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; Chapter 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
TYPE (S)
<*typeList*>
OPERATION(S*)*
<*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
TYPE(S) <*sortList*>
OPERATION(S*)*
<*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** $s_1,s_2,...s_n$ o **TYPE** s, si se declara un sólo tipo

En la cláusula **OPERATION (S)** se declaran las funcionalidades de las operaciones con la sintaxis habitual:
OPERATIONS op1;op2;...opn;. Si se declara sólo una operación la sintaxis es OPERATION op1;.
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 * Boolean →Boolean;
_=_ : Boolean * Boolean -> Boolean;
_or_: Boolean * Boolean →Boolean;
xor: Boolean * Boolean -> Boolean;
*implies*: Boolean * Boolean →Boolean;
*equivalence*: Boolean * Boolean →Boolean;
if_then_else_endif: Boolean * Boolean * Boolean→Boolean;

Para facilitar el uso de formas condicionales la CLASS *Boolean* provee a la operación *if_then_else_endif* y las ecuaciones asociadas son:

if_then_else_ endif (true, x, y) = x
if_then_else_ endif (false, x, y) = y

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.

La clase Boolean está implícitamente importada por todas las especificaciones y no es necesario incluirla en una cláusula IMPORTS.

Se presentan a continuación ejemplos de especificaciones básicas NEREUS.

Ejemplo 1: La clase Pila

CLASS Pila [elem]

BASIC CONSTRUCTORS inicPila, agregarPila

EFFECTIVE

TYPE Pila

OPERATIONS

inicPila: -> Pila;

agregarPila: Pila * elem -> Pila;

vaciaPila: Pila -> Boolean;

tope: Pila(p) -> elem

  pre: not vaciaPila(p);

eliminarPila: Pila (p) -> Pila

  pre: not vaciaPila(p);

AXIOMS p:Pila;e:elem;

tope(agregarPila(p,e))= e;

vaciaPila(inicPila()) = True;

vaciaPila (agregarPila(p,e)) = False;
eliminarPila(agregarPila(p,e))=p;
END-CLASS

Ejemplo 2: La clase Fila

CLASS Fila [Elemento]
BASIC CONSTRUCTORS inicFila, agregarFila
EFFECTIVE
TYPE Fila
OPERATIONS
inicFila: -> Fila;
agregarFila: Fila * 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;
eliminarFila (agregarFila(f, e)) =
if vaciaFila(f)
then inicFila()
else agregarFila (eliminarFila (f), e)
endif;
recuperarFila (agregarFila (f, e)) =
if vaciaFila (f)
then e
else recuperarFila (f)
endif;
END-CLASS

Ejemplo 3: La clase Lista

CLASS Lista [Elemento]
IMPORTS Nat
BASIC CONSTRUCTORS inicLista, agregarLista
EFFECTIVE
TYPE Lista
OPERATIONS
inicLista: -> Lista;
longLista: Lista-> Nat;
agregarLista: Lista (l) * Nat (i) * Elemento -> Lista
  pre: (i >= 1) and (i <= longLista(l)+ 1);
eliminarLista: Lista (l) * Nat (i) -> Lista
  pre: (i >= 1) and (i<= longLista(l));
recuperarLista: Lista (l) * Integer (i) -> Elemento
  pre: (i >= 1) and (i <= longLista(l));
AXIOMS l:Lista; e: Elemento; i,j:Nat;
longLista(inicLista())= 0;
longLista (agregarLista (l,e,i)) = 1 + longLista (l);
(i = j) **implies** (eliminarLista(agregarLista(l,i,e), j) = l);
(i > j) **implies** (eliminarLista (agregarLista(l, i,e), j) =
                agregarLista(eliminarLista(l,j), i-1, e));
(i < j) **implies** (eliminarLista(agregarLista(l,e,i), j) =
                agregarLista(eliminarLista(l,j-1), e,i));
(i = j) **implies** (recuperarLista (agregarLista(l,i, e), j) = e);
(i > j) **implies** (recuperarLista(agregarLista(l, i, e),j) = recuperarLista (l, j));
(i < j) **implies** (recuperarLista(agregarLista(l, i, e), j) = recuperarLista(l, j-1));
END-CLASS

Ejemplo 4: 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]
BASIC CONSTRUCTORS inicArbin, crearArbin
EFFECTIVE
TYPE Arbin
OPERATIONS
inicArbin: -> Arbin;
crearArbin: Arbin * Arbin * 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

Ejemplo 5. La clase Heap
CLASS Heap [Elemento]
BASIC CONSTRUCTORS inicHeap, agregarHeap
EFFECTIVE
TYPE Heap
OPERATIONS
inicHeap: -> Heap;
agregarHeap: Heap * Elemento -> Heap;
vacioHeap: Heap -> Boolean;
raizHeap: Heap(h) -> Elemento
  pre: not vacioHeap(h);

```
eliminarHeap: Heap (h) -> Heap
  pre: not vacioHeap(h);
AXIOMS h:Heap; e:Elemento;
vacioHeap(inicHeap()) = True;
vacioHeap(agregarHeap(h,e)) = False;
vacioHeap(h) implies (raizHeap(agregarHeap(h,e)) = e);
not vacioHeap(h) implies
(raizHeap(agregarHeap(h,e)) =
  if (e <raizHeap(h))
    then e
    else raizHeap(h)
  endif);
vacioHeap(h) implies (eliminarHeap(agregarHeap(h,e)) = inicHeap());
not vacioHeap(h) implies
(eliminarHeap(agregarHeap(h,e)) =
  if (e < raizHeap(h))
   then h
   else agregarHeap(eliminarHeap(h),e)
  endif);
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
TYPE(S)
<*typeList*>
OPERATION(S)
<*operationList*>
EFFECTIVE
TYPE(S)
<*typeList*>

OPERATION(S)
<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 6: 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]
DEFERRED
TYPE Recorrible
OPERATIONS
fin: Recorrible -> Boolean;
primero: Recorrible (t) -> Elemento
  pre: not fin (t);
resto: Recorrible (t) -> Recorrible
  pre: not fin(t);
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 claúsula **INHERITS.**

CLASS className [<*parameterList*>]
IMPORTS <*importList*>
INHERITS <*inheritList*>
BASIC CONSTRUCTORS
<*constructorList*>
DEFERRED
TYPE (S)
<*typeList*>
OPERATION(S)
<*OperationList*>
EFFECTIVE
TYPE(S)
<*typeList*>
OPERATION(S*)*
<*OperationsList*>
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** y **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.*

<*bindingList*> es una lista de renombres separados por comas, precedidos por la palabra clave **rename** de la siguiente forma:

rename nombreOrigen **as** nombreDestino, que indica los renombres de miembros de ClassName para ser utilizados dentro de la clase donde se está instanciando className. Es decir, nombreOrigen corresponde a la parte propia de la clase ClassName, y será referenciado con nombreDestino dentro de la clase que se está definiendo.

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 operationes 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 7: La clase PostArbin

La clase *PostArbin* refina *Arbin* (Ej. 4) especificando árboles binarios que se recorren en postorden. Nótese que hereda las operaciones y axiomas de *Arbin* y *Recorrible* (Ej. 6). 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] rename inicArbin as inicPostArbin, crearArbin
as crearPostArbin, Recorrible [Elem]  rename fin as vacioArbin
EFFECTIVE
TYPE PostArbin
OPERATIONS  primero, resto, vacioArbin;
AXIOMS t1,t2: PostArbin; x : Elem;
vacioArbin(t1) and vacioArbin(t2)  implies
     primero (crearPostArbin (t1,t2,x)) = x;

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

not vacioArbin(t1)  implies
   primero (crearPostArbin(t1,t2,x)) = primero (t1);

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

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

not vacioArbin(t1) implies
   resto(crearPostArbin(t1,t2,x)) = crearPostArbin (resto(t1), t2,x);

END-CLASS
```

Se muestra a continuación la expansión de la clase PostB-Tree:

CLASS PostArbin [Elem]
BASIC CONSTRUCTORS inicPostArbin, crearPostArbin
EFFECTIVE
TYPE PostArbin
OPERATIONS
vacioArbin: PostArbin -> Boolean;
primero: PostArbin (t) → Elem
pre: not vacioArbin(t);
resto: PostArbin (t) → PostArbin
pre: not vacioArbin(t);
inicPostArbin: → PostArbin;
crearPostArbin: PostArbin * PostArbin * Elem → PostArbin;
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(createPostArbin(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) **implies**
primero (crearPostArbin(t1,t2,x)) = x;
vacioArbin(t1) and (not vacioArbin(t2)) **implies**
primero (crearPostArbin(t1,t2,x)) = primero (t2);
not vacioArbin(t1) **implies**
primero (crearPostArbin(t1,t2,x)) = primero (t1);
vacioArbin(t1) and vacioArbin(t2) **implies**
resto(crearPostArbin(t1,t2,x)) = inicPostArbin;
vacioArbin(t1) and (not vacioArbin(t2)) **implies**
resto (crearPostArbin (t1, t2, x)) = crearPostArbin(t1,resto(t2),x);
not vacioArbin(t1) **implies**
resto(crearPostArbin(t1,t2,x)) = crearPostArbin (resto(t1), t2,x)**;**
END-CLASS

Referencias

Meyer, Bertrand (1997) Object-Oriented Software Construction, Capítulo 6, Prentice