UN SISTEMA DE TIPOS RECURSIVOS PARCIALMENTE ORDENADO

Juan Manuel García García

Morelia, Mich.
Otoño de 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

A mis padres

A mis hermanos

Agradezco a todas las personas que contribuyeron a la realización de este trabajo, muy especialmente a mis asesores : Hanna Oktaba y Felipe Bracho, por el enorme apoyo y estímulo que me brindaron no solamente a lo largo del desarrollo de esta tesis sino además durante mis estudios de maestria. Doy las gracias de igual forma a Carlos Velarde por sus interesantes observaciones y comentarios que contribuyeron a una mejor exposición del tema. A Luis Gabriel Cruz Valdespino por su ayuda en la edición del texto así como a Edgar L. Chávez, José Ma. Campaña y Nektli Rojas por haberme alentado siempre a seguir adelante.

INDICE

INTRODUCCION

Entre las características importantes de los lenguajes de programación se encuentra la establecida por el sistema de tipos. El tipo de un dato determina los valores permitidos junto con el conjunto de operaciones que pueden utilizarse para manipular esos valores. Un lenguaje variará tanto como los tipos de valores que se permitan, las operaciones que puedan aplicarse a estos y los métodos para controlar la secuencia de esas operaciones.

Si bien en la memoria de la computadora todo lo que se tiene son secuencias de bits, que pueden representar tanto números como caracteres, apuntadores, instrucciones, etc., resulta muy útil organizar estos contenidos de acuerdo a sus propósitos. Los tipos surgen de manera informal para catalogar los datos de acuerdo a su uso y comportamiento. Esta clasificación de los objetos en términos de los propósitos para los cuales son usados eventualmente resulta en sistemas de tipos más o menos bien definidos.

El principal propósito de un sistema de tipos es el evitar que el programador tenga que ocuparse de los detalles de representación. Además de que un programa bien tipificado no hará un uso incorrecto de sus valores a tiempo de ejecución. Por otro lado un valor de un tipo dado tendrá una representación que facilite la realización de la operaciones asociadas. Tenemos entonces que una adecuada tipificación puede redundar en una mayor eficiencia de los programas.

Los lenguajes de programación en los cuales el tipo de una expresión puede ser determinado estáticamente (i.e. a tiempo de compilación) son denominados *tipificados estáticamente*. Sin embargo la propiedad de que todas las variables y expresiones tengan un tipo asociado a tiempo de compilación puede ser muy

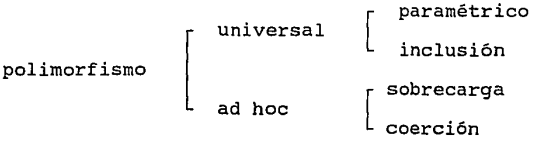
restrictiva. En su lugar podemos pedir que se garantice que todas las expresiones sean consistentes en sus tipos aunque los tipos en sí no puedan ser determinados estáticamente. Esto puede hacerse introduciendo algún mecanismo de verificación de tipos a tiempo de ejecución. A los lenguajes en los cuales todas sus expresiones son consistentes en tipos se les conoce como lenguajes *fuertemente tipificados*. Si un lenguaje es fuertemente tipificado su compilador asegura que los programas que acepta se ejecutaran sin errores de tipos.

Si bien en un lenguaje tipificado estáticamente tenemos que las inconsistencias en el uso de los datos pueden ser detectadas a tiempo de compilación, además de que se tiene una mayor eficiencia, se adolece de una perdida de flexibilidad y poder expresivo. No se tienen, por ejemplo, procedimientos genéricos, tales como los ordenamientos, que sean aplicables de manera uniforme a un rango de tipos.

Sin embargo puede aumentarse la flexibilidad así como la expresividad de los lenguajes fuertemente tipificados mediante la introducción del *polimorfismo*.

Los lenguajes tipificados convencionales son *monomórficos*, en el sentido de que todos los valores y variables tienen un tipo único, en contraste con los lenguajes polimórficos en los cuales algunos valores y variables pueden poseer más de un tipo. Las funciones polimórficas, por ejemplo, son funciones cuyos parámetros pueden tener más de un tipo.

Cardelli y Wegner (1985, pp. 475-479) nos proporcionan una clasificación muy adecuada de las diferentes especies de polimorfismo. De acuerdo a estos autores tendríamos la siguiente clasificación :



Mientras que en el *polimorfismo universal* las funciones polimórficas presentan un comportamiento similar para un rango infinito de tipos que comparten una estructura semejante, en el polimorfismo *ad-hoc* las funciones operan sobre un conjunto finito de tipos, los cuales a veces no están relacionados. En su implementación una función universalmente polimórfica ejecutará el mismo código para argumentos de cualquier tipo admisible mientras que una función polimórfica ad-hoc ejecutará diferente código para cada tipo de argumento.

Existen dos tipos de polimorfismo universal. En el *polimorfismo paramétrico*, una función polimórfica tiene parámetros de tipo que determinan el tipo del parámetro en el momento de la aplicación. En el *polimorfismo por inclusión* se tiene que un mismo objeto puede considerarse como perteneciente a diferentes tipos.

Una variedad de polimorfismo por inclusión se da en la *subtipificación*. En la subtipificación se tiene que los valores de un tipo pueden ser considerados como un subconjunto de los valores de otro tipo. De esta manera, objetos de un subtipo puede ser manipulados uniformemente como pertenecientes a los supertipos.

El polimorfismo por inclusión puede encontrarse en muchos lenguajes comunes, entre los cuales podemos citar a Simula 67, Smalltalk, LISP, Flavors, Amber, etc.

En el presente trabajo estudiaremos un sistema de tipos donde existe una relación de subtipificación entre los tipos. En este sistema de tipos contaremos con un conjunto de tipos básicos así como de un conjunto finito de reglas para construir tipos a partir de los que ya se tienen. De particular interés resulta la construcción de tipos recursivos.

Los tipos de datos recursivos, tales como las listas, pilas, colas, árboles, etc., son de uso común en la programación. La mayoría de los lenguajes de programación no cuentan con maneras

de especificar a estos tipos de datos explícitamente, sino que en su lugar proporcionan mecanismos de bajo nivel, como el uso de apuntadores, para su implementación. Sin embargo el uso de apuntadores puede resultar perjudicial para la programación. Puede resultar nocivo en la misma medida en que lo es el uso del salto incondicional (goto). Un lenguaje de programación que permita construir tipos recursivos hará en sus programas un uso más disciplinado de la memoria dinámica.

Un problema central a tratar en la presente tesis será la definición de una noción adecuada de subtipos en los tipos de datos recursivos.

Desarrollaremos nuestro sistema de tipos sobre el cálculo lambda. En nuestro enfoque consideraremos al λ-cálculo como una especie de lenguaje de programación cuyas expresiones tipificaremos. De hecho, el cálculo lambda es considerado como el primer lenguaje de programación funcional, aún cuando originalmente no fue pensado como un lenguaje de programación puesto que en la época en que se originó no existían todavía las computadoras.

El trabajo original de Church sobre el λ-cálculo (presentado en su obra *The Calculi of Lambda Conversion*, 1941) fue motivado por el propósito de crear un cálculo que capturara las nociones de función y aplicación de funciones. Este enfoque difiere del tradicional (que considera a las funciones como conjuntos) en que trata sobre los aspectos *computacionales* de las funciones. Esta teoría posteriormente fundamentaría el desarrollo de los lenguajes funcionales, entre los que podemos mencionar al Lisp, ML, Iswim, APL, FP, Miranda, Haskell, etc.

En el primer capítulo del presente trabajo haremos una breve revisión de los aspectos esenciales del cálculo lambda puro (es decir, no tipificado), con el objeto de sentar las bases para la posterior discusión. A partir del segundo capítulo , comenzaremos a estudiar las propiedades de nuestro lambda cálculo tipificado con tipos recursivos y relación de orden parcial en tipos.

El sistema de tipos que es objeto del presente estudio, fue propuesto por Luca Cardelli y Roberto Amadio, quienes resolvieron el problema de la definición de una noción adecuada de subtipos dentro de los tipos recursivos, además de encontrar un algoritmo para decidir cuando un tipo es subtipo de otro. El algoritmo de Cardelli y Amadio tiene la desventaja de ser de orden exponencial. Dexter Kozen, Palsberg y Schwartzbach son los autores de un algoritmo de subtipificación más eficiente; concretamente, de orden cuadrático.

Una aportación de esta tesis es la extensión del algoritmo de Kozen, Palsberg y Schwartzbach, para la construcción de un operador de coerción. Esto es, desarrollamos un algoritmo que para un par de tipos en relación de subtipos construye un término lambda bien tipificado que es un mapeo del subtipo al supertipo y que además preserva la estructura de los tipos de datos. El estudio de las propiedades de los operadores de coerción será una parte importante en el desarrollo de nuestro análisis.

CAPITULO 1

CALCULO LAMBDA PURO

1.1. Cálculo lambda sin tipos.

En la notación usual en matemáticas una función, en el sentido de una regla que asocia a un elemento de un conjunto llamado dominio un elemento único de un conjunto denominado rango, se denota mediante expresiones de la forma :

$$F(x) = x^2-1$$

En la ecuación anterior se declara a una función F en la cual a cada valor x en el dominio se le asocia el valor de x^2-1. Es decir, F denotaría a la *abstracción funcional* :

$$x \mapsto x^2-1$$

Una manera equivalente de denotar a la misma abstracción funcional es mediante la expresión :

$$\lambda x.(x^2-1)$$

La notación anterior es conocida como *notación lambda*. En la notación matemática usual F(5) denota al valor resultante de *aplicar* al valor 5 la función F. En la notación lambda, la *aplicación funcional* se denotaría mediante la expresión :

$$(\lambda x.(x^2-1))5$$

El *cálculo lambda* es simplemente un cálculo de aplicación de funciones basado en esta notación. En el cálculo lambda, la aplicación de funciones se reduce a un proceso de substitución.

Por ejemplo, supongamos que hemos definido a

$$F \equiv \lambda x.\,(x^2-1)$$

y queremos determinar el valor de F(5); podemos encontrarlo substituyendo 5 por x en la expresión acotada por esa variable. Más específicamente, para calcular F(5) substituimos a F por la expresión lambda, esto es :

$$\lambda x.\,(x^2-1)\,(5)$$

La expresión se reduce a una copia del *cuerpo* de F (i.e. x^2-1) en la cual toda ocurrencia libre de x es sustituida por (5).

$$[5/x]\,(x^2-1) \rightarrow (5^2-1) \rightarrow (25-1) \rightarrow 24$$

Un aspecto importante sobre el λ-cálculo es que en este, a diferencia del concepto usual de funciones en teoría de conjuntos, no existen las nociones de *dominio* y *rango* de una función, sino que una expresión (en λ-cálculo puro) puede aplicarse a cualquier otra expresión, llegando a darse el caso en el cual una función se aplique a si misma. Lo anterior no es posible de realizar dentro de la teoría clásica de conjuntos sin caer en paradojas bien conocidas.

El λ-cálculo es una teoría elegante y sencilla que sirve como marco de referencia para el estudio de la computabilidad así como de la semántica de los lenguajes de programación. En el presente trabajo enfocamos al λ-cálculo como un lenguaje de programación muy simple pero lo suficientemente poderoso como para que cualquier otro lenguaje de programación pueda ser definido en términos de este.

Este capítulo es una introducción muy elemental al cálculo lambda puro, únicamente con el propósito de introducir el "lenguaje de programación" sobre el cual construiremos el sistema de tipos que es objeto del presente trabajo. Para un estudio más profundo del cálculo lambda puro remitimos al lector a la bibliografía, en especial a Barendregt (1984).

Con el objeto de hacer más legibles las expresiones del cálculo lambda introduciremos alguna "azúcar" sintáctica. Para usar un número menor de paréntesis adoptaremos la convención de que todos los operadores se asocian a la izquierda. Por ejemplo *((F E) G)* puede abreviarse como *(F E G)*, que a su vez puede abreviarse *F E G*. Sin embargo, *(F (E G))* puede abreviarse como *F (E G)* pero no *F E G*. De manera similar, *(λx.(f x))* puede ser abreviada como *λx.(f x)* y (λx.x) puede abreviarse λx.x. En ocasiones, dentro del presente capítulo, introduciremos expresiones aritméticas dentro de las expresiones del λ-cálculo, pero será únicamente con fines ilustrativos.

1.3. Reglas de reducción.

En la sección anterior afirmamos que el cálculo lambda es básicamente un lenguaje y además expusimos las reglas sintácticas que nos permiten formar a las expresiones de ese lenguaje. Además existe un aspecto manipulativo o computacional, esto es, un conjunto de reglas de reescritura que nos permite convertir unas cadenas en otras. Esta es la razón de que le otorguemos el nombre de "cálculo" al λ-cálculo.

En esta sección precisaremos la idea de substitución de variables así como las nociones de reglas de reescritura que utilizamos libremente en la sección 1.1.

Definición 1.1. Definimos al *conjunto de variables libres* de una expresión e, que denotamos como $fv(e)$, inductivamente como :

$$fv(x) = \{x\}$$

$$fv(e_1\ e_2) = fv(e_1) \cup fv(e_2)$$

$$fv(\lambda x.e) = fv(e) - \{x\}$$

Por ejemplo, sea *E* la expresión lambda dada por :

$$(\lambda x.(\lambda y.(x\ y\ z)))$$

entonces el conjunto de variables libres de E, esto es $fv(E)$, estará dado por :

$$\begin{aligned} fv(E) &= fv(\lambda x.(\lambda y.(x\ y\ z))) = fv(\lambda y.(x\ y\ z)) - \{x\} \\ &= (\ fv((x\ y)\ z) - \{y\}\ ) - \{x\} \\ &= (\ (fv(x\ y) \cup fv(z)) - \{y\}\ ) - \{x\} \\ &= (\ \{x,y,z\} - \{y\}\ ) - \{x\} = \{z\} \end{aligned}$$

En la expresión $\lambda x.e$, se dice que x es un identificador *acotado*. Un identificador es *libre* en una expresión e si y solo si pertenece a $fv(e)$. En el ejemplo anterior, x e y son variables acotadas en E mientras que z es una variable libre.

Diremos que una expresión es *cerrada* si no contiene variables libres y será *abierta* en caso contrario.

A continuación precisaremos la noción de *substitución* de una variable en una expresión.

Definición 1.2. Denotaremos simbólicamente como $[e_1/x]e_2$ a la *substitución por* e_1 *de las ocurrencias de* x *en* e_2, que definiremos inductivamente como sigue :

$$[e/x_i]x_j = \begin{cases} e & si\ i=j \\ x_j & si\ i\neq j \end{cases}$$

$$[e_1/x_i](\lambda x_j.e_2) = \begin{cases} \lambda x_j.e_2 & si\ i=j \\ \lambda x_j.[e_1/x_i]e_2 & si\ i\neq j\ y\ x_j \notin fv(e_1) \\ \lambda x_k.[e_1/x_i]([x_k/x_j]e_2) & donde\ k\neq i,\ k\neq j, \\ & x_k \notin fv(e_1) \cup fv(e_2) \end{cases}$$

$$[e/x_i](e_1\ e_2) = ([e/x_i]e_1\ [e/x_i]e_2)$$

Para completar el cálculo lambda, necesitamos definir las reglas que nos permiten convertir o reescribir una λ-expresión en otra.

Definición 1.3. Definiremos las tres reglas de reescritura para λ-expresiones :

(1) α-conversión (*renombramiento*)

$$\lambda x_i.e \rightarrow \lambda x_j.[x_j/x_i]e \quad \textit{donde } x_j \notin fv(e)$$

(2) β-conversión (*aplicación*)

$$(\lambda x.e_1)e_2 \rightarrow [e_2/x]e_1$$

(3) η-conversión

$$\lambda x.(e\ x) \leftrightarrow e \quad \textit{si } x \notin fv(e)$$

La primera regla nos indica que una expresión puede ser convertida en otra mediante la sustitución de una variable acotada dentro de su ámbito por cualquier otro identificador que no ocurra en el ámbito, mientras que en la segunda regla se precisa la noción de que una aplicación funcional se reduce a una sustitución en el cuerpo de la abstracción de el "parámetro formal" por el "parámetro actual". Por otra parte, la tercer regla nos indica que una función en cuyo cuerpo no ocurre el parámetro de la función es tratada como una constante.

Si las dos ultimas reglas de reescritura solo se usan en un solo sentido tenemos las *reglas de reducción*, que serían las siguientes :

(1) β-reducción

$$(\lambda x.e_1)e_2 \Rightarrow [e_2/x]e_1$$

(2) η-reducción

$$\lambda x.(e\ x) \rightarrow e \quad \textit{si } x \notin fv(e)$$

Diremos que *una expresión X se reduce a una expresión Y* (expresado simbólicamente como X ⇒ Y) si X puede ser convertida

a Y por cero o más aplicaciones de las reglas de reducción.

Ejemplos de la aplicación de α-reducciones son los siguientes :

(i) $\lambda x.x \Rightarrow \lambda a.a \Rightarrow \lambda y.y$

(ii) $\lambda x.(x+1) \Rightarrow \lambda u.(u+1)$

(iii) $\lambda x.(\lambda y.(x+y)) \Rightarrow \lambda a.(\lambda y.(a+y)) \Rightarrow \lambda a.(\lambda b.(a+b))$

Mientras que algunos ejemplos de β-reducciones serían :

(iv) $(\lambda x.x)\ y \Rightarrow y$

(v) $\lambda y.(\lambda x.(\lambda z.(z+x)\ 3)\ y)\ 1 \Rightarrow \lambda y.(\lambda x.(3+x)\ y)\ 1$
$\Rightarrow \lambda y.(3+y)\ 1 \Rightarrow 3+1$

Como podemos observar en el ejemplo (iv), la λ-expresión (λx.x) aplicada a cualquier otra λ-expresión se reduce a esta última siempre. Es por esta razón que podríamos "interpretar" a (λx.x) como la función identidad.

Por otro lado, en el ejemplo (v) las reducciones se llevaron a cabo de la expresión más interna hasta la más externa. Podemos verificar que al realizar la reducción en otro orden se llega al mismo valor final. Por ejemplo, evaluando de la expresión más externa hacia la más interna se tiene :

$\lambda y.(\lambda x.(\lambda z.(z+x)\ 3)\ y)\ 1 \Rightarrow \lambda x.(\lambda z.(z+x)\ 3)\ 1$
$\Rightarrow \lambda z.(z+1)\ 3 \Rightarrow 3+1$

Esta propiedad del λ-cálculo que radica en que el valor final arrojado por la reducción de una expresión no depende del orden de evaluación de la misma, es muy importante y por lo tanto abundaremos sobre ello a continuación.

1.4. Formas normales y confluencia.

Definición 1.3. Decimos que una λ-expresión está en su *forma normal* si no puede ser reducida usando β o η-reducción.

Por ejemplo, algunas λ-expresiones y sus correspondientes formas normales serían las siguientes :

No normal	Normal
(λx.x) y	y
λy.(f y) (a)	f a
(λx.x) (λx.x)	(λx.x)
λx.(x x) y	y y
λx.(x y) (λx.(x x))	y y

Sin embargo no todas las λ-expresiones tienen una forma normal. Considérese la expresión definida como :

$$\Omega \equiv \lambda x.(x\ x)\ \lambda x.(x\ x)$$

y sea ω = λx.(x x), entonces :

$$\Omega \Rightarrow \omega\ \omega \Rightarrow \lambda x.(x\ x)\ \omega \Rightarrow \omega\ \omega \Rightarrow \ldots$$

de donde puede observarse que a Ω puede aplicarsele indefinidamente la β-reducción, por lo cual esta λ-expresión no es normalizable.

Dado que Ω lleva a reducciones sin fin, podríamos pensar, de acuerdo a la noción de "aplicación" formalmente presentada que una expresión está indefinida si contiene a Ω. Observemos en particular la expresión (λx.a) Ω; si primero reducimos la parte derecha de la expresión, se tiene :

$$(\lambda x.a)\ \Omega \Rightarrow (\lambda x.a)\ (\omega\ \omega) \Rightarrow (\lambda x.a)\ (\omega\ \omega) \Rightarrow \ldots$$

pero evaluando la parte izquierda se tendría :

$$(\lambda x.a)\ \Omega \Rightarrow a$$

Hemos visto entonces que, dependiendo del orden de evaluación se puede o no alcanzar la forma normal.

Sin embargo, si una expresión puede ser reducida a una forma normal está forma normal es única. Esto puede deducirse de una propiedad más fuerte denominada *confluencia* que posee el λ-cálculo y que definiremos enseguida :

Definición 1.4. Decimos que una relación **R** sobre un conjunto **S** es *confluente* si y solo si para todos los X, X_1, X_2 elementos de **S** se cumple que :
Si tenemos que XRX_1 y XRX_2 entonces existe una X' en **S** tal que X_1RX' y X_2RX'.

A continuación enunciaremos un importante teorema, demostrado por primera vez por Alonzo Church y J. Barkley Rosser en 1936. Para su demostración y la de su corolario, remitimos al lector a la bibliografía anteriormente citada.

Teorema 1.1. (Church-Rosser) La reducción en el cálculo lambda es confluente.

Corolario. Si una λ-expresión tiene forma normal entonces esta es única (excepto por renombramiento).

1.5. Operador de punto fijo.

A continuación introduciremos un operador que es central en la teoría de λ-cálculo puro y que también será particularmente interesante para la teoría que desarrollaremos en capítulos posteriores; es el denominado *operador de punto fijo* por razones que expondremos a continuación.

Consideremos la λ-expresión **W** definida como :

$$W \equiv \lambda x.F(x\ x)$$

para un operador F en particular. Observemos que al aplicar β-reducción se tiene :

$$W\ W \equiv \lambda x.F(x\ x)\ W \rightarrow F(W\ W)$$

Dado que $W\ W \Rightarrow F(W\ W)$, decimos que $W\ W$ es punto fijo del operador F, y para hacer nuestro resultado independiente de F, hacemos abstracción funcional obteniendo el combinador :

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

El anterior es el *operador de punto fijo* que aplicado a cualquier operador nos devuelve un punto fijo, esto es, para cualquier operador F se cumple que :

$$Y\ F \rightarrow F\ (Y\ F) \rightarrow F\ (F\ (Y\ F)) \rightarrow \ldots$$

Este operador es particularmente útil cuando se requiere definir alguna función de manera recursiva. Para ejemplificar el uso de el operador anterior en definiciones recursivas, considérese a la función factorial :

$$fact \equiv \lambda n.if(n{=}0, 1, n*(fact\ \ n{-}1))$$

en donde suponemos que la expresión :

$$if(\exp_1, \exp_2, \exp_3)$$

es tal que devuelve el resultado de evaluar $\exp_2$ en el caso de que $\exp_1$ sea verdadero y devuelve el valor de $\exp_3$ en caso contrario.

Si consideramos al operador :

$$\Psi \equiv \lambda f.\lambda n.if(n{=}0, 1, n*(f\ \ n{-}1))$$

entonces tenemos que :

$$fact = \Psi(fact)$$

es decir, *fact* es punto fijo del operador Ψ y por lo tanto

podemos afirmar que :

$$fact = Y(\Psi)$$

o en forma más explícita :

$$fact = Y(\lambda f.\lambda n.if(n=0,1,n*(f\ n-1)))$$

Es de esta manera como una definición recursiva de una función puede expresarse en λ-cálculo usando el operador de punto fijo.

En capítulos posteriores volveremos a tratar sobre el operador de punto fijo, pero ya en el contexto de un λ-cálculo tipificado.

1.6. Resumen.

En este capítulo hemos introducido el λ-cálculo como un lenguaje formal, además de un conjunto de reglas de manipulación de las expresiones del lenguaje. Hemos también expuesto algunas propiedades centrales de este cálculo, entre ellas la enunciada en el teorema de Church-Rosser.

En el cálculo lambda puro que hemos presentado no existe ninguna tipificación de sus expresiones, es decir, las expresiones de este lenguaje son indistintas en el mismo sentido en el que, en principio, no hay ninguna distinción entre las instrucciones y los datos que se encuentran almacenados en la memoria de una computadora.

A partir del siguiente capítulo introduciremos la noción de "tipo" en las expresiones del λ-cálculo, lo cual nos permitirá hacer una distinción entre "operaciones" y "valores" a los cuales se pueden aplicar estas.

CAPITULO 2

UN CALCULO LAMBDA TIPIFICADO

En el presente capítulo introduciremos un λ-cálculo tipificado donde además tenemos la capacidad de definir tipos de datos recursivos. Todo nuestro trabajo se desarrolla en base a las propiedades de este λ-cálculo en particular.

2.1. Tipos.

En los lenguajes de programación un tipo es interpretado normalmente como una colección de valores que comparten una estructura similar o un conjunto común de operaciones bajo las cuales presentan un comportamiento semejante.

Así el concepto de tipo es necesario en la programación para regular el uso apropiado de los datos dentro de ciertos contextos. Al tipificar, esto es, al asignar un tipo a una variable, asumimos el rango de valores que esta variable puede tomar así como las operaciones que es válido realizar sobre ésta. La tipificación está pues muy relacionada con la validez de un programa. Un programa fuertemente tipificado no hará un uso inadecuado de los valores con los que trabaja a tiempo de ejecución.

Más aún, la tipificación puede considerarse un mecanismo efectivo de abstracción. Al restringir los valores que pueden asumir las variables podemos ignorar los detalles irrelevantes de la implementación subyacente así como identificar a estas variables con elementos abstractos de un conjunto. Por ejemplo, resulta particularmente útil en la practica de la programación pensar en los datos como enteros, caracteres, reales, booleanos, etc. y no simplemente como secuencias de bits.

La mayoría de los estudios teóricos sobre tipos se han centrado en λ-cálculos tipificados, considerados ya sea como lenguajes de programación funcional muy simples o como metalenguajes para entender otro tipo de lenguajes (entre ellos los imperativos). De manera general, un λ-cálculo tipificado consiste de una sintaxis de expresiones, un sistema de inferencia de tipos así como de un conjunto de reglas de reducción, que nos indican como hacer "cálculos" con las expresiones. En ocasiones se proporciona una semántica formal de la expresiones y sus tipos.

Los λ-cálculos tipificados proporcionan modelos útiles de lenguajes de programación, si bien no logran captar completamente todas las intrincadas características que pueden tener los lenguaje prácticos como Pascal, Ada o Lisp. Por ejemplo, una diferencia relevante entre lambda cálculo y muchos de los lenguajes de programación es la ausencia de la asignación. Existe sin embargo una categoría de lenguajes cuyas características, entre ellas la ausencia de efectos laterales, están fuertemente basadas en las del cálculo lambda. Estos son los lenguajes funcionales, denominados de esa manera por estar basados en el cálculo de funciones, entre los que se encuentran ML, Haskell, etc. y que son objeto de interés y estudio cada vez más amplios (véase Hudak,1989).

Los λ-cálculos tipificados tienen en común estar basados en un cálculo de funciones como el que hemos presentado en el capitulo anterior, en donde la construcción principal es la *abstracción funcional* que usamos para definir funciones. Además tenemos la *aplicación* que nos permite hacer uso de las funciones que hemos definido. El dominio de la función se especifica asignando un tipo al parámetro formal de la abstracción. Si M es una expresión, en donde puede ocurrir la variable x de tipo σ (abreviado como $x{:}\sigma$), entonces $\lambda x{:}\sigma.M$ es la función definida al considerar a x como un parámetro en M. Dado que $x{:}\sigma$ especifica explícitamente que el parámetro formal x es de tipo σ, el dominio de $\lambda x{:}\sigma.M$ es σ. El rango de $\lambda x{:}\sigma.M$ estaría determinado a partir de la forma de M usando las reglas de tipificación del lenguaje.

La mayoría de los lenguajes de programación inician con un conjunto de tipos de datos básicos, como los numeros naturales, los booleanos, los caracteres, etc. y además proporcionan diferentes maneras de construir funciones y estructuras de datos más complicadas sobre estos. La descripción de las maneras en que se construyen nuevos valores a partir de los anteriores es lo que se conoce como un sistema de tipos. Un resumen de las principales características de los sistemas de tipos mas relevantes se encuentra en Mitchell (1989).

El sistema de tipos que estudiaremos en el presente trabajo fue propuesto originalmente por Cardelli y Amadio (1990).

Dentro del sistema de tipos, que estamos a punto de definir formalmente, tenemos ciertos *tipos básicos* tales como *Unit*, el tipo unitario que solamente tiene un valor, *Int*, el tipo entero, *Bool*, el tipo de los valores booleanos, etc. Además estarían los *tipos estructurados* los cuales construiríamos a partir de los tipos básicos mediante el empleo de tres constructores de tipos : el producto cartesiano, la unión disjunta y el mapeo entre tipos. Por ejemplo :

Int × Int, los pares ordenados de enteros;
Unit + Int, la unión disjunta de Unit e Int;
Int → Int, funciones de enteros a enteros.

2.2. Tipos recursivos.

Por otra parte dentro de nuestro sistema podremos definir *tipos de datos recursivos*.

Las estructuras de datos recursivas tales como las listas, pilas, colas o los árboles son frecuentemente utilizados en la práctica de la programación de computadoras. La mayoría de los lenguajes de programación (entre ellos Pascal, C y Ada) proporcionan mecanismos de bajo nivel, como es el uso de apuntadores, para la creación y manipulación de estas estructuras de datos. Sin embargo podríamos afirmar que el uso de apuntadores

es tan perjudicial en la programación como puede serlo el utilizar como estructura de control al salto incondicional (goto). De la misma manera en que la introducción de estructuras de control de alto nivel eliminó la necesidad de los saltos incondicionales, podríamos decir que la posibilidad de definir tipos de datos recursivos en los lenguajes de programación eliminaría por completo la necesidad del uso de apuntadores, haciendo así a los programas más legibles y correctos.

Los tipos de datos recursivos pueden considerarse como soluciones a ecuaciones recursivas. Por ejemplo denotaremos como:

$$\mu t.\mathit{Unit}+(\mathit{Int}\times t) \tag{2.1}$$

a la solución (en caso de que esta exista) de la ecuación de la forma :

$$t = \mathit{Unit}+(\mathit{Int}\times t) \tag{2.2}$$

Ejemplos comunes de tipos de datos recursivos son los siguientes :

$$\mathit{Tree} = \mathit{Unit} + \mathit{Int}\times(\mathit{Tree}\times\mathit{Tree}) \tag{2.3}$$

el conjunto de árboles binarios con nodos de tipo entero, y :

$$\mathit{List} = \mathit{Unit} + (\mathit{Int}\times\mathit{List}) \tag{2.4}$$

la colección de listas de enteros, y que es el mismo tipo de ecuación que mencionamos anteriormente.

Entonces, de manera equivalente decimos que :

$$L \doteq \mu t.\mathit{Unit}+(\mathit{Int}\times t) \tag{2.5}$$

si L satisface la ecuación de listas, esto es si la igualdad :

$$L = \mathit{Unit}+(\mathit{Int}\times L) \tag{2.6}$$

es demostrable.

Para determinar cuando un valor pertenece a un tipo

recursivo necesitamos introducir la noción de *desdoblamiento* (*unfolding*). Denotaremos por desdoblamiento del tipo recursivo :

$$\mu t.\alpha \tag{2.7}$$

a la sustitución :

$$[\mu t.\alpha/t]\alpha \tag{2.8}$$

Entonces diremos que un valor pertenece a un tipo de dato recursivo si pertenece a uno de sus desdoblamientos. Esto particularmente tiene sentido cuando el tipo de dato recursivo es tal que al hacer el desdoblamiento se tienen uniones disjuntas con tipos de datos no recursivos. Para ilustrar este punto, considérese al tipo de dato L definido anteriormente en (2.5).

El primer desdoblamiento estaría dado por :

$$\begin{aligned}&[\mu t.Unit+(Int\times t)/t]\,(Unit+(Int\times t))\\ &= Unit+(Int\times(\mu t.Unit+(Int\times t)))\end{aligned} \tag{2.9}$$

en donde tenemos una unión disjunta de un tipo de dato no recursivo, en este caso Unit, y un tipo de dato que es producto cartesiano de Int con nuestro tipo recursivo. De lo anterior podemos deducir que Unit es una subcolección del tipo L.
En el segundo desdoblamiento tenemos :

$$Unit+Int\times Unit+(Int\times Int\times(\mu t.Unit+(Int\times t))) \tag{2.10}$$

de donde los valores de Int × Unit también están contenidos en L. Hasta aquí tenemos a la lista vacía (contenida en Unit) y a las listas de un entero (pares ordenados del tipo Int × Unit). Puede verse fácilmente que los sucesivos desdoblamientos del tipo L nos dan sumas disjuntos de tipos de la forma Int × ... × Int × Unit que corresponden a nuestra noción de listas de enteros.

En el anterior ejemplo hemos manejado de manera implícita la noción de expansiones finitas de un tipo de dato recursivo así como la noción de subtipos de un tipo dado. Todas estas nociones serán precisadas y formalizadas posteriormente.

2.3. Términos tipificados.

Comencemos por introducir la sintaxis de las expresiones con las cuales denotaremos a los tipos, es decir, las expresiones de tipos. En una notación BNF informal, las expresiones de tipos quedarían definidas como sigue :

$$\alpha ::= t \mid \bot \mid \top \mid (\alpha \rightarrow \beta) \mid (\alpha + \beta) \mid (\alpha \times \beta) \mid \mu t.\alpha \qquad (2.11)$$

en donde t es una variable de tipo. Denotaremos por **TVar** al conjunto de variables de tipo t,s,... y **Type** será el conjunto de expresiones de tipo dadas por (2.11). Además distinguiremos como **Tp** al subconjunto de expresiones de tipos no-recursivas dadas por:

$$\alpha ::= t \mid \bot \mid \top \mid (\alpha \rightarrow \beta) \mid (\alpha + \beta) \mid (\alpha \times \beta) \qquad (2.12)$$

y denotaremos por μ**Tp** al subconjunto de expresiones de tipo en forma canónica con la sintaxis :

$$\begin{aligned} \alpha ::= {} & t \mid \bot \mid \top \mid (\alpha \rightarrow \beta) \mid (\alpha + \beta) \mid (\alpha \times \beta) \\ & \mid \mu t.\alpha \rightarrow \beta \mid \mu t.\alpha + \beta \mid \mu t.\alpha \times \beta \end{aligned} \qquad (2.13)$$

donde en $\mu t.\alpha \rightarrow ß$, $\mu t.\alpha + ß$ y $\mu t.\alpha \times ß$, t debe ocurrir libre en α y ß.

En los constructores de tipos omitiremos en algunas ocasiones los paréntesis suponiendo la siguiente precedencia en los constructores de tipos : el operador de producto cartesiano $\times$ tiene la precedencia más alta y el operador de unión disjunta + tiene la más baja. Por ejemplo :

$$\alpha + \beta \times \gamma \quad es \quad \alpha + (\beta \times \gamma) \qquad (2.14)$$

Además estableceremos un conjunto de reglas para asociar un tipo a cada expresión del λ-cálculo. Como indicamos anteriormente, usaremos la notación :

$$x{:}\sigma$$

para indicar que la variable x es de tipo σ.

Si tenemos que $x:\alpha$ y $y:\beta$, entonces la pareja ordenada $\langle x,y \rangle$ debe ser del tipo $\alpha \times \beta$, esto es :

$$\langle x,y \rangle : \alpha \times \beta$$

y también :

$$\text{inl}(x) : \alpha + \beta$$
$$\text{inr}(y) : \alpha + \beta$$

Las anteriores reglas no se mencionan en el λ-cálculo propuesto por Cardelli-Amadio pero nosotros las hemos incluido para tener la posibilidad de formar términos que pertenezcan a los tipos producto y unión. Las reglas que introduciremos a continuación han sido establecidas por Cardelli-Amadio.

Si M es una expresión de tipo β, en donde puede ocurrir la variable x de tipo α, se tiene que la abstracción $\lambda x{:}\alpha.M$ puede considerarse como una función cuyo dominio es el tipo α y cuyo rango es de tipo β, esto es :

$$(\lambda x.M) : \alpha \rightarrow \beta$$

Por otra parte, si M es una función de dominio α y de rango β, esto es $M:\alpha \rightarrow \beta$, entonces puede aplicarse a un término N de tipo α y por lo tanto se tiene :

$$(M\ N) : \beta$$

Incluiremos también la regla que especifica que un término que pertenece a un tipo recursivo puede también considerarse perteneciente a los desdoblamientos del tipo. Esto es, si $M:\mu t.\alpha$ entonces :

$$(unfold_{\mu t.\alpha}\ M) : [\mu t.\alpha / t]\,\alpha$$

en donde el operador de desdoblamiento $unfold_{\mu t.\alpha}$ es el operador de *coerción* que explícitamente asocia a un término de tipo recursivo el término correspondiente del tipo desdoblado.

De manera similar, introducimos una regla que nos permite llevar un término de tipo desdoblado al correspondiente tipo doblado. Es decir, si $M:[\mu t.\alpha/t]\alpha$ entonces :

$$(fold_{\mu t.\alpha}\ M) : \mu t.\alpha$$

en donde ahora $fold_{\mu t.\alpha}$ es el operador de coerción que lleva del tipo desdoblado al tipo sin desdoblamiento.

Se tiene entonces que *fold* y *unfold* forman una familia de operadores ya que existe un par de ellos para cada tipo recursivo. Sin perder generalidad, haremos referencia a esos operadores como si fuesen únicos, omitiendo el subíndice de tipo entendiendo que se trata del operador correspondiente al tipo recursivo del término al que se aplique.

2.4. Reglas de conversión.

Para completar la definición de nuestro λ-cálculo tipificado tenemos que añadir las reglas de conversión y reducción que, como señalamos anteriormente, son las que nos proporcionan el aspecto "computacional" o, de manera más especifica, nos indican como se "evalúan" las expresiones de nuestro lenguaje.

En primer lugar tendríamos a la α, β y η-conversión, que son prácticamente las mismas que teníamos en λ-cálculo puro.

$$(\lambda x.M) \leftrightarrow (\lambda y.\,[y/x]M) \quad \textit{donde}\ \ y \notin fv(M) \qquad (\alpha)$$

$$(\lambda x.M)\,N \leftrightarrow [N/x]\,M \qquad (\beta)$$

$$\lambda x.\,(M\,x) \leftrightarrow M \quad \textit{si}\ x \notin fv(M) \qquad (\eta)$$

Introduciremos además la regla que establece que los operadores *fold* y *unfold* son inversos, es decir :

$$fold(unfold\ x) \leftrightarrow x \qquad unfold(fold\ x) \leftrightarrow x \qquad (\mu)$$

Como consecuencia inmediata de la regla anterior tenemos que un tipo recursivo es isomorfo con su desdoblamiento, de acuerdo a nuestra intuición de que el desdoblamiento de un tipo es solamente una manera más explícita de enunciarlo, tal como pudimos observar en la sección 2.2.

Además de estas reglas de reducción que hemos tomado del citado artículo de Cardelli y Amadio, añadiremos en el siguiente capítulo reglas de reducción para términos pertenecientes a los tipos producto y unión. Dichas reglas se requieren introducir únicamente para poder hacer reducibles los términos de estos tipos y de ese modo tener un cálculo lambda más práctico, sin que esto altere en lo esencial al sistema de tipos propuesto.

Cardelli y Amadio (1990, pág.9) enuncian además las tres siguientes reglas de reducción :

$$[\mu t.\alpha/t]\,\alpha \twoheadrightarrow \mu t.\alpha$$

$$x{:}\bot \twoheadrightarrow y{:}\bot$$

$$x{:}\top \twoheadrightarrow y{:}\top$$

De la primera de ellas podemos decir que quizás no sea muy adecuado establecer tal regla en este punto, puesto que estamos dando las normas para la reducción de *términos*, no de los *tipos*. Sin embargo cabe aclarar que dicha regla sería importante en el contexto de un λ-cálculo tipificado de segundo orden, es decir, un λ-cálculo en el cual tuviéramos reglas para transformar tipos de modo tal que un mismo término pudiera tener varios posibles tipos. Este sería entonces un tratamiento muy adecuado del *polimorfismo*. En el presente trabajo hemos preferido elaborar sobre una teoría de primer orden. Las otras dos reglas, además de estar poco fundamentadas, son irrelevantes para propósitos de nuestro estudio. Por estás razones decidimos prescindir de estas reglas que aparecen en el trabajo original de Cardelli-Amadio.

2.5. Subtipos.

Si entendemos a los tipos intuitivamente como colecciones de valores, a las subcolecciones les denominaremos subtipos. Más formalmente, la relación de subtipo representa una relación de orden parcial dentro de los tipos.

El tipo básico $\perp$ (*bottom*) es subtipo de cualquier otro tipo y por otra parte, todos los tipos son subtipos de $\top$ (*top*). Es decir, para todo α en **Type** se cumplen las relaciones:

$$\perp < \alpha \qquad \alpha < \top$$

Las funciones tienen una regla de subtipificación que es *antimonotónica* en el primer argumento, esto es :

$$\alpha \to \beta < \alpha' \to \beta' \quad si \quad \alpha' < \alpha \quad y \quad \beta < \beta'$$

Por ejemplo, tenemos que si Nat es el tipo de los números naturales e Int es el tipo de los enteros se cumple que Nat < Int y si f:Int→C es una función cuyo argumento es un entero, en particular puede tomar valores naturales como argumento, por lo cual también f puede considerarse del tipo Nat→C, de donde resulta lógico afirmar que Int→C < Nat→C.

Para las otras construcciones de tipos podemos encontrar reglas de subtipificación adecuadas. Para los productos cartesianos tenemos :

$$\alpha \times \beta < \alpha' \times \beta' \quad si \quad \alpha < \alpha' \; y \; \beta < \beta'$$

y para la suma disjunta se tiene :

$$\alpha + \beta < \alpha' + \beta' \quad si \quad \alpha < \alpha' \; y \; \beta < \beta'$$

Para los tipos de datos recursivos no resulta tan sencillo determinar una regla de subtipificación apropiada. Por ejemplo, podríamos afirmar que :

$$\text{Si } \alpha < \beta \text{ entonces } \mu t.\alpha < \mu t.\beta \qquad (2.15)$$

Con esta regla podríamos deducir, por ejemplo, que :

$$\mu t.\, \top \to t < \mu t.\, \perp \to t$$

dado que $\perp < \top$ y por lo tanto $\top \to t < \perp \to t$. De manera similar tendríamos :

$$\mu t.t\to\bot < \mu t.t\to\top \qquad (2.16)$$

pero esto implicaría, por desdoblamiento de (2.16) a la relación

$$(\alpha\to\bot)\to\bot < (\beta\to\top)\to\top \qquad (2.17)$$

donde $\alpha = \mu t.t\to\bot$ y $\beta = \mu t.t\to\top$, y esta no satisface la relación de antimonoticidad. La relación anterior implicaría que $(\beta\to\top) < (\alpha\to\bot)$ y esto a su vez requeriría que $\alpha < \beta$ y, de manera contradictoria, $\top < \bot$.

En capítulos posteriores abordaremos con detalle el problema de la subtipificación de tipos recursivos.

2.6. Resumen.

En este capítulo hemos expuesto al λ-cálculo tipificado que será objeto de estudio en el presente trabajo, y al cual denominaremos a partir de este momento cálculo $\lambda^{\to,\mu}$, siguiendo un poco la notación utilizada en Mitchell (1989).

El cálculo $\lambda^{\to,\mu}$ está dado por las expresiones tipificadas construidas por *identificadores*, *abstracción* y *aplicación funcionales*. Los constructores de tipos son básicamente producto cartesiano, unión disjunta y flecha. Además tenemos el constructor μ de tipos recursivos. Completan al cálculo $\lambda^{\to,\mu}$ las reglas de reducción descritas en la sección 2.5 y las reglas de tipificación expuestas de manera informal en la sección 2.4.

En el siguiente capítulo formalizaremos las reglas de tipificación del cálculo $\lambda^{\to,\mu}$ y comenzaremos a estudiar sus propiedades.

CAPITULO 3

TERMINOS Y TIPOS

En el capítulo anterior presentamos un cálculo lambda tipificado que además posee tipos recursivos. En el presente capítulo formalizaremos las reglas que nos permiten elaborar a los términos bien tipificados de nuestro lenguaje. Además mostraremos algunas de las propiedades que lo distinguen de sistemas análogos. De particular interés resulta la construcción que realizaremos del operador de punto fijo.

3.1. Reglas de tipificación.

Una regla de tipificación es una regla de inferencia que permite deducir la tipificación de un término a partir de un conjunto de hipótesis de asignación de tipos.

Como en el capítulo anterior, un término tipificado $x{:}\sigma$ significa que x posee tipo σ. Un *contexto* es un conjunto finito de asignaciones de tipo, de la forma

$$\Gamma = \{x_1{:}\tau_1, x_2{:}\tau_2, \ldots, x_n{:}\tau_n\}$$

La expresión de la forma

$$\Gamma \rhd M{:}A$$

significa que en el contexto dado por Γ el término M debe poseer el tipo A.

Una *regla de inferencia de tipos* es una regla de la forma:

$$\frac{\Gamma_1 \triangleright M_1 : A_1 \ldots \Gamma_n \triangleright M_n : A_n}{\Gamma \triangleright N : B}$$

y significa que dadas las hipótesis de tipificación $\Gamma_1 \triangleright M_1 : A_1$ hasta $\Gamma_n \triangleright M_n : A_n$ se puede deducir la tipificación $\Gamma \triangleright N : B$.
Por ejemplo en la regla:

$$\frac{\Gamma \triangleright M : A \qquad \Gamma \triangleright N : B}{\Gamma \triangleright \langle M, N \rangle : A \times B}$$

se expresa que partiendo de que en el contexto Γ el término M tiene tipo A, y en el mismo contexto Γ el término N posee el tipo B entonces puede inferirse que en el contexto Γ el término <M,N> es del tipo A×B.

El λ-cálculo tipificado presentado en el capítulo anterior se formaliza en un conjunto de reglas de inferencia de tipos que presentamos a continuación.

Definición 3.1.- El λ-cálculo tipificado $\lambda^{\rightarrow,\times}$ está definido por las siguientes reglas:

$$\Gamma, X:A \triangleright X:A \qquad \text{(assumption)}$$

$$\frac{\Gamma, X:A \triangleright M:B}{\Gamma \triangleright (\lambda x:A.M):A \rightarrow B} \qquad \text{(abstraction)}$$

$$\frac{\Gamma \triangleright M:A \rightarrow B \quad \Gamma \triangleright N:A}{\Gamma \triangleright (M\ N):B} \qquad \text{(application)}$$

$$\frac{\Gamma \triangleright M:A \quad \Gamma \triangleright N:B}{\Gamma \triangleright \langle M,N \rangle : A \times B} \qquad \text{(pairing)}$$

$$\frac{\Gamma \triangleright M : A \times B}{\Gamma \triangleright \pi_1(M) : A}\ \text{(projection)} \qquad \frac{\Gamma \triangleright M : A \times B}{\Gamma \triangleright \pi_2(M) : B}\ \text{(projection)}$$

$$\frac{\Gamma \triangleright M : A}{\Gamma \triangleright \text{inl}(M) : A+B}\ \text{(injection)} \qquad \frac{\Gamma \triangleright M : B}{\Gamma \triangleright \text{inr}(M) : A+B}\ \text{(injection)}$$

$$\frac{\Gamma \triangleright P : A+B \qquad \Gamma, x:A \triangleright M : C \qquad \Gamma, y:B \triangleright N : C}{\Gamma \triangleright \text{case}(P, \lambda x:A.M,\ \lambda y:B.N) : C}\ \text{(by-cases)}$$

$$\frac{\Gamma \triangleright M : [\mu t.\alpha/t]\alpha}{\Gamma \triangleright (\text{fold } M) : \mu t.\alpha}\ \text{(fold)} \qquad \frac{\Gamma \triangleright M : \mu t.\alpha}{\Gamma \triangleright (\text{unfold } M) : [\mu t.\alpha/t]\alpha}\ \text{(unfold)}$$

3.2. Reglas de reducción.

Además también tenemos las siguientes reglas de reducción.

Definición 3.2.- Las reglas de reducción del sistema $\lambda^{\to,\mu}$ se listan a continuación:

$$(\lambda x:A.M)\ N \Rightarrow [N/x]\ M \qquad (\beta\text{-red})$$
$$\pi_1(\langle M,N\rangle) \Rightarrow M \qquad (\pi_1\text{-red})$$
$$\pi_2(\langle M,N\rangle) \Rightarrow N \qquad (\pi_2\text{-red})$$

$$\text{case}(\text{inl}(P),\ \lambda x:A.M,\ \lambda y:B.N) \Rightarrow (\lambda x.M)P \qquad (\text{inl-red})$$
$$\text{case}(\text{inr}(P),\ \lambda x:A.M,\ \lambda y:B.N) \Rightarrow (\lambda y.N)P \qquad (\text{inr-red})$$

$$(\text{fold } (\text{unfold } M)) \Rightarrow M \qquad (\text{fold-unfold})$$
$$(\text{unfold } (\text{fold } M)) \Rightarrow M \qquad (\text{unfold-fold})$$

Decimos que un término M se *reduce* a N (denotado como $M \Rightarrow N$) si el término N puede obtenerse a partir de la aplicación de una o más reglas de reducción al término M.

Una propiedad importante de las reglas de reducción

introducidas en la definición anterior es que preservan el tipo de los términos.

Teorema 3.1.- Si un término M del λ-cálculo $\lambda^{\rightarrow,\mu}$ se reduce a un término N, entonces M y N poseen el mismo tipo.

Demostración:

La demostración tiene que realizarse por inducción sobre el número de aplicaciones de las reglas de reducción dadas en la definición 3.2.

Para el caso de β-reducción tendríamos por ejemplo que si $(\lambda x{:}A.M){:}A\rightarrow B$ y $N{:}A$ entonces por (application) se cumple que:

$$((\lambda x{:}A.M)N){:}B$$

y por otro lado $[N/x]M{:}B$.

De manera similar pueden demostrarse que las demás reglas de reducción preservan el tipo de los términos.

3.3. Normalización y tipos recursivos.

El sistema $\lambda^{\rightarrow,\mu}$ difiere de otros λ-cálculos tipificados en la introducción de tipos recursivos de la forma $\mu t.\alpha$ así como de los operadores de coerción *fold* y *unfold*.

Sin embargo la introducción de tipos recursivos modifica notablemente las propiedades que se tienen en el λ-cálculo tipificado $\lambda^{\rightarrow,+,\times}$ (véase Mitchell, 1989) particularmente una característica importante que es la *normalización fuerte*.

En un λ-cálculo tipificado carente de tipos recursivos se tiene que todo término es reducible a una forma normal y esta propiedad se denomina normalización fuerte.

Nuestro sistema $\lambda^{\rightarrow,\mu}$ carece de esta propiedad. De hecho posee

una característica opuesta que estableceremos en el siguiente teorema.

Teorema 3.2.- Dado cualquier tipo α en el λ-cálculo tipificado $\lambda^{\rightarrow,\mu}$ existe un término de tipo α que es irreducible.

Demostración:

Procederemos con la construcción y tipificación del término irreducible del tipo α.

De la regla (assumption):

$$\Gamma, x:\mu t.t\rightarrow\alpha \triangleright x:\mu t.t\rightarrow\alpha \qquad (3.1)$$

y de (unfold) se obtiene:

$$\Gamma, x:\mu t.t\rightarrow\alpha \triangleright (\text{unfold } x):(\mu t.t\rightarrow\alpha)\rightarrow\alpha \qquad (3.2)$$

De (application) utilizando (3.1) y (3.2) como hipótesis:

$$\Gamma, x:\mu t.t\rightarrow\alpha \triangleright ((\text{unfold } x)x):\alpha \qquad (3.3)$$

De (3.3) utilizando (abstraction) se tiene:

$$\Gamma \triangleright (\lambda x:\mu t.t\rightarrow\alpha.((\text{unfold } x)x)):(\mu t.t\rightarrow\alpha)\rightarrow\alpha \qquad (3.4)$$

A partir de (3.4), mediante la regla (fold) obtenemos:

$$\Gamma \triangleright (\text{fold } (\lambda x.((\text{unfold } x)x))):\mu t.t\rightarrow\alpha \qquad (3.5)$$

En donde, por brevedad, hemos omitido el tipo del parámetro x.

Finalmente, de (3.4), (3.5) y (application) se tiene:

$$\Gamma \triangleright (\lambda x.((\text{unfold } x)x))(\text{fold}(\lambda x.((\text{unfold } x)x))):\alpha \qquad (3.6)$$

Por lo tanto, si definimos:

$$\Omega_\alpha \equiv (\lambda x.((\text{unfold } x)x))(\text{fold}(\lambda x.((\text{unfold } x)x)))$$

entonces Ω_α es de tipo α suponiendo que x es de tipo $\mu t.t \to \alpha$. Además Ω_α es el término irreducible que buscamos.

Obsérvese que si $\omega_\alpha \equiv (\lambda x.((\text{unfold } x)x))$ entonces $\Omega_\alpha = (\omega_\alpha(\text{fold } \omega_\alpha))$ y puede comprobarse que:

$$(\omega_\alpha(\text{fold } \omega_\alpha)) = (\lambda x.((\text{unfold } x)x))(\text{fold } \omega_\alpha)$$

es irreducible, puesto que:

$(\lambda x.((\text{unfold } x)x))(\text{fold } \omega_\alpha) \Rightarrow^{\beta} ((\text{unfold } (\text{fold } \omega_\alpha))(\text{fold } \omega_\alpha)) \Rightarrow^{\text{unfold-fold}} (\omega_\alpha(\text{fold } \omega_\alpha)) \Rightarrow^{\beta} \ldots$

Como este término Ω_α es independiente del tipo α que escojamos entonces para cualquier tipo existe este término no normalizable.

3.4. Operadores de punto fijo.

De manera análoga a como construimos un término irreducible para cada tipo α de nuestro sistema $\lambda^{\to,\mu}$ podemos llegar a construir un operador de punto fijo, muy parecido al que se tiene en λ-cálculo puro, para cada uno de los tipos.

A diferencia de los λ-cálculos fuertemente normalizables, en donde los operadores de punto fijo se postulan (Hudak, 1989, p.376), en el cálculo $\lambda^{\to,\mu}$ podemos construirlos explícitamente.

Teorema 3.3.- Dado cualquier tipo α, existe un *operador de punto fijo* Y_α tal que para todo término $f:\alpha \to \alpha$ se cumple:

$$(Y_\alpha\ f) \Rightarrow (f\ (Y_\alpha\ f)).$$

Demostración:
Se puede verificar que el operador definido como:

$$Y_\alpha \equiv \lambda F.((\lambda x.(F\ ((\text{unfold}\ x)x)))\ (\text{fold}\ \lambda x.(F\ ((\text{unfold}\ x)x)))) : (\alpha\to\alpha)\to\alpha$$

satisface la propiedad de ser operador de punto fijo. En efecto, $(Y_\alpha\ f)$ donde $f:\alpha\to\alpha$, es el término:

$$(\lambda x.(f((\text{unfold}\ x)x)))(\text{fold}\ \lambda x.(f((\text{unfold}\ x)x)))$$

que por β-reducción se reduce a:

$$(f((\text{unfold}\ (\text{fold}\ \lambda x.(f((\text{unfold}\ x)x))))\ (\text{fold}\ \lambda x.(f((\text{unfold}\ x)x)))))$$

Este término se reduce por (unfold-fold) a:

$$(f((\lambda x.(f((\text{unfold}\ x)x)))(\text{fold}\ (\lambda x.(f((\text{unfold}\ x)x))))))$$

el cual es precisamente:

$$(f\ (Y_\alpha\ f))$$

Sólo nos resta demostrar que el operador Y_α está bien tipificado y que de hecho su tipo es $(\alpha\to\alpha)\to\alpha$.

De (assump) tenemos:

$$\Gamma, x:\mu t.t\to\alpha,\ F:\alpha\to\alpha \triangleright x:\mu t.t\to\alpha \qquad (3.7)$$

donde Γ es cualquier contexto.

Aplicando (unfold) a (3.7) se tiene:

$$\Gamma, x:\mu t.t\to\alpha,\ F:\alpha\to\alpha \triangleright (\text{unfold}\ x):(\mu t.t\to\alpha)\to\alpha \qquad (3.8)$$

Y por (applic) (3.7) y (3.8):

$$\Gamma, x:\mu t.t\to\alpha,\ F:\alpha\to\alpha \triangleright ((\text{unfold}\ x)x):\alpha \qquad (3.9)$$

Y de (assump) y (applic) a (3.9):

$$\Gamma, x:\mu t.t\to\alpha,\ F:\alpha\to\alpha \triangleright (F\ ((\text{unfold}\ x)x)):\alpha \qquad (3.10)$$

Por (abstrac) se llega a:

$$\Gamma, F:\alpha\to\alpha \triangleright \lambda x:\mu t.t\to\alpha.(F((\text{unfold}\ x)x)):(\mu t.t\to\alpha)\to\alpha \qquad (3.11)$$

De donde por (fold) se tiene:

$$\Gamma, F:\alpha\to\alpha \triangleright (\text{fold}\ \lambda x.(F((\text{unfold}\ x)x))):\mu t.t\to\alpha \qquad (3.12)$$

en donde hemos omitido por brevedad el tipo del parámetro x.

De (3.11) y (3.12) haciendo aplicación obtenemos:

$$\Gamma, F:\alpha\to\alpha \triangleright (\lambda x.(F((\text{unfold}\ x)x)))\ (\text{fold}\ \lambda x.(F((\text{unfold}\ x)x))):\alpha \qquad (3.13)$$

Y haciendo abstracción en (3.13):

$$\Gamma \triangleright \lambda F.((\lambda x.(F((\text{unfold}\ x)x)))\ (\text{fold}\ \lambda x.(F((\text{unfold}\ x)x)))):(\alpha\to\alpha)\to\alpha \qquad (3.14)$$

Entonces hemos demostrado que Y_α es del tipo $(\alpha\to\alpha)\to\alpha$. Como la anterior construcción es independiente de α, entonces hemos encontrado un operador de punto fijo para cada uno de los posibles tipos del sistema $\lambda^{\to,\mu}$.

3.5. Operaciones recursivas.

El operador de punto fijo es especialmente importante en la definición de operaciones recursivas. Dado que en nuestro sistema contamos con tipos de datos recursivos requerimos de operaciones recursivas para la manipulación de términos pertenecientes a esos tipos.

Por ejemplo, el tipo dado por:

$$\mu t.(\text{Unit} + \text{Int}\times t)$$

representa al tipo de las listas de enteros. En esta definición Unit es el tipo formado por un solo elemento, el elemento nulo, representado por <>.

Podemos representar a un término de este tipo por ejemplo, por la expresión :

<2,3,4>

aunque esta expresión es sintácticamente incorrecta. Sin embargo la utilizaremos para abreviar el término equivalente:

fold(inr <2,fold(inr <3,fold(inr <4,fold(inl <>)>)>)>)

que es sintácticamente correcto y está bien tipificado.

Es fácilmente demostrable que el término anterior es del tipo $\mu t.(Unit + Int \times t)$.

Para ilustrar como se puede utilizar el operador de punto fijo para definir operaciones recursivas sobre tipos de datos recursivos, construyamos un operador sobre las listas que arriba mencionamos.

Considere el operador *mapcar* (similar al que posee el mismo nombre en LISP) que sirve para "iterar" la aplicación de una función sobre los elementos de una lista.

Por ejemplo la función $\lambda x.x+1$ incrementa en 1 a su argumento y mientras que (mapcar $\lambda x.x+1$) incrementaría en uno a cada uno de los elementos de una lista. Si *l*=<1,2,3> entonces:
((mapcar $\lambda x.x+1$) *l*) se reduce a la lista <2,3,4>.

Procedamos entonces a la construcción del operador *mapcar*:

Primero observemos que si $L:\mu t.(Unit + Int \times t)$ entonces $(unfold\ L):Unit + Int \times(\mu t.Unit + Int \times t)$.

Por lo tanto (unfold L) es ó bien la lista nula <>:Unit o tiene al menos un elemento al frente (es decir, pertenece al tipo $Int \times(\mu t.Unit + Int \times t)$).

En el primer, caso el operador:

$$\lambda x.\mathrm{inl}(x):\mathrm{Unit} \rightarrow (\mathrm{Unit} + \mathrm{Int} \times (\mu t.\mathrm{Unit} + \mathrm{Int} \times t))$$

nos devuelve la lista nula de correspondiente tipo.

Mientras que si no es la lista nula, aplicamos f al primer elemento de la lista y concatenamos con el resultado de aplicar recursivamente *mapcar* al resto de la lista:

$$\lambda y.\mathrm{inr}(<f(\pi_1(y)),((\mathrm{mapcar}\ f)(\pi_2(y)))>) \\ :\mathrm{Int} \times (\mu t.\mathrm{Unit}+\mathrm{Int}\times t) \rightarrow (\mathrm{Unit}+\mathrm{Int}\times(\mu t.\mathrm{Unit}+\mathrm{Int}\times t)) \quad (3.15)$$

Considerando entonces ambos casos:

$$\mathrm{case}((\mathrm{unfold}\ L),\lambda x.\mathrm{inl}(x), \\ \lambda y.\mathrm{inr}(<(f\ \pi_1(y)),((\mathrm{mapcar}\ f)(\pi_2(y)))>))$$

como esta expresión es del tipo:

$$\mathrm{Unit} + \mathrm{Int} \times (\mu t.\mathrm{Unit} + \mathrm{Int} \times t)$$

aplicamos *fold* para que pertenezca al tipo normalizado

$$\mathrm{fold}(\mathrm{case}((\mathrm{unfold}\ L),\lambda x.\mathrm{inl}(x),\lambda y.\mathrm{inr}(<f(\pi_1(y)), \\ ((\mathrm{mapcar}\ f)(\pi_2(y)))>)))$$

Haciendo abstracción funcional sobre L y f obtenemos la definición recursiva de *mapcar*:

$$\mathrm{mapcar} \equiv \lambda f.\lambda L.\mathrm{fold}(\mathrm{case}((\mathrm{unfold}\ L),\lambda x.\mathrm{inl}(x), \\ \lambda y.\mathrm{inr}(<f(\pi_1(y)),((\mathrm{mapcar}\ f)(\pi_2(y)))>)))$$

Es decir, mapcar es el punto fijo del funcional:

$$\lambda F.\lambda f.\lambda L.\mathrm{fold}(\mathrm{case}((\mathrm{unfold}\ L),\lambda x.\mathrm{inl}(x), \\ \lambda y.\mathrm{inr}(<f(\pi_1(y)),((F\ f)(\pi_2(y)))>)))$$

por lo tanto, mapcar es:

$$Y_\tau(\lambda F.\lambda f.\lambda L.\text{fold}(\text{case}((\text{unfold } L),\lambda x.\text{inl}(x),$$
$$\lambda y.\text{inr}(<f(\pi_1(y)),((F\ f)(\pi_2(y)))>))))) \qquad (3.16)$$

donde

$$\tau \equiv (\text{Int}\to\text{Int})\to(\mu t.(\text{Unit}+\text{Int}\times t)\to\mu t.(\text{Unit}+\text{Int}\times t)$$

De hecho, este resultado puede generalizarse para cualquier par de tipos A y B de tal manera que si:

$$\tau \equiv (A\to B)\to(\mu t.(\text{Unit}+A\times t)\to\mu t.(\text{Unit}+B\times t)$$

el operador definido en (3.16) es el operador *mapcar* correspondiente.

Demostraremos a continuación que si f:A→B es una función que toma elementos de A y los mapea a B y L:μt.(Unit + A×t) es una lista de elementos de tipo A, entonces el operador dado en (3.16) aplicado a f nos devuelve una función que convierte listas de elementos de A a listas de elementos de B.

Sea Γ_1 el contexto de tipificación dado por

$$\Gamma_1 = \Gamma \cup \{L:\mu t.\text{Unit} + A\times t, f:A\to B,$$
$$F:(A\to B)\to(\mu t.\text{Unit}\ A\times t \to \mu t.\text{Unit} + B\times t)\}$$

Entonces por (assump):

$$\Gamma_1 \rhd L:\mu t.\text{Unit} + A\times t \qquad (3.17)$$

Y aplicando (unfold):

$$\Gamma_1 \rhd (\text{unfold } L):\text{Unit} + A\times(\mu t.\text{Unit} + A\times t) \qquad (3.18)$$

Ahora sea:

$$\Gamma_2 = \Gamma_1 \cup \{x:A\times(\mu t.\text{Unit} + A\times t)\}$$

entonces por (proj):

$$\Gamma_2 \triangleright \pi_1(x):A \tag{3.19}$$

y

$$\Gamma_2 \triangleright \pi_2(x):\mu t.\mathrm{Unit} + A\times t \tag{3.20}$$

de donde por (applic) a (3.19):

$$\Gamma_2 \triangleright (f(\pi_1(x))):B \tag{3.21}$$

Por otra parte, de (applic):

$$\Gamma_2 \triangleright (F\ f):(\mu t.\mathrm{Unit} + A\times t)\rightarrow(\mu t.\mathrm{Unit} + B\times t) \tag{3.22}$$

y utilizando (applic) en (3.20) y (3.22)

$$\Gamma_2 \triangleright ((F\ f)\ \pi_2(x)):\mu t.\mathrm{Unit} + B\times t \tag{3.23}$$

Haciendo (pairing) a (3.21) y (3.23):

$$\Gamma_2 \triangleright \langle f(\pi_1(x)),((F\ f)\pi_2(x))\rangle:B \times \mu t.(\mathrm{Unit} + B\times t) \tag{3.24}$$

Y por (injection) en (3.24) se tiene:

$$\Gamma_2 \triangleright \mathrm{inr}\ \langle f(\pi_1(x)),((F\ f)\pi_2(x))\rangle :\mathrm{Unit} + B \times \mu t.(\mathrm{Unit} + B\times t) \tag{3.25}$$

Por otra parte de (assump):

$$\Gamma_1, x:\mathrm{Unit} \triangleright x:\mathrm{Unit} \tag{3.26}$$

y aplicando (injection)

$$\Gamma_1, x:\mathrm{Unit} \triangleright \mathrm{inl}(x):\mathrm{Unit} + B \times \mu t.(\mathrm{Unit} + B\times t) \tag{3.27}$$

De (3.18),(3.25) y (3.27) se tiene:

$\Gamma_1 \triangleright$ case ((unfold L),λx:Unit.inl(x),
λy.inr<(f(π_1(y))),((F f)(π_2(y)))>):
Unit + B × μt.(Unit + B×t)

Aplicando (fold):

$\Gamma_1 \triangleright$ fold(case ((unfold L),λx.inl(x),
λy.inr<(f(π_1(y))),((F f)(π_2(y)))>):
μt.(Unit + B×t)

Descargando las hipótesis de tipificación en Γ_1 se tiene:

$\Gamma \triangleright \lambda$f.$\lambda$L.fold(case((unfold L),$\lambda$x.inl(x),
λy.inr<(f(π_1(y))),((F f)(π_2(y)))>)
:(A→B)→(μt.(Unit + A×t)→μt.(Unit + B×t))

Veamos que el operador *mapcar* dado por (3.16) es precisamente el que, al aplicarse sobre una lista, devuelve otra lista cuyos elementos son a su vez el mapeo -mediante la función f- de los correspondientes elementos de la primera lista:

Sea *l* la lista dada por <1,2,3> entonces el término:
((mapcar λx.x+1) *l*)
se reduce a:

((λf.λL.fold(case((unfold L),λx.inl(x),
λy.inr <(f(π_1(y))),((mapcar f)(π_2(y)))>))
(λz.z+1))(<1,2,3>)

que por β-reducción se reduce a:

(λL.fold(case((unfold L),λx.inl(x),
λy.inr <(λz.z+1)(π_1(y)),(mapcar(λz.z+1)(π_2(y)))>))
(<1,2,3>)

y tras aplicar nuevamente β-reducción se tiene:

```
fold(case(<1,2,3>,λx.inl(x),
  λy.inr <(λz.z+1)(π1(y)),(mapcar (λz.z+1)(π2(y)))>))
```

reduciendo el *case* obtenemos:

```
fold(inr <(λz.z+1)(π1 <1,2,3>),
     mapcar(λz.z+1)(π2 <1,2,3>)>)
```

reduciendo las proyecciones se llega al término

```
fold(inr <(λz.z+1) 1, (mapcar(λz.z+1))<2,3>>)
```

o sea :

```
⇒ fold(inr <2, (mapcar(λz.z+1))<2,3>)>)
```

y haciendo sucesivamente desarrollos recursivos:

```
⇒ fold(inr <2, fold(inr <3, (mapcar(λz.z+1)<3>)>)>)
```

hasta reducirse a:

```
fold(inr <2,fold(inr <3,fold(inr <4,fold(inl <>)>)>)>)
```

que es equivalente a la lista:

<2,3,4>:μt.Unit + Int × t.

Con este ejemplo no solamente pretendemos ilustrar el uso del operador de punto fijo, sino que además hemos mostrado como "programar" en nuestro lenguaje abstracto mediante el uso de funciones recursivas. Podemos entonces hacer hincapié en lo adecuado que resulta nuestro lenguaje como base teórica de un lenguaje de programación útil en la práctica.

3.6. Resumen.

Hemos presentado las reglas de inferencia de términos tipificados así como las reglas de reducción de términos. Se ha demostrado no solamente que nuestro cálculo no es normalizable sino que además para cada tipo puede construirse un término que no tiene forma normal.

Además hemos presentado una construcción del operador de punto fijo para cualquier tipo. Este operador será especialmente importante para la definición constructiva de una relación de subtipos en los tipos recursivos. Por otro lado mostramos con un ejemplo como se utiliza al operador de punto fijo para transformar a los tipos de datos recursivos.

CAPITULO 4

SUBTIPOS

En este capítulo construiremos una relación de orden parcial entre los tipos que hemos definido en capítulos anteriores, de modo tal que este ordenamiento parcial en los tipos capture la noción usual de subtipos en programación.

En el trabajo de Cardelli-Amadio (1990) la relación de subtipos es desarrollada siguiendo cuatro enfoques diferentes: De acuerdo a un ordenamiento en árboles infinitos, a un conjunto de reglas de tipificación, a un algoritmo y a un conjunto de modelos (relaciones de equivalencia parcial). El algoritmo presentado es básicamente un algoritmo de resolución sobre un sistema de ecuaciones regulares. En el artículo de Cardelli-Amadio se demuestra la consistencia y completitud entre los ordenamientos inducidos por los árboles, las reglas y el algoritmo.

En el presente capítulo estudiaremos un enfoque constructivo de la relación de subtipos. Para definir la noción de subtipos construiremos una colección de *operadores de coerción* que nos permitan convertir explícitamente los tipos de los términos. Demostraremos la consistencia de este enfoque constructivo con el sistema de Cardelli-Amadio.

4.1. Reglas de subtipos.

El conjunto de reglas de subtipificación que detallaremos a continuación siguen una sintaxis similar a las reglas de inferencia de términos tipificados expuestas en la sección (3.1).

En estas reglas de subtipos un *contexto* será un conjunto finito de hipótesis acerca de subtipos, de la forma:

$$\Gamma = \{t_1 < s_1, t_2 < s_2, \ldots, t_n < s_n\}$$

mientras que la notación:

$$\Gamma \vdash \alpha < \beta$$

significa que en el contexto de subtipos Γ se cumple que α es subtipo de β.

Las reglas de inferencia son de la forma:

$$\frac{\Gamma_1 \vdash \alpha_1 < \beta_1 \ \ldots \ \Gamma_n \vdash \alpha_n < \beta_n}{\Gamma_{n+1} \vdash \alpha_{n+1} < \beta_{n+1}}$$

en donde la validez de $\Gamma_1 \vdash \alpha_1 < \beta_1$ hasta $\Gamma_n \vdash \alpha_n < \beta_n$ implican la validez de $\Gamma_{n+1} \vdash \alpha_{n+1} < \beta_{n+1}$.

Definición 4.1.- El sistema de inferencia de subtipos en el cálculo $\lambda^{\rightarrow,\mu}$ esta dado por los *axiomas*:

$$\Gamma \vdash \alpha < \alpha \qquad (eq_R)$$

$$\Gamma \vdash \bot < \alpha \qquad (\bot_R)$$

$$\Gamma \vdash \alpha < \top \qquad (\top_R)$$

$$\Gamma \vdash t < s \quad \text{si } t < s \in \Gamma \qquad (assmp_R)$$

y por las siguientes *reglas de inferencia*:

$$\frac{\Gamma \vdash \alpha < \alpha' \qquad \Gamma \vdash \beta < \beta'}{\Gamma \vdash \alpha \times \beta < \alpha' \times \beta'} \qquad (\times_R)$$

$$\frac{\Gamma \vdash \alpha < \beta \qquad \Gamma \vdash \beta < \gamma}{\Gamma \vdash \alpha < \gamma} \qquad (trans_R)$$

$$\frac{\Gamma \vdash \alpha < \alpha' \qquad \Gamma \vdash \beta < \beta'}{\Gamma \vdash \alpha+\beta < \alpha'+\beta'} \quad (+_R)$$

$$\frac{\Gamma \vdash \alpha' < \alpha \qquad \Gamma \vdash \beta < \beta'}{\Gamma \vdash \alpha\rightarrow\beta < \alpha'\rightarrow\beta'} \quad (\rightarrow_R)$$

$$\frac{\Gamma \cup \{t<s\} \quad \vdash \quad \alpha < \beta}{\Gamma \vdash \mu t.\alpha < \mu s.\beta} \quad (\mu_R)$$

donde, en esta última regla, t y s no están en Γ, t es variable libre en α pero no en β y s es variable libre en β pero no en α. □

Puede observarse que este conjunto de axiomas y reglas definen adecuadamente la noción de subtipos que habíamos presentado en el capítulo 2.

Las propiedades de preorden - reflexividad y transitividad - quedan establecidas en (eq_R) y ($trans_R$). Por otro lado ($\bot_R$) y (T_R) nos definen los tipos $\bot$ (bottom) subtipo de cualquier otro tipo y T (top) del cual es subtipo cualquier otro tipo.

Como ejemplo del uso de estas reglas de inferencia consideremos la deducción de la relación:

$$\mu t.\text{Unit} + \text{Nat} \times t < \mu s.\text{Unit} + \text{Int} \times s$$

es decir, demostraremos que el tipo de las listas de números naturales es subtipo del tipo de las listas de números enteros.

Partiremos de un contexto Γ tal que: Nat < Int $\in \Gamma$.

Entonces, por ($assmp_R$) tenemos:

$$\Gamma \cup \{t < s\} \vdash \text{Nat} < \text{Int} \qquad (4.1)$$

y de (assmp_R) se tiene también:

$$\Gamma \cup \{t < s\} \vdash t < s \qquad (4.2)$$

Por lo tanto, de (4.1) y (4.2) utilizando $(\times_R)$ se tiene:

$$\Gamma \cup \{t < s\} \vdash \text{Nat} \times t < \text{Int} \times s \qquad (4.3)$$

Por otro lado, de (eq_R):

$$\Gamma \cup \{t < s\} \vdash \text{Unit} < \text{Unit} \qquad (4.4)$$

Y aplicando $(+_R)$ a (4.3) y (4.4):

$$\Gamma \cup \{t < s\} \vdash \text{Unit} + \text{Nat} \times t < \text{Unit} + \text{Int} \times s \qquad (4.5)$$

Por ultimo, de la regla (μ_R) en (4.5) se obtiene:

$$\Gamma \vdash \mu t.\text{Unit} + \text{Nat} \times t < \mu s.\text{Unit} + \text{Int} \times s$$

Dado un conjunto de tipos básicos con relaciones de subtipos entre ellos, las reglas de inferencia de subtipos nos permiten definir un sistema de subtipos basados en la *estructura* de los tipos, de manera independiente a cualquier noción semántica.

4.2. Coerciones explícitas.

En el planteamiento constructivo del sistema de subtipos, en lugar de proporcionar un conjunto de relaciones de subtipos entre algunos tipos básicos presentamos un conjunto de términos constantes $C_{t,s}: t \to s$ de *coerciones explícitas* para diferentes tipos básicos t y s, y en lugar de tener reglas de inferencia para subtipos, tenemos una construcción de una familia de operadores de coerción de modo tal que $\alpha < \beta$ si y solo si existe un operador $C_{\alpha,\beta}$ tal que para todo $M:\alpha$ se tiene que $(C_{\alpha,\beta}\ M):\beta$, y $C_{\alpha,\beta}$ pertenece a la colección de operadores de coerción.

Definición 4.2.- Dado un conjunto finito de términos constantes:

$$\{C_{\alpha,\beta}:\alpha\rightarrow\beta \mid \alpha,\beta \in \mathbf{Tp}\}$$

definimos al conjunto de operadores de coerción *Oc* de la siguiente manera:

(i) Todos los términos $C_{\alpha,\beta} \in Oc$.

(ii) Si $M:\alpha\rightarrow\beta$ y $N:\beta\rightarrow\gamma \in Oc$ entonces:
$\lambda x:\alpha.(N(M\ x)) \in Oc$.

(iii) Si $M:\alpha'\rightarrow\alpha$ y $N:\beta\rightarrow\beta' \in Oc$ entonces:
$\lambda f:\alpha\rightarrow\beta.\lambda x:\alpha'.(N(f(M\ x)))$ pertenece a *Oc*.

(iv) Si $M:\alpha\rightarrow\alpha'$ y $N:\beta\rightarrow\beta'$ están en *Oc* entonces:
$\lambda x:(\alpha\times\beta).<(M\ \pi_1(x)),(N\ \pi_2(x))>$ pertenece a *Oc*.

(v) Si $M:\alpha\rightarrow\alpha'$ y $N:\beta\rightarrow\beta'$ están en *Oc* entonces:
$\lambda x:(\alpha+\beta).\mathrm{case}(x,\lambda y.\mathrm{inl}(M\ y),\lambda z.\mathrm{inr}(N\ z))$
pertenece a *Oc*.

(vi) Si hemos construido un término $M:\alpha\rightarrow\beta$ en donde $x:t\rightarrow s$ es una variable libre en M, t es variable libre en α y s en β, podemos construir un término $M':[\mu t.\alpha/t]\alpha \rightarrow [\mu s.\beta/s]\beta$ que resulta de reemplazar todas las ocurrencias de $x:t\rightarrow s$ por $x:\mu t.\alpha \rightarrow \mu t.\beta$, entonces:

$$Y(\lambda x:\mu t.\alpha\rightarrow\mu s.\beta.\lambda y:\mu t.\alpha.(\mathrm{fold}(M'\ (\mathrm{unfold}\ y))))$$

también pertenece a *Oc*.

(vii) Para cualquier tipo α, el operador de identidad $\lambda x:\alpha.x$ también está en *Oc*. □

Como puede observarse existe una correspondencia entre las reglas de inferencia dadas en la definición 4.1 y las construcciones de la definición 4.2, de modo tal que cada una de las construcciones de operadores de coerción sería la demostración por construcción de una deducción de subtipos.

Por ejemplo, la construcción (i) es contraparte de ($assump_R$), (ii) es la construcción asociada a ($trans_R$), (iii) está asociada a ($\neg_R$), (iv) a ($\times_R$), (v) a ($+_R$) y por último (vi) está relacionada con (μ_R).

Solamente los axiomas ($\bot_R$) y (T_R) no tienen un equivalente constructivo. Para ($\bot_R$) puede ponerse al operador $\lambda x{:}\bot.\Omega_\alpha{:}\bot\to\alpha$ que asocia al término irreducible en $\bot$ el correspondiente término irreducible Ω_α de tipo α que construimos en el capítulo anterior. También podríamos introducir a la familia de operadores $C_{\alpha,T}{:}\alpha\to T$ (para todo tipo α) como parte de *Oc* para tener un operador asociado al axioma (T_R).

Si tomamos en cuenta estas dos colecciones de operadores dentro de *Oc* entonces se cumple la propiedad que se establece en el siguiente teorema, en el cual se tiene que dos tipos están en relación de subtipos si y solo si existe un operador entre estos dos tipos que pertenezca a la familia Oc, esto es :

Teorema 4.1.- Dado un conjunto finito de coerciones explícitas $C_{t,s}{:}t\to s$, donde t y s son tipos básicos, entonces tenemos un contexto de subtipos dado por:

$$\Gamma = \{t<s \mid \exists\ C_{t,s}{:}t\to s \text{ coerción explícita } \}$$

Y se cumple que:

$$\Gamma \vdash \alpha < \beta$$

si y solo si existe un operador $F{:}\alpha\to\beta$ que pertenezca al conjunto de operadores de coerción que se define en 4.2. (*Observación*: Debido a que los operadores de identidad $\lambda x{:}\alpha.x$ forman parte de *Oc* este operador de coerción no está determinado de manera única.) □

Demostración: La demostración de este teorema es inmediata a partir del hecho señalado anteriormente de que cada regla de inferencia dada en la definición 4.1 tiene su correspondiente

construcción de operadores en la definición 4.2, de tal manera que dada una demostración de una relación de subtipos uno puede construir el operador de coerción realizando la construcción correspondiente en cada uno de los pasos de la demostración. De manera análoga uno puede obtener a partir del "parsing" del operador de coerción la información necesaria para formar una demostración de la relación de subtipos.

4.3. Coerción en tipos recursivos.

Analizemos en un ejemplo en particular como se construiría el término de coerción para tipos recursivos.

Regresemos a la relación:

$$\mu t.\mathrm{Unit} + \mathrm{Nat} \times t < \mu s.\mathrm{Unit} + \mathrm{Int} \times s$$

recordando que nos indica que las listas de números naturales son subtipo de las listas de enteros.

Partimos de la suposición de que Nat < Int y que por lo tanto tenemos un término C_{NI}:Nat→Int.

Partiendo de que tenemos un término F:t→s entonces construyamos de acuerdo a (4) de la definición 4.2, el término:

$$\lambda y{:}(\mathrm{Nat} \times t).\langle (C_{NI}\ \pi_1(y)),\ (F\ \pi_2(y))\rangle : \mathrm{Int} \times s$$

Este término estaría en *Oc* al igual que el operador identidad λx:Unit.x y de acuerdo a (v) de la definición 4.2, también pertenecería a *Oc* el operador

$$\lambda x.(\mathrm{Unit}+\mathrm{Nat} \times t).\mathrm{case}(x,\lambda y.\mathrm{inl}(y),\ \lambda z.\mathrm{inr}(\langle (C_{NI}\ \pi_1(y)),(F\ \pi_2(y))\rangle))$$

Obsérvese que hemos hecho β-reducción al interior de los argumentos de inl e inr para disminuir la complejidad del término, aunque en rigor en este deberían aparecer los operadores

que hemos estado definiendo inductivamente.

Por lo tanto ahora tenemos construido el operador de coerción de tipo Unit + Nat × t → Unit + Int × s.

De acuerdo con la construcción (vi) de la definición 4.2 se sustituirán las instancias de t por $\mu t.\alpha$ y las de s por $\mu s.\beta$ llevándonos al término:

$$\lambda x{:}\text{Unit} + \text{Nat} \times \mu t.\alpha.\text{case}(x,\lambda z{:}\text{Unit}.\text{inl}(z),$$
$$\lambda y{:}\text{Nat} \times \mu t.\alpha.\text{inr}\langle(C_{\text{Nat}}\ \pi_1(y)),(F\ \pi_2(y))\rangle)$$
$$:(\text{Unit} + \text{Nat} \times \mu t.\alpha) \rightarrow (\text{Unit} + \text{Int} \times \mu s.\beta)$$

Si definimos a M' como el término anterior, entonces de acuerdo a la regla (vi) de la def. 4.2, el operador:

$$Y(\lambda F.\lambda m.(\text{fold}(M'(\text{unfold m})))) : \mu t.\alpha \rightarrow \mu t.\beta$$

es precisamente el operador de coerción que necesitamos para probar que $\mu t.\alpha < \mu t.\beta$.

Del ejemplo anterior podemos deducir que el operador de coerción en tipos recursivos es un operador recursivo que aplica coerción a cada uno de los tipos que resultan del desdoblamiento del tipo recursivo. Este enfoque hace que la subtipificación de tipos recursivos sea más lógica y evidente que la relación definida en la teoría de Cardelli, aunque como sabemos del teorema 4.1 ambos tratamientos sean por completo equivalentes.

4.4. Invariancia de estructura.

Las construcciones definidas en la definición 4.2 no son arbitrarias pues además de ser consistentes con el sistema de reglas de inferencia proporcionado en la definición 4.1, están formuladas de tal manera que cualquier operador de coerción deje invariante la estructura sintáctica de los términos a los cuales se aplican. Es decir, la coerción cambiará la tipificación de los términos pero no la estructura de estos.

Cuando afirmamos que los operadores de coerción dejan invariante la estructura de los términos queremos decir que estos operadores mapean productos cartesianos a productos cartesianos, sumas disjuntas a sumas disjuntas, funciones a funciones y, lo más importante, tipos recursivos a tipos recursivos con similaridad en el desdoblamiento.

Probaremos la anterior afirmación de una manera gradual demostrando esta propiedad para cada uno de los constructores de tipos.

Lema 4.1.- Si $\alpha < \alpha'$ y $\beta < \beta'$ con operadores de coerción M y N respectivamente siendo $O:(\alpha \times \beta) \rightarrow (\alpha' \times \beta')$ el operador de coerción de la relación $\alpha \times \beta < \alpha' \times \beta'$, entonces para todo término $\langle a,b \rangle : \alpha \times \beta$ se cumple:

$$(O\ \langle a,b \rangle) \Rightarrow \langle (M,a),(N,b) \rangle$$

Demostración:

Tenemos que el operador para la relación $\alpha \times \beta < \alpha' \times \beta'$ esta dado por el término

$$O \equiv \lambda x:(\alpha \times \beta).\langle (M\ \pi_1(x)),(N\ \pi_2(x)) \rangle$$

Por lo tanto:

$$\begin{aligned}(O\ \langle a,b \rangle) &= ((\lambda x:\alpha \times \beta.\langle (M\ \pi_1(x)),(N\ \pi_2(x)) \rangle)\ \langle a,b \rangle) \\ &\Rightarrow^{\beta} \langle (M\ \pi_1(\langle a,b \rangle)),(N\ \pi_2(\langle a,b \rangle)) \rangle \\ &\Rightarrow^{\pi 1} \langle (M\ a),(N\ \pi_2(\langle a,b \rangle)) \rangle \Rightarrow^{\pi 2} \langle (M\ a),(N\ b) \rangle\end{aligned}$$

Lema 4.2.- Si $\alpha < \alpha'$ y $\beta < \beta'$ con operadores de coerción M y N respectivamente, con $O:(\alpha + \beta) \rightarrow (\alpha' + \beta')$ el operador de coerción de la relación $\alpha + \beta < \alpha' + \beta'$, entonces para todo $a:\alpha$ y $b:\beta$ se cumple:

$$\begin{aligned}O(inl(a)) &\Rightarrow inl(M(a)) \\ O(inr(b)) &\Rightarrow inr(N(b))\end{aligned}$$

Demostración: Para la relación $\alpha + \beta < \alpha' + \beta'$ se define:

$$O \equiv (\lambda x:(\alpha + \beta).case(x,\lambda y.inl(M\ y),\lambda z.inr(N\ z)))$$

Primero, reduciendo:

$$
\begin{aligned}
&(O\ (inl(a))) = \\
&\quad (\lambda x{:}\alpha + \beta.case(x,\lambda y.inl(M\ y),\lambda z.inr(N\ z)))\ \ (inl(a)) \\
&\quad \Rightarrow^{\beta}\ case(inl(a),\lambda y.inl(M\ y),\lambda z.inr(N\ z)) \\
&\quad \Rightarrow^{inl}\ (\lambda y.inl(M\ y))a \Rightarrow^{\beta}\ inl(M\ a)
\end{aligned}
$$

Y por otra parte:

$$
\begin{aligned}
&(O\ (inr(b))) = \\
&\quad (\lambda x{:}\alpha + \beta.case(x,\lambda y.inl(M\ y),\lambda z.inr(N\ z)))\ \ (inr(b)) \\
&\quad \Rightarrow^{\beta}\ case(inr(b),\lambda y.inl(M\ y),\lambda z.inr(N\ z)) \\
&\quad \Rightarrow^{inr}\ (\lambda y.inr(N\ z))b \Rightarrow^{\beta}\ inr(N\ b)
\end{aligned}
$$

Lema 4.3.- Si $\alpha' < \alpha$ y $\beta < \beta'$ con M y N como operadores de coerción respectivos y O es el operador de coerción asociado con la relación $\alpha\rightarrow\beta < \alpha'\rightarrow\beta'$ entonces para todo $f{:}\alpha\rightarrow\beta$ y $a{:}\alpha'$ se cumple:

$$(O\ f)(a) \Rightarrow (N\ (f\ (M\ a)))$$

Demostración: Inmediata a partir de la definición:

$$O \equiv (\lambda f{:}\alpha\rightarrow\beta.\lambda x{:}\alpha'.(N\ (f\ (M\ x))))$$

Teorema 4.2.- Los operadores de coerción en tipos recursivos dejan invariante la estructura del tipo.

Demostración:

Para tipos de datos recursivos recordemos que el operador de coerción está dado por el operador recursivo:

$$O \equiv Y(\lambda F{:}\mu t.\alpha\rightarrow\mu s.\beta.\lambda y{:}\mu t.\alpha.(fold(M'(unfold\ y)))) \qquad (4.6)$$

donde $F \in \mathbf{fv}(M')$ y M' se construye a partir del operador de coerción $M{:}\alpha\rightarrow\beta$, suponiendo que existe un $F{:}\mu t.\alpha\rightarrow\mu s.\beta$.

Probaremos la invariancia de la estructura por inducción sobre las llamadas recursivas del operador O. Recordemos la

propiedad central del operador de punto fijo:

$$(Y\ f) \Rightarrow f(Y\ f)$$

Y utilizando ésta propiedad en la expresión (4.6) tenemos que la primer llamada recursiva a O se reduce a:

$$\begin{aligned}&(((\lambda F{:}\mu t.\alpha{\to}\mu s.\beta.\lambda y{:}\mu t.\alpha.(fold(M'(unfold\ y)))\ O)\ m)\\ &\quad \Rightarrow^{\beta} ((\lambda y{:}\mu t.\alpha.(fold([O/F]\ M'(unfold\ y))))\ m)\\ &\quad\quad \Rightarrow^{\beta} (fold([O/F]\ M'(unfold\ m)))\end{aligned}$$

En resumen, tenemos que:

$$(O\ m) \Rightarrow (fold([O/F]\ M'(unfold\ m))) \qquad (4.7)$$

Recordemos que $M'{:}[\mu t.\alpha/t]\alpha \to [\mu s.\beta/s]\beta$, es decir, M' opera sobre el tipo desdoblado.

Al desdoblar el tipo $\mu t.\alpha$ en $[\mu t.\alpha/t]\alpha$ obtenemos un tipo construido a partir de $\to$,+ o $\times$.

Como hemos construido M' con las reglas correspondientes en la definición 4.2, a funciones, unión o producto, suponiendo que existe un operador de coerción $F{:}\mu t.\alpha{\to}\mu s.\beta$, entonces M' debe satisfacer los lemas 4.1,4.2,4.3, si F preserva la estructura.

Por hipótesis de inducción O preserva la estructura del tipo en la llamada recursiva más anidada. Por esto y lo que señalamos en el párrafo anterior [O/F]M' preserva la estructura.

Es fácil comprobar que los operadores *fold* y *unfold* dejan invariante la estructura de un término y por lo tanto:

$$(fold([O/F]M'\ (unfold\ m)))$$

es invariante en la estructura, con lo que el teorema queda demostrado.

Veamos para un caso particular como es que opera la coerción en tipos recursivos para ilustrar la propiedad que establecimos

en el teorema anterior.

Ejemplo 4.1.

Considérese el término:

$$l \equiv (\text{fold}(\text{inr } \langle 2, \text{fold}(\text{inr } \langle 3, \text{fold}(\text{inr } \langle 4, \text{fold } (\text{inl } \langle\rangle)\rangle)\rangle)\rangle)\rangle)$$

que es del tipo:

$$\mu t.\text{Unit} + \text{Nat} \times t.$$

Recordemos de la sección 4.3 que el operador de coerción para la relación

$$\mu t.\text{Unit} + \text{Nat} \times t < \mu t.\text{Unit} + \text{Int} \times t$$

está dado por el combinador:

$$O \equiv Y(\lambda F.\lambda m.(\text{fold}(M' \ (\text{unfold } m))))$$

donde M' es:

$$M' \equiv \lambda x{:}\text{Unit} + \text{Nat} \times \mu t.\alpha.\text{case}(x, \lambda z{:}\text{Unit}.\text{inl}(z), \lambda y{:}\text{Nat} \times \mu t.\alpha.\text{inr} \langle (C_{Nl}\ \pi_1(y)), (F\ \pi_2(y))\rangle)$$

Como habíamos visto anteriormente cualquier operador de coerción en tipos recursivos satisface la reducción:

$$(O\ l) \Rightarrow (\text{fold}([O/F]M'(\text{unfold } l)))$$

substituyendo el valor de l en el último término tenemos:

$$(\text{fold}([O/F]M'(\text{unfold}(\text{fold}(\text{inr } \langle 2, \text{fold}(\text{inr } \langle 3, \text{fold}(\text{inr } \langle 4, \text{fold}(\text{inl } \langle\rangle)\rangle)\rangle)\rangle)\rangle)))))$$

que por la regla de reducción (unfold-fold) se reduce a:

$$(\text{fold } ([O/F]M'(\text{inr } \langle 2, \text{fold}(\text{inr } \langle 3, \text{fold}(\text{inr } \langle 4,$$

```
          fold(inl <>)>)>)>)))
```

expandiendo el término [O/F]M' se obtiene:

```
     (fold(λx.case(x,λz:Unit.inl(z),λy:Nat × μt.α.
          inr<(C_NI π1(y)),(O π2(y))>)(inr <2,fold(inr <3,
               fold(inr <4,fold(inl <>)>)>)>)))
```

Y por β-reducción se llega a:

```
(fold(case(inr <2,fold(inr <3,fold(inr <4,fold(inl <>)>)>)>),
     λz:Unit.inl(z),λy:Nat × μt.α.inr<(C_NI π1(y)),(O π2(y))>)))
```

Reduciendo el *case* se tiene el término:

```
     (fold(λy.inr<(C_NI π1(y)),(O π2(y))>
          (<2,fold(inr <3,fold(inr <4,fold(inl <>)>)>)>)))
```

que por β-reducción se reduce a:

```
     (fold(inr <(C_NI 2),
             (O fold(inr <3,fold(inr <4,fold(inl <>)>)>))>))
```

Observemos que este término es precisamente la coerción de natural a entero del frente de la lista concatenado al resultado de llamar recursivamente al operador O sobre el resto de la lista.

En la siguiente aplicación recursiva del operador se tiene:

```
     (fold(inr <(C_NI 2),(fold(inr <(C_NI 3),(O fold(inr <4,
          fold(inl <>)>)>))>))>))
```

Y luego:

```
     (fold(inr <(C_NI 2),(fold(inr <(C_NI 3),(fold(inr <(C_NI 4),
          (O fold(inl <>))>))>))>))
```

Hasta llegar finalmente a:

$$(\text{fold}(\text{inr} <(C_{NI}\ 2), (\text{fold}(\text{inr} <(C_{NI}\ 3), (\text{fold}(\text{inr} <(C_{NI}\ 4),$$
$$(\text{fold}(\text{inl} <>))>))>))>))$$

Puede observarse que esta lista es similar a la original en el sentido de que tienen la misma estructura pero los elementos básicos de ésta última son el resultado de la coerción de las componentes de la lista original.

4.5. Extensiones al sistema de subtipos.

Es posible hacer varias extensiones al sistema de subtipos que permitirían enriquecerlo, haciéndolo más complejo.

Por ejemplo, si incluimos en el conjunto de operadores de coerción a *fold* y *unfold* sería equivalente a añadir al conjunto de reglas de inferencia los dos axiomas :

$$\Gamma \vdash [\mu t.\alpha/t]\alpha < \mu t.\alpha \qquad (\text{fold}_R)$$

$$\Gamma \vdash \mu t.\alpha < [\mu t.\alpha/t]\alpha \qquad (\text{unfold}_R)$$

Estos dos axiomas corresponden a nuestra intuición de que un tipo y sus desdoblamientos son idénticos. De hecho, Cardelli y Amadio en lugar de introducir las dos relaciones anteriores incluyen a la igualdad :

$$[\mu t.\alpha/t]\alpha = \mu t.\alpha$$

aunque en tal caso tendríamos que definir con precisión que es lo que se entiende por igualdad de tipos.

Por otro lado, podemos incluir en *Oc* al par de operadores *inl* e *inr* lo cual equivaldría a agregar a las reglas de subtipos el par de reglas :

$$\alpha < \alpha + \beta \qquad (\text{inl}_R)$$

$$\beta < \alpha + \beta \qquad (\text{inr}_R)$$

Estas reglas nos permitirían probar, por ejemplo, que las expansiones finitas de un tipo recursivo son subtipos del tipo recursivo. Por ejemplo el tipo Int × Int × Unit es subtipo de las listas de enteros, esto es:

$$\text{Int} \times \text{Int} \times \text{Unit} < \mu t.\text{Unit} + \text{Int} \times t$$

La relación anterior es imposible de demostrar en el sistema formal dado por la definición 4.1 a menos que se incluyan las reglas presentadas en esta sección.

4.6. Resumen.

Hemos presentado en este capítulo nuestra teoría constructiva de los subtipos en tipos de datos recursivos. En este enfoque particular determinamos de manera inductiva en la definición 4.2, a una familia de operadores o combinadores que preservan la estructura de los tipos de los términos a los cuales se aplican.

Decimos entonces que un tipo α es subtipo de algún tipo β si existe un término de tipo $\alpha \rightarrow \beta$ que pertenezca a la familia de operadores definidos en la sección 4.2.

Hemos también mostrado que este enfoque es consistente con el sistema de subtipos desarrollado por Cardelli y Amadio.

CAPITULO 5

TIPOS Y ARBOLES REGULARES

Un problema fundamental dentro del sistema de tipos que es objeto del presente estudio es la decidibilidad de la relación de subtipos, esto es, si existe un algoritmo para determinar si dados dos tipos cualquiera s y t, se tiene que s < t.

En gran medida el trabajo realizado por Cardelli y Amadio (1990) va dirigido a determinar este algoritmo. En su artículo, Cardelli y Amadio demuestran que la relación de subtipificación es decidible y encuentran un algoritmo de complejidad exponencial.

Para poder deducir su algoritmo, Cardelli y Amadio encuentran un mapeo de los tipos a los árboles infinitos que resultan de la expansión de los tipos recursivos. Al definir un orden parcial dentro de los árboles pueden entonces inducir un orden parcial dentro de los tipos de datos recursivos. Utilizando propiedades de los árboles regulares, que fueron ampliamente estudiadas por Courcelle (1986), se traduce el problema de la subtipificación en los árboles a un sistema de ecuaciones regulares. Esto es posible gracias a que, como demostró Courcelle, los árboles infinitos regulares (aquellos que poseen un número finito de subárboles distintos entre sí) pueden ser representados por un sistema de ecuaciones regulares. De esta manera el algoritmo encontrado por Cardelli y Amadio es, básicamente, un algoritmo de unificación sobre un sistema de ecuaciones regulares. Dicho algoritmo tiene la desventaja de ser de complejidad exponencial y, por lo tanto, de poca utilidad práctica.

En un estudio realizado más recientemente (Kozen, Palsberg y Schwartzbach, 1993) se encontró un algoritmo de subtipificación

más eficiente (de orden cuadrático) para el sistema de tipos de Cardelli y Amadio. Si bien en su artículo Cardelli y Amadio ya señalaban que el problema de la equivalencia de tipos de datos recursivos era reducible al problema de la equivalencia de autómatas de estado finito, mencionaban que para el problema de la subtipificación no existía algo análogo. Sin embargo, Kozen, Palsberg y Schwartzbach encontraron que es posible reducir el problema de la subtipificación al problema de determinar si el lenguaje regular aceptado por un autómata es vacío. Para poder llegar a esta conclusión, Kozen et. al. retoman la equivalencia entre los tipos recursivos y los árboles para construir un autómata que acepte aquellas trayectorias en los árboles que violen las reglas de subtipificación. Además de encontrar un algoritmo más eficiente, Kozen y sus colaboradores, hacen un planteamiento conceptualmente más simple que el propuesto originalmente por Cardelli y Amadio.

En el presente capítulo expondremos la relación existente entre los tipos recursivos y los árboles regulares que sirvieron de base a los trabajos realizados tanto por Cardelli y Amadio como por Kozen, Palsberg y Schwartzbach.

5.1. Arboles etiquetados.

Comenzaremos presentando la teoría sobre árboles etiquetados regulares planteándola de una manera muy general para después reducirla a nuestro caso particular en el que los nodos estarán etiquetados por los símbolos utilizados en nuestras expresiones de tipos.

Sea Σ un alfabeto finito donde a cada símbolo se le asocia una *aridad*, que representaremos como un superíndice. Denominaremos Σ_n al conjunto de elementos de Σ de aridad n. Como es usual ω representa al conjunto de números naturales y ω^* al conjunto de cadenas de longitud finita sobre ω (es decir, sucesiones finitas de números naturales).

Definición 5.1. Un *árbol* sobre Σ es una función parcial

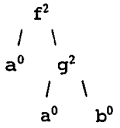
$$t:\omega^* \to \Sigma$$

cuyo dominio Dom(t) satisface las siguientes propiedades :

- Dom(t) no es vacío y es cerrado bajo prefijos.
- si $t(\alpha) \in \Sigma_n$ entonces $\alpha i \in$ Dom(t) para i = 0,...,n-1

El conjunto de todos los árboles sobre Σ es denotado como **Tree(Σ)**. Un elemento $\alpha \in \omega^*$ es una *hoja* de t si $\alpha \in$ Dom(t) y α no es prefijo propio de ningún otro elemento de Dom(t) o, de manera equivalente, si $t(\alpha) \in \Sigma_0$. □

Ejemplo 5.1. Supongamos que Σ = { f^2, g^2, a^0, b^0 } entonces el árbol A dado por la representación gráfica siguiente :



estaría dado formalmente como la función cuyo dominio es:

$$\text{Dom(A)} = \{\varepsilon, 0, 1, 10, 11\}$$

y que está definida como:

$$A(\varepsilon) = f^2$$
$$A(0) = a^0$$
$$A(1) = g^2$$
$$A(10) = a^0$$
$$A(11) = b^0$$

Se puede comprobar fácilmente que la función A satisface las propiedades dadas en la definición 5.1. Obsérvese además que 0, 10 y 11 son hojas de A. □

Diremos que un árbol t es finito si su dominio Dom(t) es un conjunto finito. Denotaremos al conjunto de árboles finitos sobre Σ como $\mathbf{Tree}_{fin}(\Sigma)$. De manera un tanto informal, definiremos una *trayectoria* como un subconjunto de Dom(t) que pueda ser ordenado linealmente bajo la relación de prefijos. A un elemento de Dom(t) le llamaremos *posición*.

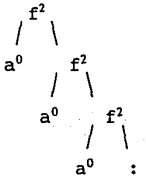
Definición 5.2. Sea t un árbol y $\alpha \in$ Dom(t). Definimos la función parcial $t{\downarrow}\alpha:\omega^* \to \Sigma$ por

$$t{\downarrow}\alpha(\beta) = t(\alpha\beta)$$

si $t{\downarrow}\alpha$ tiene dominio no vacío entonces es un árbol y es denominado el *subárbol de t en la posición* α. □

Definición 5.3. Se dice que un árbol t es *regular* si tiene un número finito de subárboles distintos; esto es, si el conjunto $\{ t{\downarrow}\alpha \mid \alpha \in \omega^* \}$ es finito. □

Ejemplo 5.2. Considérese el árbol infinito B definido sobre el alfabeto considerado en el ejemplo 5.1.:



El árbol B es regular y su dominio está determinado por la expresión regular 1^*+1^*0. □

En general, una característica muy importante de los árboles regulares es que sus dominios pueden ser descritos mediante expresiones regulares (Courcelle, 1986). Debido a esta propiedad es posible representar a los árboles regulares mediante autómatas

de estado finito, precisamente aquellos que aceptan las cadenas pertenecientes a los dominios de los árboles.

5.2. Autómatas.

En esta sección describiremos a los autómatas que representan a los árboles regulares y a los que denominaremos *autómatas de trayectorias*.

Definición 5.4. Sea Σ un alfabeto finito con aridades. Un *autómata de trayectorias* sobre Σ es la tupla :

$$M = (Q, \Sigma, q_0, \delta, l)$$

donde :

- Q es un conjunto finito de *estados*.
- $q_0 \in Q$ es el *estado inicial*.
- $\delta : Q \times \omega \to Q$ es una función parcial denominada la *función de transición* y
- $l : Q \to \Sigma$ es una función total denominada *función de etiquetado* tal que para cada estado $q \in Q$, si $l(q) \in \Sigma_n$ entonces $\delta(q,i)$ esta definido únicamente para $i = 0,1,\ldots,n-1$. □

Sea M un autómata de trayectorias. La función parcial δ se extiende de manera natural a la función parcial :

$$\hat{\delta} : Q \times \omega^* \to Q$$

inductivamente como sigue :

$$\hat{\delta}(q,\varepsilon) = q$$

$$\hat{\delta}(q,\alpha i) = \delta(\hat{\delta}(q,\alpha), i)$$

Para cualquier $q \in Q$ el dominio de $\hat{\delta}$ no es vacío y es cerrado bajo prefijo. Más aún, debido a la condición que garantiza la existencia de i-sucesores en la definición 5.4, la función parcial compuesta $l \cdot \hat{\delta}$ es un árbol.

Definición 5.5. Sea M un autómata de trayectorias. El *árbol representado por* M es el árbol t_M definido por :

$$t_M(\alpha) = l(\hat{\delta}(q_0, \alpha))$$

Un árbol t se dice *representable* si $t = t_M$ para algún autómata M. □

El siguiente resultado (expuesto en Kozen et. al. 1993) es de fundamental importancia para nuestro trabajo, debido a que nos permitirá formular un algoritmo de subtipificación.

Lema 5.1. Sea $t \in \text{Tree}(\Sigma)$. Las siguientes condiciones son equivalentes :

(i) t es regular
(ii) t es representable
(iii) t es descrito por un conjunto finito de ecuaciones que involucran al operador μ. □

Para la demostración de la equivalencia entre los incisos (i) y (ii) consultar Kozen et.al. (1993) y del inciso (iii) véase Courcelle (1986).

Ejemplo 5.3. El árbol regular mostrado en el ejem. 5.2 está representado por el autómata de la fig. 5.1. Obsérvese que la recurrencia de los subárboles se traduce en ciclos dentro del autómata que los representa.

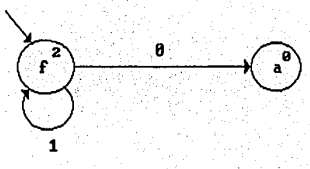


Fig. 5.1

5.3. Expansiones en árbol.

A continuación describiremos, siguiendo las ideas de Cardelli y Amadio, como asociar a cualquier tipo recursivo un árbol etiquetado, regular, lo que nos permitirá en consecuencia asociar a cada tipo recursivo un autómata de representación.

Considérese el alfabeto definido enseguida:

$$L = \{\bot^0, \top^0, \rightarrow^2, +^2, \times^2\} \cup \{t^0 \mid t \in TVar\}$$

donde los superíndices indican la aridad del símbolo. Entonces, Tree(L) y $\text{Tree}_{fin}(L)$ serán respectivamente el conjunto de árboles con etiquetas en L y el subconjunto de árboles finitos.

Entonces un árbol $A \in \omega^* \rightarrow L$ es una función parcial de ω^* (posiciones) a L (etiquetas de nodos), cuyo dominio es no-vacío y cerrado por prefijo y tal que cada nodo tiene un número de descendientes igual a la aridad de la etiqueta asociada.

Por ejemplo, el árbol dado por la representación gráfica siguiente:

$$A = \quad \begin{array}{ccc} & & \rightarrow^2 \\ & \swarrow & \searrow \\ \bot^0 & & +^2 \\ & & \swarrow \quad \searrow \\ & & s^0 \quad t^0 \end{array}$$

está definido por el mapeo :

$$\begin{aligned} A(\varepsilon) &= \rightarrow^2 \\ A(0) &= \bot^0 \\ A(1) &= +^2 \\ A(10) &= s^0 \\ A(11) &= t^0 \end{aligned}$$

y cualquiera otra secuencia de naturales no pertenece al dominio de A.

Enseguida podemos definir una función T : Type → Tree(L) de los tipos recursivos a los árboles infinitos. Esta función se define por inducción sobre las construcciones de tipos.

La función T está dada por :

$$T(\bot) \doteq \bot \qquad T(\top) \doteq \top \qquad T(t) \doteq t$$

$$T(\alpha\rightarrow\beta) \doteq \overset{\rightarrow}{\underset{T(\alpha)}{/}\ \backslash\ \underset{T(\beta)}{}} \qquad T(\alpha+\beta) \doteq \overset{+}{\underset{T(\alpha)}{/}\ \backslash\ \underset{T(\beta)}{}} \qquad T(\alpha\times\beta) \doteq \overset{\times}{\underset{T(\alpha)}{/}\ \backslash\ \underset{T(\beta)}{}}$$

$$T(\mu t.\alpha) \doteq \begin{cases} \bot & \text{si } \alpha \equiv \mu t_1 \ldots \mu t_n . t \ (t_i \neq t, i=1\ldots n) \\ T([\mu t.\alpha/t]\alpha) & \text{en cualquier otro caso} \end{cases}$$

Una propiedad muy importante del mapeo T es el enunciado a continuación :

Teorema 5.1. El mapeo T induce una biyección entre los tipos de datos no-recursivos **Tp** y los árboles finitos $\mathbf{Tree}_{fin}(L)$.

La demostración de este teorema es inmediata por inducción sobre los constructores de **Tp** ($\rightarrow, +, \times$).

El siguiente resultado es mencionado en el artículo de Cardelli y Amadio (1990) sin ser demostrado; por ser de fundamental importancia para el posterior desarrollo de nuestro trabajo decidimos no solamente incluirlo sino además demostrarlo

de manera completa.

Teorema 5.2. Para todo tipo $\tau \in \mu\mathbf{Tp}$ se tiene que $T(\tau)$ es árbol regular.

Demostración.

Si $\tau \in \mathbf{Tp}$ (tipos no recursivos) entonces, por el teorema 5.1, $T(\tau)$ es finito y por lo tanto regular. Demostremos el teorema para el caso recursivo por inducción sobre el número de variables acotadas por el operador μ.

Sea τ de la forma $\mu t.\alpha$ donde t es la única variable acotada, de tal manera que $T(\alpha)$ es finito. Sea

$$\Pi = \{ \pi \in \omega^* \mid (T(\alpha))(\pi) = t^0 \}$$

esto es, el conjunto de hojas de $T(\alpha)$ etiquetadas por t. Obsérvese que $\varepsilon \notin \Pi$ pues, en caso contrario, tendríamos $\tau \equiv \mu t.t$ que se toma equivalente a $\perp$.

Tenemos que para todo $\pi \in \Pi$ se cumple la igualdad :

$$(T([\mu t.\alpha/t]\alpha))(\pi\gamma) = (T(\mu t.\alpha))(\gamma) \qquad (5.1)$$

siempre que $\gamma \in \mathrm{Dom}(T(\mu t.\alpha))$. Por otra parte, de la definición de T se tiene :

$$T(\mu t.\alpha) = T([\mu t.\alpha/t]\alpha)$$

y de la ec. 5.1, se deduce que :

$$(T(\mu t.\alpha))(\pi\gamma) = (T(\mu t.\alpha))(\gamma) \qquad (5.2)$$

para todo $\pi \in \Pi$.

Definamos a la relación binaria R_Π sobre $\mathrm{Dom}(T(\mu t.\alpha))$ como el conjunto :

$$\{ (\gamma_1,\gamma_2) \mid \gamma_1 = \pi^{\bullet}\gamma_2 \text{ ó } \gamma_2 = \pi^{\bullet}\gamma_1 \text{ para algún } \pi \in \Pi \}$$

donde $\pi^{\bullet}$ denota a cero o más concatenaciones de π. Obviamente R_Π es una relación de equivalencia. Además, de la ec. 5.2, tenemos que si $(\gamma_1,\gamma_2) \in R_\Pi$ entonces

$$(T(\mu t.\alpha))\downarrow\gamma_1 = (T(\mu t.\alpha))\downarrow\gamma_2 \qquad (5.3)$$

es decir, los subárboles en las posiciones γ_1 y γ_2 son iguales.

De (5.3) se concluye que $\{ T(\mu t.\alpha)\downarrow\gamma \mid \gamma \in \omega^{\bullet} \}$ tiene a lo más tantos elementos como el conjunto cociente $\mathrm{Dom}(T(\mu t.\alpha))/R_\Pi$.

Demostremos, por reducción al absurdo, que si $\phi \in \mathrm{Dom}(T(\mu t.\alpha))$ entonces $\phi \in [\psi]_R$ para alguna $\psi \in \mathrm{Dom}(T(\alpha))$.

Supongamos que existe $\phi \in \mathrm{Dom}(T(\mu t.\alpha))$ que no es equivalente bajo R_Π con ningún elemento de $\mathrm{Dom}(T(\alpha))$. Sea ϕ' la cadena más pequeña de $[\phi]_R$. Entonces ϕ' no tiene a ningún elemento de Π como prefijo, y por lo tanto :

$$(T(\mu t.\alpha))(\phi') = (T([\mu t.\alpha/t]\alpha))(\phi') = (T(\alpha))(\phi') \qquad (5.4)$$

pero como $\phi' \notin \mathrm{Dom}(T(\alpha))$, de la igualdad (5.4) tendríamos que $\phi' \notin \mathrm{Dom}(T(\mu t.\alpha))$ lo cual es una contradicción. Hemos probado entonces que si $\phi \in \mathrm{Dom}(T(\mu t.\alpha))$ entonces es equivalente bajo R_Π con algún elemento perteneciente al conjunto $\mathrm{Dom}(T(\alpha))$ que, como señalamos anteriormente, es finito.

Tenemos entonces que la cardinalidad del conjunto cociente $\mathrm{Dom}(T(\mu t.\alpha))/R_\Pi$ es menor o igual a la cardinalidad de $\mathrm{Dom}(T(\alpha))$, y por lo tanto el conjunto de subárboles de $T(\mu t.\alpha)$ es finito, esto es, $T(\mu t.\alpha)$ es regular.

Supongamos ahora que tenemos n variables acotadas. Definiremos al conjunto Π respecto a la n-ésima variable acotada t_n, de la manera siguiente :

$$\Pi = \{ \pi \in \omega^* \mid (T(\mu t_1 \ldots \mu t_{n-1}.\alpha))(\pi) = t_n \}$$

y la relación de equivalencia R_Π en forma análoga a como lo hicimos en el caso anterior. De manera similar tendremos que el conjunto cociente $Dom(T(\mu t_1 \ldots \mu t_n.\alpha))/R_\Pi$ tiene tantos elementos como subárboles tiene $T(\mu t_1 \ldots \mu t_{n-1}.\alpha)$. Pero, por hipótesis de inducción, $T(\mu t_1 \ldots \mu t_{n-1}.\alpha)$ es regular y por lo tanto el cociente es finito. Podemos concluir que $T(\mu t_1 \ldots \mu t_n.\alpha)$ es regular. □

Debido al teorema anterior se tiene que para los tipos α, β, decidir si $T(\alpha) = T(\beta)$ es reducible al problema de la equivalencia de autómatas determinísticos de estado finito.

Cardelli y Amadio fundamentan su estudio en esta propiedad y, basándose en la teoría de Courcelle (ver Courcelle, 1986), hacen una traducción de árboles a sistemas de ecuaciones regulares, lo cual les permite plantear un algoritmo para resolver el problema de la relación de subtipos entre tipos recursivos, pero antes es necesario establecer un ordenamiento parcial en Tree(L) que induzca de manera natural una relación similar en los tipos recursivos.

5.4. Aproximaciones finitas.

En la sección anterior presentamos un mapeo de los tipos a los árboles. El ordenamiento parcial de los tipos no recursivos (el establecido por las reglas dadas en la definición 4.1, con excepción de la regla (μ_R)) induce un ordenamiento parcial sobre los árboles finitos $Tree_{fin}(L)$, de la manera siguiente :

Para $A,B \in Tree_{fin}(L)$, $A <_{fin} B \Leftrightarrow T^{-1}(A) < T^{-1}(B)$

A continuación veremos como Cardelli y Amadio extienden ésta noción a los árboles infinitos. Se define una familia de funciones :

$$\{ |_k : Tree(L) \rightarrow Tree_{fin}(L),\ k \in \omega \}$$

Dado $A \in Tree(L)$ su *corte al k-ésimo nivel* está definido como sigue :

$$A|_k(\pi) \equiv A(\pi) \text{ si } |\pi| < k$$

$A|_k(\pi) \equiv \perp$ si $|\pi| = k$, $A(\pi)$ está definido y π es *variante* en A

$A|_k(\pi) \equiv \top$ si $|\pi| = k$, $A(\pi)$ está definido y π es *contravariante* en A,

donde π es variante (contravariante) en A si a lo largo de la trayectoria que lleva de la raíz a π seleccionamos el lado izquierdo del nodo $\rightarrow$ un número par (impar) de veces, mientras que $|\pi|$ denota a la longitud de la cadena π.

Debido a que T es una biyección se puede extender ésta noción de aproximación finita a los tipos recursivos, de la manera siguiente :

$$\alpha|_k \equiv T^{-1}((T(\alpha))|_k)$$

Por ejemplo, si $\alpha = (t \rightarrow s) \rightarrow (t \rightarrow (t \rightarrow s))$ entonces :

$$\alpha|_1 = \perp \qquad \alpha|_2 = \top \rightarrow \perp$$

$$\alpha|_3 = (\perp \rightarrow \top) \rightarrow (\top \rightarrow \perp)$$

$$\alpha|_4 = (t \rightarrow s) \rightarrow (t \rightarrow (\top \rightarrow \perp))$$

mientras que $\alpha|_5 = \alpha$.

A continuación se extiende el ordenamiento parcial a los árboles infinitos y de esta manera se induce un ordenamiento sobre los tipos recursivos.

Definición 5.6. (Ordenamiento en árboles) Definimos la relación de orden parcial en árboles finitos (denotado como $<_{fin}$) como la relación dada por:

$$A <_{fin} B \Leftrightarrow T^{-1}(A) <_R T^{-1}(B)$$

para $A, B \in Tree_{fin}(L)$. Entonces la relación entre árboles $Tree(L)$ denotada como $<_\infty$, se define así :

$$A <_\infty B \Leftrightarrow \forall k \ (A|_k <_{fin} B|_k)$$

para $A, B \in Tree(L)$. Por último definimos la relación entre tipos denotada por $<_T$, como :

$$\alpha <_T \beta \Leftrightarrow T(\alpha) <_\infty T(\beta)$$

para todo par de tipos α y $\beta \in \mu Tp$. □

Teorema 5.3 (Consistencia de ordenamiento en árboles)
Sean $<_R$ la relación de orden parcial dada por las reglas definidas en 4.1 y $<_T$ la relación de orden parcial de la definición 5.6, entonces :

$$\alpha <_R \beta \Leftrightarrow \alpha <_T \beta$$

para cualquier par de tipos α y $\beta \in \mu Tp$. □

Para la demostración del teorema 5.3 remitimos al lector a la obra de Cardelli y Amadio. En nuestro trabajo este resultado no es necesario pues en el siguiente capítulo presentaremos una relación mas directa entre los subtipos y sus correspondientes árboles regulares que nos conducirá a un algoritmo de subtipificación.

5.5. Resumen.

Hemos presentado en este capítulo una introducción a la teoría de los árboles y sus representaciones mediante autómatas. Se ha establecido un mapeo de los tipos a los árboles regulares que tiene la propiedad de ser una biyección entre los tipos no recursivos y los árboles finitos. Se ha demostrado además que la imagen bajo este mapeo de cualquier tipo es siempre un árbol regular. Al llevar a cabo esta demostración se han encontrado clases de equivalencia asociadas a los subárboles de la imagen, las cuales nos permitirán encontrar a los autómatas de representación de los tipos. Esto último lo realizaremos en el siguiente capítulo.

CAPITULO 6

UN ALGORITMO DE SUBTIPIFICACION

Utilizando las ideas expuestas en el capítulo anterior, formularemos en el presente, un algoritmo que no solamente nos permitirá determinar cuando un par de tipos están en relación de subtipos sino además encontrar explícitamente el operador de coerción entre esos tipos. El algoritmo que presentaremos está basado en una generalización del desarrollado por Kozen, Palsberg y Schwartzbach, (al que denominaremos como algoritmo KPS) para abarcar todos los constructores de tipos que se han contemplado a través del presente trabajo.

Demostraremos además la consistencia de nuestro algoritmo respecto al ordenamiento parcial en tipos que definimos en el capítulo 4.

6.1. Subtipos y árboles.

En esta sección estableceremos un resultado que vinculará a los árboles de representación de un par de tipos que se encuentren en relación de subtipos. Veremos que si dos tipos son comparables (de acuerdo a la relación de subtipificación) entonces los árboles de representación tienen una estructura similar, en el sentido que estableceremos en el siguiente teorema.

Teorema 6.1. Sean α y β dos tipos pertenecientes a μ**Tp**. Se cumple que $\Gamma \vdash \alpha < \beta$, de acuerdo a las reglas de la definición 4.1, si y solo si para toda cadena $\pi \in \mathrm{Dom}(T(\alpha)) \cap \mathrm{Dom}(T(\beta))$ se cumple alguna de las siguientes condiciones, en donde $(T(\alpha))(\pi) = \sigma_1$ y $(T(\beta))(\pi) = \sigma_2$,

(I) $\sigma_1 = \bot^0$ y π es variante o $\sigma_1 = \top^0$ y π es covariante,

(II) $\sigma_2 = \top^0$ y π es variante o $\sigma_2 = \bot^0$ y π es covariante,

(III) σ_1 y $\sigma_2 \in L_0$ y $\sigma_1 < \sigma_2 \in \Gamma$, ó $\sigma_1 > \sigma_2 \in \Gamma$ dependiendo si π es variante o covariante.

(IV) $\sigma_1 = \sigma_2$ □

Demostración.

(Tipos no recursivos) :

(⇒) La demostración de esta parte se debe hacer por inducción sobre los reglas de subtipificación dadas en la definición 4.1.

Para la regla de reflexividad (eq_R), si $\Gamma \vdash \alpha < \alpha$, es evidente que todas los elementos del dominio de $T(\alpha)$ cumplen con la condición (IV).

Si tenemos que $\Gamma \vdash \bot < \alpha$, de acuerdo a la regla ($\bot_R$), entonces $\mathrm{Dom}(T(\bot)) \cap \mathrm{Dom}(T(\alpha)) = \{ \varepsilon \}$ y $(T(\bot))(\varepsilon)$ cumple con la condición (I). De manera similar, se puede demostrar la validez para las reglas ($\top_R$) y ($assmp_R$), cuyos árboles finitos asociados tienen únicamente como intersección de sus dominios a la cadena nula y esta satisface, respectivamente, las condiciones (II) y (III).

Ahora, supongamos que $\Gamma \vdash \alpha < \alpha'$ y $\Gamma \vdash \beta < \beta'$ cumplen con el teorema. De acuerdo con la regla ($\times_R$) se tiene que :

$$\Gamma \vdash \alpha \times \beta < \alpha' \times \beta'$$

Sea $\pi \in \mathrm{Dom}(T(\alpha \times \beta)) \cap \mathrm{Dom}(T(\alpha' \times \beta'))$. Si $\pi = \varepsilon$, entonces π cumple con la condición (IV). Si $\pi = 0\phi$ entonces $\phi \in \mathrm{Dom}(T(\alpha)) \cap \mathrm{Dom}(T(\beta))$ y, por hipótesis, cumple con alguna de la condiciones dadas, y por lo tanto, dado que $(T(\alpha \times \beta))(0\phi) = (T(\alpha))(\phi)$ y $(T(\alpha' \times \beta'))(0\phi) = (T(\alpha'))(\phi)$, entonces π satisface la misma condición que ϕ. De manera similar, si $\pi = 1\phi$ se tiene que ($T(\alpha$

$\times\beta))(1\phi) = (T(\beta))(\phi)$ y $(T(\alpha' \times\beta'))(1\phi) = (T(\beta'))(\phi)$ y entonces π cumple la misma condición que ϕ.

De esta misma forma podemos probar el resultado para la regla ($+_R$). En el caso de la regla ($\rightarrow_R$) hay que tomar en cuenta que $(T(\alpha \rightarrow \beta))(0\phi) = (T(\alpha))(\phi)$ pero ϕ es variante si 0ϕ es contravariante y viceversa.

($\Leftarrow$) Ahora demostraremos que si un par de árboles finitos, A y B, cumplen con las condiciones del teorema entonces los tipos asociados están en relación de subtipos. Haremos la demostración por inducción sobre el numero de elementos en la intersección de los dominios de los árboles.

Supongamos que la intersección de los dominios es un conjunto de un solo elemento. La cerradura bajo prefijo nos garantiza que este elemento debe ser la cadena nula ε. Entonces tenemos los siguientes casos :

Si ε cumple con la condición (I), entonces $T^1(A) = \perp$ dado que ε es variante, y por la regla ($\perp_R$), tenemos $T^1(A) < T^1(B)$. Si satisface la condición (II), se tiene que $T^1(B) = \top$ y por lo tanto $T^1(A) < T^1(B)$. Obsérvese que si se cumple la condición (II) y (IV), se tiene de inmediato la relación de subtipos.

Ahora analicemos el caso en que la intersección de los dominios en los árboles tiene más de un elemento.

Primero tomemos la cadena nula ε que, debido a la cerradura bajo prefijo, se encuentra en la intersección de los dominios. Si ε cumple con alguna de las condiciones, esta debe ser la (IV), ya que de cumplirse alguna otra, tendríamos que ε sería el único elemento de la intersección. Entonces $A(\varepsilon) = B(\varepsilon)$ y además deben ser iguales a $\rightarrow^2$, $+^2$ ó $\times^2$. Debido a lo anterior 0 y 1 $\in$ Dom(A) $\cap$ Dom(B).

Tomemos una cadena en Dom(A) $\cap$ Dom(B) diferente a ε, entonces esta cadena será de la forma 0ϕ ó 1ϕ. Es evidente que si

0ϕ cumple con alguna de las cuatro condiciones, también lo hará ϕ. Entonces las cadenas de Dom(A↓0) ∩ Dom(B↓0) satisfacen las condiciones del teorema y tenemos que T^1(A↓0) < T^1(B↓0) si A(ε) = B(ε) = $\times^2$ ó $+^2$, y T^1(A↓0) > T^1(B↓0) si A(ε) = B(ε) = $\rightarrow^2$. Por otro lado, si 1ϕ cumple con alguna de las condiciones, entonces ϕ cumple con la misma condición de 1ϕ.
Por lo tanto, T^1(A↓1) < T^1(B↓1) y podemos concluir entonces que T^1(A) < T^1(B).

(Tipos recursivos)

Tomemos un par de tipos recursivos $\mu s.\alpha$ y $\mu t.\beta$, en donde s y t son las únicas variables acotadas en α y β.

Si suponiendo que s < t podemos demostrar que $\alpha < \beta$ entonces, de acuerdo a la regla (μ_R) de la definición 4.1, se tiene que $\mu s.\alpha < \mu t.\beta$. Por otra parte, dado que α y β son tipos no recursivos, entonces de acuerdo a la primera parte de esta demostración, cumplen con el teorema. De la misma manera en que procedimos en la demostración del teorema 5.2, podemos construir una partición en Dom($T(\mu s.\alpha)$) ∩ Dom($T(\mu t.\beta)$) tal que cada elemento en esta intersección es equivalente a algún elemento de Dom($T(\alpha)$) ∩ Dom($T(\beta)$). Es fácil verificar que todos los elementos de una clase de equivalencia satisfacen la misma condición de las cuatro dadas en el enunciado del teorema.

Es sencillo, procediendo por inducción, probar este resultado cuando el número de variables acotadas por el operador μ es mayor a 1. □

6.2. De tipos a autómatas.

Puesto que es posible asociar a cada tipo un árbol regular y como los árboles regulares pueden ser representados mediante autómatas, entonces existe un autómata asociado a cada tipo recursivo. En esta sección describiremos a los autómatas de representación de tipos.

Definición 6.1. Sea $\tau \in \mu\mathbf{Tp}$. Sea R_Π la relación sobre $\mathrm{Dom}(T(\tau))$ de equivalencia en subárboles (tal como se definió en la demostración del teorema 5.2). Entonces *el autómata de representación de* τ, denotado por M_τ, es la tupla :

$$M_\tau = (Q, L, q_0, \delta, l)$$

donde :

- Q es el conjunto de *estados*, dado por:

  $$Q = \mathrm{Dom}(T(\tau))/R_\Pi,$$

- $L = \{\bot^0, \top^0, +^2, \times^2, \rightarrow^2\} \cup \{\ t^0 \mid t \in \mathbf{TVar}\ \}$,

- $q_0 = [\varepsilon]_R$, es el *estado inicial*,

- δ: $Q \times \omega \rightarrow Q$, denominada la *función de transición*, es la función parcial dada por :

  $$\delta([\phi]_R, i) = [\phi i]_R$$

  siempre que $\phi i \in \mathrm{Dom}(T(\tau))$,

- $l: Q \rightarrow L$, es la *función de etiquetado*, dada por :

  $$l([\phi]_R) = (T(\tau))[\phi]_R$$

□

Si construimos la extensión de δ, como la función $\hat{\delta}: Q \times \omega^* \rightarrow Q$, definida como :

$$\hat{\delta}(q, \varepsilon) = q$$

$$\hat{\delta}(q, \phi i) = \delta(\hat{\delta}(q, \phi), i)$$

entonces es fácil verificar que $l(\hat{\delta}(q_0, \phi)) = (T(\tau))(\phi)$, de donde resulta la validez del siguiente lema :

Lema 6.1. Sea $\tau \in \mu$Tp. Si M_τ es el autómata de representación de τ entonces M_τ es el autómata de representación del árbol regular $T(\tau)$.

Ejemplo 6.1. Consideremos el tipo recursivo :

$$\tau = \mu t.\text{Unit} + \text{Int} \times t$$

entonces el conjunto $\Pi = \{ 11 \}$ es respecto al cual se hace la relación de equivalencia R_Π sobre $\text{Dom}(T(\tau))$. Entonces las clases de equivalencia serán las siguientes :

$$[\varepsilon]_R = \{ \varepsilon, 11, 1111, 111111, \ldots\}$$

$$[0]_R = \{ 0, 110, 11110, 1111110, \ldots\}$$

$$[1]_R = \{ 1, 111, 11111, 1111111, \ldots\}$$

$$[10]_R = \{ 10, 1110, 111110, 11111110, \ldots\}$$

y el autómata de representación será $M_\tau = (Q, L, q_0, \delta, l)$ donde :

$$Q = \{ [\varepsilon]_R, [0]_R, [1]_R, [10]_R \}$$

y $q_0 = [\varepsilon]_R$. La función δ estará dada por :

$$\delta([\varepsilon]_R, 0) = [0]_R$$

$$\delta([\varepsilon]_R, 1) = [1]_R$$

$$\delta([1]_R, 0) = [10]_R$$

$$\delta([1]_R, 1) = [\varepsilon]_R$$

Mientras que la función l se define como :

$$l([\varepsilon]_R) = +^2$$

$$l([0]_R) = \text{Unit}^0$$

$$l([1]_R) = {}'\times^2$$

$$l([10]_R) = \text{Int}^0$$

La representación gráfica de M_r se muestra en la figura 6.1.

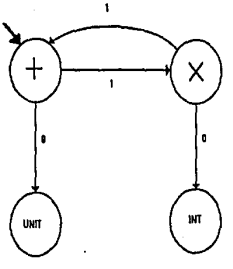


Fig. 6.1

6.3. Algoritmo de subtipificación.

En la primera sección de este capítulo mostramos que dos tipos se encuentran en relación de subtipos si y solo si las etiquetas de los nodos que se encuentran en posiciones similares en las expansiones en árbol de los tipos, cumplen determinadas condiciones. De tal manera que, para determinar si un tipo es subtipo o no de cualquier otro tipo, bastaría buscar un par de nodos en los árboles que no cumplieran con las condiciones dadas en el teorema 6.1. La idea del algoritmo KPS es construir un autómata que acepte las cadenas que son contraejemplo de dichas condiciones. Este autómata es el producto de los autómatas de representación de los tipos cuya relación de subtipos se quiere determinar. Demostraremos en seguida la consistencia de dicho

algoritmo, pero primero demos algunas definiciones preliminares.

Definición 6.2. (Autómata producto)

Sean $M_1 = (Q_1, L, q_1, \delta_1, l_1)$ y $M_2 = (Q_2, L, q_2, \delta_2, l_2)$ autómatas de representación. Definimos al *autómata producto* de M_1 y M_2, denotado $M_1 \otimes M_2$, como el autómata :

$$M_1 \otimes M_2 = (Q_1 \times Q_2, L, q_0, \delta, l)$$

donde $q_0 = (q_1, q_2)$, la función de transición δ está definida como $\delta((p,q),i) = (\delta_1(p,i), \delta_2(q,i))$ para $p \in Q_1$, $q \in Q_2$ e $i \in \omega$; finalmente la función etiquetadora l, esta definida como : $l(p,q) = (l_1(p), l_2(q))$ para $p \in Q_1$, $q \in Q_2$. □

Teorema 6.2. (Consistencia del algoritmo KPS).

Sean M_α y M_β los autómatas de representación de los tipos α y β, respectivamente. Sea F un conjunto de estados finales en el autómata $M_\alpha \otimes M_\beta$ tales que si $(p,q) \in F$ entonces $l(p,q)$ no cumplen con ninguna de las condiciones (I) a (IV) del teorema 6.1.

Entonces $\alpha < \beta$ si y solo si el lenguaje aceptado por $M_\alpha \otimes M_\beta$ es vacío.

Demostración.

La validez del teorema resulta inmediata del teorema 6.1 y del hecho de que si una cadena ϕ es aceptada por el autómata entonces :

$$l(\hat{\delta}((q_1,q_2),\phi)) = ((T(\alpha))(\phi), (T(\beta))(\phi))$$

y, en consecuencia existen $(T(\alpha))(\phi)$ y $(T(\beta))(\phi)$ que no cumplen con ninguna de las condiciones del teorema 6.1. □

El algoritmo KPS consiste en construir el autómata producto para el par de tipos y, haciendo un recorrido de los estados del autómata, determinar si algún estado viola las condiciones de subtipos. Para hacer este recorrido se requiere una pila cuyos elementos pertenecen al conjunto $Q_1 \times Q_2 \times \{0,1\}$ y un arreglo para

registrar cuando un nodo ha sido visitado o no. Obsérvese que en la siguiente descripción del algoritmo no se requiere construir explícitamente al autómata producto.

Algoritmo 6.1. (Algoritmo KPS)

Sean $M_\alpha = (Q_\alpha, L, q_0^\alpha, \delta_\alpha, l_\alpha)$ y $M_\beta = (Q_\beta, L, q_0^\beta, \delta_\beta, l_\beta)$ los autómatas de representación de los tipos α y β, respectivamente.

```
Hacer Push(q0^α, q0^β, 0);
Mientras la Pila no este vacía repetir
     inicio
          Hacer (p,q,i) = Pop();
          Si (p,q) no ha sido visitado
               Si l_α(p) = l_β(q) ∈ {+², ×², →²} entonces
                    inicio
                      Si l_α(p) = →² entonces
                            Push(δ_α(p,0), δ_β(q,0), i+1 mod 2)
                      si_no Push(δ_α(p,0), δ_β(q,0), i) ;
                      Push(δ_α(p,1), δ_β(q,1), i);
                    fin
               si_no
                    Si Ordenado(p,q,i) = falso entonces
                      Terminar (Fracaso);
     fin;
Terminar (Exito).
```

donde el predicado Ordenado(p,q,i) es cierto si :

1) $i = 0$ y $l(p) = \bot^0$ ó $l(q) = \top^0$ ó $l(p) < l(q)$

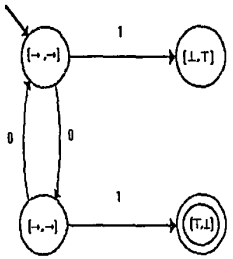
ó

2) $i = 1$ y $l(p) = \top^0$ ó $l(q) = \bot^0$ ó $l(p) > l(q)$ □

Ejemplo 6.2. Consideremos los tipos $\mu v.(v \to \bot)$ y $\mu u.(u \to \top)$. Los autómatas de representación de estos tipos se muestran en la figura 6.3.

Fig. 6.3

El autómata producto se muestra en la figura 6.4. Dado que existe un estado final alcanzable desde el estado inicial, se tiene que estos tipos en particular no cumplen con la relación de subtipos. El lenguaje aceptado por el autómata es (00)*01.



(Figura 6.3)

6.4. Construcción del operador de coerción.

El recorrido que se hace en el algoritmo KPS del autómata producto de los autómatas de representación de un par de tipos, no solamente establece si esos tipos están en relación de subtipos sino que además proporciona la información suficiente para construir el operador de coerción asociado a esta relación. De esta manera podemos extender el algoritmo KPS para construir una función que calcule el operador de coerción u origine una condición de error en caso de que los tipos no estén en relación de subtipos.

Algoritmo 6.2. (Construcción del operador de coerción)

Sean $M_\alpha = (Q_\alpha, L, q_0^\alpha, \delta_\alpha, l_\alpha)$ y $M_\beta = (Q_\beta, L, q_0^\beta, \delta_\beta, l_\beta)$ los autómatas de representación de los tipos α y β, respectivamente.

La función $COC(q_0^\alpha, q_0^\beta, 0)$ devuelve un λ-término que es el operador de coerción de α a β, si $\alpha < \beta$ y en caso contrario se origina una condición de error.

```
COC(p,q,i)
Inicio
     Si l_α(p) = l_β(q) = σ ∈ {+², ×², →²} entonces
       inicio
          Si (δ_α(p,0),δ_β(q,0)) ya ha sido visitado
               Hacer Op_1 = f, donde f es un nuevo símbolo
               Añadir Op_1 a la lista de variables acotadas
                     en (δ_α(p,0),δ_β(q,0))
          si_no
               Si σ = →² entonces
                    Hacer Op_1 = COC(δ_α(p,0),δ_β(q,0),i+1 mod 2);
               si_no
                    Hacer Op_1 = COC(δ_α(p,0),δ_β(q,0),i);

          Si (δ_α(p,1),δ_β(q,1)) ya ha sido visitado
               Hacer Op_2 = f, donde f es un nuevo símbolo
```

```
                Añadir Op2 a la lista de variables acotadas
                        en (δα(p,1),δβ(q,1))
            si_no
                Hacer Op2 = COC(δα(p,1),δβ(q,1),i);
            En caso de que σ sea
              (+) Hacer T = λx.case(x,λy.inl(Op1 y),λz.inr(Op2 z))
              (×) Hacer T = λx.<(Op1 (π1 x)),(Op2 (π2 x))>
              (→) Hacer T = λx.λy.(Op2 (x (Op1 y)))
            Para toda F en la lista de var. acotadas en (p,q)
                Hacer T = Y(λF.λx.(fold (T (unfold x)))
        fin
    sino
      Si lα(p) = s^0 y lβ(q) = t^0 entonces
        si i = 0 y s < t entonces
                hacer T = C:s→t
        sino  si i = 1 y t < s entonces
                    hacer T = C:t→s
              sino Error
      sino Error;
    Devolver T;
Fin.                                                          □
```

Puede observarse que el algoritmo anterior no es más que el recorrido recursivo en profundidad del autómata producto de las representaciones de los tipos. Cada vez que se regresa a un estado del autómata después de visitar los estados conexos se construye el operador de coerción asociado.

Los teoremas 6.1 y 6.2 así como los resultados obtenidos en el capítulo 4 (especialmente el teorema 4.1) nos permiten asegurar que el algoritmo anterior es correcto.

6.5. Resumen.

En este capítulo mostramos que si un par de tipos se encuentran en relación de subtipos entonces los árboles regulares asociados deben satisfacer cierta correspondencia en sus estructuras y las etiquetas que se encuentran en sus nodos. De esta manera, se define a un autómata que acepta precisamente el lenguaje formado por las cadenas que, evaluadas en los árboles, nos dan nodos que no cumplen con dicha correspondencia.

Si el lenguaje aceptado por el autómata es vacío entonces los tipos están en relación de subtipos. Esta es básicamente la idea del algoritmo KPS.

Por ultimo, hicimos una extensión al algoritmo KPS para tener un algoritmo que construya el operador de coerción entre dos tipos que estén en relación de subtipos.

CONCLUSIONES

A lo largo del presente trabajo hemos estudiado un sistema de tipos particular, definido sobre el λ-cálculo. Las características centrales de nuestro sistema es que posee tipos recursivos y una relación de orden parcial a la que denominamos relación de subtipos.

En el primer capítulo hicimos una breve introducción al λ-cálculo puro, presentando algunas de sus propiedades más relevantes. En el desarrollo de nuestro trabajo utilizamos al cálculo lambda como un lenguaje de programación sencillo sobre el cual definimos nuestro sistema de tipos. Dentro del cálculo lambda puro no existe ninguna distinción entre sus expresiones de la misma manera en que, en principio, no puede distinguirse entre los datos y las instrucciones dentro de la memoria de una computadora. Al introducir la noción de tipo dentro del lambda cálculo se hace la separación de las expresiones, en operaciones y valores a los cuales se pueden aplicar dichas operaciones.

Es a partir del segundo capítulo que comenzamos a exponer nuestro sistema de tipos. En ese capítulo precisamos las nociones tanto de tipo como de tipo recursivo. Definimos entonces un cálculo lambda tipificado, esto es, un cálculo lambda en el cual a cada expresión se le asocia un tipo de acuerdo a ciertas reglas de tipificación. También expusimos las reglas de reducción dentro de este λ-cálculo tipificado, las cuales representan el aspecto "computacional" de nuestro lenguaje.

En el tercer capítulo establecimos formalmente tanto las reglas para formar términos bien tipificados así como las reglas de reducción de términos. Mostramos que en nuestro λ-cálculo tipificado, a diferencia de lo que ocurre en sistemas parecidos, no existe normalización fuerte. Más aún, mostramos que para cada

tipo existe un término que no es normalizable. Esta característica, que podría parecer indeseable, es la que nos da la posibilidad de construir operadores de punto fijo que están bien tipificados. Encontramos que cada tipo tiene asociado un operador de punto fijo, lo cual nos permite definir operaciones recursivas sobre todos los tipos de nuestro sistema. En el capítulo señalado desarrollamos por completo un ejemplo de como se aplican los operadores de punto fijo para realizar operaciones sobre tipos de datos recursivos.

En el capítulo cuarto precisamos la idea de subtipo proporcionando un conjunto de reglas que nos permiten inferir cuando un tipo es subtipo de cualquier otro tipo. Mostramos que para cada par de tipos en relación de subtipos, existe un término bien tipificado que mapea los valores del subtipo al supertipo. Al conjunto de operadores así formados les dimos la denominación de operadores de coerción. Entre las características importantes de los operadores de coerción se encuentra que preservan la estructura de los datos.

En el quinto capítulo, siguiendo las ideas propuestas tanto por Cardelli y Amadio (1990) como por Kozen, Palsberg y Schwartzbach (1993), establecimos una relación entre los tipos recursivos y los árboles regulares etiquetados. Presentamos un ordenamiento parcial en los árboles (idea original de Cardelli y Amadio) que es consistente con el orden parcial de subtipos. En lugar de utilizar esta idea, encontramos una relación más directa entre los subtipos y los árboles. En el inicio del capítulo sexto encontramos una correspondencia que existe entre los árboles de representación de un par de tipos en relación de subtipo. Esta correspondencia es condición necesarias y suficiente para establecer una relación de orden.

Basándonos en lo anterior propusimos una generalización del algoritmo de Kozen, Palsberg y Schwartzbach, a la que denominamos algoritmo KPS, para determinar cuando un tipo es subtipo de algún otro, y mostramos la validez del algoritmo. Haciendo una extensión al algoritmo KPS originamos un algoritmo para

construir, de manera eficiente, el operador de coerción asociado a una relación de subtipos.

Si bien nuestro desarrollo está fuertemente basado en los citados trabajos de Cardelli y Amadio (1990), y Kozen, Palsberg y Schwartzbach (1993), hemos hecho varias aportaciones. En su artículo original, Cardelli y Amadio presentan inicialmente al sistema de tipos aquí estudiado, incluyendo los constructores de producto cartesiano y unión disjunta, pero terminan centrando su estudio en el caso del constructor de función. En el artículo de Kozen et. al. los constructores de suma y unión disjunta son por completo eliminados, obteniéndose un sistema de tipos más simple. En el presente trabajo hemos contemplado siempre al conjunto completo de constructores de tipos.

Hemos querido, además de estudiar las propiedades del sistema de tipos, analizar las propiedades computacionales de nuestro cálculo tipificado. Para llevar a cabo esto, dimos reglas que permitieran reducir los términos de tipo unión disjunta y producto cartesiano. Al introducir estas reglas estamos acercándonos más a un lenguaje de programación con utilidad práctica.

Una contribución importante de nuestro trabajo ha sido el estudio de los operadores de coerción así como el algoritmo que permite su construcción.

A partir de aquí se pueden abrir varias líneas de investigación. La semántica de nuestro sistema de tipos sería sin duda un terreno fértil para posteriores desarrollos. Otro reto se encuentra en la formulación de un algoritmo de subtipificación para un sistema de tipos que no solamente contemple tipos recursivos sino además cuantificadores universales y existenciales sobre tipos, es decir, tipos parametrizados y tipos de datos encapsulados.

BIBLIOGRAFIA

Asperti, Andrea and Giuseppe Longo. **Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist.** MIT Press, Foundation of Computing Series, 1991.

Barendregt, H.P. **The Lambda Calculus : Its Syntax and Semantics.** North-Holland,1984.

Cardelli,L. and Robert M. Amadio. "Subtyping Recursive Types. " **Systems Research Center Report No. 62**, DEC,1990.

Cardelli, L. and Peter Wegner. "On Understanding Types, Data Abstraction and Polymorphism." **ACM Computing Surveys**, Vol. 17, No. 4, December 1985.

Courcelle,B. "Equivalence and Transformation of regular systems - applications to recursive program schemes and grammars." **Theoretical Computer Science**, 42, pp. 1-122, 1986.

Gallier, Jean. "Constructive Logics. Part I : A Tutorial on Proof Systems and Typed λ-Calculi." **Paris Research Laboratory Report No. 8**, DEC, May 1991.

Hopcroft, John E., and Jeffrey Ullman. **Introduction to Automata Theory, Languages and Computation.** Addison-Wesley, 1979.

Hudak, Paul. "Conception, Evolution, and Application of Functional Programming Languages." **ACM Computing Surveys**, Vol. 21, No. 3, September 1989.

Kozen, Dexter, Jens Palsberg, and Michael I. Schwartzbach. "Efficient Recursive Subtyping." **Proc. of the Annual ACM**

Symposium on Principles of Programming Languages.pp. 419-428.ACM, January 1993.

MacLennan, B.J. **Functional Programming: Practice and Theory.** Addison-Wesley, 1990.

Mitchell, J.C. "Type systems for programming languages." **Technical Report CS-89-1277**, Stanford University, July 1989.