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

Apunte

El Lenguaje FD4

En este apunte veremos el lenguaje FD4 que utilizaremos como núcleo del compilador, que como veremos es muy cercano al teórico visto en el apunte anterior.

1. Sintaxis de FD4

Mantendremos la sintaxis de los tipos igual que en PCF.

Los tipos son naturales o funciones entre tipos.

$$\tau ::= \mathbb{N} \mid \tau \to \tau$$

El constructor de tipos $\to$ asocia a derecha. Por ejemplo:

$$\tau \to \sigma \to \gamma \quad \equiv \quad \tau \to (\sigma \to \gamma)$$

La sintaxis es la de un λ-cálculo con naturales y un operador de punto fijo que nos da recursión general.

t	::=	x	variables
	|	$\mathsf{fun}(x : \tau).\, t$	abstracción (λ)
	|	$t\ t$	aplicación
	|	n	constantes naturales (0,1,2,...)
	|	$t\ +\ t$	suma
	|	$t\ -\ t$	resta
	|	$\mathsf{ifz}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$	condicional (compara con 0)
	|	$\mathsf{fix}(f : \tau)(x : \tau).\, t$	punto fijo
	|	$\mathsf{let}\ (x : \tau) = t\ \mathsf{in}\ t$	operador *let* de declaración de variables locales
	|	$\mathsf{print}\ msg\ t$	impresión de mensaje y valor de un término

El operador $\mathsf{let}\ \_ = \_\ \mathsf{in}\ \_$ no es recursivo, es decir, dados términos p y q, un tipo τ, y una variable x, el término $\mathsf{let}\ (x\colon \tau) = p\ \mathsf{in}\ q$ declara una variable $x\colon \tau$ asociada al término p en el entorno de evaluación de q, pero el término p es evaluado con el entorno de evaluación previo al declarar x. Por ejemplo, en el término

$$\mathsf{let}\ (x\colon \mathbb{N}) = 5\ \mathsf{in}\ (\mathsf{let}\ (x\colon \mathbb{N}) = x + 1\ \mathsf{in}\ x + 10)$$

la ocurrencia de la variable x en la expresión $x + 1$ está ligada al primer let. Es decir que al momento de evaluarse esta ocurrencia de x toma el valor 5, mientras que en la expresión $x + 10$, la variable x debería ligarse al segundo let. Por lo tanto el valor del término debe ser 16. ¿Qué ocurriría con este término si los let fueran recursivos?

El operador de impresión de mensajes, print, va a introducir cambios sustanciales tanto en la definición del lenguaje como en la definición de su semántica. Formalmente, para cada cadena de caracteres msg, la construcción $\mathsf{print}\ msg$ es un operador del lenguaje. Esto nos evita tener que introducir cadenas de texto como elemento del lenguaje. Consecuentemente, podemos pensar al operador print como una familia de operadores. Como partimos de un alfabeto finito —los caracteres alfanuméricos— el conjunto de cadenas

de caracteres es numerable, y por ende, se introducen una cantidad numerable de operadores al lenguaje. Notar que con los naturales y con los operadores que utilizan variables sucede lo mismo.

Al igual que en PCF, la aplicación asocia a izquierda.

$$f\ x\ y \quad \equiv \quad (f\ x)\ y$$

2. Reglas de Tipado

Las reglas de tipado de FD4 son similares a las de PCF, con el agregado de la regla de print.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ (T-Var)} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{print}\ msg\ t : \mathbb{N}} \text{ (T-Print)}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \mathsf{fun}(x : \tau).\, t : \tau \to \sigma} \text{ (T-Abs)} \qquad \Gamma \vdash n : \mathbb{N} \text{ (T-Nat)}$$

$$\frac{\Gamma \vdash t : \tau \to \sigma \qquad \Gamma \vdash p : \tau}{\Gamma \vdash t\ p : \sigma} \text{ (T-App)} \qquad \frac{\Gamma \vdash p : \mathbb{N} \qquad \Gamma \vdash q : \mathbb{N}}{\Gamma \vdash p + q : \mathbb{N}} \text{ (T-Add)}$$

$$\frac{\Gamma, f : (\tau \to \sigma), x : \tau \vdash t : \sigma}{\Gamma \vdash \mathsf{fix}(f : \tau \to \sigma)(x : \tau).\, t : \tau \to \sigma} \text{ (T-Fix)} \qquad \frac{\Gamma \vdash p : \mathbb{N} \qquad \Gamma \vdash q : \mathbb{N}}{\Gamma \vdash p - q : \mathbb{N}} \text{ (T-Subt)}$$

$$\frac{\Gamma \vdash p : \tau \qquad \Gamma, x : \tau \vdash q : \sigma}{\Gamma \vdash \mathsf{let}\ (x : \tau) = p\ \mathsf{in}\ q : \sigma} \text{ (T-Let)} \qquad \frac{\Gamma \vdash c : \mathbb{N} \quad \Gamma \vdash t : \tau \quad \Gamma \vdash p : \tau}{\Gamma \vdash \mathsf{ifz}\ c\ \mathsf{then}\ p\ \mathsf{else}\ q : \tau} \text{ (T-Ifz)}$$

3. Semántica Operacional Big-Step

Al igual que con el lenguaje PCF le daremos a FD4 una semántica CBV, donde los valores son los números naturales, las abstracciones, y los puntos fijos sin aplicar:

$$v ::= n \mid \mathsf{fun}(x : \tau).\, t \mid \mathsf{fix}(f : \tau)(x : \tau).\, t$$

En FD4, ya que introducimos un nuevo operador (print) que nos introduce un *efecto* adicional sobre la evaluación de términos, sumaremos una lista de cadenas de texto que se van ir coleccionando a medida que progrese la evaluación de los términos. Es un cambio sutil, pero además introduce que tengamos que tomar una decisión concreta: el orden en que queremos que los mensajes se coleccionen. Por ejemplo, si tenemos la aplicación de una función a su argumento, $(f\ x)$, ¿cuales mensajes se imprimen primero, los de f o los de x?.

A continuación se introduce una semántica big-step para términos cerrados, donde además asumimos que los mensajes se coleccionan de izquierda a derecha. En el ejemplo anterior, primero los generados por la evaluación del término f para obtener una función, luego los generados por la evaluación de x para la obtención del argumento, y finalmente, los generados por la evaluación del cuerpo de la función.

Definimos entonces una relación de evaluación big-step donde para un término cerrado t, una lista de cadenas os, y un valor v, $t \Downarrow^{os} v$, indica que el término t evalúa al valor v con mensajes de salida os.

$$\mathsf{fun}(x:\tau).\,t \Downarrow^{\varnothing} \mathsf{fun}(x:\tau).\,t \qquad \text{(E-Abs)}$$

$$\mathsf{fix}(f:\tau)(x:\sigma).\,t \Downarrow^{\varnothing} \mathsf{fix}(f:\tau)(x:\sigma).\,t \qquad \text{(E-Fix)}$$

$$\frac{t \Downarrow^{ot} \mathsf{fun}(x:\tau).\,t' \quad u \Downarrow^{ou} v' \quad [v'/x]t' \Downarrow^{os} v}{t\ u \Downarrow^{ot+ou+os} v} \qquad \text{(E-App)}$$

$$\frac{t \Downarrow^{ot} v' \quad v' = \mathsf{fix}(f:\tau)(x:\sigma).\,s \quad u \Downarrow^{ou} v'' \quad [v'/f, v''/x]s \Downarrow^{os} v}{t\ u \Downarrow^{ot+ou+os} v} \qquad \text{(E-AppFix)}$$

$$\frac{t \Downarrow^{ot} w \quad [w/x]\,p \Downarrow^{op} v}{\mathsf{let}\ (x:\tau) = t\ \mathsf{in}\ p \Downarrow^{ot+op} v} \qquad \text{(E-Let)}$$

$$n \Downarrow^{\varnothing} n \qquad \text{(E-Nat)}$$

$$\frac{p \Downarrow^{op} n \quad q \Downarrow^{oq} m}{p - q \Downarrow^{op+oq} n \dot{-} m} \qquad \text{(E-Subt)}$$

$$\frac{p \Downarrow^{op} n \quad q \Downarrow^{oq} m}{p + q \Downarrow^{op+oq} n + m} \qquad \text{(E-Add)}$$

$$\frac{p \Downarrow^{op} 0 \quad q \Downarrow^{oq} v}{\mathsf{ifz}\ p\ \mathsf{then}\ q\ \mathsf{else}\ r \Downarrow^{op+oq} v} \qquad \text{(E-IfzT)}$$

$$\frac{p \Downarrow^{op} n \quad n \neq 0 \quad r \Downarrow^{or} v}{\mathsf{ifz}\ p\ \mathsf{then}\ q\ \mathsf{else}\ r \Downarrow^{op+or} v} \qquad \text{(E-IfzF)}$$

$$\frac{p \Downarrow^{op} n}{\mathsf{print}\ msg\ p \Downarrow^{op+[msg+n]} n} \qquad \text{(E-Print)}$$

Notar que la resta la definimos utilizando la resta natural definida como

$$x \dot{-} y \doteq \begin{cases} x - y & x \geqslant y \\ 0 & y > x \end{cases}$$

Además en la regla del operador print, al momento de concatenar el mensaje, utilizamos implícitamente la conversión de naturales a cadenas de caracteres para mostrar en la salida.

En una implementación es útil poder darle nombres a términos, y permitir definiciones "top-level", como hacen muchos lenguajes. En ese caso, se debe llevar un entorno Γ que asocie a cada variable libre su valor[1], y plantear la semántica con respecto a ese entorno (es decir, con la forma $\Gamma \vdash t \Downarrow^{o} v$), agregando también la siguiente regla:

$$\frac{(x \mapsto v) \in \Gamma}{\Gamma \vdash x \Downarrow^{\varnothing} v} \qquad \text{(E-Var)}$$

4. Sintaxis concreta de FD4

En la implementación usaremos la siguiente sintaxis concreta, donde un programa es una lista de declaraciones. Estas declaraciones permiten asociar un identificador a un término.

El operador print lo introducimos como un operador que toma una cadena alfanumérica entre comillas, de la misma manera que tomamos los naturales dentro del lenguaje.

[1] Notar que este entorno no es el mismo que para el tipado: asocia cada variable a su *definición*, en vez de a su *tipo*.

τ ::= `Nat` | τ `->` τ

t ::= x | `fun (`x `:` τ`) ->` t | t t
| n | t `+` t | t `-` t | `ifz` t `then` t `else` t
| `let (`x `:` τ`) =` t `in` t | `fix (`f `:` τ`) (`x `:` τ`) ->` t | `print "`msg`"` t
| `(`t`)`

$decl$ ::= `let` x `=` t

$prog$ ::= $decl$ $prog$ | ϵ

5. Ejercicios

Ej. 1. ¿Cuál es el resultado de evaluar `2-2-2`? ¿Por qué?

Ej. 2. Luego de ejecutar una línea como:

```
FD4> let v = print "Prueba" (1 + 1)
```

¿Qué cambios ocurren en el estado global? ¿Qué pasa cuando luego usamos `v` en otro término? Por ejemplo, ¿qué ocurre al evaluar `v + v`?

Repita el ejercicio para:

```
FD4> let f = fun (x:Nat) -> print "Prueba 2" (x + x)
```

y `f 1 + f 1`.

Ej. 3. ¿Cuál es la salida al evaluar el siguiente fragmento? ¿Por qué?

```
FD4> print "Hola " (print "mundo!" 2)
```