XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

Un Sistema de Tipos Polimórfico para un Lenguaje Funcional con Definiciones de Constantes y Tipos Abstractos

Verónica Gaspes
Alvaro Tasistro *

ESLAI - Argentina
Escuela Superior Latinoamericana de Informática.
Casilla de Correo 9193
(1000) Buenos Aires - Rep. Argentina

Resumen. Se define un lenguaje funcional con definiciones de constantes globales y locales y de tipos abstractos. Se define un sistema de tipos polimórfico para ese lenguaje. La semántica del lenguaje y el sistema de tipos se expresan como sistemas de deducción en el estilo de la semántica natural.

Contenido.

* Tambien: Universidad de la República - Uruguay.
Instituto de Computación. Julio Herrera y Reissig 565, Piso 5
Montevideo, Uruguay

1. Introducción.

1.1 La noción de tipo en programación.

El desarrollo del concepto de tipo adquiere motivaciones en dominios más amplios que el de los lenguajes de programación, en general relacionados con teorías formales (ej.: teoría de conjuntos) y se liga al de la clasificación de los objetos de un cierto universo en términos de los propósitos para los cuales ellos son usados (CarWeg 85).

En particular en el cálculo λ todos los objetos son, en principio, representados por una clase de objetos sintácticos (expresiones) interpretados uniformemente como funciones. La noción de tipo aparece a raíz de la especialización de ciertas expresiones como representaciones de funciones específicas.

Con este enfoque, las aplicaciones sólo podrán ser interpretadas cuando el operando denote un objeto del dominio de la función representada por el operador. Si esta restricción no es impuesta en las reglas de formación de expresiones, entonces el lenguaje admitirá expresiones sintácticamente válidas sin significado.

Las expresiones construidas sin respetar una semántica preestablecida son llamadas en programación programas erróneos. Los aspectos de *seguridad* en programación tienen que ver con los mecanismos en base a los cuales puede detectarse y evitarse la construcción de programas erróneos.

Desde este punto de vista, el desarrollo del concepto de tipo en los lenguajes de programación tiene por objeto la definición de lenguajes en los cuales pueda imponerse que cada programa bien expresado deba satisfacer una cierta especificación. En un lenguaje con tipos cada expresión bien formada satisface una cierta especificación (eventualmente incompleta) que es llamada su tipo.

El concepto de tipo es también útil como herramienta de diseño. Los programas pueden ser interpretados como transformaciones entre objetos con propiedades conocidas. Las propiedades de los objetos y las transformaciones, una vez especificadas, deben ser representadas en el lenguaje de programación. Las primitivas del lenguaje definen un nivel de especificaciones para las cuales la construcción es trivial. Para problemas arbitrariamente complejos es útil considerar niveles de descripción en los cuales ciertas especificaciones convenientemente escogidas son usadas como primitivas. Cada una de ellas se representa separadamente en forma análoga hasta alcanzar el nivel de las primitivas del lenguaje de programación.

La anterior estrategia es la base de las técnicas de partición y refinamiento en programación y puede explicarse en términos de tipos: en un lenguaje dado, los tipos primitivos definen una clase de objetos con propiedades precisamente especificadas y representación inmediata. La posibilidad de definir tipos adecuados para construir descripciones abstractas de problemas y desarrollar las representaciones de estos tipos en forma modular constituye otra línea de interés en el desarrollo de los lenguajes de programación, asociada a los conceptos de *tipo abstracto de datos*, *encapsulamiento* y *modularización*.

1.2 Sistemas de tipos.

Los sistemas de tipos son los mecanismos que formalizan la introducción del concepto de tipo en los lenguajes de programación. Los tipos de las expresiones pueden estar ya presentes cuando se define la sintaxis y la semántica de éstas, o bien pueden introducirse a posteriori, sobre un lenguaje originalmente definido sin tipos. Este último es el enfoque seguido en este trabajo.

Un sistema de tipos define un conjunto de expresiones como tipos válidos y reglas de asociación de tipos a expresiones del lenguaje. Aquellas expresiones a las que el sistema no asigna tipo son, de hecho, excluídas del lenguaje. El objetivo de seguridad comentado antes puede ahora ser formulado como un requerimiento para el sistema de tipos [NorPet 89]: *serán excluídas del lenguaje exactamente aquellas expresiones en las cuales una operación sea usada fuera de su dominio.*

La tradición predominante en los lenguajes de programación impone también la restricción de que la verificación de tipos de las expresiones sea realizada en tiempo de compilación, por lo cual la relación entre expresiones y tipos:

a tiene tipo *A*

debe ser decidible. Este último requisito excluye los sistemas en los que los tipos representan especificaciones completas de programas, como los desarrollados a partir de [M-Lof 82], como en [NorPet 89].

Como consecuencia, en la generalidad de los lenguajes de programación los tipos denotan propiedades parciales de las expresiones. Más aún, el requisito de seguridad sobre las expresiones con tipo sólo se satisface parcialmente: son permitidas ciertas expresiones que, al ser evaluadas, resultan en la aplicación de operaciones fuera de su dominio (ej.: división por cero, cabeza o resto de la lista vacía). Estos casos son resueltos adaptando la semántica del lenguaje de modo que les sean asignados valores de error especiales. En general, la *robustez* de un sistema de tipos tiene que ver con la clase de restricciones que es impuesta a las expresiones para garantizar su buen comportamiento.

La noción de robustez es la versión informal del concepto de consistencia del sistema de tipos como sistema formal en relación a un modelo de comportamiento de las expresiones determinado por la semántica del lenguaje. Simétrico al de consistencia, existe el concepto de completitud, que podría expresarse por el requisito: *todas aquellas expresiones con sentido deben ser aceptadas por el sistema de tipos.* Informalmente, esta propiedad es manejada como la *flexibilidad* del sistema de tipos.

Obviamente el extremo de flexibilidad es ocupado por los lenguajes sin tipos (LISP, Scheme entre los funcionales). Opuestos a ellos son los lenguajes en que los tipos se introducen ya al definir la sintaxis y la semántica de las expresiones de modo de asegurar que cada una de ellas recibe un único tipo. En estos casos sólo se asigna semántica a las expresiones con tipo, de modo que el sistema es robusto por definición. El caso caracteriza a los sistemas *monomórficos*. En estos sistemas son expresiones diferentes (y deben ser definidas por separado) la que toma la cabeza de una lista de enteros y la que aplica idéntica transformación a una lista de booleanos. El caso es común en los lenguajes imperativos (ej.: Pascal) donde esta decisión de diseño puede aprovecharse para optimizar el código generado en la compilación, pero sus desventajas en flexibilidad ya comienzan a

pesar cuando esos lenguajes son usados en la práctica y son definitivamente inaceptables para lenguajes funcionales cuyas mejores propiedades provienen de las facilidades que proveen para construir módulos de aplicación general. Frente a este panorama, la práctica comprueba la utilidad de la noción de tipo desde el punto de vista de la seguridad de la programación y justifica el interés por definir un compromiso aceptable entre los extremos arriba mencionados.

La alternativa a la rigidez de los sistemas monomórficos reside precisamente en un enfoque en que la semántica de las expresiones sea definida sin utilizar el concepto de tipo. De este modo, operadores genéricos (como "cabeza de lista") pueden ser definidos sin restricciones. Luego se da un sistema de reglas que asigna tipos a las expresiones del lenguaje según su forma, y en el cual distintas ocurrencias de un operador genérico pueden recibir distintos tipos según cómo sean utilizadas. Este enfoque caracteriza a los sistemas *polimórficos*.

Vinculada a la noción de polimorfismo está la de *inferencia de tipos*. Como se ha dicho, la asignación de tipos a expresiones pueden darse por reglas que siguen la estructura de las últimas, por lo cual la decisión del tipo de una expresión puede hacerse sin necesidad de declaraciones redundantes como en los lenguajes convencionales. El uso obligatorio de las declaraciones es incómodo, sobre todo cuando no representan una herramienta de estructuración sino apenas un requisito formal.

Por último, la utilización de la idea de tipo como mecanismo abstracto de diseño puede incorporarse en los lenguajes de programación en la forma de definiciones de tipos, usualmente llamados *tipos abstractos*. Definir un tipo abstracto implica definir una clase de expresiones cuyas propiedades deben ser especificadas como los resultados de un cierto conjunto de operaciones asociadas al tipo. Tales resultados se definen en términos de expresiones de otros tipos predefinidos. Por lo anterior, las definiciones de tipos abstractos deben proveer medios para especificar el conjunto de operaciones asociadas.

1.3 Estructura del trabajo.

En [Milner 78] y [Holm 89] se definen sistemas de tipos polimórficos para un lenguaje funcional mínimo.

El propósito de este trabajo es proveer una definición análoga para un lenguaje funcional extendido. Las extensiones propuestas son:

- Añadir como constructores de tipos los operadores de *producto cartesiano*, *unión disjunta* y el *tipo de un elemento* ("1").
- Introducir *definiciones de tipos abstractos*.
- Introducir *definiciones de constantes globales*.

Para los dos primeros puntos se siguen las sugerencias de [Holm 89].

Para cumplir el propósito mencionado, debe definirse un lenguaje funcional con las referidas construcciones. En el capítulo 2, acompañadas de comentarios informales, aparecen la sintaxis y la semántica del lenguaje, presentándose asimismo el formalismo utilizado para la especificación, que es del estilo de la semántica natural [Kahn 87].

El capítulo 3 presenta el sistema de tipos en el mismo formalismo. Finalmente, el capítulo 4 contiene conclusiones y comentarios sobre desarrollos complementarios.

2. Especificación del lenguaje.

2.1 Sintaxis y descripción informal.

Se considera un lenguaje funcional con dos clases de expresiones: términos evaluables y definiciones. Los términos evaluables son básicamente los términos del cálculo λ equipado con constantes primitivas y definiciones de constantes locales. La siguiente es su sintaxis:

```
[Términos Evaluables]
t ∈ E-TERM sii

   t ::= x      |                          [Variable]
         c      |                          [Constante]
        λx.d    |                          [Abstracción Funcional]
        (d e)   |                          [Aplicación]
        true    | false |                  [Booleanos]
        num n   |                          [Números]
        ()      |                          [Nulo]
        + d e   - d e |
        * d e | / d e |                    [Aritméticas]
        = d e                              [Igualdad]
        and d e |
        or d e  | not d |                  [Lógicas]
        pair d e |
        fst d | snd d  |                   [de Pares]
        inl d | inr d  |
        case d x.e y.f |                   [de Uniones Disjuntas]
        if d e f |                         [Condicional]
        local c e in d |                   [Def. de constante local]
        abs(T) d | rep(T) d                [de Tipos Abstractos]

   donde
        x,y ∈ VAR  (variables)
        c   ∈ CNST (constantes)
        n ∈ {1,2,...
        d,e ∈ E-TERM (términos evaluables)
        T   ∈ TYPE-CNST (constantes de tipos)
        VAR, CNST, TYPE-CNST disjuntos
■
```

En el lenguaje se distinguen las constantes de las variables: las primeras son introducidas por definiciones y las segundas son las variables ligadas en las abstracciones funcionales. Esto se expresa en la existencia de los conjuntos disjuntos VAR y CNST.

Las primeras cuatro formas corresponden a los términos del cálculo λ con constantes. Luego hay constantes primitivas: los booleanos, números, el elemento nulo (único de un tipo 1) y los operadores aritméticos, relacionales y lógicos. Luego se incluyen los operadores asociados a los pares ordenados (constructor pair y selectores fst y snd) y uniones disjuntas (constructores inl y inr y discriminador case). Hay, naturalmente, un condicional. La construcción local corresponde a la definición de constantes locales y abs y rep están relacionados con la manipulación de elementos de tipos abstractos como se explicará más adelante.

Las expresiones del lenguaje se completan considerando las definiciones: hay definiciones de constantes globales y de tipos abstractos, con las formas:

[*Definición de constante global*]
cdef ∈ CNST-DEF sii

cdef ::= define *c* *d*

donde *c* ∈ CNST, *d* ∈ E-TERM

■

[*Definición de tipo abstracto*]
tdef ∈ TYPE-DEF sii

tdef ::= abstype *T* (*Xlist*) = *Tterm* with *cdeflist*

donde *T* ∈ TYPE-CNST, *Xlist* ∈ TYPE-VAR*,
Tterm ∈ TYPE-TERM, *cdeflist* ∈ CNST-DEF*

■

Las definiciones de constantes globales introducen constantes asociadas a términos evaluables. Informalmente, una constante es un nombre para un término evaluable complejo.

El lenguaje tiene también un lenguaje de tipos, según se define en la siguiente sintaxis:

[*Términos de Tipos*]
T ∈ TYPE-TERM sii

Tterm ::=	*t*	¦	[*Variable de tipo*]
	1	¦	[*Tipo de un elemento*]
	bool	¦	[*Booleano*]
	number	¦	[*Números*]
	$A \longrightarrow B$	¦	[*Funciones*]
	$A \times B$	¦	[*Pares*]
	$A \cup B$	¦	[*Uniones Disjuntas*]
	T (*Tlist*)		[*Tipo abstracto*]

donde *t* ∈ TYPE-VAR, *A*, *B* ∈ TYPE-TERM
T ∈ TYPE-CNST, *Tlist* ∈ TYPE-TERM*

■

Las definiciones de tipos abstractos introducen constantes de tipos asociadas a términos del lenguaje de tipos, como en el siguiente ejemplo, que corresponde a la definición del tipo *lista de objetos genéricos*:

abstype list (X) = 1 ∪ (X × list(X))
with [...]

La definición especifica básicamente la representación del tipo abstracto introducido en términos de tipos predefinidos. Lo primero es el nombre del nuevo tipo (una constante de tipo). Luego sigue la lista de variables de tipo de las que depende la definición. Las variables de tipo denotan, obviamente, tipos genéricos. A la derecha de =, un término de tipo describe la representación del tipo definido.

Lo que sigue a with es una lista de definiciones de constantes globales que representarán las operaciones asociadas al tipo abstracto. Para dar las definiciones de estas operaciones se usa un mecanismo similar al usado en ML [GOUHP 85] y sugerido también en [Holm 89]:

Cada constante de tipo *T* introducida por una definición de tipo abstracto tiene asociadas dos constantes primitivas abs(*T*) y rep(*T*) que pueden evaluarse cuando son aplicados a un término

evaluable, (ver sintaxis de los términos evaluables). Informalmente: abs(*T*) aplicado a un término evaluable que tiene por tipo a la representación de *T* da como resultado el objeto de tipo *T* representado por ese término evaluable. Simétricamente, rep(*T*) aplicado a un objeto de tipo *T* da como resultado un objeto cuyo tipo es el de la representación de *T*. De acuerdo a lo sugerido por esta definición, abs(*T*) será llamado el *constructor* del tipo abstracto *T* y rep(*T*) su *destructor*.

Usando estas primitivas pueden definirse operaciones para cada tipo abstracto en términos de las operaciones asociadas a los tipos usados en su representación. Un ejemplo es la definición de la operación que toma cabeza de una lista genérica:

define hd λx.case (rep(list) x) (z.*fail*) (z.fst z)

La constante -hd- se define como una abstracción funcional, que espera un argumento -x- que debe ser una lista (definida como en el ejemplo anterior). Las listas se representan por uniones, por lo cual la expresión -rep(list) x- es, o bien el objeto () (único del tipo 1), o bien un par. Para discriminar entre objetos de una unión se usa la construcción case.

La semántica de -case *d* x.*e* y.*f*- es como sigue: *d* es un objeto de una unión disjunta A ∪ B. Se aplica una de las dos abstracciones -λx.*e* ó λy.*f*- a *d* según éste pertenezca a A o B, respectivamente. El orden entre los operandos de la unión es, pues, relevante para interpretar case.

En el ejemplo, cada uno de los tres operandos de case se encierra entre paréntesis, para distinguirlos. Si -rep(list) x- es () entonces le será aplicada la abstracción de resultado constante *fail*, que es una expresión convencional de error agregada al conjunto de valores. En otro caso, la representación de x es un par, del cual se toma el primer componente, que es precisamente la cabeza de la lista.

A cada tipo abstracto puede, como se ha dicho, asociarse una lista de definiciones como la anterior. Puede imponerse que las primitivas abs y rep sólo sean usadas en el alcance de with, dentro de la definición del tipo abstracto, con lo cual las operaciones definidas para éste esconderían su representación. Esta es la propiedad de encapsulamiento en las definiciones de tipos abstractos.

El efecto de las definiciones es esencialmente modificar el contexto en que se computan los términos evaluables. Para especificar este efecto debe darse significado a una secuencia de expresiones del lenguaje. Esta secuencia puede pensarse como una abstracción de un ciclo del estilo "*Read-Eval-Print*". Aquí se define como una sesión:

[*Sesiones*]
SESS = MESS*
[*Mensajes*]
MESS = CNST-DEF ∪ TYPE-DEF ∪ E-TERM

Ahora puede definirse formalmente la semántica del lenguaje pero previamente se presentará el formalismo a ser usado.

2.2 Formalismo de especificación.

Para dar la semántica del lenguaje podría usarse semántica denotacional (como en [Milner 78]) o de reducciones (como en [Holm 89]). Aquí se prueba como alternativa una variante en el estilo de la semántica natural presentada en [Kahn 87] como un lenguaje para definir uniformemente semántica y sistemas de tipos, y aplicada de esa forma en [CLÓDDK 86]. La aproximación natural supera a la denotacional en claridad y en la abstracción ganada al definir el sistema de tipos. No es incompatible con la especificación de cálculo de reducciones como en [Holm 89]. Como ella, es operacional, en el sentido que especifica un procedimiento para la computación de las expresiones y estructural, en el sentido que sigue la estructura de las expresiones. En el capítulo 4 el formalismo elegido es evaluado en relación al enfoque de reducciones.

La idea general en la semántica natural es construir *sistemas de deducción* que caractericen predicados a ser definidos sobre una clase de expresiones. Por ejemplo, la evaluación de una *expresión* E a un *valor* v, realizada en un *contexto* ρ puede caracterizarse por un predicado de la forma:

$$\rho \vdash E \longrightarrow v$$

llamado un *secuente*.

En el caso anterior, el valor de la expresión E puede depender no sólo de la forma de E sino de valores asociados a ciertos objetos sintácticos que ocurren en E (en general, identificadores). Los contextos representan, precisamente, la referida asociación entre dichos objetos sintácticos y valores.

Una especificación en este formalismo debe interpretarse como un sistema de deducción, en el que se pretende verificar si aserciones como la anterior son teoremas. El sistema de deducción se define dando axiomas y reglas de inferencia.

Las reglas de inferencia son de la forma:

$$\frac{f_1 \; . \; . \; . \; f_n \;\|\; c_1 \; . \; . \; c_m}{f}$$

es decir, constan de un *numerador* y un *denominador*. El primero es un conjunto no ordenado de secuentes $-f_i-$ (*premisas*) y un conjunto de fórmulas $-c_j-$ (*condiciones* que deben verificarse para aplicar la regla). El denominador es un secuente llamado la *conclusión* de la regla.

Los axiomas pueden verse como reglas sin numerador. Una *especificación* es un conjunto no ordenado de reglas.

Para especificar semántica de un lenguaje como el considerado se usarán, en general, reglas de la forma:

$$\frac{\rho_1 \vdash E_1 \longrightarrow v_1 \qquad \rho_n \vdash E_n \longrightarrow v_n}{\rho \vdash E \longrightarrow f(v_1, \ldots, v_n)}$$

y, por supuesto, axiomas:

$$\rho \vdash E \longrightarrow v$$

Evaluar una expresión e en un contexto ρ significa, con esta semántica, encontrar una demostración de

$$\rho \vdash e \longrightarrow v$$

para algún v.

Con esto en mente, las reglas con premisas pueden interpretarse como sigue: para evaluar E en ρ, evalúense $E_1, \ldots, E_n$ en $\rho_1, \ldots, \rho_n$ y apliquese f a los resultados obtenidos. Los axiomas

corresponden a expresiones que pueden evaluarse sin realizar ulteriores cálculos. Esta interpretación da el carácter operacional de la especificación.

Los secuentes y condiciones en semántica natural son fórmulas de un lenguaje de primer orden, de modo que, para construir la especificación requerida, debe darse una sintaxis para contextos, expresiones y valores. Más aún, símbolos como f en las reglas de arriba son constructores sintácticos, en el caso, de valores.

En este trabajo se usan sobre contextos y expresiones funciones que no son constructores sintácticos y que son especificados separadamente. Esto se aparta de la semántica natural pura pero simplifica significativamente la especificación.

Se usarán también algunas convenciones de modularización: una especificación puede recibir un nombre y ser referenciada por él en la construcción de otras especificaciones. Esto se indica escribiendo el nombre de la especificación referenciada como superíndice del símbolo $\vdash$.

La semántica del lenguaje considerado se da en la sección siguiente.

2.3 Semántica formal.

Primeramente se definen los contextos, expresiones y valores sobre los que se construirá la especificación.

Por supuesto, las expresiones a ser consideradas serán en este caso los mensajes y sesiones del lenguaje. La evaluación de una expresión será caracterizada por una regla que, o bien es axioma, o bien tiene premisas que representan las evaluaciones de las expresiones componentes de la original. Esto muestra la condición estructural de la especificación. Los valores serán un subconjunto de las expresiones del lenguaje.

Por su parte, los contextos para la evaluación de mensajes y sesiones deben reflejar el conjunto de constantes y tipos que las sucesivas definiciones introducen. Para esto se definen:

[*Contexto de Constantes*]
$\gamma \in$ CNST-CNTXT sii
γ: CNST $\rightarrow$ E-TERM
dada como conjunto de pares
■
[*Contexto de Tipos*]
$\tau \subseteq$ TYPE-CNST
■
[*Contexto de Constantes y Tipos*]
$\rho \in$ CT-CNTXT sii
$\rho \in$ CNST-CNTXT $\times$ TYPE-CNTXT
■

Además de estas definiciones se usarán, como se ha dicho, algunas operaciones sobre contextos y expresiones que no son constructores sintácticos. En particular, ciertos contextos son representados por funciones parciales dadas por extensión. Sobre éstas se usará:

[*Unión*]
$\chi \cup \langle\langle a,b\rangle\rangle$
donde χ es una función parcial dada por extensión
si $\langle a,c\rangle \in \chi$ entonces se sustituye por $\langle a,b\rangle$
■

[*Selector*]
$\chi(a)$ está definido si $\exists\ \langle a,b\rangle \in \chi$ y, en ese caso,
denota b
■

Ahora pueden presentarse las reglas (resumidas) que definen la evaluación de los términos evaluables. La especificación es llamada **TermEval**.

TermEval

Se definen predicados de la forma:

$$\rho \vdash e \longrightarrow v$$

donde ρ es un contexto de constantes y tipos,
$e \in$ E-TERM $\cup$ FIX
$v \in$ E-TERM
y
FIX = $\langle$fix x e, con $x \in$ VAR, $e \in$ E-TERM $\cup$ FIX$\rangle$

En lo que sigue, $c \in$ CNST, $x,y \in$ VAR,
$d,e,f,g \in$ E-TERM
$n \in \{1,2,\ldots$

[*Términos Canónicos*]

$\rho \vdash \lambda x.d \longrightarrow \lambda x.d$
$\rho \vdash$ **true** $\longrightarrow$ **true**
$\rho \vdash$ **false** $\longrightarrow$ **false**
$\rho \vdash$ **num** n $\longrightarrow$ **num** n
$\rho \vdash ()$ $\longrightarrow ()$
$\rho \vdash$ **pair** d e $\longrightarrow$ **pair** d e
$\rho \vdash$ **inl** d $\longrightarrow$ **inl** d
$\rho \vdash$ **inr** d $\longrightarrow$ **inr** d
$\rho \vdash$ **abs**(T) d $\longrightarrow$ **abs**(T) d

[*Constante*]

$$\frac{\langle\gamma,\tau\rangle \vdash \gamma(c) \longrightarrow v}{\langle\gamma,\tau\rangle \vdash c \longrightarrow v}$$

[*Aplicación*]

$$\frac{\rho \vdash d \longrightarrow \lambda x.f \quad \rho \vdash [e/x]f \longrightarrow v}{\rho \vdash (d\ e) \longrightarrow v}$$

[*Aritméticas*]
[*Igualdad*]

$$\frac{\rho \vdash d \longrightarrow \textbf{num}\ m \quad \rho \vdash e \longrightarrow \textbf{num}\ n}{\rho \vdash +\ d\ e \longrightarrow \textbf{sum}\ (m,n)}$$

[*Lógicas*]

$$\frac{\rho \vdash d \longrightarrow \textbf{false}}{\rho \vdash \textbf{and}\ d\ e \longrightarrow \textbf{false}}$$

.....

$$\frac{\rho \vdash e \longrightarrow \text{false}}{\rho \vdash \text{and } d\ e \longrightarrow \text{false}}$$

$$\frac{\rho \vdash d \longrightarrow \text{true} \quad \rho \vdash e \longrightarrow \text{true}}{\rho \vdash \text{and } d\ e \longrightarrow \text{true}}$$

[...]

[*Selectores de Pares Ordenados*]

$$\frac{\rho \vdash d \longrightarrow \text{pair } e\ f \quad \rho \vdash e \longrightarrow v}{\rho \vdash \text{fst } d \longrightarrow v}$$

$$\frac{\rho \vdash d \longrightarrow \text{pair } e\ f \quad \rho \vdash f \longrightarrow v}{\rho \vdash \text{snd } d \longrightarrow v}$$

[*Discriminador de Uniones Disjuntas*]

$$\frac{\rho \vdash d \longrightarrow \text{inl } g \quad \rho \vdash [d/x]e \longrightarrow v}{\rho \vdash \text{case } d\ x.e\ y.f \longrightarrow v}$$

$$\frac{\rho \vdash d \longrightarrow \text{inr } g \quad \rho \vdash [d/y]f \longrightarrow v}{\rho \vdash \text{case } d\ x.e\ y.f \longrightarrow v}$$

[*Condicional*]

$$\frac{\rho \vdash d \longrightarrow \text{true} \quad \rho \vdash e \longrightarrow v}{\rho \vdash \text{if } d\ e\ f \longrightarrow v}$$

$$\frac{\rho \vdash d \longrightarrow \text{false} \quad \rho \vdash f \longrightarrow v}{\rho \vdash \text{if } d\ e\ f \longrightarrow v}$$

[*Definición de Constante Local*]

$$\frac{\langle \gamma \cup \{\langle c, \text{fix } xc\ xce\rangle\}, \tau\rangle \vdash d \longrightarrow v}{\langle \gamma, \tau\rangle \vdash \text{local } c\ e \text{ in } d \longrightarrow v}$$

$[xc = \text{CTOV}\langle c, c\rangle,\ xce = \text{CTOV}\langle c, e\rangle]$

[*Operador de Punto Fijo*]

$$\frac{\rho \vdash [\text{fix } x\ e/x]e \longrightarrow v}{\rho \vdash \text{fix } x\ e \longrightarrow v}$$

[*Destructor de Tipos Abstractos*]

$$\frac{\langle \gamma, \tau\rangle \vdash e \longrightarrow \text{abs}(T)\ f \quad \langle \gamma, \tau\rangle \vdash f \longrightarrow v \parallel T \in \tau}{\langle \gamma, \tau\rangle \vdash \text{rep}(T)\ e \longrightarrow v}$$

■

Las primeras reglas son axiomas. Corresponden a términos que denotan ellos mismas su propio valor. Usando terminología de reducciones se denomina a éstos, términos canónicos. De hecho los términos canónicos definen el conjunto de valores posibles de las términos evaluables. Es decir, si un término evaluable puede evaluar a v en cualquier contexto, entonces v es un término canónico, según se deduce fácilmente de las formas de las reglas.

El siguiente caso corresponde a las constantes definidas. La evaluación de una constante se expresa como la evaluación del

término evaluable asociado a ella por la definición que la introdujo. Esta asociación debe estar definida en el contexto de nconstantes.

Luego se define la evaluación de una aplicación funcional: se evalúa el operador de la aplicación que debe resultar en una abstracción funcional. Luego se evalúa el cuerpo de la abstracción resultante, donde las ocurrencias del argumento nominal son sustituídas por el operando de la aplicación. Se usa la *sustitución* en términos evaluables, definida como en el cálculo λ.

Claramente, la regla de la aplicación representa la regla β del cálculo λ. Desde que el operando de la aplicación no es evaluado, se está sugiriendo una evaluación en orden normal. De hecho, la intención es que la especificación admita una implementación con estrategia de evaluación perezosa ("lazy").

Las reglas *Aritméticas* y de *Igualdad* corresponden a las evaluaciones de los correspondientes términos evaluables (ver sintaxis) y son todas análogas a la de la suma. Los operandos deben ser evaluados a números y se usa una primitiva SUM con operandos números y la semántica habitual de la suma. Las otras reglas se omiten para ahorrar espacio.

Las reglas de evaluación llamadas *Lógicas* no son, en principio, muy diferentes de las anteriores. Sin embargo, se hace notar en este caso la intención de implementar una estrategia de evaluación perezosa, desde que se evalúan sólo los operandos que sean necesarios para determinar el resultado de la operación.

Las reglas para *Selectores de Pares Ordenados* son bastante autoexplicativas. El operando en ambos casos debe ser evaluado a un par y el resultado de la operación original es el operando del constructor de pares que corresponde a cada proyección.

Las reglas para el *Discriminador de Uniones Disjuntas* formalizan la explicación dada más arriba. Se omite, pues, mayor detalle.

Las reglas para el *Condicional* son también bastante obvias. Nuevamente se hace notar la estrategia perezosa de evaluación.

Más interesante es la evaluación de las definiciones locales. En *local* c *e in* d se define la constante c como una abreviatura para el término evaluable e, lo cual debe considerarse al evaluar d. El término e puede contener ocurrencias de la constante c, lo cual puede interpretarse, o bien como una definición recursiva o bien como una redefinición de c. En este último caso, las ocurrencias de c en e se interpretan como asociadas a un cierto término evaluable según una definición anterior. En los lenguajes que admiten esta interpretación, existen dos formas de definiciones locales (ej.: *let* y *letrec*) para indicar expresamente cada caso. Aquí se toma la decisión de considerar un único tipo de definicion que es, en general, recursiva, con lo cual se impide que una constante sea redefinida usando una definición anterior, por considerarse que esta práctica no es consistente con el estilo puramente aplicativo.

Para conseguir el objetivo mencionado debe considerarse a c definida como punto fijo de una abstracción de la forma $\lambda c.e$. Esta última expresión no está bien formada, sin embargo, puesto que el argumento de una abstracción debe ser una variable y no una constante. La función CTOV se usa para solucionar este problema. Su especificación es:

$$\text{CTOV}(c,e) = [xc/c]e$$

donde $c \in$ CNST, $e \in$ E-TERM, $xc \in$ VAR, xc no ocurre en e.

es decir, las ocurrencias de la constante c en e son sustituídas por una variable no usada xc.

Usando CTOV puede expresarse la definición de la constante c como punto fijo de $\lambda xc.xce$ donde xc = CTOV(c,c) y xce = CTOV(c,e).

La construcción *fix x e* es usada con el propósito de representar el punto fijo de $\lambda x.e$. No pertenece a E-TERM desde que

no puede ser usado directamente en las expresiones del lenguaje, de donde surge la necesidad de definir FIX aparte de E-TERM, como se hizo en la especificación.

Finalmente, aparece la regla para evaluar el destructor rep de un tipo abstracto T. Este debe estar definido en el contexto de tipos, lo cual se impone como condición para aplicar la regla. El operando de rep(T) debe ser un objeto de tipo T construido por abs(T) aplicado a una cierta representación f. La evaluación del término original es la evaluación de f.

Esto completa la explicación de las reglas de evaluación de términos evaluables. Deben considerarse también las definiciones, las cuales afectan los contextos de evaluación. Como ya se dijo, esto se expresa dando semántica al concepto de sesión, por medio de la especificación SessEval:

SessEval

Se definen predicados

$$\rho \vdash s \longrightarrow v$$

donde ρ es un contexto de constantes y tipos,
$s \in$ SESS
$v \in$ SESS

[*Sesión Vacía*]

$$\rho \vdash [\,] \longrightarrow [\,]$$

[*Definición de Constante Global*]

$$\frac{\langle \gamma \cup \{\langle c, \textbf{fix } xc\ xce\rangle\}, \tau\rangle \vdash s \longrightarrow s'}{\langle \gamma, \tau\rangle \vdash [\textbf{define } c\ e\ ;s] \longrightarrow s'}$$

$$(xc = \text{CTOV}\ \langle c, c\rangle,\ xce = \text{CTOV}\ \langle c, e\rangle)$$

[*Definición de Tipo Abstracto*]

$$\frac{\langle \gamma, \tau \cup \{T\}\rangle \vdash cdeflist \circ s \longrightarrow s'}{\langle \gamma, \tau\rangle \vdash [\textbf{abstype } T\ (Xlist) = Tterm \textbf{ with } cdeflist\ ;\ s] \longrightarrow s'}$$

[*Término Evaluable*]

$$\frac{\rho \overset{\text{TermEval}}{\vdash\!\!\!-\!\!\!-\!\!\!-} e \longrightarrow v}{\rho \vdash [e;s] \longrightarrow [v;s]}$$

■

La primera regla es el único axioma de esta especificación y corresponde a la sesión vacía que es la única forma canónica (y valor posible) de sesión.

La siguiente regla se aplica a una sesión cuyo primer mensaje es una definición de constante global. El tratamiento es análogo al explicado antes para las constantes locales: se agrega al contexto de constantes un par que define como expresión asociada a la constante al punto fijo del término evaluable indicado en la definición. Luego, por un tratamiento usual para secuencias, se evalúa el resto de la sesión en el nuevo contexto.

La regla para las definiciones de tipos abstractos tiene una interpretación similar, aunque parezca más compleja. La constante de tipo definida es incorporada al contexto de tipos y la evaluación continúa sobre el resto de la sesión, prefijada con la lista de definiciones de operaciones para el tipo introducido. Esto se nota por $\circ$.

Finalmente, cuando el primer mensaje de la sesión a evaluar

es un término evaluable, se intenta dar valor a éste usando la especificación **TermEval** para proseguir con el resto de la sesión.

Con esta especificación puede interpretarse la evaluación de una sesión s como la búsqueda de una demostración para el secuente:

$$\langle\langle\rangle,\langle\rangle\rangle \vdash s \Longrightarrow v, \text{ para algún } v$$

es decir, iniciada con contextos vacíos.

Nótese que no cualquier predicado $\rho \vdash E \Longrightarrow v$ podrá ser probado, ya sea en una u otra especificación. De hecho, las alternativas son tres: la prueba puede tener éxito (dando lugar a la evaluación de la expresión) o bien fallar o ser infinita. Ejemplos de las dos últimas alternativas son las evaluaciones de los términos evaluables

$$\textbf{if } \lambda x.x \; d \; e$$

y

$$\textbf{local } c \; c \textbf{ in } c$$

Las evaluaciones fallidas son las de términos evaluables bien formados pero sin significado. Las evaluaciones infinitas corresponden a programas que no terminan. De la especificación **SessEval** puede verse que la evaluación de una sesión falla o es infinita sii falla o es infinita la de algún término evaluable de la sesión.

Precisamente, para impedir la evaluación de términos sin significado se introduce la noción de tipo. El sistema de tipos que se definirá en el capítulo siguiente caracteriza expresiones del lenguaje con significado.

3. Sistema de Tipos.

3.1 Especificación.

La intención en este capítulo es definir sistemas de deducción análogos a los del capítulo anterior para asignar tipos a las expresiones del lenguaje.

Primeramente se definen los tipos del lenguaje.

[Tipos]

$t \in \mathbb{TYPE}$ sii

$t ::=$ **1**	¦	*[Tipo de un elemento]*
bool	¦	*[Booleano]*
number	¦	*[Numero]*
$A \longrightarrow B$	¦	*[Función]*
$A \times B$	¦	*[Producto]*
$A \bigcup B$	¦	*[Unión Disjunta]*
$T(Tlist)$		*[Abstracto]*

donde $A, B \in \mathbb{TYPE}$, $T \in \mathbb{TYPE}$-CNST, $Tlist \in \mathbb{TYPE}*$

■

La asignación de tipo al cuerpo de un término evaluable que introduce variables ligadas dependerá de hipótesis sobre los tipos de estas variables.

Por este motivo se da la siguiente definición:

[Contexto de Variables]

Es una función parcial

$$\varphi : \mathbb{VAR} \rightarrow\!\!\!\!\!+ \; \mathbb{TYPE}$$

dada por extensión.

■

También se modifica la noción de contexto de tipos, asociando a cada constante de tipo introducida el término de tipo que especifica su representación y la lista de variables de tipo de la que éste depende.

[*Contexto de Tipos*]
Es una función parcial
τ : TYPE-CNST $\mapsto$ (TYPE-VAR* × TYPE-TERM)
dada por extensión.
■

[*Contexto de Inferencia de Tipos*]
Es un par $\Gamma = \langle \rho, \varphi \rangle$ donde
ρ es un contexto de constantes y tipos
φ es un contexto de variables
■

Sobre los nuevos contextos definidos se usan las mismas operaciones del capítulo anterior. Los pares miembros de un contexto de variables son notados $x{:}A$.

Ahora puede darse la especificación de la asignación de tipos a términos evaluables, llamada **TermType**:

TermType
Se definen secuentes de la forma
$$\Gamma \vdash e{:}A$$
con $e \in$ E-TERM, $A \in$ TYPE
En lo que sigue $c \in$ CNST, $d,e,f \in$ E-TERM, $x,y \in$ VAR
$A,B,C \in$ TYPE, $T \in$ TYPE-CNST,
$Tterm \in$ TYPE-TERM, $Xlist \in$ TYPE-VAR*

[*Un Elemento*] $\Gamma \vdash ():1$

[*Booleanos*]
$\Gamma \vdash$ **true**: **bool**
$\Gamma \vdash$ **false**: **bool**

[*Números*]
$\Gamma \vdash$ **num** n: **number**

[*Abstracción Funcional*]
$$\frac{\langle \rho, \varphi \cup \langle x{:}A \rangle \rangle \vdash d{:}B}{\langle \rho, \varphi \rangle \vdash \lambda x.d : A \rightarrow B}$$

[*Constructor de Pares*]
$$\frac{\Gamma \vdash d{:}A \qquad \Gamma \vdash e{:}B}{\Gamma \vdash \textbf{pair}\ d\ e : A \times B}$$

[*Constructores de Uniones Disjuntas*]
$$\frac{\Gamma \vdash d{:}A}{\Gamma \vdash \textbf{inl}\ d : A \cup B}$$

$$\frac{\Gamma \vdash d{:}B}{\Gamma \vdash \textbf{inr}\ d : A \cup B}$$

[*Constructor de Tipos Abstractos*]
$$\frac{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash d : [\sigma]Tterm \quad \| \quad \tau(T) = \langle Xlist, Tterm \rangle}{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash \textbf{abs}(T)\ d : T([\sigma]Xlist)}$$

$(\sigma$: TYPE-VAR $\rightarrow$ TYPE)

.....

[Variable]

$$\frac{\|\varphi(x) = A}{\langle \rho, \varphi \rangle \vdash x : A}$$

[Constante]

$$\frac{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash \gamma(c) : A}{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash c : A}$$

[Aplicación]

$$\frac{\Gamma \vdash d : A \to B \qquad \Gamma \vdash e : A}{\Gamma \vdash (d\ e) : B}$$

[Aritméticas]

$$\frac{\Gamma \vdash d : \text{number} \qquad \Gamma \vdash e : \text{number}}{\Gamma \vdash +\ d\ e : \text{number}}$$

[Igualdad]

$$\frac{\Gamma \vdash d : \text{number} \qquad \Gamma \vdash e : \text{number}}{\Gamma \vdash =\ d\ e : \text{bool}}$$

[Lógicas]

$$\frac{\Gamma \vdash d : \text{bool} \qquad \Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{and}\ d\ e : \text{bool}}$$

[...]

[Discriminador de Uniones Disjuntas]

$$\frac{\langle \rho, \varphi \rangle \vdash d : A \cup B \quad \langle \rho, \varphi \cup \langle x : A \rangle \rangle \vdash e : C \quad \langle \rho, \varphi \cup \langle y : B \rangle \rangle \vdash f : C}{\langle \rho, \varphi \rangle \vdash \text{case}\ d\ x.e\ y.f : C}$$

[Selectores de Pares]

$$\frac{\Gamma \vdash d : A \times B}{\Gamma \vdash \text{fst}\ d : A}$$

$$\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \text{snd}\ e : B}$$

[Condicional]

$$\frac{\Gamma \vdash d : \text{bool} \qquad \Gamma \vdash e : A \qquad \Gamma \vdash f : A}{\Gamma \vdash \text{if}\ d\ e\ f : A}$$

[Definicion de Constante Local]

$$\frac{\langle\langle \gamma \cup \langle\langle c, \text{fix}\ xc\ xce \rangle\rangle, \tau \rangle, \varphi \rangle \vdash d : A}{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash \text{local}\ c\ e\ \text{in}\ d : A}$$

$$(xc = \text{CTOV}\ \langle c, c \rangle,\ xce = \text{CTOV}\ \langle c, e \rangle)$$

[Operador de Punto Fijo]

$$\frac{\langle \rho, \varphi \cup \langle x : A \rangle \rangle \vdash d : A}{\langle \rho, \varphi \rangle \vdash \text{fix}\ x\ d : A}$$

[Destructor de Tipos Abstractos]

$$\frac{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash d : T([\sigma]Xlist) \parallel \tau(T) = \langle Xlist, Tterm \rangle}{\langle\langle \gamma, \tau \rangle, \varphi \rangle \vdash \text{rep}(T)\ d : [\sigma]Tterm}$$

$$(\sigma : \text{TYPE-VAR} \to \text{TYPE})$$

■

Las primeras nueve reglas corresponden a los tipos de las formas canónicas (valores de términos evaluables):

El tipo 1 tiene un único valor (), **true** y **false** son los valores de bool, y los valores de **number** son precisamente de la forma num *n*. Esto establecen los primeros axiomas.

Enseguida se tratan los valores de los tipos construidos mediante los constructores primitivos de tipos. El primero de ellos es la abstracción funcional, que representa la forma de los valores del tipo de las funciones del tipo *A* en el tipo *B*. Puede probarse que una abstracción tiene tal tipo si, asumiendo que su argumento tiene tipo *A*, es posible probar que el cuerpo tiene tipo *B*. La hipótesis sobre el tipo del argumento se registra en el contexto de variables.

Las reglas correspondientes a los constructores de pares ordenados y uniones disjuntas son bastante intuitivas. Ellas definen la forma de los valores de dichos tipos.

La asignación de tipo a un término de la forma abs(*T*) *d* merece una explicación más detallada.

En primer lugar, la constante de tipo *T* debe haber sido definida, lo que equivale a establecer, como condición de la regla, que tiene una imagen en τ. Precisamente, $\tau(T)$ denota el par formado por el término de tipo que especifica la representación de *T* -*Tterm*- y la lista de variables de tipo -*Xlist*- que ocurren en él. El término *d* debe ser entonces de un tipo obtenido de *Tterm* por instanciación de sus variables en tipos cualesquiera. Esta instanciación está representada por la función σ que es, de hecho, una sustitución de variables por tipos. La aplicación de σ a *Tterm* se nota por $[\sigma]Tterm$. Finalmente, si el tipo requerido para *d* ha sido probado puede concluirse que el tipo del término original es, precisamente, la instancia de *T* definida por σ, es decir $T([\sigma]Xlist)$.

Los tipos de las variables pueden probarse si han sido introducidos en el correspondiente contexto (al pretender asignar tipo a la expresión que introdujo la variable). Análogamente puede entenderse la regla para las constantes. Existe, con todo, una diferencia: en el caso de la constante, no se registra ésta en el contexto asociada a un tipo sino ál (punto fijo del) término evaluable que ella representa. Las consecuencias de esta decisión son comentadas en 3.2.

Del resto de las reglas, las siguientes son comentadas en detalle, siendo las otras muy intuitivas:

La regla de la aplicación fuerza que el operador tenga el tipo de una función y que el operando tenga tipo del dominio de la misma para dar al resultado el tipo del codominio.

La del discriminador case introduce variables ligadas, lo cual es natural a partir de su semántica.

La regla de las definiciones locales introduce constantes en el contexto correspondiente. Véase que se asocia a la constante la aplicación del operador de punto fijo al término indicado en la definición y no el tipo de éste.

La del operador de punto fijo **fix**, también introduce una variable ligada y su cuerpo (naturalmente) debe ser del tipo supuesto para la referida variable.

Finalmente, la regla del destructor de tipos abstractos opera en forma simétrica a la del constructor: si la constante de tipo involucrada está definida y el término al que se aplica el destructor tiene por tipo una instancia del tipo abstracto obtenida por una sustitución de las variables de tipo, entonces el término original tiene por tipo una instancia de la representación del tipo abstracto obtenida por idéntica sustitución.

Lo que resta es extender la asignación de tipos a sesiones:

SessType

Se definen secuentes de la forma

$$\rho \vdash s$$

donde $s \in \mathbb{SESS}$.
En lo que sigue $c \in \mathbb{CNST}$, $e \in \mathbb{E\text{-}TERM}$,
$A \in \mathbb{TYPE}$, $T \in \mathbb{TYPE\text{-}CNST}$,
$Tterm \in \mathbb{TYPE\text{-}TERM}$, $Xlist \in \mathbb{TYPE\text{-}VAR}^*$
$ldef \in \mathbb{CNST\text{-}DEF}^*$

[*Definición de Constante Global*]

$$\frac{\langle \gamma \cup \{\langle c, \mathbf{fix}\ xc\ xce\rangle\}, \tau\rangle \vdash s}{\langle \gamma, \tau\rangle \vdash [\mathbf{define}\ c\ e\ ; s]} \quad (xc = \mathrm{CTOV}(c,c),\ xce = \mathrm{CTOV}(c,e))$$

[*Definición de Tipo Abstracto*]

$$\frac{\tau \overset{\mathbf{Type}}{\vdash\!\!\!-\!\!\!-} \langle T, Xlist, Tterm\rangle \qquad \langle \gamma, \tau \cup \{\langle T, \langle Xlist, Tterm\rangle\rangle\}\rangle \vdash ldef \circ s}{\langle \gamma, \tau\rangle \vdash [\mathbf{abstype}\ T\ (Xlist)\ Tterm\ \mathbf{with}\ ldef\ ; s]}$$

[*Término Evaluable*]

$$\frac{\langle \rho, \langle\rangle\rangle \overset{\mathbf{TermType}}{\vdash\!\!\!-\!\!\!-\!\!\!-} e : A \qquad \rho \vdash s}{\rho \vdash [e ; s]}$$

[*Sesión Vacía*]

$$\rho \vdash [\,]$$

■

Las reglas deben interpretarse como un mecanismo de verificación de la buena formación de los mensajes de las sesiones en relación al sistema de tipos. De hecho, la sesión s está "*bien tipada*" si puede probarse en SessType:

$$\langle \langle\rangle, \langle\rangle\rangle \vdash s$$

La regla para la sesión cuyo primer mensaje es una definición de constante global opera introduciendo la constante asociada al punto fijo del término evaluable correspondiente en el contexto de constantes, para proseguir con el resto. La regla para las definiciones de tipos abstractos es similar. Se verifica la buena formación del término de tipo con un sistema que impone tipos a tales términos, llamado **Type**. No se incluye la especificación de este sistema para ahorrar espacio. Aquí puede forzarse encapsulamiento sustituyendo la segunda premisa por:

$$\langle \gamma, \tau \cup \{\langle T, \langle Xlist, Tterm\rangle\rangle\}\rangle \vdash ldef \qquad \langle \gamma, \tau\rangle \vdash s$$

haciendo que la definición de T sólo pueda usarse en el alcance de **with**. De hecho, dicha definición sólo se usa para asignar tipos a términos en que se apliquen el constructor o el destructor de T, lo cual, si hay encapsulamiento, sólo puede ocurrir en el alcance mencionado.

La regla para términos evaluables deriva la decisión sobre el tipado de la sesión a la asignación de algún tipo al término evaluable que es su primer mensaje. Finalmente, la sesión vacía está bien tipada.

3.2 Variables, constantes y polimorfismo.

La fuente original de polimorfismo del sistema está constituída por las reglas correspondientes a la abstracción funcional, aplicación y variable, es decir, las de asignación de tipos al cálculo λ puro (Hindley 69). Estas reglas imponen sobre las aplicaciones la restricción de que su operando sea del tipo del dominio del operador y se definen con abstracción de cualquier conjunto de constantes de tipos que se pueda introducir en el lenguaje, siendo por esto inherentemente polimórficas.

Para una expresión como:

$$\lambda x.\, x$$

más de un tipo puede ser asignado, en particular, cualquiera de la forma $A \rightarrow A$. En las derivaciones correspondientes a una tal asignación, la regla de la abstracción funcional introduce la hipótesis $x:A$ sobre el tipo de su argumento. El éxito de la derivación dependerá de la forma en que el cuerpo de la abstracción restrinja el tipo del argumento. En el ejemplo, el cuerpo es la propia variable. La regla de la variable no impone ninguna restricción a su propio tipo: puede probarse que una variable tiene cualquier tipo que le haya sido asociado en una hipótesis. Esto muestra que cualquier deducción de la forma indicada será exitosa y también ilustra la naturaleza polimórfica de la regla de la abstracción funcional.

En el lenguaje presentado se extiende el cálculo λ con construcciones adicionales para términos evaluables. Algunas de estas construcciones imponen restricciones al tipo de sus operandos, comportándose del mismo modo que la aplicación en relación a la abstracción funcional. Véase, por ejemplo:

λx. **pair (x) (+ x (num 1))**

La primera ocurrencia de x en el cuerpo de la abstracción no recibe restricciones sobre su tipo, puesto que pair es polimórfico en sus argumentos. Sin embargo la presencia de x como argumento de + restringe su tipo a number. Como consecuencia, una derivación para el término completo sólo tendrá éxito si se hace la hipótesis x:number, resultando en la única asignación de tipo posible para el término:

λx. pair (x) (+ x (num 1)) : number × number

Esto muestra que el tipo de cualquier ocurrencia de una variable en el cuerpo de la abstracción que la introduce es restringido al tipo de la ocurrencia de tipo menos general. Por supuesto, el razonamiento hecho para las variables de las abstracciones funcionales es válido para cualquier construcción que introduzca variables ligadas.

En principio, puede pensarse que el comportamiento de las constantes en relación a los tipos admite una caracterización análoga a las de las variables. La idea proviene de que la semántica clásica de las declaraciones locales **local** *x e* **in** *d* es usualmente dada como equivalente a la de (λx.*d e*), lo cual podría extenderse de modo similar a las declaraciones globales.

Sin embargo, debe considerarse que las constantes son nombres para términos complejos y eventualmente polimórficos. Si el tratamiento en la asignación de tipos a constantes fuera idéntico al de las variables, distintas ocurrencias de la constante *x* en el cuerpo *d* de la declaración local serían todas restringidas a un mismo tipo. Considérese:

local id λx.x in if (id true) (id num 1) 0

La constante id representa un término polimórfico. Tiene sentido, entonces, que las dos ocurrencias de id en el cuerpo de la declaración puedan ser restringidas a tipos diferentes, en particular: bool → bool y number → number.

Si se tratara la asignación de tipos de la definición de constante como en la aplicación correspondiente:

(λid.if (id true) (id num 1) 0 λx.x)

se tendría un término al que no puede asignarse tipo, pues no existiría ninguna hipótesis sobre el tipo de id compatible con los de sus dos ocurrencias en el cuerpo de la abstracción.

Esto justifica la distinción entre constantes y variables como formas sintácticas y del punto de vista de la asignación de tipos. Las reglas de definiciones locales y globales añaden al contexto la constante introducida asociada al término que ella representa. Luego, la regla de la constante convierte la derivación de un tipo cualquiera para ella en la derivación del mismo tipo para su término asociado. Si el término es polimórfico, distintos tipos para distintas ocurrencias de la constante podrán ser deducidos.

4. Desarrollos complementarios y conclusiones.

4.1 Propiedades del sistema de tipos.

La culminación natural del desarrollo es la demostración de que el sistema de tipos propuesto es sólido (en el sentido de que la evaluación de una sesión bien tipada no puede ser fallida) y la proposición de un algoritmo de inferencia que implemente las reglas del sistema. En relación al primer punto, según lo discutido en 3.1 y 2.3, es suficiente probar que si puede asignarse un tipo a un término evaluable, entonces su evaluación no es fallida.

Esto obliga a caracterizar las demostraciones fallidas de secuentes $\rho \vdash e \Longrightarrow v$ en TermEval. Tal caracterización es engorrosa y conduce a una prueba cuya complejidad es análoga a la de describir un interpretador para el lenguaje, basado en la semantica dada. En contrapartida, la verificación puede hacerse de una forma bastante directa si la semántica se da en la forma de un cálculo de reducciones, extendiendo la de [Holm 89].

Obviamente, el sistema de tipos no es completo, e.e. existen términos evaluables con valor a los que no puede asignarse tipo. Si el sistema fuera completo podría decidirse si un término evaluable tiene o no una evaluación fallida. En el sistema presentado pueden darse casos como:

if true (num 1) (+ true 0)

donde el tercer operando no puede recibir tipo y, por lo tanto, tampoco el término completo, pero éste puede ser evaluado a num 1.

Finalmente, el sistema de tipos es decidible, lo cual es obvio a partir de la forma de sus reglas. Pueden especificarse, entonces, algoritmos de inferencia que implementen este sistema. En [Holm 89] y [CLÉDDK 86] se discute este problema para casos más simples que el aquí presentado. El primero presenta un algoritmo en estilo funcional y el segundo prueba que el sistema puede interpretarse como un programa Prolog. Extender estos resultados para el caso de este trabajo no es difícil aunque sí laborioso.

4.2 Conclusiones.

Se ha definido un sistema de tipos polimórfico para un lenguaje funcional, en presencia de definiciones de constantes globales y locales y de tipos abstractos. Esto extiende los resultados de [Milner 78] y [Holm 89] y formaliza construcciones similares a las de otros lenguajes funcionales.

El uso de sistemas de deducciones para especificar la semántica del lenguaje provee un tratamiento nítido de los contextos de constantes y tipos de los que dependen las evaluaciones de las expresiones. En este aspecto, el planteo es preferible al de semántica de reducciones. Sin embargo, con este último enfoque la propiedad de solidez del sistema de tipos puede probarse fácilmente.

Un algoritmo de inferencia de tipos para este lenguaje ha sido implementado usando el lenguaje Scheme. Otras implementaciones en Prolog son realizables con facilidad a partir de la especificación dada.

Referencias.

[CarWeg 85] **On Understanding Types, Data Abstraction and Polymorphism.** Cardelli L., Wegner P. *ACM Computing Surveys Vol 17, No 4, December 1985.*

[ClÉDDK 86] **A Simple Applicative Language: Mini-ML.** Clément D., Despeyroux J., Despeyroux T., Kahn G. *INRIA - Rapport de Recherche 529, Mai 1986.*

[CouHP 85] **The ML Handbook.** Cousineau G., Huet G., Paulson L. *En Méthodes et Langages de l'Intelligence Artificielle. INRIA, 1985.*

[HinSel 86] **An introduction to combinators and the lambda-calculus.** Hindley J., Seldin J. *Cambridge University Press, 1986.*

[Holm 83] **Polymorphic Type Systems - A Proof Theoretic Approach.** Holmström S. *Programming Methodology Group, University of Goteborg, Report 6, September 1983.*

[Kahn 87] **Natural Semantics.** Kahn G. *INRIA - Rapport de Recherche 601, Février 1987.*

[Milner 78] **A Theory of Type Polymorphism in Programming.** Milner R. *Journal of Computer and System Sciences, Vol. 17, No. 3, 1978.*

[M-Löf 82] **Constructive Mathematics and Computer Programming.** Martin-Löf P. *In Proceedings of the 6th. International Congress for Logic, Methodology and Philosophy of Science, Hannover 1979, North Holland 1982.*

[NorPet 83] **Types and Specifications.** Nordström B., Peterson K. *Programming Methodology Group, University of Goteborg, Report 10, September 1983.*