Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación

Trabajo de Tesis

La Semántica de Acción para el Lenguaje PCF

que presenta para obtener el título de:
Licenciado en Ciencias de la Computación

El alumno: Ismael Everardo Bárcenas Patiño
Asesor: José de Jesús Lavalle Martínez

Agradecimientos

Quiero dar mis agradecimientos y dedicar este trabajo a Everardo y Luz María, por darme la vida y la mejor familia para crecer. Gracias a Sinthia por su incondicional apoyo. Un agradecimiento muy especial es para Jesús Lavalle, asesor de este trabajo, por mostrarme el mejor camino hacia una vida en la ciencia. Gracias a César Bautísta, Mireya Tovar y Jesús Lavalle, jurado de este trabajo, por sus atinadas observaciones que ayudaron en el desarrollo de esta tesis. También agradesco a todos y cada uno de mis profesores, por crear un buen ambiente para mi desarrollo académico. Also, I want to thank Ken Slonneger.

Contenido

Lista de tablas

Introducción

Los lenguajes proveen medios de comunicación por medio de sonidos y símbolos escritos. Los seres humanos comienzan a aprender lenguajes como una consecuencia de su experiencia cotidiana, pero en la Lingüistica, la ciencia de los lenguajes, las formas y significados de los lenguajes son sujetos a una examinación más rigurosa. Esta ciencia tambien puede ser aplicada a los lenguajes de programación. En contraste a los lenguajes naturales, los cuales sirven para comunicar pensamientos y sentimientos, los lenguajes de programación pueden ser vistos como lenguajes artificiales definidos para la comunicación con computadoras.

El análisis matemático de un lenguaje de programación comienza con la formulación de un *modelo* del lenguaje de programación [Mit96], para tal propósito utilizaremos las siguientes definiciones:

- **El lenguaje objeto** es el lenguaje al cual se está estudiando.

- **El meta-lenguaje** es el lenguaje que utilizamos para describir al lenguaje objeto así como a su significado.

- **La sintaxis** se refiere al modo en que los símbolos pueden ser combinados para crear oraciones bien formadas en el lenguaje. En cuanto los lenguajes de programación, los símbolos se combinan para crear *programas*. La sintáxis define la relación formal entre los componentes de un lenguaje, de tal modo que provee una descripción estructural de las expresiones que forman cadenas válidas en el lenguaje. La sintáxis se encarga sólamente de la forma y la estructura de los símbolos en un lenguaje, sin tomar en consideración su significado.

- **La semántica** interpreta el significado de cadenas sintácticamente válidas en un lenguaje. Para los lenguajes naturales esto significa

relacionar oraciones y frases con objetos, pensamientos y sentimientos de nuestras experiencias. Para los lenguajes de programación, la semántica describe el comportamiento que una computadora tiene cuando un programa, escrito en algún lenguaje, es ejecutado. Este comportamiento es descrito por la relación entre la entrada y salida de un programa ó por una explicación paso-a-paso de como se ejecutará un programa en una máquina, ya sea real ó abstracta.

- **La pragmática** se refiere a los aspectos del lenguaje que involucran a los usuarios, fenómenos psicológicos y sociológicos, tales como la utilidad, rangos de aplicación y efectos en los usuarios. En lo que refiere a los lenguajes de programación, la pragmática incluye aspectos tales como son las facilidad de implementación, eficiencia en la aplicación y la metodología de programación.

La sintaxis debe ser especificada antes que la semántica, ya que el significado puede ser interpretado sólo para expresiones bien formadas en el lenguaje. Análogamente, la semántica necesita ser formulada antes de considerar los aspectos pragmáticos, ya que la interacción con los seres humanos puede ser considerada sólo para expresiones cuyo significado es entendido.

Hay muchas aplicaciones que motivan el análisis de la estructura semántica de los lenguajes de programación. Una definición formal de un lenguaje provee una referencia precisa, completa y estándar para los usuarios e implementadores, entonces las omisiones, contradicciones y ambigüedades típicas de especificaciones de semánticas informales, tales como las del Reporte de Algol 60 [Ten76], pueden ser evitadas. Incluso si una definición formal no fuera comprensible para el programador promedio, podría proveer las bases para una descripción informal más adecuada. Un marco de trabajo general e independiente del lenguaje de los conceptos semánticos, ayuda a estandarizar terminología, clarificar diferencias y similitudes entre los lenguajes, y nos permíte una formulación y prueba rigurosas de las propiedades semánticas del lenguaje. Un diseñador de lenguajes puede analizar construcciones propuestas que le ayudarán a encontrar restricciones indeseables, incompatibilidades, ambigüedades, etc.

El lenguaje PCF(*Programming Computable Functions*) originalmente formulado por Dana Scott [Mit96], es un lenguaje funcional con tipos, basado en el cálculo lambda. Este lenguaje está diseñado para ser fácilmente analizado. La principal ventaja en el estudio de este lenguaje es que al estar basado en el Cálculo Lambda, sus propiedades son fácilmente generalizables, lo cual,

es de gran utilidad en el estudio de una buena variedad de lenguajes. Dada la importancia del cálculo lambda en el estudio de PCF, el primer capítulo presenta las principales características de éste. El segundo capítulo describe de manera detallada la sintáxis de PCF, asi como también, se presentan tres distíntas semánticas de PCF, las cuales son, la denotacional, la axiomática y la operacional.

La semántica de acción es un formalismo para la especificación formal de lenguajes de programación [Mos96]. Su mayor ventaja sobre otras semánticas es pragmática, es decir, se ajusta sin problemas a lenguajes de programación grandes, tales como, C, Java, Standard ML, etc. Cabe notar la reusabilidad de la semántica de acción, lo que nos permite realizar extensiones y cambios a la semántica proporcionales a los realizados en el lenguaje. El tercer capítulo de este trabajo está dedicado al estudio de la semántica de acción, y se presenta una descripción completa de PCF mediante esta semántica.

En las conclusiones se describen las ventajas de la semántica de acción sobre la semántica denotacional. Finalmente se presenta la bibliografía.

Capítulo 1

Cálculo Lambda

El cálculo lambda (cálculo λ) fue creado orginialmente por Church con el fin de formar una Teoría General de Funciones y Lógica [Bar93, Mit96, SK95]. Ha sido probado que el cálculo λ es incompleto [Bar93, Cut80, SK95], pero puede representar todas las funciones computables [Bar93, Cut80].

El modelo original del cálculo λ representa funciones que no tienen algún *tipo* en particular, dada la naturaleza de esta teoría para representar funciones computables y su consecuente utilidad para el estudio de los lenguajes de programación, fue necesario crear un modelo que representara funciones con tipos.

Este capítulo presenta los dos modelos de cálculo λ, el cálculo λ tipificado y el cálculo λ no tipificado.

1.1 Cálculo lambda no tipificado

En esta sección se describen los elementos necesarios del cálculo λ para la representación de funciones sin tipos. Un término–λ es un término del cálculo λ y se define a continuación.

Definición 1. *El conjunto de términos–λ Λ, está construido a partir de un conjunto infinito de variables $V = \{v, v', v'', \ldots\}$ de esta manera:*

$$\begin{array}{ll} x \in V & \Rightarrow x \in \Lambda, \\ M, N \in \Lambda & \Rightarrow (M\ N) \in \Lambda, \\ M \in \Lambda, x \in V & \Rightarrow (\lambda x.M) \in \Lambda. \end{array}$$

Convención 1. *1. $x, y, z, \ldots$ denotan variables arbitrarias; $M, N, L, \ldots$ denotan términos–λ arbitrarios.*

2. Usaremos $F\ M_1\ \ldots M_n$ en lugar de $(\ldots((F\ M_1)M_2)\ldots M_n)$ y $\lambda x_1 \ldots x_n.M$ en lugar de $(\lambda x_1(\lambda x_2(\ldots(\lambda x_n.M)\ldots)))$.

3. Los paréntesis más externos no se escriben.

Definición 2. *1. El conjunto de las variables libres $FV(M)$ de $M \in \Lambda$, se define mediante inducción por:*

$$\begin{array}{ll} FV(x) & = \{x\}; \\ FV(M\ N) & = FV(M) \cup FV(N); \\ FV(\lambda x.M) & = FV(M) - \{x\} \end{array}$$

2. M es un término–λ cerrado (o combinador), si $FV(M) = \oslash$. El conjunto de términos–λ cerrados es denotado por Λ^0.

3. La sustitución de una variable (libre) por una expresión en otra expresión lambda se denota $E[v \to E_1]$ y se define por:

$$\begin{array}{ll} v[v \to E_1] & = E_1 \text{ para cualquier variable } v \\ x[v \to E_1] & = x \text{ para cualquier variable } x \neq v \\ c[v \to E_1] & = c \text{ para cualquier constante } c \\ (E\ E_2)[v \to E_1] & = ((E[v \to E_1])(E_2[v \to E_1])) \\ (\lambda v.E)[v \to E_1] & = (\lambda v.E) \\ (\lambda x.E)[v \to E_1] & = \lambda x.(E[v \to E_1]) \text{ cuando } x \neq v \text{ y } x \notin FV(E_1) \\ (\lambda x.E)[v \to E_1] & = \lambda z.(E[x \to z][v \to E_1]) \\ & \quad \text{cuando } x \neq v \text{ y } x \in FV(E_1), \\ & \quad \text{donde } z \neq v \text{ y } z \notin FV(E\ E_1) \end{array}$$

donde $E, E_1, E_2 \in \Lambda$.

Definición 3. *Sean $v, w, E \in \Lambda$ y $w \notin FV(E)$. Entonces*

$$\lambda v.E =_\alpha \lambda w.E[v \to w]$$

Notación 1. *$M \equiv N$ denota que M y N son el mismo término o que pueden ser obtenidos uno del otro por medio del renombramiento de variables acotadas.*

Lema 1 (Lema de sustitución). *Sean $M, N, L \in \Lambda$. Supongase $x \not\equiv y$ y $x \notin FV(L)$. Entonces*

$$M\,[x \to N]\,[y \to L] \equiv\ M\,[y \to L]\,[x \to N\,[y \to L]]\,.$$

Este lema se puede demostrar mediante inducción sobre la estructura de M.

A continuación se presenta el cálculo λ como una teoría formal de ecuaciones entre términos–λ.

Definición 4. *Sean $v, E, E_1 \in \Lambda$. Entonces,*

$$(\lambda v.E)E_1 = E[v \to E_1]$$

Observación:
$E_1 = E_2$ implica:

- $E_1\ E\ = E_2\ E$
- $E\ E_1 = E\ E_2$
- $\lambda x.E_1 = \lambda x.E_2$

Hasta aquí se ha mostrado una definición formal del cálculo λ sin tipos, a continuación se presentan los elementos necesarios para poder modelar recursión por medio del cálculo λ.

Teorema 1 (Teorema del punto fijo). *1. $\forall F \in \Lambda\ \exists X \in \Lambda :\ F\ X\ = X$.*

2. Existe un combinador de punto fijo

$$Y \equiv \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

tal que

$$\forall F \in \Lambda :\ F(YF)\ =\ YF.$$

Demostración.

1. Sean $W \equiv \lambda x.F(xx)$ y $X \equiv WW$. Entonces $X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv FX$

2. Por la prueba de (1). Notese que

$$YF = (\lambda x.F(x\,x))(\lambda x.F(x\,x)) \equiv X$$

y por (1)

$$FX = F(YF) = X = YF. \quad \blacksquare$$

Definición 5. *Sea $C\,[x_1, x_2, \ldots, x_i]$ un término $C \in \Lambda$, $x_i \in \Lambda$ variables libres contenidas en C y $i \in \mathbb{N}$.*

Corolario 1. *Dado un término $C \equiv\ C\,[f, x]$, entonces*

$$\exists F \in \Lambda \forall X \in \Lambda : F\ X =\ C\,[F, X]\,.$$

Donde $C[F, X]$ es el resultante de la sustitución $C\,[F \to f]\,[X \to x]$.

Demostración. De hecho, se puede construir F suponiendo que tiene la propiedad requerida y calculandola hacia atras:

$$\begin{array}{ll} & \forall X \in \Lambda : F\ X \ = C\,[F, X] \\ \Leftarrow & F\ x \qquad\qquad = C\,[F, x] \\ \Leftarrow & F \qquad\qquad = \lambda x.C\,[F, x] \\ \Leftarrow & F \qquad\qquad = \lambda(\lambda f\ x.C\,[f, x])F \\ \Leftarrow & F \qquad\qquad \equiv Y(\lambda f\ x.C\,[f, x]). \quad \blacksquare \end{array}$$

1.1.1 Definibilidad

En esta sección se mostrará que todas las funciones computables son definibles en el cálculo λ, para tal fin, comenzaremos definiendo numerales y funciones numéricas sobre ellos.

Definición 6. *1. $F^n(M)$ con $n \in \mathbb{N}$ y $F, M \in \Lambda$, se define mediante inducción por:*

$$\begin{array}{ll} F^0(M) & \equiv M; \\ F^{n+1}(M) & \equiv F(F^n(M)). \end{array}$$

2. *Los* numerales Church $c_0, c_1, \ldots$ *son definidos:*

$$c_n \equiv \lambda f\, x.f^n(x).$$

Para representar las operaciones de adición, multiplicación y exponenciación sobre los numerales requerimos del Lema 2 y la Proposición 1.

Lema 2. *Sean c un numeral, $n, m \in \mathbb{N}$ y $x, y \in \Lambda$ entonces:*

1. $(c_n x)^m(y) = x^{n*m}(y)$;
2. $(c_n)^m(x) = c_{(n^m)}(x)$, *para* $m > 0$.

Prueba.

1. Si $m = 0$ entonces (1) se cumple. Sea $m = k + 1$ entonces

$$\begin{aligned}(c_n x)^{k+1}(y) &= c_n x((c_n x)^k(y)) \\ &=_{\mathbf{HI}} c_n x(x^{n*k}(y)) \\ &= x^n(x^{n*k}(y)) \\ &= x^{n+n*k}(y) \\ &= x^{n*(k+1)}(y).\end{aligned}$$

2. Si $m = 1$, entonces (2) se cumple. Si $m = k + 1$ entonces

$$\begin{aligned}c_n^{k+1}(x) &= c_n(c_n^k(x)) \\ &=_{\mathbf{HI}} c_n(c_{(n^k)}(x)) \\ &= \lambda y.(c_{(n^k)}(x))^n(y) \\ &=_{(\mathbf{1})} \lambda y.x^{n^k * n}(y) \\ &= c_{(n^{k+1})}x. \quad \blacksquare\end{aligned}$$

La siguiente proposición nos proporciona una representación de las funciones de suma, multiplicación y exponenciación en los número naturales.

Proposición 1 (J. B. Rosser). *Sean*

$$\begin{aligned}A_+ &\equiv \lambda xypq.xp(ypq); \\ A_* &\equiv \lambda xyz.x(yz); \\ A_{exp} &\equiv \lambda xy.yx.\end{aligned}$$

Entonces $\forall n, m \in \mathbb{N}$:

1. $A_+ c_n c_m = c_{n+m}$.
2. $A_* c_n c_m = c_{n.m}$.
3. $A_{exp} c_n c_m = c_{(n^m)}$, *excepto para* $m = 0$

Prueba.

1. Por inducción sobre m.
2. Por (1) del lema anterior.
3. Por (2) del lema anterior tenemos para $m > 0$

$$A_{exp} c_n c_m = c_m c_n = \lambda x.c_n^m(x) = \lambda x.c_{(n^m)} x = c_{(n^m)},$$

ya que $\lambda x.M\, x = M$ si $M = \lambda y.M'[y]$ y $x \notin FV(M)$. De hecho

$$\begin{aligned} \lambda x.M\, x &= \lambda x.(\lambda y.M'[y])x \\ &= \lambda x.M'[x] \\ &\equiv \lambda y.M'[x] \\ &= M. \quad \blacksquare \end{aligned}$$

A continuación se presenta la representación de los valores booleanos, un operador condicional, así como los pares ordenados dentro del cálculo λ.

Definición 7. *1.* $\mathbf{true} \equiv \lambda xy.x$, $\mathbf{false} \equiv \lambda xy.y$.

2. *Si B es un Booleano, es decir, un término que es* **true** *o* **false**, *entonces*

$$\mathbf{if}\ B\ \mathbf{then}\ P\ \mathbf{else}\ Q$$

puede ser representado por BPQ. De hecho, $\mathbf{true}PQ = P$ *y* $\mathbf{false}PQ = Q$.

Definición 8. *Para $M, N \in \Lambda$ se escribe*

$$[M, N] \equiv \lambda z.z\ M\ N.$$

entonces

$$[M, N]\,\mathbf{true} = M$$
$$[M, N]\,\mathbf{false} = N$$

así $[M, N]$ puede ser usado como un par ordenado.

Definición 9. *1. Una función numérica es un mapeo* $f : \mathbb{N}^p \to \mathbb{N}$ *para cualquier* p.

2. Una función numerica f *con* p *argumentos es llamada definible–*λ *si tiene algún combinador* F

$$F\, c_{n_1} \ldots c_{n_p} = c_{f(n_1,\ldots,n_p)}$$

$\forall n_1, \ldots, n_p \in \mathbb{N}$.

Definición 10. *1. Las funciones iniciales son funciones numéricas* U_r^i, S^+, Z *definidas:*

$$\begin{aligned} U_r^i(x_1, \ldots, x_r) &= x_i, \quad 1 \leq i \leq r; \\ S^+(n) &= n + 1; \\ Z(n) &= 0. \end{aligned}$$

2. Sea $P(n)$ *una relación numérica. Entonces*

$$\mu m.P(m)$$

denota al último número m *tal que* $P(m)$ *se cumple, si existe ese número, en otro caso es indefinida.*

La clase $\mathcal{R}$ de funciones recursivas es la clase más pequeña de funciones numéricas que contiene las funciones iniciales, y es cerrada bajo composición, recursión primitva y minimilización [Bar93, Cut80].

Lema 3. *Las funciones iniciales son definibles–*λ.

Prueba. Si

$$\begin{aligned} U_p^i &\equiv \lambda x_1 \ldots x_p.x_i; \\ S^+ &\equiv \lambda xyz.y(xyz); \\ Z &\equiv \lambda x.c_0. \end{aligned}$$

entonces U_p^i, S^+, Z son definibles–λ ■

Definición 11.

$$\overrightarrow{x} = x_1, x_2, \ldots, x_i$$

donde $x \in \Lambda$ *y* $i \in \mathbb{N}$

Lema 4. *Las funciones definibles–λ son cerradas bajo composición.*

Prueba. Sean $g, h_1, \ldots, h_m$ definidas–λ por $G, H_1, \ldots, H_m$ respectivamente. Entonces

$$f(\overrightarrow{n}) = g(h_1(\overrightarrow{n}), \ldots, h_m(\overrightarrow{n}))$$

es definida–λ por

$$F \equiv \lambda \overrightarrow{x}.G(H_1 \overrightarrow{x}) \ldots (H_m \overrightarrow{x}) \quad \blacksquare$$

Lema 5. *Las funciones definibles–λ son cerradas bajo recursión primitiva.*

Prueba. Sea f definida por

$$f(0, \overrightarrow{n}) = g(\overrightarrow{n}) \quad f(k+1, \overrightarrow{n}) = h(f(k, \overrightarrow{n}), k, \overrightarrow{n})$$

donde g, h son definidas–λ por G, H respectivamente, y sea

$$T \equiv \lambda p. \left[S^+(p\,\mathbf{true}), H(p\,\mathbf{false})(p\,\mathbf{true})\right].$$

Entonces para todo k se tiene

$$T\left[c_k, c_{f(k)}\right] = \left[fS^+c_k, Hc_{f(k)}c_k\right] = \left[c_{k+1}, c_{f(k+1)}\right].$$

Por inducción sobre k se tiene que

$$\left[c_k, c_{f(k)}\right] = T^k \left[c_0, c_{f(0)}\right].$$

Por lo tanto

$$c_{f(k)} = c_k T \left[c_0, c_{f(0)}\right] \mathbf{false},$$

y f puede ser definida–λ por

$$F \equiv \lambda k.kT\,[c_0, G]\,\mathbf{false}. \quad \blacksquare$$

Lema 6. *Las funciones definibles–λ son cerradas bajo minimalización.*

Prueba. Sea f definida $f(\overrightarrow{n}) = \mu m\,[g(\overrightarrow{n}, m) = 0]$, donde $\overrightarrow{n} = n_1, \ldots, n_k$ y g es definida–λ por G. Sea

$$\mathbf{zero} \equiv \lambda n.n(\mathbf{true}\,\mathbf{false})\mathbf{true}.$$

Entonces

$$\begin{aligned} \mathbf{zero}\, c_0 &= \mathbf{true}, \\ \mathbf{zero}\, c_{n+1} &= \mathbf{false}. \end{aligned}$$

Por el Corolario 1 anterior sabemos que existe un término H tal que

$$H\,\overrightarrow{n}\,y = \mathbf{if}\ (\mathbf{zero}(G\,\overrightarrow{n}\,y))\ \mathbf{then}\ y\ \mathbf{else}\ H\,\overrightarrow{n}\,(S^+y).$$

Sea $F = \lambda\overrightarrow{n}.H\,\overrightarrow{x}\,c0$. Entonces F define–λ a f:

$$\begin{aligned} Fc_{\overrightarrow{x}} &= Hc_{\overrightarrow{n}}c_0 \\ &= c_0, \quad \text{si } Gc_{\overrightarrow{n}}c_0 = c_0, \\ &= Hc_{\overrightarrow{n}}c_1 \quad \text{en otro caso;} \\ &= c_1, \quad \text{si } Gc_{\overrightarrow{n}}c_1 = c_0, \\ &= Hc_{\overrightarrow{n}}c_2 \quad \text{en otro caso;} \\ &= c_2, \quad \text{si } \ldots \\ &= \ldots \end{aligned}$$

donde $c_{\overrightarrow{n}} = c_{n_1} \ldots c_{n_k}$ ■

Dado que las funciones númericas son definibles–λ y que la funciones definibles–λ son cerradas bajo minilización, recursión y composición, entonces:

Teorema 2. *Todas la funciones recursivas son definibles–λ.*

La prueba de este teorema, es inductiva de los Lemas 3-6.

1.1.2 Reducción

La reducción de términos–λ es útil para un análisis de convertibilidad. Según el Teorema de Church-Roser, dos términos son convertibles, si hay un término al cual los dos primeros se reducen. En muchos casos la inconvertibilidad de dos términos puede ser probada mostrando que éstos no se reducen a un término común.

Definición 12. • *Una relación binaria R en Λ es* compatible *si*

$$\begin{aligned} M\,R\,N \;\Rightarrow\; & (ZM)\,R\,(ZN), \\ & (MZ)\,R\,(NZ)\ y \\ & (\lambda x.M)\,R\,(\lambda x.N). \end{aligned}$$

- *Una relación de* congruencia *en Λ es una relación de equivalencia compatible.*
- *Una relación de* reducción *en Λ es una relación compatible, reflexiva y transitiva.*

Definición 13. *Las relaciones binarias $\rightarrow_\beta, \twoheadrightarrow_\beta$ y $=_\beta$ en Λ estan definidas:*

1. *(a)* $(\lambda x.M)N \rightarrow_\beta M[N \rightarrow x]$;
   (b) $M \rightarrow_\beta N \Rightarrow ZM \rightarrow_\beta ZN, MZ \rightarrow_\beta NZ$ *y* $\lambda x.M \rightarrow_\beta \lambda x.N$.
2. *(a)* $M \twoheadrightarrow_\beta M$;
   (b) $M \rightarrow_\beta N \Rightarrow M \twoheadrightarrow_\beta N$;
   (c) $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \Rightarrow M \twoheadrightarrow_\beta L$.
3. *(a)* $M \rightarrow_\beta N \Rightarrow M =_\beta N$;
   (b) $M =_\beta N \Rightarrow M =_\beta N$;
   (c) $M =_\beta N, N =_\beta L \Rightarrow M =_\beta L$.

1. *Si $M \rightarrow_\beta N$ se dice que M se reduce–β a N en un paso.*
   Nótese que por definición $\rightarrow_\beta$ es compatible.
2. *Si $M \twoheadrightarrow_\beta N$ se dice que M se reduce–β a N.*
   La relación $\twoheadrightarrow_\beta$ es la cerradura transitiva reflexiva de $\rightarrow_\beta$ y por lo tanto una relación de reducción.
3. *Si $M =_\beta N$ se dice que M es convertible–β a N.*
   La relación $=_\beta$ es una relación de congruencia.

Definición 14. • *Un redex–β es un término de la forma $(\lambda x.M)N$.*

- *Un término–λ t está en forma–β normal ($\beta - nf$) si t no tiene un redex–β como subexpresión.*
- *Un término M tiene una forma–β normal si $M =_\beta N$ y N está en $\beta - nf$, para algún N.*

Lema 7. *Sean $M, M', N \in \Lambda$.*

1. *Suponga que M está en $\beta - nf$. Entonces*

$$M \twoheadrightarrow_\beta N \Rightarrow N \equiv M.$$

2. *Si $M \rightarrow_\beta M'$, entonces $M[N \rightarrow x] \rightarrow_\beta M'[N \rightarrow x]$.*

Demostración.

1. Si M es una $\beta - nf$, entonces M no contiene un β–redex. Entonces $M \not\rightarrow_\beta N$. Por lo tanto si $M \twoheadrightarrow_\beta N$, entonces ésto debe ser porque $M \equiv N$.
2. Por inducción en la generación de $\rightarrow_\beta$. ■

Algunas consecuencias del Teorema de Confluencia son las siguientes:

- El cálculo λ es consistente, i.e, $\lambda \not\vdash \mathbf{true}{=}\mathbf{false}$.
- No todos términos–λ tienen forma normal, por ejemplo, $(\lambda x.xx)(\lambda x.xx)$.
- El orden de reducción β no es único, pero sin importar el orden, siempre se llega a una forma normal única, si es que exíste.

La justificación de estas consecuencias se presentan a continuación.

Para la demostración del Teorema de Confluencia es necesario probar que:

$$\begin{array}{ccc} M & \xrightarrow[\beta]{} & N_2 \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\beta} \\ N_1 & \xrightarrow[\beta]{} & N_3 \end{array}$$

donde la reducción de N_1 a N_3, de M a N_2 y de N_2 a N_3 pueden ser en uno o más pasos. Además $M, N_1, N_2, N_3 \in \Lambda$.

La idea intuitiva de la demostración de este Lema es la suguiente: sea $M \to_\beta N_1$ una reducción de un paso resultante del cambio de un redex$-\beta$ R en M a su contracción R' en N_1. Si hacemos un análisis minucioso de que pasa con R durante la reducción $M \twoheadrightarrow_\beta N_2$, entonces reduciendo todos los residuos de R en N_2, el término N_3 puede encontrarse. Con el fin de hacer este análisis, un conjunto extendido $\underline{\lambda} \supseteq \lambda$ y una reducción $\underline{\beta}$ son definidas. El subrayado es usado en una forma similar al *trazado de isótopos radioactivos* en la Biología experimental [Bar93].

Definición 15. • $\underline{\Lambda}$ *es un conjunto de términos, definido inductivamente a continuación:*

$$\begin{array}{ll} x \in V & \Rightarrow x \in \underline{\Lambda}; \\ M, N \in \underline{\Lambda} & \Rightarrow (MN) \in \underline{\Lambda}; \\ M \in \underline{\Lambda}, x \in V & \Rightarrow (\lambda x.M) \in \underline{\Lambda}; \\ M, N \in \underline{\Lambda}, x \in V & \Rightarrow ((\underline{\lambda} x.M)N) \in \underline{\Lambda}. \end{array}$$

- *Las reducciones subrayadas ($\to_{\underline{\beta}}$, $\twoheadrightarrow_{\underline{\beta}}$) están definidas con las siguientes reglas de contracción:*

$$\begin{array}{ll} (\lambda x.M)N & \to M[N \to x], \\ (\underline{\lambda} x.M)N & \to M[N \to x]. \end{array}$$

  Entonces $\to$ es extendida a la relación compatible $\to_{\underline{\beta}}$ y $\twoheadrightarrow_{\underline{\beta}}$ es la cerradura reflexiva transitiva de $\to_{\underline{\beta}}$

- *Si $M \in \underline{\Lambda}$, entonces $\mid M \mid \in \Lambda$ es obtenido a partir de M dejando fuera todos los subrayados. Por ejemplo, $\mid (\lambda x.x)((\underline{\lambda} x.x)(\lambda x.x)) \mid\mid \equiv \mid (\mid\mid)$.*

- *Sustitución para* $\underline{\Lambda}$ *se define por:*

$$\begin{array}{ll}
v[v \to E_1] & = E_1 \text{ para cualquier variable } v \\
x[v \to E_1] & = x \text{ para cualquier variable } x \neq v \\
c[v \to E_1] & = c \text{ para cualquier constante } c \\
(E\ E_2)[v \to E_1] & = ((E[v \to E_1])(E_2[v \to E_1])) \\
(\lambda v.E)[v \to E_1] & = (\lambda v.E) \\
(\lambda x.E)[v \to E_1] & = \lambda x.(E[v \to E_1]) \text{ cuando } x \neq v \text{ y } x \notin FV(E_1) \\
(\lambda x.E)[v \to E_1] & = \lambda z.(E[x \to z][v \to E_1]) \\
 & \quad \text{cuando } x \neq v \text{ y } x \in FV(E_1), \\
 & \quad \text{donde } z \neq v \text{ y } z \notin FV(E\ E_1) \\
((\underline{\lambda} x.M)N)[L \to y] & \equiv (\underline{\lambda} x.M[L \to y])(N[L \to y]) \\
\text{donde } E, E_1, E_2 \in \Lambda. &
\end{array}$$

Definición 16. *Un mapeo* $\varphi : \Lambda \to \Lambda$ *se define mediante inducción por:*

$$\begin{array}{ll}
\varphi(x) & \equiv x; \\
\varphi(MN) & \equiv \varphi(M)\varphi(N), \text{ si } M, N \in \underline{\Lambda}; \\
\varphi(\lambda x.M) & \equiv \lambda x.\varphi(M); \\
\varphi((\underline{\lambda} x.M)N) & \equiv \varphi(M)[\varphi(N) \to x].
\end{array}$$

En otras palabras, el mapeo φ contrae todos los redex$-\beta$ que son subrayados, de adentro hacia afuera.

Notación 2. *Si* $\mid M \mid\equiv N$ *o* $\varphi(M) \equiv N$, *entonces se denotarán:*

$$M \underset{||}{\longrightarrow} N \text{ o } M \underset{\varphi}{\longrightarrow} N.$$

Lema 8.

$$\begin{array}{ccc}
M' & \underset{\underline{\beta}}{\longrightarrow} & N' \\
{\scriptstyle ||}\downarrow & & \downarrow{\scriptstyle ||} \\
M & \underset{\beta}{\longrightarrow} & N
\end{array}$$

donde la reducción de M' *a* N' *puede ser en uno o más pasos, al igual que la de* M *a* N. *Además,* $M', N' \in \underline{\Lambda}$ *y* $M, N \in \Lambda$.

Demostración. Supongase que $M \to_\beta N$. Entonces N es obtenido por la contracción de un $\beta-$redex en M y N' puede ser obtenido por la contracción del correspondiente $\beta-$redex en M'. ■

Lema 9. *Sean $M, M', N, L \in \underline{\Lambda}$. Entonces*

1. *Supongase $x \not\equiv y$ y $x \notin FV(L)$. Entonces*

$$M[N \to x][L \to y] \equiv M[L \to y][N \to x[L \to y]]$$

2.

$$\varphi(M[N \to x]) \equiv \varphi(M)[\varphi(N) \to x]$$

3.

$$\begin{array}{ccc} M & \xrightarrow[\underline{\beta}]{} & N \\ {\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle\varphi} \\ \varphi(M) & \xrightarrow[\beta]{} & \varphi(N) \end{array}$$

donde la reducción de M a N puede ser en uno o más pasos, al igual que de $\varphi(M)$ a $\varphi(N)$. Además $M, N \in \underline{\Lambda}$.

Demostración.

1. Claramente por inducción sobre la estructura de M, (1) se cumple.
2. Por inducción en la estructura de M, usando (1) en el caso $M \equiv (\underline{\lambda} y.P)Q$. La condición de (1) puede ser asumida para sostener la convención acerca de la variables libres.
3. Por inducción en la generación de $\twoheadrightarrow_\beta$, usando (2). ■

Lema 10.

$$\begin{array}{ccc} M & \xrightarrow[||]{} & N \\ {\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle\beta} \\ L & = & L \end{array}$$

donde la reducción de N a L puede ser en uno o más pasos. Además $M \in \underline{\Lambda}$, $N, L \in \Lambda$.

La demostración es por inducción sobre la estructura de M.

Lema 11.

$$\begin{array}{ccc} M & \xrightarrow[\beta]{} & N_2 \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\beta} \\ N_1 & \xrightarrow[\beta]{} & N_3 \end{array}$$

donde la reducción de N_1 a N_3, de M a N_2 y de N_2 a N_3 pueden ser en uno o más pasos. Además $M, N_1, N_2, N_3 \in \Lambda$.

Demostración. Si N_1 es el resultado de la contracción de la ocurrencia redex $(\lambda x.P)Q$ en M, y si $M' \in \underline{\Lambda}$ puede ser obtenida de M reemplazando R por $R' \equiv (\underline{\lambda}x.P)Q$. Entonces $\mid M' \mid$ y $\varphi(M') \equiv N_1$. Por transitividad de los lemas anteriores queda probado este lema. ■

Teorema 3 (Teorema de Confluencia). *Si $M \twoheadrightarrow_\beta N_1, M \twoheadrightarrow_\beta N_2$, entonces para algún N_3 se tiene que $N_1 \twoheadrightarrow_\beta N_3$ y $N_2 \twoheadrightarrow_\beta N_3$.*

Demostración. Si $M \twoheadrightarrow_\beta N_1$, entonces $M \equiv M_0 \to_\beta M_1 \to_\beta \ldots M_n \equiv N_1$. Entonces la propiedad Church-Roser (este teorema) concuerda con el lema anterior y el siguiente diagrama:

$$\begin{array}{ccc} M & \longrightarrow & N_2 \\ \downarrow & & \downarrow \\ M_1 & \longrightarrow & . \\ \downarrow & & \downarrow \\ . & \longrightarrow & . \\ \vdots & \vdots & \vdots \\ & \longrightarrow & \\ & \vdots & \\ . & \longrightarrow & . \\ \downarrow & & \downarrow \\ N_1 & \longrightarrow & . \end{array}$$

■

1.2 Cálculo Lambda tipificado

Exíste dos tipos de sistemas en el cálculo λ con tipos: el sistema Curry y el sistema Church [Bar93].

El sistema Curry [Bar93], llamado sistema de asignación de tipos, no expresa sintácticamente los tipos. El tipo o tipos de cada término es deducido a través de su contexto, cuando es posible, ya que este proceso es indecidible [Bar93].

El sistema Church [Bar93], hace referencia explícita y sintáctica al tipo único de cada término.

A continuación se muestra un ejemplo del mismo término en los dos diferentes sistemas:

$$\begin{aligned} &\vdash_{\mathbf{Curry}} && (\lambda x.x) : \ (\sigma \to \sigma), \\ &\vdash_{\mathbf{Church}} && (\lambda x : \sigma.x) : \ (\sigma \to \sigma). \end{aligned}$$

Dado que el lenguaje PCF está basado en el cálculo λ con tipo único para cada término [HO95, Mit96], se presenta en esta sección solamente el sistema Church.

1.2.1 Sistema Church

Definición 17. *Sea $\mathbb{T}$ un conjunto de tipos. El conjunto de términos–λ de tipo $\mathbb{T}$, llamados pseudotérminos, se escriben $\Lambda_{\mathbb{T}}$ y se definen:*

$$\Lambda_{\mathbb{T}} = \ V \mid \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}} \mid \lambda x : \mathbb{T}.\Lambda_{\mathbb{T}}$$

donde V denota el conjunto de variables.

Definición 18. *El cálculo lambda tipificado Church–$\lambda \to$ se define mediante:*

1. *El conjunto de tipos $\mathbb{T} = Type(\lambda \to)$ es definido por*

   $$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T}.$$

   Las variables para tipos $\mathbb{V}$ se denotan diferente que las variables para términos–λ V. Por ejemplo $\mathbb{V} = nat, bool$ y $V = x, y, \ldots$

2. *En una* declaración *de la forma $M : \sigma$ con $M \in \Lambda_{\mathbb{T}}$ y $\sigma \in \mathbb{T}$. El tipo σ es el predicado y el término M es el sujeto de la declaración.*

3. *Una base es un conjunto de declaraciones con solamente variables distintas como sujetos.*

Definición 19. *Una declaración* $M : \sigma$ *es derivable de la base* Γ*, escrita* $\Gamma \vdash M : \sigma$*, si* $M : \sigma$ *puede ser producida usando las siguientes reglas:*

$$\Gamma \vdash x : \sigma, \quad si\ (x : \sigma) \in \Gamma;$$
$$\frac{\Gamma \vdash M : (\sigma \to \tau)\ \ \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau};$$
$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.M) : (\sigma \to \tau)}.$$

Definición 20. *El conjunto de términos–*$\lambda \to$*, escrito* $\Lambda(\lambda \to)$*, es definido por*

$$\Lambda(\lambda \to) = \{M \in \Lambda_{\mathbb{T}} \mid \exists \Gamma, \sigma \quad \Gamma \vdash M : \sigma\}$$

Definición 21. *Las relaciones binarias* $\to_\beta, \twoheadrightarrow_\beta$ *y* $=_\beta$ *en* $\Lambda_{\mathbb{T}}$ *son generadas por la siguiente regla de contracción*

$$(\lambda x : \sigma.M)N \to\ M[N \to x]$$

El teorema de Church-Rosser tambien se cumple para el sistema Church, se omite la demostración dada la similitud a la expuesta para el cálculo λ no tipificado.

El cálculo λ tipificado conserva la mayoría de la características del cálculo λ no tipificado. A continuación mostramos algúnas de ellas, así como una nueva, que es la unicidad de tipos.

Proposición 2 (Lema de base). *Sea* Γ *una base.*

1. *Si* $\Gamma' \supseteq \Gamma$ *es otra base, entonces* $\Gamma \vdash M : \sigma \Rightarrow\ \Gamma' \vdash M : \sigma$.
2. $\Gamma \vdash M : \sigma \Rightarrow\ FV(M) \subseteq dom(\Gamma)$
3. $\Gamma \vdash M : \sigma \Rightarrow\ \Gamma - FV(M) \vdash M : \sigma$

Demostración.

1. Por inducción en la derivación de $M : \sigma$. Se presentan los siguientes casos:

(a) $M : \sigma$ es $x : \sigma$ y es elemento de Γ. Entonces también $x : \sigma \in \Gamma'$ y por consiguiente $\Gamma' \vdash : \sigma$.

(b) $M : \sigma$ es $(M_1 M_2) : \sigma$ seguido directamente de $M_1 : (\tau \to \sigma)$ y $M_2 : \tau$ para algún τ. Por hipótesis de inducción se tiene que $\Gamma' \vdash M_1 : (\tau \to \sigma)$ y $\Gamma' \vdash M_2 : \tau$. De ahi que $\Gamma' \vdash (M_1 M_2) : \sigma$.

(c) $M : \sigma$ es $(\lambda x.M_1) : (\sigma_1 \to \sigma_2)$ seguido directamente de $\Gamma, x : \sigma_1 \vdash M_1 : \sigma_2$. Suponiendo que la variable acotada x no ocurre en $dom(\Gamma')$. Entonces $\Gamma', x : \sigma_1$ es también una base, por la cual $\Gamma, x : \sigma_1$. Por lo tanto por **HI** se tiene que $\Gamma', x : \sigma_1 \vdash M_1 : \sigma_2$ y en consecuencia $\Gamma' \vdash (\lambda x.M_1) : (\sigma_1 \to \sigma_2)$.

2. Por inducción de la derivación de $M : \sigma$. Sólo se trata el caso $M : \sigma$ es $(\lambda x.M_1) : (\sigma_1 \to \sigma_2)$ seguido directamente de $\Gamma, x : \sigma_1 \vdash M_1 : \sigma_2$. Sea $y \in FV(\lambda x.M_1)$, entonces $y \in FV(M_1)$ y $y \not\equiv x$. Por **HI** se tiene que $y \in dom(\Gamma, x : \sigma_1)$ y por lo tanto $y \in dom(\Gamma)$

3. Por inducción de la derivación de $M : \sigma$. Sólo se trata el caso $M : \sigma$ es $(M_1 M_2) : \sigma$ seguido directamente de $M_1 : (\tau \to \sigma)$ y $M_2 : \tau$ para algún τ. Por **HI** se tiene que $\Gamma - FV(M_1) \vdash M_1 : (\tau \to \sigma)$ y $\Gamma - FV(M_2) \vdash M_2 : \tau$. Por (1) se tiene que $\Gamma - FV(M_1 M_2) \vdash (M_1 M_2) : \sigma$. ■

Proposición 3 (Lema de generación). *1.* $\Gamma \vdash x : \sigma \Rightarrow (x : \sigma) \in \Gamma$.

2. $\Gamma \vdash M\ N : \tau \Rightarrow \exists \sigma[\Gamma \vdash M : (\sigma \to \tau)\ y\ \Gamma \vdash N : \sigma]$

3. $\Gamma \vdash (\lambda x : \sigma.M) : \rho \Rightarrow \exists \tau[\rho = (\sigma \to \tau)\ y\ \Gamma, x : \sigma \vdash M : \tau]$. ■

La demostración se hace por inducción sobre la longitud de la derivación.

Proposición 4 (Tipificación de subtérminos). *Si M tiene un tipo, entonces todo subtérmino de M también tiene un tipo.*

La demostración se hace por inducción en la generación de M.

Proposición 5 (Lema de sustitución). *1.* $\Gamma \vdash M : \sigma \Rightarrow \Gamma[\tau \to \alpha] \vdash M[\tau \to \alpha] : \sigma[\tau \to \alpha]$.

2. Suponga $\Gamma, x : \sigma \vdash M : \tau\ y\ \Gamma \vdash N : \sigma$. *Entonces* $\Gamma \vdash M[N \to x] : \tau$.

Prueba.

1. Por inducción en la derivación de $M : \sigma$ se cumple (1).

2. Por inducción en la generación de $\Gamma, x : \sigma \vdash M : \tau$ se cumple (2).

Proposición 6 (Teorema de la Reducción del Sujeto). *Sea* $M \twoheadrightarrow_\beta M'$. *Entonces*

$$\Gamma \vdash M : \sigma \Rightarrow \Gamma \vdash M' : \sigma$$

Demostración. Por inducción en la generación de $\twoheadrightarrow_\beta$. Supongase $M \equiv (\lambda x.P)Q$ y $M' \equiv P[Q \to x]$. Si

$$\Gamma \vdash (\lambda x.P)Q : \sigma,$$

entonces

$$\Gamma \vdash (\lambda x.P) : (\tau \to \sigma) \text{ y } \Gamma \vdash Q : \tau.$$

Además

$$\Gamma, x : \tau \vdash P : \sigma \text{ y } \Gamma \vdash Q : \tau$$

y por el lema anterior se tiene que

$$\Gamma \vdash P[Q \to x] : \sigma \quad \blacksquare$$

Proposición 7 (Teorema de Unicidad de Tipos). *1. Supongase* $\Gamma \vdash M : \sigma$ *y* $\Gamma \vdash M : \sigma'$. *Entonces* $\sigma \equiv \sigma'$.

2. Supongase $\Gamma \vdash M : \sigma, \Gamma \vdash M' : \sigma'$ *y* $M =_\beta M'$. *Entonces* $\sigma \equiv \sigma'$

Prueba.

1. Por inducción en la estructura de M se cumple (1)

2. Por el Teorema Church-Rosser para $\Lambda_{\mathbb{T}}$, la proposición anterior y (1). $\blacksquare$

Capítulo 2

Lenguaje PCF

En este capítulo se estudian las principales características del Lenguaje PCF. La primera sección presenta la sintáxis, en la siguiente se presenta su semántica desde distintas técnicas, como son la semántica axiomática, denotacional y operacional, con el objeto de establecer parámetros para su comparación con la Semántica de Acción. A continuación se exponen las estrategias de reducción para PCF, las cuales están basadas en la ya expuesta reducción β.

El lenguaje funcional para *Programar Funciones Computables* llamado PCF fue creado por Scott [HO95, Mit96], y es escencialmente el cálculo λ tipificado al estílo Church, aumentado por operadores básicos aritméticos, condicionales y de punto fijo [HO95].

2.1 Sintáxis

Ya que PCF está basado en el sistema Church, cada expresión debe tener un tipo único. Los tipos son $nat, bool, function$ y *producto cartesiano* cuyos valores son los números naturales, los valores booleanos, las funciones y los pares, respectivamente. El *producto cartesiano* de tipos σ y τ es escrito $\sigma \times \tau$. La notación para el tipo de una función con dominio σ y rango τ es $\sigma \rightarrow \tau$.

La sintáxis abstracta de un lenguaje tiene que ver con la estructura composicional de las frases de los programas. A continuación se describe la sintáxis abstracta de PCF [Mit96] usando una gramática BNF:

$$
\begin{array}{ll}
\langle \sigma\_exp \rangle & ::= \langle \sigma\_var \rangle | \text{if } \langle bool\_exp \rangle \text{ then } \langle \sigma\_exp \rangle \text{ else } \langle \sigma\_exp \rangle | \\
& \quad \langle \sigma\_application \rangle | \langle \sigma\_projection \rangle | \langle \sigma\_fixed\_point \rangle \\
\langle \sigma\_application \rangle & ::= \langle \tau \to \sigma\_exp \rangle \langle \tau\_exp \rangle \\
\langle \sigma\_projection \rangle & ::= \mathbf{Proj}_1 \langle \sigma \times \tau\_exp \rangle | \mathbf{Proj}_2 \langle \tau \times \sigma\_exp \rangle \\
\langle \sigma\_fixed\_point \rangle & ::= fix_\sigma \langle \sigma \to \sigma\_exp \rangle \\
\langle \sigma \to \tau\_exp \rangle & ::= \lambda x : \sigma . \langle \tau\_exp \rangle \\
\langle \sigma \times \tau\_exp \rangle & ::= \langle \langle \sigma\_exp \rangle, \langle \tau\_exp \rangle \rangle \\
\langle bool\_exp \rangle & ::= true | false | Eq? \langle nat\_exp \rangle \langle nat\_exp \rangle \\
\langle nat\_exp \rangle & ::= 0 | 1 | 2 | \ldots | \langle nat\_exp \rangle + \langle nat\_exp \rangle
\end{array}
$$

Dada la naturaleza del cálculo λ es posible hacer extensiones a la sintáxis de PCF sin alterar la escencia del mismo.

La recursión en PCF es posible gracias al siguiente axioma [Mit96]:

$$fix_\sigma = \lambda f : \sigma \to \sigma . f(fix_\sigma f).$$

2.2 Semántica

La mayoria de los lenguajes de programación tienen varias categorias sintácticas [Mit96]. Las dos características que distínguen un *programa* de una forma sintáctica arbitraria son, que los programas no se refieren a variables sin acotar o sin declarar, y que los programas deben tener un tipo o forma apropiada que produce un valor imprimible o un efecto observable.

En PCF, las dos categorías sintácticas son los tipos y términos. Los números naturales *nat* y valores booleanos *bool* son observables, pero los valores $nat \to nat$ no lo son, esto es porque todas las funciones recursivas son definibles en PCF [Mit96].

Sea τ un tipo observable en PCF, entonces τ es *nat* ó en caso contrario es *bool*. Un programa PCF es un término bien formado, cerrado y de tipo observable. Un *resultado* es una forma normal cerrada de tipo observable.

La semántica de programas es una relación entre los programas y sus resultados [Mit96].

2.2.1 Semántica axiomática

En forma general, una semántica axiomática consiste de un sistema de pruebas para deducir la propiedades de los programas. Estas propiedades pueden ser ecuaciones, aserciones acerca de salidas de programas con determinadas entradas u otras propiedades. La semántica axiomática de PCF está basada en ecuaciones [Mit96], y se presenta a continuación.

Ecuaciones del sistema de pruebas para PCF

Axiomas
Igualdad

(ref) $M = M$

Tipos *nat* y *bool*

(add) $0 + 0 = 0, 0 + 1 = 1, \ldots, 3 + 5 = 8, \ldots$

($Eq?$) $Eq?n\, n = true, Eq?n\, m = false$ (m,n numerales distintos)

(cond) if $true$ then M else $N = M$, if $false$ then M else $N = N$

Pares

(proj) $\mathbf{Proj}_1\langle M, N\rangle = M \;\; \mathbf{Proj}_2\langle M, N\rangle = N$

(sp) $\langle \mathbf{Proj}_1 P, \mathbf{Proj}_2 P\rangle = P$

Binding

(α) $\lambda x : \sigma.M = \lambda y : \sigma.\, [y \to x]\, M$ y acotada en M

Funciones

(β) $(\lambda x : \sigma.M)N = [N \to x]\, M$

(η) $\lambda x : \sigma.M\; x = M$, x acotada en M

Recursión

(fix) $fix_\sigma = \lambda f : \sigma \to \sigma.f(fix_\sigma f)$

Reglas de Inferencia
Equivalencia

$$(sim),(tran) \quad \frac{M = N}{N = M} \qquad \frac{M = N, N = P}{M = P}$$

Congruencia Tipos *nat* y *bool*

$$\frac{M = N, P = Q}{M + P = N + Q} \qquad \frac{M = N, P = Q}{Eq?M\ P = Eq?N\ Q}$$

$$\frac{M_1 = M_2, N_1 = N_2, P_1 = P_2}{\text{if } M_1 \text{ then } N_1 \text{ else } P_1 = \text{if } M_2 \text{ then } N_2 \text{ else } P_2}$$

Pares

$$\frac{M = N}{\mathbf{Proj}_i M = \mathbf{Proj}_i N} \qquad \frac{M = N, P = Q}{\langle M, P\rangle = \langle N, Q\rangle}$$

Funciones

$$\frac{M = N}{\lambda x : \sigma.M = \lambda x : \sigma.N} \qquad \frac{M = N, P = Q}{M\ P = N\ Q}$$

Si $M = N$, se escribe $M =_{ax} N$.

La semántica axiomática fue la primera forma de interpretar el significado de los lenguajes de programación y a pesar de la elegancia y estética de dicha técnica, ésta presenta problemas de indecibilidad [HO95].

2.2.2 Semántica denotacional

No es de competencia de este trabajo el estudio de la extensa teoría de la semántica denotacional, asi que, se presenta un bosquejo de esta teoría con el único fin de establecer parámetros de comparación entre las distíntas semánticas. Lease [Mit96, Ten76] para un estudio más riguroso y formal acerca de la semántica denotacional.

La semántica denotacional de PCF asigna el valor de un número natural ó un valor correspondiente a la no terminación a cada expresion de tipo *nat*, un valor booleano o un valor correspondiente a la no terminación a cada expresión de tipo *bool*, una función matemática a una expresión de tipo función ó un par de valores al tipo *producto cartesiano*. El valor matemático de una

expresión es llamado *denotación*. Si un término tiene variables libres, su denotación generalmente dependerá de los valores asumidos por estas variables.

Para dar denotaciones a los términos, primero se escoge un conjunto de valores para cada tipo. El conjunto de los valores matemáticos de tipo *nat* incluyen todos los números naturales y el símbolo $\bot_{nat}$ representando la no terminación. Este valor es necesario ya que PCF puede representar todas las funciones computables, y en consecuencia no todas estas funciones terminan. El conjunto de denotaciones de tipo *bool* incluye $true, false$ y el símbolo de no terminación $\bot_{bool}$. Los valores matemáticos para el tipo *producto cartesiano* $(\sigma \times \tau)$ son pares ordenados. Los valores matemáticos de tipo $\sigma \to \tau$ son funciones de σ a τ.

Una vez elegido el conjunto de valores para cada tipo, se le asigna significado a los términos por medio de la elección de un *ambiente*, el cual es un mapeo de las variables hacia los valores. Si x es una variables de tipo σ, y η es un ambiente, entonces $\eta(x)$ deber ser el valor matemático de tipo σ. Se define el significado de $[\![M]\!]_\eta$ de término M en el ambiente η inductivamente, en el sentido acostumbrado en la lógica de primer orden. Específicamente, el significado de $[\![x]\!]_\eta$ el valor dado a la variable x por el ambiente llamado $\eta(x)$. El significado de una aplicación $[\![MN]\!]$ es obtenido por la aplicación de la función $[\![M]\!]_\eta$ que denota a M al argumento $[\![N]\!]_\eta$ que denota a N. Una propiedad importante es que el significado del término de tipo σ siempre va a tener valores matemáticos asociados con su tipo. Por lo tanto, en el caso de un aplicación MN, por ejemplo, las reglas de tipificación garantizan que la denotación de M será un función, y que la denotación de N será un valor en el dominio de M. En este sentido, las reglas de tipificación sintácticas de PCF evitan las posibles complicaciones en la semántica denotacional del lenguaje.

La semántica denotacional es *composicional*, lo cual significa que el significado de cualquier expresión está determinado por el significado de sus subexpresiones. Por ejemplo:

$$[\![\text{if } B \text{ then } M \text{ else } N]\!]_\eta = \begin{cases} [\![M]\!]_\eta & \text{si } [\![B]\!]_\eta \text{ es } \mathbf{true} \\ [\![N]\!]_\eta & \text{si } [\![B]\!]_\eta \text{ es } \mathbf{false} \\ \bot & \text{en otro caso} \end{cases}$$

donde $\bot$ representa cuando la evaluación no termina. Una consecuencia inmediata de la forma composicional es que si B', M' y N' tienen las misma

denotaciones que B, M y N, respectivamente, entonces

$$[\![\text{if } B \text{ then } M \text{ else } N]\!]_\eta = [\![\text{if } B' \text{ then } M' \text{ else } N']\!]_\eta$$

La razón por la cual estas dos expresiones son equivalentes es porque el significado de la expresión condicional if B then M else N no depende de factores como la forma sintáctica de B, M y N, sino de sus significado semántico.

Si M produce N a través de la semántica denotacional, se escribe $M =_{den} N$.

Proposición 8 ([Mit96]). *Sea M un término de PCF. Si $M =_{den} N$, entonces $M =_{ax} N$.*

2.2.3 Semántica operacional

Una semántica operacional puede ser dada en varias formas. La representación matemática más común son los sistemas de pruebas, ya sea para deducir un resultado final de evaluación ó para transformar una evaluación a través de una secuencia de pasos. Una alternativa que puede proveer un idea más clara en la implementación práctica, es la definición de una máquina abstracta, la cual es una computadora teórica que evalúa programas a través de una serie de máquinas de estados. Las representaciones más prácticas de la semántica operacional son los compiladores y los intérpretes simbólicos.

Axiomas de reducción para PCF

Tipos *nat* y *bool*

(add) $0 + 0 \to 0, 0 + 1 \to 1, \ldots, 3 + 5 \to 8, \ldots$

($Eq?$) $Eq?n\, n \to true, Eq?n\, m \to false$ n, m numerales distintos

(cond) if $true$ then M else $N \to M$, if $false$ then M else $N \to N$

Pares $\sigma \times \tau$

(proj) $\mathbf{Proj}_1\langle M, N\rangle \to M \quad \mathbf{Proj}_2\langle M, N\rangle \to N$

Renombramiento de variables acotadas

(α) $\lambda x : \sigma M = \lambda y : \sigma.\, [y \to x]\, M,$ con y acotada en M

Funciones $(\sigma \to \tau)$

$$(\beta)\ (\lambda x : \sigma.M)N \to [N \to x]\, M$$

Recursión

$$(fix)\ fix_\sigma \to \lambda f : \sigma \to \sigma.f(fix_\sigma f)$$

Los axiomas de reducción están escritos con el símbolo $\to$ en lugar de $=$, para dar énfasis a la dirección de la reducción. Intuitivamente se dice que, $M \to N$ significa que con una evaluación en un sólo paso, la expresión M puede ser transformada a la expresión N. Se define una *función parcial de evaluación* en el sistema de reducción por $eval(M) = N$ si M puede ser reducida a la forma normal N en cero o más pasos.

Si M produce N aplicando cualquiera de los axiomas de reducción, se escribe $M =_{op} N$.

Proposición 9 ([Mit96]). *Sea M un término de PCF. Si $M =_{op} N$, entonces $M =_{den} N$.*

2.3 Reducción

Definición 22. *Sea PCF el conjunto de términos de PCF.*

Definición 23. *Sea A un conjuto de reglas de reducción, sea $a \in A$ una regla de reducción y $M \in PCF$. Si M se reduce a N aplicando una sola vez la regla de reducción a se escribe $M \to_a N$ o $M \to N$. Nótese que $N \in PCF$ también.*

Definición 24. *Sea $M \in PCF$. Si ($M \to M'$ y $M' \to N$) o $M \twoheadrightarrow_\alpha N$, entonces $M \twoheadrightarrow N$.*

PCF hereda [Mit96] del cálculo λ:

- Confluencia.
- Consistencia.

Una estrategia de reducción es una función parcial F de términos a términos, con la propiedad de que si $F(M) = N$, entonces $M \to N$. Esta función es llamada *estrategia* ya que la función puede ser usada escogiendo algúna de las varias reducciónes posibles. Para cualquier estrategia F se define una función parcial de evaluación $eval_F : PCF \to PCF$ en las expresiones PCF.

Definición 25. *Sean $F : PCF \to PCF$ una función parcial y $M \in PCF$. Si $F(M) = N \Rightarrow M \to N$.*

Definición 26. *Sea $eval_F : PCF \to PCF$.*

$$eval_F(M) = \begin{cases} M & si\ F(M)\ no\ está\ definida \\ N & si\ F(M) = M'\ y\ eval_F(M') = N \end{cases}$$

La función de evaluación $eval_F$ es el equivalente matemático de un intérprete determinístico que repite pasos de reducción, siguiendo la estrategia F, hasta que esta estrategia ya no puede ser aplicada más.

2.3.1 Reducción más-izquierda

La relación de reducción $\overset{\text{left}}{\to}$ se define [Mit96]:

Axiómas

$$\frac{M \to N}{M \overset{\text{left}}{\to} N} \qquad \text{donde } M \to N \text{ es un axióma de reducción}$$

Reglas para subtérminos
nat y *bool*

$$\frac{M \overset{\text{left}}{\to} M'}{M + N \overset{\text{left}}{\to} M' + N} \qquad \frac{M \overset{\text{left}}{\to} M'}{N + M \overset{\text{left}}{\to} N + M'} \qquad \text{donde } N \text{ está en forma normal}$$

$$\frac{M \overset{\text{lef}}{\to} M'}{Eq?M\ N \overset{\text{left}}{\to} Eq?M'\ N} \qquad \frac{M \overset{\text{lef}}{\to} M'}{Eq?N\ M \overset{\text{left}}{\to} Eq?N\ M'} \qquad \text{donde } N \text{ está en forma normal}$$

$$\frac{M \overset{\text{left}}{\to} M'}{\text{if } M \text{ then } N \text{ else } P \overset{\text{lef}}{\to} \text{if } M' \text{ then } N \text{ else } P}$$

$$\frac{N \overset{\text{left}}{\to} N'}{\text{if } M \text{ then } N \text{ else } P \overset{\text{lef}}{\to} \text{if } M \text{ then } N' \text{ else } P} \quad \text{donde } M \text{ está en forma normal}$$

$$\frac{P \overset{\text{left}}{\to} P'}{\text{if } M \text{ then } N \text{ else } P \overset{\text{lef}}{\to} \text{if } M \text{ then } N \text{ else } P'} \quad \text{donde } M, N \text{ están en forma normal}$$

Pares

$$\frac{M \overset{\text{left}}{\to} M'}{\langle M, N\rangle \overset{\text{lef}}{\to} \langle M', N\rangle} \qquad \frac{N \overset{\text{left}}{\to} N'}{\langle M, N\rangle \overset{\text{left}}{\to} \langle M, N'\rangle} \quad \text{donde } M \text{ está en forma normal}$$

$$\frac{M \overset{\text{left}}{\to} M'}{\mathbf{Proj}_i M \overset{\text{left}}{\to} \mathbf{Proj}_i M'}$$

Funciones

$$\frac{M \overset{\text{left}}{\to} M'}{M\ N \overset{\text{left}}{\to} M'\ N} \qquad \frac{N \overset{\text{left}}{\to} N'}{M\ N \overset{\text{left}}{\to} M\ N'} \quad \text{donde } M \text{ está en forma normal}$$

$$\frac{M \overset{\text{left}}{\to} M'}{\lambda x : \sigma.M \overset{\text{left}}{\to} \lambda x : \sigma.M'}$$

Proposición 10 ([Mit96]). *Sean $M, N \in PCF$ y N en forma normal.*

$(M \overset{\text{left}}{\to} N) \Leftrightarrow (M \twoheadrightarrow N)$.

Proposición 11 ([Mit96]). *Sean $M, N \in PCF$. y N en forma normal.*

$(eval_{\text{left}}(M) = N) \Leftrightarrow (M \twoheadrightarrow N)$

Como su nombre lo indica, la reducción más-izquierda, toma como orden de reducción las subexpresiones que están más a la izquierda de la expresión en cuestión. Es importante resaltar que la reducción más-izquierda termina de aplicarse hasta que se llega a una forma normal, en caso de haberla.

2.3.2 Reducción floja

Las proposiciones 10 y 11 sólo contemplan términos cerrados de tipo observable. Si sólo contemplamos términos de tipo observable para las reducciones, la estrategia resultante es la reducción floja, la cual omite la reducción de muchos subtérminos.

La relación de reducción $\overset{\text{lazy}}{\to}$ se define [Mit96]:

Axiómas

$$\frac{M \to N}{M \overset{\text{lazy}}{\to} N} \quad M \to N \text{ es un axióma de reducción}$$

Reglas para los subtérminos

nat y *bool*

$$\frac{M \overset{\text{lazy}}{\to} M'}{M + N \overset{\text{lazy}}{\to} M' + N} \qquad \frac{M \overset{\text{lazy}}{\to} M'}{n + M \overset{\text{lazy}}{\to} n + M'} \quad \text{donde } n \text{ es un numeral}$$

$$\frac{M \overset{\text{lazy}}{\to} M'}{Eq?M\ N \overset{\text{lazy}}{\to} Eq?M'\ N} \qquad \frac{M \overset{\text{lazy}}{\to} M'}{Eq?n\ M \overset{\text{lazy}}{\to} Eq?n\ M'} \quad \text{donde } n \text{ es un numeral}$$

$$\frac{M \overset{\text{lazy}}{\to} M'}{\text{if } M \text{ then } N \text{ else } P \overset{\text{lazy}}{\to} \text{if } M' \text{ then } N \text{ else } P}$$

Pares

$$\frac{M \overset{\text{lazy}}{\to} M'}{\mathbf{Proj}_i M \overset{\text{lazy}}{\to} \mathbf{Proj}_i M'}$$

Funciones

$$\frac{M \overset{\text{lazy}}{\to} M'}{M\ N \overset{\text{lazy}}{\to} M'\ N}$$

Proposición 12 ([Mit96]). *Sea $M \in PCF$ y ($M \neq \lambda x : \sigma M_1$ o $M \neq \langle M_1, M_2 \rangle$).*

$(M \overset{\text{lazy}}{\to} N) \Leftrightarrow (\overset{\text{left}}{\to})$.

Corolario 2 ([Mit96]). *Sean P un programa PCF y R un resultado(forma normal cerrada del mismo tipo).*

$(P \overset{\text{lazy}}{\twoheadrightarrow} R) \Leftrightarrow (P \overset{\text{left}}{\twoheadrightarrow} R)$

2.3.3 Reducción voraz

Como la reducción floja, la reducción voraz sólo produce numerales o booleanos constantes de un programa completo(cerrado), y no produce formas normales ó términos abiertos completamente reducidos que pueden tener variables libres.

V es un valor si V es una constante, una variable, una abstracción lambda ó un par de valores.

Se define la función parcial $\overset{\text{eager}}{\to}$ con domino y contradominio en PCF:

Axiómas

$(\lambda x : \sigma M)V \overset{\text{eager}}{\to} [V/x]M$ donde V es un valor

$\mathbf{Proj}_i\langle V_1, V_2\rangle \overset{\text{eager}}{\to} V_i$ donde $i = 1, 2$ y V_1, V_2 son valores,

$fix_{\lambda\to\tau}V \overset{\text{eager}}{\to} V(delay_{\sigma\to\tau}[fix_{\sigma\to\tau}V])$ donde V es un valor

$0 + 0 \overset{\text{eager}}{\to} 0, 0 + 1 \overset{\text{eager}}{\to} 1, \ldots, 3 + 5 \overset{\text{eager}}{\to} 8, \ldots$

$Eq?nn \overset{\text{eager}}{\to} true, Eq?nm \overset{\text{eager}}{\to} false$ donde n, m son numerales distintos

if $true$ then M else $N \overset{\text{eager}}{\to} M$, if $false$ then M else $N \overset{\text{eager}}{\to} N$

Reglas para los subtérminos

nat

$$\frac{M \overset{\text{eager}}{\to} M'}{M + N \overset{\text{eager}}{\to} M' + N} \qquad \frac{M \overset{\text{eager}}{\to} M'}{n + M \overset{\text{eager}}{\to} n + M'} \quad \text{donde } n \text{ es un numeral}$$

bool

$$\frac{M \overset{\text{eager}}{\to} M'}{Eq?M\ N \overset{\text{eager}}{\to} Eq?M'\ N} \qquad \frac{M \overset{\text{eager}}{\to} M'}{Eq?n\ M \overset{\text{eager}}{\to} Eq?n\ M'} \quad \text{donde } n \text{ es un numeral}$$

$$\frac{M \overset{\text{eager}}{\to} M'}{\text{if } M \text{ then } N \text{ else } P \overset{\text{eager}}{\to} \text{if } M' \text{ then } N \text{ else } P}$$

Pares

$$\frac{M \overset{\text{eager}}{\to} M'}{\langle M, N\rangle \overset{\text{eager}}{\to} \langle M', N\rangle} \qquad \frac{N \overset{\text{eager}}{\to} N'}{\langle V, N\rangle \overset{\text{eager}}{\to} \langle V, N'\rangle} \quad \text{donde } V \text{ es un valor}$$

$$\frac{M \overset{\text{eager}}{\rightarrow} M'}{\mathbf{Proj}_i M \overset{\text{eager}}{\rightarrow} \mathbf{Proj}_i M'}$$

Funciones

$$\frac{M \overset{\text{eager}}{\rightarrow} M'}{M\ N \overset{\text{eager}}{\rightarrow} M'\ N} \qquad \frac{N \overset{\text{eager}}{\rightarrow} N'}{V\ N \overset{\text{eager}}{\rightarrow} V\ N'} \quad V \text{ es un valor}$$

con

$$delay_{\sigma\to\tau}\,[M] \overset{\text{def}}{=} \lambda x : \sigma . M x \qquad x \text{ no libre en } M_{\sigma\to\tau}$$

Ya que la función $\overset{\text{eager}}{\rightarrow}$ elige un término sólo si éste no es un valor, entonces se define:

$$eval_V(M) = \begin{cases} M & \text{si } M \text{ es un valor} \\ N & \text{si } M \overset{\text{eager}}{\rightarrow} M' \text{ y } eval_V(M') = N \end{cases}$$

Existen dos razones para implementar la reducción eager en lugar de la más-izquierda en la práctica [Mit96]. La primera es que incluso para lenguajes puramente funcionales como PCF, la implementación de la reducción más-izquierda es menos eficiente. La razón de esta ineficiencia es que cuando un argumento como $f\,x$ tiene que ser pasado a una función g, es necesario pasar un apuntador al código para f guardarlo en un registro de ambiente léxico apropiado. Como resultado, existe un costo elevado en la implementación del llamado de funciones. Es más simple llamar a f con el argumento x e inmediatamente pasar el resultado de esta evaluación a g. La segunda razón es que la implementación de la evaluación más-izquierda conlleva efectos secundarios. Como es ejemplificado por los muchos artificios utilizados en Algol 60, la combinación de la evaluación más-izquierda y la asignación es con frecuencia muy confusa. Además de los efectos secundarios, la evalución más-izquierda no coincide con la evaluación no-determinista y paralela, ya que el orden en que la asignaciones a una variable son hechas generalmente afectan la salida del programa. No hay garantía de que en diferentes ordenes de evaluación resulten en la misma salida. Ya que la mayoria de los lenguajes hacen uso de la asignación, muchas de las ventajas de las reducciones más-izquierda ó floja son perdídas.

Capítulo 3

Semántica de Acción

La semántica de acción (AS) fue creada por Mosses [Mos96], con el fin de resolver los problemas pragmáticos de la semántica denotacional [SK95]. AS utiliza ecuaciones semánticas para dar definiciones inductivas de funciones composicionales semánticas que mapean árboles sintácticos a entidades semánticas.

Las entidades semánticas utilizadas en AS son: acciones, datos y productores. Las acciones representan el comportamiento computacional de los programas. La ejecución de una acción, la cual puede ser parte de otra acción, sólo puede tener uno de los siguientes estados finales [Mos96]:

- Completes: correspondiente a la terminación normal;
- Escapes: correspondiente a una terminación no normal;
- Fails: correspondiente al abandono de la actual alternativa;
- Diverges: correspondiente a la no terminación.

Dependiendo de la clase de información procesada, las acciones son clasificadas en distintas facetas:

- Faceta funcional: cuando la acción procesa información temporal;
- Faceta declarativa: cuando la acción procesa información de ámbito;
- Faceta básica: cuando la acción especifica flujos de control.

Los lenguajes puramente funcionales solamente manejan las facetas básica, funcional y declarativa, pero exísten otras [Mos96, SK95] presentes en los distíntos paradigmas de programación tales como:

- Faceta imperativa: cuando la acción procesa información estable;
- Faceta comunicativa: cuando la acción procesa información permanente.

La información procesada por las acciones es llamada Datos, y es clasificada [Mos96] de la siguiente forma:

- Temporal: tuplas de datos, correspondientes a resultados intermedios;
- de ámbito: mapeo de tokens a datos, correspondientes a la tabla de símbolos;
- Estable: datos almacenados en memoria, correspondientes a los valores asignados a las variables;
- Permanente: datos comunicados entre acciones distribuidas.

Los productores son entidades que pueden ser evaluadas para producir datos durante la ejecución de las acciones.

3.1 Semántica de acción de PCF

En esta sección se presenta una descripción completa de PCF por medio de AS. Una descripción AS de una lenguaje de programación está dividida en tres partes: sintaxis abstracta, funciones semánticas y entidades semánticas.

3.1.1 Sintaxis abstracta

La estructura de PCF está especificada por una gramática libre de contexto como usualmente lo es.

$$
\begin{aligned}
\text{Exp} \quad &= \text{Variable} \\
&\mid [\![\text{“if”} \ \text{Bool\_exp} \ \text{“then”} \ \text{Exp} \ \text{“else”} \ \text{Exp}]\!] \\
&\mid \text{Application} \\
&\mid \text{Projection} \\
&\mid \text{Fixed\_point}
\end{aligned}
$$

Esta gramática indica que una expresión en PCF consiste de algúna de las siguientes opciones:

- Una variable;
- Una expresión condicional;
- La aplicación de una función;
- La proyección de un par;
- Un operador de punto fijo.

Las otras ecuaciones tienen el mismo sentido que las ecuaciones para la sintaxis abstracta presentadas en el Capítulo 2. El nivel de abstracción de la notación de acciones permite no ser tan explícitos acerca de los tipos de las expresiones. La sintaxis abstracta completa está descrita en la Tabla 3.1.

3.1.2 Funciones semánticas

Las funciones semánticas representan el significado del lenguaje de programación. Cada función semántica mapea la sintáxis abstracta de un programa a la acción que representa su significado. Ya que AS es composicional [Mos96] la semántica de un programa compuestro de frases, está determinada en terminos de la semántica de sus subfrases.

$$evaluate\_ :: Exp \rightarrow action[\text{giving a value} \mid \text{diverging} \mid \text{fails}]$$

La función semántica **evaluate_** indíca que para cada árbol sintáctico abstracto E en el domino de la sintáxis abstracta **Exp**, la entidad semántica **evaluate** E es una acción que da un valor, diverge o falla. El valor puede ser un número natural o un valor booleano. Ya que todas las funciones recursivas son $\lambda-$definibles [Bar93, Cut80], la evaluación de una función puede diverger. La función **evaluate _** puede fallar cuando una expresión no pertenece al lenguaje.

Tabla 3.1: Sintaxis abstracta de PCF

module: Sintaxis abstracta. gramar:

Exp	=	Variable
	|	⟦ "if" Bool_exp "then" Exp "else" Exp⟧
	|	Application
	|	Projection
	|	Fixed_point
Application	=	⟦ Function Exp⟧
Projection	=	⟦"$Proj_1$("Pair")"⟧
	|	⟦"$Proj_2$("Pair")"⟧
Fixed_point	=	⟦"Fix"Function⟧
Fix	=	⟦"λ"Variable":"Type"."Variable"(Fix"Variable")"⟧
Function	=	⟦"λ"Variable":"Type"."Exp⟧
Pair	=	⟦"("Exp","Exp")"⟧
Type	=	"Natural"
	|	"Truth-value"
Bool_exp	=	"True"
	|	"False"
	|	⟦"Eq?" Nat_exp Nat_exp⟧
Nat_exp	=	Numeral
	|	⟦Nat_exp "+" Nat_exp⟧
Numeral	=	Digit_1
	|	⟦Digit_1 Digit_2⟧
Digit_1	=	"1" | "2" | "3" | "5" | "6" | "7" | "8" | "9"
Digit_2	=	"0"
	|	⟦"0" Digit_2⟧
	|	Numeral

endgrammar. closed. endmodule: Sintaxis Abstracta.

Tabla 3.2: Semántica de acción de PCF

module: Funciones semánticas. needs: Sintaxis abstracta, Entidades semánticas
introduces: *evaluate_*
variables: $E, E_1, E_2 : Exp$;.

*evaluate*_ :: $Expr \rightarrow action$[giving a value]

evaluate ⟦"if" E_1:Bool_exp "then" E_2:Exp "else" E_3:Exp⟧ =
  evaluate E_1 then
  | | check(the given truth value is true) then
  | | | evaluate E_2
  | or
  | | check (the given truth value is false) then
  | | | evaluate E_3.
evaluate $[\![E_1 : Function \ \ E_2 : Expresion]\!] =$
  | evaluate E_2
  then
  | enact the application of E_1 to the given value
evaluate ⟦"Proj_1("E:Par")"⟧ =
  | evaluate "(" E ")"
  then
  | give the first value
evaluate ⟦"Proj_2("E:Par")"⟧ =
  | evaluate "(" E ")"
  then
  | give the second value
evaluate ⟦"(" E_1:Exp "," E_2:Exp ")"⟧ =
  | evaluate E_1 and evaluate E_2
  then
  | give the value ♯1 and give the value ♯2
evaluate ⟦"True"⟧ = give the value true
evaluate ⟦"False"⟧ = give the value false
evaluate ⟦"Eq?" E_1:Nat_exp E_2:Nat_exp⟧ =
  | evaluate E_1 and evaluate E_2 then
  | | check(the given value ♯1 is the given value ♯2) then
  | | | give the value true
  | or
  | | check(not(the given value ♯1 is the given value ♯2)) then
  | | | give the value false
evaluate $N : Numeral =$ give the natural number of N.
evaluate$[\![E_1 : Nat\_exp$ "+" $E_2 : Nat\_exp]\!] =$

La definición AS de la operación suma en PCF es:

evaluate $N : Numeral =$ give the natural number of N.
evaluate$[\![E_1 : Nat\_exp$ "+" $E_2 : Nat\_exp]\!] =$
| evaluate E_1 and evaluate E_2
then
| give the sum of
| (the given value ♯1, the given value ♯2).

$[\![E_1 : Nat\_exp$ "+" $E_2 : Nat\_exp]\!]$ significa que: cuando E_1 y E_2 son evaluados, producen los valores v_1 y v_2 respectivamente, y luego la suma de estos valores es dada como resultado. Nótese lo parecida que esta explicación informal es con respecto a la descrita antes, la cual es completamente formal [Wat99]. **give** Y es una acción de faceta funcional que da el valor producido por Y. El combinador de acción, **and**, es un combinador básico que representa un orden de la ejecuación, dependiente de la implementación. Ya que no hay interferencia entre las dos subacciones de **and** en la función anterior, se garantiza que el orden de evaluación no es relevante. El combinador **and** hace tuplas de valores temporales de las subacciones. El combinador **then** pasa datos temporales de la primera subacción a la segunda subacción. **give** d produce un valor temporal sólo si el valor dado es de tipo d. Nótese que *the* y of solo son usados para hacer mas amigable la notación de acción.

evaluate $[\![$"if" E_1:Bool_exp "then" E_2:Exp "else" E_3:Exp$]\!] =$
evaluate E_1 then
| | check(the given truth value is true) then
| | | evaluate E_2
| or
| | check (the given truth value is false) then
| | | evaluate E_3.

check Y termina cuando Y produce el valor *true* y falla cuando Y produce el valor *false*. El combinador de acción **or** representa una elección determinista, ya que una de las dos subacciones siempre falla. De acuerdo con la especificación de PCF en [Mit96], la función condicional falla cuando $E2$ y E_3 tienen distintos tipos.

Tabla 3.3: Entidades Semánticas

module: Entidades Semánticas
includes: Notación de Acción
introduces: Value

$value \ = \ natural \mid truth{-}value$

endmodule: Entidades Semánticas.

$evaluate \ [\![E_1 : Function \ \ E_2 : Expresion]\!] =$
| evaluate E_2
then
| enact the application of E_1 to the given value

Esta función falla cuando el valor obtenido de la evaluación de E_2 no tiene el tipo del dominio de la función E_1. **enact** _ el valor que le dan, a la función E_1 como un argumento. Ya que las funciones PCF son una abstracción λ [HO95, Mit96], la aplicación de funciones en PCF, está basada en las reglas de reducción del λ, expuestas en el Capítulo 1. La lista completa de las funciones semánticas de PCF están en la Tabla 3.2, y están en el mismo sentido que las ya descritas.

3.1.3 Entidades semánticas

Las entidades semánticas son de libre elección en la especificación general de la **notación de acciones** [Mos96, Wat99]. El uso de *includes:* en la Tabla 3.3, en lugar de *needs:* significa que la notación, además de ser importada, es también exportada.

Conclusión

A pesar de lo elegante y poderosa que es la teoría de la semántica denotacional, tiene demasiados problemas pragmáticos [Mos96, Wat99] en su aplicación a lenguajes del estilo de Pascal, C, Java, etc. Estos problemas ya se han observado en las descripciones de pequeños e ilustrativos lenguajes, propuestos en textos pedagógicos sobre la semántica denotacional [Mos96, SK95, Wat99]. Es de especial mención, los problemas surgidos al hacer cambios o extensiones al lenguaje descrito, lo cual requiere, muchos cambios en la definición de la semántica, en ocasiones, las ecuaciones semánticas requieren su total reformulación. El desarrolló de AS fue impulsado para la solución de este tipo de problemas, lo cual, junto con las características compartidas con la semántica operacional, hace de AS la alternativa más adecuada en la descripción y desarrollo de lenguajes prácticos (C, Java, etc.).

Es común que los implementadores de los lenguajes de programación tengan problemas con la interpretación de la notación obscura de las descripciones de lenguajes, ya sea por medio de la semántica axiomática, operacional o denotacional, dada su naturaleza matemática. Se ha mostrado en el Capítulo 3, que la notación de acción es de notable similitud al lenguaje natural (ingles), lo cual representa una buena ventaja para los implementadores. A pesar de la fácil interpretación de la semántica de acción, no se pierde formalidad [Wat99]. Tal formalidad, así como los fundamentos matemáticos de AS, están soportados por una fuerte teoría algebráica de acciones [Las99]. Otra ventaja de la notación tan natural de AS, es que la descripción completa de un lenguaje se puede considerar como el diseño de un compilador del lenguaje en cuestion, esto último, es gracias a la similitud de la notación de acción con la notación de los lenguajes utilizados generalmente para desarrollar tales compiladores.

El principal objetivo de este trabajo, es el establecimiento de una referen-

cia mas, en el estudio de los fundamentos de los lenguajes de programación, la cual es, la descripción completa del lenguaje PCF mediante la semántica de acción [BL04]. Como se ha mencionado antes, dada la naturaleza de la notación de acción, esta descripción de PCF se considera como el diseño de un interprete de PCF. Se establece la implementación de este diseño, como trabajo futuro, ya que ésta no fue posible, dados los propósitos de este trabajo.

Bibliografía

[Bar93] H.P. Barendregt. *Handook of logic in computer science: Lambda Calculi with Types*, volume II. Oxford University Press, 1993.

[BL04] I. E. Bárcenas and J. Lavalle. An action semantics of the PCF languague. *Proceedings of IX Ibero-American Workshops on Artificial Intelligence*, pages 42–49, 2004.

[Cut80] Nigel Cutland. *Computability: An introduction to recursive function theory*. Cambridge University Press, 1980.

[HO95] J. M. E. Hyland and C. H. L. Ong. Pi-calculus, dialogue game and PCF. *Proceedings of the 7th ACM Conference on Functional Programming ACM Press*, pages 96–107, 1995.

[Las99] S. B. Lassen. An algebra of actions. *Proceedings of the 2nd International Workshop of Action Semantics*, pages 89–109, 1999.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[Mos96] Peter D. Moses. Theory and practice of action semantics. *Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science*, 1996.

[SK95] Kennet Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.

[Ten76] R. D. Tennet. The denotational semantics of programming languages. *Communications of the ACM*, 19:437–453, 1976.

[Wat99] David A. Watt. The static and dynamic semantics of Standard ML. *Proc. 2nd International Workshop on Action Semantics*, pages 155–172, 1999.