LA ABSTRACCIÓN DE DATOS Y SU PROCESO GRADUAL DE CONSTRUCCIÓN

THE GRADUAL PROCESS OF DATA ABSTRACTION

FRANCISCO MORENO
Profesor Departamento de Ingeniería de Sistemas e Informática, Universidad Nacional Sede Medellín

JAIME ECHEVERRI
Profesor Departamento de Ingeniería de Sistemas, Universidad de Medellín, jaecheverri@udem.edu.co

ROBERTO FLÓREZ
Profesor Departamento de Ingeniería de Sistemas, Universidad de Antioquia, rflorez@udea.edu.co

Recibido para revisar Junio 21 de 2007, aceptado Agosto 31 de 2007, versión final Septiembre 11 de 2007

RESUMEN: En este artículo se presenta el proceso gradual de construcción de tipos abstractos de datos (TADs), como los Fraccionarios y los Polinomios, a partir de TADs esenciales, como los Lógicos y los Enteros. Se propone un conjunto de categorías funcionales para clasificar las funciones de un TAD. Las funciones se especifican mediante programación funcional, es decir, no se utilizan construcciones estructuradas como la asignación, la secuencia y los ciclos. Hasta ahora no se encuentra reportada la especificación de funciones con programación funcional para la simplificación en el TAD de los Enteros ni para la suma ordenada en el TAD de los polinomios. Además se muestra la relación inherente entre las funciones de un TAD y la sobrecarga de operadores.

PALABRAS CLAVE: Tipos abstractos de datos, programación orientada a objetos, programación funcional, encapsulamiento, sobrecarga de operadores.

ABSTRACT: In this paper we present the gradual process of building abstract data types (ADTs), such as Fractions and Polynomials. They are built from basic ADTs, such as Booleans and Integers. We propose a set of functional categories in order to classify ADT functions. These functions are stated in functional programming, that is without using variable assignment, sequence, or loops. Functions stated in this way to accomplish the simplification task of an Integer and to get an ordered sum for a Polynomial are not reported in literature so far. We also point to the inherent relation between ADT functions and operator overloading.

KEYWORDS: Abstract data types, object oriented programing, functional programming, encapsulation, operator overloading.

1. INTRODUCCIÓN

Uno de los temas de mayor importancia en el desarrollo de software es la definición de tipos abstractos de datos (TADs). Un tipo de datos es abstracto si está definido por su conjunto de funciones, sin importar la implementación [1] [2]. Por el contrario, una implementación selecciona una estructura de datos para representar el TAD [3].

Tal definición establece una conexión con la programación orientada a objetos (POO) con una de sus principales características:

el encapsulamiento. Aunque existe una fuerte analogía entre los TADs y la POO, ésta añade mecanismos como la herencia, el paso de mensajes y el polimorfismo [4].

Una función (o método) [5] "*es un proceso caracterizado por la acción que se realiza y por una serie de argumentos a través de los cuales toma información y devuelve resultados*". Un TAD [5] "*es el conjunto de valores que pueden tomar los datos de ese tipo y las funciones que los manipulan*", es decir, un TAD está constituido por un conjunto de valores y por un conjunto de métodos.

La especificación formal de un TAD requiere:
- un conjunto de funciones primarias (generadoras o de creación) que generan todos los posibles valores del TAD. Un TAD es compuesto si sus valores están constituidos por otros TADs los cuales se llaman TADs componentes (véase por ejemplo el TAD Fraccionario en la Sección 4).
- un conjunto de funciones secundarias que actúan sobre un conjunto de valores (instancias en POO) de TADs y producen un TAD resultado.

Las funciones secundarias pueden agruparse en categorías funcionales, de acuerdo al tipo de acción que realizan así:
•**Básicas**: realizan operaciones inherentes al TAD, por ejemplo en el TAD de los Enteros (véase Sección 3) las funciones aritméticas.
• **Simplificación o reducción**: simplifican la representación de un valor dado de un TAD en otro valor equivalente del TAD. Por ejemplo, el Fraccionario 9/6 se simplifica a 3/2. Cada TAD posee características que establecen que reglas de simplificación son válidas.
• **Selectoras u observadoras** [6]: son aquéllas que extraen los TADs componentes de un valor de un TAD compuesto, por ejemplo extraer el numerador (Entero) de un Fraccionario.
• **Comparación**: establecen el resultado de comparaciones entre valores de TADs, por ejemplo, la función de igualdad entre dos Enteros.
• **Auxiliares o privadas**: son usadas por otras funciones del TAD y no están orientadas al usuario final. Equivalen a los métodos privados de la POO. Véase por ejemplo la función Es_Positivo2 de los Enteros en la Sección 3.
• **Otras**: son aquéllas que no clasifican en ninguna de las categorías funcionales anteriores.

Para cada una de las funciones secundarias se debe especificar el resultado que se obtiene de acuerdo a los valores de sus TADs argumentos (parámetros). Estas especificaciones se denominan axiomas. Para su definición se puede emplear programación estructurada, programación funcional, modelos abstractos, como el cálculo de predicados, entre otros paradigmas. Incluso se puede utilizar lenguaje natural [7]. Las funciones primarias no requieren especificación.
En el desarrollo de este trabajo se utilizará programación funcional, la cual carece de asignación y de construcciones estructuradas como la secuencia y los ciclos. Por lo tanto sólo se puede acudir a la recursión y a la decisión (construcción **If** – **Else**) [8]. Este tipo de programación, a diferencia de la programación estructurada (imperativa) ofrece estas ventajas:
- expresa de una manera más compacta determinadas funciones, como las que se tratan acá.
- es más *pura* [9] ya que al no usar asignación (variables) se libera al programador de los aspectos referentes al manejo de la memoria.

Además las construcciones estructuradas como la secuencia y los ciclos mientras (While) y para (For) son *azúcar sintáctico* ya que pueden expresarse mediante programación funcional [2].
A las funciones se les puede asociar un símbolo para facilitar y hacer más natural su uso. Por ejemplo, a la función Suma de los Enteros es natural asociarle el símbolo **'+'**. Por lo tanto, en vez de escribir **Suma (Entero1, Entero2)**, se escribe **Entero1 + Entero2**. Se establece entonces la conexión con la sobrecarga de operadores de la POO. La sobrecarga de operadores permite asociar a un símbolo diferentes funciones. Tal y como se expresa en [10] "*el punto de la sobrecarga de operadores es proporcionar las mismas expresiones concisas para TADs*". Por ejemplo, el mismo símbolo **'+'** se puede asociar a la Suma de Fraccionarios: **Fraccionario1 + Fraccionario2**.

El artículo se estructura así: en las secciones 2, 3, 4 y 5 se presentan los TADs Lógico, de los Enteros, de los Fraccionarios y de los Polinomios respectivamente. En la Sección 6 se presentan conclusiones y trabajos futuros.

2. TAD LÓGICO: *L*

2.1 Definición

El TAD Lógico o Booleano está constituido por dos valores mutuamente excluyentes: Verdadero y Falso.

Se han definido varias operaciones entre valores lógicos, para un listado amplio véase [11]. Se consideran como operaciones básicas Negación (**Not**), Conjunción (**And**), Disyunción (**Or**), Implicación y Equivalencia.

2.2 Funciones

/* Función generadora */
Crear () → Lógico
// Genera un valor Lógico, Verdadero o Falso

/* Funciones lógicas básicas */
Negación (Lógico) → Lógico
// Operación Not
Conjunción (Lógico, Lógico) → Lógico
// Operación And
Disyunción (Lógico, Lógico) → Lógico
// Operación Or
Implicación (Lógico, Lógico) → Lógico
Equivalencia (Lógico, Lógico) → Lógico

2.3 Axiomas

∀ **x,y** ∈ ***L***:
• **Negacion**(x) ::= If x Then Falso Else Verdadero
• **Conjuncion**(x,y) ::= If x Then y Else Falso
• **Disyuncion**(x,y) ::= If x Then Verdadero Else y
• **Implicacion**(x,y) ::= If x Then y Else Verdadero
• **Equivalencia**(x,y) ::= If x Then y Else Negacion(y)

2.4 Sobrecarga de operadores

Sean **l1, l2** ∈ ***L***. La Tabla 1 muestra la sobrecarga de operadores del TAD Lógico.

Tabla 1. Sobrecarga de operadores para el TAD Lógico
Table 1. Operator overloading of Boolean ADT

Función	Operador
Negacion (Not)	¬ l1
Conjuncion (And)	l1 ∧ l2
Disyuncion (Or)	l1 ∨ l2
Implicacion	l1 → l2
Equivalencia	l1 ↔ l2

3. TAD DE LOS ENTEROS: *E*

3.1 Definición

Los números Enteros son aquéllos cuya parte decimal es nula. Pueden ser positivos, negativos o el cero.

3.2 Funciones

/* Funciones generadoras */
Cero() → Entero
// Genera el Entero 0 (cero)
Sucesor(Entero) → Entero
// Genera el Entero sucesor de un Entero
Antecesor(Entero) → Entero
// Genera el Entero anterior de un Entero

/* Funciones aritméticas básicas */
Suma(Entero, Entero) → Entero
Resta(Entero, Entero) → Entero
Multiplicacion(Entero, Entero) → Entero
Division_Entera(Entero, Entero) → Entero

/* Función de simplificación */
Simplificacion(Entero) → Entero
//Véase Sección 3.3

/* Funciones de comparación */
Es_Positivo(Entero) → Lógico
//Dice si un Entero es mayor que cero
Es_Negativo(Entero) → Lógico
//Dice si un Entero es menor que cero
Es_Cero(Entero) → Lógico
//Dice si un Entero es el cero
Comparar(Entero, Entero) → Entero
//Véase Sección 3.3
Igual(Entero, Entero) → Lógico
Mayor(Entero, Entero) → Lógico
Menor(Entero, Entero) → Lógico

Mayor_igual(Entero, Entero) → Lógico
Menor_igual(Entero, Entero) → Lógico
Diferente(Entero, Entero) → Lógico
Signo(Entero) → Entero
//Véase Sección 3.3

/* Otras funciones */
Valor_Absoluto(Entero) → Entero
Modulo(Entero, Entero) → Entero
//Retorna el residuo de la división entera
MCD(Entero, Entero) → Entero
//Máximo común divisor
MCM(Entero, Entero) → Entero
//Mínimo común múltiplo
Factorial(Entero) → Entero
//Factorial de un Entero Positivo

/* Funciones auxiliares */
Es_Positivo_Aux(Entero) → Lógico
//Auxiliar de la función Es_Positivo
MCD_Aux(Entero, Entero) → Entero
//Auxiliar de la función MCD
MCM_Aux(Entero, Entero) → Entero
//Auxiliar de la función MCM
Existe_Pareja(Entero) →Lógico
//Auxiliar de la función Simplificación
Eliminar_Pareja(Entero) → Entero
//Auxiliar de la función simplificación

3.3 Observaciones

Sean **x**, **y** ∈ ***E***.
1. La función Simplificación reduce un Entero a su mínima expresión (EME). Un EME está conformado sólo de funciones sucesoras o antecesoras o es el cero.
Ejemplo: con las funciones generadoras es posible obtener el Entero:

Sucesor(**Antecesor**(**Sucesor**(**Sucesor**(**Antecesor**(**Cero**())))))

Este Entero es el 1, por lo tanto su EME es: **Sucesor**(**Cero**()).

2. La función **Comparar**(x, y) se define así:

$$\text{Comparar}(x,y)=\begin{cases}0 \; si \; x=y\\ 1 \; si \; x>y\\ -1 \; si \; x<y\end{cases}$$

3. La función **Signo**(x) se define así:

$$\text{Signo}(x,y)=\begin{cases}0 \; si \; x=0\\ 1 \; si \; x>0\\ -1 \; si \; x<0\end{cases}$$

3.4 Axiomas

∀ **x**, **y** ∈ ***E***; **x** e **y** no necesariamente EMEs.

• **Suma**(x, **Sucesor**(y)) ::= **Sucesor**(**Suma**(x, y))
• **Suma**(x, **Antecesor**(y))
::=**Antecesor**(**Suma**(x, y))
• **Suma**(x, **Cero**()) ::= x

• **Resta**(x, **Sucesor**(y)) ::=
Antecesor(**Resta**(x, y))
• **Resta** (x, **Antecesor**(y)) ::=
Sucesor(**Resta**(x, y))
• **Resta** (x, **Cero**()) ::= x
• **Multiplicacion**(x, **Sucesor**(y)) ::=
Suma(**Multiplicacion**(x, y), x)
• **Multiplicacion**(x, **Antecesor**(y)) ::=
Resta(**Multiplicacion**(x, y), x)
• **Multiplicacion** (x, **Cero**()) ::= **Cero**()

• **Division_Entera**(**Cero**(), **Antecesor**(y)) ::=
Cero()
• **Division_Entera** (**Cero**(), **Sucesor**(y)) ::=
Cero()
• **Division_Entera**(x, **Cero**()) ::= Error
• **Division_Entera**(**Sucesor**(x), **Sucesor**(y))
::=
If **Mayor_Igual**(**Sucesor**(x), **Sucesor**(y))
Then
Sucesor(**Division_Entera**(**Resta**(**Sucesor**(x)
, **Sucesor**(y)), **Sucesor**(y)))
Else **Cero**()
• **Division_Entera**(**Antecesor**(x),
Antecesor(y))
::= **Division_Entera**(**Valor_Absoluto**(
Antecesor(x)),
Valor_Absoluto(**Antecesor**(y)))
• **Division_Entera**(**Sucesor**(x),
Antecesor(y)) ::=
If **Mayor_Igual**(**Sucesor**(x),
Valor_Absoluto(**Antecesor**(y))) Then

Antecesor(**Division_Entera**(**Resta**(**Sucesor**(x), **Valor_Absoluto**(**Antecesor**(y))), **Antecesor**(y)))
Else **Cero**()• **Division_Entera**(**Antecesor**(x), **Sucesor**(y)) ::=
Division_Entera(**Valor_Absoluto**(**Antecesor**(x)), **Resta**(**Cero**(), **Sucesor**(y)))

La función de Simplificación utiliza dos funciones auxiliares:
-Existe_Pareja(x): detecta si en la conformación de un Entero existe al menos una función Sucesor y al menos una función Antecesor.
-Eliminar_Pareja(x): elimina de la conformación de un Entero una función Sucesor y una función Antecesor.

• **Simplificacion**(x) ::= If **Existe_Pareja**(x) Then **Simplificacion**(**Eliminar_Pareja**(x))
Else x

• **Existe_Pareja**(**Sucesor**(**Antecesor**(x))) ::= Verdadero
• **Existe_Pareja**(**Antecesor**(**Sucesor**(x))) ::= Verdadero
• **Existe_Pareja**(**Antecesor**(**Antecesor**(x))) ::= **Existe_Pareja**(**Antecesor**(x))
• **Existe_Pareja**(**Sucesor**(**Sucesor**(x))) ::= **Existe_Pareja**(**Sucesor**(x))
• **Existe_Pareja**(**Sucesor**(**Cero**())) ::= Falso
• **Existe_Pareja**(**Antecesor**(**Cero**())) ::= Falso
• **Existe_Pareja**(**Cero**()) ::= Falso
• **Eliminar_Pareja**(**Sucesor**(**Antecesor**(x)) ::= x
• **Eliminar_Pareja**(**Antecesor**(**Sucesor**(x))) ::= x
• **Eliminar_Pareja**(**Antecesor**(**Antecesor**(x))) ::= **Antecesor**(**Eliminar_Pareja**(**Antecesor**(x))
• **Eliminar_Pareja**(**Sucesor**(**Sucesor**(x))) ::= **Sucesor**(**Eliminar_Pareja**(**Sucesor**(x))
En [5] no se logra la simplificación de un Entero por medio de un axioma, en lugar de ello se establece que *implícitamente* se deben realizar las siguientes equivalencias para obtener un EME:

Sucesor (**Antecesor** (x)) = x
Antecesor (**Sucesor** (x)) = x

Por ejemplo, si se tiene el Entero: z = **Sucesor**(**Sucesor**(**Antecesor**(**Antecesor**(**Cero**()))))
La función **Simplificacion**(z) devuelve **Cero**()

El axioma aquí planteado a diferencia de las equivalencias planteadas en [5] realiza *explícitamente* la simplificación de un Entero.
Si una función no recibe EMEs, ésta invoca a otra auxiliar que recibe el Entero simplificado, por ejemplo sea **z** $\in$ ***E*** no necesariamente un EME, entonces:

• **Es_Positivo**(z) ::= **Es_Positivo_Aux**(**Simplificacion**(z))
• **Es_Positivo_Aux**(**Cero**()) ::= Falso
• **Es_Positivo_Aux**(**Sucesor**(w)) ::= Verdadero
• **Es_Positivo_Aux**(**Antecesor**(w)) ::= Falso

Donde w es un EME. Por lo tanto para el resto de las funciones se supone que **x** e **y** son EMEs, es decir antes se les aplicó la función de Simplificación.

• **Es_Negativo**(**Cero**()) ::= Falso
• **Es_Negativo**(**Sucesor**(x)) ::= Falso
• **Es_Negativo**(**Antecesor**(x)) ::= Verdadero

• **Es_Cero**(**Cero**()) ::= Verdadero
• **Es_Cero**(**Sucesor**(x)) ::= Falso
• **Es_Cero**(**Antecesor**(x)) ::= Falso

• **Comparar**(**Sucesor**(x), **Sucesor**(y)) ::= **Comparar**(x, y)
• **Comparar**(**Antecesor**(x), **Antecesor**(y)) ::= **Comparar**(x, y)
• **Comparar**(**Sucesor**(x), **Antecesor**(y)) ::= **Sucesor**(**Cero**())
• **Comparar**(**Antecesor**(x), **Sucesor**(y)) ::= **Antecesor**(**Cero**())
• **Comparar**(**Sucesor**(x), **Cero**()) ::= **Sucesor**(**Cero**())
• **Comparar**(**Antecesor**(x), **Cero**()) ::= **Antecesor**(**Cero**())
• **Comparar**(**Cero**(), **Sucesor**(x)) ::= **Antecesor**(**Cero**())
• **Comparar**(**Cero**(), **Antecesor**(x)) ::= **Sucesor**(**Cero**())
• **Comparar**(**Cero**(), **Cero**()) ::= **Cero**()

• **Igual**(x, y) ::= **Es_Cero**(**Comparar**(x, y))
• **Mayor**(x, y) ::= **Es_Positivo**(**Comparar**(x, y))

• **Menor**(x, y) ::= **Es_Negativo**(**Comparar**(x, y))
• **Mayor_Igual**(x, y) ::= ¬ **Menor**(x, y)
• **Menor_Igual**(x, y) ::= ¬ **Mayor**(x, y)
• **Diferente**(x, y) ::= ¬ **Igual**(x, y)
• **Signo**(x) ::= **Comparar**(x, **Cero**())

• **Valor_Absoluto**(**Cero**()) ::= **Cero**()
• **Valor_Absoluto**(**Sucesor**(x)) ::= **Sucesor**(x)
• **Valor_Absoluto**(**Antecesor**(x)) ::= **Sucesor**(**Valor_Absoluto**(x))

• **Modulo**(x, **Cero**()) ::= Error
• **Modulo**(**Cero**(), y) ::= **Cero**()
• **Modulo**(x, y) ::= If **Menor**(**Valor_Absoluto**(x), **Valor_Absoluto**(y)) Then x
Else **Modulo**(**Resta**(**Valor_Absoluto**(x), **Valor_Absoluto**(y)), y)

Para el MCD de los Enteros se adopta la siguiente definición [12]: "*el MCD de los enteros* b *y* c *es el mayor valor positivo entre sus divisores comunes*"; además: "*el MCD está definido para todo par de enteros* b, c *excepto si uno de los dos es cero*".
De acuerdo con lo anterior se define el axioma correspondiente al MCD:

• **MCD**(x, **Cero**()) ::= Error
• **MCD**(**Cero**(), y) ::= Error
• **MCD**(x, y) ::= If **Igual**(**Modulo**(x, y), **Cero**()) Then y Else **MCD**(y, **Modulo**(x, y))

De manera similar para obtener el MCM [12] "*El menor de los múltiplos comunes positivos de los enteros b y c recibe el nombre de* MCM". Para calcular el MCM de dos números naturales a y b se aplica la fórmula **MCM**(a, b) = a * b/**MCD**(a, b)
Por lo tanto:

• **MCM**(x, y) ::= **MCM_Aux**(**Valor_Absoluto**(x), **Valor_Absoluto**(y))
• **MCM_Aux**(x, y) ::= **Division_Entera**(**Multiplicacion**(x, y), **MCD**(x, y))
• **Factorial**(**Cero**()) ::= **Sucesor**(**Cero**())
• **Factorial**(**Sucesor**(x)) ::= **Multiplicación**(**Sucesor**(x), **Factorial**(x))
• **Factorial**(**Antecesor**(x)) ::= Error

3.5 Sobrecarga de operadores

La Tabla 2 muestra la sobrecarga del TAD de los Enteros.

Tabla 2. Sobrecarga de operadores para el TAD de los Enteros
Table 2. Operator overloading of Integer ADT

Función	Operador
Suma	x + y
Resta	x - y
Multiplicacion	x * y
Division_Entera	x / y
Es_Positivo	$?_{+}x$
Es_Negativo	?.x
Es_Cero	$?_{0}x$
Comparar	x ~ y
Igual	x = y
Mayor	x > y
Menor	x < y
Mayor_Igual	x ≥ y
Menor_Igual	x ≤ y
Diferente	x ≠ y
Valor_Absoluto	$\|x\|$
Modulo	x % y
Factorial	x!

4. TAD DE LOS FRACCIONARIOS: *F*

4.1 Definición

Sean **n**, **d** ∈ ***E***. Un Fraccionario es una pareja **(n, d)** donde **n** es el numerador y **d** es el denominador. Esta pareja representa la división (que produce un número real) **n/d** la cual se deja indicada.
Ejemplo: (3, 2) representa 3/2, es decir el real 1,5.

4.2 Funciones

/* Función generadora */
Crear(Entero, Entero) → Fraccionario
// Genera un Fraccionario

/* Funciones selectoras */
Num(Fraccionario) → Entero
// Devuelve el numerador de un Fraccionario
Den(Fraccionario) → Entero
// Devuelve el denominador de un Fraccionario

/* Función de simplificación */
Reduccion(Fraccionario) → Fraccionario

/* Funciones aritméticas básicas */

Suma(Fraccionario, Fraccionario) → Fraccionario
Resta(Fraccionario, Fraccionario) → Fraccionario
Multiplicacion(Fraccionario, Fraccionario) → Fraccionario
Division(Fraccionario, Fraccionario) → Fraccionario

/* Funciones de comparación */
Comparar(Fraccionario, Fraccionario) → Entero
// Ver observación en la Sección 4.3
Igual(Fraccionario, Fraccionario) → Lógico
Mayor(Fraccionario, Fraccionario) → Lógico
Menor(Fraccionario, Fraccionario) → Lógico
Mayor_Igual(Fraccionario, Fraccionario) → Lógico
Menor_Igual(Fraccionario, Fraccionario) → Lógico
Diferente(Fraccionario, Fraccionario) → Lógico

/* Función auxiliar */
Comparar_Aux(Fraccionario, Fraccionario) → Entero
//Auxiliar para la función Comparar

4.3 Observaciones

1. La reducción de un Fraccionario consiste en simplificarlo dividiendo (división entera) tanto el numerador como el denominador entre el **MCD** de ellos dos, por ejemplo (9, 6) se convierte en (3, 2) ya que **MCD**(9, 6) = 3.

2. La comparación entre dos Fraccionarios es análoga a la comparación de Enteros (véase la observación 2 de la Sección 3.3).

4.4 Axiomas

∀ **x, y** ∈ ***F***
• **Reduccion**(p) ::= **Crear**(**Num**(p)/**MCD**(**Num**(p), **Den**(p)), **Den**(p)/**MCD**(**Num**(p), **Den**(p)))

• **Suma**(p, q) ::= **Reduccion**(**Crear**(**Num**(p) * **Den**(q) + **Num**(q) * **Den**(p)), **Den**(p) * **Den**(q)))
• **Resta**(p, q) ::= **Reduccion**(**Crear**(**Num**(p) * **Den**(q) - **Num**(q) * **Den**(p)), **Den**(p) * **Den**(q)))

• **Multiplicacion**(p, q) ::= **Reduccion**(**Crear**(**Num**(p) * **Num**(q),
Den(p) * **Den**(q)))
• **Division**(p, q) ::= **Reduccion**(**Crear**(**Num**(p) * **Den**(q), **Den**(p) * **Num**(q)))

• **Comparar**(p, q) ::= **Comparar_Aux**(**Reduccion**(p), **Reduccion**(q))
• **Comparar_Aux** (p, q) ::= **Num**(p) * (**MCM**(
Den(p), **Den**(q))/**Den**(p)) ~ **Num**(q) * (**MCM**(
Den(p), **Den**(q))/**Den**(q))

• **Igual**(p, q) ::= $?_0$(**Comparar**(p, q))
• **Mayor**(p, q) ::= $?_+$(**Comparar**(p, q))
• **Menor**(p, q) ::= $?_-$(**Comparar**(p, q))
• **Mayor_Igual**(p, q) ::= ¬ **Menor**(p, q)
• **Menor_Igual**(p, q) ::= ¬ **Mayor**(p, q)
• **Diferente**(p, q) ::= ¬ **Igual**(p, q)

Nótese el uso de la sobrecarga de operadores de los Enteros expuestos en la Tabla 2 y el uso del operador ~ (Comparar) de los Enteros.

4.5 Sobrecarga de operadores

La Tabla 3 muestra la sobrecarga de operadores del TAD de los Fraccionarios.

Tabla 3. Sobrecarga de operadores para el TAD de los Fraccionarios
Table 3. Operator overloading of Fraction ADT

Función	Operador
Suma	p + q
Resta	p - q
Multiplicacion	p * q
Division_Entera	p / q
Comparar	p ~ q
Igual	p = q
Mayor	p > q
Menor	p < q
Mayor_Igual	p ≥ q
Menor_Igual	p ≤ q
Diferente	p ≠ q

5. TAD DE LOS POLINOMIOS: *P*

5.1 Definición

Un Polinomio es una secuencia de términos de la forma $\mathbf{cx}^{\mathbf{e}}$, en la cual **c** es un coeficiente

y **e** es un exponente. Sean **c, e, x** ∈ ***F*,** donde x es la variable independiente.
Se propone que **c, e** y **x** ∈ ***F*** pero podrían restringirse a ***E***.

5.2 Observaciones

Se presenta un TAD para Polinomios teniendo en cuenta dos alternativas:
a) sin restricciones: los términos pueden estar en cualquier orden y puede existir más de un término con el mismo exponente,
b) con restricciones (PCR): sólo puede existir un término por exponente y los términos deben estar ordenados descendentemente por exponente.
Se comienza con el TAD Polinomio sin restricciones.

5.3 Funciones

/* Funciones generadoras */
Cero() → **Polinomio**
// Genera un Polinomio sin términos
Adicionar_Termino(Polinomio, Fraccionario, Fraccionario) → Polinomio
// Incluye un término en el Polinomio.

/* Función de simplificación */
ATS(Polinomio) → Polinomio
//Agrupa los términos semejantes de un Polinomio, dejando sólo un término por exponente.

/* Funiones de comparación */
Es_Anulable(Polinomio) → Polinomio
//Dice si un Polinomio se anula o no, véase Sección 5.4
Ordenamiento(Polinomio) → Polinomio
//Ordena los términos de un Polinomio en forma descendente por exponente

/* Funciones básicas */
Suma(Polinomio, Polinomio) → Polinomio
Resta(Polinomio, Polinomio) → Polinomio
Multiplica_Ter(Polinomio, Fraccionario, Fraccionario) → Polinomio
// Multiplica un Polinomio por un término
Multiplicacion(Polinomio, Polinomio) → Polinomio
Division(Polinomio, Polinomio) → Polinomio
Residuo(Polinomio, Polinomio) → Polinomio
//Retorna el Polinomio residuo de la división
Grado(Polinomio) → Fraccionario
// Máximo exponente de un Polinomio
Derivada(Polinomio) → Polinomio
Integral(Polinomio) → Polinomio

/* Otras Funciones */
Remover(Polinomio, Fraccionario) → Polinomio // Elimina de un Polinomio los términos que tengan un exponente dado
Coeficiente(Polinomio, Fraccionario) → Fraccionario
// Retorna la suma de los coeficientes de los términos que tengan un exponente dado

5.4 Observaciones

Un Polinomio es anulable si la suma de todos sus términos es cero, por ejemplo, $3x^2 + 8x^{-9} – 3x^2 – 8x^{-9}$.

5.5 Axiomas

∀ **p, q** ∈ ***P***; ∀ **c, d, e, f** ∈ ***F***. **c, d** son coeficientes; **e, f** son exponentes.

• **ATS**(**Cero**()) ::= **Cero**()
• **ATS**(**Adicionar_Termino**(p, c, e)) ::=
If **?**$_0$(**Coeficiente**(p, e) + c) Then
ATS(**Remover**(p, e))
Else **Adicionar_Termino**(**ATS**(**Remover**(p, e)), c + **Coeficiente**(p, e), e)

• **Es_Anulable** (**Cero**()) ::= Verdadero
• **Es_Anulable** (**Adicionar_Termino**(p, c, e)) ::=
If **?**$_0$(**Coeficiente**(p, e)–c) Then
Es_Anulable(**Remover**(p, e))
Else Falso

• **Ordenamiento**(Cero()) ::= Cero()
• **Ordenamiento**(**Adicionar_Termino**(**Cero**(), c, e) ::= **Adicionar_Termino**(**Cero**(), c, e)
• **Ordenamiento**(p) ::= **Adicionar_Termino**(**Ordenamiento**(**Remover**(p, **Grado**(p)), **Coeficiente**(p, **Grado**(p)), **Grado**(p))

• **Suma**(**Cero**(), q) ::= q
• **Suma**(**Adicionar_Termino**(p, c, e), q) ::= **Adicionar_Termino**(**Suma**(p, q), c, e)

• **Resta**(**Cero**(), q) ::= **Multiplica_Ter**(q, **Antecesor**(**Cero**()), **Cero**())
• **Resta**(p, **Cero**()) ::= p
• **Resta**(p, q) ::= **Suma**(p, **Resta**(**Cero**(), q))

• **Multiplica_Ter**(**Cero**(), d, f) ::= **Cero**()
• **Multiplica_Ter**(**Adicionar_Termino**(p, c, e), d, f) ::= **Adicionar_Termino**(**Multiplica_Ter**(p, d, f), c * d, e + f)

• **Multiplicacion**(**Cero**(), q) ::= **Cero**()
• **Multiplicacion**(**Adicionar_Termino**(p, c, e), q) ::= **Suma**(**Multiplicacion**(p, q), **Multiplica_Ter**(q, c, e))

• **Division**(p, **Cero**()) ::= Error
• **Division**(**Cero**(), q) ::= **Cero**()
• **Division**(p, q) ::= If **Grado**(p) >= **Grado**(q) Then **Adicionar_Termino**(**Division**(**Remover**(p, **Grado**(p)),q),**División**(**Coeficiente**(p,**Grado**(p)), **Coeficiente**(q, **Grado**(q))), **Resta**(**Grado**(p), **Grado**(q))) Else **Cero**()

• **Residuo**(**Cero**(), q) ::= q
• **Residuo**(p, **Cero**()) ::= Error
• **Residuo**(p, q) ::= If **Grado**(p) >= **Grado**(q) Then **Residuo**(**Division**(**Remover**(p, **Grado**(p)), q), **División**(**Coeficiente**(p, **Grado**(p)), **Coeficiente**(q, **Grado**(q))), **Resta**(**Grado**(p), **Grado**(q))) Else p

• **Grado**(**Cero**()) ::= Error
• **Grado**(**Adicionar_Termino**(**Cero**(), c, e)) ::= e
•**Grado**(**Adicionar_Termino**(**Adicionar_Termino**(p, c, e), d, f)) ::= If e > f Then **Grado**(**Adicionar_Termino**(p, c, e)) Else **Grado**(**Adicionar_Termino**(p, d, f))

• **Derivada**(**Cero**()) ::= **Cero**()
• **Derivada**(**Adicionar_Termino**(p, c, e)) ::= If **?**$_0$(e) Then **Derivada**(p) Else **Adicionar_Termino**(**Derivada**(p), c * e, e - 1)

• **Integral**(**Cero**()) ::= **Adicionar_Termino**(**Cero**(), c, **0**)
• **Integral**(**Adicionar_Termino**(p, c, e)) ::= **Adicionar_Termino**(**Integral**(p), c/(e + 1)), e + 1)
• **Remover**(**Cero**(), f) ::= **Cero**()
• **Remover**(**Adicionar_Termino**(p, c, e), f) ::= If e = f Then **Remover**(p, f) Else **Adicionar_Termino**(**Remover**(p, f), c, e)

• **Coeficiente**(**Cero**(), f) ::= **0**
• **Coeficiente**(**Adicionar_Termino**(p, c, e), f) ::= If e = f Then c + **Coeficiente**(p, f) Else **Coeficiente**(p, f)

Las funciones presentadas para polinomios que no son necesariamente PCRs trabajan igualmente para PCRs, sin embargo algunas pueden ser redefinidas.
Un PCR se puede obtener al aplicar las funciones ATS y Ordenamiento, es decir, sea **p** un Polinomio sin restricciones entonces **Ordenamiento**(**ATS**(p)) produce un PCR.
Sea **r'** un PCR representado por **Adicionar_Termino**(p', c, e) donde **p'** se refiere al resto del Polinomio de **r'**, por lo tanto el término de mayor exponente es el de coeficiente **c** y exponente **e**.
Los axiomas para las funciones Grado y Suma se redefinen así:
Sean **p'**, **q'** PCRs.
• **Grado**(**Cero**()) ::= Error
• **Grado**(**Adicionar_Termino**(p', c, e)) ::= e

• **Suma**(**Cero**(), p') ::= p'
• **Suma**(q', **Cero**()) ::= q'
• **Suma**(**Adicionar_Termino**(p', c, e), **Adicionar_Termino**(q', d, f)) ::=
If e > f Then **Adicionar_Termino**(**Suma**(p', **Adicionar_Termino**(q', d, f), c, e))
Else
If e < f Then **Adicionar_Termino**(**Suma**(**Adicionar_Termino**(p', c, e), q'), d, f)
Else
If **?**$_0$ (c + d) Then **Suma**(p, q)Else **Adicionar_Termino**(**Suma**(p', q'), c + d, e)
El axioma de Suma produce a su vez un PCR. Este axioma es una solución puramente funcional a diferencia del propuesto en [13], el cual usa iteración y la estructura secuencia, como se presenta a continuación.
Sean **a, b, c** PCRs. El símbolo **:=** es el operador de asignación en este caso.

Begin
c := **Cero**()
While ¬ **Es_Anulable**(a) ∧ ¬ **Es_Anulable**(b)
Case
Grado(a) < **Grado**(b):

c := **Adicionar_Termino**(c, **Coeficiente**(b, **Grado**(b), **Grado**(b))
b := **Remover**(b, **Grado**(b))

Grado(a) = **Grado**(b):
c := **Adicionar_Termino**(c, **Coeficiente**(a, Grado(a)) **Coeficiente**(b, **Grado**(b)), **Grado**(a))
a := **Remover**(a, **Grado**(a))
b := **Remover**(b, **Grado**(b))
Grado(a) > **Grado**(b):
a := **Adicionar_Termino**(c, **Coeficiente**(a, **Grado**(a)), **Grado**(a))
b := **Remover**(a, **Grado**(a))
End Case
End While

While ¬ **Es_Anulable**(a)
c := **Adicionar_Termino**(c, **Coeficiente**(a, **Grado**(a)), **Grado**(a))
a := **Remover**(a, **Grado**(a))
End While

While ¬ **Es_Anulable**(b)
c := **Adicionar_Termino**(c, **Coeficiente**(b, **Grado**(b)),**Grado**(b))
b := **Remover**(b, **Grado**(b))
End While
End

5.6 Sobrecarga de Operadores

La Tabla 4 muestra la sobrecarga de operadores del TAD de los polinomios.

Tabla 4. Sobrecarga de operadores para el TAD de los Polinomios
Table 4. Operator overloading of Polynomial TAD

Función	Operador
Suma	p + q
Resta	p - q
Multiplicacion	p * q
Division	p / q
Residuo	p % q
Grado	$_G$(p)
Derivada	$_D$(p)
Integral	∫p

6. CONCLUSIONES Y TRABAJOS FUTUROS

Se han especificado cuatro TADs, cada uno con sus funciones y axiomas correspondientes. Se ha mostrado como la especificación de un nuevo TAD utiliza otros TADs, lográndose así un proceso de construcción gradual. Este proceso se ejemplifica en la Figura 1.

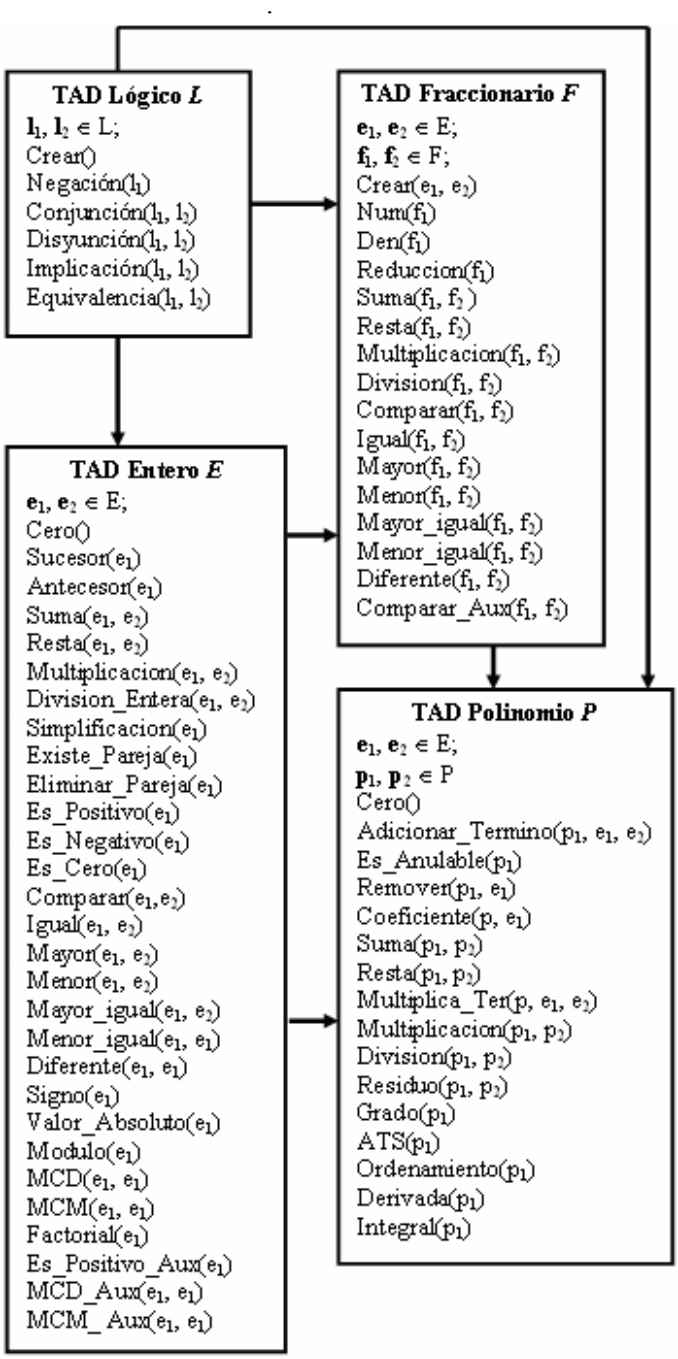

Figura 1. Relación entre TADs
Figure 1. Relationship among ADTs

En un trabajo futuro se presentarán otros tads, como los vectores, matrices y los

números complejos. También se pretende implementar los tads expuestos en el lenguaje funcional *scheme*, en el cual a partir de sus funciones primitivas lógicas (and, or, not), add1 y sub1 (las cuales equivalen a las funciones antecesor y sucesor respectivamente) se pueden construir los demás tads para observar su comportamiento. igualmente se podría trabajar con lenguajes como *maude* [14] y *haskell* [15] y establecer comparaciones entre las diferentes implementaciones. en [16] se hace una presentación de otros lenguajes similares que se podrían utilizarreferencias

REFERENCIAS

[1] VAN ROY P., HARDINI S.; Concepts, Techniques, and Models of Computer Programming, The MIT Press, 2004.

[2] ABELSON H., SUSSMAN G. J., SUSSMAN J.; Structure and Interpretation of Computer Programs, MIT Press y McGraw-Hill, 1996.

[3] AHO A. V., HOPCROFT J. E., ULLMAN J. D.; Data Structures and Algorithms, Addison-Wesley Series in Computer Science and Information Processing, 1983.

[4] LOUDENK. C.; Programming Languages: Principles and Practice, PWS-Kent, Boston, Mass., USA, 1993.

[5] JOYANES L., ZAHONERO I., FERNÁNDEZ M., SÁNCHEZ L.; Estructura de Datos: Libro de Problemas, McGraw-Hill, 1999.

[6] GULUTZAN P, PELZER T.; SQL-99 Complete Really, R & B Books, 1999.

[7] STUBBS D. F.; Data Structures With Abstract Data Types and Pascal, Brooks/Cole, 1985.

[8] HUDAK P.; Conception, Evolution, and Application of Functional Programming Languages, ACM Computing Surveys 21 (3): 359-411, Septiembre 1989.

[9] SEBESTA R. W.; Concepts of Programming Languages, Addison Wesley, 2005.

[10] DEITEL H, DEITEL P.; Como Programar en C/C++, Prentice Hall, 1994.

[11] KOLMAN B., BUSBY R. C., ROSS S.; Discrete Mathematical Structures, Prentice Hall, 2003.

[12] ZUCKERMAN H, NIVEN I.; Introducción a la Teoría de los Números, Limusa, 1976.

[13] HOROWITZ E, SAHNI S.; Fundamentals of Data Structures, Computer Science, 1976.

[14] The Maude System. Disponible en: http://maude.cs.uiuc.edu [citado 14 de Mayo de 2007]

[15] Haskell. Disponible en: http://www.haskell.org [citado 14 de Mayo de 2007]

[16] GALÁN F, CAÑETE J.; Métodos Formales Orientados a Objetos, Informe Técnico ETSI, 2000.