Especificación e Implementación de un Lenguaje de Programación Lógica con Tipos

Mario Cáccamo y Claudio Delrieux

Grupo de Investigación en Programación Declarativa e Inteligencia Artificial
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Alem 1253 - (8000) Bahía Blanca - ARGENTINA
e-mail: {mcaccamo,usdelrie}@criba.edu.ar

Palabras Clave: Programación Lógica - Teoría de Tipos - Metaintérpretes

Resumen:

Se presenta la especificación de un lenguaje de programación en lógica con tipos polimorfos, junto con el desarrollo de un intérprete para el mismo en PROLOG. El polimorfismo permite, entre otras cosas, programar procedimientos genéricos, con argumentos que no tienen un tipo declarado estáticamente. La incorporación de un sistema de tipos mejora el soporte de abstracción provisto por los lenguajes de programación lógica. Los tipos proveen un lenguaje claro para la especificación de interfases y herramientas para la depuración algorítmica Esta información puede utilizarse para la optimización de código y es de una ayuda inestimable para detectar errores de programación en tiempo de compilación.

Especificación e Implementación de un Lenguaje de Programación Lógica con Tipos

1 Introducción

El impacto de la teoría de tipos en las ciencias de la computación tiene como consecuencia actualmente su surgimiento como una disciplina independiente y cuyos resultados son de relevancia. Sin embargo, el desarrollo histórico de la programación lógica prescindió de la noción de tipo [7]. Esta ausencia es una desventaja tanto en aspectos teóricos como en el desarrollo de grandes sistemas de software en dicho paradigma.

El propósito de este trabajo es estudiar el desarrollo de un lenguaje lógico con tipos polimorfos, tanto paramétricos como por inclusión, y la implementación de un intérprete para el mismo. Existen resultados que plantean la equivalencia entre programas lógicos con tipos y programas lógicos sin tipos. A partir de este hecho podemos considerar al sistema de tipos como una facilidad puesta sobre el lenguaje. El trabajo se reduce entonces al desarrollo de un metaintérprete que permita ejecutar la parte correspondiente al sistema de tipos, y obtener un programa equivalente sin especificaciones de tipo. El entorno fue programado para ejecutarse en un sistema PROLOG.

En la sección 2 encontramos una introducción general al concepto de tipo, su evolución en los lenguajes de programación y especialmente su impacto en el paradigma lógico. En la sección 3 damos una descripción del lenguaje de tipos y de la sintaxis de $\mathcal{T}$-LOG. La sección 4 se refiere a la implementación del entorno, y se tratan aspectos pragmáticos relacionados al chequeo de tipos, de cómo determinar si un programa es correcto con respecto a su declaración, y una introducción a la inferencia de tipos. La sección 5 se presentan algunos ejemplos de programas lógicos con tipos y su ejecución en el metaintérprete. Por último, la sección 6 está dedicada a la evaluación de los resultados obtenidos, las conclusiones y el trabajo futuro.

2 Tipos

El propósito tradicional de los tipos en los lenguajes de programación ha sido proveer una indicación en tiempo de compilación o ejecución de los errores de escritura en un programa. El chequeo de tipos previo a la ejecución se realiza sobre el código de manera estática, evitándose posibles errores que resultarían en un mal funcionamiento del programa. Los tipos imponen restricciones que fuerzan la consistencia de los programas. El diseñador de un lenguaje o un compilador elije una representación que respete las propiedades esperadas de un tipo de dato y que a su vez permita desarrollar las operaciones de manera eficiente. El nivel de abstracción incorporado por los tipos permite, además, facilitar el objetivo de portabilidad de los lenguajes.

En [3] se da una definición informal de tipo que nos permite introducir, entre otros, el significado de polimorfismo paramétrico, y que soporte el concepto de subtipo. Partamos de un universo V con todos los *valores* posibles, conteniendo valores simples como enteros, estructuras de datos como listas o árboles, etc. Un *tipo* es un conjunto de elementos de V con la propiedad de ser un *ideal* [2], es decir, un tipo es una clasificación de los valores.

El polimorfismo es una característica que estuvo ausente en los sistemas de tipos de los primeros lenguajes de programación. En lenguajes como PASCAL y ALGOL las funciones y procedimientos tienen definiciones para sus operandos con tipo único. Estos lenguajes se llaman *monomorfos* en el sentido que los valores y variables pueden pertenecer a solo un tipo. Los lenguajes de programación monomorfos pueden contrastarse con los lenguajes polimorfos en los cuales ciertas variables pueden tomar valores en distintos tipos.

El polimorfismo permite programar procedimientos con argumentos que no tienen un tipo declarado estáticamente. El polimorfismo es *paramétrico* cuando un procedimiento trabaja sobre un rango de tipos de manera uniforme. Generalmente los tipos en el rango tienen alguna característica en común. La uniformidad se logra mediante los parámetros de tipo. Las funciones que exhiben polimorfismo parámetrico son llamadas *funciones genéricas* en los lenguajes como ADA. Cabe destacar que en procedimientos con tipos paramétricos se intenta usar una misma implementación para todos los tipos en el rango determinado. De lo contrario, solo se trataría de una definición por sobrecarga del procedimiento.

El polimorfismo *por inclusión* no trata con especificaciones parciales sino que se define para los tipos vinculados por la relación de subtipo. Esta relación es isomorfa a la relación de inclusión de conjuntos. Con la idea de que tipos son conjuntos de valores, la inclusión nos permite definir una relación de orden parcial. Todo objeto de un subtipo puede ser usado en el contexto de cualquier supertipo que lo contenga.

Cuál es el rol de un sistema de tipos en un lenguaje de programación en lógica? La evolución de la programación lógica prescindió de los tipos en sus orígenes. Esta ausencia confiere a la variable lógica una "voracidad" exagerada que, como veremos, es una desventaja. La semántica de un lenguaje nos permite definir, independientemente del programador y de la implementación, el significado de un programa. La tarea del programador es escribir un programa con un significado, el cual debe ser equivalente al significado "deseado". Entonces, dada una implementación correcta del lenguaje, la ejecución del programa será correcta. Informalmente el significado de un programa lógico **P** es el conjunto de cláusulas de base[1] que pueden ser deducibles, via resolución, a partir de **P**. El *significado deseado* de un programa **P** es el conjunto de cláusulas de base que el programador entiende que deben ser válidas. La clave de la programación lógica es definir qué átomos son verdaderos tal que el significado del programa sea correcto y completo con respecto al significado deseado [9].

Como conclusión el programador debería comenzar su trabajo con una especificación y desarrollar un programa que sea correcto y completo con respecto a dicha especificación. Las especificaciones tienen definiciones de tipos explícitas que en los programas están generalmente ausentes por razones de eficiencia. Luego podemos obtener programas que no son correctos con respecto al significado deseado, es decir, en el conjunto de cláusulas de base que se deducen del programa existen algunas que no son válidas para la especificación.

EJEMPLO **2.1** *Utilizaremos la definición canónica del predicado* `concatenar` *para ejemplificar lo recién expuesto. La especificación del predicado* `concatenar` *es:*

> `concatenar(Xs,Ys,Zs)` ← *es verdadero sí y sólo sí* `Zs` *es la lista* `Ys` *concatenada a la lista* `Xs`.

Un programa para esta especificación es [9]:

```
concatenar([],Xs,Xs).
concatenar([X|Xs],Ys,[X|Zs]) ← concatenar(Xs,Ys,Zs).
```

Si examinamos con detalle este ejemplo notaremos la verdadera relación entre la especificación y el programa. La especificación de `concatenar` *establece explícitamente que los argumentos son listas. Sin embargo una consulta con el segundo y tercer argumento distinto a listas será válida. La consulta* `concatenar([],a,a)` *será exitosa. Luego el programa no es equivalente a la especificación del predicado.*

[1]Una cláusula de base (ground) es una cláusula sin variables.

En este ejemplo notamos una de las desventajas más importantes de la ausencia de tipos en la programación lógica. Partiendo de la ecuación Programa = Especificación se concluye que Programa + Tipos = Especificación. A lo largo de este trabajo veremos cómo los tipos pueden controlar la denotación de un predicado y obtener así programas cuyo significado sea correcto con respecto al deseado. Siguiendo la propuesta de [8] podemos restringir los programas para eliminar respuestas incorrectas.

3 τ-log: sintaxis

El lenguaje lógico incluirá *definiciones de tipo* y *declaraciones de tipos.* Por *definición de tipo* entendemos la especificación del conjunto de valores que formarán un tipo bajo una denominación determinada. Una *declaración de tipos*, en cambio, se refiere a la especificación de los tipos de los argumentos de un predicado.

DEFINICIÓN **3.1** *Un* término *es una variable o un functor de aridad n aplicado a n términos* ($n \geq 0$). *Análogamente un* tipo *es una variable de tipo o una función constructora de tipos de aridad n aplicado a n tipos* ($n \geq 0$). *Un* tipo constante *es una constructora de tipos de aridad 0. Un tipo es* básico *(ground) si no contiene variables de tipo.*

EJEMPLO **3.1** *Ejemplos de tipos de aridad 0 son los predefinidos* `entero` *y* `car`, *como también* `hombre`, `mujer` *y* `naturales`. *Constructoras de tipo pueden ser* `lista`, `arbol_bin` *y* `pila`. *Estas constructoras dan lugar a los tipos no básicos* `lista(T)`, `arbol_bin(T)` *y* `pila(lista(T))`, *donde* `T` *es una variable de tipo. Tipos básicos son por ejemplo* `lista(entero)`, `arbol_bin(lista(entero))` *y* `pila(car)`. *A partir de las variables de tipo se definen los predicados polimorfos.*

DEFINICIÓN **3.2** *Un* predicado de tipo paramétrico *de aridad n es un predicado p de aridad n parametrizado por cero o más variables de tipo y se denota* $p(T_1, T_2, \ldots T_n)$, *con* $n \geq 0$. *Los predicados con distinto número de argumentos de tipo y mismo nombre son considerados diferentes. Un predicado p parametrizado por 0 variables de tipo se denota* $p()$.

3.1 Definiciones de tipo

Las *definiciones de tipo* definen el conjunto de valores de un tipo determinado. Se expresan con una gramática **BNF**, la cual se extendió con parámetros de tipo para expresar polimorfismo paramétrico.

DEFINICIÓN **3.3** *Una* definición de tipo *para una constructora de tipos c de aridad n es de la forma:*

$$c(T_1, \ldots, T_n) ::= f_1(T_1^1, \ldots, T_{n_1}^1); \ldots; f_k(T_1^k, \ldots, T_{n_k}^k), \quad (n, n_j \geq 0, k \geq 1),$$

donde el lado izquierdo de la producción es una constructora de tipos c aplicado a n variables de tipo T_i. *El lado derecho es la unión de k functores* f_j *aplicados a n tipos como argumentos. Para especificar subtipos usaremos el functor especial* `sub` *con un único argumento que será un subtipo del tipo que se está definiendo. Esto permite soportar polimorfismo por inclusión. Cabe aclarar que solo es posible utilizar* `sub` *para definir tipos abstractos que engloban un conjunto de subtipos. No es posible especificar un subtipo como la especialización de un supertipo. La definición de tipo establece que la constructora de tipos c es polimórfa en n parámetros.*

En adelante consideraremos que una definición de tipo es correcta si satisface los siguientes requerimientos:

1. Para cada constructora de tipos hay exactamente una definición de tipo.
2. Una variable de tipo que ocurre en el lado izquierdo de la producción también debe ocurrir en el lado derecho.
3. Las variables de tipo del lado izquierdo de la producción son diferentes.
4. Las constructoras de tipos pueden ser asignadas a un *ranking* bien formado.
5. Los functores del lado derecho de la definición son diferentes.
6. No es posible definir un subtipo p para una constructora c tal que p sea supertipo de c.

El requerimiento 2 significa que no hay variables de tipo locales y lo llamaremos *transparencia.* Todas las variables de tipo se asumen universalmente cuantificadas sobre todos los tipos definidos. Incluir variables de tipo locales resultaría en más complicaciones que ventajas. La restricción 4 limita las definiciónes de tipo recursivas de modo que se obtiene una clase de tipos decidible [10]:

DEFINICIÓN **3.4** *Dado un programa* **P**, *un* ranking r *es una función desde las constructoras de tipos en* **P** *a los enteros positivos. Este* ranking *está bien formado si para cada constructor de tipos c en el programa cuya definición de tipos contiene un constructor de tipos* c_i *en el lado derecho de la producción se cumple:*

- $r(c) \geq r(c_i)$, *si* c_i *es aplicado a variables de tipo.*
- $r(c) > r(c_i)$, *en todo otro caso.*

Las restricciones impuestas sobre las definiciones de tipo tienen el objetivo de poder desarrollar algoritmos para el chequeo de tipos. El lenguaje de tipos propuesto consiste en escribir una gramática regular que defina el conjunto de valores que éste alberga. Por esta razón diremos que el lenguaje de tipos soportado es *regular.* A partir de un lenguaje con estas características el chequeo de tipos consiste en *desarrollar un autómata finito reconocedor* para cada una de las definiciones.

EJEMPLO **3.2** *Una definición de tipo para el tipo polimorfo* `lista` *es:*

```
lista(T) ::= [];[T|lista(T)].
```

Esta definición está bien formada con respecto al ranking *y es paramétrica en* `T`. *Una lista de tipo* `T` *puede ser o la lista vacía o una estructura conformada por la constructora "." , donde el primer elemento es de tipo* `T` *y el segundo elemento es de tipo* `lista(T)`. *Toda instancia de un tipo paramétrico es subtipo del tipo general. Dados los siguientes tipos,*

```
mujer ::= ana; maria; hortensia.
varon ::= carlos; jose; horacio.
```

podría interesarnos crear un tipo abstracto `humano` *que albergue a ambos:*

```
humano ::= sub(mujer);sub(varon).
```

Cualquier variable que a partir de una declaración sea de tipo `humano` *podrá recibir tanto un objeto mujer como un varón. Este último es un ejemplo de polimorfismo* por inclusión.

3.2 Declaraciones de tipos

Una *declaración de tipos* establece los tipos de los argumentos de un predicado.

DEFINICIÓN **3.5** *Una declaración de tipos para un predicado* $p(T_1, \ldots, T_m)$ *de aridad* n *tiene la siguiente forma:*

$$\textit{predicado } p(T_1, \ldots, T_m)(S_1, \ldots, S_n) \quad (n, m \geq 0)$$

donde T_i *son variables de tipo y* S_j *son tipos. La declaración de tipos establece que el predicado* $p(T_1, \ldots, T_m)$ *es polimorfo en* m *parámetros y que el* j*-ésimo argumento tiene tipo* S_j*.*

EJEMPLO **3.3** *La declaración de tipos para el predicado* `concatenar` *es:*

```
predicado concatenar(T)(lista(T),lista(T),lista(T)).
```

3.3 Programas lógicos con tipos

La sintaxis de los programas lógicos con tipos paramétricos propuesta es una extensión de la sintaxis de la programación lógica tradicional, específicamente la del lenguaje PROLOG.

DEFINICIÓN **3.6** *Un* programa lógico paramétrico *es un conjunto de cláusulas:*

$$H :- B_1, B_2, \ldots, B_n. \quad (n \geq 0)$$

donde el átomo de la cabeza de la cláusula, H*, es de la forma:*

$$p(T_1, \ldots, T_k)(t_1, t_2, \ldots, t_r) \quad (k, r \geq 0)$$

donde los T_i *son diferentes variables de tipo y los* t_j *son términos. Cada* B_i *tiene la siguiente forma:*

$$q(s_1, \ldots, s_l)(t'_1, \ldots, t'_m) \quad (l, m \geq 0)$$

donde s_i *son tipos cuyas variables de tipo están contenidas en* $T_1, \ldots, T_k$ *y cada* t'_j *es un término.*

DEFINICIÓN **3.7** *Una* definición para un predicado con tipos p *es el conjunto que contiene la* declaración de tipos *para* p *y todas las cláusulas que como átomo de la cabeza tenga el predicado* p*.*

Las variables de tipo están universalmente cuantificadas sobre el universo compuesto por los tipos definidos. Esta restricción es un cambio importante con respecto a la semántica conocida del lenguaje PROLOG. Sin tipos no hay límites con respecto al valor que puede tomar una variable. Ahora los únicos valores son los incluidos en la unión de todos los tipos posibles.

EJEMPLO **3.4** *El predicado* `concatenar` *puede ser definido de la siguiente manera:*

```
predicado concatenar(T)(lista(T),lista(T),lista(T)).
concatenar(T)([],X,X).
concatenar(T)([X|Xs],Ys,[X|Zs]):- concatenar(T)(Xs,Ys,Zs).
```

donde `lista(T)` *es el tipo definido en el ejemplo 3.2.*

4 Interpretación de programas lógicos con tipos

Dada una semántica para programas lógicos con tipos, es posible construir una teoría para un lenguaje lógico sin tipos desde otra teoría equivalente con tipos [6]. Desde esta equivalencia se consideran distintas alternativas para evaluar programas lógicos:

1. Traducir los programas lógicos con tipos y la consulta al programa asociado sin tipos y luego resolver a partir de la resolución SLDNF.

2. Aplicar resolución directamente al programa con tipos y utilizar un proceso de derivación modificado para cláusulas y consultas con tipos.

3. Si el programa y la consulta satisfacen el chequeo de tipos, una alternativa es ignorar la información de tipos y resolver utlizando resolución SLDNF.

La alternativa 1 consiste en desarrollar un "compilador" que tome como entrada programas en el lenguaje fuente y genere el programa equivalente en el lenguaje lógico subyacente, en nuestro caso PROLOG. La desventaja más clara es la necesidad de realizar chequeos de tipos durante la ejecución y la imposibilidad de desarrollar un ambiente donde el sistema de tipos se integre a la respuesta. Con esta representación las respuestas serán al estilo PROLOG donde los tipos estarán ausentes.

La posibilidad 2 consiste en desarrollar un metaintérprete colocado sobre PROLOG. Esta alternativa no soluciona la necesidad de realizar el chequeo de tipos en tiempo de ejecución, pero permite integrar el sistema de tipos al entorno, lo cual será fundamental para la inferencia. La principal desventaja es que no se puede evitar el chequeo repetido de ciertas estructuras que siempre o nunca satisfacen la definiciones de tipo.

La alternativa 3 parece la más seductora pues evitaría todo chequeo de tipos durante la ejecución. Se puede probar que es correcta para polimorfismo paramétrico pero no lo es para polimorfismo por inclusión. Por las razones enunciadas nuestra elección será un compromiso entre las alternativas 1 y 2. Un metaintérprete construido sobre el lenguaje PROLOG implementará un entorno donde el sistema de tipos estará integrado. Un *parser* tomará el archivo de texto conteniendo el programa fuente y controlará la sintaxis según las definiciones de la sección 3. Al mismo tiempo cada definición de tipo, declaración o cláusula será traducida y se chequeará la consistencia de los tipos. La ventaja de utilizar un metaintérprete es que es fácilmente extensible para soportar herramientas tales como depuración, trazas, etc.

El metaintérprete cuenta con dos partes diferenciadas que son el *traductor* y el *intérprete*. El *traductor* toma cada definición de tipo, declaración de tipos de los predicados o cláusula de programa y luego de verificar la sintaxis y la consistencia de tipos lo transforma en una cláusula equivalente con una sintaxis que reconoce el *intérprete*. Para la tarea de consulta de archivos el *traductor* utiliza la facilidad provista por los intérpretes PROLOG de poder definir un espacio alternativo para almacenar cláusulas. Cada uno de estos espacios se llama *Mundo* [1]. En el mundo por defecto se ubica el metaintérprete y en el mundo *programa* se ubica la traducción del programa junto con el chequeador de tipos.

El *traductor* solo es invocado por el predicado predefinido `cargar`; en cualquier otro momento es el *intérprete* quien gobierna el entorno. El cuerpo principal consiste en el típico metaintérprete de tres lineas implementado por el predicado `resolver` con el agregado del chequeo de tipos en ejecución. El punto principal es intersectar el mecanismo de unificación al momento de seleccionar el predicado a resolver. Por último debemos dar una respuesta adecuada al entorno. Dicha respuesta se contruye con el predicado `resolver` y de la información de tipos que brindan las declaraciones. Para disponer del código del metaintérprete contactarse con los autores.

4.1 Traducción y consistencia de tipos

El trabajo del traductor consiste en traducir cada definición de tipo en un procedimiento que implemente un autómata finito que reconozca todos los valores pertenecientes al tipo, y vincular la definición de un predicado con su declaración de tipos[2]. La manera en que se realicen estas actividades influirá en nuestra definición de programa correcto con respecto a las declaraciones de tipos [10]. Una definición de tipo de la forma

$$T ::= f_1(T_1^1, \ldots, T_{n_1}^1); \ldots; f_k(T_1^k, \ldots, T_{n_k}^k)$$

es traducida a un conjunto de cláusulas de la forma

$$T(f_i(X_1, \ldots, X_{n_i}), P) : -T_1^i(X_1), \ldots, T_{n_i}^i(X_{n_i}),$$

donde $X_j \neq X_i \; \forall i, j$ y P es una lista con los parámetros de tipo. De esta forma se puede aprovechar el mecanismo de inferencia de PROLOG para la inferencia de tipos. Un caso especial es la traducción de una definición de subtipo. Este punto será tratado con detalle en la sección 4.3.

Cada cláusula de la forma `H :- B` se traduce a otra cláusula de la forma

$$\texttt{H :- T}_1\texttt{(X}_1\texttt{), T}_2\texttt{(X}_2\texttt{), ...,T}_n\texttt{(X}_n\texttt{),B.} \quad (n \geq 0).$$

Se agrega una condición de tipo $\texttt{T}_i\texttt{(X}_i\texttt{)}$ por cada variable que ocurre en `H`. Dicha condición se contruye a partir de la declaración del predicado.

El efecto de esta condición de tipos es restringir mediante el chequeo de las variables de la cabeza de la *cláusula con tipos* la denotación del predicado. Junto a esta transformación debemos indicar una definición de *programa correcto* con respecto a las declaraciones de tipos tal que se garantice que predicados inconsistentes no puedan ser válidos. Un programa **P** será consistente con las declaraciones de tipos si todas sus cláusulas de la forma `H:- B` satisfacen:

1. Si X es una variable que aparece en más de un predicado de `B`, entonces debe existir un tipo para X que contenga a todos los tipos asignados a X según la declaración de cada uno de los predicados donde ocurre.

2. Si X es una variable tal que aparece en `H` y en uno o más predicados de `B`, entonces el supertipo definido según 1 debe estar contenido en el tipo para X que determina la declaración de `H`.

3. Si X es una variable que aparece más de una vez en un predicado, entonces la intersección de los tipos obtenidos para X de la declaración correspondiente debe ser no vacía.

4. Todo valor que sea argumento de un predicado debe pertenecer al tipo asignado por la declaración.

EJEMPLO **4.1** *Dadas la siguientes declaraciones de tipos*

```
predicado p()(car,lista(entero)).
predicado q(T)(T,lista(T)).
```

[2] *Folding* entre la definición y la declaración.

donde `lista(entero)` *es un subtipo del tipo paramétrico* `lista(T)` *definido en el ejemplo 3.2. La cláusula* `p()(a,[2,3])` *es consistente con la definición pues* `a` *es un valor de tipo* `car` *y* `[1,2]` *es una estructura que se corresponde a una* `lista(entero)`*. La cláusula* `p()(X,[X,3])` *es inconsistente. Según la primer aparición,* `X` *debe ser un elemento de tipo* `car`*, de la segunda ocurrencia se infiere que* `X` *debe ser un* `entero`*; los tipos* `entero` *y* `car` *tienen intersección vacía, es decir, no existe valor que satisfaga a ambos. En este caso no se cumple la condición número 3. Veamos ahora las condiciones que involucran variables del cuerpo. La cláusula* `q(entero)(a,[X]):- p()(X,[1])` *no verifica la condición 2. Según la ocurrencia de* `X` *en el cuerpo de la cláusula debe ser* `car`*, según la cabeza debe ser* `entero`*; los enteros no contienen a los caracteres. Como el chequeo de tipos se hace antes de ejecutar el cuerpo, no hay restricciones con respecto a los valores que toman las variables del cuerpo instanciadas por las de la cabeza, pero sí debe tenerse en cuenta lo contrario pues los valores que entregan los predicados del cuerpo deben ser correctos, ya que no son chequeados.*

4.2 Chequeo e inferencia de tipos

A partir de la traducción de las definiciones de tipo tenemos un autómata finito que reconoce los valores que pertenecen al tipo que describe. La definición del tipo lista del ejemplo 3.2 se traduce a

```
lista([],[T]).
lista([X|Xs],[T]):- T(X),lista(Xs,[T]).
```

Nuestra intención es desarrollar un algoritmo para el chequeo de tipos que nos facilite el proceso de traducción y que además sea inversible. La inversibilidad es una característica de la programación lógica, que consiste en utilizar la misma especificación para calcular una función y su inversa. Lograr un predicado inversible no es siempre posible. Un punto de partida teórico para la posiblidad de implementar este módulo proviene del isomorfismo Curry-Howard [5], el cual establece un isomorfismo entre una deducción lógica y una inferencia de tipos. De esa forma es posible plantear la equivalencia entre un metaintérprete y un módulo de chequeo de tipos[4].

En nuestra implementación se utilizó la técnica *generate-and-test*, la cual consiste en programar dos módulos. En el primero se generan posibles soluciones a una consulta. En el segundo se verifica la validez de cada una de éstas. Esta metodología necesita ser refinada en términos de eficiencia, pero ofrece una representación simple y elegante. Verificar si `X` es de tipo `T` podria expresarse como:

```
c_tipos(X,T):- tipo(T),T(X).
```

El predicado `tipo` verifica que `T` sea un tipo en caso que se invoque a `c_tipos` con un valor definido. Si `T` no esta instanciada `tipo(T)`, genera un tipo entre los definidos y con `T(X)` se verifica que `X` pertenezca a `T`. Observemos que si la solución propuesta por el módulo generador no es aceptada por el módulo testeador la misma estructura de control de *backtracking* de PROLOG volverá al generador. Podemos generalizar el esquema para considerar una lista de pares de la forma `[V,T]`, donde `V` es un valor y `T` es un tipo o una variable de tipo. La definición de `c_tipos` es,

```
c_tipos([]).
c_tipos([P|Ps]):- c_tipos(P),c_tipos(Ps).
c_tipos([X,T]):- tipo(T), T(X).
```

Si consideramos que `X` puede ser un valor estructurado, por ejemplo `[[1],[2]]`, verificar `T(X)` implica una llamada recursiva a `c_tipos` para cada componente del tipo `T`. Es aquí donde se hace visible la equivalencia morfológica entre el metaintérprete básico de tres líneas y el chequeo de tipos.

El chequeo inversible de tipos partía de la hipótesis implícita de que los valores estructurados estaban completos. Consideremos nuevamente el tipo paramétrico `lista` definido en el ejemplo 3.2. Un valor completo para la instancia `lista(entero)` es `[1,2,3]` y un valor incompleto puede ser por ejemplo `[1,3,X]`. ¿Cuál es el tipo de `X` en `[1,3,X]`? Aunque no podamos inferir el valor de `X` podemos decir a qué conjunto de valores pertenece, es decir, podemos inferir desde la estructura en la que está insertado un tipo para `X`. Si incorporamos este nivel de inferencia al metaintérprete terminaremos de integrar el sistema de tipos completamente. Incorporar este nivel de inferencia nos permite también optimizar el chequeo de tipos en tiempo de compilación. Solo es necesario dejar la parte no especificada de las estructuras para chequear en la ejecución. Para implementar la inferencia de tipos para las variables se agrega un argumento al chequeador que entregue una lista de pares `[V,T]` donde `V` es una variable y `T` es el tipo inferido para `V`. La nueva cláusula utiliza el predicado predefinido `var` que es válido solo si el argumento es una variable no instanciada:

```
c_tipos([],[]).
c_tipos([P|Ps],Is):- c_tipos(P,I1),c_tipos(Ps,I2),
                     concatenar(I1,I2,Is).
c_tipos([X,T],[X,T]):- var(X).
c_tipos([X,T],Is):- tipo(T), T(X,Is).
```

4.3 Subtipos

El concepto de subtipo se puede modelar con la relación de inclusión de conjuntos. Dicha relación nos permite establecer un orden parcial sobre los tipos definidos. A partir de la definición de subtipo podemos establecer otra clase de polimorfismo denominado por inclusión. Es válido instanciar un argumento de un predicado con cualquier valor que esté incluido en el tipo declarado. Cuando se definen tipos paramétricos es fácil ver que cualquier instancia básica y no básica es un subtipo del original. Por ejemplo `lista(entero)` es un subtipo de `lista(T)`. Desde este punto de vista no existe diferencia entre polimorfismo paramétrico y polimorfismo por inclusión. Pero debemos destacar que desde el punto de vista práctico -que es el que nos interesa- es conveniente distinguir entre ambos.

Para la verificación de consistencia de tipos es necesario un predicado que calcule la intersección y la unión de los tipos. Existe una restricción al momento de definir un subtipo, dado que no es posible que un tipo sea (transitivamente) supertipo de si mismo. Esta limitación determina que la inclusión debe ser estricta. A partir de un algoritmo que verifique la ausencia de ciclos en el grafo definido por la relación subtipo podremos controlar esta restricción.

EJEMPLO **4.2** *No se permite la siguiente definición de tipos:*

```
español ::= sub(catalan);sub(vasco);sub(andaluz).
vasco ::= sub(español); sub(frances).
```

5 Ejemplos

Una definición para la estructura árbol binario podría ser la siguiente:

```
arbolb(T) ::= vacio; a(T,arbolb(T),arbolb(T)).
elemento ::= sub(entero);sub(car).
```

El tipo `arbolb(T)` se define como un nodo de tipo `T`, es decir es polimorfo en el tipo del elemento, seguido por los árboles correspondientes a los hijos izquierdo y derecho respectivamente. El tipo `elemento` es un supertipo abstracto que agrupa a enteros y caracteres. Un predicado que obtenga el recorrido en preorden de un árbol binario tendría la siguiente declaración:

```
predicado preorden(T)(arbolb(T),lista(T)).
```

es decir, dado un árbol binario cuyos nodos son de tipo `T` entrega una lista de elementos de tipo `T` que representa el recorrido en preorden del mismo. La definición del predicado sería la siguiente:

```
preorden(T)(vacio,[]).
preorden(T)(a(X,A1,A2),[X|Ns]):-
                        preorden(T)(A1,N1s),
                        preorden(T)(A2,N2s),
                        concatenar(T)(N1s,N2s,Ns).
```

Las siguientes consultas ejemplifican el chequeo para la consistencia de tipos y la inferencia de tipos en un entorno $\mathcal{T}$-LOG.

```
? preorden(entero)(a(1,a(2,vacio,vacio),vacio),X).
X = [1,2]

? preorden(T)(a(1,a(2,vacio,vacio),vacio),[1,2]).
T = entero
```

Estas consultas plantean la cuestión de cuál tipo debe inferirse según la jerarquía determinada por la relación de inclusión.

```
? preorden(entero)(a(d,a(s,vacio,vacio),vacio),[d,s]).
Inconsistencia de tipos !!!!

? preorden(T)(a(d,a(1,vacio,vacio),vacio),[d,1]).
T = elemento
```

6 Conclusiones y trabajo futuro

Los tipos han acompañado de manera constante la evolución de los lenguajes de programación. No solo son importantes a la hora de chequear la consistencia de la especificación de un programa, sino que son también una herramienta fundamental para definir interfases, optimizar código y generar compiladores. Aunque la programación lógica no se mantuvo totalmente al margen, en sus orígenes prescindió de los sistemas de tipos. Así vimos mediante algunos ejemplos, como el del predicado `concatenar`, que no siempre las especificaciones son equivalentes a los programas, existiendo una brecha entre el significado del programa y el significado deseado por el programador. El problema es que los tipos están implicitos en la especificación. Haciendo explícita esta información se logra cerrar la brecha.

Se presentó para ello el desarrollo de un lenguaje de programación lógica con definiciones de tipo polimorfas y declaración de predicados parámetricos. Se discutieron distintas técnicas para la implementación de un sistema de tipos en un intérprete PROLOG.

El uso de un metaintérprete con algunas características propias de un compilador es una manera flexible y ventajosa para realizar dicha tarea. El sistema de tipos fue integrado al entorno mediante un mecanismo de inferencia que intercepta la unificación limitando la "voracidad" de la variable lógica.

La estructura del intérprete propuesto es fácilmente extensible y adaptable a distintas sintaxis y situaciones. No es díficil refinar la granularidad del metaintérprete para, por ejemplo, realizar una traza de un programa con tipos o agregar otros predicados predefinidos que permitan introducir o retractar cláusulas. Otro desarrollo futuro interesante es potenciar la capacidad deductiva del entorno para limitar la declaración de tipos a lo estrictamente necesario. Por ejemplo, dada la siguiente especificación:

```
predicado p()(entero).
predicado q()(entero).
p()(X):- q()(X).
q()(1).
q()(2).,
```

la declaración de p podría obviarse y su tipo deducirse del contexto. Podría incluirse una variedad mayor de constructoras de tipos como por ejemplo subrangos.

Referencias

[1] Arity Corporation. The Arity/Prolog Languaje Reference Manual. 1985.

[2] G. Birkhoff y T. C. Bartee. *Modern Applied Algebra.* McGraw-Hill, 1970.

[3] Luca Cardelli y Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Reviews*, 17(4):471–522, 1985.

[4] T. Frühwirth. Using Meta-interpreters for Polymorphic Type Checking. En *Proceedings of the Second Workshoop on Meta-programming in Logic*, Maurice Bruynooghe (Editor), Leuven, Belgium, 1990.

[5] Jean-Yves Girard, Yves Lafont, y Paul Taylor. *Proofs and Types.* Cambridge University Press (Cambridge Tracts in Theoretical Computer Science 7), Cambridge, Massachussets, 1989.

[6] P. M. Hill y R. W. Topor. A Semantics for Typed Logic Programs. En Frank Pfenning, editor, *Types in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1991.

[7] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, segunda edición, 1987.

[8] L. Naish. Types and the Intended Meaning of Logic Programs. En Frank Pfenning, editor, *Types in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1991.

[9] Leon Sterling y Ehud Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, second edición, 1994.

[10] E. Yardeni, T. Frühwirth, y E. Shapiro. Polymorphically Typed Logic Programs. En Frank Pfenning, editor, *Types in Logic Programming*, The MIT Press, Cambridge,

El uso de un metaintérprete con algunas características propias de un compilador es una manera flexible y ventajosa para realizar dicha tarea. El sistema de tipos fue integrado al entorno mediante un mecanismo de inferencia que intercepta la unificación limitando la "voracidad" de la variable lógica.

La estructura del intérprete propuesto es fácilmente extensible y adaptable a distintas sintaxis y situaciones. No es díficil refinar la granularidad del metaintérprete para, por ejemplo, realizar una traza de un programa con tipos o agregar otros predicados predefinidos que permitan introducir o retractar cláusulas. Otro desarrollo futuro interesante es potenciar la capacidad deductiva del entorno para limitar la declaración de tipos a lo estrictamente necesario. Por ejemplo, dada la siguiente especificación:

```
predicado p()(entero).
predicado q()(entero).
p()(X):- q()(X).
q()(1).
q()(2).,
```

la declaración de p podría obviarse y su tipo deducirse del contexto. Podría incluirse una variedad mayor de constructoras de tipos como por ejemplo subrangos.

Referencias

[1] Arity Corporation. The Arity/Prolog Languaje Reference Manual. 1985.

[2] G. Birkhoff y T. C. Bartee. *Modern Applied Algebra.* McGraw-Hill, 1970.

[3] Luca Cardelli y Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Reviews*, 17(4):471–522, 1985.

[4] T. Frühwirth. Using Meta-interpreters for Polymorphic Type Checking. En *Proceedings of the Second Workshoop on Meta-programming in Logic*, Maurice Bruynooghe (Editor), Leuven, Belgium, 1990.

[5] Jean-Yves Girard, Yves Lafont, y Paul Taylor. *Proofs and Types.* Cambridge University Press (Cambridge Tracts in Theoretical Computer Science 7), Cambridge, Massachussets, 1989.

[6] P. M. Hill y R. W. Topor. A Semantics for Typed Logic Programs. En Frank Pfenning, editor, *Types in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1991.

[7] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, segunda edición, 1987.

[8] L. Naish. Types and the Intended Meaning of Logic Programs. En Frank Pfenning, editor, *Types in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1991.

[9] Leon Sterling y Ehud Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, second edición, 1994.

[10] E. Yardeni, T. Frühwirth, y E. Shapiro. Polymorphically Typed Logic Programs. En Frank Pfenning, editor, *Types in Logic Programming*, The MIT Press, Cambridge,