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

Apunte

Compilación a Máquina Virtual

En el capítulo anterior presentamos la máquina abstracta CEK, que implementa una estrategia de reducción CBV sobre los términos de FD4 de manera (más) eficiente. Ahora, en vez de quedarnos con la sintaxis de FD4 e interpretarla, compilaremos los programas al código de una máquina *virtual* (llamada *la Macchina*) de nivel más bajo, en donde el "control" ya no es un término FD4, sino una secuencia de *instrucciones virtuales* llamada comúnmente "bytecode".

Implementaremos la compilación de FD4 a bytecode y también la máquina virtual que lo ejecuta, en Haskell.

Terminología. Llamamos "máquina abstracta" a una descripción abstracta de una máquina que opera sobre *sintaxis*, es decir sobre los términos de FD4 o el lenguaje de interés. Por el contrario, una "máquina virtual" opera sobre un lenguaje de bajo nivel propio, compuesto de instrucciones, y *plano* (sin estructura). Esta terminología es la que usaremos en la cátedra aunque, como mucha terminología en el estudio de lenguajes de programación, no es universalmente compartida. Por otro lado, el término "máquina virtual" también se usa para software que virtualiza un sistema entero como un proceso dentro de un sistema operativo (e.g. VirtualBox), algo que tiene poca relación con nuestros objetivos.

Nuestra máquina virtual será de la familia de máquinas *de pila*, en donde la ejecución mantiene una pila con los valores computados. Algunas operaciones agregan valores a la pila y algunas los consumen. En la Macchina llevaremos también un *entorno* con los valores de cada variable local, muy similarmente a la máquina CEK, y también usando una noción similar a los índices de de Bruijn.

Un bytecode no es más que una secuencia de enteros (de un tamaño fijo), representando tanto a las instrucciones, mediante sus códigos de operación (*opcode*), como a los argumentos de algunas de ellas. El proceso comienza con un término e que se compila vía una función $\mathcal{C}(-)$ (detallada abajo) para obtener bytecode. Luego, ese bytecode puede ejecutarse en la Macchina.

1. Forma de la Macchina

Los estados de la máquina son de la forma $\langle c \mid e \mid s \rangle$, donde:

- c es una secuencia de instrucciones, la cual se recorre de manera mayormente secuencial. Esta secuencia nunca se modifica, sólo se recorre. Puede convenir pensar que c es un *puntero* a una secuencia (read-only) de instrucciones.
- e es un entorno: una lista de valores donde el primer elemento contiene el valor de la variable 0, y así sucesivamente.
- s es una pila de valores, que pueden ser de distintos tipos, detallados más adelante. Algunas instrucciones agregan y/o consumen elementos de la pila, pero siempre cerca de su tope.

Los valores de la máquina pueden ser de los siguientes tipos:

- Naturales: representados con n,
- Clausuras: un par de un entorno y un puntero a código (e, c), o

- Direcciones de retorno: también son un par de entorno y puntero a código $(e, c)_{RA}$.

En **ningún caso** la máquina *observa* el tipo de un valor: las acciones de la máquina están solamente determinadas por las instrucciones. Si, por ejemplo, se ejecuta una instrucción que espera un natural en la pila pero en realidad hay una clausura, la Macchina *no hace garantías* sobre su comportamiento, quedando indefinido. De hecho, cuando implementemos esta máquina en un lenguaje de nivel más bajo, usaremos uniones no-etiquetadas para los valores[1], con lo cual no habrá manera de detectar un error.

Claramente, los términos bien tipados deben compilar a código que nunca invoca comportamiento no definido. La distinción entre los distintos tipos de valor es entonces sólo tipográfica, para hacer la lectura de este documento más amigable.

El estado inicial de la máquina es de la forma $\langle c \mid \cdot \mid \cdot \rangle$. Aquí, c es un puntero al código a evaluar, y el entorno y la pila comienzan vacíos.

2. Intuición: Fragmento Aritmético

Vamos a considerar primero el fragmento aritmético de FD4: sin funciones, sólo enteros y operaciones sobre ellos. La idea es que las constantes enteras se implementan como una instrucción `CONST` n que agrega al elemento n a la cima de la pila. Las instrucciones `ADD` y `SUB` operan sobre la cima de la pila, extrayendo sus dos argumentos de la misma y luego agregando el resultado. Al ser así, las instrucciones aparecen *luego* de las instrucciones que agregan el valor necesario a la pila.

Entonces, podemos definir la etapa de compilación como:

$$\begin{aligned} \mathcal{C}(n) &= \texttt{CONST}\ n \\ \mathcal{C}(e + e') &= \mathcal{C}(e);\ \mathcal{C}(e');\ \texttt{ADD} \\ \mathcal{C}(e - e') &= \mathcal{C}(e);\ \mathcal{C}(e');\ \texttt{SUB} \end{aligned}$$

Mientras que las reglas relevantes de la máquina son:

$$\begin{aligned} \langle \texttt{CONST}\ n;\ c \mid e \mid s \rangle &\longrightarrow \langle c \mid e \mid n : s \rangle \\ \langle \texttt{ADD};\ c \mid e \mid n : m : s \rangle &\longrightarrow \langle c \mid e \mid m + n : s \rangle \\ \langle \texttt{SUB};\ c \mid e \mid n : m : s \rangle &\longrightarrow \langle c \mid e \mid m \dot{-} n : s \rangle \end{aligned}$$

De esta manera, si compilamos $2 + (3 - 1)$, obtenemos el bytecode:

$$\texttt{CONST}\ 2;\ \texttt{CONST}\ 3;\ \texttt{CONST}\ 1;\ \texttt{SUB};\ \texttt{ADD};$$

y al evaluarlo en la máquina obtenemos la traza:

$$\begin{aligned} \langle \texttt{CONST}\ 2;\ \texttt{CONST}\ 3;\ \texttt{CONST}\ 1;\ \texttt{SUB};\ \texttt{ADD};\ c \mid e \mid s \rangle &\longrightarrow \\ \langle \texttt{CONST}\ 3;\ \texttt{CONST}\ 1;\ \texttt{SUB};\ \texttt{ADD};\ c \mid e \mid 2 : s \rangle &\longrightarrow \\ \langle \texttt{CONST}\ 1;\ \texttt{SUB};\ \texttt{ADD};\ c \mid e \mid 3 : 2 : s \rangle &\longrightarrow \\ \langle \texttt{SUB};\ \texttt{ADD};\ c \mid e \mid 1 : 3 : 2 : s \rangle &\longrightarrow \\ \langle \texttt{ADD};\ c \mid e \mid 2 : 2 : s \rangle &\longrightarrow \\ \langle c \mid e \mid 4 : s \rangle & \end{aligned}$$

para cualquier valor de c, e y s. Como regla general, para cualquier término t (del fragmento aritmético o no), ejecutar $\mathcal{C}(t)$ tiene el efecto de agregar el resultado de evaluar t a la pila.

[1] En Haskell, un tipo `data Either a b = Left a | Right b` es una unión *etiquetada* de a y b, que permite distinguir en cuál caso se está dado un valor del tipo. En contraste, las `union` de C son no-etiquetadas.

3. Compilando λ-términos

Con esta intuición sobre el manejo de la pila, tornamos la atención a los λ-términos. Tenemos dos problemas a resolver: el manejo de variables locales y el proceso de llamada y retorno. El acceso a una variable local se compila como una instrucción `ACCESS`, la cual recupera su valor desde el entorno. La aplicación se compila como la evaluación secuencial de la función y su argumento, que agregará dos elementos a la pila, y una instrucción `CALL` que efectua la llamada a función. Para soportar esto, nuestra compilación de funciones inserta una instrucción `RETURN` en el cuerpo de las funciones, para que al terminar de evaluar el cuerpo se vuelva al código que la llamó[2]. Entonces, la compilación se extiende con:

$$\begin{aligned} \mathcal{C}(v_i) &= \texttt{ACCESS}\ i \\ \mathcal{C}(fe) &= \mathcal{C}(f);\ \mathcal{C}(e);\ \texttt{CALL} \\ \mathcal{C}(\lambda t) &= \texttt{FUNCTION}(\mathcal{C}(t);\ \texttt{RETURN}) \end{aligned}$$

donde v_i representa a la variable de de Bruijn i.

Nota: La instrucción `FUNCTION`, como está escrita aquí arriba, contiene un "sub"-código y rompe con la idea de que el código es una secuencia de instrucciones. Luego veremos cómo codificarla de manera "serializada". Por ahora, podemos pensar que es una instrucción que puede contener bytecode dentro.

La idea aquí es que una instrucción `FUNCTION` agregará una clausura a la pila, compuesta del entorno actual y un puntero al cuerpo de la función. La instrucción `CALL`, tomará ese entorno y código de la pila, extenderá el entorno con el argumento de la función y *saltará* al cuerpo de la función. Antes de saltar, sin embargo, guardará una *dirección de retorno* en la pila para que sea usada por la instrucción `RETURN`. La instrucción `RETURN` salta incondicionalmente a la dirección de retorno en la pila, descartando el entorno y el puntero a código actual. Al ejecutar un `RETURN` se toma la dirección del segundo elemento de la pila, ya que en la cima está el valor resultado de evaluar el cuerpo, que permanece en la pila para que el llamante pueda usarlo.

$$\begin{aligned} \langle \texttt{ACCESS}\ i;\ c \mid e \mid s \rangle &\longrightarrow \langle c \mid e \mid e!i : s \rangle \\ \langle \texttt{CALL};\ c \mid e \mid v : (e_f, c_f) : s \rangle &\longrightarrow \langle c_f \mid v : e_f \mid (e, c)_{RA} : s \rangle \\ \langle \texttt{FUNCTION}(c_f);\ c \mid e \mid s \rangle &\longrightarrow \langle c \mid e \mid (e, c_f) : s \rangle \\ \langle \texttt{RETURN};\ \_ \mid \_ \mid v : (e, c)_{RA} : s \rangle &\longrightarrow \langle c \mid e \mid v : s \rangle \end{aligned}$$

Notar una diferencia conceptual entre las clausuras y las direcciones de retorno: las primeras requieren agregar un valor a su entorno (porque representan funciones sin aplicar), pero no así las segundas.

Veamos un pequeño ejemplo, la evaluación del término $(\lambda x.x + 4)\ 10$ dentro de algún contexto arbitrario. El bytecode generado es:

$$\texttt{FUNCTION}(\texttt{ACCESS}\ 0;\ \texttt{CONST}\ 4;\ \texttt{ADD};\ \texttt{RETURN});\ \texttt{CONST}\ 10;\ \texttt{CALL};\ k$$

donde agregamos un k, posiblemente no vacío, representando la continuación del bytecode generada por otras partes del programa. La traza que genera este código es (abreviamos con B el cuerpo de la clausura):

[2]Notar una diferencia con la máquina CEK: al llegar a un valor en CEK debíamos inspeccionar la pila para "retornar" a otra parte de la evaluación, según el tipo de nodo. Aquí toda esa complejidad se maneja durante la compilación, y la máquina salta "a ciegas".

$$\begin{aligned}
\langle \texttt{FUNCTION}(B);\ \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid & \quad e \mid & s \rangle &\longrightarrow \\
\langle \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid & \quad e \mid & (e, B) : s \rangle &\longrightarrow \\
\langle \texttt{CALL};\ k \mid & \quad e \mid & 10 : (e, B) : s \rangle &\longrightarrow \\
\langle B \mid & \ 10 : e \mid & (e, k)_{RA} : s \rangle &= \\
\langle \texttt{ACCESS}\ 0;\ \texttt{CONST}\ 4;\ \texttt{ADD};\ \texttt{RETURN} \mid & \ 10 : e \mid & (e, k)_{RA} : s \rangle &\longrightarrow \\
\langle \texttt{CONST}\ 4;\ \texttt{ADD};\ \texttt{RETURN} \mid & \ 10 : e \mid & 10 : (e, k)_{RA} : s \rangle &\longrightarrow \\
\langle \texttt{ADD};\ \texttt{RETURN} \mid & \ 10 : e \mid & 4 : 10 : (e, k)_{RA} : s \rangle &\longrightarrow \\
\langle \texttt{RETURN} \mid & \ 10 : e \mid & 14 : (e, k)_{RA} : s \rangle &\longrightarrow \\
\langle k \mid & \quad e \mid & 14 : s \rangle &
\end{aligned}$$

Como es de esperar, llegamos a la continuación del código habiendo agregado el valor 14 en la pila.

4. Let-Bindings Internos

Consideremos la compilación de $\texttt{let}\ x : \tau = e_1\ \texttt{in}\ e_2$. Intuitivamente, sólo tenemos que evaluar e_1, y luego evaluar e_2 con el valor de e_1 agregado al entorno. Para terminar, debemos quitar este valor del entorno para no afectar a las instrucciones que siguen ("olvidando" el valor al igual que lo hace un `RETURN`).

Para hacer esto, introducimos dos instrucciones `SHIFT` y `DROP`. La compilación y Macchina se modifican agregando las siguientes reglas:

$$\mathcal{C}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \ = \ \mathcal{C}(e_1);\ \texttt{SHIFT};\ \mathcal{C}(e_2);\ \texttt{DROP}$$

$$\begin{aligned}
\langle \texttt{SHIFT};\ c \mid & \quad e \mid v : s \rangle &\longrightarrow \langle c \mid v : e \mid s \rangle \\
\langle \texttt{DROP};\ c \mid v : e \mid & \quad s \rangle &\longrightarrow \langle c \mid \quad e \mid s \rangle
\end{aligned}$$

Para convencerse de la necesidad del `DROP`, considere compilar el término $(\lambda x.\ (\texttt{let}\ y = 1\ \texttt{in}\ 2) + x)\ 0$ y su ejecución.

5. Imprimiendo Valores

Hasta ahora, la Macchina no tiene ningún efecto salvo consumir electricidad. Para observar información sobre naturales utilizamos la instrucción `PRINTN` que imprime el valor (natural) al tope de la pila, que se ejecuta vía la siguiente regla

$$\langle \texttt{PRINTN};\ c \mid e \mid n : s \rangle \xrightarrow{n} \langle c \mid e \mid n : s \rangle$$

es decir: no tiene efecto en el estado de la máquina, pero imprime el entero n por la salida estándar al ejecutarse.

La máquina también puede imprimir literales de cadenas (es decir, aquellas que son conocidas antes de la ejecución). Para esto utilizamos la instrucción `PRINT`. La misma imprime la cadena que sigue a la instrucción, terminada por un `NULL`.

$$\langle \texttt{PRINT};\ x_0;\ x_1;\ \ldots\ x_n;\ \texttt{NULL};\ c \mid e \mid s \rangle \xrightarrow{x_0 x_1 \ldots x_n} \langle c \mid e \mid s \rangle$$

Cada x_i es un natural representando un punto de código Unicode (en otras palabras, el caracter representado con codificación UTF-32). Por ejemplo al caracter '`a`' le corresponde el 97, a la '`ñ`' el 241, y a nuestra querida letra 'λ' el 955.

Se deja de ejercicio la compilación del operador print.

Con esto, ya tenemos casi todo el núcleo de FD4 implementado. Sin embargo, brillan por su ausencia las funciones recursivas.

6. Fixpoints: Atando el Nudo

Advertencia: naturalmente, esta sección puede llevar algunas iteraciones para ser entendida por completo.

Una forma de compilar un punto fijo es introducir un nuevo tipo de valor para representar las clausuras de funciones recursivas:

$$(e, c)_{\text{FIX}}$$

compilando los fix de manera similar a los λ (i.e. evaluar un fix pone una clausura $(e, c)_{\text{FIX}}$ en la pila), y agregar la siguiente regla para `CALL`:

$$\text{(ERRÓNEA)} \qquad \langle \texttt{CALL};\ c \mid e \mid v : (e_f, c_f)_{\text{FIX}} : s \rangle \longrightarrow \langle c_f \mid v : (e_f, c_f)_{\text{FIX}} : e_f \mid (e, c)_{RA} : s \rangle$$

¡pero esto implica llevar etiquetas! Al llegar a un `CALL`, tenemos que distinguir si la clausura en la pila es recursiva, para decidir si pasarle su binding recursivo o no. Notar que este chequeo *debe ser dinámico*, no hay forma de decidir estáticamente y con precisión en un llamada fe si f evalúa a una función recursiva o no[3]. Entonces, esta regla atenta contra nuestro objetivo de ejecución eficiente.

Por suerte, no todo está perdido, y podemos tratar a las funciones recursivas y no recursivas uniformemente, como de hecho ocurre en los lenguajes reales. La idea en la Macchina es *atar el nudo* al momento de crear una clausura para una función recursiva, y luego usar `CALL` exactamente como estaba. Si miramos la regla (ERRÓNEA) de arriba, el binding recursivo $(e_f, c_f)_{\text{FIX}}$ no depende de v, ni de ninguna componente de la máquina al momento de hacer la llamada, por lo que podemos "hacerlo antes", al momento de crear la clausura. Entonces, vamos a compilar las funciones recursivas con un marcador distinto (recuerde que e tiene dos índices libres):

$$\mathcal{C}(\mathsf{fix}.\, e) \ = \ \texttt{FIXPOINT}(\mathcal{C}(e);\ \texttt{RETURN})$$

Hasta aquí, esto es análogo a las funciones no recursivas. La diferencia está al ejecutar esta instrucción. La regla es:

$$\langle \texttt{FIXPOINT}(c_f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, c_f) : s \rangle$$

donde e_{fix} debe extender al entorno e con un valor más, el binding recursivo para la función, para permitir que podamos llamarla como una función normal. Es decir, e_{fix} es de la forma:

$$e_{\text{fix}} = (e_1, c_f) : e$$

pero, ¿qué es e_1? Debe *también* ser un entorno que permita llamar a c_f como una función normal. Por lo tanto, también debe ser una extensión del entorno e, donde el primer valor es el binding recursivo de f, es decir

$$e_1 = (e_2, c_f) : e$$

y entonces,

$$e_{\text{fix}} = ((e_2, c_f) : e, c_f) : e$$

pero, estamos en el mismo problema de necesitar un e_2. Es claro que iterando este proceso no vamos a conseguir un entorno viable. Sin embargo, sí existe una solución engañosamente simple a la ecuación:

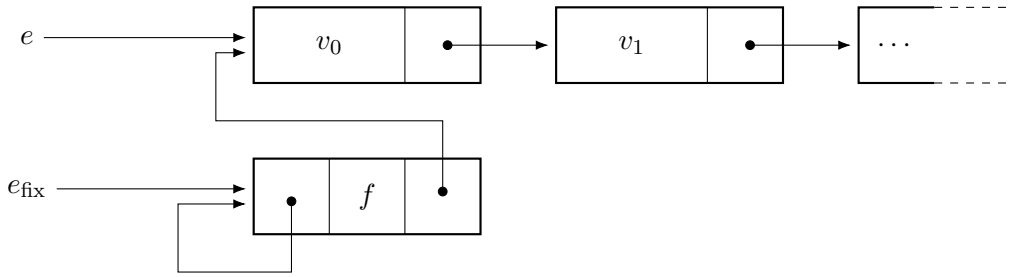
$$e_{\text{fix}} = (e_{\text{fix}}, c_f) : e$$

La solución consiste en ver a esa ecuación ¡como una definición! Si podemos formar un entorno e_{fix} *recursivo*, donde el entorno del primer valor de e_{fix} es exactamente e_{fix}, entonces podemos implementar esta regla, y de manera eficiente. Esto puede hacerse fácilmente en lenguajes que permitan estructuras de datos *cíclicas*. Por suerte, Haskell lo hace[4]. De hecho, esa misma ecuación es una definición posible de e_{fix}.

[3] El contraejemplo obvio a esto es compilar $(\lambda f.f\ 0)$, que puede tomar tanto funciones recursivas como no recursivas.

[4] Un pequeño ejemplo de esto es la definición `let unos = 1:unos` que define una lista circular (o infinita) de unos.

Para entender lo que pasa concretamente en la memoria, el siguiente diagrama puede ayudar. Antes de atar el nudo, e apunta a un entorno, que es una lista enlazada de valores. Para extenderlo, creamos un nodo, apuntado por (i.e. con el nombre) e_{fix}. El sucesor de e_{fix}, es decir el nodo siguiente en la lista, es simplemente e. Ahora bien, el valor contenido en el primer nodo de e_{fix} es una clasura: un par entorno-código. El código es c_f, y el entorno es el mismo e_{fix}.



Si programáramos en (por ejemplo) C, tendríamos que modificar unos punteros para generar la estructura cíclica del diagrama, lo que también es perfectamente implementable (aunque no tan elegantemente).

7. Compilando Programas

Otra parte de FD4 que omitimos arriba son las variables libres globales, que apuntan a definiciones "top-level" (i.e. declaraciones). Sólo vamos a soportar la compilación de programas (es decir, archivos) enteros, y no nos interesa mucho la compilación interactiva a bytecode (se puede hacer perfectamente, pero el manejo del entorno es engorroso) ni la compilación modular. Dada la compilación de let-bindings vista anteriormente, podemos pensar que un programa

$$P = \begin{array}{l} \texttt{let } v_1 = e_1 \\ \texttt{let } v_2 = e_2 \\ \cdots \\ \texttt{let } v_n = e_n \end{array}$$

puede ser traducido al *término*:

$$T_P = \begin{array}{l} \texttt{let } v_1 = e_1 \texttt{ in} \\ \quad \texttt{let } v_2 = e_2 \texttt{ in} \\ \quad\quad \cdots \\ \quad\quad\quad \texttt{let } v_{n-1} = e_{n-1} \texttt{ in} \\ \quad\quad\quad\quad e_n \end{array}$$

donde cada letbinding se evalúa en orden. Entonces, para compilar un programa P a bytecode, generamos el siguiente código:

$$\mathcal{C}(T_P);\ \texttt{STOP}$$

que en efecto evalúa todas las declaraciones top-level en orden, y luego detiene la máquina vía la instrucción `STOP`.

8. Representación Concreta e Implementación

- Representaremos un `Bytecode` simplemente como `[Int]`. Para escribirlo a un archivo, necesitamos un formato binario bien definido. Representaremos la lista codificando los enteros uno después de otro, usando 32 bits para cada uno [5]. Además, usamos un formato *little-endian* para la codificación de cada entero. No es necesario guardar la longitud del código ni insertar un terminador (¿por qué?).

[5] Ignoraremos problemas de overflow/underflow aquí. Mientras estemos lejos de 2^{31}, no habrá problema. En otras palabras, aplicaremos el algoritmo del avestruz.

- Cada instrucción (`ACCESS`, `CALL`, etc) es representada por su opcode. Por ello, salvo por su posición son indistinguibles de otros datos en el bytecode. Los valores concretos de los opcodes son irrelevantes.
- Las instrucciones que tienen un argumento, como `ACCESS`, son representadas por dos enteros, uno para el opcode y uno para su argumento. La máquina debería comportarse acordemente, avanzando dos posiciones de código en estos casos.
- La parte en la que fuimos un poco abstractos aún es con las clausuras: ¿cómo representamos `FUNCTION`(e)? Una forma es llevando la longitud de e para poder saltarla. Por ejemplo, si el opcode de `FUNCTION` es `0x42` y e es el bytecode `[10,11,12,13,14]`, representamos `FUNCTION`(e) con `[0x42,5,10,11,12,13,14]`. Al llegar al `0x42`, la máquina consume la longitud (5), guarda el puntero a e (apuntando al 10), y salta 5 posiciones hacia adelante.
- En Haskell, vamos a usar un tipo algebraico `Val` para representar a los valores.

  ```
  data Val = I Int | Fun Env Bytecode | RA Env Bytecode
  ```

  Repetimos: la máquina nunca debe analizar el constructor de este tipo para tomar una decisión. Si la instrucción es `ADD`, y los dos valores al tope de la pila no son `I`, la máquina aborta con un error. Si fuera una implementación sin etiquetas, tomaría lo que sea que esté en la pila y lo interpretaría como un entero. Los casos de `Fun` y `RA` pueden unirse en un solo constructor: no debería llevar a confusión en la máquina, pero es menos claro al leer el código.
- En vez de tener `FUNCTION` y `FIXPOINT`, tomaremos otro camino: usaremos solo `FUNCTION` y agregaremos una instrucción `FIX`, compilando los fixpoints a:

  $$\mathcal{C}(\mathsf{fix}.\, e) \;=\; \texttt{FUNCTION}(\mathcal{C}(e);\ \texttt{RETURN});\ \texttt{FIX}$$

  La ejecución de `FIX` toma la clausura de la pila y le ata el nudo recursivo como mostramos en la §6. Esto permite reusar un poco de código, pero implica que (brevemente) se rompa un invariante y exista una clausura mal formada en la pila (porque e tiene dos índices libres).
- Una nota sobre eficiencia: hay que notar que cada vez que "guardamos" un bytecode o un entorno durante la ejecución, esto es solamente la copia de un puntero, y por lo tanto se hace en $O(1)$. Esto es posible ya que los entornos son estructuras funcionales (y por lo tanto persistentes). Además, sólo existe una única pila durante la ejecución. Una ineficiencia inherente de usar listas para representar el código es que el "saltar" n instrucciones cuesta $O(n)$, pero podremos evitarlo en el próximo capítulo.

9. Bytecode en el *Mundo Real™*

Las máquinas virtuales de pila no son para nada un ejercicio teórico: muchos lenguajes populares implementan las usan para su compilación a bytecode. La principal ventaja de usar una máquina virtual es poder *definir* una arquitectura que viene a ocupar el lugar de un lenguaje intermedio. Al hacer esto, podemos compilar distintos lenguajes a la misma máquina, e implementar la máquina en distintas arquitecturas con un trabajo acotado. A la vez, nuestra arquitectura diseñada a medida puede ser más cómoda para las operaciones que más nos interesan (e.g. la primera máquina virtual para Erlang, la JAM, tenía una instrucción de un byte para el envío de un mensaje [1]). Por otro lado, la compilación es usualmente más rápida que si fuera a código nativo.

A saber, la familia Java, la familia .NET, OCaml y Python usan máquinas virtuales de pila (aunque la JVM también tiene registros). La JVM es apuntada por *muchos* lenguajes distintos, economizando la compilación vía una máquina común, y permitiendo una interoperación fluida entre distintos lenguajes. Los programas de la familia .NET se compilan a un código intermedio llamado CIL, y son ejecutados por el *common language runtime* (CLR) de manera de obtener código portable. OCaml y Python también

usan máquinas de pila para su compilación: OCaml lo hace para tener código portable entre arquitecturas, y Python para ganar algo de velocidad contra el código puramente interpretado.

La JVM implementa una técnica conocida como "compilación *just in time*" (JIT). Esencialmente, cuando la máquina está por ejecutar un bloque de código por primera vez, realiza una compilación a *código nativo*, lo recuerda, y luego lo ejecuta. De esta forma si bien el bytecode es totalmente portable, se puede aprovechar al completo el procesador al evaluarlo (pagando el precio de la compilación). Si bien este código se amortiza rápidamente para código "caliente", hay algunas heurísticas para decidir si es mejor interpretar o compilar. La máquina virtual del CLR hace *exclusivamente* compilación JIT, y nunca interpreta instrucciones.

10. Implementación

10.1. Serializando estructuras

En el archivo `Bytecompile.hs` encontrarán las definiciones para codificar y decodificar secuencias de enteros a un archivo. Pueden usar este archivo de plantilla para escribir el compilador a bytecode y la ejecución de la máquina virtual. En el mismo se define el tipo para el bytecode simplemente como una lista de enteros.

```
type Bytecode = [Int]
```

Las funciones que restan escribir son:

- `bc :: MonadFD4 m => Term -> m Bytecode`

  Compila un término a bytecode.

- `bytecompileModule :: MonadFD4 m => Module -> m Bytecode`

  Usa la función `bc` para compilar un módulo, utilizando la técnica descripta en la §7. Notar que aquí es necesario transformar variables `Global` en variables `Free` para poder usar nuestros dispositivos de manejo de variables ligadas.

- `runBC :: MonadFD4 m => Bytecode -> m ()`

  Implementa la Macchina, ejecutando bytecode.

Ya se proveen funciones `bcWrite :: Bytecode -> FilePath -> IO ()` para codificar secuencias de enteros `Bytecode` y escribirlas en un archivo, y `bcRead :: FilePath -> IO Bytecode` para leer de un archivo y decodificar a `Bytecode`.

Para codificar y decodificar cadenas puede utilizar las funciones `ord` y `chr` del módulo `Data.Char`.

10.2. Tarea 4

a) Implementar la compilación a bytecode y máquina virtual en Haskell, usando el esqueleto provisto. Dado que las instrucciones `PRINT` y `PRINTN` imprimen a la consola, la función que ejecuta la máquina debe estar en `MonadFD4`.

b) Proponer e implementar un esquema de compilación y ejecución para ifz. Pueden agregarse nuevas instrucciones a la máquina.

c) Implementar un esquema de compilación para el operador print usando las instrucciones `PRINT` y `PRINTN`.

d) Agregar opciones de compilación:

```
-m,--bytecompile        Compilar a la Macchina
-r,--runVM              Ejecutar bytecode en la Macchina
```

Al correr el compilador con la opción '`--bytecompile file.fd4`' debe generase un archivo `file.bc` con el bytecode. Similarmente, al correr con '`--runVM file.bc`', debe ejecutar el bytecode con la máquina virtual.

Referencias

[1] Joe Armstrong. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 6–1–6–26, New York, NY, USA, 2007. Association for Computing Machinery.