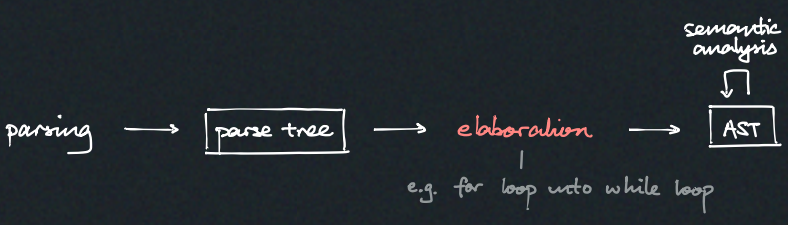
Lec 5 Liveness Analysis



Recall: temp t is live at line l if t could be used later

Inference Rules

Judgements: $live(l, x)$ — x live at line l

Rules: (↑ doesn't handle loops & conditionals)

$$\frac{l: d \leftarrow x \oplus y,\ d \neq u,\ live(l+1, u)}{live(l, u)}$$

$$\frac{l: x \leftarrow c,\ live(l+1, u),\ u \neq d}{live(l, u)} \qquad \frac{l: x \leftarrow y,\ u \neq x,\ live(l+1, u)}{live(l, u)}$$

$$\frac{l: x \leftarrow y}{live(l, y)} \qquad \frac{l: d \leftarrow x \oplus y}{live(l, x),\ live(l, y)}$$

$$\frac{l: return}{live(l, r_{ret})}$$

Saturation Algorithm — can use for all predicates

1. Start from fact = $\emptyset$
2. Pick arguments l and t, and some rule with $live(l, t)$ as conclusion and premises already in fact
3. Repeat until no facts can be derived

Analysis: worse $N_{iter} \leq N_{lines} \cdot N_{temps}$ (or N_{temps}^2 ?)

Refactoring rules

$$\frac{use(l, x)}{live(l, x)} \qquad \frac{live(l', x),\ succ(l, l'),\ \neg def(l, x)}{live(l, x)}$$

(possible successor; ⚠ when using not, be sure to saturate defs)

$use(l, x)$ — x used at l
$def(l, x)$ — x defined at l
$succ(l, l')$ — l' can be a successor of l

Ex. $\frac{l: d \leftarrow x \oplus y}{use(l, x),\ use(l, y),\ def(l, d),\ succ(l, l+1)}$ (if exists)

Loops & Conditionals

```
i := l: d ← x ⊕ y
     ⋮
   | l: goto l'
   | l: if (x ? c) then l_t else l_f
                  └ cmp op
```

$$\frac{l: goto\ l'}{succ(l, l')} \qquad \frac{l: if\ (x\ ?\ c)\ then\ l_t\ else\ l_f}{succ(l, l_t),\ succ(l, l_f),\ use(l, x)}$$

Ex (gcd) — bottom to top

		pass 1	pass 2	pass 3
l_1:	if $(x_2 \neq 0)$ then l_2 else l_8	x_1, x_2	–	–
l_2:	$q \leftarrow x_1 / x_2$	x_1, x_2	–	–
l_3:	$t \leftarrow x_2 \cdot q$	x_1, x_2, q	–	–
l_4:	$r \leftarrow x_1 - t$	x_1, t, x_2	–	–
l_5:	$x_1 \leftarrow x_2$	x_2, r	–	–
l_6:	$x_2 \leftarrow r$	r	x_1	–
l_7:	goto l_1	–	x_1, x_2	–
l_8:	ret $\leftarrow x_1$	x_1	–	–
l_9:	return	ret	–	–

Inefficient :(↑ line-oriented

→ Try variable-oriented approach

		x_1	x_2	–
l_1:	if $(x_2 \neq 0)$ then l_2 else l_8			
l_2:	$q \leftarrow x_1 / x_2$	x_2	x_1	–
l_3:	$t \leftarrow x_2 \cdot q$	x_2	x_1	q
l_4:	$r \leftarrow x_1 - t$	–	x_2	x_1 t
l_5:	$x_1 \leftarrow x_2$	–	r	x_2
l_6:	$x_2 \leftarrow r$	x_1	r	–
l_7:	goto l_1	x_1		x_2
l_8:	ret $\leftarrow x_1$	–	x_1	
l_9:	return	ret		

To build interference graph:

1. Overlapping in same row (⚠ doesn't work if there's dead code)

$$\frac{live(l, x),\ live(l, y)}{inter(x, y)}$$

(↑ must get chordal if in SSA.)

Ex. (b not used, reg alloc will use ret for a and b.)

$a \leftarrow 1$	
$b \leftarrow 2$	a
ret $\leftarrow a$	a
return	ret

2. Assignments-based (could still get non-chordal in SSA but more sparse)

$$\frac{x \leftarrow y \oplus z,\ live(l+1, u),\ u \neq x}{inter(u, x)}$$

$$\frac{l: x \leftarrow y,\ y \neq u,\ live(l+1, u),\ x \neq u}{inter(x, u)}$$

$$\frac{l: x \leftarrow c,\ u \neq x,\ live(l+1, u)}{inter(x, u)}$$

Elaboration

Goal: minimal, clear repr of programme

→ Remove syntactic sugar
→ Explicit scopes

By examples

```
for (int x = 4, x < 8128, x++) { y = y + x
  ↓
{ int x = 4;                    ↙ scope of x matters!
  while (x < 8128) ...        }
```

or in AST,

decl (x, int, while (x < 8128) { y = y+4, x = x + 1 })