

**ESPECIFICACIÓN ALGEBRAICA
DE TIPOS DE DATOS ABSTRACTOS:
EL LENGUAJE NEREUS
2022**

Liliana Favre


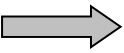
Se describen a continuación las construcciones del lenguaje algebraico *Nereus* utilizado en el curso de “Análisis y diseño de algoritmos 1” para especificar tipos de datos abstractos.

1. ESPECIFICACIONES BÁSICAS

La especificación de un TDA se encapsula dentro de una clase en *Nereus*. A continuación se muestra 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.

<pre>CLASS className [<parameterList>] IMPORTS <importList> BASIC CONSTRUCTORS<constructorList> EFFECTIVE TYPE(S) <sortList> OPERATION(S) <OperationList></pre>		<i>Signatura</i>
<hr style="border-top: 1px dashed black;"/>		
<pre>AXIOMS <varList> <axiomList> END-CLASS</pre>		<i>Axiomas</i>

El encabezamiento de la clase declara el nombre de la clase y puede incluir una lista de parámetros <parameterList>. *Nereus* provee la posibilidad de restringir cada uno de

estos parámetros a un determinado tipo. La lista de parámetros *<parameterList>* está compuesta por pares de la forma *C1:C2* separados por comas, donde *C1* es un parámetro formal genérico restringido por una clase existente *C2*. En particular, *C1:ANY* expresa una parametrización sin restricción de tipos. Este tipo de parametrización sin restricción puede abreviarse simplemente como *C1*. En las secciones 2 y 3 se ampliará la descripción de clases parametrizadas.

La cláusula **IMPORTS** lista a las especificaciones importadas, es decir expresa relaciones cliente (“usa-a”). 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 y operaciones definidos en forma completa. Se ampliará la descripción de esta cláusula en la sección 2.

Una declaración de tipos tiene la forma **TYPES** *s₁,s₂,...s_n* o **TYPE** *s*, si se declara un único tipo.

En la cláusula **OPERATION(S)** se declaran las funcionalidades de las operaciones con la sintaxis **OPERATIONS** *op1;op2;...opn*;. Si se declara una única 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:**”.

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;
_or_: Boolean * Boolean →Boolean;
_xor_: Boolean * Boolean -> Boolean;
_=>_: Boolean * Boolean →Boolean;
_<=>_: Boolean * Boolean →Boolean;

```

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. La

lista de axiomas no puede ser vacía. Si no hubiera axiomas, porque las operaciones son totalmente diferidas, o porque no son necesarios, simplemente se omitirá la cláusula **AXIOMS** por completo.

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, *vacíaLista (inicLista())=True* puede escribirse como *vacíaLista(inicLista())*.

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

Se muestran a continuación algunos axiomas de la clase *Boolean*:

AXIOMS x,y,z: Boolean;

not True = False;

not False = True;

False and x = False;

x and False = False;

True and True = True;

...

END-CLASS

Todas las cláusulas son opcionales y no existe un orden entre ellas. 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:ANY]

BASIC CONSTRUCTORS inicPila, agregarPila

EFFECTIVE

TYPE Pila

OPERATIONS

inicPila: -> Pila;

```

agregarPila: Pila * elem -> Pila;
vacíaPila: Pila -> Boolean;
tope: Pila(p) -> elem
  pre: not vacíaPila(p);
eliminarPila: Pila (p) -> Pila
  pre: not vacíaPila(p);

```

```

AXIOMS p:Pila;e:elem;
tope(agregarPila(p,e))= e;
vacíaPila(inicPila()) = True;
vacíaPila (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;
vacíaFila: Fila -> Boolean;
recuperarFila: Fila(f) -> Elemento
  pre: not vacíaFila (f);
eliminarFila: Fila(f) -> Fila
  pre: not vacíaFila(f);

AXIOMS f: Fila; e: Elemento;
vacíaFila (inicFila() ) =True;
vacíaFila (agregarFila (f,e)) = False;
vacíaFila (f) => eliminarFila (agregarFila(f, e)) = inicFila();
not vacíaFila (f) => eliminarFila(agregarFila (f,e)) = agregarFila (eliminarFila (f), e);
vacíaFila (f) => recuperarFila (agregarFila ( f, e)) = e;
not vacíaFila (f) => recuperarFila (agregarFila ( f, e)) = recuperarFila (f);
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,i,e)) = 1 + longLista (l);

(i == j) => (eliminarLista(agregarLista(l,i,e), j) = l);

(i > j) => (eliminarLista (agregarLista(l, i,e), j) =
agregarLista(eliminarLista(l,j), i-1, e));

(i < j) => (eliminarLista(agregarLista(l,e,i), j) =
agregarLista(eliminarLista(l,j-1), e,i));

(i == j) => (recuperarLista (agregarLista(l,i, e), j) = e);

(i > j) => (recuperarLista(agregarLista(l, i, e),j) = recuperarLista (l, j));

(i < j) => (recuperarLista(agregarLista(l, i, e), j) = recuperarLista(l, j-1));

END-CLASS

Aclaración:

Nat = <S,F> donde S= {Boolean, Nat} y F = {==, <, >, <=, >=, suc, 0, 1,+,*}

El ejemplo 10 incluye una especificación de Nat.

Ejemplo 4: La clase Arbin (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 (modificadoras) *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 de Nat

CLASS Heap
IMPORTS Nat
BASIC CONSTRUCTORS inicHeap, agregarHeap
EFFECTIVE
TYPE Heap
OPERATIONS
inicHeap: -> Heap;
agregarHeap: Heap * Nat -> Heap;
vacioHeap: Heap -> Boolean;
raizHeap: Heap(h) -> Nat
pre: not vacioHeap(h);
eliminarHeap: Heap (h) -> Heap
pre: not vacioHeap(h);

AXIOMS h:Heap; e: Nat;
vacioHeap(inicHeap()) = True;
vacioHeap(agregarHeap(h,e)) = False;
vacioHeap(h) => (raizHeap(agregarHeap(h,e)) = e);
(not vacioHeap(h) and e < raizHeap(h)) =>
raizHeap(agregarHeap(h,e)) = e;
(not vacioHeap(h) and e >= raizHeap(h)) =>
raizHeap(agregarHeap(h,e)) = raizHeap(h);
vacioHeap(h) => eliminarHeap(agregarHeap(h,e)) = inicHeap();
(not vacioHeap(h) and e < raizHeap(h)) =>
eliminarHeap(agregarHeap(h,e)) = h;
(not vacioHeap(h) and e >= raizHeap(h)) =>
eliminarHeap(agregarHeap(h,e)) = agregarHeap(eliminarHeap(h),e);
END-CLASS

Ejemplo 6. La Clase Arbus (árbol binario de búsqueda)

```
CLASS Arbus
IMPORTS Nat
BASIC CONSTRUCTORS inicArbus, agregarArbus
EFFECTIVE
TYPE Arbus
OPERATIONS
inicArbus: - > Arbus;
agregarArbus: Arbus * Nat - > Arbus;
crearArbus: Arbus * Arbus * Nat -> Arbus;
vacioArbus: Arbus -> Boolean;
raiz: Arbus (t) -> Nat
  pre: not vacioArbus(t);
subIzquierdo: Arbus (t) - > Arbus
  pre: not vacioArbus (t);
subDerecho: Arbus (t) -> Arbus
  pre: not vacioArbus (t);
eliminarArbus: Arbus(t) * Nat (n) - > Arbus
  pre: not vacioArbus(t);

AXIOMS b1,b2 :Arbus; i, j, k: Nat;
agregarArbus ( inicArbus ( ), i) = crearArbus (inicArbus(), inicArbus(), i);
(i==j) => ( agregarArbus(crearArbus (b1,b2,j), i) = crearArbus(b1, b2, j));
( i < j) => ( agregarArbus(crearArbus (b1,b2,j), i) =
  crearArbus(agregarArbus(b1,i) b2, j)) );
( i > j) => ( agregarArbus(crearArbus (b1,b2,j), i) =
  crearArbus(b1, agregarArbus(b2,i), j)) );
crearArbus(inicArbus( ), inicArbus ( ), i) = agregarArbus(inicArbus(), i);
crearArbus(inicArbus ( ), agregarArbus(b2,k), i) =
agregarArbus( agregarArbus(b2,k), i);
crearArbus(agregarArbus(b1,j), inicArbus( ), i) = agregarArbus(agregarArbus(b1,j),i);
crearArbus( agregarArbus(b1,j), agregarArbus(b2,k), i) =
agregarArbus(agregarArbus(crearArbus(b1,b2,i),j), k);
(i==j) => eliminarArbus(agregarArbus(b1,i),j) = b1;
((not i==j) and not esvacio(b1) ) =>
eliminarArbus(agregarArbus(b1,i),j) = agregarArbus(eliminarArbus(b1,j), i);
((not i==j) and esvacio(b1)) =>
eliminarArbus(agregarArbus(b1,i),j) = agregarArbus(inicArbus(),i);
vacioArbus( inicArbus ( )) = True;
vacioArbus ( agregarArbus (b1,i)) = False;
subIzquierdo(agregarArbus(inicArbus(),i)) = inicArbus();
subDerecho(agregarArbus(inicArbus(),i)) = inicArbus();
```



```

(not vacioArbus(b1) and (i < raiz(b1)) =>
subIzquierdo (agregarArbus (b1,i)) = agregarArbus(subIzquierdo(b1), i );
(not vacioArbus(b1) and (i >= raiz(b1)) =>
      subIzquierdo (agregarArbus (b1,i)) = subIzquierdo(b1);
(not vacioArbus(b1) and (i > raiz(b1)) ) = >
      subDerecho( agregar Arbus (b1, i) =
      agregarArbus (subDerecho(b1),i);
(not vacioArbus(b1) and (i <= raiz(b1)) ) = >
      subDerecho( agregar Arbus (b1 , i)) = subDerecho (b1) ;
vacioArbus(b1) => raiz(agregarArbus (b1,i)) = i;
not vacioArbus(b1) => raiz(agregarArbus (b1,i)) = raizArbus(b1);
END-CLASS

```

Otra especificación para ARBUS

```

CLASS Arbus
IMPORTS Nat
BASIC CONSTRUCTORS inicArbus, agregarArbus
EFFECTIVE
TYPE Arbus
OPERATIONS
inicArbus: - > Arbus;
agregarArbus: Arbus * Nat - > Arbus;
crearArbus: Arbus * Arbus * Nat -> Arbus;
vacioArbus: Arbus -> Boolean;
raiz: Arbus (t) -> Nat
  pre: not vacioArbus(t);
subIzquierdo: Arbus (t) - > Arbus
  pre: not vacioArbus (t);
subDerecho: Arbus (t) -> Arbus
  pre: not vacioArbus (t);
eliminarArbus: Arbus(t) * Nat (n) - > Arbus
  pre: not vacioArbus(t);

AXIOMS b1,b2 :Arbus; i, j, k: Nat;
agregarArbus ( inicArbus( ), i) = crearArbus (inicArbus(), inicArbus(), i);
(i==j) => ( agregarArbus(crearArbus (b1,b2,j), i) = crearArbus(b1, b2, j));
(i < j) => ( agregarArbus(crearArbus (b1,b2,j), i) =
      crearArbus(agregarArbus(b1,i) b2, j)) );
(i > j) => ( agregarArbus(crearArbus (b1,b2,j), i) =
      crearArbus(b1, agregarArbus(b2,i), j)) );
crearArbus(inicArbus( ), inicArbus ( ), i) = agregarArbus(inicArbus(), i);
crearArbus(inicArbus ( ), agregarArbus(b2,k), i) =
agregarArbus( agregarArbus(b2,k), i);

```

```

crearArbus(agregarArbus(b1,j), inicArbus( ), i) = agregarArbus(agregarArbus(b1,j),i);
crearArbus( agregarArbus(b1,j), agregarArbus(b2,k), i) =
crearArbus(agregarArbus(agregarArbus(b1,j), k), b2, i);
(i==j) => eliminarArbus(agregarArbus(b1,i),j) = b1;
((not i==j) and not esvacio(b1) ) =>
eliminarArbus(agregarArbus(b1,i),j) = agregarArbus(eliminarArbus(b1,j), i);
((not i==j) and esvacio(b1)) =>
eliminarArbus(agregarArbus(b1,i),j) = agregarArbus(inicArbus(),i);
vacioArbus( inicArbus( ) ) = True;
vacioArbus ( agregarArbus (b1,i)) = False;
subIzquierdo(crearArbus(t1,t2,e))= t1;
subDerecho(crearArbus(t1,t2,e) ) = t2;
raiz(crearArbus(t1,t2,e)) =e;
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 8 y Ejemplo 9).

Ejemplo 7: La clase ConOrden

La clase ConOrden especifica cualquier conjunto de valores ordenados.

```
CLASS ConOrden
IMPORTS Boolean
DEFERRED
TYPE ConOrden
OPERATIONS
_==_: ConOrden * ConOrden -> Boolean;
_<=: ConOrden * ConOrden -> Boolean;
EFFECTIVE
OPERATIONS
_>=: ConOrden * ConOrden -> Boolean;
_<_: ConOrden * ConOrden -> Boolean;
_>_: ConOrden * ConOrden -> Boolean;
AXIOMS a,b,c: ConOrden;
a == a;
a <= a;
(a==b) or not (a==b);
(a==b) => (b==a);
(a <=b) or (b<= a);
(a<=b ) and (b<= c) => (a<=c);
(a<=b) and (b<=a) = (a==b);
(a >= b) = (b <= a),
(a > b) = not(a <= b);
(a<b) = (b > a);
END-CLASS
```

Ejemplo 8: 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

```

La cláusula **HIDE**

En una especificación se pueden remover símbolos de la signatura y por ende de los modelos. Nereus provee la cláusula **HIDE** para expresarlo listando los símbolos a ocultarse (tipo, operaciones). Cuando un tipo es removido lo son también las operaciones y axiomas que lo incluyen.

La sintaxis de esta cláusula es la siguiente:

HIDE <hideList> donde <hideList> es una lista de símbolos ocultos

El siguiente ejemplo muestra su uso.

Ejemplo 9. La clase ListaConOrden

La clase *ListaConOrden* especifica listas cuyos elementos están restringidos a elementos con orden. Tiene una operación *ordenar* que permite ordenar ascendentemente los elementos de cualquier lista. Los clientes de esta clase sólo necesitan usar la operación *ordenar* y la operación *insertar* se oculta a través de la cláusula **HIDE**.

```

CLASS ListaConOrden[elem:ConOrden]
IMPORTS Nat
BASIC CONSTRUCTORS inicLista, agregarLista
EFFECTIVE
TYPE ListaConOrden
OPERATIONS
inicLista: -> ListaConOrden;
agregarLista: elem * ListaConOrden -> ListaConOrden; /*agrega al principio*/
longLista: ListaConOrden -> Nat;
eliminarLista: ListaConOrden (l) * Nat (i) -> ListaConOrden
  pre: (i >= 1) and (i <= longLista(l)); /*elimina el i-ésimo elemento*/
recuperarLista: Lista (l) * Nat (i) -> elem
  pre: (i >= 1) and (i <= longLista(l)); /*recupera el i-ésimo elemento*/
ordenar: ListaConOrden -> ListaConOrden; /*ordena la lista*/

```

```

insertar: elem * ListaConOrden -> ListaConOrden; /*insertar con orden*/
AXIOMS s:ListaConOrden; x,y,e: elem; i,j:Nat;
longLista(inicLista())= 0;
longLista (agregarLista (e, s)) = 1 + longLista (s);
(j == 1) => (eliminarLista(agregarLista (e,s), j) = s);
not (j== 1) => (eliminarLista (agregarLista(e, s), j) =
                agregarLista (e, eliminarLista(s, j-1)));
(j == 1) => (recuperarLista (agregarLista (e, s), j) = e);
not (j== 1) => (recuperarLista(agregarLista (e, s),j) = recuperarLista (s, j-1));
ordenar(inicLista())= inicLista();
ordenar(agregarLista(x,s)) = insertar(x,ordenar(s));
insertar(x, inicLista())= agregarLista(x, inicLista());
(x<= y) => (insertar(x, agregarLista(y,s)) =
            agregarLista(x, insertar(y,s)));
(not (x <= y)) => (insertar(x, agregarLista(y, s)) =
            agregarLista(y, insertar(x,s)));
HIDE insertar;
END-CLASS

```

3. HERENCIA DE ESPECIFICACIONES

La cláusula **INHERITS** y la cláusula **IS-SUBTYPE-OF** permiten especificar relaciones de herencia entre especificaciones. Se muestra a continuación la sintaxis de una clase Nereus que las incluye.

```

CLASS className [<parameterList>]
IMPORTS <importList>
INHERITS <inheritList>
IS-SUBTYPE-OF <subtypeList>
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 expresada tanto a partir tanto de la cláusula **INHERITS** como de la cláusula **IS-SUBTYPE-OF**.

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. Es decir pertenecerán a su “parte propia”. El tipo de interés de la clase es también implícitamente renombrado cada vez que la clase es sustituida o renombrada.

La cláusula **INHERITS** soporta herencia enfocada en la reusabilidad de especificaciones. La herencia desde el punto de vista de “comportamiento” se expresa a través de la cláusula **IS-SUBTYPE-OF** donde *<subtypeList>* es una lista no vacía de clases heredadas, separadas por comas. Un concepto relacionado a este tipo de herencia es el polimorfismo donde se satisface la propiedad que una instancia de una clase es a la vez una instancia de la superclase. El casting de un supertipo a un subtipo está implícito. Un término de un subtipo puede ser considerado un término de un supertipo. Por el contrario, el casting de un término del supertipo al subtipo debe explicitarse a partir de la notación *as*. El casting de un término *t* del supertipo a un término *s* del subtipo se denota como *t as s*.

Nereus permite definir instancias locales de una clase en las cláusulas **IMPORTS**, **INHERITS** y **IS-SUBTYPE-OF** 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 forma siguiente

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.

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 10: La clase ConOrdenNatural

```

CLASS ConOrdenNatural
INHERITS ConOrden
BASIC CONSTRUCTORS 0,suc;
EFFECTIVE
TYPE ConOrdenNatural

```

OPERATIONS

```
_==_, _<=_;  
0: -> ConOrdenNatural;  
suc: ConOrdenNatural -> ConOrdenNatural;  
AXIOMS a,b: ConOrdenNatural;  
not (0 == suc(a));  
(suc(a) == suc(b)) = (a == b);  
0 <= suc(a);  
not (suc(a) <= 0);  
(suc (a) <= suc (b)) = (a <= b);  
END-CLASS
```

Ejemplo 11: La clase Nat

```
CLASS Nat  
INHERITS ConOrdenNatural  
EFFECTIVE  
OPERATIONS  
_+_ : Nat * Nat -> Nat,;  
_*_ : Nat * Nat -> Nat;  
1: → Nat;  
  
AXIOMS x,y: Nat;  
x + 0 = x;  
x + y = y + x;  
x + suc(y) = suc(x+y);  
(x + y) + z = x + (y +z);  
x * 0 = 0;  
1 * x = x;  
suc(x) * y = y + (x + y);  
x * y = y *x;  
(x * y) * z = x * (y * z);  
1 = suc (0)  
END-CLASS
```

Ejemplo 12: La clase Heap[Elemento:ConOrden]

Nereus provee la posibilidad de restringir cada uno de los parámetros a un determinado tipo además instanciarse con cualquiera de sus subtipos.

El parámetro *Elemento* de la clase *Heap* restringe la instanciación a la clase *ConOrden* o sus subclases (por ejemplo Nat).

CLASS Heap [Elemento: ConOrden]
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) => (raizHeap(agregarHeap(h,e)) = e);
(not vacioHeap(h) and e < raizHeap(h)) =>
raizHeap(agregarHeap(h,e)) = e;
(not vacioHeap(h) and e >= raizHeap(h)) =>
raizHeap(agregarHeap(h,e)) = raizHeap(h);
vacioHeap(h) => eliminarHeap(agregarHeap(h,e)) = inicHeap();
(not vacioHeap(h) and e < raizHeap(h)) =>
eliminarHeap(agregarHeap(h,e)) = h;
(not vacioHeap(h) and e >= raizHeap(h)) =>
eliminarHeap(agregarHeap(h,e)) = agregarHeap(eliminarHeap(h),e);
END-CLASS

Ejemplo 13: La clase Arbus[Elemento:ConOrden]

CLASS Arbus [Elemento: Con Orden]
BASIC CONSTRUCTORS inicArbus, agregarArbus
EFFECTIVE
TYPE Arbus
OPERATIONS
inicArbus: - > Arbus;
agregarArbus: Arbus * Elemento - > Arbus;
crearArbus: Arbus * Arbus ** Elemento -> Arbus;
vacioArbus: Arbus -> Boolean;
raiz: Arbus (t) -> Elemento
 pre: not vacioArbus(t);
subIzquierdo: Arbus (t) - > Arbus
 pre. not vacioArbus (t);
subDerecho: Arbus (t) -> Arbus

pre: not vacioArbus (t);
eliminarArbus: Arbus(t) * Elemento (e) -> Arbus
pre: not vacioArbus(t);

AXIOMS b1,b2 :Arbus; i, j, k: Nat;
agregarArbus (inicArbus,i) = crearArbus (inicArbus(), inicArbus(), i);
(i==j) => (agregarArbus(crearArbus (b1,b2,j), i) = crearArbus(b1, b2, j));
(i < j) => (agregarArbus(crearArbus (b1,b2,j), i) =
 crearArbus(agregarArbus(b1,i) b2, j)));
(i > j) => (agregarArbus(crearArbus (b1,b2,j), i) =
 crearArbus(b1, agregarArbus(b2,i), j)));
crearArbus(inicArbus(), inicArbus (), i) = agregarArbus(inicArbus(), i);
crearArbus(inicArbus (), agregarArbus(b2,k), i) =
agregarArbus(agregarArbus(b2,k), i);
crearArbus(agregarArbus(b1,j), inicArbus(), i) = agregarArbus(agregarArbus(b1,j),i);
crearArbus(agregarArbus(b1,j), agregarArbus(b2,k), i) =
crearArbus(agregarArbus(agregarArbus(b1,j), k), b2, i);
(i==j) => eliminarArbus(agregarArbus(b1,i),j) = b1;
((not i==j) and not vacioArbus(b1)) =>
eliminarArbus(agregarArbus(b1,i),j) = agregarArbus(eliminarArbus(b1,j), i);
((not i==j) and vacioArbus (b1)) =>
eliminarArbus(agregarArbus(b1,i),j) = agregarArbus(inicArbus(),i);
vacioArbus(inicArbus ()) = True;
vacioArbus (agregarArbus (b1,i)) = False;
subIzquierdo(agregarArbus(inicArbus(),i)) = inicArbus();
subDerecho(agregarArbus(inicArbus(),i)) = inicArbus();

(not vacioArbus(b1) and (i < raiz(b1))) =>
subIzquierdo (agregarArbus (b1,i)) = agregarArbus(subIzquierdo(b1), i);
(not vacioArbus(b1) and (i >= raiz(b1))) =>
 subIzquierdo (agregarArbus (b1,i)) = subIzquierdo(b1);
(not vacioArbus(b1) and (i > raiz(b1))) =>
 subDerecho(agregar Arbus (b1, i) =
 agregarArbus (subDerecho(b1),i);
(not vacioArbus(b1) and (i <= raiz(b1))) =>
 subDerecho(agregar Arbus (b1 , i)) = subDerecho (b1) ;
vacioArbus(b1) => raiz(agregarArbus (b1,i)) = i;
not vacioArbus(b1) => raiz(agregarArbus (b1,i)) = raizArbus(b1);
END-CLASS

Otra forma de especificar Arbus

CLASS Arbus [Elemento: Con Orden]
BASIC CONSTRUCTORS inicArbus, agregarArbus

EFFECTIVE

TYPE Arbus

OPERATIONS

inicArbus: \rightarrow Arbus;

agregarArbus: Arbus * Elemento \rightarrow Arbus;

crearArbus: Arbus * Arbus ** Elemento \rightarrow Arbus;

vacioArbus: Arbus \rightarrow Boolean;

raiz: Arbus (t) \rightarrow Elemento

pre: not vacioArbus(t);

subIzquierdo: Arbus (t) \rightarrow Arbus

pre. not vacioArbus (t);

subDerecho: Arbus (t) \rightarrow Arbus

pre: not vacioArbus (t);

eliminarArbus: Arbus(t) * Elemento (e) \rightarrow Arbus

pre: not vacioArbus(t);

AXIOMS b1,b2 :Arbus; i, j, k: Nat;

agregarArbus (inicArbus,i) = crearArbus (inicArbus(), inicArbus(), i);

(i==j) \Rightarrow (agregarArbus(crearArbus (b1,b2,j), i) = crearArbus(b1, b2, j));

(i < j) \Rightarrow (agregarArbus(crearArbus (b1,b2,j), i) =
crearArbus(agregarArbus(b1,i) b2, j)));

(i > j) \Rightarrow (agregarArbus(crearArbus (b1,b2,j), i) =
crearArbus(b1, agregarArbus(b2,i), j)));

crearArbus(inicArbus(), inicArbus (), i) = agregarArbus(inicArbus(), i);

crearArbus(inicArbus (), agregarArbus(b2,k), i) =

agregarArbus(agregarArbus(b2,k), i);

crearArbus(agregarArbus(b1,j), inicArbus(), i) = agregarArbus(agregarArbus(b1,j),i);

crearArbus(agregarArbus(b1,j), agregarArbus(b2,k), i) =

crearArbus(agregarArbus(agregarArbus(b1,j), k), b2, i);

(i==j) \Rightarrow eliminarArbus(agregarArbus(b1,i),j) = b1;

((not i==j) and not vacioArbus (b1)) \Rightarrow

eliminarArbus(agregarArbus(b1,i),j) = agregarArbus(eliminarArbus(b1,j), i);

((not i==j) and vacioArbus (b1)) \Rightarrow

eliminarArbus(agregarArbus(b1,i),j) = agregarArbus(inicArbus(),i);

vacioArbus(inicArbus ()) = True;

vacioArbus (agregarArbus (b1,i)) = False;

subIzquierdo(crearArbus(t1,t2,e)) = t1;

subDerecho(crearArbus(t1,t2,e)) = t2;

raiz(crearArbus(t1,t2,e)) = e;

END-CLASS

Ejemplo 14: La clase Arbus a partir de una relación de herencia con Arbin

CLASS Arbus [Elemento: Con Orden]

IS-SUBTYPE-OF Arbin[Elemento]

BASIC CONSTRUCTORS inicArbus, agregarArbus

EFFECTIVE

OPERATIONS

agregarArbus: Arbus * Elemento - > Arbus;

eliminarArbus: Arbus(t) * Elemento (e) - > Arbus

pre: not vacioArbin(t);

AXIOMS b1,b2 :Arbus; i, j, k: Elemento;

agregarArbus (inicArbin() as Arbus, i) =

crearArbin (inicArbin(), inicArbin(), i) as Arbus;

(i==j) =>

(agregarArbus(crearArbin (b1,b2,j) as Arbus, i) = crearArbin(b1, b2, j) as Arbus);

(i < j) => (agregarArbus(crearArbin (b1,b2,j), i) as Arbus =

crearArbin(agregarArbus(b1,i) b2, j) as Arbus));

(i > j) => (agregarArbus(crearArbin (b1,b2,j), i) as Arbus =

crearArbin(b1, agregarArbus(b2,i), j) as Arbus));

crearArbin(inicArbin(), inicArbin (), i) as Arbus =

agregarArbus(inicArbin() as Arbus, i);

crearArbin(inicArbin (), agregarArbus(b2,k), i) =

agregarArbus(agregarArbus(b2,k), i);

crearArbin(agregarArbus(b1,j), inicArbin(), i) as Arbus =

agregarArbus(agregarArbus(b1,j),i);

crearArbin(agregarArbus(b1,j), agregarArbus(b2,k), i) as Arbus =

crearArbin(agregarArbus(agregarArbus(b1,j), k), b2, i) as Arbus;

(i==j) => eliminarArbus(agregarArbus(b1,i),j) = b1;

((not i==j) and (not vacioArbin (b1)) =>

eliminarArbus(agregarArbus(b1,i),j) = agregarArbus(eliminarArbus(b1,j), i);

((not i==j) and (vacioArbin(b1)) =>

(eliminarArbus(agregarArbus(b1,i),j) =

agregarArbus(inicArbin() as Arbus,i);

END-CLASS

Ejemplo 15: La clase PostArbin

La clase *PostArbin* refina *Arbin* (Ej. 4) especificando árboles binarios que se recorren en postorden. *PostArbin* hereda las operaciones y axiomas de *Arbin* y *Recorrible* (Ej. 8). 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*. Las funciones

primero, *resto* y *fin* se declaran efectivas como *PostArbin*. Nótese que sólo basta listar sus nombres y no repetir sus funcionalidades.

CLASS PostArbin [Elem]

IS-SUBTYPE-OF Arbin [Elem]; Recorrible [Elem] **rename** fin as vacioArbin

EFFECTIVE

TYPE PostArbin

OPERATIONS primero, resto, vacioArbin;

AXIOMS t1,t2: PostArbin; x : Elem;

vacioArbin(t1) and vacioArbin(t2) => primero (crearArbin (t1,t2,x)) as PostArbin = x;

vacioArbin(t1) and (not vacioArbin(t2)) =>

primero (crearArbin (t1,t2,x) as PostArbin) = primero (t2);

not vacioArbin(t1) => primero (crearArbin(t1,t2,x) as PostArbin) = primero (t1);

vacioArbin(t1) and vacioArbin(t2) =>

resto(crearArbin(t1,t2,x) as PostArbin) = inicArbin() as PostArbin;

vacioArbin(t1) and (not vacioArbin(t2)) =>

resto (crearArbin (t1, t2, x)as PostArbin) = crearArbin(t1,resto(t2),x) as PostArbin;

not vacioArbin(t1) =>

resto(crearArbin(t1,t2,x) as PostArbin) = crearArbin (resto(t1), t2, x) as PostArbin;

END-CLASS