Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-422 — Compiladores

Apunte

Optimizando la Máquina Virtual

En el apunte anterior definimos la Macchina, una máquina virtual de pila, con las siguientes reglas de ejecución:

$$
\begin{array}{rlll}
\langle \texttt{CONST}\ n;\ c \mid & e \mid & s \rangle & \longrightarrow \langle c \mid e \mid n : s \rangle \\
\langle \texttt{ADD};\ c \mid & e \mid & n : m : s \rangle & \longrightarrow \langle c \mid e \mid m + n : s \rangle \\
\langle \texttt{SUB};\ c \mid & e \mid & n : m : s \rangle & \longrightarrow \langle c \mid e \mid m \dot{-} n : s \rangle \\
\langle \texttt{ACCESS}\ i;\ c \mid & e \mid & s \rangle & \longrightarrow \langle c \mid e \mid e!i : s \rangle \\
\langle \texttt{CALL};\ c \mid & e \mid & v : (e_f, c_f) : s \rangle & \longrightarrow \langle c_f \mid v : e_f \mid (e, c)_{RA} : s \rangle \\
\langle \texttt{FUNCTION}(c_f);\ c \mid & e \mid & s \rangle & \longrightarrow \langle c \mid e \mid (e, c_f) : s \rangle \\
\langle \texttt{RETURN};\ \_ \mid & \_ \mid & v : (e, c)_{RA} : s \rangle & \longrightarrow \langle c \mid e \mid v : s \rangle \\
\langle \texttt{PRINTN};\ c \mid & e \mid & n : s \rangle & \xrightarrow{n} \langle c \mid e \mid n : s \rangle \\
\langle \texttt{PRINT};\ x_0;\ \ldots\ x_n;\ \texttt{NULL};\ c \mid & e \mid & s \rangle & \xrightarrow{x_0 \ldots x_n} \langle c \mid e \mid s \rangle \\
\langle \texttt{FIXPOINT}(c_f);\ c \mid & e \mid & s \rangle & \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, c_f) : s \rangle \\
\langle \texttt{SHIFT};\ c \mid & e \mid & v : s \rangle & \longrightarrow \langle c \mid v : e \mid s \rangle \\
\langle \texttt{DROP};\ c \mid & v : e \mid & s \rangle & \longrightarrow \langle c \mid e \mid s \rangle
\end{array}
$$

Estas reglas deben ser completadas con las relevantes a su implementación de ifz.

1. La Macchina Implementada en C

Como prometimos en el apunte anterior, la máquina virtual puede implementarse en un lenguaje de más bajo nivel donde cada instrucción tiene un costo muy bajo. El archivo `macc.c` en el repo contiene la implementación, y usa el mismo formato y opcodes que la versión en Haskell, por lo cual deberían ser equivalentes.

El código está documentado como para ser leído, pero exponemos las ideas principales de la máquina aquí. En la función `run` hay 3 variables: `c`, `e` y `s`, punteros a código, entorno y pila respectivamente. El entorno `e` es una lista enlazada, inicialmente vacía. La pila está representada por un arreglo `stack`, y un puntero `s` que apunta a la primera posición libre (es decir, una posición adelante de la cima). Por ello, el pushear un valor `v` a la pila se hace simplemente con `*s++ = v`, y quitar un valor se hace con `v = *--s`. La máquina interpreta la instrucción apuntada por el puntero de código, es decir el valor `*c`, incrementa el puntero `c` y luego salta al código que implementa esa instrucción (via el `switch`).

La implementación de la instrucción `ADD`, por ejemplo, es entonces esencialmente:

```
case ADD: {
    uint32_t y = (*--s).i;
    uint32_t x = (*--s).i;
    (*s++).i = x+y;
    break;
}
```

En la implementación en `macc.c`, la pila no es de enteros, sino de `union value`: una unión (no etiquetada) de los distintos tipos de valores que soporte la máquina.

```
union value {
  int i;
  struct clo clo;
};
typedef union value value;
```

Es decir, también incluye clausuras y direcciones de retorno que definimos con la estructura `clo`.[1]

```
struct clo {
  env  clo_env;
  code clo_body;
};
```

Pregunta. ¿Por qué no usamos un arreglo para los entornos?

2. Llamadas (y Recursión) de Cola

Consideremos el siguiente fragmento de código FD4.

```
let g (x:Nat) : Nat = ...
let f (x:Nat) : Nat = g (1 + x)
      in f 0
```

Analicemos qué ocurre al evaluar la aplicación `f 0`. Esta aplicación nos llevará a ejecutar una aplicación `g 1`, donde tendremos en la pila el valor del argumento (1), una clausura para `g`, y la dirección de retorno de `f`. Luego de la llamada, `f` retorna el resultado con `RETURN`. La traza es algo como:

$$
\begin{array}{rrrrr}
 & \langle \texttt{CALL; RETURN} \mid & e \mid & 1 : (e_g, c_g) : (e', c)_{RA} : s \rangle & (e_0) \\
\longrightarrow & \langle c_g \mid & 1 : e_g \mid & (e, c_{RET})_{RA} : (e', c)_{RA} : s \rangle & (e_1) \\
\longrightarrow^* & \langle \texttt{RETURN} \mid & 1 : e_g \mid & v : (e, c_{RET})_{RA} : (e', c)_{RA} : s \rangle & (e_2) \\
\longrightarrow & \langle \texttt{RETURN} \mid & e \mid & v : (e', c)_{RA} : s \rangle & (e_3) \\
\longrightarrow & \langle c \mid & e' \mid & v : s \rangle & (e_4)
\end{array}
$$

donde c_{RET} apunta al `RETURN` y v es el resultado de evaluar g 1.

En el estado (e_0), cuando `f` está por llamar a `g`, tiene su propia dirección de retorno guardada en la pila, después de los dos elementos necesarios para efectuar el `CALL`. Al hacer la llamada, en (e_1), agrega una nueva dirección de retorno para el `RETURN` de `f`, y salta a `g`. La función `g` hace alguna cantidad de pasos, y llega a un `RETURN` (estado (e_2)) habiendo agregado exactamente un elemento a la pila. Finalmente, en (e_3), `g` retorna a `f` y en (e_4) `f` retorna a su llamante. Resumiendo, `f` queda como "intermediario" de la llamada a `g`, guardando una dirección de retorno a su `RETURN`.

En este caso, cuando una función directamente retorna el resultado de una llamada *sin operar sobre él*, decimos que es una llamada *de cola*. En otras palabras, la llamada a `g` desde `f` es de cola porque es lo *último* que hace `f`. En estos casos, podemos optimizar la ejecución mediante la *omisión* de la dirección de retorno intermedia, y pasando la propia de la función llamante en su lugar. Para ello, introducimos una nueva instrucción `TAILCALL` a tal efecto. Si `f` fuera compilada con una `TAILCALL` a `g`, la traza sería algo como:

$$
\begin{array}{rrrrr}
 & \langle \texttt{TAILCALL} \mid & e \mid & 1 : (e_g, c_g) : (e', c)_{RA} : s \rangle \\
\longrightarrow & \langle c_g \mid & 1 : e_g \mid & (e', c)_{RA} : s \rangle \\
\longrightarrow^* & \langle \texttt{RETURN} \mid & 1 : e_g \mid & v : (e', c)_{RA} : s \rangle \\
\longrightarrow & \langle c \mid & e' \mid & v : s \rangle
\end{array}
$$

[1] Para las direcciones de retorno, usamos el caso `clo`. Si bien existe una diferencia conceptual entre clausuras y direcciones de retorno, es totalmente inútil crear un nuevo caso idéntico. El caso `clo` bien podría estar nombrado `clo_or_ra`.

Con lo que ahorramos un elemento en la pila y un paso. Aunque a simple vista esto no parece ser una gran optimización, la verdadera ganancia se manifiesta en las funciones recursivas de cola.

Las funciones *recursivas de cola* son aquellas funciones en las cuales sus llamadas recursivas ocurren sólo en posición de cola. Por ejemplo:

```
let rec suma (m n : Nat) : Nat =
  ifz n then m
        else suma (1 + m) (n - 1)
```

Aquí, la optimización de llamadas de cola reduce *asintóticamente* la memoria consumida por esta función, de $O(n)$ a $O(1)$. Notar la diferencia con la versión de `suma` vista anteriormente, que no es recursiva de cola. Ser consciente de la recursión de cola y poder escribir código en esa forma es una habilidad esencial para los programadores funcionales. De hecho, es muy probable que su implementación de la máquina CEK y de la Macchina en Haskell sean recursivas de cola. Sin embargo, no es una propiedad única a los lenguajes funcionales: en C, la recursión de cola puede implementarse como un salto al inicio de la función, algo que todo compilador decente realiza cuando se habilitan las optimizaciones.

3. Compilando Llamadas de Cola

Por suerte, tenemos una definición muy concreta de lo que es una llamada de cola: un `CALL` seguido de un `RETURN`. La idea para mejorar la compilación es dividirla en dos funciones $\mathcal{C}(-)$ y $\mathcal{T}(-)$, mutuamente recursivas. La primera es la compilación de un término general a bytecode, mientras que la segunda es usada para la compilación de un término *en posición de cola*. La compilación en posición de cola $\mathcal{T}(-)$ tiene una regla especial para una aplicación, generando una llamada de cola. En todo otro caso se interpreta solamente como $\mathcal{C}(-)$, pero agregando un `RETURN` al final.

$$\begin{aligned} \mathcal{T}(ab) &= \mathcal{C}(a);\ \mathcal{C}(b);\ \texttt{TAILCALL} \\ \mathcal{T}(t) &= \mathcal{C}(t);\ \texttt{RETURN} \qquad\qquad \textit{(en otro caso)} \end{aligned}$$

$$\begin{aligned} \mathcal{C}(\lambda t) &= \texttt{FUNCTION}(\mathcal{T}(t)) \\ \mathcal{C}(ab) &= \mathcal{C}(a);\ \mathcal{C}(b);\ \texttt{CALL} \\ &\ldots \textit{otras reglas de } \mathcal{C}(-) \ldots \end{aligned}$$

Con estas reglas, vemos que la compilación (con $\mathcal{C}(-)$) de un término $\lambda.fe$ resulta:

$$\texttt{FUNCTION}(\mathcal{C}(f);\ \mathcal{C}(e);\ \texttt{TAILCALL})$$

como es de esperar.

Pregunta. Una idea para implementar esta optimización es tomar el bytecode compilado de la forma original y reemplazar las ocurrencias de `[CALL; RETURN]` por `[TAILCALL]`. ¿Qué problemas puede haber con este enfoque?

Hay que notar que la mayoría de las llamadas recursivas ocurren bajo un ifz, por lo que esta optimización como está escrita no surte mucho efecto. Pero podemos notar que si un ifz está en posición de cola, entonces sus ramas también lo están. En otras palabras, si el ifz es lo último a evaluar antes de retornar, entonces sus ramas también tienen esa propiedad (cuando son ejecutadas). Entonces, podemos agregar una regla como la siguiente a $\mathcal{T}(-)$:

$$\mathcal{T}(\textsf{ifz } c \textsf{ then } t \textsf{ else } e) = \mathcal{C}(c);\ \texttt{CJUMP}\ n;\ \mathcal{T}(t);\ \mathcal{T}(e)$$

donde la instrucción `CJUMP` chequea si el valor de la pila es 0 o no. En el caso de que sea 0, no hace nada. Esto causará que se ejecute el código $\mathcal{T}(t)$ y se retorne antes de llegar al código de $\mathcal{T}(e)$. En el caso de que el valor de la pila no sea 0, la instruccion `CJUMP` salta n posiciones. Tomando $n = |\mathcal{T}(t)|$ conseguimos que salte a la rama del else. Puede notar cierta familiaridad entre `CJUMP` y un salto condicional relativo en lenguaje ensamblador: son exactamente lo mismo.

Por otro lado, hay otra instrucción que podemos obviar antes de un `RETURN`: la instrucción `DROP`. Esta instrucción sólo afecta el entorno actual de la máquina, que se ve descartado al ejecutar `RETURN`. Como `DROP` sólo se genera al final de un letbinding, podemos optimizar los lets en posición de cola agregando otro caso a la función $\mathcal{T}(-)$.

$$\mathcal{T}(\texttt{let}\ x = m\ \texttt{in}\ n) = \mathcal{C}(m);\ \texttt{SHIFT};\ \mathcal{T}(n)$$

De esta forma, evitamos el `DROP`.

$$\mathcal{C}(\lambda.\ \texttt{let}\ f = g\ \texttt{in}\ fx) = \texttt{FUNCTION}(\mathcal{C}(g);\ \texttt{SHIFT};\ \texttt{ACCESS}\ 0;\ \mathcal{C}(x);\ \texttt{TAILCALL})$$

4. Tareas

1. Descargue el archivo `macc.c` y complete las partes faltantes, incluyendo las necesarias para su implementación de ifz.

2. Implemente la instrucción `TAILCALL` en ambas Macchinas (Haskell y C).

3. Cambie la compilación a bytecode para que use llamadas de cola en lo posible, siguiendo §2. Intente verificar la mejora en uso de memoria para alguna función apropiada.

4. Un programa siempre termina con `STOP`, con lo cual es inútil tener `DROP`s antes del mismo. Arregle la compilación para evitarlo. ¿Puede mejorarse algo más?

5 (opcional). En un programa como el siguiente:

```
let f (x:Nat) : Nat =
  let _  : Nat = print "x = " x in
  let x2 : Nat = ... in
  let _  : Nat = print "x2 = " x2 in
  let x3 : Nat = ... in
  let _  : Nat = print "x3 = " x3 in
  let x4 : Nat = ... in
  let _  : Nat = print "x4 = " x4 in
  let x5 : Nat = ... in
```

cada línea con un print introduce un nuevo valor al entorno en cual se evalúa la función, pero ninguno de estos valores son usados (los '_' representan alguna variable no usada en el resto de la función). ¿Puede mejorar la compilación para solucionar este problema? ¿Necesita instrucciones nuevas?

6 (opcional). Demuestre que la nueva compilación nunca genera un código más grande que versión del apunte anterior.

7 (opcional). Para el programa:

```
let rec f (x:Nat) : Nat = f x
let ret : Nat = f 0
```

¿Usan memoria constante ambas Macchinas? Explique.

5. Otras Fuentes

Ver [1, §15.6]. También [3] y [2] pueden servir.

Referencias

[1] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, new ed edition, July 2004.

[2] HaskellWiki. Tail recursion. `https://wiki.haskell.org/Tail_recursion`.

[3] Guy Lewis Steele. Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the 1977 Annual Conference*, ACM '77, page 153–162, New York, NY, USA, 1977. Association for Computing Machinery.