

Ejercicios opcionales del apunte Optimizando la Máquina Virtual

Ejercicio 5

Se modifica la función `bcc`, añadiendo un nuevo comportamiento cuando tenemos un término `let`, cuya variable luego no es usada. Para su implementación se agrega una nueva instrucción `POP` que simplemente saca un elemento de la pila.

Ejercicio 6

Para demostrar que la nueva compilación nunca genera un código más grande que la versión del apunte anterior, haremos una prueba por inducción en la estructura del término de entrada a ambas versiones de la función `bcc`.¹

Antes, veamos los cambios que quedaron en las distintas versiones de la función:

```
bcc (V _ (Global x)) = failFD4 $ "Variable global en bytecode "++x
bcc (Const _ (CNat n)) = return [CONST, n]
- bcc (Lam _ _ _ (Sc1 t)) = do t' <- bcc t
-   return $ [FUNCTION, length t' + 1] ++ t' ++ [RETURN]
+ bcc (Lam _ _ _ (Sc1 t)) = do t' <- bcc t -- bcc agrega un RETURN al final
+   return $ [FUNCTION, length t'] ++ t'
bcc (App _ t1 t2) = do t1' <- bcc t1
                    t2' <- bcc t2
                    return $ t1'++ t2'++ [CALL]
@@ -127,8 +130,8 @@ bcc (BinaryOp _ Add t1 t2) = do t1' <- bcc t1
bcc (BinaryOp _ Sub t1 t2) = do t1' <- bcc t1
                    t2' <- bcc t2
                    return $ t1'++ t2' ++ [SUB]
- bcc (Fix _ _ _ _ (Sc2 t)) = do t' <- bcc t
-   return $ [FUNCTION, length t' + 1] ++ t' ++ [RETURN, FIX]
+ bcc (Fix _ _ _ _ (Sc2 t)) = do t' <- bcc t
+   return $ [FUNCTION, length t'] ++ t' ++ [FIX]
bcc (If2 _ c t1 t2) = do c' <- bcc c
                    t1' <- bcc t1
                    t2' <- bcc t2
@@ -139,6 +142,31 @@ bcc (Let _ _ _ t1 (Sc1 t2)) = do t1' <- bcc t1
                    t2' <- bcc t2
                    return $ t1' ++ [SHIFT] ++ t2' ++ [DROP]
```

¹La prueba está hecha previo a cambios del ejercicio anterior, que introducirán unos reemplazos de términos que no influyen

Notación: llamaremos bcc a la implementación original de la función y bcc' a la versión nueva.

También veamos la implementación de la nueva función bct :

```
+ bct :: MonadFD4 m => TTerm -> m Bytecode
+ bct (App _ t1 t2) = do t1' <- bcc t1
+                      t2' <- bcc t2
+                      return $ t1' ++ t2' ++ [TAILCALL]
+ bct (IfZ _ c t1 t2) = do c' <- bcc c
+                          t1' <- bct t1
+                          t2' <- bct t2
+                          return $ c' ++ [IFZ, length t1'] ++ t1' ++ t2'
+ bct (Let _ _ t1 (Sc1 t2)) = do t1' <- bcc t1
+                               t2' <- bct t2
+                               return $ t1' ++ [SHIFT] ++ t2'
+ bct t = do t' <- bcc t
+           return $ t' ++ [RETURN]
+
```

Nos es de interés esta función porque antes de demostrar lo solicitado probaremos el siguiente lema auxiliar:

Lema 1. *Dado un $TTerm$ t , la longitud del Bytecode producido por bct es menor o igual a la longitud del Bytecode producido por bcc más 1. Es decir,*

$$len(bct\ t) \leq len(bcc\ t) + 1$$

Demostración. Haremos la prueba por inducción en la estructura del término t .

Casos base: los casos base entrarán en el último patrón de la función, por lo que vemos fácilmente que vale $len(bct\ t) = len(bcc\ t) + 1$

Hipótesis de inducción: dado un término t , para cada subtérmino t' de t supondremos que vale $len(bct\ t') \leq len(bcc\ t') + 1$

Veamos que vale $len(bct\ t) \leq len(bcc\ t) + 1$ separando por casos:

- Caso $t = App\ _\ t_1\ t_2$:

$$\begin{aligned} len(bct\ t) &= len(bcc\ t_1\ ++\ bcc\ t_2\ ++\ [TAILCALL]) = \\ &len(bcc\ t_1\ ++\ bcc\ t_2\ ++\ [CALL]) = len(bcc\ t) \leq len(bcc\ t) + 1 \end{aligned}$$

- Caso $t = Ifz\ _\ c\ t_1\ t_2$

$$\begin{aligned}
len(bct\ t) &= len(bcc\ c \# [IFZ, _] \# bct\ t_1 \# bct\ t_2) \leq \\
&len(bcc\ c) + 2 + len(bcc\ t_1) + 1 + len(bcc\ t_2) + 1 = \\
&len(bcc\ c) + len(bcc\ t_1) + len(bcc\ t_2) + 4 = \\
&len(bcc\ c \# [IFZ, _] \# bcc\ t_1 \# [JUMP, _] \# bcc\ t_2) = \\
&len(bcc\ t) \leq len(bcc\ t) + 1
\end{aligned}$$

- Caso $t = Let\ _ _ _ t_1\ (Sc1\ t_2)$

$$\begin{aligned}
len(bct\ t) &= len(bcc\ t_1 \# bct\ t_2 \# [SHIFT]) = \\
&len(bcc\ t_1) + len(bct\ t_2) + 1 + 1 = \\
&len(bcc\ t_1 \# bcc\ t_2 \# [SHIFT]) + 1 = \\
&len(bcc\ t) + 1
\end{aligned}$$

- Otros casos: se procede de igual manera que en la prueba de los casos bases puesto que entran en el mismo patrón

□

Ahora sí, estamos en condiciones de probar lo que queríamos probar en un principio:

Teorema 1. *Dado un TTerm t , la longitud del Bytecode producido por bcc' es menor o igual a la longitud del Bytecode producido por bcc . Es decir,*

$$len(bcc'\ t) \leq len(bcc\ t)$$

Demostración. Procederemos por inducción en la estructura del término t .

Casos bases: se puede verificar que la definición de los términos V y Const no cambió.

Hipótesis de inducción: dado un término t , para cada subtérmino t' de t supondremos que vale $len(bcc'\ t') \leq len(bcc\ t')$

Veamos que vale que $len(bcc'\ t) \leq len(bcc\ t)$ separando por casos:

- Caso $t = Lam \dots t_1$

$$\begin{aligned}
 len(bcc' t) &= len([FUNCTION, _] \# bct t_1) = 2 + len(bct t_1) \leq \\
 &2 + len(bcc t_1) + 1 = \\
 &len([FUNCTION, _] \# bcc t_1 \# [RETURN]) = \\
 &bcc t
 \end{aligned}$$

- Caso $t = Fix \dots (Sc2 t)$

$$\begin{aligned}
 len(bcc' t) &= len([FUNCTION, _] \# bct t_1 \# [FIX]) = \\
 &2 + len(bct t_1) + 1 \leq \\
 &2 + len(bcc t_1) + 1 + 1 = \\
 &len([FUNCTION, _] \# bcc t_1 \# [RETURN, FIX]) = \\
 &bcc t
 \end{aligned}$$

- Otros casos: se puede ver que la función no cambia.

□

Ejercicio 7

En la versión anterior de la máquina, el uso de memoria para correr el programa no hubiese sido constante. En cada llamada a la función f , aumenta el tamaño de la pila, al deber agregarse la dirección de retorno cada vez. Con las optimizaciones realizadas en esta entrega, la nueva máquina corre el programa con memoria constante, puesto que con cada llamada a f no crece la pila. Esto ocurre porque no se guardan las direcciones intermedias de retorno de las sucesivas llamadas a f que se encuentran en posición de cola.