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

Apunte de cátedra: El Lenguaje PCF

El lenguaje PCF es un lenguaje funcional tipado propuesto por Plotkin [1]. PCF está basado en apuntes de Dana Scott que circularon en forma privada y que solo fueron publicados muchos años después [2].
Usaremos una versión más reducida del lenguaje (sin booleanos), que llamamos PCF_0, pero que mantiene la característica de ser Turing completo.

1. Sintaxis

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:

$$\tau_1 \to \tau_2 \to \tau_3 \quad \equiv \quad \tau_1 \to (\tau_2 \to \tau_3)$$

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,...)
	|	$\mathsf{add}\ t\ t$	suma
	|	$\mathsf{subt}\ t\ t$	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 definición local

La aplicación asocia a izquierda. Por ejemplo:

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

Como ejemplo de programa PCF_0, podemos definir la multiplicación como:

$\mathsf{fix}(mul : \mathbb{N} \to \mathbb{N} \to \mathbb{N})(m : \mathbb{N}).$
  $\mathsf{fun}(n : \mathbb{N}).$
    $\mathsf{ifz}\ n\ \mathsf{then}\ 0\ \mathsf{else}\ (\mathsf{ifz}\ \mathsf{subt}\ n\ 1\ \mathsf{then}\ m\ \mathsf{else}\ \mathsf{add}\ m\ (mul\ m\ (\mathsf{subt}\ n\ 1)))$

2. Reglas de Tipado

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \text{(T-Var)} \qquad \frac{}{\Gamma \vdash n : \mathbb{N}} \quad \text{(T-Nat)}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \mathsf{fun}(x : \tau).\, t : \tau \to \sigma} \quad \text{(T-Abs)} \qquad \frac{\Gamma \vdash t_1 : \mathbb{N} \qquad \Gamma \vdash t_2 : \mathbb{N}}{\Gamma \vdash \mathsf{add}\ t_1\ t_2 : \mathbb{N}} \quad \text{(T-Add)}$$

$$\frac{\Gamma \vdash t_1 : \tau \to \sigma \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\ t_2 : \sigma} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash t_1 : \mathbb{N} \qquad \Gamma \vdash t_2 : \mathbb{N}}{\Gamma \vdash \mathsf{subt}\ t_1\ t_2 : \mathbb{N}} \quad \text{(T-Subt)}$$

$$\frac{\Gamma, f : (\tau \to \sigma), x : \tau \vdash t : \sigma}{\Gamma \vdash \mathsf{fix}(f : \tau \to \sigma)(x : \tau).\, t : \tau \to \sigma} \quad \text{(T-Fix)}$$

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma, x : \tau \vdash t_2 : \sigma}{\Gamma \vdash \mathsf{let}\ (x : \tau) := t_1\ \mathsf{in}\ t_2 : \sigma} \quad \text{(T-Let)} \qquad \frac{\Gamma \vdash c : \mathbb{N} \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash \mathsf{ifz}\ c\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \tau} \quad \text{(T-Ifz)}$$

3. Semántica Operacional Big-Step

Le daremos al lenguaje una semántica CBV (Call by value). Los valores son los números naturales, las abstracciones, y los puntos fijos sin aplicar:

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

Las reglas de la semántica big-step para términos cerrados es:

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

$$\frac{t \Downarrow v' \quad v' = \mathsf{fix}(f : \tau_1)(x : \tau_2).\, s \quad u \Downarrow v'' \quad [v'/f, v''/x]s \Downarrow v}{t\ u \Downarrow v}\ \text{(E-AppFix)}$$

$$\frac{t \Downarrow w \quad [w/x]\, t' \Downarrow v}{\mathsf{let}\ (x\colon \tau) \coloneqq t\ \mathsf{in}\ t' \Downarrow v}\ \text{(E-Let)}$$

$$\mathsf{fun}(x : \tau).\, t' \Downarrow \mathsf{fun}(x : \tau).\, t'\ \text{(E-Abs)}$$

$$\mathsf{fix}(f : \tau_1)(x : \tau_2).\, t \Downarrow \mathsf{fix}(f : \tau_1)(x : \tau_2).\, t\ \text{(E-Fix)}$$

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

$$\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{\mathsf{subt}\ t_1\ t_2 \Downarrow n_1 \dot{-} n_2}\ \text{(E-Subt)}$$

$$\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{\mathsf{add}\ t_1\ t_2 \Downarrow n_1 + n_2}\ \text{(E-Add)}$$

$$\frac{t_1 \Downarrow 0 \quad t_2 \Downarrow v}{\mathsf{ifz}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v}\ \text{(E-IfzT)}$$

$$\frac{t_1 \Downarrow n \quad n \neq 0 \quad t_3 \Downarrow v}{\mathsf{ifz}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v}\ \text{(E-IfzF)}$$

No hay una regla para variables ya que supusimos términos cerrados. Además, los puntos fijos no reducen sin ser aplicados (son valores), ya que la idea es que representen funciones, cosa que es garantizada por el tipado.
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, y plantear la semántica con respecto a ese entorno (es decir, con la forma $\Gamma \vdash t \Downarrow v$), agregando también la siguiente regla:

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

Notar que este entorno no es el mismo que para el tipado: asocia cada variable a su *definición*, en vez de a su *tipo*. Este entorno sólo es modificado por declaraciones globales y no puede ser modificado por los términos de PCF_0.

4. Ejercicios

Ejercicio 1. Implementar en PCF_0 las siguientes funciones:

a) resta,

b) exponenciación, y

c) factorial.

Estime sus complejidades en tiempo de ejecución.

Ejercicio 2. Definir representaciones para:

a) Booleanos. Esto requiere representar constructores true, false, y un ifthenelse.

i. ¿Es posible representar ifthenelse simplemente como un término (es decir, una función ternaria), como en la codificación de Church?

ii. ¿Puede arreglarse la codificación, tal vez modificando como se codifican las ramas, para que ifthenelse sea simplemente un término?

iii. Considere el ejercicio anterior en un lenguaje con evaluación Lazy.

b) Pares. Esto requiere representar un constructor de pares pair, y proyecciones proj_1 y proj_2.

Dado que PCF_0 no es polimórfico, es necesario dar una versión para cada tipo que a uno le interese. En este ejercicio, suponer que sólo nos interesan los pares con componentes en `Nat`.

Ejercicio 3. Implementar el algoritmo de Euclides para calcular el máximo común divisor. Este algoritmo está dado por las siguientes ecuaciones:

$$\begin{array}{lcll} \gcd(m,0) & = & m & \\ \gcd(0,n) & = & n & \\ \gcd(m,n) & = & \gcd(m-n,n) & \text{si } m \geqslant n \\ \gcd(m,n) & = & \gcd(m,n-m) & \text{si } m < n \end{array}$$

Ejercicio 4. El sistema T de Gödel visto en ALP tiene un operador de recursión R sobre los naturales. Definirlo en PCF_0 usando recursión general. Recordar que el operador $\mathsf{R}_\tau\ z\ s$ define una función $f : \mathbb{N} \to \tau$ tal que:

$$\begin{array}{lcl} f\ 0 & = & z \\ f\ (\mathsf{succ}\ n) & = & s\ (f\ n)\ n \end{array}$$

Nuevamente, debido a que PCF_0 no es polimórfico, debemos definir un recursor `R_T` para cada tipo `T`.

Ejercicio 5. Probar que PCF_0 es Turing-completo. Dado que por el ejercicio anterior sabemos que se pueden definir todas las funciones primitivas recursivas, solo falta probar que se puede definir un operador de minimización. Es decir un operador que, para una función $f : \mathbb{N} \to \mathbb{N}$, devuelve el mínimo $z : \mathbb{N}$ tal que $fz \Downarrow 0$.

Ejercicio 6. ¿Es la β-reducción una regla ecuacional válida en PCF_0? Es decir: ¿son los términos $(\mathsf{fun}(x : \tau) \to t_1)\, t_2$ y $[t_2/x]t_1$ equivalentes?

Referencias

[1] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223 – 255, 1977.

[2] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci.*, 121(1–2):411–440, December 1993.