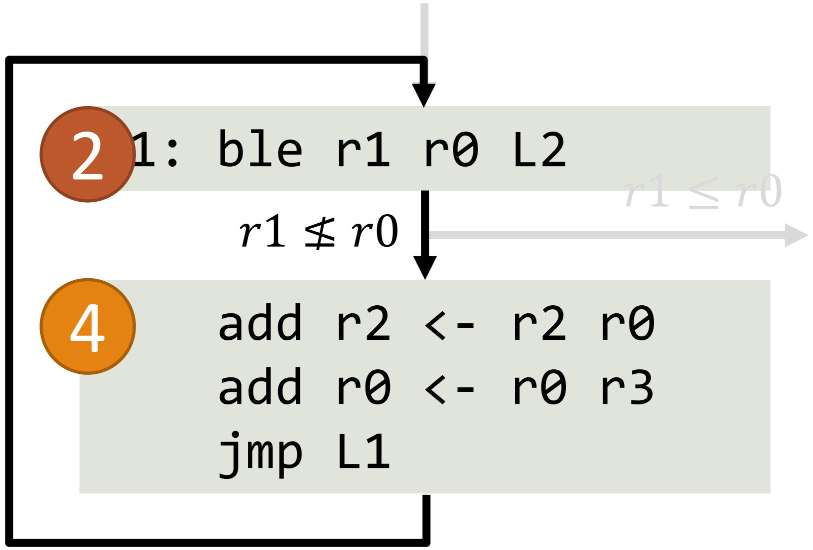
Basic induction variables:

- If $f(v)[x] = c + v[x]$, $f^i(v_0)[x] = ci + v_0[x]$

Induction variables (if y_1 … are basic induction variables or symbolic constants and $x \not\equiv y_i$):

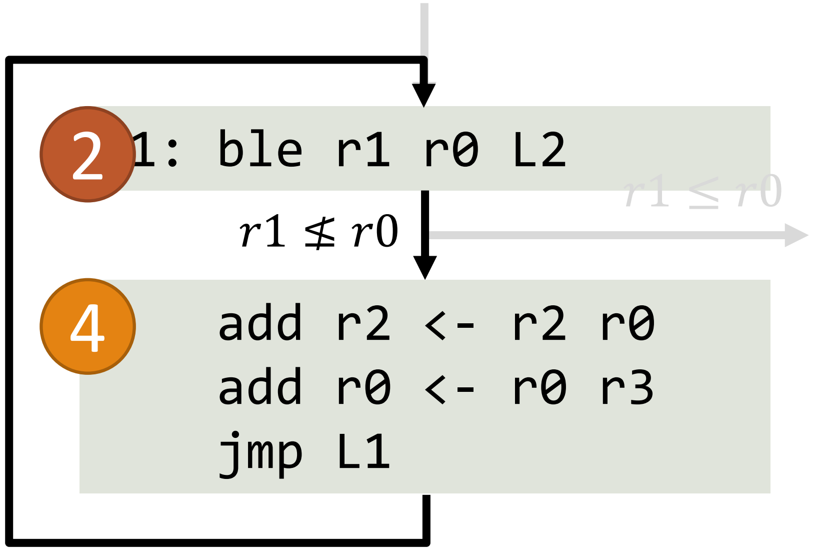
- If $f(v)[x] = c_0 + c_1 v[y_1] + \cdots$, $f^i(v_0)[x] = c_0 + c_1 f^i(v_0)[y_1] + \cdots$

Loop Example (Data-Flow Analysis)



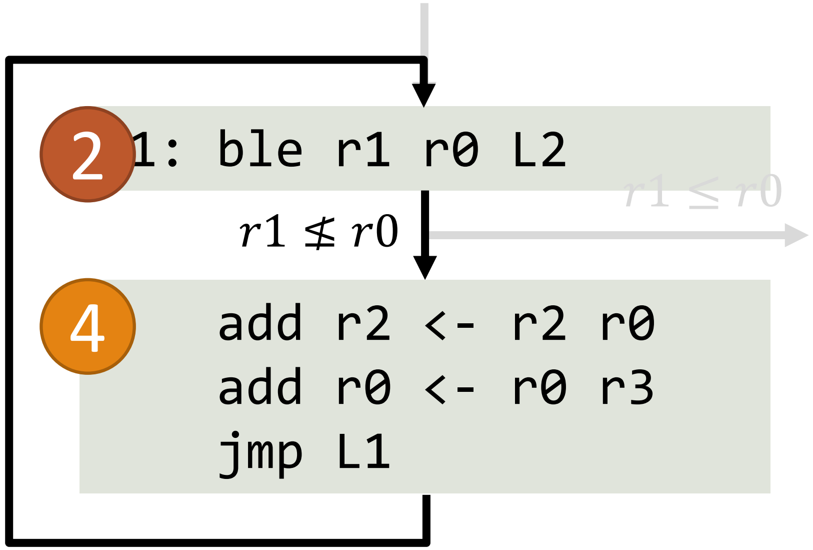
Var	$f_{B_4}(v)$	$f^i_{B_4}(v_0)$
r0	$v[r0] + v[r3]$	
r1	$v[r1]$	
r2	$v[r2] + v[r0]$	
r3	$v[r3]$	

Loop Example (Data-Flow Analysis)



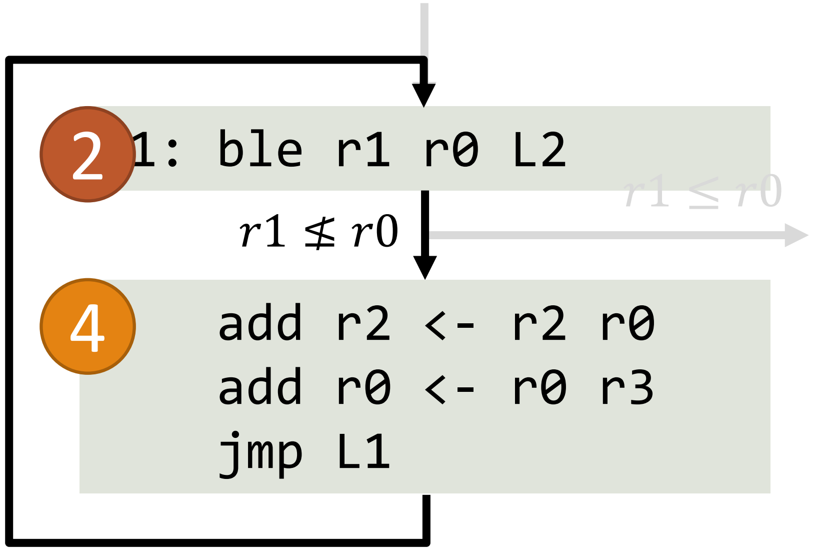
Var	$f_{B_4}(v)$	$f^{i}_{B_4}(v_0)$
r0	$v[r0] + v[r3]$	
r1	$v[r1]$	$v_0[r1]$
r2	$v[r2] + v[r0]$	
r3	$v[r3]$	$v_0[r3]$

Loop Example (Data-Flow Analysis)



Var	$f_{B_4}(v)$	$f_{B_4}^{i}(v_0)$
r0	$v[r0] + v[r3]$	$v_0[r0] + v_0[r3]i$
r1	$v[r1]$	$v_0[r1]$
r2	$v[r2] + v[r0]$	
r3	$v[r3]$	$v_0[r3]$

Loop Example (Data-Flow Analysis)



Var	$f_{B_4}(v)$	$f^i_{B_4}(v_0)$
r0	$v[r0] + v[r3]$	$v_0[r0] + v_0[r3]i$
r1	$v[r1]$	$v_0[r1]$
r2	$v[r2] + v[r0]$	$\perp$
r3	$v[r3]$	$v_0[r3]$

Finding the Number of Iterations

Use f^i to compute value on back edges.

We want to find i_{max} such that:

$$f^i(v_0)[r1] \nleq f^i(v_0)[r0]$$

Finding the Number of Iterations

Use f^i to compute value on back edges.

We want to find i_{max} such that:

$$f^i(v_0)[r1] \nleq f^i(v_0)[r0]$$

$$v_0[r1] \nleq v_0[r0] + v_0[r3]i_{max}$$

$$\frac{v_0[r1] - v_0[r0]}{v_0[r3]} > i_{max}$$

$$v_0[r1] = i_{max} + 1$$

Loop Unrolling

```
    li r0 <- 0
    syscall IO.in_int
    li r2 <- 0
    li r3 <- 1
L1: ble r1 r0 L2
    add r2 <- r2 r0
    add r0 <- r0 r3
    jmp L1
L2: mov r1 <- r2
    syscall IO.out_int
```

Now we know initial value of `r1` sets number of iterations.

- Check it against the loop unrolling factor to handle extra iterations.

Loop Unrolling

```
    li r0 <- 0
    syscall IO.in_int
    li r2 <- 0
    li r3 <- 1
    li r4 <- 3; factor
    div r5 <- r1 r4
    mul r5 <- r5 r4
    sub r5 <- r1 r5
    bz r5 L1
    add r2 <- r2 r0
    add r0 <- r0 r3
    beq r5 r0 L1
    add r2 <- r2 r0
    add r0 <- r0 r3
L1: beq r1 r0 L2
```

Unrolling factor

r5 <- r1 mod r4

Handle extra iterations.

Auto-Vectorization

Automatic Vectorization

Similar to loop unrolling:

- Consecutive iterations with ***independent*** arithmetic.
- Perform arithmetic for several iterations together in vector.
- Usually implemented over arrays.

```
let x : List <- getlist() in
while not isvoid(x) loop {
    x.incrBy(2);
    x <- x.next();
} pool
```

Automatic Vectorization

Similar to loop unrolling:

- Consecutive iterations with ***independent*** arithmetic.
- Perform arithmetic for several iterations together in vector.
- Usually implemented over arrays.

Inline these.

```
let x : Lis       st() in
while not isvo d(x) loop {
    x.incrBy(2);
    x <- x.next();
} pool
```

Automatic Vectorization

Similar to loop unrolling:

- Consecutive iterations with ***independent*** arithmetic.
- Perform arithmetic for several iterations together in vector.
- Usually implemented over arrays.

Unroll this.

```
let x : List <- getlist() in
while not isvoid(x) loop {
    x.incrBy(2);
    x <- x.next();
} pool
```

Automatic Vectorization (Cool ASM)

```
    li t1 <- 2
L1: bz r0 L2
    ld t2 <- r0[3]  ; x.incrby(2)
    add t3 <- t2 t1
    st r0[3] <- t3
    ld t4 <- r0[4]  ; x<-x.next()
    ld t5 <- t4[3]  ; x.incrby(2)
    add t6 <- t5 t1
    st t4[3] <- t6
    ld r0 <- t4[4]  ; x<-x.next()
    jmp L1
```

Automatic Vectorization (Cool ASM)

```
    li t1 <- 2
L1: bz r0 L2
    ld t2 <- r0[3]   ; x.incrby(2)
    add t3 <- t2 t1
    st r0[3] <- t3
    ld t4 <- r0[4]   ; x<-
    ld t5 <- t4[3]   ; x.i
    add t6 <- t5 t1
    st t4[3] <- t6
    ld r0 <- t4[4]   ; x<-x.next()
    jmp L1
```

Code reordering

```
    li t1 <- 2
L1: bz r0 L2
    ld t4 <- r0[4]
    ld t2 <- r0[3]
    ld t5 <- t4[3]
    add t3 <- t2 t1
    add t6 <- t5 t1
    st r0[3] <- t3
    st t4[3] <- t6
    ld r0 <- t4[4]
    jmp L1
```

Automatic Vectorization (Cool ASM)

1. Group arithmetic together.
2. Pack temporaries in vector registers.
3. Replace add with vector-add.
4. Unpack vector result.

```
    li t1 <- 2
L1: bz r0 L2
    ld t4 <- r0[4]
    ld t2 <- r0[3]
    ld t5 <- t4[3]
    add t3 <- t2 t1
    add t6 <- t5 t1
    st r0[3] <- t3
    st t4[3] <- t6
    ld r0 <- t4[4]
    jmp L1
```

Automatic Vectorization (Cool ASM)

1. Group arithmetic together.
2. Pack temporaries in vector registers.
3. Replace add with vector-add.
4. Unpack vector result.

```
    li vr10 <- 2
    li vr11 <- 2
L1: bz r0 L2
    ld t4 <- r0[4]
    ld vr00 <- r0[3]
    ld vr01 <- t4[3]
    vadd vr0 <- vr0 vr1
    st r0[3] <- vr00
    st t4[3] <- vr01
    ld r0 <- t4[4]
    jmp L1
```

A Simple Interprocedural Analysis

A Simple Interprocedural Analysis

Idea: Treat method calls as control flow.

If method instance is known:

- Add CFG edge from call to top of method body.
- Add CFG edge from end of method to statement-after-call.
- Similar to inlining, but without the code bloat.

Extension: “clone” method’s CFG nodes for each invocation.

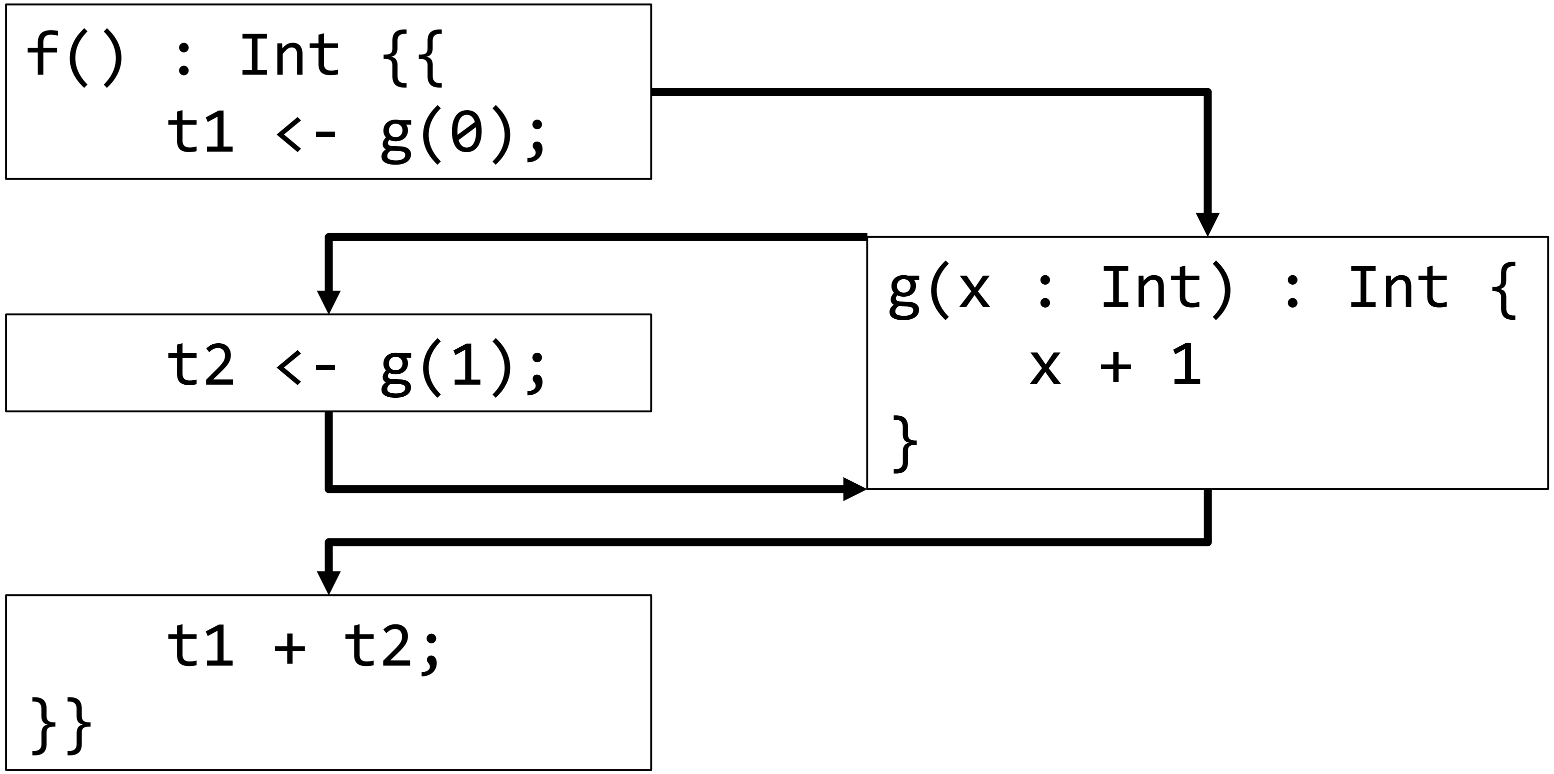
This analysis has difficulty with recursion.

Interprocedural Example

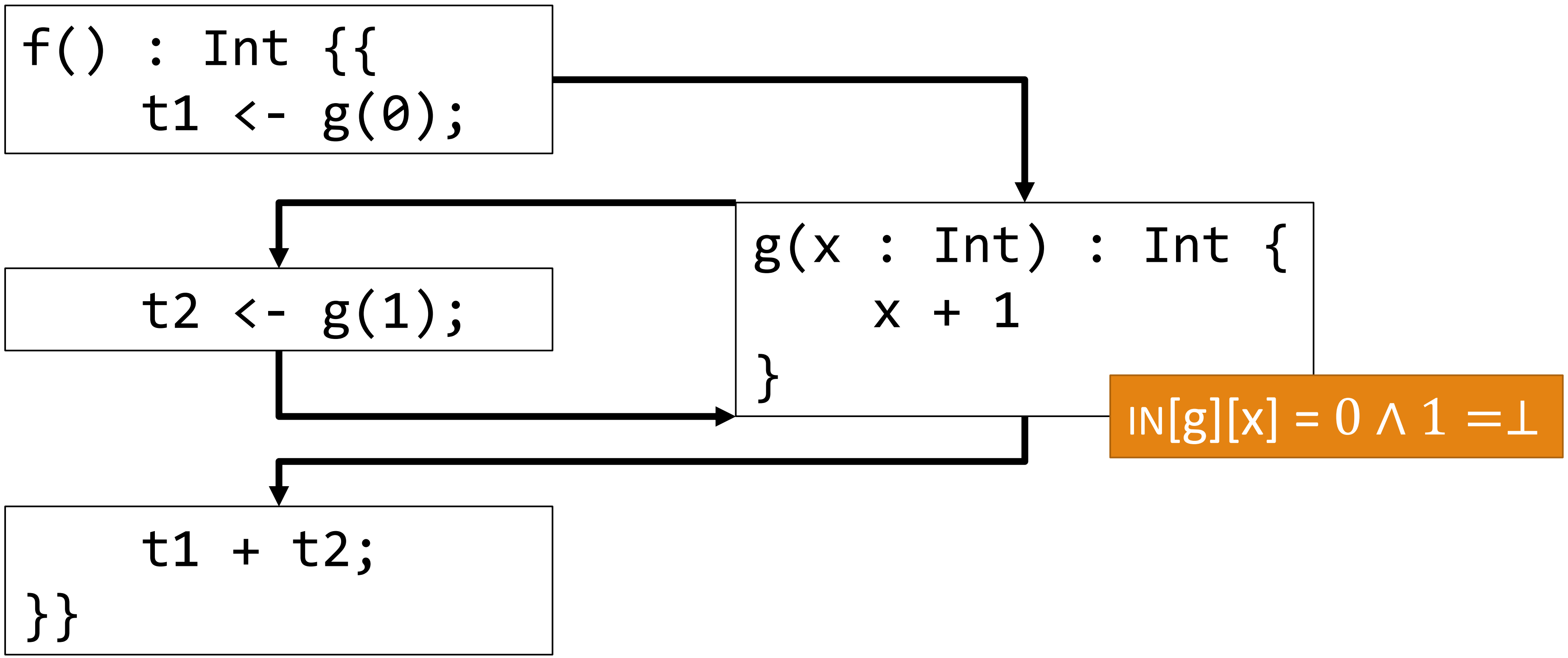
```
f() : Int {{
    t1 <- g(0);
    t2 <- g(1);
    t1 + t2;
}}
```

```
g(x : Int) : Int {
    x + 1
}
```

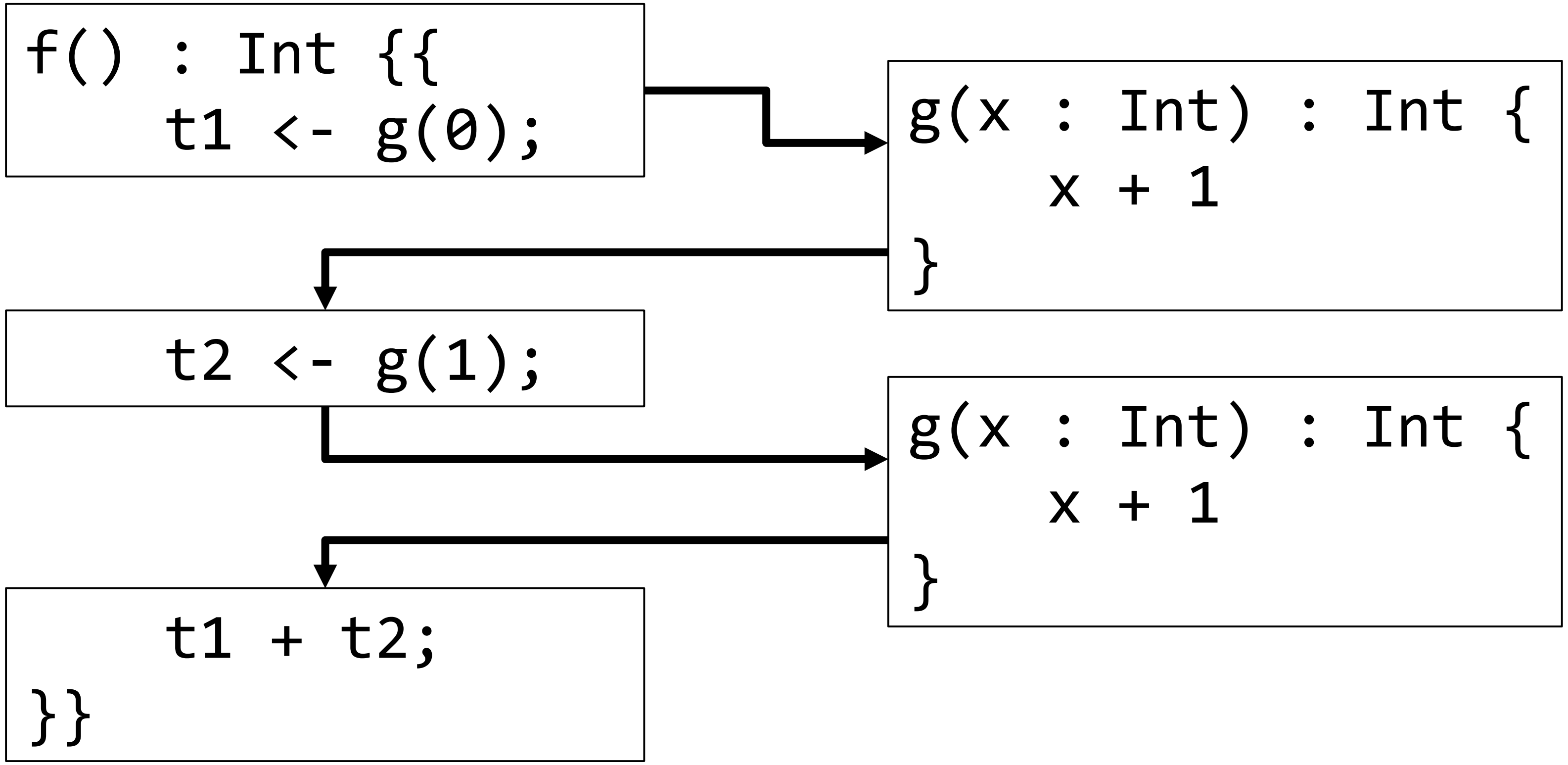
Interprocedural Example



Interprocedural Example



Interprocedural Example



Interprocedural Example