ESPECIFICACIÓN ALGEBRAICA DE TIPOS DE DATOS ABSTRACTOS: EL LENGUAJE NEREUS

Liliana Favre

Se describen a continuación las construcciones del lenguaje algebraico NEREUS utilizado en el curso para especificar tipos de datos abstractos. Una introducción a las especificaciones algebraicas y su relación con el paradigma de orientación a objetos puede consultarse en (Meyer, 1997; Capítulo 6)

1. Especificaciones básicas en NEREUS

La especificación de un TDA se encapsula dentro de una clase en NEREUS. A continuación se muestra en la sintaxis de una especificación básica en NEREUS.

CLASS className [<*parameterList*>]
IMPORTS
<*importList*>
BASIC CONSTRUCTORS
<*constructorList*>
EFFECTIVE
TYPES
<*typeList*>
OPERATIONS
<*operationList*>
AXIOMS <*varList*>
<*axiomList*>
END-CLASS

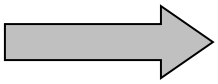
Podemos distinguir en una clase una especificación sintáctica, dada por la signatura del tipo y una especificación semántica dada por la cláusula AXIOMS.

CLASS className [<*parameterList*>]
IMPORTS <*importList*>
BASIC CONSTRUCTORS<*constructorList*>
EFFECTIVE
TYPES <*sortList*>
OPERATIONS
<*operationList*>

Signatura

Especificación sintáctica

AXIOMS <*varList*>
<*axiomList*>
END-CLASS



Axiomas

Especificación semántica

El encabezamiento de la clase declara el nombre de la clase y una lista de parámetros <*parameterList*>. La cláusula IMPORTS lista a las especificaciones importadas, es decir expresa relaciones cliente. La especificación de la nueva clase está basada en las especificaciones importadas declaradas en <*importList*>.

La cláusula **BASIC CONSTRUCTORS** lista las operaciones constructoras básicas.

La cláusula **EFFECTIVE** declara nuevos tipos, operaciones y axiomas.

Una declaración de tipos tiene la forma **TYPES** $s_1, s_2,...s_n$

En la cláusula **OPERATIONS** se declaran las funcionalidades de las operaciones con la sintaxis habitual. Es posible definir operaciones en forma parcial. El dominio de definición de una función parcial puede hacerse explícito mediante el uso de aserciones, que deben suceder a la funcionalidad de la operación tras la palabra clave "**pre:**".

Tras la palabra clave **AXIOMS** se declaran pares de la forma *v1: C1* donde *v1* es una variable universalmente cuantificada de tipo *C1*. Los axiomas incluidos a continuación de esta declaración expresan las propiedades requeridas por la especificación a partir de expresiones en lógica de primer orden construidas sobre términos y fórmulas.

Un término es una variable *tipada*, una constante o una aplicación de una operación en la que los argumentos satisfacen los tipos del dominio y el rango de la operación.

Las fórmulas son atómicas o compuestas. Una fórmula atómica es una ecuación entre dos términos del mismo tipo, separados por "=".

Las ecuaciones de la forma *término = True* pueden ser abreviadas escribiendo simplemente el término. Por ejemplo, *vaciaLista (inicLista()) = TRUE* puede escribirse como *vaciaLista (inicLista())*.

Una fórmula compuesta (o predicado) puede ser construida a partir de los conectivos lógicos *not, and, or,* $\Rightarrow$ y $\Leftrightarrow$.

Se muestra a continuación la signatura de la clase *Boolean*:

CLASS Boolean **BASIC CONSTRUCTORS** True, False **EFFECTIVE** **TYPE** Boolean **OPERATIONS** True: →Boolean False: →Boolean not_: Boolean →Boolean _and_: Boolean x Boolean → Boolean _=_ : Boolean x Boolean → Boolean _or_: Boolean x Boolean → Boolean _xor_: Boolean x Boolean → Boolean _⇒_: Boolean x Boolean → Boolean _⇔:_ Boolean x Boolean → Boolean if_then_else: Boolean x Boolean x Boolean → Boolean	**AXIOMS** b, b1: Boolean not True = False not False = True b and True = b b and False = False b or True = True b or False = b True xor b = not b False xor b = b True ⇒ b = b False ⇒ b = True True ⇔ b = b False ⇔ b = not b if_then_else (True, b, b1) = b if_then_else (False, b, b1) = b1 **END-CLASS**

La operación *if_then_else* es provista por el TDA *Boolean* para facilitar el uso de formas condicionales.

Todas las cláusulas son opcionales y no existe un orden entre ellas exceptuando el impuesto por la visibilidad lineal: todo símbolo tiene que ser declarado antes de ser usado.

Se presentan a continuación ejemplos de especificaciones básicas NEREUS.

Ejemplo 1: La clase Pila

```
CLASS Pila [Elemento]
IMPORTS Boolean
BASIC CONSTRUCTORS
   inicPila, agregarPila
EFFECTIVE
TYPES Pila

OPERATIONS
    inicPila: -> Pila
    agregarPila: Pila x Elemento -> Pila
    vaciaPila: Pila -> Boolean
    tope: Pila (p) -> Elemento
            pre: not vaciaPila (p)
    eliminarPila: Pila (p) -> Pila
             pre: not vaciaPila (p)

AXIOMS p:Pila; e: Elemento
    vaciaPila (inicPila()) = True
    vaciaPila (agregarPila (p,e) ) = False
    tope (agregarPila (p, e) ) = e


    eliminarPila (agregarPila (p, e)) = p


END-CLASS
```

Ejemplo 2 : La clase Fila

```
CLASS Fila [Elemento]
IMPORTS Boolean
BASIC CONSTRUCTORS
   inicFila,agregarFila
EFFECTIVE
TYPES Fila

OPERATIONS
   inicFila: -> Fila
   agregarFila: Fila x Elemento -> Fila
   vaciaFila: Fila -> Boolean
   recuperarFila: Fila (f) -> Elemento
            pre: not vaciaFila (f)
   eliminarFila: Fila (f) -> Fila
             pre: not vaciaFila (f)

AXIOMS f: Fila; e: Elemento
   vaciaFila (inicFila() ) =True
   vaciaFila (agregarFila (f,e)) = False
   recuperarFila (agregarFila ( f, e)) =
         if vaciaFila (f) then e
         else recuperarFila (f)
   eliminarFila (agregarFila(f, e)) =
         if vaciaFila(f) then inicFila()
         else agregarFila (eliminarFila (f), e)
END-CLASS
```

Ejemplo 3: La clase árbol binario

La clase *Arbin* define una especificación de árboles binarios. Sus operaciones constructoras básicas son las operaciones *inicArbin* y *crearArbin.* Las operaciones observadoras son *raiz, vacioArbin* y las transformadoras *subIzquierdo* y *subDerecho.*

```
CLASS Arbin [Elemento]
IMPORTS Boolean
BASIC CONSTRUCTORS inicArbin, crearArbin
EFFECTIVE
TYPES Arbin

OPERATIONS
        inicArbin: → Arbin
        crearArbin: Arbin x Arbin x Elemento → Arbin
        vacioArbin: Arbin → Boolean
        raiz: Arbin(t) → Elemento
                pre: not vacioArbin(t)
        subIzquierdo: Arbin (t) → Arbin
                pre: not vacioArbin(t)
        subDerecho: Arbin (t) → Arbin
                pre: not vacioArbin(t)

AXIOMS  t1,t2: Arbin; e: Elemento
        vacioArbin (inicArbin ()) = True
        vacioArbin (crearArbin(t1,t2,e)) = False
        raiz(crearArbin(t1,t2,e)) = e
        subIzquierdo(crearArbin(t1,t2,e)) = t1
        subDerecho(crearArbin(t1,t2,e)) = t2
END-CLASS
```

2. Especificaciones incompletas

A continuación se muestra la sintaxis de una especificación incompleta en NEREUS

```
CLASS  className [<parameterList>]
IMPORTS
    <importList>
BASIC CONSTRUCTORS
    <constructorList>
DEFERRED
TYPES
    <typeList>
OPERATIONS
     <operationList>
EFFECTIVE
TYPES
    <typeList>
OPERATIONS
    <operationList>
AXIOMS <varList>
    <axiomList>
END-CLASS
```

Las especificaciones incompletas agregan la cláusula **DEFERRED**. La misma declara tipos y operaciones que no están completamente definidos debido a que, o bien no hay suficientes ecuaciones para definir el comportamiento de las nuevas operaciones o, no hay suficientes operaciones para generar todos los valores de un tipo.

La cláusula **EFFECTIVE** agrega tipos y operaciones completamente definidos o completa la definición de algún tipo u operación definido en forma incompleta en alguna superclase. Una operación preexistente cuya definición se completa puede ser declarada en la cláusula **EFFECTIVE** dando sólo su nombre.

Se presentan a continuación ejemplos de especificaciones incompletas.

Ejemplo 4: La clase Recorrible

La clase *Recorrible* especifica en forma incompleta el comportamiento de estructuras que pueden ser recorridas. Independientemente del tipo de estructura que se trate y cualquiera sea la forma de recorrerla, necesitaremos operaciones para acceder a un elemento (*primero*), a la estructura restante (*resto*), y reconocer el final (*fin*). Estas operaciones no pueden definirse en forma completa hasta no saber el tipo de estructura por ejemplo, un árbol o una lista.

```
CLASS Recorrible [Elemento]
IMPORTS  Boolean
DEFERRED
TYPES Recorrible
OPERATIONS
        primero: Recorrible (t) → Elemento
                pre: not fin (t)
        resto: Recorrible (t) → Recorrible
                pre: not fin(t)
        fin: Recorrible ->Boolean
END-CLASS
```

3. Herencia de especificaciones

La cláusula **INHERITS** permite especificar relaciones de herencia entre especificaciones. Se muestra a continuación la sintaxis de una clase NEREUS que incluye a la cláusula **INHERITS.**

CLASS className [<*parameterList*>]
IMPORTS <*importList*>
INHERITS <*inheritList*>
BASIC CONSTRUCTORS
    <*constructorList*>
DEFERRED
TYPES
    <*typeList*>
OPERATIONS
    <*operationList*>
EFFECTIVE
TYPES
    <*typeList*>
OPERATIONS
    <*operationList*>
AXIOMS <*varList*> <*axiomList*>
END-CLASS

NEREUS soporta herencia múltiple. La cláusula **INHERITS** expresa que la clase es construida a partir de la unión de las clases que aparecen en <*inheritList*>. Los componentes de cada una de ellas serán componentes de la nueva clase, y sus propios tipos y operaciones serán tipos y operaciones de la nueva clase.

NEREUS permite definir instancias locales de una clase en las cláusulas **IMPORTS** e **INHERITS** mediante la siguiente sintaxis:

ClassName [<parameterList>] [<bindingList>]

donde los elementos de <*parameterList*> pueden ser pares de nombres de clases *C1:C2*, donde *C2* es una subespecificación importada de *ClassName* y los de <*bindingList*> pares de tipos *s1: s2* y/o pares de operaciones *o1: o2* donde *o2* y *s2* pertenecen a la parte propia de *ClassName*. Se admiten pares de la forma *undefine: o2* indicando que la operación *o2* y los axiomas asociados desaparecen de la especificación. *o2* no puede ser una constructora básica.

El tipo de interés de la clase es también implícitamente renombrado cada vez que la clase es sustituida o renombrada.

Durante el proceso de construcción de una especificación de una clase puede suceder que dos o más tipos o funciones provenientes de especificaciones diferentes tengan el mismo nombre. En NEREUS, dentro de una misma especificación, dos tipos (u operaciones con la misma aridad) se identifican. Si esto no es lo que se busca, NEREUS ofrece el mecanismo de renombre de tipos y operaciones.

Ejemplo 5: La clase PostArbin

La clase *PostArbin* refina *Arbin* (Ej. 3) especificando árboles binarios que se recorren en postorden. Nótese que hereda las operaciones y axiomas de *Arbin* y *Recorrible* (Ej. 4). Las operaciones *primero* y *resto* heredadas de *Recorrible* se definen ahora en forma completa a partir de un conjunto de axiomas. La operación *fin* se define también en forma completa renombrándola por *vacioArbin* que proviene de *Arbin*.
Los tipos *Arbin* y *Recorrible* se renombran por *PostArbin*. Las funciones *primero, resto* y *fin* se declaran efectivas como *PostArbin*. Nótese que sólo basta listar sus nombres y no repetir sus funcionalidades. *Arbin* se instancia con *Elem* y las operaciones *inicArbin* y *crearArbin* son renombradas.

```
CLASS PostArbin [Elem]
INHERITS Arbin [Elem] [inicPostArbin: inicArbin; crearPostArbin: crearArbin],
         Recorrible [Elem]  [vacioArbin:fin]
EFFECTIVE
TYPES PostArbin

OPERATIONS  primero, resto, vacioArbin

AXIOMS t1, t2: PostArbin; x: Elem
    vacioArbin(t1) and vacioArbin(t2)  =>primero (crearPostArbin (t1,t2,x)) = x

    vacioArbin(t1) and (not vacioArbin(t2))  =>
            primero (crearPostArbin (t1,t2,x)) = primero (t2)

    not vacioArbin(t1) => primero (crearPostArbin(t1,t2,x)) = primero (t1)

    vacioArbin(t1) and vacioArbin(t2) =>
            resto(crearPostArbin(t1,t2,x)) = inicPostArbin()

    vacioArbin(t1) and (not vacioArbin(t2))  =>
            resto (crearPostArbin (t1, t2, x)) = crearPostArbin(t1,resto(t2),x)

    not vacioArbin(t1) =>
           resto(crearPostArbin(t1,t2,x)) = crearPostArbin (resto(t1), t2,x)

END-CLASS
```

Se muestra a continuación la expansión de la clase PostArbin:

CLASS PostArbin [Elem]
IMPORTS Boolean
BASIC CONSTRUCTORS inicPostArbin, crearPostArbin
EFFECTIVE
TYPES PostArbin
OPERATIONS
primero: PostArbin (t) → Elem
pre: not vacioArbin(t)
resto: PostArbin (t) → PostArbin
pre: not vacioArbin(t)
inicPostArbin: → PostArbin
crearPostArbin: PostArbin x PostArbin x Elem → PostArbin
vacioArbin: PostArbin -> Boolean
raiz: PostArbin (t) → Elem
pre: not vacioArbin(t)
subIzquierdo: PostArbin (t)→ PostArbin
pre: not vacioArbin(t)
subDerecho: PostArbin (t) → PostArbin
pre: not vacioArbin(t)

AXIOMS t1,t2: PostArbin ; e: Elem
vacioArbin(inicPostArbin) = True
vacioArbin(crearPostArbin(t1,t2,e)) = False
raiz(crearPostArbin(t1,t2,e)) = e
subIzquierdo(crearPostArbin(t1,t2,e)) = t1
subDerecho(crearPostArbin(t1,t2,e)) = t2
vacioArbin(t1) and vacioArbin(t2) => primero (crearPostArbin(t1,t2,x)) = x
vacioArbin(t1) and (not vacioArbin(t2)) =>
primero (crearPostArbin(t1,t2,x)) = primero (t2)
not vacioArbin(t1) => primero (crearPostArbin(t1,t2,x)) = primero (t1)
vacioArbin(t1) and vacioArbin(t2) =>
resto(crearPostArbin(t1,t2,x)) = inicPostArbin
vacioArbin(t1) and (not vacioArbin(t2)) =>
resto (crearPostArbin (t1, t2, x)) = crearPostArbin(t1,resto(t2),x)
not vacioArbin(t1) =>
resto(crearPostArbin(t1,t2,x)) = crearPostArbin (resto(t1), t2,x)
END-CLASS

Referencias

Meyer, Bertrand (1997) Object-Oriented Software Construction, Prentice Hall