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

Apunte

Máquinas Abstractas

Las *máquinas abstractas* son un modelo de computación basado en conjunto de estados, con estados iniciales y finales, y un conjunto de reglas para avanzar de un estado a otro.

Se busca que una máquina abstracta sea la suficientemente concreta como para guiar la implementación en una máquina real, pero lo suficientemente abstracta como para ser independiente de cualquier máquina real. Idealmente, en una máquina abstracta cada paso o avance de un estado a otro debe poder hacerse en $O(1)$.

Vamos a definir diferentes máquinas abstractas para (core) FD4. En estas máquinas los tipos no juegan ningún rol, por lo que los omitiremos. Por lo tanto, los términos de este lenguaje son:

$$t \quad ::= \quad x \mid \mathsf{fun}\ x.t \mid t\ t \mid n \mid \mathsf{print}\ str\ t \mid t + t \mid t - t \mid \mathsf{ifz}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{fix}\ f\ x.t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t$$

y los valores son:

$$v \quad ::= \quad n \mid \mathsf{fun}\ x.t \mid \mathsf{fix}\ f\ x.t$$

Mientras que una semántica operacional estructural es útil para probar propiedades de más alto nivel, como por ejemplo la seguridad del lenguaje, una máquina abstracta es más apropiada para entender cómo se puede ejecutar un lenguaje y estudiar propiedades de bajo nivel como ser el costo computacional de ejecutar un programa. La semántica operacional natural o de paso grande es muy alto nivel. Para obtener una máquina abstracta apropiada, procederemos incrementalmente, desde semánticas de más alto nivel hacia semánticas de más bajo nivel.

1. Semántica Operacional Estructural de Paso Chico

La semántica operacional estructural de paso chico (que abreviaremos como SOS) define el significado de un término a través de pasos de reducción de un término a otro, en vez de una "evaluación" de un término a un valor como en la semántica de paso grande. Representamos la relación de un paso que imprime una cadena str con $\overset{str}{\rightsquigarrow}$, y simplificamos a $\rightsquigarrow$ cuando la cadena es vacía. Tenemos, por ejemplo, que $(\lambda x.x)\, y \rightsquigarrow y$, que $2 + (4 + 1) \rightsquigarrow 2 + 5$ y que $(\mathsf{print}\ \text{“}x = \text{”}\ 3) \overset{\text{“}x=3\text{”}}{\rightsquigarrow} 3$.

Definiremos esta relación mediante las reglas de inferencia a continuación. Las reglas se dividen en dos tipos: las reglas de *computación*, que efectuan los pasos computacionales, y las reglas de *congruencia*, que permiten aplicar las reglas de computación estructuralmente dentro de un término.

Reglas de computación:

$$(\mathsf{fun}\ x.t)\ v \rightsquigarrow [v/x]t \qquad \mathsf{ifz}\ 0\ \mathsf{then}\ t\ \mathsf{else}\ t' \rightsquigarrow t \qquad \mathsf{print}\ s\ v \overset{s \cdot v}{\rightsquigarrow} v$$

$$(\mathsf{fix}\ f\ x.t)\ v \rightsquigarrow [\mathsf{fix}\ f\ x.t/f, v/x]t$$

$$\mathsf{ifz}\ np\ \mathsf{then}\ t\ \mathsf{else}\ t' \rightsquigarrow t' \qquad n \oplus m \rightsquigarrow \underline{n \oplus m}$$

donde $\oplus$ es uno de los operadores binarios de FD4 ($+$ o $-$), y las operaciones subrayadas realizan la operación matemática correspondiente (suma o resta de naturales, respectivamente). Con la metavariable np indicamos un natural positivo (mayor a 0).

Reglas de congruencia:

$$\frac{t \overset{s}{\rightsquigarrow} t'}{t\ u \overset{s}{\rightsquigarrow} t'\ u} \qquad \frac{t \overset{s}{\rightsquigarrow} t'}{\mathsf{print}\ s'\ t \overset{s}{\rightsquigarrow} \mathsf{print}\ s'\ t'}$$

$$\frac{u \overset{s}{\rightsquigarrow} u'}{v\ u \overset{s}{\rightsquigarrow} v\ u'} \qquad \frac{t \overset{s}{\rightsquigarrow} t'}{t \oplus u \overset{s}{\rightsquigarrow} t' \oplus u}$$

$$\frac{t \overset{s}{\rightsquigarrow} t'}{\mathsf{ifz}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \overset{s}{\rightsquigarrow} \mathsf{ifz}\ t'\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2} \qquad \frac{u \overset{s}{\rightsquigarrow} u'}{v \oplus u \overset{s}{\rightsquigarrow} v \oplus u'}$$

A pesar de ser una semántica de más bajo nivel que la de paso grande, sigue siendo demasiado abstracta. Consideremos el siguiente paso de evaluación:

$$\frac{\dfrac{\dfrac{6-1 \rightsquigarrow 5}{\mathsf{print}\ ``"\ (6-1) \rightsquigarrow \mathsf{print}\ ``"\ 5}}{4-(\mathsf{print}\ ``"\ (6-1)) \rightsquigarrow 4-(\mathsf{print}\ ``"\ 5)}}{20+(4-(\mathsf{print}\ ``"\ (6-1))) \rightsquigarrow 20+(4-(\mathsf{print}\ ``"\ 5))}$$

¡En un paso, pasan demasiadas cosas! Primero debemos usar varias reglas de congruencia hasta encontrar una regla de computación y luego aplicar dicho regla de computación. Además, en el paso siguiente, tendremos que repetir esta búsqueda para encontrar la próxima expresión a reducir. Finalmente, al evaluar un programa, no queremos realmente crear de manera concreta los términos intermedios (como $4-(\mathsf{print}\ ``"\ (6-1))$ más arriba) ni hacer substituciones. Primero, haremos una pequeña reformulación de la relación $\rightsquigarrow$.

Ej. 1. En las reglas de paso chico mostradas más arribas faltan las reglas de computación y congruencia de los términos let. Completarlas.

2. Contextos de Evaluación

Felleisen y Hieb [3], introdujeron una forma más concisa de expresar la SOS. La idea es dejar sólo los pasos de computación (que vamos a denotar con $\rightsquigarrow_c$) y especificar dónde se realiza la computación mediante *contextos de evaluación*. En general, un contexto C es un programa con un agujero $[\,]$. Escribimos $C[t]$ para denotar el término que se obtiene al llenar el agujero del contexto C con un término t. Un contexto *de evaluación* es un programa con un agujero *en el lugar donde se va a realizar la próxima computación.* Las formas de construir un contexto están en biyección con las reglas de congruencia de la SOS. En core FD4, los contextos de evaluación son:

E	::=	$\square$	Un agujero: aquí ocurre la reducción.
	|	$E\ t$	Se debe evaluar la función.
	|	$v\ E$	Se debe evaluar el argumento (una vez evaluada la función).
	|	$\mathsf{ifz}\ E\ \mathsf{then}\ t\ \mathsf{else}\ e$	Se debe evaluar la condición.
	|	$E \oplus t$	Se debe evaluar el primer argumento.
	|	$v \oplus E$	Se debe evaluar el segundo argumento.
	|	$\mathsf{print}\ E$	Se debe evaluar el argumento.

Al "codificar" las reglas de congruencia, tenemos que $E[t] \overset{s}{\rightsquigarrow} E[t']$ siempre que $t \overset{s}{\rightsquigarrow} t'$. A su vez, dados términos r y r' tal que $r \overset{s}{\rightsquigarrow} r'$ podemos expresar a r y r' *en forma única* como $r = E[t]$ y $r' = E[t']$ tal que $t \overset{s}{\rightsquigarrow}_c t'$.

Por ejemplo, el término

$$2 + (\mathsf{ifz}\ ((\mathsf{fun}\ x.\, x)\ (5 - 1))\ \mathsf{then}\ 1\ \mathsf{else}\ 2)$$

se puede descomponer en el contexto $C[\bullet] = 2 + (\mathsf{ifz}\ ((\mathsf{fun}\ x.\, x)\ \bullet)\ \mathsf{then}\ 1\ \mathsf{else}\ 2)$ y el término $(5 - 1)$.

En la factorización de un término en su contexto de evaluación y su redex, está codificada la búsqueda del redex dentro del término que hacíamos con reglas de congruencia en la SOS. En la siguiente máquina haremos explícita esta búsqueda.

Ej. 2. En los contextos de evaluación E de más arriba faltan los casos correspondientes a los términos let. Completar la definición de E.

3. La máquina CK

La máquina CK es una visión más concreta de la semántica operacional $\rightsquigarrow$, ya que modela las aplicaciones de reglas de congruencia, que viene a ser la búsqueda del siguiente redex, como pasos de la máquina misma. Más aún, logra *reusar* esta búsqueda entre cada paso computacional al computar el contexto de evaluación de forma incremental entre paso y paso.

Esta máquina obtiene su nombre porque sus estados estan compuestos de un "control" o "código" C y una "continuación" K. El código representa el término que está en posición de evaluación, mientras que la continuación representa al contexto de evaluación dentro del cual ocurre el código.

Si miramos en detalle los contextos de evaluación veremos que son una lista que finaliza en un agujero. Si invertimos esa lista podemos representar a los contextos de evaluación como una pila de marcos o *frames*, donde los marcos son:

$$fr ::= \square\ t \mid v\ \square \mid \mathsf{ifz}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ e \mid \square \oplus t \mid v \oplus\ \square \mid \mathsf{print}\ str\ \square$$

y la pila de marcos la representamos como:

$$K ::= \epsilon \mid fr \succ K$$

En particular, el contexto $C[\bullet] = 2 + (\mathsf{ifz}\ ((\mathsf{fun}\ x.\, x)\ \bullet)\ \mathsf{then}\ 1\ \mathsf{else}\ 2)$ queda representado por la pila:

$$(\mathsf{fun}\ x.\, x)\ \square \succ \mathsf{ifz}\ \square\ \mathsf{then}\ 1\ \mathsf{else}\ 2 \succ 2 + \square \succ \epsilon$$

ya que aplicando los marcos, en orden, al agujero $\bullet$ podemos reconstruir C.

Ej. 3. Extender la definición de los marcos con los correspondientes a los contextos de términos let.

Definimos la máquina CK: sus estados consisten de pares de términos a ejecutar (el control C de la computación), y continuaciones en K. Diferenciamos entre dos tipos de estado:

- un estado $\langle t,\ k\rangle$ que consiste de un término t y una continuación k.
- un estado $\langle\!\langle v,\ k\rangle\!\rangle$ que consiste de un valor v y una continuación k.

Definimos la máquina en dos fases, según el tipo estado que transforma: una fase de búsqueda y otra de reducción. En la primer fase, toma un estado $\langle t,\ k\rangle$ y, analizando el término t va construyendo la continuación, hasta encontrar un valor. Al llegar a un valor, pasamos a la fase de reducción.

$$
\begin{array}{rrcll}
\langle & \mathsf{print}\ s\ t, & k\rangle & \to \langle t, & \mathsf{print}\ s\ \square \succ k\rangle \\
\langle & t \oplus u, & k\rangle & \to \langle t, & \square \oplus u \succ k\rangle \\
\langle & \mathsf{ifz}\ c\ \mathsf{then}\ t\ \mathsf{else}\ e, & k\rangle & \to \langle c, & \mathsf{ifz}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ e \succ k\rangle \\
\langle & t\ u, & k\rangle & \to \langle t, & \square\ u \succ k\rangle \\
\langle & v, & k\rangle & \to \langle\langle v, & k\rangle\rangle
\end{array}
$$

En la segunda fase, toma un estado $\langle\langle v,\ k\rangle\rangle$ y dependiendo de la continuación, opera sobre este valor.

$$
\begin{array}{rrcll}
\langle\langle\ n, & \mathsf{print}\ s\ \square \succ k\rangle\rangle & \overset{s\cdot n}{\to} \langle\langle & n, & k\rangle\rangle \\
\langle\langle\ n, & \square \oplus u \succ k\rangle\rangle & \to \langle & u, & n \oplus \square \succ k\rangle \\
\langle\langle\ n', & n \oplus \square \succ k\rangle\rangle & \to \langle\langle & \underline{n \oplus n'}, & k\rangle\rangle \\
\langle\langle\ 0, & \mathsf{ifz}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ e \succ k\rangle\rangle & \to \langle & t, & k\rangle \\
\langle\langle np, & \mathsf{ifz}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ e \succ k\rangle\rangle & \to \langle & e, & k\rangle \\
\langle\langle\ v, & \square\ u \succ k\rangle\rangle & \to \langle & u, & v\ \square \succ k\rangle \\
\langle\langle\ v, & (\mathsf{fun}\ x.\, t)\ \square \succ k\rangle\rangle & \to \langle & [v/x]t, & k\rangle \\
\langle\langle\ v, & (\mathsf{fix}\ f\ x.\, t)\ \square \succ k\rangle\rangle & \to \langle & [\mathsf{fix}\ f\ x.\, t/f, v/x]t, & k\rangle
\end{array}
$$

Intuitivamente, las reglas de la primer fase van agregando un marco a la continuación y descendiendo por el término a ejecutar. Las reglas de la segunda fase, donde dependiendo del marco en el que se encuentra el valor se decide como seguir.

Para ejecutar la máquina, formamos un estado inicial a partir de un término t y una continuación vacía: $\langle t,\ \epsilon\rangle$. Los estados finales son de la forma $\langle\langle v,\ \epsilon\rangle\rangle$: valores con la continuación vacía.

A diferencia de la SOS, cuando se calcula una regla de computación no se vuelve a empezar desde el término completo, sino que se continúa desde el lugar en el que se estaba parado ya que la continuación sigue estando presente. Para dar un ejemplo concreto, si tenemos $(M\ N) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$, la máquina descenderá agregando varios $\square + 1$ a la continuación hasta llegar a $M\ N$. Luego se evaluará M, y pasará a evaluarse N, sin tener que volver a recorrer los $\square + 1$ acumulados en la continuación (hasta que eso sea necesario).

La máquina CK es de más bajo nivel que la semántica operacional de paso chico y además, como implementación, es más eficiente. Cada uno de sus pasos es suficientemente chico como para que pueda ser considerado una operación en una máquina real, excepto por una operación: la sustitución. La sustitución es una operación costosa que es al menos $O(n)$ en el tamaño del término.[1] Para eliminar la sustitución vamos a introducir una nueva máquina.

Ejemplo de ejecución

Tomamos $t = \mathsf{ifz}\ (1 - 2)\ \mathsf{then}\ (\mathsf{fun}\ x.\, x + 4)\ 1\ \mathsf{else}\ 3 + 20$.

[1] La sustitución podría ser cuadrática si hay que hacer renombre de variables

$$
\begin{array}{ll}
 & \langle \mathsf{ifz}\ 1-2\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20,\ \epsilon\rangle \\
\to & \langle 1-2,\ \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon\rangle \\
\to & \langle 1,\ \Box-2 \succ \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon\rangle \\
\to & \langle\!\langle 1,\ \Box-2 \succ \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon\rangle\!\rangle \\
\to & \langle 2,\ 1-\Box \succ \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon\rangle \\
\to & \langle\!\langle 2,\ 1-\Box \succ \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon\rangle\!\rangle \\
\to & \langle\!\langle 0,\ \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon\rangle\!\rangle \\
\to & \langle (\mathsf{fun}\ x.\, x+4)\ 1,\ \epsilon\rangle \\
\to & \langle (\mathsf{fun}\ x.\, x+4),\ \Box\ 1 \succ \epsilon\rangle \\
\to & \langle\!\langle (\mathsf{fun}\ x.\, x+4),\ \Box\ 1 \succ \epsilon\rangle\!\rangle \\
\to & \langle 1,\ (\mathsf{fun}\ x.\, x+4)\ \Box \succ \epsilon\rangle \\
\to & \langle\!\langle 1,\ (\mathsf{fun}\ x.\, x+4)\ \Box \succ \epsilon\rangle\!\rangle \\
\to & \langle 1+4,\ \epsilon\rangle \\
\to & \langle 1,\ \Box+4 \succ \epsilon\rangle \\
\to & \langle\!\langle 1,\ \Box+4 \succ \epsilon\rangle\!\rangle \\
\to & \langle 4,\ 1+\Box \succ \epsilon\rangle \\
\to & \langle\!\langle 4,\ 1+\Box \succ \epsilon\rangle\!\rangle \\
\to & \langle\!\langle 5,\ \epsilon\rangle\!\rangle
\end{array}
$$

Ej. 4. Completar la definición de la máquina CK con los casos de los términos let.

4. La máquina CEK

Para solucionar los problemas de la máquina CK, definiremos la máquina CEK [2, 4]. Evitamos hacer sustituciones mediante un entorno ρ que asocia variables a valores. Como es usual anotamos com $\rho(x)$ al valor de x en el entorno ρ, y con $\rho(x \mapsto v)$ al entorno que se comporta como ρ excepto en x donde tiene valor v. El entorno $\varnothing$ es el entorno vacío. Teniendo un entorno, no hace falta hacer una sustitución $[v/x]t$. En su lugar, asociamos en el entorno x a v, y luego computamos con t. Al encontrar una variable x, simplemente buscamos su valor v en el entorno. Esencialmente, vamos a hacer las sustituciones incrementalmente a medida que se necesiten. Si bien esta idea funciona, tenemos que tener ciertos cuidados:

Una variable no debe escapar a su entorno. Al extender un entorno $\rho(x \mapsto v)$, x sólo debe estar asociada a v dentro del alcance de la variable x. Fuera de su alcance hay que restaurar el entorno original ρ. De no hacerlo así, podría ocurrir que se asocie el valor equivado a una variable. Considere el término

$$(\mathsf{fun}\ x.\, \mathsf{ifz}\ ((\mathsf{fun}\ x.\, x)\ 2)\ \mathsf{then}\ 7\ \mathsf{else}\ x)\ 3$$

Si al ejecutar el subtérmino $(\mathsf{fun}\ x.\, x)\ 2$, x sigue asociado a 2, el resultado de la ejecución será 2, en lugar de 3, ya que al evaluar la variable x en el else predominará la última adición de x al entorno. Por lo tanto debemos guardar el entorno antes de ampliarlo y restaurarlo cuando termine el alcance de la computación actual.

Para poder restaurar el entorno modificamos los marcos para que permitan almacenar un entorno (y luego restaurarlo).

Hay que preservar el entorno de las variables que quedan libres. Considere el siguiente término:

$$(\mathsf{fun}\ (x : \mathbb{N}).\, (\mathsf{fun}\ (f : \mathbb{N} \to \mathbb{N}).\, f\ 20)\ (\mathit{suma}\ x))\ 10$$

Al ejecutarlo, esperamos que nos devuelva algo equivalente a *suma* 10 20. Como paso intermedio, llegaremos al valor *suma* x, y lo pasaremos como argumento a la abstracción con argumento f. Para que esto tenga

sentido, debemos recordar que en ese argumento que pasamos, x vale 10. La solución está en el concepto de *clausura*.

Una clausura (en inglés *closure*) es un término junto con un entorno que da significado a las variables libres en ese término, efectivamente *cerrando* el término [5]. Las clausuras son un concepto extremadamente útil que aparecen en la compilación de varios lenguajes como ser lenguajes de alto orden, lenguajes que permiten funciones anidadas, y lenguajes orientados a objetos.

Redefinimos los valores del lenguaje para usar clausuras:

$$\begin{array}{lcl} v & ::= & n \mid clos \\ clos & ::= & \mathsf{clos_{fun}}(\rho, x, t) \mid \mathsf{clos_{fix}}(\rho, f, x, t) \end{array}$$

En el caso particular de nuestro lenguaje distinguimos entre dos tipos de clausuras, ya que tenemos dos construcciones de valores que pueden tener variables libres. Por ejemplo, la clausura $\mathsf{clos_{fun}}(\rho, x, t)$ representa al término $(\mathsf{fun}\ x.\, .t)$ evaluado en el entorno ρ.

Durante la evaluación, la máquina llevará un entorno que acumula los valores de cada variable libre, que crece a medida que se hacen β-reducciones. Por ello, deberemos guardar un entorno en algunos frames: por ejemplo, en el frame $\Box\ t$, que representa que debemos luego evaluar t porque aparece como argumento, guardaremos el entorno en donde llegamos a t para poder evaluarlo correctamente. Lo mismo ocurre para $\mathsf{ifz}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ e$, y en general ocurriría para todo frame que contenga un término posiblemente abierto.

En base a la discusión anterior los marcos quedan:

$$\begin{array}{lcl} fr & ::= & \rho \cdot \Box\ t \\ & \mid & clos\ \Box \\ & \mid & \rho \cdot \mathsf{ifz}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ e \\ & \mid & \rho \cdot \Box \oplus t \\ & \mid & v \oplus \Box \\ & \mid & \mathsf{print}\ str\ \Box \end{array}$$

Ej. 5. Extienda la definición de los marcos con los correspondientes a los términos let. ¿Cambia algo con respecto a la extensión realizada en el ejercicio 3?

Los estados de la máquina CEK tendrán tres componentes: el control C, un entorno E, y una continuación K. Al igual que en la máquina CK, la separamos la máquina CEK en dos fases, según el control sea un término o un valor. Cuando el control es un término tenemos un estado $\langle t, \rho, k\rangle$, donde ahora el entorno ρ explica las variables libres en t. Cuando el control es un valor tenemos un estado $\langle\langle v, k\rangle\rangle$. En este caso no necesitamos un entorno ya que los valores no contienen variables libres (porque las funciones fueron cerradas dentro de una clausura).

La reglas de la fase de búsqueda son:

$$\begin{array}{lcllr}
\langle \mathsf{print}\ s\ t & ,\rho,k\rangle & \to & \langle t & ,\rho, \quad \mathsf{print}\ s\ \Box \succ k\rangle \\
\langle t \oplus u & ,\rho,k\rangle & \to & \langle t & ,\rho, \quad \rho \cdot \Box \oplus u \succ k\rangle \\
\langle \mathsf{ifz}\ c\ \mathsf{then}\ t\ \mathsf{else}\ e & ,\rho,k\rangle & \to & \langle c & ,\rho, \quad \rho \cdot \mathsf{ifz}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ e \succ k\rangle \\
\langle t\ u & ,\rho,k\rangle & \to & \langle t & ,\rho, \quad \rho \cdot \Box\ u \succ k\rangle \\
\langle x & ,\rho,k\rangle & \to & \langle\langle \rho(x) & , \quad k\rangle\rangle \\
\langle n & ,\rho,k\rangle & \to & \langle\langle n & , \quad k\rangle\rangle \\
\langle \mathsf{fun}\ x.t & ,\rho,k\rangle & \to & \langle\langle \mathsf{clos_{fun}}(\rho, x, t) & , \quad k\rangle\rangle \\
\langle \mathsf{fix}\ f\ x.t & ,\rho,k\rangle & \to & \langle\langle \mathsf{clos_{fix}}(\rho, f, x, t), & \quad k\rangle\rangle
\end{array}$$

Las reglas de las fase de reducción son:

$$
\begin{array}{llcll}
\langle\!\langle v & , & \mathsf{print}\ s\ \Box \succ k \rangle\!\rangle & \overset{s\cdot v}{\to} & \langle\!\langle v & , & k \rangle\!\rangle \\
\langle\!\langle n & , & \rho \cdot \Box \oplus u \succ k \rangle\!\rangle & \to & \langle u & , \rho, & n \oplus \Box \succ k \rangle \\
\langle\!\langle n' & , & n \oplus \Box \succ k \rangle\!\rangle & \to & \langle\!\langle \underline{n \oplus n'}, & & k \rangle\!\rangle \\
\langle\!\langle 0 & , \rho \cdot \mathsf{ifz}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ e \succ k \rangle\!\rangle & & \to & \langle t & , \rho, & k \rangle \\
\langle\!\langle np & , \rho \cdot \mathsf{ifz}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ e \succ k \rangle\!\rangle & & \to & \langle e & , \rho, & k \rangle \\
\langle\!\langle clos, & & \rho \cdot \Box\ t \succ k \rangle\!\rangle & \to & \langle t & , \rho, & clos\ \Box \succ k \rangle \\
\langle\!\langle v & , & \mathsf{clos}_{\mathsf{fun}}(\rho, x, t)\ \Box \succ k \rangle\!\rangle & \to & \langle t & , \rho(x \mapsto v), & k \rangle \\
\langle\!\langle v & , & (\mathsf{clos}_{\mathsf{fix}}(\rho, f, x, t)\ \Box \succ k \rangle\!\rangle & \to & \langle t & , \rho(f \mapsto \mathsf{clos}_{\mathsf{fix}}(\rho, f, x, t), x \mapsto v), & k \rangle
\end{array}
$$

Una explicación intuitiva de las reglas es la siguiente:

Fase de búsqueda: las cuatro primeras reglas son análogas a las correspondientes en la máquina CK, excepto que la regla de los operadores binarios, la de ifz y la de la aplicación guardan un entorno en la continuación. Luego, aparece una regla nueva que cuando encuentra una variable, continua la evaluación con el valor de la misma en el entorno. Finalmente las últimas tres reglas lidian con términos ya evaluados en forma similar a la máquina CK, excepto que en el caso de funciones y puntos fijos se debe construir una clausura para obtener un valor.

Fase de reducción: Las tres primeras reglas son análogas a las correspondientes en la máquina CK. Las dos reglas siguientes lidian con la continuación ifz, restaurando el entorno guardado en la continuación en la fase de búsqueda. La siguiente regla es el caso de una aplicación en que la función ya se evaluó (y se obtuvo una clausura) y por lo tanto la ejecución debe continuar con el argumento, restaurando el entorno que se encontraba en la continuación. Las últimas dos reglas corresponden al caso en el que el argumento ya fue evaluado y por lo tanto hay que aplicar la función. Esto se logra continuando con el cuerpo de la función o punto fijo en un entorno que asocia el valor calculado al argumento. En el caso del punto fijo también asocia el nombre de la función recursiva a la clausura.

Ejemplo de ejecución

Tomamos de nuevo $t = \mathsf{ifz}\ (1-2)\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20$.

$$
\begin{array}{ll}
 & \langle \mathsf{ifz}\ (1-2)\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20,\ \varnothing,\ \epsilon \rangle \\
\to & \langle 1-2,\ \varnothing,\ \varnothing \cdot \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon \rangle \\
\to & \langle 1,\ \varnothing,\ \varnothing \cdot \Box - 2 \succ \varnothing \cdot \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon \rangle \\
\to & \langle\!\langle 1,\ \varnothing \cdot \Box - 2 \succ \varnothing \cdot \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon \rangle\!\rangle \\
\to & \langle 2,\ \varnothing,\ 1 - \Box \succ \varnothing \cdot \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon \rangle \\
\to & \langle\!\langle 2,\ 1 - \Box \succ \varnothing \cdot \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon \rangle\!\rangle \\
\to & \langle\!\langle 0,\ \varnothing \cdot \mathsf{ifz}\ \Box\ \mathsf{then}\ (\mathsf{fun}\ x.\, x+4)\ 1\ \mathsf{else}\ 3+20 \succ \epsilon \rangle\!\rangle \\
\to & \langle (\mathsf{fun}\ x.\, x+4)\ 1,\ \varnothing,\ \epsilon \rangle \\
\to & \langle (\mathsf{fun}\ x.\, x+4),\ \varnothing,\ \varnothing \cdot \Box\ 1 \succ \epsilon \rangle
\end{array}
$$

Hasta aquí, la ejecución fue muy similar a la máquina CK, salvo por llevar el entorno $\varnothing$ en el marco del ifz y del operador binario de resta. Estos entornos se guardaron en los marcos y luego se restauraron cuando pasamos de la fase de reducción a la fase de búsqueda. Aquí llegamos al primer paso interesante: al evaluar una abstracción, pasamos a la fase de reducción con una clausura, que contiene al entorno actual (aunque en este caso siga vacío). El paso subsiguiente, comienza a evaluar el argumento (1), restaurando el entorno en el cual apareció.

$\rightarrow \quad \langle\!\langle \mathsf{clos}_{\mathsf{fun}}(\varnothing, x, x+4),\ \varnothing \cdot \Box\ 1 \succ \epsilon \rangle\!\rangle$
$\rightarrow \quad \langle 1,\ \varnothing,\ \mathsf{clos}_{\mathsf{fun}}(\varnothing, x, x+4)\ \Box \succ \epsilon \rangle$
$\rightarrow \quad \langle\!\langle 1,\ \mathsf{clos}_{\mathsf{fun}}(\varnothing, x, x+4)\ \Box \succ \epsilon \rangle\!\rangle$
$\rightarrow \quad \langle x+4,\ \varnothing(x \mapsto 1),\ \epsilon \rangle$
$\rightarrow \quad \langle x,\ \varnothing(x \mapsto 1),\ \varnothing(x \mapsto 1) \cdot \Box + 4 \succ \epsilon \rangle$
$\rightarrow \quad \langle\!\langle 1,\ \varnothing(x \mapsto 1) \cdot \Box + 4 \succ \epsilon \rangle\!\rangle$
$\rightarrow \quad \langle 4,\ \varnothing(x \mapsto 1),\ 1 + \Box \succ \epsilon \rangle$
$\rightarrow \quad \langle\!\langle 4,\ 1 + \Box \succ \epsilon \rangle\!\rangle$
$\rightarrow \quad \langle\!\langle 5,\ \epsilon \rangle\!\rangle$

Ej. 6. Completar la definición de la máquina CEK con los casos de los términos let.

5. Implementación

Tarea Implementar en un nuevo módulo `CEK` la máquina CEK y modificar el `Main` para que ejecute los programas usando esta máquina cuando se le pasa la opción `--interactiveCEK` o `-k`.

Trabajando con índices Como siempre, trabajar con nombres es poco conveniente, así que usaremos índices de de Bruijn. En este caso, evitaremos incluso los nombres locales, y manipularemos solamente índices. La idea principal es que los entornos son una lista de valores, en donde la posición i en la lista nos da el valor de la variable i. Para implementar la regla que consume una clausura, y extiende el entorno, basta con agregar el valor v en la cabeza de la lista, lo que hará que la variable 0 (i.e. x) apunte a v, y las variables de ρ también estén en la posición correcta, ya que hemos descendido exactamente un binder.

Sugerencias concretas Sugerimos crear un tipo de datos dedicado para los valores, con tres constructores como se especifica arriba en la gramática para v. Los entornos son simplemente una lista de valores, interpretados según el párrafo anterior. Los frames pueden definirse con un tipo algebraico, por ejemplo con un caso `... | KArg Env TTerm | ...` para representar $\rho \cdot \Box\ t$. Las continuaciones `Kont` son una lista de frames. Notar la recursión mutua entre los valores y los entornos debido a las clausuras. La máquina en sí puede implementarse con dos funciones mutuamente recursivas:

```
seek    :: MonadFD4 m => TTerm -> Env -> Kont -> m Val
destroy :: MonadFD4 m => Val          -> Kont -> m Val
```

La única razón por la cual necesitamos `MonadFD4` es para buscar el valor de variables globales que aparezcan en el término. En esos casos, simplemente se expanden y se continúa la evaluación.

Impresión La máquina CEK calcula un valor en donde las funciones se expresan como clausuras. Por lo tanto para *mostrar* el resultado (y tal vez para compararlo con el evaluador dado) es conveniente transformar estos valores en términos.

¿Listas o arrays? Las listas en Haskell son listas simplemente enlazadas, en las cuales el costo de acceder a un índice i es $O(i)$. Si usáramos arrays para representar los entornos, el costo de acceso a una variable se disminuiría a $O(1)$, pero el problema es que debemos *copiar* el array cada vez que lo extendemos, mientras que las listas enlazadas pueden tomar ventaja del *sharing* para implementar la extensión en $O(1)$. Si bien no es obvio cuál opción es mejor, Accattoli [1] demuestra que las listas son estrictamente más eficientes (aunque para una máquina diferente).

Referencias

[1] Beniamino Accattoli. The complexity of abstract machines. *Electronic Proceedings in Theoretical Computer Science*, 235:1–15, 01 2017.

[2] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, pages 193–222. North-Holland, 1987.

[3] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235 – 271, 1992.

[4] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993.

[5] Peter J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 01 1964.