Tipos Algebraicos de Datos en Fork Álgebras

Pablo E. Martínez López Gabriel A. Baum

LIFIA, Departamento de Informática, Universidad Nacional de La Plata.

C.C.11, Correo Central, 1900, La Plata, Buenos Aires, República Argentina.

E-mail: {fidel,gbaum}@info.unlp.edu.ar

URL: http://www-lifia.info.unlp.edu.ar/

Resumen

La *síntesis de programas* es un proceso que permite obtener un programa eficiente a partir de una especificación, preservando su significado. Las Fork Álgebras han sido propuestas como base algebraica para la construcción de un ambiente de síntesis de programas. Los tipos algebraicos de datos (generados por un conjunto de *constructores*) son una herramienta poderosa que permite modularizar los programas y mejorar la legibilidad y la detección de errores en tiempo de compilación. Hasta ahora, los tipos de datos en fork álgebras han sido manejados de manera intuitiva, sin precisar formalmente su significado.

El objetivo de este artículo es proveer un mecanismo general de definición de tipos algebraicos de datos en el marco de las fork álgebras, y establecer una sintaxis que permita simplificar la notación al realizar las especificaciones.

Palabras clave: Síntesis de programas, Fork Álgebras, Tipos Algebraicos

1 Introducción

La *síntesis de programas* es el proceso que consiste en comenzar con una especificación evidentemente correcta, aunque tal vez ineficiente, de un problema, y a través de la aplicación de reglas de transformación que preserven el significado, obtener un programa eficiente; esta técnica es tema de estudio de la llamada programación transformacional ([11]). El proceso descrito tiene una extraordinaria similitud con el proceso de resolución de ecuaciones que se estudia en *aritmética*, pero donde en lugar de números, los objetos que se manipulan son programas; ello sugiere que podría utilizarse un marco algebraico como base para el desarrollo de un ambiente de síntesis de programas ([12]). En particular, nuestro interés se centra en las llamadas Fork Álgebras, que son álgebras relacionales extendidas con un nuevo operador, *fork*, que da origen a su nombre ([9, 8]). Estas álgebras poseen el poder expresivo suficiente para servir como base del proceso de síntesis, y además son una correcta abstracción del modelo conjuntista que considera a los programas como una relación de entrada–salida de datos; por ello han sido propuestas como el marco algebraico mencionado. La potencia de este cálculo para razonar acerca de programas ha sido ilustrada mediante gran cantidad de ejemplos ([8, 1, 6], entre otros).

Una parte importante del proceso de construcción de programas es la correcta elección y tratamiento de los tipos de datos; ellos permiten modularizar los programas y mejorar la legibilidad y la detección de errores en tiempo de compilación, entre otras ventajas. La idea básica de la especificación algebraica de tipos de datos consiste en describirlos mediante los nombres de los diferentes conjuntos de datos, los nombres de sus funciones básicas, y las propiedades características de ellas ([14]). Hasta ahora, los tipos de datos en el marco de las fork álgebras se han utilizado asumiendo un conocimiento a priori, intuitivo, de ellos, y considerando conocidas las propiedades de los mismos. Es de fundamental importancia, por lo tanto, contar con un mecanismo riguroso de definición y manipulación de tipos de datos.

Este trabajo se centra en la definición y uso de tipos de datos algebraicos en el marco de las fork álgebras. Por tipo algebraico de datos entendemos al álgebra libre generada por un conjunto de *funciones constructoras* ([14, 4]), o sea que todos los elementos del tipo pueden ser construídos mediante dichas funciones y que cada combinación sintácticamente correcta de constructores genera un elemento distinto del tipo (no hay "casos

inválidos"). También se los llama tipos inductivos, por ser definibles mediante *inducción estructural* ([3, 5]).

El método elegido para especificar tipos de datos es introducir nuevas constantes de relaciones, junto con axiomas que las mismas deben cumplir para expresar las operaciones requeridas. Además estableceremos la sintaxis de declaraciones que nos permitan abreviar la axiomatización.

En la Secc. 2 presentamos la definición de Fork Algebras, y describimos como se las usa en la construcción de programas. En la Secc. 3 presentamos simultáneamente la sintaxis de las declaraciones y la especificación de tipos algebraicos, y ofrecemos algunos ejemplos. El artículo finaliza con las conclusiones.

2 Fork Álgebras

Un programa secuencial puede ser entendido como un proceso que transforma ciertos datos de entrada en ciertos datos de salida. Una representación matemática que captura esta noción es la de relación binaria.

El concepto de álgebra de relaciones binarias fue introducido por Tarski ([13]) para dar una versión algebraica de la lógica de primer orden, en el mismo sentido que las álgebras de Boole lo hacen con el cálculo proposicional. Dado un conjunto $\mathcal{U}$, se consideran las relaciones incluídas en $\mathcal{U}\times\mathcal{U}$, y sobre ellas se define un álgebra de Boole (con las operaciones habituales de unión, intersección y complemento entre relaciones, y las constantes $\mathcal{U}\times\mathcal{U}$ y $\emptyset$ como máximo y mínimo), extendiéndola con las operaciones de composición e inversa (a veces llamada también *conversa*). Sobre esta base, Tarski definió las álgebras relacionales ([13]), una axiomatización que caracteriza abstractamente a las álgebras de relaciones. Desafortunadamente el objetivo de Tarski no se cumplió, pues las álgebras relacionales sólo tienen el poder expresivo de una lógica de primer orden con tres variables y sólo dos de ellas libres ([13]). En consecuencia, esta álgebras no son útiles como base para un formalismo que permita especificar problemas y programas.

Una solución para el problema de expresividad son las Fork Algebras ([9, 8]), que son una extensión de las álgebras relacionales con un operador llamado *fork* (al que denotaremos con ∇). Esta operación captura la noción de formación de pares y consecuentemente de estructuras complejas (de forma arborescente); la interpretación conjuntista de la relación $\mathsf{R}\nabla\mathsf{S}$ será $\{\langle x,[y,z]\rangle / \langle x,y\rangle \in \mathsf{R} \ \wedge \ \langle x,z\rangle \in \mathsf{S}\ \}$. El fork es el punto de partida de una ganancia de expresividad que se ha demostrado equivalente a la de la lógica de

primer orden con igualdad ([9]). Por otra parte se ha demostrado que las fork álgebras son representables ([7]), es decir, todos sus modelos son isomorfos; al modelo intuitivo conjuntista lo llamaremos modelo estándar.

Definición 2.1 *Una* Fork Álgebra (Abstracta) *es una estructura* $< A, +, \bullet, ;, \nabla, ^-, ^\smile, 0, \infty, 1 >$, *tal que:*

1. $< A, +, \bullet, ^-, 0, \infty >$ *es un álgebra de Boole, completa y atómica, donde* $+$ *denota la operación supremo,* $\bullet$ *denota la operación ínfimo,* $^-$ *denota la operación complemento,* 0 *denota el elemento* cero, ∞ *denota el elemento* unidad, *y* $\preceq$ *(llamada* inclusión relacional*) denota el orden subyacente*

2. $< A, ;, 1 >$ *es un semigrupo con la composición relacional* $(;)$ *y la relación identidad* (1), *es decir,* $(R;S);T = R;(S;T)$ *y* $R;1 = 1;R = R$

3. $R;S \subset T \Leftrightarrow R^\smile;\overline{T} \subset \overline{S} \Leftrightarrow \overline{T};S^\smile \subset \overline{R}$ *(regla de Schröder)*

4. $R \neq 0 \Rightarrow \infty;R;\infty = \infty$ *(regla de Tarski)*

   La operación ∇ (fork) *queda totalmente caracterizada por*

5. $R \nabla S = (R;(1 \nabla \infty)) \bullet (S;(\infty \nabla 1))$

6. $(R \nabla S);(T \nabla Q)^\smile = (R;T^\smile) \bullet (S;Q^\smile)$

7. $(1 \nabla \infty)^\smile \nabla (\infty \nabla 1)^\smile \preceq 1$ ■

Un término del álgebra puede ser considerado como la versión abstracta de una relación binaria que represente la relación de entrada-salida de datos que establece la especificación de un programa. Con esta interpretación, los operadores del álgebra son formas de combinar programas; así, la composición relacional $(;)$ representa la composición secuencial de dos programas, el supremo $(+$, o suma relacional) representa a la elección no-determinística entre dos programas, etcétera. Además, mediante ecuaciones e inecuaciones entre términos del álgebra pueden expresarse propiedades de los programas, como por ejemplo:

Definición 2.2 *Una relación* P *se dice* funcional *si, y sólo si,* $P^\smile;P \preceq 1$, inyectiva *si, y sólo si,* $P;P^\smile \preceq 1$, *y* constante *si, y sólo si,* $P \neq 0$, $P = \infty;P$, *y* $P^\smile;P \preceq 1$ ■

Las nociones de relación funcional e inyectiva se corresponden con las nociones de funcionalidad e inyectividad estándar. La noción de relación constante representa a la de un programa que acepta cualquier entrada y retorna siempre el mismo valor constante.

Un punto de importancia al utilizar relaciones para representar programas es la forma en que se representan conjuntos. Existen varias maneras de realizar la "internalización" de un conjunto; la usada en este artículo consiste en representar al conjunto X mediante un término incluído en 1, 1_X, denominado identidad parcial, y cuya versión estándar es la relación $\{< x, x >/\ x \in X\ \}$.

Determinados términos aparecen repetidos numerosas veces, por lo que es conveniente definir abreviaturas para ellos, tanto para simplificar la notación como para ofrecer una mnemotecnia sobre su interpretación en el modelo estándar; esta interpretación se enuncia luego de la definición.

Definición 2.3 *Sean las siguientes operaciones definidas por*

1. $\pi = (1 \nabla \infty)^{\smile}$ *y* $\rho = (\infty \nabla 1)^{\smile}$ (primera y segunda proyección)
2. $\mathsf{R} \otimes \mathsf{S} = (\pi ; \mathsf{R}) \nabla (\rho ; \mathsf{S})$ (producto)
3. $2 = 1 \nabla 1$ (duplicación de datos)
4. $Dom(\mathsf{R}) = (\mathsf{R} ; \mathsf{R}^{\smile}) \bullet 1$ *y* $Ran(\mathsf{R}) = (\mathsf{R}^{\smile} ; \mathsf{R}) \bullet 1$ (dominio y rango de R)
5. $\sum_{i=1}^{n} \mathsf{R}_i = \mathsf{R}_1 + \ldots + \mathsf{R}_n$, $(n \geq 1)$ (suma finita)
6. $\bigotimes_{i=1}^{n} \mathsf{R}_i = (\mathsf{R}_1 \otimes (\mathsf{R}_2 \otimes \ldots (\mathsf{R}_{n-1} \otimes \mathsf{R}_n) \ldots))$, $(n \geq 1)$ (producto finito) ■

Las operaciones π y ρ representan las relaciones de proyección respecto de la operación de formación de pares. Dado que la relación 1 representa a la relación identidad, la relación 2 representa a la operación de duplicación de sus datos de entrada, y $2^{\smile}$ resulta ser un filtro de igualdad sobre los pares, tal que sus componentes son iguales. $Dom(\mathsf{R})$ y $Ran(\mathsf{R})$ son expresiones relacionales que caracterizan el *dominio* y el *rango* de la relación R.

A modo de ejemplo mostraremos como se expresan dos programas sencillos. Si contamos con un término relacional abstracto llamado mult que represente al programa que dados dos números naturales devuelve

el resultado de multiplicarlos (mult sería la versión abstracta de $\{< [x,y], x \times y >/\ x, y \in Nat\ \}$) podemos construir un término que represente al programa que dado un número lo eleva al cuadrado de la siguiente forma: $\mathsf{sqr} = (2; \mathsf{mult})$. El uso de la relación 2 es la manera abstracta de expresar la replicación del dato de entrada, para poder multiplicarlo luego por sí mismo. Para escribir un programa que dado un par de valores devuelve el mismo par, pero con los valores cambiados, escribiremos $(\rho \nabla \pi)$, y lo llamaremos swap.

A fin de poder comparar dos especificaciones de tipos de datos en un marco relacional, debemos contar con la noción de isomorfismo entre relaciones. Las definiciones dadas están adaptadas de las que aparecen en [2].

Definición 2.4 *Sean* R *y* S *dos relaciones. Un par de relaciones* (chgDom , chgRan) *es un* homomorfismo *de* R *a* S *sii* chgDom *y* chgRan *son relaciones funcionales, y se cumple que* $\mathsf{R}; \mathsf{chgRan} \preceq \mathsf{chgDom}; \mathsf{S}$, $Dom(\mathsf{R}) \preceq Dom(\mathsf{chgDom})$, *y* $Ran(\mathsf{R}) \preceq Dom(\mathsf{chgRan})$. ■

Considerando las relaciones como programas (en el modelo estándar), un homomorfismo transforma cada elemento del dominio y del rango del programa R, de manera tal que R se comporte como el programa S.

Definición 2.5 *Sean* R *y* S *dos relaciones. Un par de relaciones* (chgDom , chgRan) *es un* isomorfismo *entre* R *y* S *sii* (chgDom , chgRan) *es un homomorfismo de* R *a* S, *y* $(\mathsf{chgDom}^{\smile}, \mathsf{chgRan}^{\smile})$ *es un homomorfismo de* S *a* R. ■

Un isomorfismo transforma cada elemento del dominio y del rango del programa R mediante relaciones uno a uno, de manera tal que R se comporte *exactamente* como el programa S.

La idea de homomorfismo (isomorfismo) se puede extender a tuplas de relaciones de la siguiente manera:

Definición 2.6 *Dadas dos tuplas de relaciones* $(\mathsf{R}_1, \ldots, \mathsf{R}_n)$ *y* $(\mathsf{S}_1, \ldots, \mathsf{S}_n)$, *diremos que son homomorfas (isomorfas) sii para todo* $1 \leq i \geq n$, *existe un homomorfismo (isomorfismo) entre* $(\mathsf{R}_i, \mathsf{S}_i)$. ■

3 Tipos Algebraicos de Datos

Un tipo algebraico de datos es el álgebra libre generada por un conjunto de *funciones constructoras* ([14, 4]), es decir, todos los elementos del tipo pueden ser construídos mediante dichas funciones y, además, cada

combinación sintácticamente correcta de constructores genera un elemento distinto del tipo. También se los conoce con el nombre de tipos inductivos, por ser definibles mediante *inducción estructural* ([3, 5]).

El método utilizado para especificar tipos de datos en el marco de las fork álgebras es introducir nuevas constantes de relaciones, junto con axiomas que las mismas deben cumplir para expresar las operaciones constructoras del tipo en cuestión. Para dar la sintaxis de las declaraciones definiremos la noción de *término de tipo*; estos términos servirán para denotar sintácticamente a los tipos. Para su definición asumiremos que *Type* es un conjunto de nombres de tipos, y que *aridad* es una función que para cada elemento de *Type* retorna su aridad (≥ 0).

Definición 3.1 *Sea Type el conjunto de nombres de tipos y sea V un conjunto de variables. Llamaremos* término de tipo con variables en *V a un elemento del siguiente conjunto inductivo:*

- $v \in V$ *es un término de tipo con variables en V*
- *si* $T \in Type$, *y* $t_1, \ldots, t_p$ *son términos de tipo con variables en V, entonces* $T(t_1, \ldots, t_p)$ *es un término de tipo con variables en V.* ■

Usaremos la siguiente sintaxis para escribir declaraciones:

$$NewType(a_1, \ldots, a_p) = < C_1 : D_1, \ldots, C_n : D_n >$$

con $NewType \in Type$, $aridad(NewType) = p$, $n \geq 1$, $a_1, \ldots, a_p$ variables de tipo, $C_1, \ldots, C_n$ nombres de relaciones, y donde se cumple que para todo $1 \leq i \geq n$, $D_i = \mathbf{Cte}$, o bien $D_i = (d_1, \ldots, d_r)$, con $r \geq 1$, y para todo $1 \leq k \geq r$, d_k es un término de tipo con variables en $\{a_1, \ldots, a_p\}$. El símbolo especial **Cte** sirve para declarar a una relación como constante, como se puede ver en la Def. 3.2; no debe pertenecer al conjunto *Type*, para evitar ambigüedad en la definición.

Cada instancia sin variables de esta declaración se denomina tipo algebraico, y será la abreviatura de:

Definición 3.2 *El tipo algebraico* $NewType(t_1, \ldots, t_p)$, *donde* $t_1, \ldots, t_p$ *son términos de tipo sin variables, se define como una tupla de relaciones* $(C_1, \ldots, C_n)$ *tal que los siguientes axiomas se satisfacen:*

1. *para todo* $1 \leq i \geq n$

- *si* $D_i = \mathbf{Cte}$, *entonces* C_i *es una relación constante*
- *si* $D_i = (d_1,\ldots,d_r)$, *entonces* C_i *es funcional e inyectiva, y* $Dom(C_i) = \left(\bigotimes_{k=1}^{r} 1_{d'_k}\right)$, *siendo* d'_k *el resultado de sustituir cada* a_h *en* d_k *por* t_h, *con* $1 \leq h \geq p$

2. *para todo* $1 \leq i,j \geq n$, *con* $i \neq j$, $C_i; C_j^{\smile} = 0$, *o sea, los rangos de constructores distintos son disjuntos*

3. *se cumple que* $(\sum_{i=1}^{n} Ran(C_i)) = \mathsf{Destr}(NewType(a_1,\ldots,a_p))$, *donde* Destr *queda definida como la mínima solución del siguiente sistema de ecuaciones recursivas:*

$\mathsf{Destr}(\mathsf{term})$

$= t_h$, si $\mathsf{term} = a_h$, con $1 \leq h \geq p$

$= \sum_{j=1}^{m} C_j'^{\smile}; Md(D_j'); C_j'$, si $\mathsf{term} = OldType(t'_1,\ldots,t'_q)$, y existe la declaración

$OldType(a'_1,\ldots,a'_q) = < C_1' : D_1', \ldots, C_m' : D_m' >$

donde $Md(D_j')$

$= 1$, si $D_j' = \mathbf{Cte}$

$= \mathsf{Destr}(d_1'')$, si $D_j' = d_1'$, y d_1'' es el resultado de sustituir cada a'_h en d_1' por t'_h, con $1 \leq h \geq q$

$= \bigotimes_{k=1}^{s} Md(d_k')$, si $D_j' = (d_1',\ldots,d_s')$, con s ≥ 2

Si este sistema tiene solución no trivial, entonces el tipo es finitamente generado.

Además, se define, para todo $1 \leq i \geq n$, 1_{C_i} *como abreviatura de* $Ran(C_i)$, *y* $1_{NewType(t_1,\ldots,t_p)}$ *como abreviatura de* $\left(\sum_{i=1}^{n} 1_{C_i}\right)$. ■

Utilizando la inyectividad de C_i y el hecho de que las funciones constructoras tienen rangos disjuntos es fácil demostrar que $\left(1_{C_i} \bullet 1_{C_j}\right) = 0$, si $i \neq j$; por lo tanto, la definición de $1_{NewType(t_1,\ldots,t_p)}$ provee una partición de $NewType(t_1,\ldots,t_p)$, lo cual es de gran utilidad para aplicar la técnica de análisis de casos en la especificación y construcción formal de programas ([9, 1])

Esta especificación puede tener muchos modelos, pero la proposición que se presenta a continuación establece que todos ellos son equivalentes, y por lo tanto resulta indistinto con cual de ellos trabajemos.

Proposición 3.3 *Si* $t_1,\ldots,t_p$ *son términos de tipo sin variables que denotan tipos monomórficos (todos sus modelos son isomorfos), entonces,* $NewType(t_1,\ldots,t_p)$ *denota un tipo monomórfico.*

Dem. Sean $NewType_1(t_1,\ldots,t_p) = (\mathsf{R}_1,\ldots,\mathsf{R}_n)$ y $NewType_2(t_1,\ldots,t_p) = (\mathsf{S}_1,\ldots,\mathsf{S}_n)$, dos modelos distintos del tipo algebraico $NewType(t_1,\ldots,t_p)$. Se construye una relación transf, de manera muy similar al sistema de ecuaciones recursivas del axioma 3; la diferencia fundamental es el reemplazo de $\mathsf{C}_j'^{\smile}$ y C_j' por $\mathsf{R}_j{}^{\smile}$ y S_j en la definición de Destr. La relación transf realiza la transformación de una representación a la otra. Se prueba que $(M(\mathsf{D}_i), \mathsf{transf})$ es un isomorfismo entre R_i y S_i, para todo $1 \leq i \geq n$, donde M prepara un término basado en transf que se corresponde con la forma de D_i. Por lo tanto, $NewType_1(t_1,\ldots,t_p)$ y $NewType_2(t_1,\ldots,t_p)$ son isomorfos. Los detalles de la demostración pueden encontrarse en [10]. ■

3.1 Ejemplos

El caso más trivial es $Unit = <$ unit : Cte $>$, donde p=0, n=1. El axioma 1 establece que unit es una relación constante; el axioma 2 no se aplica, pues n=1; el axioma 3 se satisface trivialmente, ya que este es un tipo básico (no aparece en el dominio de sus constructores), y entonces queda

$\mathsf{Destr}(Unit) = \mathsf{unit}^{\smile}; Md(\mathbf{Cte}); \mathsf{unit} = \mathsf{unit}^{\smile}; \mathsf{unit} = 1_{Unit}$.

Otro tipo, muy utilizado en los lenguajes de programación, es el tipo $Bool = <$ false : Cte, true : Cte $>$. El axioma 2 establece que verdadero y falso son dos elementos distintos. La construcción del tipo *Bool* puede generalizarse para otros tipos enumerativos; por ejemplo, $Color = <$ red : Cte, yellow : Cte, green : Cte $>$, es un tipo con tres constantes distintas (axiomas 1 y 2); el axioma 3 se satisface trivialmente, igual que para todos los tipos compuestos sólo por constantes (ver el ejemplo de *Unit*).

Todos los tipos presentados hasta ahora tienen p=0, o sea que no dependen de ningún argumento. Para mostrar el uso de variables de tipo como argumento en la definición, utilizaremos un tipo que representa a la unión disjunta de tipos: $Either(A,B) = <$ left : A, right : $B >$. Este uso de las variables de tipo permite utilizar el mecanismo conocido como *polimorfismo paramétrico*; por ejemplo, podemos escribir $Either(Bool,Color)$, o $Either(Unit,Bool)$, etc. Para cada tipo $Either(t_1,t_2)$, el axioma 1 establece que left y right son relaciones funcionales e inyectivas, y que $Dom(\mathsf{left}) = 1_{t_1}$, y $Dom(\mathsf{right}) = 1_{t_2}$; el axioma 2 establece que el rango de left y el de right son disjuntos (aunque t_1 sea igual a t_2); el axioma 3 se satisface trivialmente, $\mathsf{Destr}(Either(A,B)) = \mathsf{left}^{\smile}; Md(A); \mathsf{left} + \mathsf{right}^{\smile}; Md(B); \mathsf{right} = \mathsf{left}^{\smile}; 1_{t_1}; \mathsf{left} + \mathsf{right}^{\smile}; 1_{t_2}; \mathsf{right} = \mathsf{left}^{\smile}; \mathsf{left} + \mathsf{right}^{\smile}; \mathsf{right} = 1_{Either(t_1,t_2)}$.

Los ejemplos siguientes muestran tipos con casos inductivos no triviales, los cuales son comunmente utilizados en los lenguajes de programación.

$Num = <$ cero : **Cte**, suc : $Num >$

$List(A) = <$ nil : **Cte**, cons : $(A, List(A)) >$

El primero de ellos, el tipo de los números naturales, no utiliza variables en su definición. Los axiomas 1 y 2 son fáciles de interpretar; el axioma 3 establece que los elementos del tipo *Num* son generados finitamente (se obtienen por la aplicación de una cantidad finita de operaciones suc a la operación cero):

$\mathsf{Destr}(Num) = \mathsf{cero}^{\smile}; \mathsf{cero} + \mathsf{suc}^{\smile}; \mathsf{Destr}(Num); \mathsf{suc}.$

El segundo, el tipo de las listas, utiliza las variables de tipos, permitiendo el polimorfismo paramétrico, de la forma en que se vió para el tipo *Either*(*A*,*B*). Es de interés observar que, para el tipo *List*(t), el axioma 1 establece que $Dom(\mathsf{cons}) = \left(1_t \otimes 1_{List(t)}\right)$, y el axioma 3 establece que

$\mathsf{Destr}(List(A)) = \mathsf{nil}^{\smile}; \mathsf{nil} + \mathsf{cons}^{\smile}; \left(1_t \otimes \mathsf{Destr}(List(A))\right); \mathsf{cons},$

El siguiente ejemplo muestra la potencia del axioma 3 para establecer la generación finita, ya que la referencia inductiva aparece dentro de un término basado en *List*: $GenTree(A) = <$ node : $(A, List(GenTree(A))) >$. El axioma 3 establece un sistema de ecuaciones recursivas:

$\mathsf{Destr}(GenTree(A)) = \mathsf{node}^{\smile}; \mathsf{Destr}(List(GenTree(A))); \mathsf{node}$, y

$\mathsf{Destr}(List(GenTree(A))) = \mathsf{nil}^{\smile}; \mathsf{nil} + \mathsf{cons}^{\smile}; (\mathsf{Destr}(GenTree(A)) \otimes \mathsf{Destr}(List(GenTree(A)))); \mathsf{cons};$

la interacción entre las ecuaciones queda determinada por los términos de tipo utilizados en la declaración como dominio para las operaciones constructoras.

El poder del tercer axioma también puede verse en el caso de definiciones recursivas simultáneas,

$PList(A) = <$ vacia : **Cte**, consp : $(A, IList(A)) >$, y

$IList(B) = <$ consi : $(B, PList(B)) >$.

El sistema de ecuaciones recursivas para estas declaraciones es

$\mathsf{Destr}(PList(A)) = \mathsf{vacia}^{\smile}; \mathsf{vacia} + \mathsf{consp}^{\smile}; \mathsf{Destr}(IList(A)); \mathsf{consp}$, y

$\mathsf{Destr}(IList(A)) = \mathsf{consi}^{\smile}; \mathsf{Destr}(PList(A)); \mathsf{consi}.$

4 Conclusiones

Hemos presentado un mecanismo de definición de tipos algebraicos en el marco de las fork álgebras, junto con una sintaxis que permite simplificar la notación y que permite la definición de las internalizaciones de los tipos en base a las operaciones constructoras. Hemos mostrado que es posible especificar con este método varios tipos que aparecen comunmente en los lenguajes de programación.

Una conclusión de importancia es que la utilización de tipos de datos en un marco homogéneo es posible, y no es necesario hacerlo de manera intuitiva. Esta técnica provee un aporte de gran importancia a la aplicación de las fork álgebras en las ciencias de la computación, fundamentalmente en las áreas de derivación de programas ([1]) y de bases de datos, especialmente en el campo de optimización de consultas ([6]), proveyendo un método riguroso de especificar los tipos de datos y sus propiedades. Las especificaciones presentadas poseen varias características importantes para la construcción formal de programas: permiten una definición jerárquica de los tipos, especifican tipos monomórficos, permiten el uso del polimorfismo paramétrico, y establecen una partición del dominio del tipo, favoreciendo el análisis de casos, entre otras cosas. Además es interesante observar el uso de la operación conversa ($^\smile$) como medio de proveer un mecanismo de *pattern matching* sobre los constructores de los tipos algebraicos.

Una posible continuación de este trabajo es la extensión del método de especificación para otras clases de tipos (no algebraicos), como ser, tipos obtenidos como cociente de tipos algebraicos, tipos funcionales, tipos con restricciones en la formación de sus elementos, etc.

Referencias

[1] Gabriel A. Baum, Marcelo F. Frias, Armando M. Haeberer, and Pablo E. Martínez López. Construcción formal de programas con fork álgebras. In *24 Jornadas Argentinas de Informática e Investigación Operativa (JAIIO)*, Agosto 1995. URL: "ftp://www-lifia.info.unlp.edu.ar/pub/papers/fork-algebras/".

[2] Rudolph Berghammer, Armando M. Haeberer, Gunther Schmidt, and Paulo A. S. Veloso. Comparing two different approaches to products in abstract relation algebras. In *Proceedings of the Third International Conference on Algebraic Methodology and Software Technology, AMAST '93*, pages 167–176. Springer–Verlag, 1993.

[3] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* C. A. R. HOARE. Prentice Hall, 1988.

[4] A. Davie. *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press, 1992.

[5] Peter A. Fejer and Dan A. Simovici. *Mathematical Foundations of Computer Science*, volume I: Sets, Relations and Induction. Springer–Verlag, 1991.

[6] Marcelo F. Frias and Silvia Gordillo. Semantic optimization of queries to deductive object oriented database. In *Proceedings of the Second International Workshop on Advances in Databases and Information Systems, ADBIS'95*, pages 37–45, Moscow, June 1995. Springer-Verlag.

[7] Marcelo Fabián Frias, Gabriel A. Baum, Armando M. Haeberer, and P. A. S. Veloso. Fork algebras are representable. In *Bulletin of the Section of Logic, Vol.24, N.2*, pages 64–75, University of Lódź, June 1995.

[8] Armando M. Haeberer, Gabriel A. Baum, and Gunther Schmidt. On the smooth calculation of relational recursive expressions out of first–order non–constructive specifications involving quantifiers. In *Proceedings of the International Conference on Formal Methods in Programming and Their Applications, LNCS 735*, pages 281–298, 1993.

[9] Armando M. Haeberer and P. A. S. Veloso. Partial relations for program derivation: Adequacy, inevitability and expressiveness. In *Constructing Programs from Specifications – Proceedings of the IFIP TC2 Working Conference on Constructing Programs from Specifications*, pages 319–371, North Holland, 1991. IFIP WG. 2.1.

[10] Pablo E. Martínez López and Gabriel A. Baum. Especificación de tipos algebraicos de datos en el marco de las fork álgebras. Technical report, LIFIA, Universidad Nacional de La Plata, Febrero 1996.

[11] Helmut A. Partsch. *Specification an Transformation of Programs: A Formal Approach to Software Development.* Texts and Monographs in Computer Science. Springer–Verlag, 1990.

[12] A.J. Sampaio, Armando M. Haeberer, T. Prates, C.D. Ururahy, Marcelo F. Frias, and N.C. Albuquerque. PLATO: A tool to assist programming as term rewriting and theorem proving. In *Proceedings of the Sixth International Conference on the Theory and Practice of Software Development*, Denmark, May 1995.

[13] Alfred Tarski. On the calculus of relations. In *Journal of Symbolic Logic, vol. 6*, pages 73–89, 1941.

[14] Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier Science Publishers, 1990.