

# Máquinas Virtuales – 2

## Primeras Optimizaciones

12 de octubre de 2023

# Trampas usuales

Primero que nada... ¿qué hace esto?

```
let f (x:Nat) : Nat =  
  ifz x  
  then print "uno! " 1  
  else print "dos! " 2
```

# Trampas usuales

Primero que nada... ¿qué hace esto?

```
let f (x:Nat) : Nat =  
  ifz x  
  then print "uno! " 1  
  else print "dos! " 2
```

```
let x : Nat = print "uno! " (print "dos! " 3)
```

# Trampas usuales

Primero que nada... ¿qué hace esto?

```
let f (x:Nat) : Nat =  
  ifz x  
  then print "uno! " 1  
  else print "dos! " 2
```

```
let x : Nat = print "uno! " (print "dos! " 3)
```

```
let rec pr (x:Nat) : Nat = print "hola" (pr x)
```

# Trampas usuales

Primero que nada... ¿qué hace esto?

```
let f (x:Nat) : Nat =  
  ifz x  
  then print "uno! " 1  
  else print "dos! " 2
```

```
let x : Nat = print "uno! " (print "dos! " 3)
```

```
let rec pr (x:Nat) : Nat = print "hola" (pr x)
```

Asegúrense de que todos los backend son consistentes (evaluador naive, CEK, Macchina).

# La verdadera Macchina

- En el repo (carpeta `vm`) hay una versión de la Macchina implementada en C.

# La verdadera Macchina

- En el repo (carpeta `vm`) hay una versión de la Macchina implementada en C.
- Tienen que completarla con las instrucciones que hayan agregado para el ifz, y algunas partes más también.

# La verdadera Macchina

- En el repo (carpeta `vm`) hay una versión de la Macchina implementada en C.
- Tienen que completarla con las instrucciones que hayan agregado para el ifz, y algunas partes más también.
- La Macchina *usa un garbage collector* para las listas enlazadas de los entornos.



# Compilación del ifz

Dos nuevas instrucciones:

- JUMP  $n$ : salto *incondicional* relativo,  $\text{offset}=n$
- CJUMP  $n$ : salto *condicional* relativo,  $\text{offset}=n$ , mira el tope de la pila

# Compilación del ifz

Dos nuevas instrucciones:

- JUMP  $n$ : salto *incondicional* relativo, offset= $n$
- CJUMP  $n$ : salto *condicional* relativo, offset= $n$ , mira el tope de la pila

Compilación:


$$\mathcal{C}(\text{ifz } c \text{ then } g \text{ else } h) = \mathcal{C}(c); \text{ CJUMP}(o_1); \mathcal{C}(g); \text{ JUMP}(o_2); \mathcal{C}(h);$$

# Compilación del ifz

Dos nuevas instrucciones:

- JUMP  $n$ : salto *incondicional* relativo, offset= $n$
- CJUMP  $n$ : salto *condicional* relativo, offset= $n$ , mira el tope de la pila

Compilación:


$$\mathcal{C}(\text{ifz } c \text{ then } g \text{ else } h) = \mathcal{C}(c); \text{ CJUMP}(o_1); \mathcal{C}(g); \text{ JUMP}(o_2); \mathcal{C}(h);$$


# Compilación del ifz

Dos nuevas instrucciones:

- JUMP  $n$ : salto *incondicional* relativo,  $\text{offset}=n$
- CJUMP  $n$ : salto *condicional* relativo,  $\text{offset}=n$ , mira el tope de la pila

Compilación:

$$\mathcal{C}(\text{ifz } c \text{ then } g \text{ else } h) = \mathcal{C}(c); \text{CJUMP}(o_1); \mathcal{C}(g); \text{JUMP}(o_2); \mathcal{C}(h);$$


The diagram illustrates the control flow of the compiled code. It shows a sequence of instructions:  $\mathcal{C}(c)$ ,  $\text{CJUMP}(o_1)$ ,  $\mathcal{C}(g)$ ,  $\text{JUMP}(o_2)$ , and  $\mathcal{C}(h)$ . A horizontal line represents the flow of execution. A vertical line marks the end of  $\mathcal{C}(g)$ . From this point, a horizontal line extends to the right, then turns down and left to point at the  $\text{CJUMP}(o_1)$  instruction, representing the jump to the 'else' block. Another vertical line marks the end of  $\mathcal{C}(h)$ . From this point, a horizontal line extends to the left, then turns down and left to point at the  $\text{JUMP}(o_2)$  instruction, representing the jump back to the 'then' block.

# ¿Qué hace GCC?

```
int g(void);  
int h(void);  
  
int f(int c)  
{  
    if (c)  
        return g();  
    else  
        return h();  
}
```

## ¿Qué hace GCC?

```
int g(void);
int h(void);

int f(int c)
{
    if (c)
        return g();
    else
        return h();
}
```

```
f:
----- pushq   --%rbp
----- movq    --%rsp, %rbp
----- subq    --$16, %rsp
----- movl    --%edi, -4(%rbp)
----- cmpl    --$0, -4(%rbp)
----- je     --.L2
----- call   --g@PLT
----- jmp    --.L3
.L2:
----- call   --h@PLT
.L3:
----- leave
----- ret
```

# Macchina... eficiente?

Pensemos en la ejecución de esta función simple:

```
let g (y:Nat) : Nat = ...  
let f (x:Nat) : Nat = g(x+1)
```

# Macchina... eficiente?

Pensemos en la ejecución de esta función simple:

```
let g (y:Nat) : Nat = ...  
let f (x:Nat) : Nat = g(x+1)
```

La función *f* compila a algo como:

```
FUNCTION(g; ACCESS 0; CONST 1; ADD; CALL; RETURN)
```



# Una traza

Veamos una ejecución de una llamada  $f_{99}$

$\langle \text{ACCESS } g; \text{ ACCESS } 0; \text{ CONST } 1; \text{ ADD}; \text{ CALL}; \text{ RETURN} \mid 99 : e \mid RA : k \rangle$

# Una traza

Veamos una ejecución de una llamada  $f$  99

$\langle \text{ACCESS } g; \text{ ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$RA : k \rangle$
$\langle \text{ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$(e_g, c_g) : RA : k \rangle$

# Una traza

Veamos una ejecución de una llamada  $f$  99

$\langle \text{ACCESS } g; \text{ ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$RA : k \rangle$
$\langle \text{ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$(e_g, c_g) : RA : k \rangle$
$\langle \text{CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$99 : (e_g, c_g) : RA : k \rangle$

# Una traza

Veamos una ejecución de una llamada  $f$  99

$\langle \text{ACCESS } g; \text{ ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$RA : k \rangle$
$\quad \langle \text{ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$(e_g, c_g) : RA : k \rangle$
$\quad \quad \langle \text{CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$99 : (e_g, c_g) : RA : k \rangle$
$\quad \quad \quad \langle \text{ADD; CALL; RETURN} \mid$	$99 : e \mid$	$1 : 99 : (e_g, c_g) : RA : k \rangle$

# Una traza

Veamos una ejecución de una llamada f 99

$\langle \text{ACCESS } g; \text{ ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$RA : k \rangle$
$\langle \text{ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$(e_g, c_g) : RA : k \rangle$
$\langle \text{CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$99 : (e_g, c_g) : RA : k \rangle$
$\langle \text{ADD; CALL; RETURN} \mid$	$99 : e \mid$	$1 : 99 : (e_g, c_g) : RA : k \rangle$
$\langle \text{CALL; RETURN} \mid$	$99 : e \mid$	$100 : (e_g, c_g) : RA : k \rangle$

# Una traza

Veamos una ejecución de una llamada f 99

$\langle \text{ACCESS } g; \text{ ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$RA : k \rangle$
$\quad \langle \text{ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$(e_g, c_g) : RA : k \rangle$
$\quad \quad \langle \text{CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$99 : (e_g, c_g) : RA : k \rangle$
$\quad \quad \quad \langle \text{ADD; CALL; RETURN} \mid$	$99 : e \mid$	$1 : 99 : (e_g, c_g) : RA : k \rangle$
$\quad \quad \quad \quad \langle \text{CALL; RETURN} \mid$	$99 : e \mid$	$100 : (e_g, c_g) : RA : k \rangle$
$\quad \quad \quad \quad \quad \langle c_g \mid$	$100 : e_g \mid$	$(99 : e, p)_{RA} : RA : k \rangle$

# Una traza

Veamos una ejecución de una llamada f 99

$\langle \text{ACCESS } g; \text{ ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$RA : k \rangle$
$\langle \text{ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$(e_g, c_g) : RA : k \rangle$
$\langle \text{CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$99 : (e_g, c_g) : RA : k \rangle$
$\langle \text{ADD; CALL; RETURN} \mid$	$99 : e \mid$	$1 : 99 : (e_g, c_g) : RA : k \rangle$
$\langle \text{CALL; RETURN} \mid$	$99 : e \mid$	$100 : (e_g, c_g) : RA : k \rangle$
$\langle c_g \mid$	$100 : e_g \mid$	$(99 : e, p)_{RA} : RA : k \rangle$
$\langle \text{RETURN} \mid$	$? \mid$	$g_{100} : (99 : e, p)_{RA} : RA : k \rangle$

# Una traza

Veamos una ejecución de una llamada f 99

$\langle \text{ACCESS } g; \text{ ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$RA : k \rangle$
$\langle \text{ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$(e_g, c_g) : RA : k \rangle$
$\langle \text{CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$99 : (e_g, c_g) : RA : k \rangle$
$\langle \text{ADD; CALL; RETURN} \mid$	$99 : e \mid$	$1 : 99 : (e_g, c_g) : RA : k \rangle$
$\langle \text{CALL; RETURN} \mid$	$99 : e \mid$	$100 : (e_g, c_g) : RA : k \rangle$
$\langle c_g \mid$	$100 : e_g \mid$	$(99 : e, p)_{RA} : RA : k \rangle$
$\langle \text{RETURN} \mid$	$? \mid$	$g_{100} : (99 : e, p)_{RA} : RA : k \rangle$
$\langle \text{RETURN} \mid$	$99 : e \mid$	$g_{100} : RA : k \rangle$



# Una traza

Veamos una ejecución de una llamada f 99

$\langle \text{ACCESS } g; \text{ ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$RA : k \rangle$
$\quad \langle \text{ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$(e_g, c_g) : RA : k \rangle$
$\quad \quad \langle \text{CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$99 : (e_g, c_g) : RA : k \rangle$
$\quad \quad \quad \langle \text{ADD; CALL; RETURN} \mid$	$99 : e \mid$	$1 : 99 : (e_g, c_g) : RA : k \rangle$
$\quad \quad \quad \quad \langle \text{CALL; RETURN} \mid$	$99 : e \mid$	$100 : (e_g, c_g) : RA : k \rangle$
$\quad \quad \quad \quad \quad \langle c_g \mid$	$100 : e_g \mid$	$(99 : e, p)_{RA} : RA : k \rangle$
$\quad \quad \quad \quad \quad \quad \langle \text{RETURN} \mid$	$? \mid$	$g_{100} : (99 : e, p)_{RA} : RA : k \rangle$
$\quad \quad \quad \quad \quad \quad \quad \langle \text{RETURN} \mid$	$99 : e \mid$	$g_{100} : RA : k \rangle$
$\quad \quad \quad \quad \quad \quad \quad \quad \langle c \mid$	$e \mid$	$g_{100} : k \rangle$

# Una traza

Veamos una ejecución de una llamada f 99

$\langle \text{ACCESS } g; \text{ ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$RA : k \rangle$
$\langle \text{ACCESS } 0; \text{ CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$(e_g, c_g) : RA : k \rangle$
$\langle \text{CONST } 1; \text{ ADD; CALL; RETURN} \mid$	$99 : e \mid$	$99 : (e_g, c_g) : RA : k \rangle$
$\langle \text{ADD; CALL; RETURN} \mid$	$99 : e \mid$	$1 : 99 : (e_g, c_g) : RA : k \rangle$
$\langle \text{CALL; RETURN} \mid$	$99 : e \mid$	$100 : (e_g, c_g) : RA : k \rangle$
$\langle c_g \mid$	$100 : e_g \mid$	$(99 : e, p)_{RA} : RA : k \rangle$
$\langle \text{RETURN} \mid$	$? \mid$	$g_{100} : (99 : e, p)_{RA} : RA : k \rangle$
$\langle \text{RETURN} \mid$	$99 : e \mid$	$g_{100} : RA : k \rangle$
$\langle c \mid$	$e \mid$	$g_{100} : k \rangle$

¿Para qué está la pila?

# Rebotes

```
f:          g:          h:
...
t = g(x) ---> ...
          ...
          z = h(y) ---> ...
                      ...
                      <--- return
          ...
          w = h(z) ---> ...
                      ...
                      <--- return
          ...
          <--- return
...
...
return
```

# Rebotes

```
f:                g:                h:
...
t = g(x) ----> ...
...
z = h(y) ----> ...
...
               <--- return
...
w = h(z) ----> ...
...
               <--- return
...
<--- return
...
...
return
```

La pila guarda a dónde retornar el control (entre otras cosas).

# Pero... ¿hace falta?

```
f:           g:           h:
...
t = g(x) ----> ...
               ...
               z = h(y) ----> ...
                               ...
                               <--- return
               ...
               w = h(z) ----> ...
                               ...
                               <--- return
               <--- return
return
```

# Pero... ¿hace falta?

```
f:      g:      h:
...
t = g(x) ----> ...
              ...
              z = h(y) ----> ...
                              <--- return
                              ...
                              w = h(z) ----> ...
                                      <--- return
                                      <--- return
return
```

*Llamada de cola:* una llamada que es “lo último” que hace la función.

# Pero... ¿hace falta?

f:	g:	h:	
...			
t = g(x) --->	...		
	...		
	z = h(y) --->	...	
		...	
		<---	return
	...		
	w = h(z) --->	...	
		...	
		<---	return
	<---	return	
return			

*Llamada de cola:* una llamada que es “lo último” que hace la función.

<----- return

Pensemos en la siguiente implementación de fact:

```
let rec fact (n:Nat) : Nat =  
  ifz n  
  then 1  
  else n * fact (n-1)
```



Pensemos en la siguiente implementación de fact:

```
let rec fact (n:Nat) : Nat =  
  ifz n  
  then 1  
  else n * fact (n-1)
```

fact 5

Pensemos en la siguiente implementación de fact:

```
let rec fact (n:Nat) : Nat =  
  ifz n  
  then 1  
  else n * fact (n-1)
```

fact 5

5 \* fact 4

Pensemos en la siguiente implementación de fact:

```
let rec fact (n:Nat) : Nat =  
  ifz n  
  then 1  
  else n * fact (n-1)
```

fact 5

5 \* fact 4

5 \* 4 \* fact 3

## Recursión de cola

Pensemos en la siguiente implementación de fact:

```
let rec fact (n:Nat) : Nat =  
  ifz n  
  then 1  
  else n * fact (n-1)
```

fact 5

5 \* fact 4

5 \* 4 \* fact 3

20 \* fact 3

## Recursión de cola

Pensemos en la siguiente implementación de fact:

```
let rec fact (n:Nat) : Nat =  
  ifz n  
  then 1  
  else n * fact (n-1)
```

```
fact 5  
5 * fact 4  
5 * (4 * fact 3)  
20 * fact 3
```

No es recursiva de cola: luego de la llamada recursiva, hay que multiplicar.  
Usa  $O(n)$  espacio en la pila.

## *Recursión de cola, segundo intento*

```
let fact (n:Nat) : Nat =  
  let rec f (acc:Nat) (n:Nat) : Nat =  
    ifz n  
    then acc  
    else f (acc * n) (n-1)  
  in  
  f 1 n
```

## Recursión de cola, segundo intento

```
let fact (n:Nat) : Nat =  
  let rec f (acc:Nat) (n:Nat) : Nat =  
    ifz n  
    then acc  
    else f (acc * n) (n-1)  
  in  
  f 1 n
```

fact 5

## Recursión de cola, segundo intento

```
let fact (n:Nat) : Nat =  
  let rec f (acc:Nat) (n:Nat) : Nat =  
    ifz n  
    then acc  
    else f (acc * n) (n-1)  
  in  
  f 1 n
```

```
fact 5  
f 1 5
```



## Recursión de cola, segundo intento

```
let fact (n:Nat) : Nat =  
  let rec f (acc:Nat) (n:Nat) : Nat =  
    ifz n  
    then acc  
    else f (acc * n) (n-1)  
  in  
  f 1 n
```

```
fact 5  
f 1 5  
f 5 4
```

## Recursión de cola, segundo intento

```
let fact (n:Nat) : Nat =  
  let rec f (acc:Nat) (n:Nat) : Nat =  
    ifz n  
    then acc  
    else f (acc * n) (n-1)  
  in  
  f 1 n
```

```
fact 5  
f 1 5  
f 5 4  
f 20 3
```

## Recursión de cola, segundo intento

```
let fact (n:Nat) : Nat =  
  let rec f (acc:Nat) (n:Nat) : Nat =  
    ifz n  
    then acc  
    else f (acc * n) (n-1)  
  in  
  f 1 n
```

```
fact 5  
f 1 5  
f 5 4  
f 20 3  
f 60 2
```

## Recursión de cola, segundo intento

```
let fact (n:Nat) : Nat =  
  let rec f (acc:Nat) (n:Nat) : Nat =  
    ifz n  
    then acc  
    else f (acc * n) (n-1)  
  in  
  f 1 n
```

```
fact 5  
f 1 5  
f 5 4  
f 20 3  
f 60 2  
f 120 1
```

## Recursión de cola, segundo intento

```
let fact (n:Nat) : Nat =  
  let rec f (acc:Nat) (n:Nat) : Nat =  
    ifz n  
    then acc  
    else f (acc * n) (n-1)  
  in  
  f 1 n
```

```
fact 5  
f 1 5  
f 5 4  
f 20 3  
f 60 2  
f 120 1  
120
```

## Recursión de cola, segundo intento

```
let fact (n:Nat) : Nat =  
  let rec f (acc:Nat) (n:Nat) : Nat =  
    ifz n  
    then acc  
    else f (acc * n) (n-1)  
  in  
  f 1 n
```

```
fact 5  
f 1 5  
f 5 4  
f 20 3  
f 60 2  
f 120 1  
120
```

Sí es recursiva de cola  
Usa espacio  $O(1)$

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ ACCESS } 0; \text{ CALL}; \text{ RETURN}); \text{ CONST } 0; \text{ CALL}; k$$

$$\langle T \mid \epsilon \mid \epsilon \rangle$$

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ACCESS } 0; \text{CALL}; \text{RETURN}); \text{CONST } 0; \text{CALL}; k$$

$$\begin{array}{l} \langle T \mid \epsilon \mid \epsilon \rangle \\ \langle \text{CALL} \mid \epsilon \mid 0 : (f, e_{\text{fix}}) : \epsilon \rangle \end{array}$$



# En la Macchina

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ACCESS } 0; \text{CALL}; \text{RETURN}); \text{CONST } 0; \text{CALL}; k$$

$$\begin{array}{ll} \langle T \mid & \epsilon \mid \epsilon \rangle \\ \langle \text{CALL} \mid & \epsilon \mid 0 : (f, e_{\text{fix}}) : \epsilon \rangle \\ \langle \text{ACCESS } 1; \text{ACCESS } 0; \text{CALL}; \text{RETURN} \mid & 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \end{array}$$

# En la Macchina

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ ACCESS } 0; \text{ CALL}; \text{ RETURN}); \text{ CONST } 0; \text{ CALL}; \ k$$

$$\begin{array}{ll}
 \langle T \mid & \epsilon \mid \epsilon \rangle \\
 \langle \text{CALL} \mid & \epsilon \mid 0 : (f, e_{\text{fix}}) : \epsilon \rangle \\
 \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ CALL}; \text{ RETURN} \mid & 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \\
 \langle \text{CALL}; \text{ RETURN} \mid & 0 : e_{\text{fix}} \mid 0 : (f, e_{\text{fix}}) : (\epsilon, k)_{RA} : \epsilon \rangle
 \end{array}$$

# En la Macchina

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ACCESS } 0; \text{CALL}; \text{RETURN}); \text{CONST } 0; \text{CALL}; k$$

$$\begin{array}{ll}
 \langle T \mid & \epsilon \mid \epsilon \rangle \\
 \langle \text{CALL} \mid & \epsilon \mid 0 : (f, e_{\text{fix}}) : \epsilon \rangle \\
 \langle \text{ACCESS } 1; \text{ACCESS } 0; \text{CALL}; \text{RETURN} \mid & 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \\
 \langle \text{CALL}; \text{RETURN} \mid & 0 : e_{\text{fix}} \mid 0 : (f, e_{\text{fix}}) : (\epsilon, k)_{RA} : \epsilon \rangle \\
 \langle \text{ACCESS } 1; \text{ACCESS } 0; \text{CALL}; \text{RETURN} \mid & 0 : e_{\text{fix}} \mid (0 : e_{\text{fix}}, R)_{RA} : (\epsilon, k)_{RA} : \epsilon \rangle
 \end{array}$$

# En la Macchina

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ACCESS } 0; \text{CALL}; \text{RETURN}); \text{CONST } 0; \text{CALL}; k$$

$$\begin{array}{ll}
 \langle T \mid & \epsilon \mid \epsilon \rangle \\
 \langle \text{CALL} \mid & \epsilon \mid 0 : (f, e_{\text{fix}}) : \epsilon \rangle \\
 \langle \text{ACCESS } 1; \text{ACCESS } 0; \text{CALL}; \text{RETURN} \mid & 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \\
 \langle \text{CALL}; \text{RETURN} \mid & 0 : e_{\text{fix}} \mid 0 : (f, e_{\text{fix}}) : (\epsilon, k)_{RA} : \epsilon \rangle \\
 \langle \text{ACCESS } 1; \text{ACCESS } 0; \text{CALL}; \text{RETURN} \mid & 0 : e_{\text{fix}} \mid (0 : e_{\text{fix}}, R)_{RA} : (\epsilon, k)_{RA} : \epsilon \rangle \\
 \langle \text{ACCESS } 1; \text{ACCESS } 0; \text{CALL}; \text{RETURN} \mid & 0 : e_{\text{fix}} \mid (0 : e_{\text{fix}}, R)_{RA} : (0 : e_{\text{fix}}, R)_{RA} : (\epsilon, k)_{RA} : \epsilon \rangle
 \end{array}$$

...

## Idea: instrucción TAILCALL

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL}); \text{ CONST } 0; \text{ CALL}; k$$

$$\langle T \mid \epsilon \mid \epsilon \rangle$$

## Idea: instrucción TAILCALL

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL}); \text{ CONST } 0; \text{ CALL}; k$$

$$\begin{array}{l} \langle T \mid \quad \quad \epsilon \mid \quad \quad \quad \epsilon \rangle \\ \langle \text{CALL} \mid \quad \quad \epsilon \mid \quad \quad 0 : (f, e_{\text{fix}}) : \epsilon \rangle \end{array}$$

## Idea: instrucción TAILCALL

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL}); \text{ CONST } 0; \text{ CALL}; \ k$$

$$\begin{array}{lcl} \langle T \mid & \epsilon \mid & \epsilon \rangle \\ \langle \text{CALL} \mid & \epsilon \mid & 0 : (f, e_{\text{fix}}) : \epsilon \rangle \\ \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL} \mid & 0 : e_{\text{fix}} \mid & (\epsilon, k)_{RA} : \epsilon \rangle \end{array}$$

# Idea: instrucción TAILCALL

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL}); \text{ CONST } 0; \text{ CALL}; k$$

$$\begin{aligned} & \langle T \mid \epsilon \mid \epsilon \rangle \\ & \langle \text{CALL} \mid \epsilon \mid 0 : (f, e_{\text{fix}}) : \epsilon \rangle \\ & \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL} \mid 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \\ & \langle \text{TAILCALL} \mid 0 : e_{\text{fix}} \mid 0 : (f, e_{\text{fix}}) : (\epsilon, k)_{RA} : \epsilon \rangle \end{aligned}$$



# Idea: instrucción TAILCALL

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL}); \text{ CONST } 0; \text{ CALL}; \ k$$

$$\begin{array}{l} \langle T \mid \epsilon \mid \epsilon \rangle \\ \langle \text{CALL} \mid \epsilon \mid 0 : (f, e_{\text{fix}}) : \epsilon \rangle \\ \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL} \mid 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \\ \langle \text{TAILCALL} \mid 0 : e_{\text{fix}} \mid 0 : (f, e_{\text{fix}}) : (\epsilon, k)_{RA} : \epsilon \rangle \\ \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL} \mid 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \end{array}$$

# Idea: instrucción TAILCALL

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL}); \text{ CONST } 0; \text{ CALL}; k$$

$$\begin{array}{l} \langle T \mid \epsilon \mid \epsilon \rangle \\ \langle \text{CALL} \mid \epsilon \mid 0 : (f, e_{\text{fix}}) : \epsilon \rangle \\ \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL} \mid 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \\ \langle \text{TAILCALL} \mid 0 : e_{\text{fix}} \mid 0 : (f, e_{\text{fix}}) : (\epsilon, k)_{RA} : \epsilon \rangle \\ \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL} \mid 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \\ \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL} \mid 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \end{array}$$

# Idea: instrucción TAILCALL

$$\mathcal{C}(\text{let rec } f \ x = f \ x \text{ in } f \ 0) \approx$$

$$T = \text{FIXPOINT}(\text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL}); \text{ CONST } 0; \text{ CALL}; \ k$$

$$\begin{array}{l} \langle T \mid \epsilon \mid \epsilon \rangle \\ \langle \text{CALL} \mid \epsilon \mid 0 : (f, e_{\text{fix}}) : \epsilon \rangle \\ \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL} \mid 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \\ \langle \text{TAILCALL} \mid 0 : e_{\text{fix}} \mid 0 : (f, e_{\text{fix}}) : (\epsilon, k)_{RA} : \epsilon \rangle \\ \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL} \mid 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \\ \langle \text{ACCESS } 1; \text{ ACCESS } 0; \text{ TAILCALL} \mid 0 : e_{\text{fix}} \mid (\epsilon, k)_{RA} : \epsilon \rangle \end{array}$$

...

# Compilando llamadas de cola

¿Cómo detectar “lo último” que hace una función?

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{C}(t); \text{RETURN})$$

# Compilando llamadas de cola

¿Cómo detectar “lo último” que hace una función?

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{C}(t); \text{RETURN})$$

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{T}(t))$$

# Compilando llamadas de cola

¿Cómo detectar “lo último” que hace una función?

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{C}(t); \text{RETURN})$$

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{T}(t))$$

$$\mathcal{T}(f\ e) = \mathcal{C}(f); \mathcal{C}(e); \text{TAILCALL}$$

# Compilando llamadas de cola

¿Cómo detectar “lo último” que hace una función?

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{C}(t); \text{RETURN})$$

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{T}(t))$$

$$\begin{aligned}\mathcal{T}(f\ e) &= \mathcal{C}(f); \mathcal{C}(e); \text{TAILCALL} \\ \mathcal{T}(\text{ifz } c \text{ then } t \text{ else } f) &= \mathcal{C}(c); \dots; \mathcal{T}(t); \dots; \mathcal{T}(e)\end{aligned}$$

# Compilando llamadas de cola

¿Cómo detectar “lo último” que hace una función?

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{C}(t); \text{RETURN})$$

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{T}(t))$$

$$\begin{aligned}\mathcal{T}(f\ e) &= \mathcal{C}(f); \mathcal{C}(e); \text{TAILCALL} \\ \mathcal{T}(\text{ifz } c \text{ then } t \text{ else } f) &= \mathcal{C}(c); \dots; \mathcal{T}(t); \dots; \mathcal{T}(e) \\ \mathcal{T}(\text{let } x = M \text{ in } N) &= \mathcal{C}(M); \text{SHIFT}; \mathcal{T}(N);\end{aligned}$$



# Compilando llamadas de cola

¿Cómo detectar “lo último” que hace una función?

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{C}(t); \text{RETURN})$$

$$\mathcal{C}(\lambda t) = \text{FUNCTION}(\mathcal{T}(t))$$

$$\begin{aligned}\mathcal{T}(f\ e) &= \mathcal{C}(f); \mathcal{C}(e); \text{TAILCALL} \\ \mathcal{T}(\text{ifz } c \text{ then } t \text{ else } f) &= \mathcal{C}(c); \dots; \mathcal{T}(t); \dots; \mathcal{T}(e) \\ \mathcal{T}(\text{let } x = M \text{ in } N) &= \mathcal{C}(M); \text{SHIFT}; \mathcal{T}(N); \\ \mathcal{T}(X) &= \mathcal{C}(X); \text{RETURN}\end{aligned}$$

## ¿Qué hace GCC? - “Event loop”

```
int recv(void);  
int handle(int);  
  
int srv() {  
    int x = recv();  
    handle(x);  
    srv();  
}  
  
// gcc -O2
```

## ¿Qué hace GCC? - “Event loop”

```
int recv(void);  
int handle(int);
```

```
int srv() {  
    int x = recv();  
    handle(x);  
    srv();  
}  
// gcc -O2
```

```
int recv(void);  
int handle(int);
```

```
int srv() {  
L:  
    int x = recv();  
    handle(x);  
    goto L;  
}
```

## ¿Qué hace GCC? - Factorial

```
int fact(int n)
{
    if (n == 0)
        return 1;
    return n * fact(n-1);
}
```

```
//gcc -c -O3 -S fact.c
```

## ¿Qué hace GCC? - Factorial

```
int fact(int n)
{
    if (n == 0)
        return 1;
    return n * fact(n-1);
}
```

```
//gcc -c -O3 -S fact.c
```

```
fact:
-----movl    $1, %eax
-----testl   %edi, %edi
-----je      .L1
.L2:
-----imull   %edi, %eax
-----subl    $1, %edi
-----jne     .L2
.L1:
-----ret
```

## ¿Qué hace GCC? - Factorial

```
int fact(int n)
{
    if (n == 0)
        return 1;
    return n * fact(n-1);
}
```

```
//gcc -c -O3 -S fact.c
```

```
fact:
-----movl    $1, %eax
-----testl   %edi, %edi
-----je      .L1
.L2:
-----imull   %edi, %eax
-----subl    $1, %edi
-----jne     .L2
.L1:
-----ret
```

Requiere asociatividad de \* para enteros.

## ¿Qué hace GCC? - Josephus

```
int j(int n)
{
    if (n == 1)
        return 1;
    if (n&1)
        return 2*j(n/2) + 1;
    else
        return 2*j(n/2) - 1;
}
```

```
//gcc -c -O3 -S josephus.c
```

## ¿Qué hace GCC? - Josephus

```
int j(int n)
{
    if (n == 1)
        return 1;
    if (n&1)
        return 2*j(n/2) + 1;
    else
        return 2*j(n/2) - 1;
}
```

```
//gcc -c -O3 -S josephus.c
```

```
int j(int n) {
    /* Inv:  $s(n_0) = m*s(n) + b$  */
    int m=1, b=0;
    while (n != 1) {
        if (n&1) { b += m; m *= 2; }
        else     { b -= m; m *= 2; }
        n = n/2;
    }
    return m*1 + b;
    /* por el invariante, esto es  $s(n_0)$  */
}
```



# ¿Qué hace GCC? - Llamadas externas

```
extern int next(int);  
extern int f(int);  
  
int g(int n) {  
    return f(next(n));  
}
```

## ¿Qué hace GCC? - Llamadas externas

```
extern int next(int);  
extern int f(int);  
  
int g(int n) {  
    return f(next(n));  
}
```

```
g:  
- - - - - subq - - - - $8, %rsp  
- - - - - call - - - - next@PLT  
- - - - - addq - - - - $8, %rsp  
- - - - - movl - - - - %eax, %edi  
- - - - - jmp  - - - - f@PLT
```

## ¿Qué hace GCC? - Llamadas mutuas

```
extern int next(int);  
int f(int);  
__attribute__((noinline))  
int g(int n) {  
    return f(next(n));  
}  
__attribute__((noinline))  
int f(int n) {  
    return g(next(n));  
}
```

## ¿Qué hace GCC? - Llamadas mutuas

```
extern int next(int);
int f(int);
__attribute__((noinline))
int g(int n) {
    return f(next(n));
}
__attribute__((noinline))
int f(int n) {
    return g(next(n));
}
```

```
f:
-----subq-----$8, %rsp
-----call-----next@PLT
-----addq-----$8, %rsp
-----movl-----%eax, %edi
-----jmp-----g

g:
-----subq-----$8, %rsp
-----call-----next@PLT
-----addq-----$8, %rsp
-----movl-----%eax, %edi
-----jmp-----f
```

# ¿Qué hace GCC? - Llamadas mutuas

```
extern int next(int);
int f(int);
__attribute__((noinline))
int g(int n) {
    return f(next(n));
}
__attribute__((noinline))
int f(int n) {
    return g(next(n));
}
```

```
f:
-----subq-----$8, %rsp
-----call-----next@PLT
-----addq-----$8, %rsp
-----movl-----%eax, %edi
-----jmp-----g

g:
-----subq-----$8, %rsp
-----call-----next@PLT
-----addq-----$8, %rsp
-----movl-----%eax, %edi
-----jmp-----f
```

Ni un call a la vista.

# ¿Qué hace GCC? - Llamadas mutuas

```
extern int next(int);
int f(int);
__attribute__((noinline))
int g(int n) {
    return f(next(n));
}
__attribute__((noinline))
int f(int n) {
    return g(next(n));
}
```

```
f:
-----subq----$8, %rsp
-----call----next@PLT
-----addq----$8, %rsp
-----movl----%eax, %edi
-----jmp-----g

g:
-----subq----$8, %rsp
-----call----next@PLT
-----addq----$8, %rsp
-----movl----%eax, %edi
-----jmp-----f
```

Ni un call a la vista.  
Tarea: probar sin  
noinline y con  
static.

## Cerrando

- Las funciones recursivas de cola son *muy* comunes en programación funcional: hay que saber detectar cuándo una función se puede escribir de esa forma para minimizar el uso de memoria.

# Cerrando

- Las funciones recursivas de cola son *muy* comunes en programación funcional: hay que saber detectar cuándo una función se puede escribir de esa forma para minimizar el uso de memoria. (¿Escribieron alguna recientemente?)



# Cerrando

- Las funciones recursivas de cola son *muy* comunes en programación funcional: hay que saber detectar cuándo una función se puede escribir de esa forma para minimizar el uso de memoria. (¿Escribieron alguna recientemente?)
- Son análogas a un bucle en un lenguaje imperativo.

# Cerrando

- Las funciones recursivas de cola son *muy* comunes en programación funcional: hay que saber detectar cuándo una función se puede escribir de esa forma para minimizar el uso de memoria. (¿Escribieron alguna recientemente?)
- Son análogas a un bucle en un lenguaje imperativo.
- Un buen compilador puede (a veces) transformar una función para hacerla recursiva de cola.

# Cerrando

- Las funciones recursivas de cola son *muy* comunes en programación funcional: hay que saber detectar cuándo una función se puede escribir de esa forma para minimizar el uso de memoria. (¿Escribieron alguna recientemente?)
- Son análogas a un bucle en un lenguaje imperativo.
- Un buen compilador puede (a veces) transformar una función para hacerla recursiva de cola.
- Una moraleja de alto nivel: “resolver” la recursión de cola suena difícil (incluso detectarla de manera precisa). Resolver las *llamadas* de cola es algo más general y *más fácil*.

# Cerrando

- Las funciones recursivas de cola son *muy* comunes en programación funcional: hay que saber detectar cuándo una función se puede escribir de esa forma para minimizar el uso de memoria. (¿Escribieron alguna recientemente?)
- Son análogas a un bucle en un lenguaje imperativo.
- Un buen compilador puede (a veces) transformar una función para hacerla recursiva de cola.
- Una moraleja de alto nivel: “resolver” la recursión de cola suena difícil (incluso detectarla de manera precisa). Resolver las *llamadas* de cola es algo más general y *más fácil*.
- Buscar foldr y foldl: ¿cuál puede hacerse recursiva de cola?
- Técnicas avanzadas: *continuation passing style*. Toda llamada es de cola. Ver “Continuaciones: La Venganza del GOTO” de Guido Macchi.