CSC 6580
Spring 2020

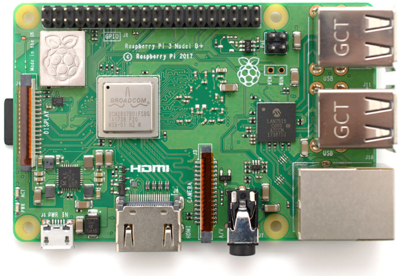
Instructor: Stacy Prowell

Mycroft
Type-Based Decompilation

Type-Based Decompilation*
(or Program Reconstruction via Type Reconstruction)

Alan Mycroft

Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK
http://www.cl.cam.ac.uk/users/am



Co-Founder of the
Raspberry Pi Foundation

Type “Reconstruction”

This is really the assignment of a *plausible* type.
The original types are lost in compilation.

instruction		generated constraint
`mov`	`r4,r6`	$t6 = t4$
`ld.w`	n`[r3],r5`	$t3 = \mathit{ptr}(\mathit{mem}(n : t5))$
`xor`	`r2a,r1b,r1c`	$t2a = \mathit{int}, t1b = \mathit{int}, t1c = \mathit{int}$
`add`	`r2a,r1b,r1c`	$t2a = \mathit{ptr}(\alpha), t1b = \mathit{int}, t1c = \mathit{ptr}(\alpha) \vee$
		$t2a = \mathit{int}, t1b = \mathit{ptr}(\alpha'), t1c = \mathit{ptr}(\alpha') \vee$
		$t2a = \mathit{int}, t1b = \mathit{int}, t1c = \mathit{int}$
`ld.w`	`(r5)[r0],r3`	$t0 = \mathit{ptr}(\mathit{array}(t3)), t5 = \mathit{int} \vee$
		$t0 = \mathit{int}, t5 = \mathit{ptr}(\mathit{array}(t3))$
`mov`	`#42,r7`	$t7 = \texttt{int}$
`mov`	`#0,r7`	$t7 = \texttt{int} \vee t7 = \mathit{ptr}(\alpha'')$

Iterative Source Code


```
;    int f(struct A *x)
;    {   int r = 0;
;        for (; x!=0; x = x->tl) r += x->hd;
;        return r;
;    }
;
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

Fig. 1. Iterative summation of a list

Iterative Source Code


```
;   int f(struct A *x)
;   {   int r = 0;
;       for (; x!=0; x = x->tl) r += x->hd;
;       return r;
;   }
;
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

Fig. 1. Iterative summation of a list

Iterative Source Code

```
struct A { int hd; struct A *tl; };
int f(struct A *x)
{    int r = 0;
     for (; x!=0; x = x->tl) r += x->hd;
     return r;
}
```

```
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
        r += x->hd;
    return r;
}
```

```
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
        r += x->hd;
    return r;
}
```

```
for (; x!=0; x = x->t1)
```

```
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
        r += x->hd;
    return r;
}
```

```
for (; x!=0; x = x->t1)
```

```
while (x!=0) x = x->t1;
```

Convert to while loop

```
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
        r += x->hd;
    return r;
}
```

```
for (; x!=0; x = x->t1)
```

```
while (x!=0) x = x->t1;
```

Unroll once

```
if (x!=0)
  do x = x->t1; while (x!=0);
```

```c
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
        r += x->hd;
    return r;
}
```

```
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

```
r0 <- x
r1 <- r
```

```c
for (; x!=0; x = x->t1)
```

```c
while (x!=0) x = x->t1;
```

```c
if (x!=0)
  do x = x->t1; while (x!=0);
```

```c
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
       r += x->hd;
    return r;
}
```

```
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

```
r0 <- x
r1 <- r
```

```c
for (; x!=0; x = x->t1)
```

```c
while (x!=0) x = x->t1;
```

```c
if (x!=0)
  do x = x->t1; while (x!=0);
```

```c
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
        r += x->hd;
    return r;
}
```

```
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

```
r0 <- x
r1 <- r
0[r0] <- x->hd
```

```c
for (; x!=0; x = x->t1)
```

```c
while (x!=0) x = x->t1;
```

```c
if (x!=0)
  do x = x->t1; while (x!=0);
```

```c
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
        r += x->hd;
    return r;
}
```

```
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

```
r0 <- x
r1 <- r
0[r0] <- x->hd
4[r0] <- x->tl
```

```c
for (; x!=0; x = x->t1)
```

```c
while (x!=0) x = x->t1;
```

```c
if (x!=0)
  do x = x->t1; while (x!=0);
```

```
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
        r += x->hd;
    return r;
}
```

```
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

```
r0 <- x
r1 <- r
0[r0] <- x->hd
4[r0] <- x->tl
```

```
for (; x!=0; x = x->t1)
```

```
while (x!=0) x = x->t1;
```

```
if (x!=0)
  do x = x->t1; while (x!=0);
```

```
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
        r += x->hd;
    return r;
}
```

```
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

```
r0 <- x
r1 <- r
0[r0] <- x->hd
4[r0] <- x->tl
```

```
for (; x!=0; x = x->t1)
```

```
while (x!=0) x = x->t1;
```

```
if (x!=0)
  do x = x->t1; while (x!=0);
```

```
struct A { int hd; struct A *tl; };

int f(struct A *x)
{   int r = 0;
    for (;
         x!=0;
         x = x->tl)
        r += x->hd;
    return r;
}
```

```
f:
        mov     #0,r1
        cmp     #0,r0
        beq     L4F2
L3F2:
        ld.w    0[r0],r2
        add     r2,r1,r1
        ld.w    4[r0],r0
        cmp     #0,r0
        bne     L3F2
L4F2:
        mov     r1,r0
        ret
```

```
for (; x!=0; x = x->t1)
```

```
while (x!=0) x = x->t1;
```

```
if (x!=0)
  do x = x->t1; while (x!=0);
```

SSA

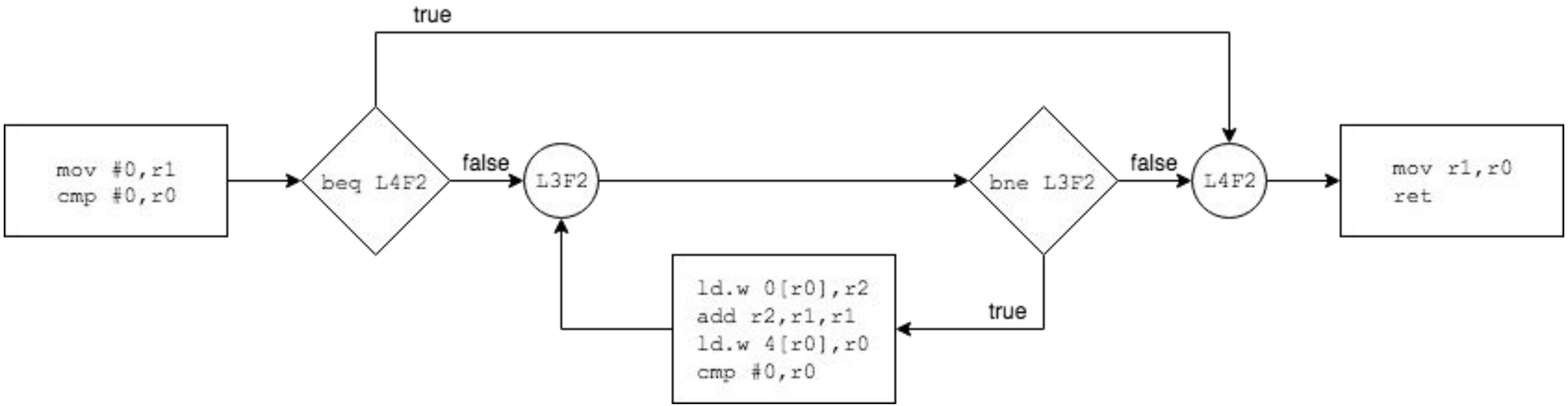
```
f:
      mov   #0,r1
      cmp   #0,r0
      beq   L4F2
L3F2:
      ld.w  0[r0],r2
      add   r2,r1,r1
      ld.w  4[r0],r0
      cmp   #0,r0
      bne   L3F2
L4F2:
      mov   r1,r0
      ret
```

We want to convert to SSA form.

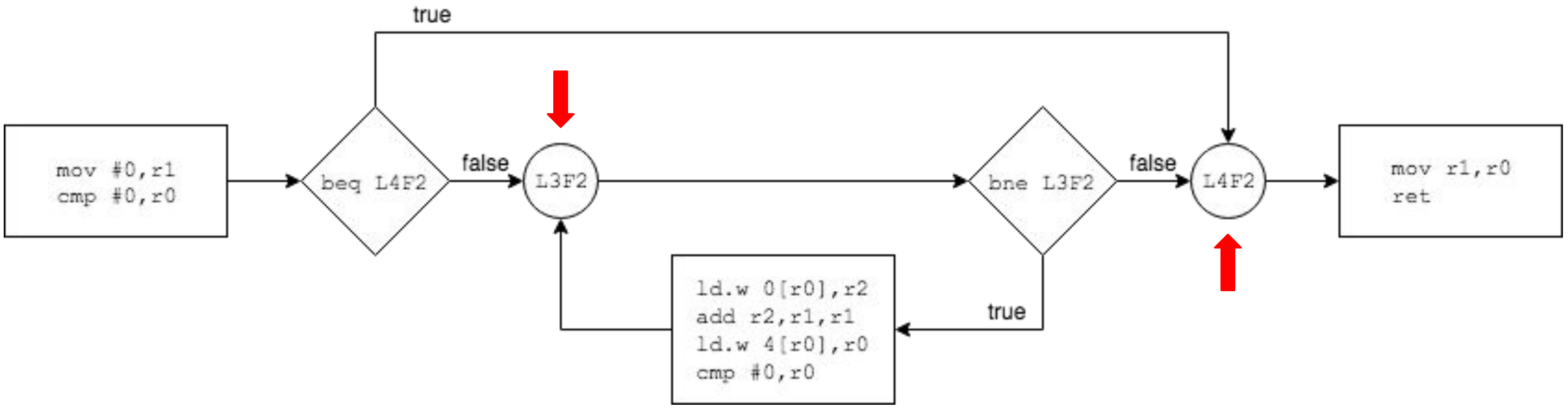
That was easy before… but now we have a loop. How do we convert to SSA when we have loops and branching?

The same register might end up with different values at a particular point based on the path it took to get there.

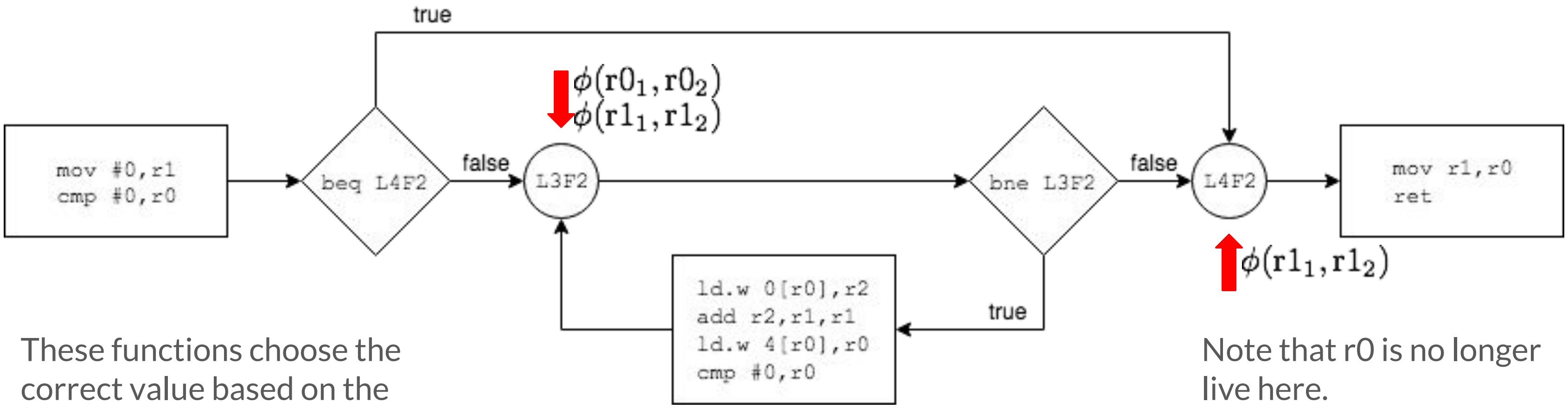
SSA: Flowgraph



SSA: Path Merge



SSA: Path Merge ϕ-Functions



These functions choose the correct value based on the arrival path.

When we arrive at, say, L3F2, we set r0 = $\phi(r0_1, r0_2)$, etc.

Note that r0 is no longer live here.

SSA: With ϕ-Functions

```
f:
      mov   #0,r1
      cmp   #0,r0
      beq   L4F2
L3F2:
      mov   φ(r0₁,r0₂),r0
      mov   φ(r1₁,r1₂),r1
      ld.w  0[r0],r2
      add   r2,r1,r1
      ld.w  4[r0],r0
      cmp   #0,r0
      bne   L3F2
L4F2:
      mov   φ(r1₁,r1₂),r1
      mov   r1,r0
      ret
```

Now that we have accounted for the branching, we can rewrite to SSA form.

SSA: Relabeling

```
f:
        mov   r0,r0a
        mov   #0,r1a
        cmp   #0,r0a
        beq   L4F2
L3F2:
        mov   ϕ(r0a,r0₂),r0
        mov   ϕ(r1a,r1₂),r1
        ld.w  0[r0],r2
        add   r2,r1,r1
        ld.w  4[r0],r0
        cmp   #0,r0
        bne   L3F2
L4F2:
        mov   ϕ(r1a,r1₂),r1
        mov   r1,r0
        ret
```

Subsequent references to r0 get relabeled to r0a.

Same for r1a.

SSA: Relabeling

```
f:
     mov   r0,r0a
     mov   #0,r1a
     cmp   #0,r0a
     beq   L4F2
L3F2:
     mov   φ(r0a,r0₂),r0b
     mov   φ(r1a,r1₂),r1b
     ld.w  0[r0b],r2
     add   r2,r1b,r1
     ld.w  4[r0b],r0
     cmp   #0,r0
     bne   L3F2
L4F2:
     mov   φ(r1a,r1₂),r1
     mov   r1,r0
     ret
```

SSA: Relabeling

```
f:
        mov   r0,r0a
        mov   #0,r1a
        cmp   #0,r0a
        beq   L4F2
L3F2:
        mov   φ(r0a,r0c),r0b
        mov   φ(r1a,r1c),r1b
        ld.w  0[r0b],r2a
        add   r2a,r1b,r1c
        ld.w  4[r0b],r0c
        cmp   #0,r0c
        bne   L3F2
L4F2:
        mov   φ(r1a,r1c),r1
        mov   r1,r0
        ret
```

SSA: Relabeling

```
f:
        mov   r0,r0a
        mov   #0,r1a
        cmp   #0,r0a
        beq   L4F2
L3F2:
        mov   φ(r0a,r0c),r0b
        mov   φ(r1a,r1c),r1b
        ld.w  0[r0b],r2a
        add   r2a,r1b,r1c
        ld.w  4[r0b],r0c
        cmp   #0,r0c
        bne   L3F2
L4F2:
        mov   φ(r1a,r1c),r1d
        mov   r1d,r0d
        ret
```

SSA: Relabeling Complete

```
f:
      mov   r0,r0a
      mov   #0,r1a
      cmp   #0,r0a
      beq   L4F2
L3F2:
      mov   ϕ(r0a,r0c),r0b
      mov   ϕ(r1a,r1c),r1b
      ld.w  0[r0b],r2a
      add   r2a,r1b,r1c
      ld.w  4[r0b],r0c
      cmp   #0,r0c
      bne   L3F2
L4F2:
      mov   ϕ(r1a,r1c),r1d
      mov   r1d,r0d
      ret
```

Now the program is in SSA form.

Type Constraints

```
f:
      mov   r0,r0a
      mov   #0,r1a
      cmp   #0,r0a
      beq   L4F2
L3F2:
      mov   ϕ(r0a,r0c),r0b
      mov   ϕ(r1a,r1c),r1b
      ld.w  0[r0b],r2a
      add   r2a,r1b,r1c
      ld.w  4[r0b],r0c
      cmp   #0,r0c
      bne   L3F2
L4F2:
      mov   ϕ(r1a,r1c),r1d
      mov   r1d,r0d
      ret
```

instruction		generated constraint
mov	r4,r6	$t6 = t4$
ld.w	n[r3],r5	$t3 = ptr(mem(n : t5))$
xor	r2a,r1b,r1c	$t2a = int, t1b = int, t1c = int$
add	r2a,r1b,r1c	$t2a = ptr(\alpha), t1b = int, t1c = ptr(\alpha) \vee$
		$t2a = int, t1b = ptr(\alpha'), t1c = ptr(\alpha') \vee$
		$t2a = int, t1b = int, t1c = int$
ld.w	(r5)[r0],r3	$t0 = ptr(array(t3)), t5 = int \vee$
		$t0 = int, t5 = ptr(array(t3))$
mov	#42,r7	$t7 = \mathrm{int}$
mov	#0,r7	$t7 = \mathrm{int} \vee t7 = ptr(\alpha'')$

Type Constraints

```
f:
        mov    r0,r0a          t0a = t0
        mov    #0,r1a
        cmp    #0,r0a
        beq    L4F2
L3F2:
        mov    φ(r0a,r0c),r0b
        mov    φ(r1a,r1c),r1b
        ld.w   0[r0b],r2a
        add    r2a,r1b,r1c
        ld.w   4[r0b],r0c
        cmp    #0,r0c
        bne    L3F2
L4F2:
        mov    φ(r1a,r1c),r1d
        mov    r1d,r0d
        ret
```

instruction		generated constraint
mov	r4,r6	$t6 = t4$
ld.w	n[r3],r5	$t3 = ptr(mem(n : t5))$
xor	r2a,r1b,r1c	$t2a = int, t1b = int, t1c = int$
add	r2a,r1b,r1c	$t2a = ptr(\alpha), t1b = int, t1c = ptr(\alpha) \vee$
		$t2a = int, t1b = ptr(\alpha'), t1c = ptr(\alpha') \vee$
		$t2a = int, t1b = int, t1c = int$
ld.w	(r5)[r0],r3	$t0 = ptr(array(t3)), t5 = int \vee$
		$t0 = int, t5 = ptr(array(t3))$
mov	#42,r7	$t7 = \mathtt{int}$
mov	#0,r7	$t7 = \mathtt{int} \vee t7 = ptr(\alpha'')$

Type Constraints

```
f:
      mov   r0,r0a            t0a = t0
      mov   #0,r1a
      cmp   #0,r0a
      beq   L4F2
L3F2:
      mov   ϕ(r0a,r0c),r0b
      mov   ϕ(r1a,r1c),r1b
      ld.w  0[r0b],r2a
      add   r2a,r1b,r1c
      ld.w  4[r0b],r0c
      cmp   #0,r0c
      bne   L3F2
L4F2:
      mov   ϕ(r1a,r1c),r1d
      mov   r1d,r0d
      ret
```

instruction		generated constraint
mov	r4,r6	$t6 = t4$
ld.w	n[r3],r5	$t3 = ptr(mem(n : t5))$
xor	r2a,r1b,r1c	$t2a = int, t1b = int, t1c = int$
add	r2a,r1b,r1c	$t2a = ptr(\alpha), t1b = int, t1c = ptr(\alpha) \vee$
		$t2a = int, t1b = ptr(\alpha'), t1c = ptr(\alpha') \vee$
		$t2a = int, t1b = int, t1c = int$
ld.w	(r5)[r0],r3	$t0 = ptr(array(t3)), t5 = int \vee$
		$t0 = int, t5 = ptr(array(t3))$
mov	#42,r7	$t7 = \mathrm{int}$
mov	#0,r7	$t7 = \mathrm{int} \vee t7 = ptr(\alpha'')$

Type Constraints

```
f:
      mov   r0,r0a
      mov   #0,r1a
      cmp   #0,r0a
      beq   L4F2
L3F2:
      mov   ϕ(r0a,r0c),r0b
      mov   ϕ(r1a,r1c),r1b
      ld.w  0[r0b],r2a
      add   r2a,r1b,r1c
      ld.w  4[r0b],r0c
      cmp   #0,r0c
      bne   L3F2
L4F2:
      mov   ϕ(r1a,r1c),r1d
      mov   r1d,r0d
      ret
```

$t0a = t0$

$t1a = \text{int} \vee t1a = ptr(\alpha_1)$

A new type variable that must be resolved later

instruction		generated constraint
mov	r4,r6	$t6 = t4$
ld.w	n[r3],r5	$t3 = ptr(mem(n : t5))$
xor	r2a,r1b,r1c	$t2a = int, t1b = int, t1c = int$
add	r2a,r1b,r1c	$t2a = ptr(\alpha), t1b = int, t1c = ptr(\alpha) \vee$
		$t2a = int, t1b = ptr(\alpha'), t1c = ptr(\alpha') \vee$
		$t2a = int, t1b = int, t1c = int$
ld.w	(r5)[r0],r3	$t0 = ptr(array(t3)), t5 = int \vee$
		$t0 = int, t5 = ptr(array(t3))$
mov	#42,r7	$t7 = \text{int}$
mov	#0,r7	$t7 = \text{int} \vee t7 = ptr(\alpha'')$

Type Constraints

```
f:
      mov   r0,r0a              t0a = t0
      mov   #0,r1a              t1a = int ∨ t1a = ptr(α1)
      cmp   #0,r0a              t0a = int ∨ t0a = ptr(α2)
      beq   L4F2
L3F2:
      mov   ϕ(r0a,r0c),r0b      t0b = t0a, t0b = t0c
      mov   ϕ(r1a,r1c),r1b      t1b = t1a, t1b = t1c
      ld.w  0[r0b],r2a          t0b = ptr(mem(0:t2a))
      add   r2a,r1b,r1c
      ld.w  4[r0b],r0c
      cmp   #0,r0c
      bne   L3F2
L4F2:
      mov   ϕ(r1a,r1c),r1d
      mov   r1d,r0d
      ret
```

```
ld.w      n[r3],r5      | t3 = ptr(mem(n : t5))
```

```
f:
      mov   r0,r0a               t0a = t0
      mov   #0,r1a               t1a = int ∨ t1a = ptr(α1)
      cmp   #0,r0a               t0a = int ∨ t0a = ptr(α2)
      beq   L4F2
L3F2:
      mov   ϕ(r0a,r0c),r0b       t0b = t0a, t0b = t0c
      mov   ϕ(r1a,r1c),r1b       t1b = t1a, t1b = t1c
      ld.w  0[r0b],r2a           t0b = ptr(mem(0:t2a)
      add   r2a,r1b,r1c          t2a = ptr(α3),t1b = int,t1c = ptr(α3)∨
                                 t2a = int,t1b = ptr(α4),t1c = ptr(α4)∨
                                 t2a = int,t1b = int,t1c = int
      ld.w  4[r0b],r0c           t0b = ptr(mem(4:t0c))
      cmp   #0,r0c               t0c = int ∨ t0c = ptr(α5)
      bne   L3F2
L4F2:
      mov   ϕ(r1a,r1c),r1d       t1d = t1a,t1d = t1c
      mov   r1d,r0d              t0d = t1d
      ret
```

Now we have a system of type constraints.

Still need to annotate the function itself.

```
f:                           tf = t0 → t99
    mov   r0,r0a             t0a = t0
    mov   #0,r1a             t1a = int ∨ t1a = ptr(α1)
    cmp   #0,r0a             t0a = int ∨ t0a = ptr(α2)
    beq   L4F2
L3F2:
    mov   φ(r0a,r0c),r0b     t0b = t0a, t0b = t0c
    mov   φ(r1a,r1c),r1b     t1b = t1a, t1b = t1c
    ld.w  0[r0b],r2a         t0b = ptr(mem(0:t2a)
    add   r2a,r1b,r1c        t2a = ptr(α3),t1b = int,t1c = ptr(α3)∨
                             t2a = int,t1b = ptr(α4),t1c = ptr(α4)∨
                             t2a = int,t1b = int,t1c = int
    ld.w  4[r0b],r0c         t0b = ptr(mem(4:t0c))
    cmp   #0,r0c             t0c = int ∨ t0c = ptr(α5)
    bne   L3F2
L4F2:
    mov   φ(r1a,r1c),r1d     t1d = t1a,t1d = t1c
    mov   r1d,r0d            t0d = t1d
    ret                      t99 = t0d
```

Now the type constraint annotation is complete.

1. $tf = t0 \rightarrow t99$
2. $t0a = t0$
3. $t1a = \mathtt{int} \lor t1a = ptr(\alpha_1)$
4. $t0a = \mathtt{int} \lor t0a = ptr(\alpha_2)$
5. $t0b = t0a,\ t0b = t0c$
6. $t1b = t1a,\ t1b = t1c$
7. $t0b = ptr(mem(0{:}t2a))$
8. $t2a = ptr(\alpha_3), t1b = \mathtt{int}, t1c = ptr(\alpha_3) \lor$
   $t2a = \mathtt{int}, t1b = ptr(\alpha_4), t1c = ptr(\alpha_4) \lor$
   $t2a = \mathtt{int}, t1b = \mathtt{int}, t1c = \mathtt{int}$
9. $t0b = ptr(mem(4{:}t0c))$
10. $t0c = \mathtt{int} \lor t0c = ptr(\alpha_5)$
11. $t1d = t1a, t1d = t1c$
12. $t0d = t1d$
13. $t99 = t0d$

We have a system of equations.
There may be many solutions,
one solution, or no solutions.

How can we have no solutions?

Solving

1. $tf = t0 \rightarrow t99$
2. $t0a = t0$
3. $t1a = \mathtt{int} \lor t1a = ptr(\alpha_1)$
4. $t0a = \mathtt{int} \lor t0a = ptr(\alpha_2)$
5. $t0b = t0a,\ t0b = t0c$
6. $t1b = t1a,\ t1b = t1c$
7. $t0b = ptr(mem(0{:}t2a)$
8. $t2a = ptr(\alpha_3), t1b = \mathtt{int}, t1c = ptr(\alpha_3) \lor$
   $t2a = \mathtt{int}, t1b = ptr(\alpha_4), t1c = ptr(\alpha_4) \lor$
   $t2a = \mathtt{int}, t1b = \mathtt{int}, t1c = \mathtt{int}$
9. $t0b = ptr(mem(4{:}t0c))$
10. $t0c = \mathtt{int} \lor t0c = ptr(\alpha_5)$
11. $t1d = t1a, t1d = t1c$
12. $t0d = t1d$
13. $t99 = t0d$

The author notes that line 5 gives $t0b = t0c$, and when combined with lines 7 and 9, we have the following.

$$
\begin{aligned}
t0c &= t0b \\
&= ptr(mem(0{:}t2a)) \\
&= ptr(mem(4{:}t0c))
\end{aligned}
$$

Solving

```
1.   tf = t0 → t99
2.   t0a = t0
3.   t1a = int ∨ t1a = ptr(α_1)
4.   t0a = int ∨ t0a = ptr(α_2)
5.   t0b = t0a, t0b = t0c
6.   t1b = t1a, t1b = t1c
7.   t0b = ptr(mem(0:t2a)
8.   t2a = ptr(α_3),t1b = int,t1c = ptr(α_3)∨
     t2a = int,t1b = ptr(α_4),t1c = ptr(α_4)∨
     t2a = int,t1b = int,t1c = int
9.   t0b = ptr(mem(4:t0c))
10.  t0c = int ∨ t0c = ptr(α_5)
11.  t1d = t1a,t1d = t1c
12.  t0d = t1d
13.  t99 = t0d
```

The author notes that line 5 gives `t0b = t0c`, and when combined with lines 7 and 9, we have the following.

$$\begin{aligned}\boxed{t0c} &= t0b \\ &= ptr(mem(0{:}t2a)) \\ &= ptr(mem(4{:}\boxed{t0c}))\end{aligned}$$

This fails **occurs check**. This is a rule that says that unification of a variable *V* and some structure *S* fails if *S* contains *V*.

This prevents creating infinite loops during type checking or unification.

Solving

1. $tf = t0 \rightarrow t99$
2. $t0a = t0$
3. $t1a = \mathtt{int} \lor t1a = ptr(\alpha_1)$
4. $t0a = \mathtt{int} \lor t0a = ptr(\alpha_2)$
5. $t0b = t0a,\ t0b = t0c$
6. $t1b = t1a,\ t1b = t1c$
7. $t0b = ptr(mem(0{:}t2a)$
8. $t2a = ptr(\alpha_3), t1b = \mathtt{int}, t1c = ptr(\alpha_3) \lor$
   $t2a = \mathtt{int}, t1b = ptr(\alpha_4), t1c = ptr(\alpha_4) \lor$
   $t2a = \mathtt{int}, t1b = \mathtt{int}, t1c = \mathtt{int}$
9. $t0b = ptr(mem(4{:}t0c))$
10. $t0c = \mathtt{int} \lor t0c = ptr(\alpha_5)$
11. $t1d = t1a, t1d = t1c$
12. $t0d = t1d$
13. $t99 = t0d$

$$
\begin{aligned}
t0c = t0b &\\
= ptr(mem(0{:}t2a)) &\\
= ptr(mem(4{:}t0c)) &
\end{aligned}
$$

Note that these have different offsets

This is solved by creating a structure to break the cycle.

```
struct G { t2a m0; t0c m4; }
```

This permits us to rewrite the prior expression.

$$
\begin{aligned}
t0c = t0b &= ptr(mem(0{:}t2a, 4{:}t0c))\\
&= ptr(\{t2a\ m0;\ t0c\ m4;\})\\
&= ptr(\mathtt{struct\ G})
\end{aligned}
$$

Solving

1. $tf = t0 \rightarrow t99$
2. $t0a = t0$
3. $t1a = \mathtt{int} \vee t1a = ptr(\alpha_1)$
4. $t0a = \mathtt{int} \vee t0a = ptr(\alpha_2)$
5. $t0b = t0a,\ t0b = t0c$
6. $t1b = t1a,\ t1b = t1c$
7. $t0b = ptr(mem(0{:}t2a)$
8. $t2a = ptr(\alpha_3), t1b = \mathtt{int}, t1c = ptr(\alpha_3) \vee$
   $t2a = \mathtt{int}, t1b = ptr(\alpha_4), t1c = ptr(\alpha_4) \vee$
   $t2a = \mathtt{int}, t1b = \mathtt{int}, t1c = \mathtt{int}$
9. $t0b = ptr(mem(4{:}t0c))$
10. $t0c = \mathtt{int} \vee t0c = ptr(\alpha_5)$
11. $t1d = t1a, t1d = t1c$
12. $t0d = t1d$
13. $t99 = t0d$

This new expression passes occurs check.

$t0c = t0b = ptr(\mathtt{struct\ G})$

We can continue to solve.

Solving

1. $tf = t0 \rightarrow t99$
2. $t0a = t0$
3. $t1a = \mathtt{int} \lor t1a = ptr(\alpha_1)$
4. $t0a = \mathtt{int} \lor t0a = ptr(\alpha_2)$
5. $t0b = t0a,\ t0b = t0c$
6. $t1b = t1a,\ t1b = t1c$
7. $t0b = ptr(mem(0{:}t2a)$
8. $t2a = ptr(\alpha_3), t1b = \mathtt{int}, t1c = ptr(\alpha_3) \lor$
   $t2a = \mathtt{int}, t1b = ptr(\alpha_4), t1c = ptr(\alpha_4) \lor$
   $t2a = \mathtt{int}, t1b = \mathtt{int}, t1c = \mathtt{int}$
9. $t0b = ptr(mem(4{:}t0c))$
10. $t0c = \mathtt{int} \lor t0c = ptr(\alpha_5)$
11. $t1d = t1a, t1d = t1c$
12. $t0d = t1d$
13. $t99 = t0d$

From 13, 12, and 11 we have the following.

$t99 = t0d = t1d = t1a$ **and** $t1c$

Solving

1. $tf = t0 \rightarrow t99$
2. $t0a = t0$
3. $t1a = \texttt{int} \vee t1a = ptr(\alpha_1)$
4. $t0a = \texttt{int} \vee t0a = ptr(\alpha_2)$
5. $t0b = t0a,\ t0b = t0c$
6. $t1b = t1a,\ t1b = t1c$
7. $t0b = ptr(mem(0{:}t2a))$
8. $t2a = ptr(\alpha_3), t1b = \texttt{int}, t1c = ptr(\alpha_3) \vee$
   $t2a = \texttt{int}, t1b = ptr(\alpha_4), t1c = ptr(\alpha_4) \vee$
   $t2a = \texttt{int}, t1b = \texttt{int}, t1c = \texttt{int}$
9. $t0b = ptr(mem(4{:}t0c))$
10. $t0c = \texttt{int} \vee t0c = ptr(\alpha_5)$
11. $t1d = t1a, t1d = t1c$
12. $t0d = t1d$
13. $t99 = t0d$

Continuing we use line 6.

$$
\begin{aligned}
t99 = t0d = t1d &= t1a \ \textbf{and}\ t1c \\
&= t1a \ \textbf{and}\ t1b \\
&= t1a \ \textbf{and}\ t1a \\
&= t1a = t1b = t1c
\end{aligned}
$$

From line 8 we know that, since $t1b$ and $t1c$ must be the same, only the second and third clauses can apply.

$$t1b = ptr(\alpha_4), t1c = ptr(\alpha_4) \vee$$
$$t1b = \texttt{int}, t1c = \texttt{int}$$

Solving

1. $tf = t0 \to t99$
2. $t0a = t0$
3. $t1a = \texttt{int} \lor t1a = ptr(\alpha_1)$
4. $t0a = \texttt{int} \lor t0a = ptr(\alpha_2)$
5. $t0b = t0a,\ t0b = t0c$
6. $t1b = t1a,\ t1b = t1c$
7. $t0b = ptr(mem(0{:}t2a))$
8. $t2a = ptr(\alpha_3), t1b = \texttt{int}, t1c = ptr(\alpha_3) \lor$
   $t2a = \texttt{int}, t1b = ptr(\alpha_4), t1c = ptr(\alpha_4) \lor$
   $t2a = \texttt{int}, t1b = \texttt{int}, t1c = \texttt{int}$
9. $t0b = ptr(mem(4{:}t0c))$
10. $t0c = \texttt{int} \lor t0c = ptr(\alpha_5)$
11. $t1d = t1a, t1d = t1c$
12. $t0d = t1d$
13. $t99 = t0d$

We conclude that $t99$ must be either `int` or $ptr(\alpha_5)$ (which is also $ptr(\alpha_1)$ due to line 3).

What can we deduce about $t0$?

$t0 = t0a = t0b$ (lines 2 and 5)
$\quad = ptr(\texttt{struct G})$ (prior result)

This gives us two possible types for the function.

$tf = t0 \to t99$
$\quad = ptr(\texttt{struct G}) \to (ptr(\alpha_4)$ **or** $\texttt{int})$

Rejecting the "parasitic" solution gives:

$tf = ptr(\texttt{struct G}) \to \texttt{int}$

Mycroft Intel X86-64 Version

Aside: Compiling

Different compilers can produce very different output. Some will "optimize away" the stack frame maintenance (if it is not needed) and some will eliminate intermediate results if they are not needed.

Let's try the program with `gcc` and `clang` and a few different optimization levels.

Compile

```
struct A { int hd; struct A *tl; };
int f(struct A *x)
{   int r = 0;
    for (; x!=0; x = x->tl) r += x->hd;
    return r;
}
```

```
$ gcc -c list.c
$ objdump -d -Mintel list.o
```

54 bytes

```
0000000000000000 <f>:
   0:  f3 0f 1e fa             endbr64
   4:  55                      push   rbp
   5:  48 89 e5                mov    rbp,rsp
   8:  48 89 7d e8             mov    QWORD PTR [rbp-0x18],rdi
   c:  c7 45 fc 00 00 00 00    mov    DWORD PTR [rbp-0x4],0x0
  13:  eb 15                   jmp    2a <f+0x2a>
  15:  48 8b 45 e8             mov    rax,QWORD PTR [rbp-0x18]
  19:  8b 00                   mov    eax,DWORD PTR [rax]
  1b:  01 45 fc                add    DWORD PTR [rbp-0x4],eax
  1e:  48 8b 45 e8             mov    rax,QWORD PTR [rbp-0x18]
  22:  48 8b 40 08             mov    rax,QWORD PTR [rax+0x8]
  26:  48 89 45 e8             mov    QWORD PTR [rbp-0x18],rax
  2a:  48 83 7d e8 00          cmp    QWORD PTR [rbp-0x18],0x0
  2f:  75 e4                   jne    15 <f+0x15>
  31:  8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  34:  5d                      pop    rbp
  35:  c3                      ret
```

This code is very direct. It's like the first version we produced in the register coloring example.

Compile

```c
struct A { int hd; struct A *tl; };
int f(struct A *x)
{   int r = 0;
    for (; x!=0; x = x->tl) r += x->hd;
    return r;
}
```

```
$ gcc -c -O2 list.c
$ objdump -d -Mintel list.o
```

33 bytes

```
0000000000000000 <f>:
   0:  f3 0f 1e fa             endbr64
   4:  31 c0                   xor    eax,eax
   6:  48 85 ff                test   rdi,rdi
   9:  74 15                   je     20 <f+0x20>
   b:  0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]
  10:  03 07                   add    eax,DWORD PTR [rdi]
  12:  48 8b 7f 08             mov    rdi,QWORD PTR [rdi+0x8]
  16:  48 85 ff                test   rdi,rdi
  19:  75 f5                   jne    10 <f+0x10>
  1b:  c3                      ret
  1c:  0f 1f 40 00             nop    DWORD PTR [rax+0x0]
  20:  c3                      ret
```

This code clearly has some optimizations. Note how the intermediate values are never written back, but live in the registers. Also note the use of nop instructions to align jump targets on 16-byte boundaries.

Compile

```
struct A { int hd; struct A *tl; };
int f(struct A *x)
{   int r = 0;
    for (; x!=0; x = x->tl) r += x->hd;
    return r;
}
```

```
$ gcc -c -Os list.c
$ objdump -d -Mintel list.o
```

```
20 bytes
```

```
0000000000000000 <f>:
   0:  f3 0f 1e fa              endbr64
   4:  31 c0                    xor    eax,eax
   6:  48 85 ff                 test   rdi,rdi
   9:  74 08                    je     13 <f+0x13>
   b:  03 07                    add    eax,DWORD PTR [rdi]
   d:  48 8b 7f 08              mov    rdi,QWORD PTR [rdi+0x8]
  11:  eb f3                    jmp    6 <f+0x6>
  13:  c3                       ret
```

More analysis is done and the compiler discovers three important things: alignment is not needed here, it can reuse the `test rdi, rdi` / `je 13` code, and only a single return is needed.

Compile

```
struct A { int hd; struct A *tl; };
int f(struct A *x)
{   int r = 0;
    for (; x!=0; x = x->tl) r += x->hd;
    return r;
}
```

```
$ clang -c -O list.c
$ objdump -d -Mintel list.o
```

```
28 bytes
```

```
0000000000000000 <f>:
   0:  31 c0                  xor    eax,eax
   2:  48 85 ff               test   rdi,rdi
   5:  74 14                  je     1b <f+0x1b>
   7:  66 0f 1f 84 00 00 00   nop    WORD PTR [rax+rax*1+0x0]
   e:  00 00
  10:  03 07                  add    eax,DWORD PTR [rdi]
  12:  48 8b 7f 08            mov    rdi,QWORD PTR [rdi+0x8]
  16:  48 85 ff               test   rdi,rdi
  19:  75 f5                  jne    10 <f+0x10>
  1b:  c3                     ret
```

Out of the gate the `clang` compiler seems to do better with the default optimization level. Note that it still aligns the top of the loop (`0x10`), but does not align all the jump targets (`0x1b`).

Compile

```
struct A { int hd; struct A *tl; };
int f(struct A *x)
{   int r = 0;
    for (; x!=0; x = x->tl) r += x->hd;
    return r;
}
```

```
$ clang -c -Oz list.c
$ objdump -d -Mintel list.o
```

```
16 bytes!
```

```
0000000000000000 <f>:
   0:  31 c0                  xor     eax,eax
   2:  48 85 ff               test    rdi,rdi
   5:  74 08                  je      f <f+0xf>
   7:  03 07                  add     eax,DWORD PTR [rdi]
   9:  48 8b 7f 08            mov     rdi,QWORD PTR [rdi+0x8]
   d:  eb f3                  jmp     2 <f+0x2>
   f:  c3                     ret
```

Here `clang` does better than the best `gcc` version. How is this possible? Well, the difference is the missing 4-byte `endbr64` instruction, but that will probably be *included* in future versions.

Mycroft on Intel

Let's go with the `gcc -O2` version and see if we can apply Mycroft's method.

```
0000000000000000 <f>:
   0:  f3 0f 1e fa             endbr64
   4:  31 c0                   xor    eax,eax
   6:  48 85 ff                test   rdi,rdi
   9:  74 15                   je     20 <f+0x20>
   b:  0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]
  10:  03 07                   add    eax,DWORD PTR [rdi]
  12:  48 8b 7f 08             mov    rdi,QWORD PTR [rdi+0x8]
  16:  48 85 ff                test   rdi,rdi
  19:  75 f5                   jne    10 <f+0x10>
  1b:  c3                      ret
  1c:  0f 1f 40 00             nop    DWORD PTR [rax+0x0]
  20:  c3                      ret
```

SSA

```
<f>:
   4:   xor     eax,eax
   6:   test    rdi,rdi
   9:   je      20 <f+0x20>
   b:   nop     DWORD PTR [rax+rax*1+0x0]
  10:   add     eax,DWORD PTR [rdi]
  12:   mov     rdi,QWORD PTR [rdi+0x8]
  16:   test    rdi,rdi
  19:   jne     10 <f+0x10>
  1b:   ret
  1c:   nop     DWORD PTR [rax+0x0]
  20:   ret
```

SSA

```
<f>:
   4:   xor    eax,eax
   6:   test   rdi,rdi
   9:   je     20 <f+0x20>
   b:   nop    DWORD PTR [rax+rax*1+0x0]
  10:   add    eax,DWORD PTR [rdi]
  12:   mov    rdi,QWORD PTR [rdi+0x8]
  16:   test   rdi,rdi
  19:   jne    10 <f+0x10>
  1b:   ret
  1c:   nop    DWORD PTR [rax+0x0]
  20:   ret
```

Let's get rid of extraneous addresses.
Let's also get rid of the no-ops.

SSA

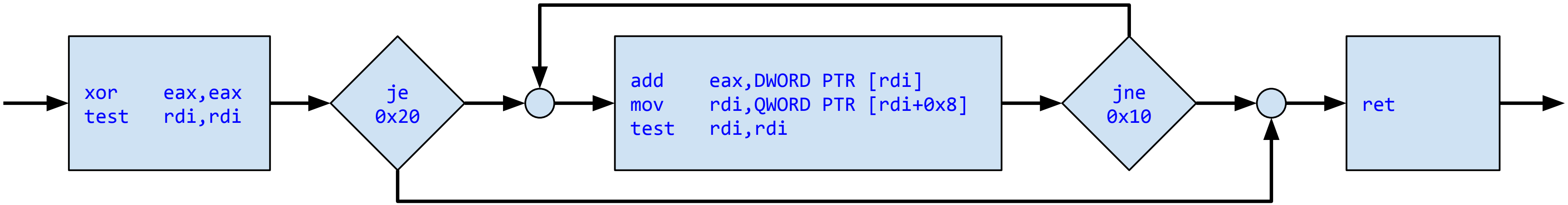
```
<f>:
        xor     eax,eax
        test    rdi,rdi
        je      20 <f+0x20>
  10:   add     eax,DWORD PTR [rdi]
        mov     rdi,QWORD PTR [rdi+0x8]
        test    rdi,rdi
        jne     10 <f+0x10>
  20:   ret
```

We don't need both returns.

Note that none of these changes will impact the type analysis.

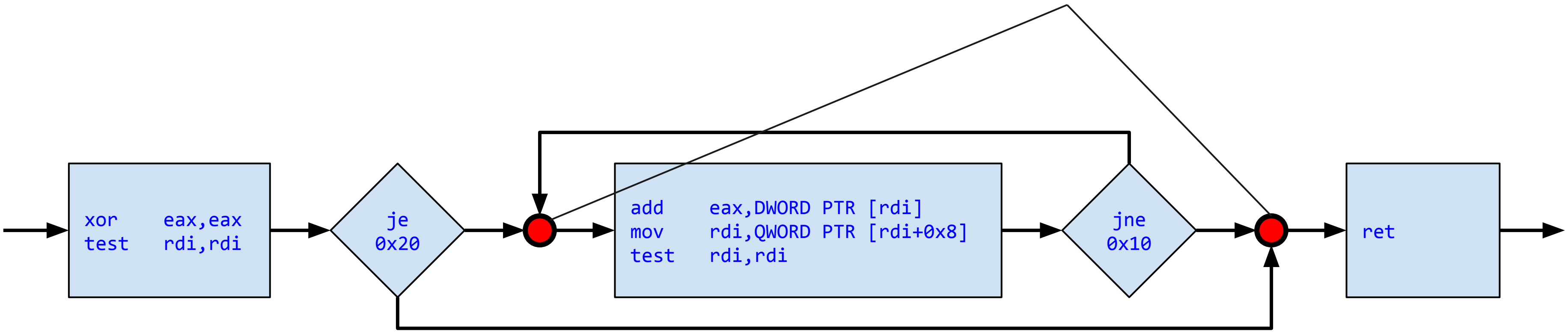
Now we need to consider the program flow.

SSA: Flowgraph



SSA: Flowgraph

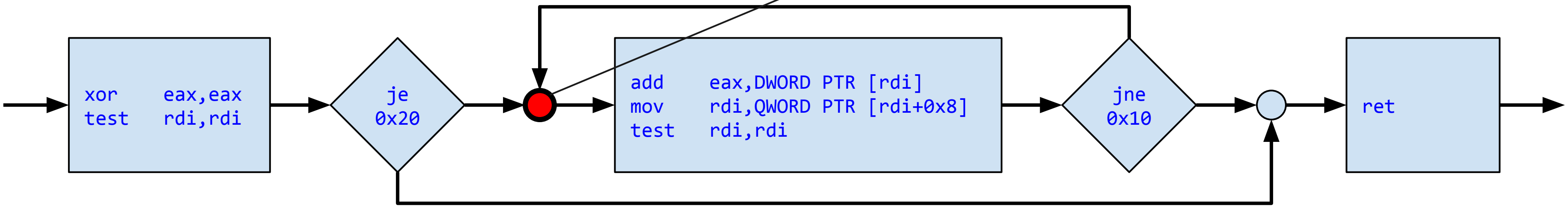
Flow join points
Create ϕ functions



SSA: Flowgraph

Both eax and rdi are live here

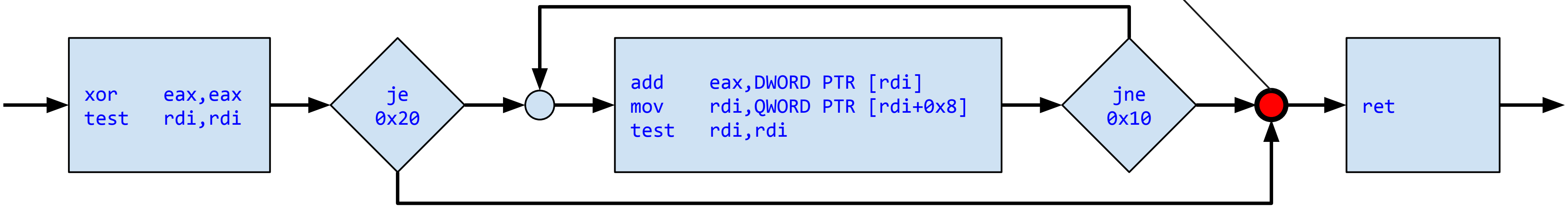
$\phi(\mathtt{eax}_0,\mathtt{eax}_1)$ and $\phi(\mathtt{rdi}_0,\mathtt{rdi}_1)$



SSA: Flowgraph

Only eax is live here, as the return value

$\phi(\mathtt{eax_0}, \mathtt{eax_1})$



SSA

```
<f>:
        xor     eax,eax
        test    rdi,rdi
        je      20 <f+0x20>
  10:   add     eax,DWORD PTR [rdi]
        mov     rdi,QWORD PTR [rdi+0x8]
        test    rdi,rdi
        jne     10 <f+0x10>
  20:   ret
```

We can add the ϕ functions

SSA

```
<f>:
       xor    eax,eax
       test   rdi,rdi
       je     20 <f+0x20>
  10:  mov    eax, ϕ(eax₀,eax₁)
       mov    rdi, ϕ(rdi₀,rdi₁)
       add    eax,DWORD PTR [rdi]
       mov    rdi,QWORD PTR [rdi+0x8]
       test   rdi,rdi
       jne    10 <f+0x10>
  20:  mov    eax, ϕ(eax₀,eax₁)
       ret
```

SSA

```
<f>:
        xor     eax,eax
        test    rdi,rdi
        je      20 <f+0x20>
  10:   mov     eax, ϕ(eax₀,eax₁)
        mov     rdi, ϕ(rdi₀,rdi₁)
        add     eax,DWORD PTR [rdi]
        mov     rdi,QWORD PTR [rdi+0x8]
        test    rdi,rdi
        jne     10 <f+0x10>
  20:   mov     eax, ϕ(eax₀,eax₁)
        ret
```

Now let's convert this to SSA

Let's write `eax1` and `rdi1` for the initial values. Doing this helps avoid a mistake where we forget to convert something. We will only be done when *all* instances of `eax` and `rdi` have a numeric suffix.

Again, this is equivalent to building a trace table.

SSA

```
<f>:
       xor    eax,eax1
       test   rdi1,rdi1
       je     20 <f+0x20>
  10:  mov    eax, ϕ(eax_0,eax_1)
       mov    rdi, ϕ(rdi1,rdi_1)
       add    eax,DWORD PTR [rdi]
       mov    rdi,QWORD PTR [rdi+0x8]
       test   rdi,rdi
       jne    10 <f+0x10>
  20:  mov    eax, ϕ(eax_0,eax_1)
       ret
```

Note test does not change the value.

We can replace rdi_0 with rdi1, which arrives on the indicated path.

SSA

```
<f>:
        xor     eax2,eax1
        test    rdi1,rdi1
        je      20 <f+0x20>
  10:   mov     eax, ϕ(eax2,eax₁)
        mov     rdi, ϕ(rdi1,rdi₁)
        add     eax,DWORD PTR [rdi]
        mov     rdi,QWORD PTR [rdi+0x8]
        test    rdi,rdi
        jne     10 <f+0x10>
  20:   mov     eax, ϕ(eax2,eax₁)
        ret
```

Continuing, the next value for eax is eax2.

We replace eax_0 with eax2 which arrives along the indicated paths.

SSA

```
<f>:
        xor     eax2,eax1
        test    rdi1,rdi1
        je      20 <f+0x20>
  10:   mov     eax3, ϕ(eax2,eax_1)
        mov     rdi2, ϕ(rdi1,rdi_1)
        add     eax4,DWORD PTR [rdi2]
        mov     rdi3,QWORD PTR [rdi2+0x8]
        test    rdi3,rdi3
        jne     10 <f+0x10>
  20:   mov     eax, ϕ(eax2,eax_1)
        ret
```

We continue...

SSA

```
<f>:
        xor    eax2,eax1
        test   rdi1,rdi1
        je     20 <f+0x20>
  10:   mov    eax3, ϕ(eax2,eax4)
        mov    rdi2, ϕ(rdi1,rdi3)
        add    eax4,DWORD PTR [rdi2]
        mov    rdi3,QWORD PTR [rdi2+0x8]
        test   rdi3,rdi3
        jne    10 <f+0x10>
  20:   mov    eax5, ϕ(eax2,eax4)
        ret
```

Now we need to consider the other arriving path at `0x20`. (And the backward branch to `0x10`.)

The only trick is to be sure you put the correct values in the ϕ functions. The order doesn't matter, but you need to show the correct values.

SSA

```
<f>:
        xor    eax2,eax1
        test   rdi1,rdi1
        je     20 <f+0x20>
  10:   mov    eax3, ϕ(eax2,eax4)
        mov    rdi2, ϕ(rdi1,rdi3)
        add    eax4,DWORD PTR [rdi2]
        mov    rdi3,QWORD PTR [rdi2+0x8]
        test   rdi3,rdi3
        jne    10 <f+0x10>
  20:   mov    eax5, ϕ(eax2,eax4)
        ret
```

The program returns the value in `eax5`. The argument to the program is in `rdi` (or `rdi1`).

The function signature is:
`f: T(rdi1) -> T(eax5)`

(I'll use `T(x)` for the type of `x`, unlike Mycroft.)

SSA

```
<f>:
        xor     eax2,eax1
        test    rdi1,rdi1
        je      20 <f+0x20>
  10:   mov     eax3, ϕ(eax2,eax4)
        mov     rdi2, ϕ(rdi1,rdi3)
        add     eax4,DWORD PTR [rdi2]
        mov     rdi3,QWORD PTR [rdi2+0x8]
        test    rdi3,rdi3
        jne     10 <f+0x10>
  20:   mov     eax5, ϕ(eax2,eax4)
        ret
```

Now let's apply the type constraints to the program.

Type Reconstruction

```
<f>:
        xor    eax2,eax1
        test   rdi1,rdi1
        je     20 <f+0x20>
  10:   mov    eax3, ϕ(eax2,eax4)
        mov    rdi2, ϕ(rdi1,rdi3)
        add    eax4,DWORD PTR [rdi2]
        mov    rdi3,QWORD PTR [rdi2+0x8]
        test   rdi3,rdi3
        jne    10 <f+0x10>
  20:   mov    eax5, ϕ(eax2,eax4)
        ret
```

`T(eax2) = T(eax1) = int?`

From Mycroft's constraints. Why not a pointer? Not enough bits.

Type Reconstruction

```
<f>:
        xor     eax2,eax1
        test    rdi1,rdi1
        je      20 <f+0x20>
  10:   mov     eax3, ϕ(eax2,eax4)
        mov     rdi2, ϕ(rdi1,rdi3)
        add     eax4,DWORD PTR [rdi2]
        mov     rdi3,QWORD PTR [rdi2+0x8]
        test    rdi3,rdi3
        jne     10 <f+0x10>
  20:   mov     eax5, ϕ(eax2,eax4)
        ret
```

`T(eax2) = T(eax1) = int32_t | uint32_t`

Since eax is 32 bits, we might use an explicit 32-bit type.

Type Reconstruction

```
<f>:
       xor    eax2,eax1
       test   rdi1,rdi1
       je     20 <f+0x20>
  10:  mov    eax3, ϕ(eax2,eax4)
       mov    rdi2, ϕ(rdi1,rdi3)
       add    eax4,DWORD PTR [rdi2]
       mov    rdi3,QWORD PTR [rdi2+0x8]
       test   rdi3,rdi3
       jne    10 <f+0x10>
  20:  mov    eax5, ϕ(eax2,eax4)
       ret
```

`T(eax2) = T(eax1) = int32_t | uint32_t`

These types come from `stdint.h`. You should always use these types. But signed or unsigned?

Type Reconstruction

```
<f>:
        xor    eax2,eax1
        test   rdi1,rdi1
        je     20 <f+0x20>
  10:   mov    eax3, ϕ(eax2,eax4)
        mov    rdi2, ϕ(rdi1,rdi3)
        add    eax4,DWORD PTR [rdi2]
        mov    rdi3,QWORD PTR [rdi2+0x8]
        test   rdi3,rdi3
        jne    10 <f+0x10>
  20:   mov    eax5, ϕ(eax2,eax4)
        ret
```

```
T(eax2) = T(eax1) = int32_t | uint32_t
T(rdi1) = int64_t | uint64_t | ptr(a)
```

It might be an integer, or it might be a pointer (because 64 bits is the right size). But a pointer to what? Call it *a* for now.

Type Reconstruction

```
<f>:
        xor     eax2,eax1
        test    rdi1,rdi1
        je      20 <f+0x20>
  10:   mov     eax3, ϕ(eax2,eax4)
        mov     rdi2, ϕ(rdi1,rdi3)
        add     eax4,DWORD PTR [rdi2]
        mov     rdi3,QWORD PTR [rdi2+0x8]
        test    rdi3,rdi3
        jne     10 <f+0x10>
  20:   mov     eax5, ϕ(eax2,eax4)
        ret
```

```
T(eax2) = T(eax1) = int32_t | uint32_t
T(rdi1) = int64_t | uint64_t | ptr(α)

T(eax3) = T(eax2) = T(eax4)
T(rdi2) = T(rdi1) = T(rdi3)
```

At this point we have an *invariant*. The types could (potentially) differ for `eax` and `rdi` during the loop, but must converge at the top of the loop body.

Type Reconstruction

```
<f>:
        xor     eax2,eax1                    T(eax2) = T(eax1) = int32_t | uint32_t
        test    rdi1,rdi1                    T(rdi1) = int64_t | uint64_t | ptr(α)
        je      20 <f+0x20>
  10:   mov     eax3, ϕ(eax2,eax4)           T(eax3) = T(eax2) = T(eax4)
        mov     rdi2, ϕ(rdi1,rdi3)           T(rdi2) = T(rdi1) = T(rdi3)
        add     eax4,DWORD PTR [rdi2]        T(rdi2) = ptr(T(eax4)@0)
        mov     rdi3,QWORD PTR [rdi2+0x8]    T(rdi2) = ptr(T(rdi3)@8)
        test    rdi3,rdi3
        jne     10 <f+0x10>
  20:   mov     eax5, ϕ(eax2,eax4)
        ret
```

Careful! All we really know is that `rdi2` "points to" something of type `T(eax4)` at offset 0, and something of type `T(rdi3)` at offset 8.

Type Reconstruction

```
<f>:
        xor    eax2,eax1                       T(eax2) = T(eax1) = int32_t | uint32_t
        test   rdi1,rdi1                       T(rdi1) = int64_t | uint64_t | ptr(a)
        je     20 <f+0x20>
  10:   mov    eax3, φ(eax2,eax4)              T(eax3) = T(eax2) = T(eax4)
        mov    rdi2, φ(rdi1,rdi3)              T(rdi2) = T(rdi1) = T(rdi3)
        add    eax4,DWORD PTR [rdi2]           T(rdi2) = ptr(T(eax4)@0)
        mov    rdi3,QWORD PTR [rdi2+0x8]       T(rdi2) = ptr(T(rdi3)@8)
        test   rdi3,rdi3                       T(rdi3) = int64_t | uint64_t | ptr(b)
        jne    10 <f+0x10>
  20:   mov    eax5, φ(eax2,eax4)
        ret
```

This could be a different type (not *a*), since `rdi` is changed. Better safe than sorry.

Type Reconstruction

```
<f>:
        xor     eax2,eax1                   T(eax2) = T(eax1) = int32_t | uint32_t
        test    rdi1,rdi1                   T(rdi1) = int64_t | uint64_t | ptr(a)
        je      20 <f+0x20>
  10:   mov     eax3, ϕ(eax2,eax4)          T(eax3) = T(eax2) = T(eax4)
        mov     rdi2, ϕ(rdi1,rdi3)          T(rdi2) = T(rdi1) = T(rdi3)
        add     eax4,DWORD PTR [rdi2]       T(rdi2) = ptr(T(eax4)@0)
        mov     rdi3,QWORD PTR [rdi2+0x8]   T(rdi2) = ptr(T(rdi3)@8)
        test    rdi3,rdi3                   T(rdi3) = int64_t | uint64_t | ptr(b)
        jne     10 <f+0x10>
  20:   mov     eax5, ϕ(eax2,eax4)          T(eax5) = T(eax2) = T(eax4)
        ret
```

Type Reconstruction

We still can't tell if eax should be signed or unsigned; let's just call it *signed*.

```
<f>:
        xor     eax2,eax1                      T(eax2) = T(eax1) = int32_t
        test    rdi1,rdi1                      T(rdi1) = int64_t | uint64_t | ptr(a)
        je      20 <f+0x20>
  10:   mov     eax3, φ(eax2,eax4)             T(eax3) = T(eax2) = T(eax4)
        mov     rdi2, φ(rdi1,rdi3)             T(rdi2) = T(rdi1) = T(rdi3)
        add     eax4,DWORD PTR [rdi2]          T(rdi2) = ptr(T(eax4)@0)
        mov     rdi3,QWORD PTR [rdi2+0x8]      T(rdi2) = ptr(T(rdi3)@8)
        test    rdi3,rdi3                      T(rdi3) = int64_t | uint64_t | ptr(b)
        jne     10 <f+0x10>
  20:   mov     eax5, φ(eax2,eax4)             T(eax5) = T(eax2) = T(eax4)
        ret
```

Type Reconstruction

Clearly `rdi2` is a pointer, and `T(rdi1) = T(rdi2) = T(rdi3)`, so we can resolve that.

```
<f>:
       xor    eax2,eax1                   T(eax2) = T(eax1) = int32_t
       test   rdi1,rdi1                   T(rdi1) = ptr(a)
       je     20 <f+0x20>
  10:  mov    eax3, ϕ(eax2,eax4)          T(eax3) = T(eax2) = T(eax4)
       mov    rdi2, ϕ(rdi1,rdi3)          T(rdi2) = T(rdi1) = T(rdi3)
       add    eax4,DWORD PTR [rdi2]       T(rdi2) = ptr(T(eax4)@0)
       mov    rdi3,QWORD PTR [rdi2+0x8]   T(rdi2) = ptr(T(rdi3)@8)
       test   rdi3,rdi3                   T(rdi3) = ptr(b)
       jne    10 <f+0x10>
  20:  mov    eax5, ϕ(eax2,eax4)          T(eax5) = T(eax2) = T(eax4)
       ret
```

Let's flow these updates through.

Type Reconstruction

```
<f>:
       xor    eax2,eax1                  T(eax2) = T(eax1) = int32_t
       test   rdi1,rdi1                  T(rdi1) = ptr(a)
       je     20 <f+0x20>
  10:  mov    eax3, φ(eax2,eax4)         T(eax3) = T(eax2) = T(eax4) = int32_t
       mov    rdi2, φ(rdi1,rdi3)         T(rdi2) = T(rdi1) = T(rdi3) = ptr(a)
       add    eax4,DWORD PTR [rdi2]      T(rdi2) = ptr(T(eax4)@0)
       mov    rdi3,QWORD PTR [rdi2+0x8]  T(rdi2) = ptr(T(rdi3)@8)
       test   rdi3,rdi3                  T(rdi3) = ptr(b)
       jne    10 <f+0x10>
  20:  mov    eax5, φ(eax2,eax4)         T(eax5) = T(eax2) = T(eax4) = int32_t
       ret
```

Type Reconstruction

Let's flow these updates through. We note that we end up with a `=` b.

```
<f>:
        xor    eax2,eax1                  T(eax2) = T(eax1) = int32_t
        test   rdi1,rdi1                  T(rdi1) = ptr(a)
        je     20 <f+0x20>
  10:   mov    eax3, φ(eax2,eax4)         T(eax3) = T(eax2) = T(eax4) = int32_t
        mov    rdi2, φ(rdi1,rdi3)         T(rdi2) = T(rdi1) = T(rdi3) = ptr(a)
        add    eax4,DWORD PTR [rdi2]      T(rdi2) = ptr(T(eax4)@0)
        mov    rdi3,QWORD PTR [rdi2+0x8]  T(rdi2) = ptr(T(rdi3)@8)
        test   rdi3,rdi3                  T(rdi3) = ptr(a)
        jne    10 <f+0x10>
  20:   mov    eax5, φ(eax2,eax4)         T(eax5) = T(eax2) = T(eax4) = int32_t
        ret
```

Type Reconstruction

We know `T(rdi3) = T(rdi2) = ptr(T(rdi3)@8)`, so we are going to *fail occurs check*. The solution is:

```
T(rdi2) = ptr({T(eax4)@0;T(rdi3)@8})
        = ptr({uint_32,ptr(α)})
        = ptr(struct X)

struct X { uint_32; struct X *; }
```

```
<f>:
      xor    eax2,eax1                    T(eax2) = T(eax1) = int32_t
      test   rdi1,rdi1                    T(rdi1) = ptr(α)
      je     20 <f+0x20>
 10:  mov    eax3, ϕ(eax2,eax4)           T(eax3) = T(eax2) = T(eax4) = int32_t
      mov    rdi2, ϕ(rdi1,rdi3)           T(rdi2) = T(rdi1) = T(rdi3) = ptr(α)
      add    eax4,DWORD PTR [rdi2]        T(rdi2) = ptr(uint_32 @0)
      mov    rdi3,QWORD PTR [rdi2+0x8]    T(rdi2) = ptr(ptr(α) @8)
      test   rdi3,rdi3                    T(rdi3) = ptr(α)
      jne    10 <f+0x10>
 20:  mov    eax5, ϕ(eax2,eax4)           T(eax5) = T(eax2) = T(eax4) = int32_t
      ret
```

Let's make it more C-like, now that we have a structure.

Type Reconstruction

```
<f>:
        xor     eax2,eax1                    T(eax2) = T(eax1) = int32_t
        test    rdi1,rdi1                    T(rdi1) = struct X *
        je      20 <f+0x20>
  10:   mov     eax3, φ(eax2,eax4)           T(eax3) = T(eax2) = T(eax4) = int32_t
        mov     rdi2, φ(rdi1,rdi3)           T(rdi2) = T(rdi1) = T(rdi3) = struct X *
        add     eax4,DWORD PTR [rdi2]        T(rdi2) = ptr(uint_32 @0)
        mov     rdi3,QWORD PTR [rdi2+0x8]    T(rdi2) = ptr(struct X * @8)
        test    rdi3,rdi3                    T(rdi3) = struct X *
        jne     10 <f+0x10>
  20:   mov     eax5, φ(eax2,eax4)           T(eax5) = T(eax2) = T(eax4) = int32_t
        ret
```

Type Reconstruction

Finally, let's figure out the function signature. The only argument is `rdi`, and we have `T(rdi1) = struct X *`. The return value is `rax`, and we have `T(rax) = int32_t`.

`f: T(rdi1) -> T(eax5)` is now:

`int32_t f(struct X *x)`

```
<f>:
        xor     eax2,eax1                    T(eax2) = T(eax1) = int32_t
        test    rdi1,rdi1                    T(rdi1) = struct X *
        je      20 <f+0x20>
  10:   mov     eax3, ϕ(eax2,eax4)           T(eax3) = T(eax2) = T(eax4) = int32_t
        mov     rdi2, ϕ(rdi1,rdi3)           T(rdi2) = T(rdi1) = T(rdi3) = struct X *
        add     eax4,DWORD PTR [rdi2]        T(rdi2) = ptr(uint_32 @0)
        mov     rdi3,QWORD PTR [rdi2+0x8]    T(rdi2) = ptr(struct X * @8)
        test    rdi3,rdi3                    T(rdi3) = struct X *
        jne     10 <f+0x10>
  20:   mov     eax5, ϕ(eax2,eax4)           T(eax5) = T(eax2) = T(eax4) = int32_t
        ret
```

Single Static Assignment (SSA)

Undoing Register Coloring

Register coloring is part of the process of mapping the resources needed for an algorithm to the resources available on an actual physical processor.

This is an essential process for compilation, but it can complicate analysis of a binary program.

Given an unknown program, we *don't know* what the original variables were, and so we don't know how to map registers back to variables. A very useful approximation is to just *assume every new value could be a new variable.* This is especially important when you don't have type information, so a register might hold an integer in one place, and a pointer to an integer in another.

Example

At right is a (part of) a function to compute the address of an object in a packed data structure. This function is part of a larger graphics rendering package. It might be called millions of times to perform a rendering.

Recall that the arguments are `rdi`, `rsi`, `rdx`, and the return value is `rax`.

```
0000000000000000 <compute_offset>:
   0:0f b7 c6           movzx   eax,si
   3:f7 ea              imul    edx
   5:48 8d 04 07        lea     rax,[rdi+rax*1]
   9:c3                 ret
```

Example

Let's try to analyze this without SSA. What is `rax`?

We end up with something like the following.

`T(rax) = ptr(a)`

But the first line shows that `T(rax)` is unlikely to be a pointer (only 16 bits).

It could be an offset and `rdi` could be an address, or the other way around. Which is right?

```
<compute_offset>:
movzx   eax,si              0:si -> eax
imul    edx                 eax * edx -> eax
lea     rax,[rdi+rax*1]     rdi+rax -> rax
ret
```

Example

Let's put it in SSA form.

To simplify life the registers are normalized and the width indicated with a slash.

Multiplication is expanded to show the registers involved: destination, source, source.

Let's analyze it now.

```
<compute_offset>:
movzx  rax1/d, rsi1/w
imul   rax2/d, rax1/d, rdx1/d
lea    rax3,[rdi1+rax2*1]
ret
```

Example

```
<compute_offset>:
movzx  rax1/d, rsi1/w               T(rax1) = uint16_t
imul   rax2/d, rax1/d, rdx1/d       T(rax2) = int32_t | uint32_t
lea    rax3,[rdi1+rax2*1]           T(rax3) = int64_t | uint64_t | ptr(a)
ret
```

On the first line `rax` is a 16-bit integer. On the second line the multiplication expands it to a 32-bit integer. On the third line it is potentially a pointer, depending on the type of `rdi`.

Example

```
<compute_offset>:
movzx  rax1/d, rsi1/w                T(rax1) = uint16_t
imul   rax2/d, rax1/d, rdx1/d        T(rax2) = int32_t | uint32_t
lea    rax3,[rdi1+rax2*1]            T(rax3) = int64_t | uint64_t | ptr(a)
ret
```

SSA exposes these different values and let's us talk about the fact that the type of `rax` on line 3 is likely different from the type of `rax` on line 1.

In fact, `rdi` holds a base pointer, `rsi` holds a 16-bit object width (up to 64KiB), and `rdx` holds the object number. This code computes the address of (a pointer to) the correct object in a packed array and returns it (via `rax`).

Last Homework:
Due April 30, 2020

Finalize your structuring code

Make sure you have done as much reduction as you can.

Output

Write out the address of a structure, followed by the structure.

```
0x15fef:
if
    0x0000000000015fef: sub edi, 1
    0x0000000000015ff2: jne 0x15fe0
then
    if
        0x0000000000015fe0: mov rax, rdx
        0x0000000000015fe3: mul rsi
        0x0000000000015fe6: jo 0x1620c
    then
        0x000000000001620c: mov r8d, 1
        0x0000000000016212: or rdx, 0xffffffffffffffff
        0x0000000000016216: jmp 0x15fef
        L := 0x15fef
    else
        0x0000000000015fec: mov rdx, rax
        L := 0x15fef
    fi
else
    0x0000000000015ff4: or r12d, r8d
    0x0000000000015ff7: jmp 0x160b0
    L := 0x160b0
fi
```

Output

Write out the address of a structure, followed by the structure.

If all branches end with the same label setting, factor it out.

This creates a new sequence.

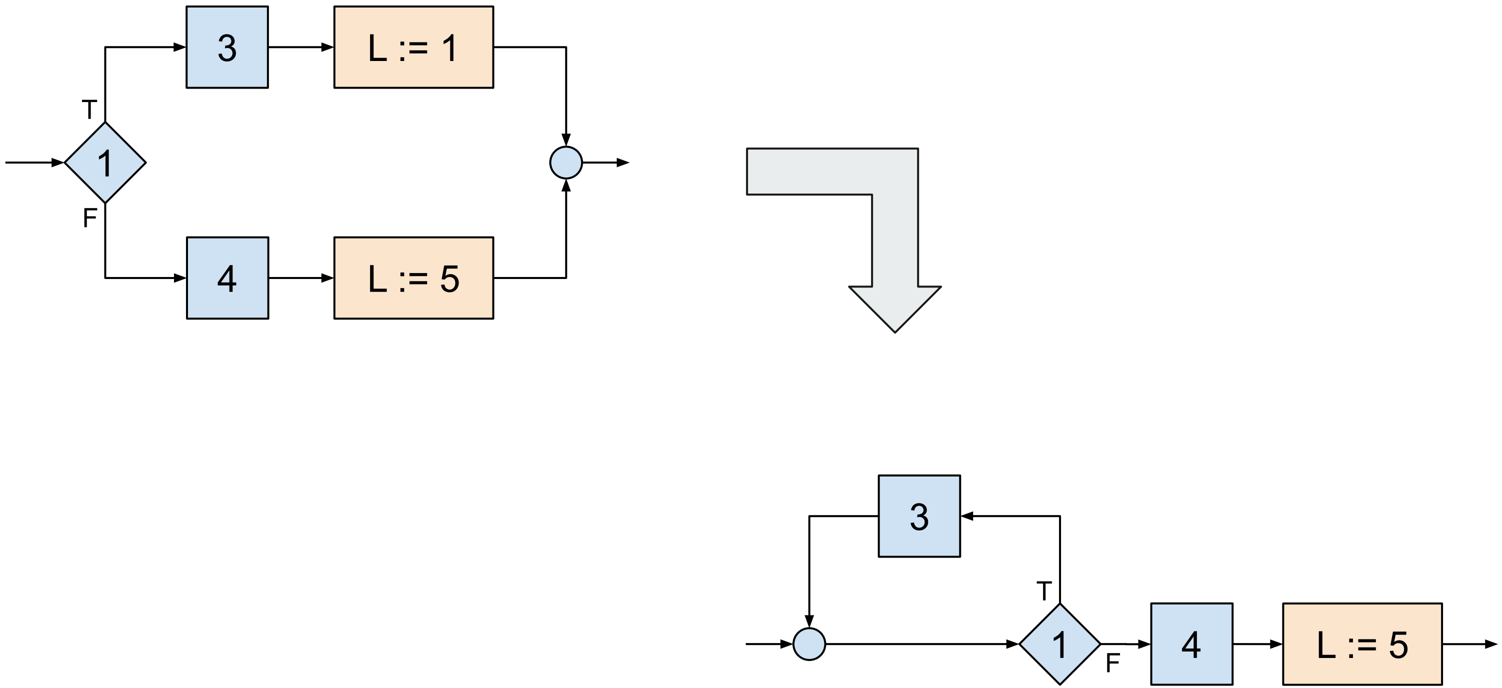
```
0x15fef:
if
    0x0000000000015fef: sub edi, 1
    0x0000000000015ff2: jne 0x15fe0
then
    if
        0x0000000000015fe0: mov rax, rdx
        0x0000000000015fe3: mul rsi
        0x0000000000015fe6: jo 0x1620c
    then
        0x000000000001620c: mov r8d, 1
        0x0000000000016212: or rdx, 0xffffffffffffffff
        0x0000000000016216: jmp 0x15fef
    else
        0x0000000000015fec: mov rdx, rax
    fi
    L := 0x15fef
else
    0x0000000000015ff4: or r12d, r8d
    0x0000000000015ff7: jmp 0x160b0
    L := 0x160b0
fi
```

Output

This is a simple self-loop.

If you find one of these it is easy to convert to a loop.

```
0x15fef:
if
    0x0000000000015fef: sub edi, 1
    0x0000000000015ff2: jne 0x15fe0
then
    if
        0x0000000000015fe0: mov rax, rdx
        0x0000000000015fe3: mul rsi
        0x0000000000015fe6: jo 0x1620c
    then
        0x000000000001620c: mov r8d, 1
        0x0000000000016212: or rdx, 0xffffffffffffffff
        0x0000000000016216: jmp 0x15fef
    else
        0x0000000000015fec: mov rdx, rax
    fi
    L := 0x15fef
else
    0x0000000000015ff4: or r12d, r8d
    0x0000000000015ff7: jmp 0x160b0
    L := 0x160b0
fi
```


3
L := 1
T
1
F
4
L := 5
3
T
1
F
4
L := 5

Output

This is a simple self-loop.

If you find one of these it is easy to convert to a loop.

(Don't need to do this.)

```
0x15fef:
while
    0x0000000000015fef: sub edi, 1
    0x0000000000015ff2: jne 0x15fe0
do
    if
        0x0000000000015fe0: mov rax, rdx
        0x0000000000015fe3: mul rsi
        0x0000000000015fe6: jo 0x1620c
    then
        0x000000000001620c: mov r8d, 1
        0x0000000000016212: or rdx, 0xffffffffffffffff
        0x0000000000016216: jmp 0x15fef
    else
        0x0000000000015fec: mov rdx, rax
    fi
end
0x0000000000015ff4: or r12d, r8d
0x0000000000015ff7: jmp 0x160b0
L := 0x160b0
```

Output

What you *do* need to do is, once you have done as much reduction as you can, write out the result as a graph.

Use GML (https://www.graphviz.org/) to create a digraph.

Nodes are the addresses of your structures, and the edges are the remaining references.

```
digraph "/usr/bin/ls" {
  "0x15fef" -> "0x15fef"
  "0x15fef" -> "0x160b0"
  ...
}
```

Next time:
Satisfiability Modulo Theories (SMT)