Informe entrega 3

Luciano Scola y María Agustina Torrano

2024

Compiladores

1 Ejercicio 6

Veamos que la nueva compilación nunca genera un código más grande que la versión del apunte anterior separando los términos que sufrieron modificaciones en casos.

- **Lam.**

  En la versión anterior teníamos :

```
bcc (Lam i n ty (Sc1 t)) b = do
  bco <- bcc t
  let l = length bco
  return ((FUNCTION:l + 1:bco) ++ (RETURN:b))
```

  y con la nueva implementación tenemos ahora el siguiente código definido para `bcc`

```
bcc (Lam i n ty (Sc1 t)) b = do
  bco <- bct t []
  let l = length bco
  return ((FUNCTION:l:bco) ++ b)
```

  Observemos que, a diferencia del caso anterior, `bcc` no agrega la instrucción `RETURN` en el bytecode. Sin embargo, el llamado a `bct` agrega al final del bytecode un `RETURN` o un `TAILCALL`. De cualquier forma, en el peor de los casos, el código de la compilación mantendrá su longitud.

- **App.** En la versión anterior teníamos :

```
bcc (App i t t') b = do
  be <- bcc t' (CALL:b)
  bcc t be
```

  y con la nueva implementación tenemos dos opciones, que se llame con la función `bcc` o que se llame con la función `bct`, la cual está definida para la aplicación de la siguiente manera:

```
bct (App i t t') b = do
  be <- bcc t' (TAILCALL:b)
  bcc t be
```

  Es claro que en la nueva compilación de una aplicación, la cantidad de bytecode no aumenta, ya que en el caso en que utilice `bct`, simplemente se cambia `CALL` por `TAILCALL`, manteniendo el tamaño del mismo.

- **Fix.** En la versión anterior teníamos:

```
bcc (Fix i f fty x ty (Sc2 t)) b = do
  bco <- bcc t
  let l = length bco
  return ((FUNCTION:l+1:bco) ++ (RETURN:FIX:b))
```

  En la nueva versión tenemos:

```
bcc (Fix i f fty x ty (Sc2 t)) b = do
  bco <- bct t []
  let l = length bco
  return ((FUNCTION:l:bco) ++ (FIX:b))
```

  Observemos que la demostración es análoga al caso de `Fun`.

- **IfZ.** En la versión anterior teníamos:

```
bcc (IfZ i t1 t2 t3) b = do
  b'' <- bc t2
  b''' <- bc t3
  let (l1,l2) = (length b'',length b''')
  bcc t1 ((IFZ:l1+2:b'') ++ (JUMP:l2:b''') ++ b)
```

  En esta versión un término `IfZ` puede compilarse de dos formas distintas dependiendo si está en posición de cola (`bct`) o no (`bcc`). El segundo caso va a hacer lo mismo que en la versión anterior, por lo tanto, generará el mismo código. Veamos que sucede en el primer caso:

```
bct (IfZ i t1 t2 t3) b = do
  b'' <- bct t2 []
  b''' <- bct t3 []
  let l1 = length b''
  bcc t1 ((IFZ:l1:b'') ++ b''' ++ b)
```

  Observemos que el `JUMP` no será necesario porque, en el caso de que la condición sea 0, se ejecutará el código de `b''` y este retornará antes de llegar al código de `b'''`. Es decir, el código generado por la nueva compilación tendrá, al menos, dos elementos menos que la original (`JUMP` y `l2`).

- **Let.** En la versión anterior teníamos:

```
bcc (Let i x ty t (Sc1 t')) b = do
  be <- bcc t' (DROP:b)
  bcc t (SHIFT:be)
```

En la nueva versión del `Let` verificamos que el nuevo valor definido sea utilizado en el resto de la función. Esto lo realiza la función `localScope :: TTerm -> Int -> Bool`. Si no lo encuentra, se utiliza la función `indexShift :: TTerm -> Int -> TTerm`, la cuál resta en 1 las variables ligadas del cuerpo del `Let` para no tener en cuenta este nuevo valor. Se debe tener en cuanta que si la variable no es utilizada, no es necesario agregar el nuevo valor al entorno, es decir, podemos descartar las instrucciones `SHIFT` y `DROP`. Sin embargo, será necesario sacar dicho valor de la pila, por lo tanto, se introduce una nueva instrucción `POP`. Entonces, tenemos lo siguiente:

```
bcc (Let i x ty t (Sc1 t')) b = do
  if localScope t' 0
  then do
    be <- bcc t' (DROP:b)
    bcc t (SHIFT:be)
  else do
    let t'' = indexShift t' 0
    be <- bcc t'' b
    bcc t (POP:be)
```

Observemos que si el nuevo valor es utilizado, el código generado será el mismo. Por otro lado, si no se utiliza, se descartan el `SHIFT` y el `DROP`, pero se agrega el `POP`, es decir, la nueva compilación tendrá, al menos, un elemento menos que la original.

Otro cambio en el código se produce al tener un `DROP` antes de un `STOP` o un `RETURN`. En ambos casos, la instrucción `DROP` no es necesaria porque esta sólo afecta al entorno actual de la máquina, el cual se ve descartado al ejecutar `STOP` o `RETURN`. Es por esto que en la nueva versión se tiene lo siguiente:

```
bcc (Let i x ty t (Sc1 t')) b@[STOP] =
  if localScope t' 0
  then do
    be <- bcc t' b
    bcc t (SHIFT:be)
  else do
    let t'' = indexShift t' 0
```

```
    be <- bcc t'' b
    bcc t (POP:be)
bcc (Let i x ty t (Sc1 t')) b@(RETURN:_) = do
  if localScope t' 0
  then do
    be <- bcc t' b
    bcc t (SHIFT:be)
  else do
    let t'' = indexShift t' 0
    be <- bcc t'' b
    bcc t (POP:be)
```

Por lo tanto, la nueva compilación tendrá, al menos, un elemento menos que la original. Al igual que en el caso anterior, si el nuevo valor no es utilizado, se descartan el `SHIFT` y el `DROP`, pero se agrega el `POP`. Por otro lado, si el valor es utilizado, se deja el `SHIFT`, pero se descarta el `DROP`.

Lo mismo sucede si se trata de un `Let` en posición de cola. La instrucción `DROP` no será necesaria porque el entorno actual de la máquina se verá descartado:

```
bct (Let i x ty t (Sc1 t')) b = do
  if localScope t' 0
  then do
    be <- bct t' b
    bcc t (SHIFT:be)
  else do
    let t'' = indexShift t' 0
    be <- bct t'' b
    bcc t (POP:be)
```

El resto de los casos quedaron igual, por lo tanto la nueva compilación generará el mismo código que en la versión anterior.

2 Ejercicio 7

En la Macchina en C, previo a la optimización, no se utilizaba memoria constante porque los múltiples llamados a la función `f` requerían de instrucciones innecesarias que ocupaban

espacio en el stack. Con la optimización de `TAILCALL` omitimos las direcciones de retorno intermedias que genera `f` manteniendo la memoria constante.