Lenguajes de Programación, 2019-I[*]
Nota de clase 7: El cálculo lambda con tipos simples y el lenguaje PCF[**]

Favio E. Miranda Perea Lourdes Del Carmen González Huesca
Facultad de Ciencias UNAM

27 de septiembre de 2018

1. El Cálculo Lambda con Tipos Simples $\lambda^{\rightarrow}$

En esta sección extendemos nuestro lenguaje EAB con un mecanismo para definir funciones. Nuestro lenguaje es esencialmente el lenguaje tipado más simple con un constructor para funciones y se conoce como cálculo lambda con tipos simples, denotado $\lambda^{\rightarrow}$. Este es un sistema de cálculo lambda mucho más débil que el cálculo lambda puro, en particular el sistema de tipos va a prohibir la autoaplicación de expresiones. Por simplicidad sólo tratamos los nuevos constructores de lenguaje correspondientes a la abstracción lambda y a la aplicación. Sin embargo para los ejemplos suponemos disponibles las funciones aritméticas y booleanas.

1.1. Sintaxis de $\lambda^{\rightarrow}$

A la sintaxis concreta agregamos:

$$e ::= \ldots \mid \lambda x : \mathsf{T}.e \mid e\ e$$

Es de importancia observar lo siguiente:

- El tipo del parámetro en una abstracción se da explicitamente, esto facilita la implementación de la semántica estática.
- La abstracción de funciones incorpora a la sintaxis del lenguaje anotaciones de tipos. Esto se conoce como tipificación à la Church.
- Las funciones son anónimas, es decir no tienen nombre.

La sintaxis abstracta se extiende como sigue:

$$t ::= \ldots \mid \texttt{lam}(\mathsf{T}, x.t) \mid \texttt{app}(t, t)$$

[*]Agradecemos la colaboración de Susana Hahn Martín-Lunas en la revisión 2018-I

[**]Estas notas se basan en los libros de Harper y Pierce y en material de Frank Pfenning, Ralf Hinze y Gerwin Klein.

Recordemos que en cualquier caso de ligado se utiliza la α-equivalencia. Es decir, expresiones que difieren exclusivamente en sus variables ligadas se consideran idénticas.

1.2. Semántica Dinámica de $\lambda^{\rightarrow}$

Respecto a la semántica dinámica tenemos las siguientes novedades.

Las abstracciones funcionales se consideran valores:

$$v \quad ::= \quad \ldots \mid \texttt{lam}(\mathsf{T}, x.e)$$

Las nuevas reglas de evaluación son:

$$\frac{e_1 \rightarrow e_1'}{\texttt{app}(e_1, e_2) \rightarrow \texttt{app}(e_1', e_2)}\ (eappi)$$

$$\frac{e_2 \rightarrow e_2'}{\texttt{app}(v, e_2) \rightarrow \texttt{app}(v, e_2')}\ (eappd)$$

$$\frac{}{\texttt{app}(\texttt{lam}(\mathsf{T}, x.e), v) \rightarrow e[x := v]}\ (\beta)$$

En su conjunto las reglas de evaluación modelan la estrategia de llamada por valor, para evaluar una aplicación $e_1 e_2$ es necesario primero reducir e_1 hasta que sea una función explícita, es decir, un valor funcional, para después evaluar el argumento e_2 hasta que sea un valor explícito. Es hasta ese momento cuando se puede ejecutar la aplicación mediante la regla β.

En este caso es muy común usar sintaxis concreta para la evaluación. Por ejemplo, la regla (β) es:

$$\frac{}{(\lambda x : \mathsf{T}.e)v \rightarrow e[x := v]}$$

1.3. Semántica Estática de $\lambda^{\rightarrow}$

Respecto al sistema de tipos, se agrega el constructor de tipos de función

$$\mathsf{T} ::= \ldots \mid \mathsf{T} \rightarrow \mathsf{T}$$

Obsérvese que esta es la primera ocasión en que el sistema de tipos se vuelve infinito, es decir hay una infinidad de tipos. En particular el constructor de tipos función genera tipos para funciones de orden superior, es decir, funciones que pueden recibir funciones como argumentos o devolver funciones como resultados. Por ejemplo:

- $\mathsf{Nat} \rightarrow (\mathsf{Nat} \rightarrow \mathsf{Nat})$ es el tipo de funciones que devuelven una funcion de naturales en naturales para cada natural dado.
- $(\mathsf{Nat} \rightarrow \mathsf{Nat}) \rightarrow \mathsf{Nat}$ es el tipo de funciones que devuelven un natural para cada función de naturales en naturales dada.

- $(\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat})$ es el tipo de funciones que devuelven una función de naturales en naturales a cada función de naturales en naturales dada.

Es de gran importancia observar que los tipos función se asocian a la derecha, es decir,

$$\mathsf{T}_1 \to \mathsf{T}_2 \to \ldots \to \mathsf{T}_{n-1} \to \mathsf{T}_n \text{ significa } \mathsf{T}_1 \to (\mathsf{T}_2 \to \ldots \to (\mathsf{T}_{n-1} \to \mathsf{T}_n) \ldots)$$

Las nuevas reglas de tipado son:

$$\frac{\Gamma, x : \mathsf{T} \vdash e : \mathsf{S}}{\Gamma \vdash \mathtt{lam}(\mathsf{T}, x.e) : \mathsf{T} \to \mathsf{S}} \ (tabs)$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T} \to \mathsf{S} \quad \Gamma \vdash e_2 : \mathsf{T}}{\Gamma \vdash \mathtt{app}(e_1, e_2) : \mathsf{S}} \ (tapp)$$

2. Propiedades importantes de $\lambda^{\to}$

Para terminar las generalidades del cálculo lambda presentamos algunas de sus propiedades relevantes

2.1. Seguridad del lenguaje $\lambda^{\to}$

A continuación damos las herramientas necesarias para probar la necesidad del lenguaje.

Lema 1 (Inversión) *Si* $\Gamma \vdash t : \mathsf{T}$ *entonces*

1. *Si* $t = x$ *entonces* $\Gamma = \Gamma', x : \mathsf{T}$
2. *Si* $t = \lambda x : \mathsf{R}.e$ *entonces* $\mathsf{T} = \mathsf{R} \to \mathsf{S}$ *y* $\Gamma, x : \mathsf{R} \vdash e : \mathsf{S}$
3. *Si* $t = e_1 e_2$ *entonces existe* S *tal que* $\Gamma \vdash e_1 : \mathsf{S} \to \mathsf{T}$ *y* $\Gamma \vdash e_2 : \mathsf{S}$

Demostración. Inmediato por inspección de las reglas de tipado. ⊣

Lema 2 (Sustitución) *Si* $\Gamma, x : \mathsf{T} \vdash t : \mathsf{S}$ *y* $\Gamma \vdash e : \mathsf{T}$ *entonces* $\Gamma \vdash t[x := e] : \mathsf{S}$

Demostración. Inducción sobre la derivación del primer juicio. ⊣

Proposición 1 (Preservación) *Si* $\Gamma \vdash e : \mathsf{T}$ *y* $e \to e'$ *entonces* $\Gamma \vdash e' : \mathsf{T}$

Demostración. Inducción sobre $e \to e'$ ⊣

Lema 3 (Formas canónicas) *Un valor de tipo función es una abstracción lambda. Es decir, si* e *es un valor y* $\Gamma \vdash e : \mathsf{T} \to \mathsf{S}$ *entonces* $e = \lambda x : \mathsf{T}.e'$ *para algunos* x, e' *tales que* $\Gamma, x : \mathsf{T} \vdash e' : \mathsf{S}$

Demostración. Directo. ⊣

Proposición 2 (Progreso) *Si* $\vdash e : \mathsf{T}$ *entonces* e *es un valor o existe* e' *tal que* $e \to e'$

Demostración. Inducción sobre las reglas de tipado ⊣

2.2. Terminación de $\lambda^{\to}$

La propiedad de terminación o normalización fuerte vuelve a ser válida en el caso del cálculo lambda con tipos simples. Esta es la instancia más simple de una técnica importante en el estudio de lenguajes de programación conocida como *terminación basada en tipos* donde la semántica estática garantiza que los programas terminarán adecuadamente.

Proposición 3 (Terminación de $\lambda^{\to}$) *Si* $\vdash e : \mathsf{T}$ *entonces existe un valor* v *tal que* $e \to^{\star} v$.

La demostración de este hecho no es simple y cae fuera de los límites de este curso. Es útil intentar una prueba por inducción sobre $\vdash$ y ver por qué falla.

Obsérvese que la propiedad de progreso no garantiza la terminación pues podría ser que la evaluación progrese indefinidamente sin alcanzar un valor, este es el caso de los lenguajes con recursión general.

3. Inferencia de tipos en $\lambda^{\to}$

El problema de la inferencia de tipos deja ver la gran diferencia entre implementar un lenguaje fuerte o estrictamente tipado[1] (como Pascal o Java, y en nuestro caso el cálculo lambda a la Church, donde las abstracciones son de la forma $\lambda x : \mathsf{T}.e$) y un lenguaje tipado pero sin anotaciones de tipos (como Haskell, ML o el cálculo lambda a la Curry, donde las abstracciones son de la forma $\lambda x.e$) Estos dos estilos generan problemas relacionados con respecto al proceso de revisar si los tipos de un programa dado son correctos:

- Problema de la verificación de tipos[2] (en lenguajes con anotaciones de tipos): dados una expresión e con anotaciones de tipos, un contexto Γ y un tipo T verificar si $\Gamma \vdash e : \mathsf{T}$.
- Problema de la inferencia de tipos (en lenguajes sin anotaciones de tipos): dada una expresión e sin anotaciones de tipos, hallar una expresión e^a con anotaciones de tipos tal que
  - Existen Γ, T tales que $\Gamma \vdash e^a : \mathsf{T}$
  - $erase(e^a) = e$, donde $erase$ es la función que elimina todas las anotaciones de tipos de una expresión.

[1] Es decir, un lenguaje donde el programador está obligado a hacer anotaciones de tipos en el programa
[2] Type checking problem

Si bien los dos problemas están relacionados, el primero es mucho más simple de resolver pues basta analizar sintácticamente a la expresion con anotaciones e para reconstruir una derivación del tipado (basandonos en las propiedades de inversión del tipado). Por lo tanto en lo que sigue nos ocupamos del problema de inferencia en el cálculo lambda $\lambda^{\rightarrow}$ a la Curry, es decir, sin anotaciones de tipos.

Empecemos con algunos ejemplos sencillos

- Si $e =_{def} \lambda x.x + 5$ entonces podemos tomar $e^a =_{def} \lambda x : \mathsf{Nat}\,.x + 5$ y probar $\vdash e^a : \mathsf{Nat} \rightarrow \mathsf{Nat}$
- Si $e =_{def} \lambda x.\,\mathsf{iszero}\,(x * y)$ entonces podemos tomar $e^a =_{def} \lambda x : \mathsf{Nat}\,.\,\mathsf{iszero}\,(x * y)$ y probar $y : \mathsf{Nat} \vdash e^a : \mathsf{Nat} \rightarrow \mathsf{Bool}$
- Si $e =_{def} \lambda x.\,\mathsf{iszero}(x + 2) * z$ entonces podemos tomar $e^a =_{def} \lambda x : \mathsf{Nat}\,.\,\mathsf{iszero}(x + 2) * z$ pero e^a no es tipable.
- Si $e =_{def} \bar{1} =_{def} \lambda s.\lambda z.sz$ podemos tomar $e^a =_{def} \lambda s : \mathsf{T} \rightarrow \mathsf{S}.\lambda z : \mathsf{T}.sz$ y probar

  $$\vdash e^a : (\mathsf{T} \rightarrow \mathsf{S}) \rightarrow \mathsf{T} \rightarrow \mathsf{S}$$

  para cualesquiera tipos T, S, es decir existen una infinidad de formas de definir e^a.

En el último caso cabe preguntarse ¿ que expresión e^a es la más general posible y en tal caso cómo encontrarla ?

A continuación presentamos un algoritmo de inferencia de tipos para la versión a la curry del cálculo $\lambda^{\rightarrow}$ que dada e genera la expresión e^a con el tipo más general posible T, llamado tipo principal para e, en el sentido de que cualquier otro tipo para e^a puede obtenerse mediante sustitución a partir de T.

Definición 1 *La función erase que elimina anotaciones de tipos en una expresión de $\lambda^{\rightarrow}$ se define como sigue:*

$$erase(x) = x$$

$$erase(e_1 e_2) = erase(e_1)\ erase(e_2)$$

$$erase(\lambda x : \mathsf{T}.e) = \lambda x.\, erase(e)$$

3.1. El algoritmo de inferencia $\mathcal{W}$

Para definir el algoritmo de inferencia, llamado algoritmo $\mathcal{W}$, se extiende la categoría de tipos como sigue:

$$\mathsf{T}, \mathsf{S} ::= X \mid \mathsf{Nat} \mid \mathsf{Bool} \mid \mathsf{T} \rightarrow \mathsf{T}$$

donde X denota a una variable de tipo. Aquí suponemos que existe un conjunto infinitonumerable de variables de tipo que por supuesto es ajeno con el conjunto de variables del cálculo lambda.

El algoritmo se implementa mediante una función $\mathcal{W}$ que toma una expresión sin anotaciones de tipo e y devuelve no sólo la expresión e^a anotada sino el juicio de tipado $\Gamma \vdash e^a : \mathsf{T}$ más general posible, en el sentido de que otra derivación de tipos para e se puede obtener como una instancia de sustitución del juicio construido por $\mathcal{W}$.

Definición 2 *La función $\mathcal{W}$ que dado un término sin anotaciones e, devuelve un contexto Γ, un término con anotaciones e^a y un tipo T, tales que su cumple el tipado más general posible $\Gamma \vdash e^a : \mathsf{T}$, se define recursivamente como sigue:*

- *Variables: $\mathcal{W}(x)$ devuelve $x : X \vdash x : X$ donde X es una variable de tipo nueva. Convenimos aquí en denotar a esta variable de tipo nueva con la letra mayúscula correspondiente a la variable dada a menos que ésta ya esté en uso.*
- *Aplicación: $\mathcal{W}(e_1 e_2)$ devuelve $(\Gamma_1 \cup \Gamma_2)\mu \vdash (e_1^a e_2^a)\mu : X\mu$ donde*
  - *$\mathcal{W}(e_1)$ devuelve $\Gamma_1 \vdash e_1^a : \mathsf{T}_1$*
  - *$\mathcal{W}(e_2)$ devuelve $\Gamma_2 \vdash e_2^a : \mathsf{T}_2$*
  - *$\mu = umg(\{\mathsf{S}_1 = \mathsf{S}_2 \mid x : \mathsf{S}_1 \in \Gamma_1,\ x : \mathsf{S}_2 \in \Gamma_2\} \cup \{\mathsf{T}_1 = \mathsf{T}_2 \to X\})$ con X una variable de tipo nueva.*
- *Abstracción: Si $\mathcal{W}(e)$ devuelve $\Gamma \vdash e^a : \mathsf{S}$ entonces*
  - *Si Γ tiene una declaración para x, digamos $x : \mathsf{R} \in \Gamma$, entonces $\mathcal{W}(\lambda x.e)$ devuelve*
    $$\Gamma \setminus \{x : \mathsf{R}\} \vdash \lambda x : \mathsf{R}.e^a : \mathsf{R} \to \mathsf{S}$$
  - *Si Γ no tiene una declaración para x, entonces $\mathcal{W}(\lambda x.e)$ devuelve*
    $$\Gamma \vdash \lambda x : X.e^a : X \to \mathsf{S}$$
    siendo X una variable nueva.

Veamos algunos ejemplos:

- $e = \lambda x.\lambda y.y$ Las llamadas al algoritmo son:
  $$\mathcal{W}(e) \rightsquigarrow \mathcal{W}(\lambda y.y) \rightsquigarrow \mathcal{W}(y)$$
  Calculamos ahora las llamadas en orden inverso:

$\mathcal{W}(y)$	$y : Y \vdash y : Y$
$\mathcal{W}(\lambda y.y)$	$\{y : Y\} \setminus \{y : Y\} \vdash \lambda y : Y.y : Y \to Y$ $\vdash \lambda y : Y.y : Y \to Y$
$\mathcal{W}(\lambda x.\lambda y.y)$	$\vdash \lambda x : X.\lambda y : Y.y : X \to (Y \to Y)$

- $e = xyz$. Las llamadas al algoritmo son:
  $$\mathcal{W}(e) \rightsquigarrow \mathcal{W}(xy), \mathcal{W}(z) \rightsquigarrow \mathcal{W}(x), \mathcal{W}(y), \mathcal{W}(z)$$
  Calculamos ahora las llamadas en orden inverso:

$\mathcal{W}(x)$	$x : X \vdash x : X$
$\mathcal{W}(y)$	$y : Y \vdash y : Y$
$\mathcal{W}(z)$	$z : Z \vdash z : Z$
$\mathcal{W}(xy)$	$\big(\{x : X\} \cup \{y : Y\}\big)\mu \vdash (xy)\mu : V\mu,\ \mu = umg(\varnothing \cup \{X = Y \to V\})$ $\big(\{x : X,\ y : Y\}\big)\mu \vdash (xy)\mu : V\mu,\ \mu = [X := Y \to V]$ $\{x : Y \to V,\ y : Y\} \vdash xy : V$
$\mathcal{W}(xyz)$	$\big(\{x : Y \to V,\ y : Y\} \cup \{z : Z\}\big)\mu \vdash (xyz)\mu : U\mu,\ \mu = umg(\varnothing \cup \{V = Z \to U\})$ $\big(\{x : Y \to V,\ y : Y,\ z : Z\}\big)\mu \vdash (xyz)\mu : U\mu,\ \mu = [V := Z \to U]$ $\{x : Y \to (Z \to U),\ y : Y,\ z : Z\} \vdash xyz : U$

- $e = xx$

  Las llamadas al algoritmo son:

  $$\mathcal{W}(e) \rightsquigarrow \mathcal{W}(x), \mathcal{W}(x)$$

  Calculamos ahora las llamadas en orden inverso:

$\mathcal{W}(x)$	$x : X \vdash x : X$
$\mathcal{W}(x)$	$x : Y \vdash x : Y \quad Y$ nueva
$\mathcal{W}(xx)$	$\big(\{x : X\} \cup \{x : Y\}\big)\mu \vdash (xx)\mu : V\mu,\ \mu = umg(\{X = Y\} \cup \{X = Y \to V\})$ $\big(\{x : X,\ x : Y\}\big)\mu \vdash (xx)\mu : V\mu,\ \mu = umg(\{Y = Y \to V\}) \circ [X := Y]$ Falla, puesto que $\{Y = Y \to V\}$ no es unificable

- $e = \lambda s.\lambda z.sz$ Las llamadas al algoritmo son:

  $$\mathcal{W}(e) \rightsquigarrow \mathcal{W}(\lambda z.sz) \rightsquigarrow \mathcal{W}(sz) \rightsquigarrow \mathcal{W}(s), \mathcal{W}(z)$$

  Calculamos ahora las llamadas en orden inverso:

$\mathcal{W}(s)$	$s : S \vdash s : S$
$\mathcal{W}(z)$	$z : Z \vdash z : Z$
$\mathcal{W}(sz)$	$(\{s : S\} \cup \{z : Z\})\mu \vdash (sz)\mu : Y\mu,\ \mu = umg(\varnothing \cup \{S = Z \to Y\})$ $(\{s : S,\ z : Z\})\mu \vdash (sz)\mu : Y\mu,\ \mu = [S := Z \to Y]$ $\{s : Z \to Y,\ z : Z\} \vdash sz : Y$
$\mathcal{W}(\lambda z.sz)$	$\{s : Z \to Y,\ z : Z\} \setminus \{z : Z\} \vdash \lambda z : Z.sz : Z \to Y$ $\{s : Z \to Y\} \vdash \lambda z : Z.sz : Z \to Y$
$\mathcal{W}(\lambda s.\lambda z.sz)$	$\{s : Z \to Y\} \setminus \{s : Z \to Y\} \vdash \lambda s : Z \to Y.\lambda z : Z.sz : (Z \to Y) \to Z \to Y$ $\vdash \lambda s : Z \to Y.\lambda z : Z.sz : (Z \to Y) \to Z \to Y$

- $e = (xz)(yz)$ Las llamadas al algoritmo son:

  $$\mathcal{W}(e) \rightsquigarrow \mathcal{W}(xz), \mathcal{W}(yz) \rightsquigarrow \mathcal{W}(x), \mathcal{W}(z), \mathcal{W}(y), \mathcal{W}(z)$$

  Calculamos ahora las llamadas en orden inverso:

$\mathcal{W}(x)$	$x : X \vdash x : X$
$\mathcal{W}(z)$	$z : Z \vdash z : Z$
$\mathcal{W}(y)$	$y : Y \vdash y : Y$
$\mathcal{W}(z)$	$z : V \vdash z : V \quad V$ nueva
$\mathcal{W}(xz)$	$\big(\{x : X\} \cup \{z : Z\}\big)\mu \vdash (xz)\mu : U\mu,\ \mu = umg(\varnothing \cup \{X = Z \to U\})$ $\big(\{x : X,\ z : Z\}\big)\mu \vdash (xz)\mu : U\mu,\ \mu = [X := Z \to U]$ $\{x : Z \to U,\ z : Z\} \vdash xz : U$
$\mathcal{W}(yz)$	$\big(\{y : Y\} \cup \{z : V\}\big)\mu \vdash (yz)\mu : W\mu,\ \mu = umg(\varnothing \cup \{Y = V \to W\})$ $\big(\{y : Y,\ z : V\}\big)\mu \vdash (yz)\mu : W\mu,\ \mu = [Y := V \to W]$ $\{y : V \to W,\ z : V\} \vdash yz : W$
$\mathcal{W}((xz)(yz))$	$\big(\{x : Z \to U,\ z : Z\} \cup \{y : V \to W,\ z : V\}\big)\mu \vdash \big((xz)(yz)\big)\mu : T\mu,$ $\mu = umg(\{Z = V\} \cup \{U = W \to T\})$ $\big(\{x : Z \to U,\ z : Z,\ y : V \to W,\ z : V\}\big)\mu \vdash \big((xz)(yz)\big)\mu : T\mu,$ $\mu = umg(\{U = W \to T\}) \circ [Z := V]$ $\big(\{x : Z \to U,\ z : Z,\ y : V \to W,\ z : V\}\big)\mu \vdash \big((xz)(yz)\big)\mu : T\mu,$ $\mu = [U := W \to T] \circ [Z := V] = [U := W \to T,\ Z := V]$ $\{x : V \to (W \to T),\ z : V,\ y : V \to W\} \vdash (xz)(yz) : T$

3.2. Extensión a EAB

El algoritmo $\mathcal{W}$ puede extenderse fácilmente a EAB y a cualquier otra extensión donde se conozca el sistema de tipos. Como ejemplo damos la extensión del condicional booleano:

- $\mathcal{W}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)$ devuelve $(\Gamma_1 \cup \Gamma_2 \cup \Gamma_3)\mu \vdash (\mathsf{if}\ e_1^a\ \mathsf{then}\ e_2^a\ \mathsf{else}\ e_3^a)\mu : \mathsf{T}_2\mu$ donde
  - $\mathcal{W}(e_1)$ devuelve $\Gamma_1 \vdash e_1^a : \mathsf{T}_1$
  - $\mathcal{W}(e_2)$ devuelve $\Gamma_2 \vdash e_2^a : \mathsf{T}_2$
  - $\mathcal{W}(e_3)$ devuelve $\Gamma_3 \vdash e_3^a : \mathsf{T}_3$
  - $$\mu = umg\Big(\{\mathsf{S}_1 = \mathsf{S}_2 \mid x : \mathsf{S}_1 \in \Gamma_i,\ x : \mathsf{S}_2 \in \Gamma_j,\ i \neq j, 1 \leq i, j \leq 3\} \cup \{\mathsf{T}_1 = \mathsf{Bool},\ \mathsf{T}_2 = \mathsf{T}_3\}\Big)$$

4. El lenguaje PCF

El lenguaje PCF (Programming Computable Functions) debido a Scott y Plotkin es un lenguaje funcional núcleo que ha servido, entre otras cosas, como base teórica para investigación en semánticas denotacionales. PCF integra funciones y números naturales usado recursión general como medio para definir expresiones que se refieren a si mismas. En contraste con otros sistemas con recursión limitada, como el sistema T de Gödel, los programas en PCF podrían no terminar durante el proceso de evaluación, lo cual implica en particular que podemos definir funciones parciales. En este caso el programador debe garantizar que sus programas terminan pues el sistema de tipos no asegura la terminación del programa. La ventaja es que resulta posible definir un rango más amplio de funciones en el lenguaje al costo de admitir ciclos infinitos cuando la prueba de terminación de un programa es incorrecta o no se proporciona.

La idea para definir PCF es simple: agregar un operador primitivo de punto fijo fix, similar a Y, del cálculo lambda con tipos simples pero de manera que se pueda definir una semántica estática.

La extensión de la sintaxis es:

- Concreta: $\mathsf{fix}\,(x : \mathsf{T}) \Rightarrow e$
- Abstracta $\mathtt{fix}(\mathsf{T}, x.e)$

La semántica estática es:

$$\frac{\Gamma, x : \mathsf{T} \vdash e : \mathsf{T}}{\Gamma \vdash \mathtt{fix}(\mathsf{T}, x.e) : \mathsf{T}}$$

La regla es simple, obsérvese que estamos asumiendo el mismo tipo que se debe concluir.

La semántica operacional sustituye en x a la misma expresión que se desea evaluar, modelando así la recursión general:

$$\mathtt{fix}(\mathsf{T}, x.e) \to e[x := \mathtt{fix}(\mathsf{T}, x.e)]$$

Esta operación se conoce como el desdoblamiento[3] de la definición recursiva. Obsérvese además que esta instrucción genera código de programa con ciclos infinitos, por lo que se pierde la propiedad de terminación, por ejemplo tenemos

$$\texttt{fix}(\mathsf{Nat}, x.x) \to x[x := \texttt{fix}(\mathsf{Nat}, x.x)] = \texttt{fix}(\mathsf{Nat}, x.x) \to \texttt{fix}(\mathsf{Nat}, x.x) \to \texttt{fix}(\mathsf{Nat}, x.x) \to \ldots$$

Este ejemplo muestra el peligro de usar el operador `fix` de forma arbitraria. Se tiene que $\vdash \texttt{fix}(\mathsf{Nat}, x.x) : \mathsf{Nat}$ por lo que dicha expresión es un número natural distinto a todos los conocidos pues se cicla por sí mismo infinitamente.

4.1. Funciones recursivas en PCF

El operador primitivo `fix` permite definir mediante azúcar sintáctica una definición de funciones con nombre como sigue

$$\mathsf{fun}\ f\,(x : \mathsf{T}) : \mathsf{S}\ \Rightarrow\ e\ =_{def}\ \mathsf{fix}\, f : \mathsf{T} \to \mathsf{S} \Rightarrow\ (\mathsf{fun}(x : \mathsf{T})\ \Rightarrow\ e)$$

De manera que durante el proceso de análisis sintáctico cada declaración de este estilo corresponde al árbol de sintaxis abstracta $\texttt{fix}(\mathsf{T} \to \mathsf{S}, f.\texttt{lam}(\mathsf{T}, x.e))$

Por ejemplo, la función $potd(x) = 2^x$ se define como

$$\mathsf{fun}\ potd\,(x : \mathsf{Nat}) : \mathsf{Nat}\ \Rightarrow\ \big(\mathsf{if}\ \ \mathsf{iszero}\, x\ \mathsf{then}\ 1\ \mathsf{else}\ 2 * potd(x-1)\big)\ =_{def}$$

$$\mathsf{fix}\, potd : \mathsf{Nat} \to \mathsf{Nat} \Rightarrow\ (\mathsf{fun}(x : \mathsf{Nat})\ \Rightarrow\ \big(\mathsf{if}\ \ \mathsf{iszero}\, x\ \mathsf{then}\ 1\ \mathsf{else}\ 2 * potd(x-1)\big))$$

Es importante observar que la definición de funciones con nombre, si bien utiliza el operador `fix`, por si sola no genera programas con ciclos infinitos, es decir una expresión de la forma $\mathsf{fun}\ f\,(x : \mathsf{T}) : \mathsf{S}\ \Rightarrow\ e$ siempre tiene una forma normal, obtenida como sigue:

$$\texttt{fix}(\mathsf{T} \to \mathsf{S}, f.\texttt{lam}(\mathsf{T}, x.e)) \to \texttt{lam}(\mathsf{T}, x.e)[f := \texttt{fix}(\mathsf{T} \to \mathsf{S}, f.\texttt{lam}(\mathsf{T}, x.e))] \not\to$$

Es decir, el desdoblamiento de una función con nombre genera una abstracción lambda, la cual es por definición, un valor y no puede reducirse mas. Por ejemplo la función identidad se define y reduce como sigue:

$$\big(\mathsf{fun}\ id\,(x : \mathsf{Nat}) : \mathsf{Nat}\ \Rightarrow\ x\big) = \texttt{fix}(\mathsf{Nat} \to \mathsf{Nat}, f.\texttt{lam}(\mathsf{Nat}, x.x)) \to \texttt{lam}(\mathsf{Nat}, x.x) = \big(\mathsf{fun}(x : \mathsf{Nat})\ \Rightarrow\ x\big)$$

En este caso la definición de id no es recursiva, por lo que id no aparece en la expresión y el proceso de reducción simplemente elimina el nombre devolviendo la función identidad anónima $\lambda x : \mathsf{Nat}\,.x$.

En contraste una definición mediante el operador de punto fijo Y en el cálculo lambda sin tipos podría ciclarse si se elige una mala estrategia de evaluación. Por supuesto que al aplicar una función con nombre a un argumento particular esta aplicación sí podría generar un ciclo infinito.

[3] En inglés *unfold*

4.2. Expresiones letrec

En algunos lenguajes como SCHEME, existe un mecanismo de definición de funciones recursivas mediante la declaración letrec, que en su versión tipada es

$$\mathsf{letrec}\ f(x:\mathsf{T}):\mathsf{S}\ \Rightarrow\ e_1\ \mathsf{in}\ e_2$$

Como ejemplo tenemos el siguiente código en SCHEME que calcula el factorial de seis:

```
letrec fac (x) =
        if zero? x then 1 else (* x (fac sub1(x)))
       in
        (fac 6)
```

El lector debe verificar que tales declaraciones pueden introducirse a PCF mediante azúcar sintáctica.

5. Seguridad del lenguaje PCF

La prueba de seguridad del sistema es análoga a la del cálculo lambda con tipos simples. Damos aquí los detalles para el caso del nuevo operador `fix`.

Lema 4 (Inversión) *Si* $\Gamma \vdash t : \mathsf{T}$ *entonces*

Si $t = \mathtt{fix}(\mathsf{T}, x.e)$ *entonces* $\Gamma, x : \mathsf{T} \vdash e : \mathsf{T}$

Proposición 4 (Preservación) *Si* $\Gamma \vdash e : \mathsf{T}$ *y* $e \to e'$ *entonces* $\Gamma \vdash e' : \mathsf{T}$

Demostración. Inducción sobre $e \to e'$. Veamos el caso para $\mathtt{fix}(\mathsf{T}, x.e) \to e[x := \mathtt{fix}(\mathsf{T}, x.e)]$. Por hipótesis se tiene $\Gamma \vdash \mathtt{fix}(\mathsf{T}, x.e) : \mathsf{T}$, de donde por el lema de inversión se tiene que $\Gamma, x : \mathsf{T} \vdash e : \mathsf{T}$. Finalmente el lema de sustitución, que sigue siendo válido, nos permite concluir que $\Gamma \vdash e[x := \mathtt{fix}(\mathsf{T}, x.e)] : \mathsf{T}$. ⊣

Proposición 5 (Progreso) *Si* $\vdash e : \mathsf{T}$ *entonces* e *es un valor o existe* e' *tal que* $e \to e'$

Demostración. Inducción sobre las reglas de tipado. Veamos el caso para $e = \mathtt{fix}(\mathsf{T}, x.e)$, en este caso e no es un valor pero si definimos $e' = e[x := \mathtt{fix}(\mathsf{T}, x.e)]$ claramente tenemos $e \to e'$. Aquí no se utiliza la HI, el lector debe verificar que en otros casos ésta es necesaria. ⊣

6. Mini Haskell MinHs

Con los conceptos y formalismos desarrollados hasta ahora ya podemos diseñar un mini lenguaje de programación donde que implemente los conceptos y mecanismos esenciales del paradigma de programación funcional. Este lenguaje, llamado Mini Haskell (MinHs) tiene las siguientes características:

- Lenguaje funcional sin efectos laterales, las funciones son individuos de primera clase.
- La estrategia de evaluación es la llamada-por-valor.
- Lenguaje fuertemente tipado. Los tipos deben ser proporcionados por el programador.

Obsérvese que HASKELL implementa la evaluación perezosa y los tipos son inferidos por el intérprete, al contrario de MinHs .

6.1. Sintaxis concreta

Presentamos aquí la sintaxis concreta de MinHs . Las definiciones de la semántica estática, dinámica, así como la implementación de un intérprete se desarrollarán en las prácticas de laboratorio.

- Variables: $x ::= < identificador >$
- Operadores primitivos: $\otimes ::= + \mid * \mid - \mid = \mid < \mid \mathsf{not} \mid \ldots$
- Tipos $\mathsf{T} ::= \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{T} \to \mathsf{T} \mid (\mathsf{T})$
- Expresiones:

$$
\begin{aligned}
e \quad ::= \quad & x \mid n \mid (e) \mid \otimes(e, \ldots, e) \mid \mathsf{true} \mid \mathsf{false} \\
& \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathtt{let}\ x : \mathsf{T} = e_1\ \mathtt{in}\ e_2\ \mathtt{end} \mid \mathsf{fun}(x : \mathsf{T}) \ \Rightarrow \ e \\
& \mathsf{fun}\ f\,(x : \mathsf{T}) : \mathsf{S} \ \Rightarrow \ e \mid e\,e
\end{aligned}
$$

A. El algoritmo de unificación de Martelli-Montanari

A continuación se describe el algoritmo de unificación de Martelli-Montanari[4]:

- Entrada: un conjunto de ecuaciones $\{s_1 = r_1, \ldots, s_k = r_k\}$ tales que se desea unificar simultaneamente s_i con r_i, $1 \leq i \leq k$.
- Salida: un unificador más general μ tal que $s_i\mu = r_i\mu$ para toda $1 \leq i \leq n$.

Colocar como ecuaciones las expresiones a unificar, escoger una de ellas de manera no determinística que tenga la forma de alguna de las siguientes opciones y realizar la acción correspondiente:

Nombre de la regla	$t_1 = t_2$	Acción
Descomposición DESC	$fs_1 \ldots s_n = ft_1 \ldots t_n$	sustituir por $\{s_i = t_i\}$
Desc. fallida DFALLA	$fs_1 \ldots s_n = gt_1 \ldots t_n$ donde $g \neq f$	falla
Eliminación ELIM	$x = x$	eliminar
Intercambio SWAP	$t = x$ donde t no es una variable	sustituir por la ecuación $x = t$
Sustitución SUST	$x = t$ donde x no figura en t	eliminar $x = t$ y aplicar la sustitución $[x := t]$ a las ecuaciones restantes
Sustitución fallida SFALLA	$x = t$ donde x figura en t y $x \neq t$	falla

[4] "An efficient unification algorithm", ACM Trans. Prog. Lang. and Systems vol. 4, 1982, p.p. 258-282

Este algoritmo termina cuando no se puede llevar a cabo ninguna acción o cuando falla. En caso de éxito se obtiene el conjunto vacío de ecuaciones y el unificador más general se obtiene al componer todas las sustituciones usadas por la regla de sustitución en el orden en que se generaron.

El caso en que se deba unificar a dos constantes iguales a genera la ecuación $a = a$ que por la regla de descomposición se sustituye por el conjunto vacío de ecuaciones, es decir, cualquier ecuación de la forma $a = a$ se elimina. En el caso de una ecuación $a = b$ con dos constantes distintas falla por la regla de descomposición fallida.

Ejemplo A.1 Sean $f^{(2)}, g^{(1)}, h^{(2)}$ y $W = \{fgxhxu,\ fzhfyyz\}$. Mostramos el proceso de ejecución del algoritmo de manera similar al razonamiento de verificación de correctud de argumentos por interpretaciones.

1. $\{fgxhxu = fzhfyyz\}$ Entrada
2. $\{gx = z,\ hxu = hfyyz\}$ DESC,1
3. $\{z = gx,\ hxu = hfyyz\}$ SWAP,2
4. $\{hxu = hfyygx\}$ SUST,3, $[z := gx]$
5. $\{\ x = fyy,\ u = gx\}$ DESC,4
6. $\{u = gfyy\}$ DESC,5, $[x := fyy]$
7. $\varnothing$ DESC,6, $[u := gfyy]$

El unificador se obtiene al componer las sustituciones utilizadas desde el inicio.

$$\begin{array}{rcl} \mu & = & [z := gx][x := fyy][u := gfyy] \\ & = & [z := gfyy, x := fyy][u := gfyy] \\ & = & [z := gfyy, x := fyy, u := gfyy] \end{array}$$

Veamos ahora un ejemplo de conjunto no unificable

Ejemplo A.2 Sean $f^{(3)}, g^{(1)}$ y $W = \{fxyx,\ fygxx\}$.

1. $\{fxyx = fygxx\}$ Entrada
2. $\{x = y,\ y = gx,\ x = x\}$ DESC,1
3. $\{x = y,\ y = gx\}$ ELIM,2
4. $\{y = gy\}$ SUST,3, $[x := y]$
5. X SFALLA,4

Para la aplicación que nos interesa se considera a los tipos del lenguaje como los términos para el algoritmo de unificación, es decir, se tienen las variables de tipo, las constantes de tipo (tipos primitivos como $\mathsf{Nat}, \mathsf{Bool}$) y como único símbolo de función al constructor de tipos $\to$ (binario) . De esta manera la regla de descomposición particular es:

Nombre de la regla	$t_1 = t_2$	Acción
Descomposición DESC	$s_1 \to s_2 = r_1 \to r_2$	sustituir por $\{s_1 = r_1,\ s_2 = r_2\}$