Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
Compiladores

# Apunte de cátedra: Una máquina virtual

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

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

**Nota de terminología** Llamamos "máquina abstracta" a una descripción abstracta de una máquina que opera sobre *sintaxis*, es decir sobre los términos de PCF o el lenguaje de interés, mientras que una máquina *virtual* opera sobre un lenguaje de bajo nivel, compuesto de instrucciones. Esta terminología, como muchas en el estudio de lenguajes de programación, no es universalmente compartida, es la que usaremos en la cátedra.

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 BVM 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 códigos de operación (*opcode*), como a los argumentos de algunas de ellas. El proceso comienza con un término $e$ que se compila via una función $\mathcal{C}(-)$ (detallada abajo) para obtener bytecode. Luego, ese bytecode puede ejecutarse en la BVM.

## 1. Forma de la Máquina

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. A veces conviene 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 del "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. De hecho, cuando implementemos esta máquina en un lenguaje de nivel más bajo, vamos a usar uniones no-etiquetadas para los valores[1]. Si una instrucción X espera (por ejemplo) un natural en la pila pero en realidad hay una clausura, *se va a ejecutar igual* y tener un comportamiento no definido. 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 $\langle c \mid \epsilon \mid \epsilon \rangle$, es decir un puntero a código con entorno y pila vacíos.

[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 cual caso se está dado un valor del tipo. En contraste, las `union` de C son no-etiquetadas.

## 2. Intuición: Fragmento Aritmético

Vamos a considerar primero el fragmento aritmético de PCF: sin funciones, sólo enteros y operadores unarios. 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 `SUCC` y `PRED` operan sobre el elemento en la cima de la pila correspondientemente. Al ser así, las instrucciones aparecen *luego* de las instrucciones que agregan el valor necesario a la pila.

Entonces, la etapa de compilación es:

$$\begin{aligned} \mathcal{C}(n) &= \texttt{CONST}\ n \\ \mathcal{C}(\mathsf{succ}\ e) &= \mathcal{C}(e);\ \texttt{SUCC} \\ \mathcal{C}(\mathsf{pred}\ e) &= \mathcal{C}(e);\ \texttt{PRED} \end{aligned}$$

Mientras que las reglas relevantes de la máquina son:

$$\begin{aligned} \langle \texttt{CONST}\ n;\ c \mid e \mid s \rangle &\longrightarrow \langle c \mid e \mid n : s \rangle \\ \langle \texttt{SUCC};\ c \mid e \mid n : s \rangle &\longrightarrow \langle c \mid e \mid n+1 : s \rangle \\ \langle \texttt{PRED};\ c \mid e \mid n : s \rangle &\longrightarrow \langle c \mid e \mid n-1 : s \rangle \end{aligned}$$

(con el abuso de notación que $0-1=0$). De esta manera, si compilamos $\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{pred}\ 1))$, obtenemos el bytecode:

$$\texttt{CONST}\ 1;\ \texttt{PRED};\ \texttt{SUCC};\ \texttt{SUCC}$$

y al evaluarlo en la máquina obtenemos la traza:

$$\begin{aligned} \langle \texttt{CONST}\ 1;\ \texttt{PRED};\ \texttt{SUCC};\ \texttt{SUCC};\ c \mid e \mid s \rangle &\longrightarrow \\ \langle \texttt{PRED};\ \texttt{SUCC};\ \texttt{SUCC};\ c \mid e \mid 1 : s \rangle &\longrightarrow \\ \langle \texttt{SUCC};\ \texttt{SUCC};\ c \mid e \mid 0 : s \rangle &\longrightarrow \\ \langle \texttt{SUCC};\ c \mid e \mid 1 : s \rangle &\longrightarrow \\ \langle c \mid e \mid 2 : s \rangle & \end{aligned}$$

para cualquier valor de $c$, $e$ y $s$. Como regla general, el ejecutar $\mathcal{C}(e)$ para una expresión $e$ tiene el efecto de pushear el resultado de evaluar $e$ en la pila. Los operadores que esperan argumentos esperan encontrarlos en el tope de la pila.

## 3. Compilando $\lambda$-Términos

Con esta intuición sobre el manejo de la pila, tornamos la atención a los $\lambda$-términos. El acceso a una variable local se compila como una instrucción `ACCESS` que busca su valor en el entorno. La aplicación se compila como la evaluación secuencial de la función y su argumento, que pusheará 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].

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

donde $v_i$ representa a la variable de de Bruijn $i$.

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 como 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 con un entorno 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 actual. La regla de `RETURN` toma la dirección

[2]Notar la 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".

del segundo elemento de la pila, ya que en la cima está el valor resultado de evaluar el cuerpo, que permanece en la pila.

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

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 $(\lambda x.\texttt{succ}\ x)\ 10$ dentro de algún contexto. El bytecode generado es:

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

donde $k$, posiblemente no vacío, es 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):

$$\begin{aligned}
\langle \texttt{FUNCTION}(B);\ \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid & \quad e \mid & & \quad s \rangle &&\longrightarrow \\
\langle \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid & \quad e \mid & & \quad (e, B) : s \rangle &&\longrightarrow \\
\langle \texttt{CALL};\ k \mid & \quad e \mid & & \ 10 : (e, B) : s \rangle &&\longrightarrow \\
\langle B \mid & \ 10 : e \mid & & \quad (e, k)_{RA} : s \rangle &&= \\
\langle \texttt{ACCESS}\ 0;\ \texttt{SUCC};\ \texttt{RETURN} \mid & \ 10 : e \mid & & \quad (e, k)_{RA} : s \rangle &&\longrightarrow \\
\langle \texttt{SUCC};\ \texttt{RETURN} \mid & \ 10 : e \mid & & \ 10 : (e, k)_{RA} : s \rangle &&\longrightarrow \\
\langle \texttt{RETURN} \mid & \ 10 : e \mid & & \ 11 : (e, k)_{RA} : s \rangle &&\longrightarrow \\
\langle k \mid & \quad e \mid & & \quad 11 : s \rangle &&
\end{aligned}$$

Como es de esperar, llegamos a la continuación del código habiendo agregado el valor 11 en la pila.

Hasta ahora, la máquina no tiene ningún efecto salvo consumir electricidad. Para observar información externamente vamos a agregar una instrucción `PRINT` que imprima el valor (natural) al tope de la pila, que se ejecuta via la regla:

$$\langle \texttt{PRINT};\ c \mid e \mid n : s \rangle \longrightarrow \langle c \mid e \mid n : s \rangle$$

es decir: no tiene efecto en la máquina, pero imprime el entero $n$ por la consola al ejecutarse (de alguna manera). Para finalizar una ejecución, tenemos una instrucción `STOP`, que detiene la máquina. Por ejemplo, entonces, para imprimir el resultado de evaluar una expresión $e$, podemos generar el bytecode $\mathcal{C}(e)$; `PRINT`; `STOP` y ejecutarlo.

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

# 4. Fixpoints: Poniéndole un Moño a la Recursión

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

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

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

y agregar el siguiente caso a `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 función 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 en un llamada $f e$ si $f$ evalua 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 es *atar el nudo* al momento de crear una clausura para una función recursiva, y luego usar `CALL` sin modificar. Si miramos la regla (ERRÓNEA) de arriba, el binding recursivo

[3] Al menos, sin complicar el sistema de tipos...

$(e_f, c_f)_{\text{FIX}}$ no depende de $v$, ni de ninguna componente de la máquina en ese momento, por lo que podemos "hacerlo antes", al momento de crear la clausura. Entonces, vamos a compilar las funciones recursivas con un marcador distinto (recordar que $e$ tiene dos índices libres):

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

Hasta aquí, 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 $e_{\text{fix}}$ extiende al entorno $e$ con un valor más, el binding recursivo para la función, lo que permite que podamos llamarla como una función normal. Es decir, es de la forma:

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

pero, ¿qué es $e_1$? Debe ser un entorno que permita llamar a $c_f$ como una función normal. Entonces, 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,

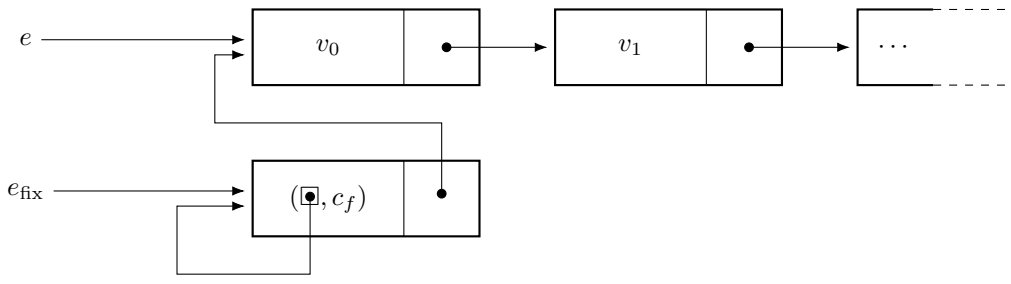
$$e_{\text{fix}} = ((e_2, c_f) : e_1, c_f) : e$$

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

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

Si podemos formar un entorno $e_{\text{fix}}$ *recursivo*, donde el entorno del primer valor de $e_{\text{fix}}$ es exactamente $e_{\text{fix}}$, entonces podemos implementar esta regla, y de manera eficiente. El generar estructuras de datos recursivas es permitido por algunos lenguajes, y Haskell es uno de ellos[4]. De hecho, la última ecuación es (módulo sintaxis) una definición válida en Haskell!

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 $e_{\text{fix}}$. El sucesor de $e_{\text{fix}}$, es decir el nodo siguiente en la lista, es simplemente $e$. Ahora bien, el valor que contiene el primer nodo de $e_{\text{fix}}$ es una clasura: un par entorno-código. El código es $c_f$, y el entorno es el mismo $e_{\text{fix}}$.




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).

# 5. Let-Bindings Internos

Consideremos la compilación de `let` $x : \tau = e_1$ `in` $e_2$. Para empezar, se desugarea al término $(\mathsf{fun}\ (x{:}\tau).\, e_2)e_1$, y al compilarlo obtenemos:

$$\texttt{FUNCTION}(\mathcal{C}(e_2);\ \texttt{RETURN});\ \mathcal{C}(e_1);\ \texttt{CALL}$$

[4] Un excelente ejemplo de esto es la lista recursiva `let fibs = 0:1:zipWith (+) fibs (tail fibs)`, que computa el $n$-ésimo número de Fibonacci... *¡en tiempo lineal!*

Al ejecutar este fragmento, vamos a agregar una clausura a la pila, con el entorno $e$ inicial, evaluar $e_1$, y luego llamar a la clausura. Esta llamada toma el mismo entorno $e$ de la clausura y salta a él, guardando el entorno actual (también $e$) y un puntero a código (una posición tras el `CALL`) como dirección de retorno. Aquí ejecutamos el código de $e_2$. El `RETURN` luego saltará tras el `CALL`, con el efecto de "olvidar" el primer valor del entorno agregado por `CALL`.

Todo esto es evidentemente ineficiente. 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 haría el `RETURN`).

Para hacer esto, introducimos dos instrucciones `SHIFT` y `DROP`, y las usamos para compilar los lets.

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

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

Implementar este cambio requiere eliminar el desugaring de los let-bindings como aplicaciones, y agregar un nodo de let-binding a la sintaxis interna. La ventaja es que evaluar los lets se vuelve mucho más barato, al evitar la creación de clausuras y los saltos.

# 6. Compilando Programas

Otra parte de PCF que omitimos arriba son las variables libres, 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). 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*:

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

donde sólo nos interesa el resultado de la última definición top-level. Entonces, para compilar un programa $P$ a bytecode, generamos el siguiente código:

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

que en efecto evalua todas las declaraciones top-level en orden, imprime el valor de la última (que debe ser de tipo $\mathbb{N}$, o se levanta un error de compilación) y detiene la máquina.

# 7. Representación Concreta e Implementación

- Representaremos un `Bytecode` simplemente como `[Int]`. Para escribirlo a un archivo, necesitamos un formato binario bien definido. Usaremos una cadena de enteros de longitud fija (usamos 32 bits, formato little endian)[5].
- 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.

[5] Ignoramos problemas de overflow/underflow aquí, mientras estemos lejos de $2^{31}$, no habrá problema.

- 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 `data` 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 `SUCC`, y el tope de la pila no es un `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`, puede solamente usarse `FUNCTION` y agregar una operación `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 en la §4. Esto permite reusar un poco de código, pero implica que (brevemente) 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 persistentes, y sólo existe una pila durante la ejecución. Una ineficiencia inherente de usar listas es que el "saltear" $n$ instrucciones cuesta $O(n)$, pero podremos evitarlo en el próximo capítulo.

# 8. Bytecode en el Mundo Real™

Las máquinas virtuales de pila no son para nada un ejercicio teórico: muchos lenguajes existentes y muy usados 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. Por otro lado, la compilación es usualmente más rápido que si fuera a código nativo.

A saber, la familia Java, los lenguajes .NET, OCaml y Python usan máquinas virtuales de pila. Todas son máquinas virtuales de pila, aunque la JVM también tiene una noción de registros. Tanto la JVM como la CLR son apuntadas por *muchos* lenguajes distintos, economizando la compilación via una máquina común, y permitiendo una interoperación fluida entre distintos lenguajes. 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 *just in time compilation* (JIT). Esencialmente, cuando la máquina está por ejecutar un bloque de código por primera vez, hace una etapa de 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, que se amortiza rápidamente).

# 9. Implementación

## 9.1. Procesando opciones de línea de comandos

En esta implementación del compilador usaremos la línea de comandos. Al correr el compilador con la opción '`stack run -- --bytecompile file.pcf`' deben generar un archivo `file.byte` con el bytecode. Similarmente, al

correr con '`stack run -- --run file.byte`', debe ejecutar el bytecode con la máquina virtual.

Las opciones de línea de comandos pueden ser muchas veces resueltas con un parser simple, pero es conveniente usar una librería ya que esto facilita mantener la documentación sincronizada cuando se hacen cambios. Usaremos la librería `optparse-applicative`, que deberá ser agregada al archivo `package.yaml` del proyecto. Esta librería no está basada en mónadas sino en funtores aplicativos [1], lo que le permite analizar la estructura de los parsers y generar automáticamente a partir del parser una ayuda como la siguiente:

```
    Compilador de PCF de la materia Compiladores 2020

    Usage: compilador [(-t|--typecheck) | (-c|--bytecompile) | (-r|--run) |
                              (-i|--interactive)] [FILES...]
      Compilador de PCF

    Available options:
      -t,--typecheck           Solo chequear tipos
      -c,--bytecompile         Compilar a la BVM
      -r,--run                 Ejecutar bytecode en la BVM
      -i,--interactive         Ejecutar en forma interactiva
      -h,--help                Show this help text
```

Luego de importar el módulo `Options.Applicative`, podemos definir un parser para las banderas:

```
data Mode = Interactive
          | Typecheck
          | Bytecompile
          | Run

-- | Parser de banderas
parseMode :: Parser Mode
parseMode =
    flag' Typecheck ( long "typecheck" <> short 't' <> help "Solo chequear tipos")
<|> flag' Bytecompile (long "bytecompile" <> short 'c' <> help "Compilar a la BVM")
<|> flag' Run (long "run" <> short 'r' <> help "Ejecutar bytecode en la BVM")
<|> flag Interactive Interactive ( long "interactive" <> short 'i'
                                   <> help "Ejecutar en forma interactiva" )

-- | Parser de opciones general, consiste de un modo y una lista de archivos a procesar
parseArgs :: Parser (Mode,[FilePath])
parseArgs = (,) <$> parseMode <*> many (argument str (metavar "FILES..."))
```

Luego podemos usar el parser definiendo el `main` como:

```
main :: IO ()
main = execParser opts >>= go
where
    opts = info (parseArgs <**> helper)
    ( fullDesc
    <> progDesc "Compilador de PCF"
    <> header "Compilador de PCF de la materia Compiladores 2020" )

    go :: (Mode,[FilePath]) -> IO ()
    go (Interactive,files) =
            do runPCF (runInputT defaultSettings (repl files))
               return ()
    go (Typecheck, files) = undefined
    go (Bytecompile, files) = undefined
    go (Run,files) = undefined
```

## 9.2. Serializando estructuras

GHC ya viene con el paquete `binary` preinstalado para serializar y deserializar datos a través del tipo de datos `ByteString`. Primero hay que habilitar los paquetes `binary` y `bytestring` en el archivo `package.yaml` del proyecto. Luego, bajar el archivo `Bytecompile.hs` del repositorio. En el mismo 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 bytecode simplemente como una lista de enteros.

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

Las funciones que restan escribir son:

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

  Compila un término a bytecode.

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

  Usa la función `bc` para compilar un módulo, utilizando la técnica descripta en la sección 6.

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

  Implementa la BVM, 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`.

## 9.3. Tareas

**a)** Implementar la compilación a bytecode y máquina virtual en Haskell, usando el esqueleto provisto. La instrucción `PRINT` debe imprimir a la consola, por lo que la función que ejecuta la máquina debe estar en `MonadPCF`.

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

**c)** Implementar let-bindings internos, y asegurarse de que se compilen de manera eficiente.

# Referencias

[1] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.