Una notación algorítmica con genericidad y herencia junto con su relación con C++ y Java

Nikos Mylonakis

4 Junio 2003

Resumen

En este artículo se presenta una nueva notación algorítmica que extiende la notación algorítmica que este departamento ha utilizado para impartir los primeros cursos de programación tanto para titulaciones informáticas como no informáticas. La principal aportación de esta nueva notación, es la definición formal de un mecanismo para estructurar los módulos de los programas que incluye genericidad y herencia. Además, se realiza una breve introducción a Java y C++, indicando la relación que existe entre nuestra notación y los lenguajes orientados a objetos mencionados. Para ello, primero se realiza una breve introducción a la programación orientada a objetos y a continuación, para cada uno de los lenguajes se presenta sus mecanismos de estructuración y se traducen los ejemplos utilizados para presentar la notación algorítmica, a estos dos lenguajes.

Índice General

1 Introducción

En este artículo se presenta una nueva notación algorítmica que extiende la notación algorítmica que este departamento ha utilizado para impartir los primeros cursos de programación tanto para titulaciones informáticas como no informáticas. Por tanto, este artículo asume el conocicimiento de los conceptos básicos de la antigua notación. La principal aportación de esta nueva notación es la definición formal de un mecanismo para estructurar los módulos de los programas que incluye genericidad y herencia. Estos mecanismos de estructuración están basados en los utilizados para el lenguaje de especificación Glider ([CJO92]).

La principal motivación de esta artículo se basa en la creencia que los lenguajes orientados a objetos que actualmente se utilizan en la industria como son C++ o Java no son los más adecuados para enseñar a programar en entornos académicos. Según Niklaus Wirth en [Wir02], el lenguaje de programación C confunde a los estudiantes haciéndoles creer que x=y y y=x significan cosas diferentes y además se les obliga a escribir x==y por el usual x=y. Según Wirth, estos pecados son suficientes para que este lenguaje hubiese sido ignorado por instituciones educativas. Por otro lado, me consta que este departamento ha empezado a liberarse de la notación algorítmica con la cual a mí hace ya unos 15 años me enseñaron a programar utilizando diseño modular, enseñando en algunos centros diseño modular en C++ y Java directamente.

Hagamos un poco de historia de cómo fue este proceso. En los años en que yo cursé mis estudios de Informática hace ahora unos 15 años, los lenguajes de programación orientado a objetos no habían alcanzado popularidad y sólo se explicaban como curiosidad en el curso de lenguajes de programación que se podía cursar a partir de tercero. Afortunadamente, yo tuve la posibilidad de impartir la asignatura de lenguajes de programación hace ahora unos 10 años y para entonces ya se explicaban más extensamente estos lenguajes dentro de un capítulo de tipos de datos, pero no se utilizaban estos lenguajes en los cursos básicos de programación. Después de acabar mi tesis doctoral en el año 2000, yo volví a impartir docencia en un curso básico de programación y para entonces los lenguajes de programación orientados a objetos ya se empezaban a enseñar en este tipo de cursos. Actualmente, como ya he dicho anteriormente, en algunos centros adscritos a este departamento se enseña programación directamente en C++ o Java.

Con el fin de frenar esta tendencia, he decidido escribir este artículo que aboga por continuar utilizando notación algorítmica para los cursos básicos de programación utilizando diseño modular, tanto para titulaciones informáticas como no informáticas. Como ya he comentado al principio, la idea básica consistiría en extender nuestra antigua notación algorítmica con genericidad y herencia para explicar diseño modular con estos mecanismos de estructuración y posteriormente dar una idea general de cómo traducir los algoritmos en esta nueva notación a lenguajes orientados a objetos como C++ o Java. Esta antigua notación, que se puede encontrar informalmente explicada en [Fra93], también consta de un nivel de especificación algebraica con ecuaciones que nosotros aquí no incluimos. Por tanto, nuestra notación no tiene un nivel de especificación propiamente dicho, aunque sí que se especifican las funciones y acciones utilizando el estilo pre/post de Hoare, primero en lenguaje natural y a continuación opcionalmente en lógica de Hoare.

Esta fuera del ámbito de este artículo presentar una notación al-

gorítmica con un nivel de especificación, varios níveles de refinamiento si fueran necesarios y un nivel de implementación que además fuese independiente de la institución ([GB92]) o lógica que se utilizase para realizar la especificación algebraica. Considero que un artículo con este contenido es un prerrequisito necesario para integrar la enseñanza de métodos formales en los cursos de ingeniería del software en la titulación de ingeniería informática de nuestra universidad. Trabajo relevante en esta dirección es el llevado a cabo por el proyecto europeo COFI/CASL [CoF98].

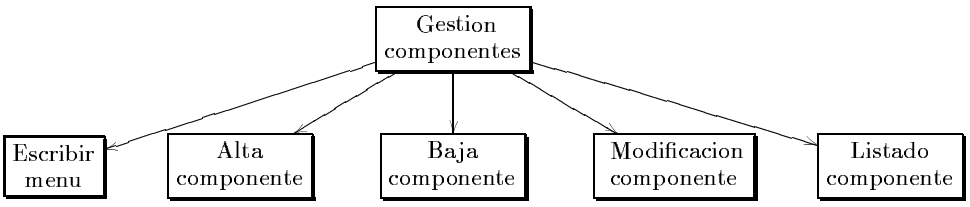
La organización del artículo es la siguiente. Primero presentamos los conceptos básicos generales de diseño modular y a continuación presentamos los conceptos de genericidad y herencia. A continuación presentamos cómo podemos definir módulos y tads en notación algorítmica y más tarde ya damos la semántica operacional del lenguaje de módulos de nuestra notación. Primero presentamos la semántica operacional del lenguaje de módulos con genericidad y herencia simple de forma semiformal y formal y a continuación presentamos la semántica operacional del lenguaje de módulos con genericidad y herencia múltiple sólo formalmente. Finalmente, damos una breve introducción a los lenguajes de programación orientados a objetos y a continuación, presentamos los lenguajes de módulos de Java y C++. Además, indicamos como se definen en estos lenguajes todos los programas presentados en la introducción en notación algorítmica

2 Diseño modular

En un primer curso básico de programación se suele aprender una notación algorítmica para diseñar programas que se vayan a implementar en un lenguaje de programación imperativo. Ejemplos de estos lenguajes son C, Pascal o Ada. A continuación, se suele presentar un método de diseño de programas para resolver problemas de tratamiento secuencial utilizando diseño descendente y esquemas algorítmicos.

Para resolver un problema en subproblemas se utiliza el sentido común, la experiencia y esquemas algorítmicos. Por ejemplo, para resolver el problema de gestionar un almacén de componentes electrónicos podríamos utilizar los subprogramas escribir_menu, alta_componente, baja_componente, modificación_ componente y listado_componente. Estos subprogramas pueden resolver el problema original utilizando un esquema de recorrido.

La representación gráfica del primer nivel del diseño sería la siguiente:



En esta representación gráfica cada nodo representa un sólo algoritmo, subprograma o acción. Los arcos del grafo denotan la relación de uso de un subprograma por otro subprograma o algoritmo.

Para resolver problemas de mayor envergadura se utiliza normalmente una nueva técnica de diseño que se conoce con el nombre de diseño modular. La resolución de un problema mediante diseño modular también

utilizará una representación gráfica similar pero en esta nueva representación, los nodos ya no sólo denotarán un algoritmo o subprograma sino un conjunto de subprogramas. Las técnicas de descomposición también se basarán en el sentido común y la experiencia pero no utilizarán el concepto de esquema. A cambio, el mecanismo de descomposición se basará en la descomposición funcional y la abstracción de datos. Para realizar estos dos tipos de descomposiciones, utilizaremos dos tipos de módulos: los módulos funcionales y los módulos de datos. Un módulo funcional se utilizará para agrupar un conjunto de subprogramas que se utilizan para resolver una misma funcionalidad o prestación que la solución del problema ha de tener. Un módulo de datos (**tad**) se utilizará para definir un tipo de datos con el conjunto de operaciones que se pueden utilizar para inicializar, modificar y actualizar el tipo de datos. Ejemplos de tipos abstractos de datos son los tipos de datos lineales que se utilizan para resolver toda clase de problemas. Estos tipos de datos representan siempre secuencias pero con diferentes políticas de acceso. Ejemplos de estos tipos de datos son las pilas, las colas y las listas. Las pilas permiten modificar una secuencia únicamente por un lado introduciendo y extrayendo elementos por el mismo lado. Las colas permiten modificar una secuencia de elementos por ambos lados, introduciendo elementos siempre por un lado y extrayendo elementos por el otro. Las listas permiten modificar y consultar una secuencia por cualquier posición de la secuencia.

En la siguiente sección veremos en detalle la extensión de la notación algorítmica que proponemos para programar utilizando diseño modular. Esta nueva notación se puede utilizar para diseñar todos los tipos de datos que se pueden ver en un curso avanzado de estructura de datos. Antes de ver la definición general de esta notación, veamos como ejemplo la definición del tad pila de enteros y el tad pila de reales. Estos dos ejemplos nos servirán para introducir el concepto de módulos genéricos.Primero tenemos que asumir predefinidos los tads Booleano, Entero y Real que tendrían la siguiente forma:

tad Booleano
interfaz
sorts
booleano ;
fsorts
ops
cierto: booleano;
falso:booleano
no:booleano
_y_:booleano x booleano → booleano
_o_:booleano x booleano → booleano
_<_:booleano x booleano → booleano
_>_:booleano x booleano → booleano
fops
finterfaz
representacion

⋮

frepresentacion
ftad

tad Entero
interfaz
usa Booleano;
sorts
entero ;
fsorts
ops
$<$ *constante_entera* $>$: entero;
suc:entero $\rightarrow$ entero
pred:entero $\rightarrow$ entero
_+_:entero x entero $\rightarrow$ entero
_-_:entero x entero $\rightarrow$ entero
_*_:entero x entero $\rightarrow$ entero
div:entero x entero $\rightarrow$ entero
mod:entero x entero $\rightarrow$ entero
_=_:entero x entero $\rightarrow$ entero
_<_:entero x entero $\rightarrow$ booleano
_>_:entero x entero $\rightarrow$ booleano
fops
finterfaz
representacion

$\vdots$

frepresentacion
ftad

Utilizamos $<$ *constante_entera* $>$:entero para denotar la signatura infinita $\ldots, -2 : entero, -1 : entero, 0 : entero, 1 : entero, 2 : entero, \ldots$ aunque en nuestra notación los tads definidos por el usuario no pueden tener signatura infinita. Para el caso de los reales procederemos de forma similar:

tad Real
interfaz
usa Booleano;
sorts
real ;
fsorts

ops
$<$ *constante_real* $>$: real
_+_:real x real $\rightarrow$ real
_-_:real x real $\rightarrow$ real
_*_:real x real $\rightarrow$ real
_/_:real x real $\rightarrow$ real
_=_:real x real $\rightarrow$ real
_<_:real x real $\rightarrow$ booleano
_>_:real x real $\rightarrow$ booleano
fops
finterfaz

representacion

⋮

frepresentacion
ftad

A continuación, pasemos a definir el tad pila de enteros:

tad Pila_ent
interfaz
const
MAX:entero = 20
fconst
usa Entero;
sorts
pila_ent;
fsorts
ops
pila_vacia: pila_ent;
Pre:{ ∅ }
Post: { Devuelve la pila vacía }
empilar: pila_ent x entero → pila_ent;
Pre:{ La pila no está llena }
Post: { Empila el entero en la pila }
desempilar: pila_ent → pila_ent;
Pre:{ La pila no está vacía }
Post: { Desempila un entero de la pila }
cumbre: pila_ent → entero;
Pre:{ La pila no está vacía }
Post: {Devuelve el elemento de la cumbre de la pila }
fops
finterfaz
representacion

tipo
tenteros = **tabla** [1..MAX] de entero;
pila_ent = **tupla**
tp:tenteros;
p:entero
ftupla
ftipo
funcion pila_vacia () **retorna** pila_ent
var
pila:pila_ent;
fvar
pila.p:=0;
retorna pila
ffuncion
accion empilar(**ent/sal** pila:pila_ent, **ent** el: entero)
pila.p:=pila.p+1;
pila.tp[pila.p]:=el
faccion

accion desempilar(**ent/sal** pila:pila_ent)
pila.p:=pila.p-1
faccion
funcion cumbre (pila:pila_ent) **retorna** entero
retorna pila.tp[pila.p]
ffuncion
frepresentacion
ftad

En esta definición vemos que el tad pila de enteros usa el tad Enteros. A continuación se define la signatura del tad indicando el sort que se define y sus operaciones. Para cada operación tenemos la aridad de la operación y a continuación su especificación. Posteriormente, tenemos la representación del sort pila_ent mediante una tupla con una tabla y un entero y finalmente tenemos la definición de las operaciones.

El problema de esta definición es que es demasiado concreta. Si quisiésemos definir una pila de reales tendríamos que definir un nuevo tad describiendo todas las operaciones de nuevo:

tad Pila_real
interfaz
const
MAX:entero = 20
fconst
usa Real;
sorts
pila_real;
fsorts
ops
pila_vacia: pila_real;
Pre:{ ∅ }
Post: { Devuelve la pila vacía }
empilar: pila_real x real → pila_real;
Pre:{ La pila no está llena }
Post: { Empila el real en la pila }
desempilar: pila_real → pila_real;
Pre:{ La pila no está vacía }
Post: { Desempila un real de la pila }
cumbre: pila_real → real;
Pre:{ La pila no está vacía }
Post: {Devuelve el real de la cumbre de la pila }
fops
finterfaz

representacion
tipo
treales = **tabla** [1..MAX] de real;
pila_el = **tupla**
tp:treales;
p:entero
ftupla
ftipo

```
  funcion pila_vacia () retorna pila_real
    var
      pila:pila_real;
    fvar
    pila.p:=0;
    retorna pila
  ffuncion
  accion empilar(ent/sal pila:pila_real, ent el: real)
    pila.p:=pila.p+1;
    pila.tp[pila.p]:=el
  faccion
  accion desempilar(ent/sal pila:pila_real)
    pila.p:=pila.p-1
  faccion
  funcion cumbre (pila:pila_real) retorna real
    retorna pila.tp[pila.p]
  ffuncion
 frepresentacion
 ftad
```

Si pudiésemos definir el tad más general pila de elementos, no tendríamos que definir el tad pila de enteros y el tad pila de reales. En vez, tendríamos que definir el tipo pila de elementos e instanciarlo dos veces como el tipo pila de enteros y pila de reales.

Para poder realizar la instanciación, los módulos y los tads podrán tener un parámetro formal que se podrá instanciar mediante un parámetro real de forma similar a los parámetros formales de los subprogramas.

En los tads y módulos sólo hay una clase de parámetros y por tanto no existen parámetros de entrada, salida, entrada/salida. Por otro lado, el concepto de parámetro es más complicado, pues el parámetro de un tad o un módulo, debe ser a su vez un módulo o un tad y el concepto de instanciación también es más complicado que la llamada de un subprograma.

La definición del tad pila de elementos sería así:

```
 tad Pila_el[X:Elem]
 interfaz
   const
     MAX:entero = 20
   fconst
   usa Entero;
   sorts
     pila_el;
   fsorts
```

```
  ops
    pila_vacia: pila_el;
      Pre:{ ∅ }
      Post: { Devuelve la pila vacía }
    empilar: pila_el x elem → pila_el;
      Pre:{ La pila no está llena }
      Post: { Empila el elemento en la pila }
    desempilar: pila_el → pila_el;
      Pre:{ La pila no está vacía }
      Post: { Desempila un elemento de la pila }
    cumbre: pila_el → elem;
      Pre:{ La pila no está vacía }
      Post: {Devuelve el elemento de la cumbre de la pila }
  fops
finterfaz
   representacion

     tipo
       telem = tabla [1..MAX] de elem;
       pila_el = tupla
                          tp:telem;
                          p:entero
                    ftupla
     ftipo
     funcion pila_vacia () retorna pila_el
       var
         pila:pila_el;
       fvar
       pila.p:=0;
       retorna pila
     ffuncion
     accion empilar(ent/sal pila:pila_el, ent el: elem)
       pila.p:=pila.p+1;
       pila.tp[pila.p]:=el
     faccion
     accion desempilar(ent/sal pila:pila_el)
       pila.p:=pila.p-1
     faccion
     funcion cumbre (pila:pila_el) retorna elem
       retorna pila.tp[pila.p]
     ffuncion
   ftad
   frepresentacion
```

Y la definición del tad Elem únicamente constaría de un género:

```
  tad Elem
  interfaz
    sorts
      elem;
    fsorts
  finterfaz
  ftad
```

Y la instanciación de pila de enteros y pila de reales sería así:

tad Pila_ent
es instancia de Pila_el[Entero] **donde** elem es entero
renombra pila_el **por** pila_ent
ftad
tad Pila_real
es instancia de Pila_el[Real] **donde** elem es real
renombra pila_el **por** pila_real
ftad

A continuación introduzcamos el concepto de herencia. Para ello primero definamos un nuevo tipo de datos. Este nuevo tipo de datos permite extraer elementos de una secuencia como las pilas pero por ambos lados de la secuencia. (La implementación que veremos no es la óptima pues con la óptima no es posible utilizar el mecanismo de herencia).

tad Dipila_el[X:Elem]
interfaz
const
MAX:entero=20
fconst
usa Entero;
sorts
dipila_el;
fsorts
ops
dipila_vacia: dipila_el;
Pre:{ $\emptyset$ }
Post: { Devuelve la estructura vacía }
empilar: dipila_el x elem $\rightarrow$ dipila_el;
Pre:{ La estructura no está llena }
Post: { Inserta el elemento por la parte derecha de la estructura }
desempilar: dipila_el $\rightarrow$ dipila_el;
Pre:{ La estructura no está vacía }
Post: { Extrae un elemento de la estructura por la parte derecha }
cumbre: dipila_el $\rightarrow$ elem;
Pre:{ La dipila no está vacía }
Post: {Devuelve el elemento de la parte derecha de la estructura }
insertar_primero: dipila_el x elem $\rightarrow$ dipila_el;
Pre:{ La dipila no está llena }
Post: { Inserta el elemento por la parte izquierda de la estructura }
suprimir_primero: dipila_el $\rightarrow$ dipila_el;
Pre:{ La dipila no está vacía }
Post: { Extrae un elemento de la estructura por la parte izquierda }
fops
finterfaz

representacion
tipo
telem = tabla [1..MAX] de elem;
dipila_el = **tupla**
tp: telem
p,q:entero
ftupla
ftipo

```
funcion dipila_vacia () retorna dipila_el
  var
    dp:dipila_el;
  fvar
  dp.p:= MAX div 2;
  dp.q:= MAX div 2 +1;
  retorna dipila
ffuncion
accion empilar(ent/sal dp:dipila_el, ent el: elem)
  dp.p:=dp.p+1;
  dp.tp[dp.p]:=el
faccion
accion desempilar(ent/sal dp:dipila_el)
  dp.p:=dp.p-1
faccion
funcion cumbre (dp:dipila_el) retorna elem
  retorna dp.tp[dipila.p]
ffuncion
accion insertar_primero(ent/sal dp:dipila_el, ent el: elem)
  dp.q:=dp.q-1;
  dp.tp[dp.q]:=el
faccion
accion suprimir_primero(ent/sal dp:dipila_el)
  dp.q:=dp.q+1
faccion
frepresentacion
ftad
```

Como vemos en esta definición existen tres operaciones que han sido implementadas de la misma forma que en la definición del tipo pila de enteros: empilar, desempilar y cumbre. Mediante una cláusula para la definición de los módulos por herencia, podremos definir por ejemplo el tad dipila de elementos como subclase de pila de elementos. La definición de esta forma permitirá heredar las operaciones definidas en el tad pila de elementos por el tad dipila de elementos de forma apropiada. En el caso en que nos interese cambiar alguna de las implementaciones de las operaciones del tad pila de elementos, bastaría definir de nuevo las operaciones que quisiesemos modificar. En este caso, tendremos que redefinir la operación pila_vacía por una nueva que llamaremos dipila_vacía. Además, podremos extender la representación introduciendo nuevas componentes a la tupla que heredemos de la superclase. Veamos como definir el tad dipila de elementos a partir del tad pila de elementos:

```
tad Dipila_elh[X:Elem]
interfaz
  const
    MAX:entero=20
  fconst
  usa Entero;
  es subclase de Pila_el[X];
  renombra pila_vacia por dipila_vacia;
  genero dipila_elh;
  ops
    insertar_primero: dipila_el x elem → dipila_el;
      Pre:{ La estructura no está llena }
      Post: { Inserta el elemento por la parte izquierda de la dipila }
    suprimir_primero: dipila_el → dipila_el;
      Pre:{ La estructura no está vacía }
      Post: { Extrae un elemento de la pila por la parte izquierda }
  fops

  tipo
    dipila_elh = tupla
                   q:entero
                 ftupla
  ftipo
  funcion dipila_vacia () retorna dipila_el
    var
      dp:dipila_el;
    fvar
    dp.p:= MAX div 2;
    dp.q:= MAX div 2 +1;
    retorna dp
  ffuncion


  accion insertar_primero(ent/sal dp:dipila_el, ent el: elem)
    dp.q:=dp.q-1;
    dp.tp[dp.q]:=el
  faccion
  accion suprimir_primero(ent/sal dp:dipila_el)
    dp.q:=dp.q+1
  faccion
ftad
```

Mediante esta definición a partir del tad Pila_el, la representación del tad Dipila_elh es igual a la de la anterior definición sin herencia, pues ahora la tupla dipila_elh consta de las componentes de pila_el más el nuevo campo q.

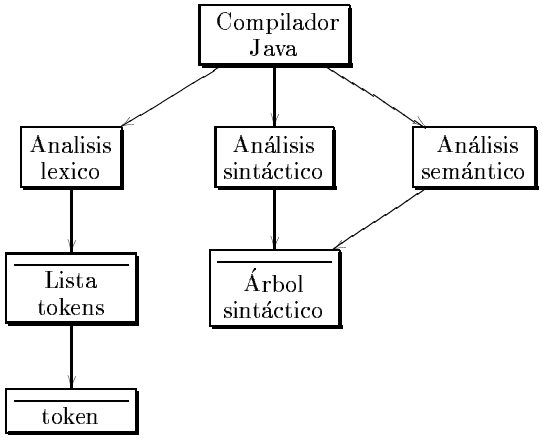
Veamos ahora una descripción general de la genericidad y la herencia que nosotros vamos a definir. La genericidad nos permite parametrizar los módulos por otras componentes que en nuestro caso son a su vez módulos. Esta construcción permite además la instanciación de los módulos genéricos sustituyendo las componentes que hacen de parámetros por componentes concretas. Como en nuestro caso estas componentes son módulos, se puede realizar una correspondencia entre las componentes de los módulos concretos y las componentes de los módulos genéricos.

La herencia es un mecanismo que nos permite definir módulos a partir

de otros. Una forma de hacerlo es estableciendo de forma explícita una relación de subclase entre los módulos que permite heredar operaciones. Un módulo M es subclase de otro módulo M1 si toda operación de M1 se puede utilizar en M. Declarando esta relación de forma explícita, el módulo M hereda automáticamente todas las operaciones de M1. El mecanismo de herencia permite redefinir operaciones heredadas escribiendo el nuevo código de las operaciones en el módulo que hereda.

Para acabar, veamos algunos módulos que podríamos utilizar para diseñar el compilador del lenguaje Java que vamos a describir más adelante. El diseño de un compilador suele descomponerse en los siguientes módulos funcionales: análisis léxico, análisis sintáctico y análisis semántico. En el módulo del análisis léxico se agrupan los subprogramas para transformar el programa en una lista de tokens. Un token es una unidad léxica con significado Ejemplos de tokens serían los identificadores de los módulos o variables, o las palabras claves como **tad**, **subclase** o **instancia**.Este módulo utilizaría el tipo abstracto de datos lista que hemos mencionado anteriormente.

El análisis sintáctico se agrupan los subprogramas que transforma la lista de tokens en un árbol sintáctico. Este árbol sintáctico es una representación del programa que facilita el análisis semántico. Para ello tendríamos que definir el tipo abstracto de datos no lineal árbol sintáctico que no describiremos. Finalmente, el análisis semántico transforma el árbol semántico en un programa equivalente al original en un lenguaje sobre una máquina abstracta con un conjunto de instrucciones determinado. Esta máquina abstracta es mucho más complicada que la máquina asociada a nuestra notación algorítmica y no la veremos en detalle. La representación gráfica del diseño modular de la parte del compilador de Java que hemos visto sería la siguiente:



3 Módulos y tads en notación algorítmica

Veamos una definición intuitiva de un tad y a continuación veremos la definición de un módulo. En la siguiente sección veremos una semántica operacional estática para los módulos:

tad < *nombre* > [< *par_formal* >]
interfaz
  usa < *modulos_y_tads* >
  es subclase de < *expresion_modulo* >
  es instancia de < *expresion_modulo* >
  renombra < *lista_pares* >
  < *signatura* >
finterfaz
representacion
  < *representacion* >
  < *subprogramas* >
frepresentacion
ftad

- El parámetro formal del tad consta de un identificador, el símbolo : y a continuación el nombre del tad o módulo. Por simplicidad, consideraremos que este tad o módulo no puede ser genérico.
- **usa** < *modulos_y_tads* >

  Al incluir el nombre de un módulo o tad en esta lista nos permite utilizar cualquier elemento de su signatura en el tad que estamos definiendo.
- **es instancia de** < *expresion_modulo* >

  El nombre indica el tad a instanciar. La instanciación asocia al parámetro formal del tad a instanciar un módulo o tad concreto. Por ejemplo para instanciar el parámetro formal X de tipo Elem de la pila de elementos utilizamos los tads Entero y Real para realizar dos instanciaciones diferentes.

  Puede haber una lista de pares que asocia a todo sort del tad o módulo del parámetro formal un sort del parámetro real con distinto nombre y lo mismo con las operaciones.

  Al instanciarse un módulo, la aridad de las operaciones cambia, sustituyendo el sort principal del módulo del cual se instancia por el sort principal del módulo que instancia.
- **es subclase de** < *expresion_modulos* >

  Mediante esta cláusula podemos definir un módulo o tad a partir de otros, heredando todas las operaciones definidas y usadas por éstos últimos. En la siguiente sección se da una definición formal de como se heredan estas operaciones.
- **renombra** < *lista_pares* > Esta lista nos permite renombrar los sorts y operaciones de los tads y módulos que se heredan, instancian o usan con nombres cualesquiera siempre que no entren en conflicto con los nombres del tad que se define. Este renombramiento es necesario cuando se realiza más de una instanciación de un tad genérico en un mismo programa.
- < *signatura* > Una signatura consta de un conjunto de tipos y un conjunto de operaciones.

  El conjunto de tipos está precedido por la palabre clave **sorts** y acabado por la palabra clave **fsorts**. Normalmente el conjunto de tipos suele consistir unicamente del nombre del tad en minúsculas pero puede haber otros tipos auxiliares. En los tads es obligatorio que exista un sort principal, con el mismo nombre que el nombre del tad. La diferencia entre estos dos nombres es que el primer carácter

del nombre del tad ha de ser una letra mayúscula, mientras que la primera letra del sort principal ha de ser minúscula.

El conjunto de operaciones está precedido de la palabra clave **ops** y acabado por la palabre clave **fops**. Para cada operación, tenemos que definir su aridad y a continuación su especificación indicando su precondición y postcondición. Toda operación que no aparezca en la signatura pero esté definida dentro de un tad o módulo, no podrá ser ni importada ni heredada por otro módulo.

- $< representacion >$ La representación consiste en una declaración de constantes y una declaración de tipos con la misma sintaxis que en notación algorítmica. Los tipos definen la representación de los sorts. Por ejemplo, el sort pila_ent del tad pila de enteros fue representado mediante una tupla con una tabla y un entero.
- $< subprogramas >$ Aquí se definen las operaciones del tad utilizando como parámetros formales la representación de los sorts del tad.

Todas las cláusulas que aparecen en un tad son opcionales. La utilización de una cláusula implica a veces no poder utilizar otras. Por ejemplo, para realizar una instanciación no se puede definir una nueva signatura, una nueva representación o nuevos subprogramas, y tampoco se puede utilizar la cláusula **es subclase de**. En cambio, para definir un tad por herencia podemos ampliar la signatura, la representación y los subprogamas del tad que heredamos.

Veamos finalmente la definición general de un módulo:

Modulo $< nombre >$ [$< par\_formal >$]
interfaz
  usa $< modulos\_y\_tads >$
  es subclase de $< expresion\_modulo >$
  es instancia de $< expresion\_modulo >$
  renombra $< lista\_pares >$
  $< signatura >$
finterfaz
representacion
  $< subprogramas >$
frepresentacion
fmodulo

Esta definición es similar a la de un tad con la diferencia básica de que no existe representación de sorts pues en las signaturas no puede haber sorts sino sólo operaciones.

4 Semántica operacional de la notación con genericidad y herencia simple

En esta sección vamos a ver como se realiza la comprobación de tipos de una expresión de un tad o de un módulo. Nos centraremos pues en la definición de la relación de subclase a partir de un algoritmo con módulos y cómo se utiliza esta relación para la realización de la comprobación de tipos de una expresión de la forma $f(e_1, \ldots, e_n)$ donde f es una función predefinida. Esta función podrá estar predefinida en el módulo o tad donde se encuentra la expresión o podrá estar definida en otro módulo al que el módulo donde se encuentra la expresion $f(e_1, \ldots, e_n)$ tiene acceso.

Un módulo tiene acceso a las siguientes componentes siempre que exista una definición apropiada en dicho módulo:

1. Un módulo tiene acceso a las componentes declaradas en el mismo módulo.
2. Un módulo tiene acceso a las componentes declarados en el interfaz de su parámetro formal.
3. Un módulo tiene acceso a las componentes de los interfaces de los módulos declarados en la cláusula usa.
4. Un módulo tiene acceso a las componentes del interfaz del módulo instanciado mediante la cláusula **es instancia de**.
5. Un módulo tiene acceso a las componentes heredadas por el interfaz del módulo declarado en la cláusula **es subclase de**.

Para definir esta comprobación de tipos, utilizaremos una definición similar a la dada en [IPW01] para Featherweight Java. Utilizaremos pues un conjunto de reglas para definir la relación de subclase y para la realización de la comprobación de tipos. Intentaremos además explicar informalmente estas reglas. Como en [IPW01], asumiremos predefinida una función que dada un nombre de un tad o un módulo devuelve todo el cuerpo del código o módulo. Veamos primero la sintaxis de los módulos un poco más detallada:

$< Modulo >$ ::= **tad** $< nombre\_m >$ [$< par\_formal >$]
interfaz
usa $< lista\_modulos >$
es subclase de $< exms >$
es instancia de $< exm >$
renombra $< lista\_pares\_ren >$
$< signatura >$
finterfaz
representacion
$< representacion >$
$< subprogramas >$
frepresentacion
ftad |
modulo $< nombre\_m >$ [$< par\_formal >$]
interfaz
usa $< lista\_modulos >$
es subclase de $< exms >$
es instancia de $< exm >$ **donde** $< lista\_pares\_is >$
renombra $< lista\_pares\_ren >$
$< signatura >$
finterfaz
representacion
$< subprogramas >$
frepresentacion
fmodulo

$< par\_formal >$ ::= $< nombre\_pf >$: $< nombre\_m >$

$< exms >$ ::= $< nombre\_m >$ | $< nombre\_m >$[$< nombre\_pf >$] |
$< nombre\_m >$ **donde** $< lista\_pares\_is >$ |
$< nombre\_m >$[$< nombre\_pf >$ **donde** $< lista\_pares\_is >$]

$< lista\_modulos > ::= \quad < nombre\_m > | < nombre\_m > , < lista\_modulos >$

$< lista\_pares\_is > ::= \quad < nombre\_comp >$ **es** $< nombre\_comp > |$
$< nombre\_comp >$ **es** $< nombre\_comp >, < lista\_pares\_is >$

$< lista\_pares\_ren > ::= \quad < nombre\_comp >$ **por** $< nombre\_comp > |$
$< nombre\_comp >$ **por** $< nombre\_comp >, < lista\_pares\_is >$

$< exm > ::= \quad < nombre\_m > | < nombre\_m > [< nombre\_m >]$
$< nombre\_m > [< nombre\_m >$ **donde** $< lista\_pares\_is >]$

$< expresion > ::= \quad < constante > | < variable > |$ f($< lista\_expresiones >$)

$< lista\_expresiones > ::= \quad < expresion > | < expresion >, < lista\_expresiones >$

Nótese que el parámetro formal de los módulos sólo puede tener como tipo asociado un nombre de módulo. Como se ve en la definición sintáctica existen dos clases de expresioness de módulo: $< exm >$ y $< exms >$. Mediante $< exms >$ definimos la cláusula **es subclase de** y mediante $< exm >$ definimos la cláusula **es instancia de**. La expresión de módulo $< exms >$ permite nombres de parámetros formales y módulos como argumentos mientras que $< exm >$ sólo permite nombres de módulo. Éstas son unas decisiones de diseño importante pues simplifican la definición formal considerablemente y la notación algorítmica resultante es suficiente.

Una notación algorítmica más general permitiría que las expresiones de módulos tuviesen como argumentos de nuevo expresiones de módulos y que estas expresiones de módulos también pudiesen ser el tipo de los parámetros formales

Pasemos a definir primero en lenguaje natural las reglas de comprobación de tipos de las expresiones. Las reglas de comprobación de tipos requieren determinar primero la relación de subclase que hay entre los diferentes módulos de un algoritmo. También requieren determinar los tipos posibles de un nombre de función en un módulo determinado. Para poder aplicar correctamente las reglas que describimos a continuación, se tiene que asumir que en el sistema de tads y módulos todos los tads y módulos tienen un nombre distinto y no existe ninguna función f definida más de una vez con la misma aridad.

Las reglas de la definición de subclase son las siguientes:

1. Un sort m es subclase de si mismo.
2. Si M es un tad declarado como subclase de M1 entonces el sort m es subclase de m1.
3. Si M es un tad parametrizado con módulo M1, declarado como subclase del tad N parametrizado con módulo N1 y el sort m1 es subclase del n1, entonces el sort m es subclase del sort n.
4. Si A es un tad que es instancia del tad B con parámetro real C, D es un tad que es instancia del tad E con parámetro real F, el sort e es subclase del sort b y el sort f es subclase del sort c entonces el sort d es subclase del sort a.
5. Si el sort m es subclase de m1 y el sort m1 es subclase del sort m2, entonces el sort m es subclase m2.

Describamos ahora la idea general de las reglas para obtener el tipo de una función f en un módulo M. Para ello tenemos que tener en cuenta los módulos a los que tiene acceso un módulo determinado. Por tanto, estas reglas buscarán primero el tipo de la función en todas las funciones definidas en el módulo, y a continuación en los interfaces del módulo del

parámetro formal, de los módulos de la cláusula usa, de los módulos de la cláusula instancia y del módulo de la cláusula subclase. Para ello, utilizaremos dos grupos de reglas para buscar el tipo de la función. El primero define la función tipoenmc que buscará el tipo de la función en el módulo principal, mientras que el segundo define la función tipoenmc_int que buscará el tipo de la función en el interfaz de los módulos asociados. Este interfaz consta del interfaz del parámetro formal si el módulo es genérico y del interfaz del módulo propiamente dicho. Hay otro grupo de reglas que se tendría que presentar para definir la función tipoenm_int que se diferencia de tipoenmc_int en que no busca el tipo de la función por el parámetro formal. Nosotros sólo describiremos semiformalmente el grupo de reglas que define la función tipoenmc y presentaremos formalmente los grupos de reglas que definen la función tipoenmc y tipoenmc_int. La interpretación semiformal de los grupos de reglas tipoenmc_int y tipoenm_int se deja como ejercicio así como la descripción formal de tipoenm_int.

Describamos semiformalmente las reglas que se requieren para definir la función tipoenmc para obtener el tipo de una función f que se utiliza en un módulo M:

- (1) Si en el módulo M tenemos la función f declarada con tipo $f : T_1$ x ... x $T_n \rightarrow T$, entonces f en el módulo f tiene tipo $f : T_1$ x ... x $T_n \rightarrow T$.
- (2a) Si en el módulo M existe una declaración de la cláusula usa y una lista de renombramientos LP, y f tiene el tipo $f : T_1$ x ... x $T_n \rightarrow T$ en uno de los interfaces de los módulos de esta cláusula, entonces en el tad M f tiene el tipo $f : T_1'$ x ... x $T_n' \rightarrow T'$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar la lista LP a los tipos $T_1, \ldots, T_n, T$.
- (2b) Si en el módulo M existe una declaración de la cláusula usa, g tiene el tipo $g : T_1$ x ... x $T_n \rightarrow T$ en uno de los interfaces de los módulos de la cláusula y M tiene una cláusula renombra donde se renombra la función $g : T_1$ x ... x $T_n \rightarrow T$ por f con la lista LP, entonces en el tad M, f tiene el tipo $f : T_1'$ x ... x $T_n' \rightarrow T'$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar la lista LP a los tipos $T_1, \ldots, T_n, T$.
- (3a) Si el módulo M está parametrizado por el módulo M1 y tiene una lista de renombramientos LP, f tiene el tipo $f : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo M1, entonces en el tad M, f tiene el tipo $f : T_1$ x ... x $T_n \rightarrow T$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar la lista LP a los tipos $T_1, \ldots, T_n, T$.
- (3b) Si el módulo M está parametrizado por el módulo M1, g tiene el tipo $g : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo M1 y M tiene una cláusula renombra donde se renombra la función $g : T_1$ x ... x $T_n \rightarrow T$ por f con la lista LP, entonces en el tad M, f tiene el tipo $f : T_1'$ x ... x $T_n' \rightarrow T'$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar la lista LP a los tipos $T_1, \ldots, T_n, T$.
- (4a) Si en el módulo M existe una declaración de la cláusula instancia con una instanciación del módulo N con parámetro formal MF, g tiene el tipo $g : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo MF , en la lista de asociaciones LP se renombra a $g : T_1$ x ... x $T_n \rightarrow T$ por f y además M tiene una lista de renombramientos LP_1, entonces en el módulo M, f tiene el tipo $f : T_1'$ x ... x $T_n' \rightarrow T'$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar las listas LP_1 y LP a los tipos $T_1, \ldots, T_n, T$.

- (4b) Si en el módulo M existe una declaración de la cláusula instancia con una instanciación del módulo N con parámetro formal MF, g tiene el tipo $g : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo MF , en la lista de asociaciones LP se renombra a $h : T_1$ x ... x $T_n \rightarrow T$ por g y además se renombra a g por f en la lista de renombramientos LP_1, entonces en el módulo M, f tiene el tipo $f : T_1'$ x ... x $T_n' \rightarrow T'$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar las listas LP_1 y LP a los tipos $T_1, \ldots, T_n, T$.
- (4c) Si en el módulo M existe una declaración de la cláusula instancia con una instanciación del módulo N con parámetro real P y lista de asociaciones LP, f no es asociada en LP, M tiene la lista de renombramientos LP_1 y f tiene el tipo $f : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo P, entonces en el módulo M, f tiene el tipo $f : T_1'$ x ... x $T_n' \rightarrow T'$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar la lista LP_1 a los tipos $T_1, \ldots, T_n, T$
- (4d) Si en el módulo M existe una declaración de la cláusula instancia con una instanciación del módulo N con parámetro real P y lista de asociaciones LP, f no es asociada en LP, g tiene el tipo $g : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo P y f renombra a $g : T_1$ x ... x $T_n \rightarrow T$ en LP_1, entonces en el módulo M, f tiene el tipo $f : T_1'$ x ... x $T_n' \rightarrow T'$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar la lista LP_1 a los tipos $T_1, \ldots, T_n, T$.
- (4e) Si en el módulo M existe una declaración de la cláusula instancia con una instanciación del módulo N con parámetro real M_1 y lista de asociaciones LP, M tiene una lista de renombramientos LP_1, f tiene el tipo $f : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo N excluyendo el interfaz del parámetro formal de N, entonces en el módulo M, f tiene el tipo $f : T_1'$ x ... x $T_n' \rightarrow T'$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar la lista LP yLP_1 a los tipos $T_1, \ldots, T_n, T$ y la sustitución del sort n por el sort m.
- (4f) Si en el módulo M existe una declaración de la cláusula instancia con una instanciación del módulo N con parámetro real M_1 y lista de asociaciones LP, g tiene el tipo $g : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo N excluyendo el interfaz del parámetro formal de N y f renombra a $g : T_1$ x ... x $T_n \rightarrow T$ en LP_1, entonces en el módulo M, f tiene el tipo $f : T_1'$ x ... x $T_n' \rightarrow T'$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar la lista LP_1 y LP a los tipos $T_1, \ldots, T_n, T$ y la sustitución del sort n por el sort m.
- (5a) Si en el módulo M existe una declaración de la cláusula es subclase con el módulo N con una lista de renombramientos LP_1, f no está declarada en el módulo M, f no está declarada en la cláusula usa y ninguna función de esta cláusula es renombrada por f, f tiene el tipo $f : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo N excluyendo el interfaz del parámetro formal de N, entonces en el módulo M, f tiene el tipo $f : T_1'$ x ... x $T_n' \rightarrow T'$ donde los tipos $T_1', \ldots, T_n', T'$ se obtienen de aplicar la lista LP y LP_1 a los tipos $T_1, \ldots, T_n, T$.
- (5b) Si en el módulo M existe una declaración de la cláusula es subclase con el módulo N con parámetro real X y lista de asociaciones LP , f no está declarada en el módulo M, f no está declarada en la cláusula usa y ninguna función de esta cláusula es renombrada por f, g tiene el tipo $g : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo N excluyendo el interfaz del parámetro formal de N y f renombra a

$g : T_1$ x ... x $T_n \rightarrow T$ en LP_1 entonces en el módulo M, f tiene el tipo $f : T'_1$ x ... x $T'_n \rightarrow T'$ donde los tipos $T'_1, \ldots, T'_n, T'$ se obtienen de aplicar la lista LP a los tipos $T_1, \ldots, T_n, T$.

- (5c) Si en el módulo M parametrizado por el módulo M1 existe una declaración de la cláusula es subclase con el módulo N con lista de asociaciones LP y una lista de renombramientos LP_1, f no está declarada en el módulo M, f no está declarada en la cláusula usa y ninguna función de esta cláusula es renombrada por f, f tiene el tipo $f : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo M_1 y f está adecuadamente asociada al parámetro formal de N, entonces en el módulo M, f tiene el tipo $f : T'_1$ x ... x $T'_n \rightarrow T'$ donde los tipos $T'_1, \ldots, T'_n, T'$ se obtienen de aplicar la lista LP_1 a los tipos $T_1, \ldots, T_n, T$.
- (5d) Si en el módulo M parametrizado por el módulo M1 existe una declaración de la cláusula es subclase con el módulo N con lista de asociaciones LP y una lista de renombramientos LP_1, g no está declarada en el módulo M, g no está declarada en la cláusula usa y ninguna función de esta cláusula es renombrada por g, g tiene el tipo $f : T_1$ x ... x $T_n \rightarrow T$ en el interfaz del módulo M_1,g está adecuadamente asociada al parámetro formal de N y g es renombrada por f en LP_1, entonces en el módulo M, f tiene el tipo $f : T'_1$ x ... x $T'_n \rightarrow T'$ donde los tipos $T'_1, \ldots, T'_n, T'$ se obtienen de aplicar la lista LP_1 a los tipos $T_1, \ldots, T_n, T$.

(Como ya hemos comentado, queda como ejercicio para el lector interpretar las reglas para obtener el tipo de una función f buscando sólo por el interfaz del módulo).

Y las reglas para obtener el tipo de una expresión son las siguientes:

1. En el módulo M con entorno de variables Γ, la variable x tiene el tipo declarado en Γ
2. Si el tipo de la función f en el módulo M es $f : T_1$ x ... x $T_n \rightarrow T$, e_1 tiene el tipo R_1 en el módulo M y entorno de variables Γ, ..., e_n tiene el tipo R_n en el módulo M y entorno de variables Γ, R_1 es subclase de T_1, ..., R_n es subclase de T_n, entonces en el módulo y entorno de variables Γ, $f(e_1, \ldots, e_n) \in T$.

Nótese que estas reglas, dada una función y un módulo pueden obtener varios tipos asociados a la función. Esto en general no generará un error de tipos. La idea básica es que estas reglas determinan un algoritmo de comprobación de tipos y este algoritmo ha de manejar adecuadamente la sobrecarga. Para ello, al comprobar el tipo de una expresión primero obtendremos el o los tipos de las subexpresiones que en general se almacenarán en una lista. El tipo de una expresión será correcto si tiene un único tipo asociado, pero eso no impide que en el proceso de comprobación del tipo de las subexpresiones éstas tengan varios tipos asociados.

Pasemos ahora a definir formalmente las reglas. Las reglas de la definición de subclase son las siguientes:

(1) $m <: m$

(2)
Modulo(M) =
tad M
$\vdots$
es subclase de M1 **donde** LPAR
$\vdots$
ftad
$\overline{m <: m1}$

(3)
Modulo(M) =
tad M[$X : M1$]
$\vdots$
es subclase de N[X **donde** LPAR]
$\vdots$
ftad
Modulo(N) =
tad N[$Y : N1$]
$\vdots$
ftad
$m1 <: n1$
$\overline{m <: n}$

(4)
Modulo(A) =
tad A
$\vdots$
es instancia de B[C **donde** LP]
$\vdots$
ftad
Modulo(D) =
tad D
$\vdots$
es instancia de E[F **donde** LP1]
$\vdots$
ftad
e <: b
f <: c
$\overline{d <: a}$

(5)
$m1 <: m2$
$m2 <: m3$
$\overline{m1 <: m3}$

Las reglas para realizar la comprobación de tipos de una expresión son las siguientes:

(1) $\quad \mathrm{M},\Gamma \vdash x \in \Gamma(x)$

$$\frac{\begin{array}{l}\text{tipoenmc(M,f)} = T_1 \text{ x } \ldots \text{x } T_n \to T \\ M, \Gamma \vdash e_1 \in R_1 \\ \vdots \\ M, \Gamma \vdash e_n \in R_n \\ R_1 <: T_1 \ldots R_n <: T_n\end{array}}{M, \Gamma \vdash f(e_1, \ldots, e_n) \in T} \quad (2)$$

Veamos ahora como definir las reglas para obtener el tipo de una función f definida por el usuario que se utiliza en el módulo M. Estas reglas definen la función tipoenmc:

$$\frac{\begin{array}{l}\text{Modulo(M)} = \\ \quad \textbf{tad } \text{M}[X : M_1] \\ \quad\quad \textbf{interfaz} \\ \quad\quad \vdots \\ \quad\quad \textbf{finterfaz} \\ \quad\quad \textbf{representacion} \\ \quad\quad \vdots \\ \quad\quad \textbf{funcion } \text{f (y1:T1,}\ldots\text{,yn:Tn) } \textbf{retorna } \text{T} \\ \quad\quad\quad \vdots \\ \quad\quad \textbf{ffuncion} \\ \quad\quad \vdots \\ \quad\quad \textbf{frepresentacion} \\ \quad \textbf{ftad}\end{array}}{tipoenmc(M, f) = T1x \ldots xTn \to T} \quad (1)$$

$$\frac{\begin{array}{l}\text{Modulo(M)} = \\ \quad \textbf{tad } \text{M}[X : M_1] \\ \quad\quad \textbf{usa } \text{LM} \\ \quad\quad \textbf{renombra } \text{LP} \\ \quad\quad \vdots \\ \quad \textbf{ftad} \\ \text{tipoenlmc(LM,f)= T1 x } \ldots \text{ x Tn} \to \text{T}\end{array}}{tipoenmc(M, f) = subst(LP, T1x \ldots xTn \to T)} \quad (2a)$$

Modulo(M) =
tad M $[X : M_1]$
usa LM
renombra LP
$\vdots$
ftad
tipoenlmc(LM,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP) = f
(2b) $\overline{tipoenmc(M, f) = subst(LP, T1x \ldots xTn \rightarrow T)}$

La función subst renombra los nombres de los sorts que aparecen en *LP*.

Modulo(M) =
tad M $[X : M_1]$
$\vdots$
renombra LP
$\vdots$
ftad
tipoenmc_int(M_1,f)= T1 x ... x Tn $\rightarrow$ T
(3a) $\overline{tipoenmc(M, f) = subst(LP, T1x \ldots xTn \rightarrow T)}$

Modulo(M) =
tad M $[X : M_1]$
$\vdots$
renombra LP
$\vdots$
ftad
tipoenmc_int(M_1,f)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP)=f
(3b) $\overline{tipoenmc(M, f) = subst(LP, T1x \ldots xTn \rightarrow T)}$

Modulo(M) =
  tad M
    ⋮
    es instancia de N[P **donde** LP]
    renombra LP_1
    ⋮
  ftad
Modulo(N) =
  tad N[$X : MF$]
    ⋮
  ftad
tipoenmc_int(MF,g)= T1 x ... x Tn → T
esrenombradopor(g:T1 x ... x Tn → T,LP) = f

(4a) $tipoenmc(M, f) = subst(LP_1, subst(LP, T1x \ldots xTn \to T))$

Modulo(M) =
  tad M
    ⋮
    es instancia de N[P **donde** LP]
    renombra LP_1
    ⋮
  ftad
Modulo(N) =
  tad N[$X : MF$]
    ⋮
  ftad
tipoenmc_int(MF,h)= T1 x ... x Tn → T
esrenombradopor(h:T1 x ... x Tn → T,LP) = g
esrenombradopor(g:T1 x ... x Tn → T,LP_1) = f

(4b) $tipoenmc(M, f) = subst(LP_1, subst(LP, T1x \ldots xTn \to T)$

Modulo(M) =
  tad M
    ⋮
    es instancia de N[P **donde** LP]
    renombra LP_1
    ⋮
  ftad
f no está asociada en LP
tipoenmc_int(P,f)= T1 x ... x Tn → T

(4c) $tipoenmc(M, f) = subst(LP_1, T1x \ldots xTn \to T)$

Modulo(M) =
  tad M
    ⋮
    es instancia de N[P **donde** LP]
    renombra LP_1
    ⋮
  ftad
f no está asociada en LP
tipoenmc_int(P,g)= T1 x ... x Tn → T
esrenombradopor(g:T1 x ... x Tn → T,LP_1)=f

(4d) $\overline{tipoenmc(M, f) = subst(LP_1, T1x \ldots xTn \rightarrow T)}$

Modulo(M) =
  tad M
    ⋮
    es instancia de N[M_1 **donde** LP]
    renombra LP_1
    ⋮
  ftad
tipoenm_int(N,f)= T1 x ... x Tn → T

(4e) $\overline{tipoenmc(M, f) = subst(LP_1, subst(LP \cup [m/n], T1x \ldots xTn \rightarrow T))}$

Modulo(M) =
  tad M
    ⋮
    es instancia de N[M_1 **donde** LP]
    renombra LP_1
    ⋮
  ftad
tipoenm_int(N,g)= T1 x ... x Tn → T
esrenombradopor(g:T1 x ... x Tn → T,LP_1) = f

(4f) $\overline{tipoenmc(M, f) = subst(LP_1, subst(LP \cup [m/n], T1x \ldots xTn \rightarrow T))}$

Nota: Estas dos últimas reglas y las dos siguientes son las únicas que utilizan la función tipoenm_int en vez de tipoenmc_int. El motivo es porque en estos dos casos no se ha de buscar la función en el módulo asociado al parámetro formal. Como ya hemos comentado, nosotros sólo definiremos tipoenmc_int y la definición tipoenm_int quedará como ejercicio para el lector teniendo que quitar únicamente las 2 reglas que buscan la función de los parámetros formales de tipoenm_int.

Modulo(M) =
tad M[$X : M_1$]
$\vdots$
es subclase de N[X **donde** LP]
renombra LP_1
$\vdots$
ftad
f no está en subprogramas
f no está en usa renombrado
tipoenm_int(N,f)= T1 x ... x Tn $\rightarrow$ T

(5a) $tipoenmc(M, f) = subst(LP_1, subst(LP, T1x \ldots xTn \rightarrow T))$

Modulo(M) =
tad M[$X : M_1$]
$\vdots$
es subclase de N[X **donde** LP]
renombra LP_1
$\vdots$
ftad
f no está en subprogramas
f no está en usa renombrado
tipoenm_int(N,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP1)=f

(5b) $tipoenmc(M, f) = subst(LP_1, subst(LP, T1x \ldots xTn \rightarrow T)$

Modulo(M) =
tad M[$X : M_1$]
$\vdots$
es subclase de N[X **donde** LP]
renombra LP_1
$\vdots$
ftad
f no está en subprogramas
f no está en usa renombrado
tipoenmc_int(M_1,f)= T1 x ... x Tn $\rightarrow$ T

(5c) $tipoenmc(M, f) = subst(LP_1, T1x \ldots xTn \rightarrow T)$

Modulo(M) =
tad M[$X : M_1$]
$\vdots$
es subclase de N[X **donde** LP]
renombra LP_1
$\vdots$
ftad
f no está en subprogramas
f no está en usa renombrado
tipoenmc_int(M_1,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(g: T1 x ... x Tn $\rightarrow$ T,LP_1) = f

(5d) $tipoenmc(M, f) = subst(LP_1, T1x \ldots xTn \rightarrow T)$

Ahora veamos como obtener el tipo de una función en una lista de módulos de la cláusula usa:

(1) tipoenlmc(M,LM,f) = tipoenmc_int(M,f)

(2) tipoenlmc(M,LM,f) = tipoenlmc(LM,f)

Y ahora veamos como definir las reglas de la función tipoenmc_int:

Modulo(M) =
tad M[$X : M_1$]
interfaz
$\vdots$
ops
$\vdots$
f :T1 $\times$ $\times$ Tn $\rightarrow$ T
{Pre: ...}
{Post: ...}
$\vdots$
fops
$\vdots$
finterfaz
representacion
$\vdots$
frepresentacion
ftad

(1) $tipoenmc\_int(M, f) = T1x \ldots xTn \rightarrow T$

Modulo(M) =
  tad M $[X : M_1]$
    ⋮
    renombra LP
    ⋮
  ftad
tipoenmc_int(M_1,f)= T1 x ... x Tn → T

(3a) $\dfrac{}{tipoenmc\_int(M, f) = subst(LP, T1x \ldots xTn \to T)}$

Modulo(M) =
  tad M $[X : M_1]$
    ⋮
    renombra LP
    ⋮
  ftad
tipoenmc_int(M_1,f)= T1 x ... x Tn → T
esrenombradopor(g:T1 x ... x Tn → T,LP)=f

(3b) $\dfrac{}{tipoenmc\_int(M, f) = subst(LP, T1x \ldots xTn \to T)}$

Modulo(M) =
  tad M
    ⋮
    es instancia de N[P **donde** LP]
    renombra LP_1
    ⋮
  ftad
Modulo(N) =
  tad N[$X : MF$]
    ⋮
  ftad
tipoenmc_int(MF,g)= T1 x ... x Tn → T
esrenombradopor(g:T1 x ... x Tn → T,LP) = f

(4a) $\dfrac{}{tipoenmc\_int(M, f) = subst(LP_1, subst(LP, T1x \ldots xTn \to T))}$

Modulo(M) =
tad M
$\vdots$
es instancia de N[P **donde** LP]
renombra LP_1
$\vdots$
ftad
Modulo(N) =
tad N[$X : MF$]
$\vdots$
ftad
tipoenmc_int(MF,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(h:T1 x ... x Tn $\rightarrow$ T,LP) = g
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP_1) = f

(4b) $tipoenmc\_int(M, f) = subst(LP_1, subst(LP, T1x \ldots xTn \rightarrow T))$

Modulo(M) =
tad M
$\vdots$
es instancia de N[P **donde** LP]
renombra LP_1
$\vdots$
ftad
f no está asociada en LP
tipoenmc_int(P,f)= T1 x ... x Tn $\rightarrow$ T

(4c) $tipoenmc\_int(M, f) = subst(LP_1, T1x \ldots xTn \rightarrow T)$

Modulo(M) =
tad M
$\vdots$
es instancia de N[P **donde** LP]
renombra LP_1
$\vdots$
ftad
f no está asociada en LP
tipoenmc_int(P,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP_1)=f

(4d) $tipoenmc\_int(M, f) = subst(LP_1, T1x \ldots xTn \rightarrow T)$

Modulo(M) =
  tad M
    ⋮
    es instancia de N[M_1 **donde** LP]
    renombra LP_1
    ⋮
  ftad
tipoenm_int(N,f)= T1 x ... x Tn → T

(4e) $$\frac{}{tipoenmc\_int(M,f) = subst(LP_1, subst(LP \cup [m/n], T1x \ldots xTn \to T))}$$

Modulo(M) =
  tad M
    ⋮
    es instancia de N[M_1 **donde** LP]
    renombra LP_1
    ⋮
  ftad
tipoenm_int(N,g)= T1 x ... x Tn → T
esrenombradopor(g:T1 x ... x Tn → T,LP_1) = f

(4f) $$\frac{}{tipoenmc\_int(M,f) = subst(LP_1, subst(LP \cup [m/n], T1x \ldots xTn \to T))}$$

Modulo(M) =
  tad M[$X : M_1$]
    ⋮
    es subclase de N[X **donde** LP]
    renombra LP_1
    ⋮
  ftad
f no está en la signatura del interfaz
tipoenm_int(N,f)= T1 x ... x Tn → T

(5a) $$\frac{}{tipoenmc\_int(M,f) = subst(LP_1, subst(LP, T1x \ldots xTn \to T))}$$

Modulo(M) =
  tad M[$X : M_1$]
    ⋮
    es subclase de N[X **donde** LP]
    renombra LP_1
    ⋮
  ftad
f no está en la signatura del interfaz
tipoenm_int(N,g)= T1 x ... x Tn → T
esrenombradopor(g:T1 x ... x Tn → T,LP1)=f

(5b) $$\frac{}{tipoenmc\_int(M,f) = subst(LP_1, subst(LP, T1x \ldots xTn \to T))}$$

Modulo(M) =
tad M[$X : M_1$]
$\vdots$
es subclase de N[X **donde** LP]
renombra LP_1
$\vdots$
ftad
f no está en la signatura del interfaz
tipoenmc_int(M_1,f)= T1 x ... x Tn $\rightarrow$ T

(5c) $$\frac{}{tipoenmc\_int(M, f) = subst(LP_1, T1x \ldots xTn \rightarrow T)}$$

Modulo(M) =
tad M[$X : M_1$]
$\vdots$
es subclase de N[X **donde** LP]
renombra LP_1
$\vdots$
ftad
f no está en la signatura del interfaz
tipoenmc_int(M_1,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP1)=f

(5d) $$\frac{}{tipoenmc\_int(M, f) = subst(LP_1, T1x \ldots xTn \rightarrow T}$$

5 Semántica operacional de la notación con genericidad y herencia múltiple

En esta sección vamos a ver como extender nuestra notación con genericidad y herencia múltiple. La idea básica es que permitiremos que un tad o un módulo tengan varios parámetros formales y que puedan heredar de varios módulos. La comprobación de tipos es muy similar pues la herencia múltiple no aporta complicaciones pues asumimos que el algoritmo de comprobación de tipos maneja la sobrecarga. Un caso que merece ser mencionado es cuando una clase D hereda una función f de una misma clase A vía dos clases B y C. Nosotros consideraremos que el mecanismo de sobrecarga no permite heredar una misma función de dos vías diferentes. Esto no generará un error y se considerará únicamente heredada por la primera vía declarada. Para ello es conveniente asumir que en nuestro sistema de tads y módulos no existen dos funciones declaradas con el mismo nombre y con el mismo tipo como ya hemos mencionado.

Veamos pues la nueva sintaxis de nuestra notación y a continuación las reglas para la comprobación de tipos:

$< Modulo >$ ::= **tad** $< nombre\_m >$ [$< par\_formales >$]
interfaz
usa $< lista\_modulos >$
es subclase de $< lista\_exms >$
es instancia de $< exm >$ **donde** $< lista\_pares\_is >$
renombra $< lista\_pares\_ren >$
$< signatura >$
finterfaz
representacion
$< representacion >$
$< subprogramas >$
frepresentacion
ftad |
modulo $< nombre\_m >$ [$< par\_formales >$]
interfaz
usa $< lista\_modulos >$
es subclase de $< lista\_exms >$
es instancia de $< exm >$ **donde** $< lista\_pares\_is >$
renombra $< lista\_pares\_ren >$
$< signatura >$
finterfaz
representacion
$< subprogramas >$
frepresentacion
fmodulo

$< par\_formales >$ ::= $< nombre\_pf >$: $< nombre\_m >$
$< nombre\_pf >$: $< nombre\_m >$, $< par\_formales >$

$< lista\_exms >$ ::= $< exms >$ | $< exms >$, $< lista\_exms >$
$< exms >$ ::= $< nombre\_m >$ | $< nombre\_m >$[$< lista\_parr >$]
$< lista\_parr >$ ::= $< parr >$ | $< parr >$, $< lista\_parr >$
$< parr >$ ::= $< nombre\_pf >$ | $< nombre\_pf >$ **donde** $< lista\_pares\_is >$ |
$< nombre\_m >$ | $< nombre\_m >$ **donde** $< lista\_pares\_is >$

$< lista\_modulos >$ ::= $< nombre\_m >$ | $< nombre\_m >$, $< lista\_modulos >$
$< lista\_pares\_is >$ ::= $< nombre\_comp >$ **es** $< nombre\_comp >$ |
$< nombre\_comp >$ **es** $< nombre\_comp >$,$< lista\_pares\_is >$

$< lista\_pares\_ren >$ ::= $< nombre\_comp >$ **por** $< nombre\_comp >$ |
$< nombre\_comp >$ **por** $< nombre\_comp >$,$< lista\_pares\_is >$

$< exm >$ ::= $< nombre\_m >$ | $< nombre\_m >$[$< lista\_modmor >$]
$< lista\_modmor >$ ::= $< nombre\_m >$ | $< nombre\_m >$ **donde** $< lista\_pares\_is >$
$< nombre\_m >$, $< lista\_modmor >$

$< expresion >$ ::= $< constante >$ | $< variable >$ | f($< lista\_expresiones >$)
$< lista\_expresiones >$ ::= $< expresion >$ | $< expresion >$,$< lista\_expresiones >$

Las reglas de la definición de subclase son las siguientes:

(1) m <: m

Modulo(M) =
tad M
⋮
es subclase de M1,..., MN
⋮
ftad

(2) $m <: m1$ **y** ... **y** $m <: mn$

Modulo(M) =
tad M[$X_1 : M_1$,...,$X_n : M_n$]
⋮
es subclase de S1[X_1 **donde** LP11,..., X_n **donde** LP1n],
..., Sj[X1 **donde** LPj1, ..., X_n **donde** LPjn],
..., Sm[X1 **donde** LPm1, ..., X_n **donde** LPmn]
⋮
ftad

Modulo(Mj) =
tad Sj[$Y_1 : Nj1$,...,$Y_n : Njn$]
⋮
ftad

m1 <: nj1
⋮

(3) $\dfrac{mn <: njn}{m <: mj}$

Modulo(A) =
tad A
$\vdots$
es instancia de B[C1 **donde** LP11, ..., CN **donde** LP1n]
$\vdots$
ftad
Modulo(D) =
tad D
$\vdots$
es instancia de E[F1 **donde** LP21, ..., FN **donde** LP2n]
$\vdots$
ftad
e <: b
f1 <: c1
$\vdots$
fn <: cn
$d <: a$ (4)

$m1 <: m2$
$m2 <: m3$
$m1 <: m3$ (5)

Las reglas para realizar la comprobación de tipos de una expresión son las siguientes:

(1) M,$\Gamma \vdash x \in \Gamma(x)$

tipoenmc(M,f) = T_1 x ... x $T_n \to T$
$M, \Gamma \vdash e_1 \in R_1 \ldots M, \Gamma \vdash e_n \in R_n$
$R_1 <: T_1 \ldots R_n <: T_n$
$M, \Gamma \vdash f(e_1, \ldots, e_n) \in T$ (2)

Veamos ahora como definir las reglas para obtener el tipo de una función f definida por el usuario que se utiliza en el módulo M:

Modulo(M) =
tad M[$X_1 : M_1, \ldots, X_N : M_N$]
$\vdots$
funcion f (y1:T1,...,yn:Tn) **retorna** T
$\vdots$
ffuncion
$\vdots$
ftad

(1) $\dfrac{}{tipoenmc(M, f) = T1x \ldots xTn \to T}$

Modulo(M) =
tad M[$X_1 : M_1, \ldots, X_N : M_N$]
usa LM
renombra LP
$\vdots$
ftad

(2a) $\dfrac{\text{tipoenlmc(LM,f)= T1 x ... x Tn} \to \text{T}}{tipoenmc(M, f) = subst(LP, T1 \ldots xTn \to T)}$

Modulo(M) =
tad M [$X_1 : M_1, \ldots, X_N : M_N$]
usa LM
renombra LP
$\vdots$
ftad

(2b) $\dfrac{\text{tipoenlmc(M,g)= T1 x ... x Tn} \to \text{T} \quad \text{esrenombradopor(g:T1 x ... x Tn} \to \text{T,LP)} = \text{f}}{tipoenmc(M, f) = subst(LP, T1x \ldots xTn \to T)}$

Modulo(M) =
tad M [$X_1 : M_1, \ldots, X_N : M_N$]
$\vdots$
renombra LP
$\vdots$
ftad

(3a) $\dfrac{\text{tipoenlmc}(\{M_1, \ldots, M_N\},\text{f})= \text{T1 x ... x Tn} \to \text{T}}{tipoenmc(M, f) = subst(LP, T1x \ldots xTn \to T)}$

Modulo(M) =
tad M $[X_1 : M_1, \ldots, X_N : M_N]$
$\vdots$
renombra LP
$\vdots$
ftad
tipoenlmc($\{M_1, \ldots, M_N\}$,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP)=f

(3b) $\quad tipoenmc(M, f) = subst(LP, T1x \ldots xTn \rightarrow T)$

Modulo(M) =
tad M
$\vdots$
es instancia de N[M_1 **donde** $LP_1, \ldots, M_N$ **donde** LP_N]
renombra LP_r
$\vdots$
ftad
Modulo(M) =
tad N$[X_1 : MF_1, \ldots, X_N : MF_N]$
$\vdots$
ftad
tipoenmc_int(MF_i,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP_i) = f

(4a) $\quad tipoenmc(M, f) = subst(LP_r, subst(LP_i, T1x \ldots xTn \rightarrow T))$

Modulo(M) =
tad M
$\vdots$
es instancia de N[M_1 **donde** $LP_1, \ldots, M_N$ **donde** LP_N]
renombra LP_r
$\vdots$
ftad
Modulo(M) =
tad N$[X_1 : MF_1, \ldots, X_N : MF_N]$
$\vdots$
ftad
tipoenmc_int(MF_i,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(h:T1 x ... x Tn $\rightarrow$ T,LP_i) = g
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP_r) = f

(4b) $\quad tipoenmc(M, f) = subst(LP_r, subst(LP_i, T1x \ldots xTn \rightarrow T))$

Modulo(M) =
tad M
$\vdots$
es instancia de N[M_1 **donde** $LP_1, \ldots, M_N$ **donde** LP_N]
renombra LP_r
$\vdots$
ftad
f no es renombrada en LP_i
tipoenmc_int(M_i,f)= T1 x ... x Tn $\rightarrow$ T

(4c) $\dfrac{}{tipoenmc(M, f) = subst(LP_r, T1x \ldots xTn \rightarrow T)}$

Modulo(M) =
tad M
$\vdots$
es instancia de N[M_1 **donde** $LP_1, \ldots, M_N$ **donde** LP_N]
renombra LP_r
$\vdots$
ftad
f no es renombrada en LP_i
tipoenmc_int(M_i,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP_r) = f

(4d) $\dfrac{}{tipoenmc(M, f) = subst(LP_r, T1x \ldots xTn \rightarrow T)}$

Modulo(M) =
tad M
$\vdots$
es instancia de N[M_1 **donde** $LP_1, \ldots, M_N$ **donde** LP_N]
renombra LP_r
$\vdots$
ftad
tipoenm_int(N,f)= T1 x ... x Tn $\rightarrow$ T

(4e) $\dfrac{}{tipoenmc(M, f) = subst(LP_r, subst(LP_1, \ldots, subst(LP_n \cup [m/n], T1x \ldots xTn \rightarrow T) \ldots)}$

Modulo(M) =
tad M
$\vdots$
es instancia de N[M_1 **donde** $LP_1, \ldots, M_N$ **donde** LP_N]
renombra LP_r
$\vdots$
ftad
tipoenm_int(N,g)= T1 x ... x Tn $\rightarrow$ T
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP_r) = f

(4f) $\dfrac{}{tipoenmc(M, f) = subst(LP_r, subst(LP_1, \ldots, subst(LP_n \cup [m/n], T1x \ldots xTn \rightarrow T) \ldots)}$

Modulo(M) =
tad M[$X_1 : M_1, \ldots, X_n : M_n$]
$\vdots$
es subclase de EXM1,...,M_i[X_1 **donde** $LP_{i1}, \ldots,$ X_n **donde** LP_{in}],
$\ldots, EXM_l$,
renombra LP_r
$\vdots$
ftad
f no está en subprogramas
f no está en usa renombrado
tipoenm_int(M_i,f)= T1 x ... x Tn → T

(5a) $tipoenmc(M, f) = subst(LP_r, subst(LP_{i1}, \ldots, subst(LP_{in}, T1x \ldots xTn \to T) \ldots)$

Modulo(M) =
tad M[$X_1 : M_1, \ldots, X_n : M_n$]
$\vdots$
es subclase de EXM1,...,M_i[X_1 **donde** $LP_{i1}, \ldots,$ X_n **donde** LP_{in}],
$\ldots, EXM_l$,
renombra LP_r
$\vdots$
ftad
f no está en subprogramas
f no está en usa renombrado
tipoenm_int(M_i,g)= T1 x ... x Tn → T
esrenombradopor(g:T1 x ... x Tn → T,LP) = f

(5b) $tipoenmc(M, f) = subst(LP_r, subst(LP_{i1}, \ldots, subst(LP_{in}, T1x \ldots xTn \to T) \ldots)$

Modulo(M) =
tad M[$X_1 : M_1, \ldots, X_N : M_N$]
$\vdots$
es subclase de EXM1,...,M_i[X_1 **donde** $LP_{i1}, \ldots,$ X_n **donde** LP_{ij_i}],
$\ldots, EXM_l$,
renombra LP_r
$\vdots$
ftad
f no está en subprogramas
f no está en usa renombrado
tipoenmc_int(M_{ik},f)= T1 x ... x Tn → T ($k \in [1..j_i]$)

(5c) $tipoenmc(M, f) = subst(LP_r, T1x \ldots xTn \to T)$

Modulo(M) =
tad M[$X_1 : M_1, \ldots, X_N : M_N$]
$\vdots$
es subclase de EXM1,...,M_i[X_1 **donde** $LP_{i1}, \ldots, X_n$ **donde** LP_{ij_i}],
$\ldots, EXM_l$,
renombra LP_r
$\vdots$
ftad
f no está en subprogramas
f no está en usa renombrado
esrenombradopor(g:T1 x ... x Tn $\rightarrow$ T,LP_r)=f
tipoenmc_int(M_{ik},f)= T1 x ... x Tn $\rightarrow$ T ($k \in [1..j_i]$)

(5d) $tipoenmc(M, f) = subst(LP_r, T1x \ldots xTn \rightarrow T)$

Ahora veamos como obtener el tipo de una función en una lista de módulos de la cláusula usa:

(1) tipoenlmc(M,LM,f) = tipoenmc_int(M,f)

(2) tipoenlmc(M,LM,f) = tipoenlmc(LM,f)

Y las reglas de la función tipoenmc_int y tipoenm_int quedan como ejercicio.

6 Notación algorítmica y lenguajes de programación OO

6.1 Introducción

Al definir la notación algorítmica, se suele definir el concepto de objeto como una entidad formada por un nombre, un tipo y un valor. En esta notación los objetos podían ser constantes y variables. En programación orientada a objetos, el concepto de objetos es distinto. Conceptualmente, un objeto está formado por un estado y un comportamiento. El estado queda definido por un conjunto de variables y el comportamiento queda definido por un conjunto de métodos que se pueden ver como acciones y funciones. Al desarrollar un programa utilizando estos lenguajes, tendremos que pensar en los objetos que requeriremos.

Para poder definir objetos, necesitamos algo similar al concepto de tad o módulo que hemos visto. Necesitamos, por tanto un mecanismo de encapsulación para que el estado de un objeto sólo se pueda modificar mediante los métodos que definen su comportamiento. Este mecanismo de encapsulación en lenguajes de programación orientados a objetos se le suele denominar clase.

Una clase es una construcción que define las variables del estado de un objeto y sus métodos. Por ejemplo, para definir la clase de puntos

de un plano, tendremos que definir dos variables reales. Adicionalmente, tenemos que definir los métodos que podemos utilizar para actualizar los estados de los objetos de esta clase.

La definición de la clase Punto_plano en Java sería la siguiente:

```
class Punto_plano extends Object {

  private double x;
  private double y;

  Punto_plano() {
    x=0.0;
    y=0.0;
  }

  public double coordX() {
    return x;
  }

  public double coordY() {
    return y;
  }
  public actX ( double x) {
    this.x=x;
  }
  public actY ( double y) {
    this.y=y;
  }
}
public class PointInstance {

  public static void main(String[] args) {
    Punto_plano mypoint;

    mypoint = new Punto_plano();

    mypoint.actX(12.0);
    mypoint.actY(13.0);
    System.out.println(mypoint.getX());
    System.out.println(mypoint.getY());
  }
}
```

En esta definición tenemos primeros las dos variables de tipo real de doble precisión y a continuación los métodos. El primer método se utiliza para construir un punto y las siguientes operaciones se utilizan para consultar o modificar los diferentes campos del punto. Analicemos un poco la definición de los métodos coordX y actX. Aparentemente coordX no tiene ningún parámetro pero en el cuerpo utiliza la variable x. Esto es posible pues las variables de la clase son visibles dentro del cuerpo de los métodos. Este método también lo podríamos haber definido de la siguiente manera:

```
  public double coordX() {
    return this.x;
  }
```

Otra forma más adecuada de ver la definición de un método es como una acción que además de los parámetros definidos (en este caso ninguno) tenemos un parámetro implícito con nombre **this**. El tipo del parámetro es el de la clase que se define y se puede ver como un tipo tupla con tantos campos como variables de clase haya. En este caso los campos de la variable this son dos con nombre x e y. Por tanto, las expresiones this.x y this.y son correctas. En el método actX tenemos un parámetro x de tipo real de doble precisión. Por lo tanto, dentro del cuerpo de la acción actX, x denota el parámetro explícito y para denotar la variable de la clase x estamos forzados a utilizar la expresión this.x. Pasemos ahora a analizar la siguiente clase, donde se realizan llamadas a los métodos de la clase Punto_plano. Para ello tenemos que declarar un objeto de la clase Punto_plano. Este proceso se hace en dos pasos que ya detallaremos más adelante. La llamada a un método se realiza de la siguiente forma:

$$< nombre\_objeto > . < metodo\_clase > (< par\_reales >)$$

En $< nombre\_objeto >$ tenemos que indicar el parámetro real del parámetro formal implícito this, y el resto de los parámetros reales son los parámetros reales de los parámetros formales. Estos conceptos son comunes en Java y C++ y es un mecanismo de definición y llamada de los métodos útil para razonar sobre sistemas concurrentes. La idea intuitiva subyacente es ver la llamada de un método como un paso de mensajes entre dos objetos que se ejecutan en paralelo. Un objeto A envía un mensaje a un objeto B para que el objeto B realice una tarea que necesita el objeto A, utilizando uno de los métodos del objeto B. Esta forma de pensar, facilita el diseño e implementación de aplicaciones que están distribuidas en una red local o en Internet.

Otro concepto que se suele asociar a los lenguajes de programación orientados a objetos es el concepto de herencia. Este concepto es muy similar al concepto de herencia explicado en la presentación de nuestra notación algorítmica. Recordamos que la herencia es un mecanismo que nos permite definir módulos a partir de otros. En los lenguajes que veremos, la forma de hacerlo es estableciendo de forma explícita una relación de subclase entre los módulos que permite heredar operaciones. Un módulo M es subclase de otro módulo M1 si toda operación de M1 se puede utilizar en M. Declarando esta relación de forma explícita, el módulo M hereda automáticamente todas las operaciones de M1. El mecanismo de herencia permite redefinir operaciones heredadas escribiendo el nuevo código de las operaciones en el módulo que hereda.

6.2 Java

6.2.1 Objetos en Java

Como ya hemos comentado en la introducción, un objeto es un conjunto de variables con un conjunto de métodos asociados. Las variables determinan el estado del objeto y los métodos su comportamiento, determinando cómo se pueden inicializar, actualizar y consultar el conjunto de variables de un objeto concreto.

Para poder definir un objeto en Java, tenemos que definir primero una clase. En una clase tenemos que determinar el tipo de cada una de las variables de los objetos de esa clase y la definición de todos los métodos asociados a esa clase de objetos.

Así, en la introducción vimos que para definir un objeto para representar un punto del plano, definimos primero la clase Punto_plano de la

siguiente forma:

```
class Punto_plano extends Object {

  private double x;
  private double y;

  Punto_plano() {
    x=0.0;
    y=0.0;
  }

  public double coordX() {
    return x;
  }

  public double coordY() {
    return y;
  }
  public actX ( double x) {
    this.x=x;
  }
  public actY ( double y) {
    this.y=y;
  }
}
```

Para poder utilizar los métodos de una clase, tenemos que definir un objeto de esa clase en otra clase. Una vez definido el objeto de una clase, todos los métodos y variables declarados con la palabra clave **public** pueden ser utilizados en esa nueva clase. Por lo tanto, una clase que haya definido un objeto de la clase Punto_plano podrá utilizar todos los métodos de esa clase. No ocurrirá lo mismo con las variables de la clase, pues éstas han sido definidas mediante la palabra clave **private**, y esto imposibilita que las variables de esta clase sean accedidas directamente por otra clase con un objeto de la clase Punto_plano.

Por lo tanto, la siguiente clase sería incorrecta:

```
public class PointInstance {

  public static void main(String[] args) {
    Punto_plano mypoint;

    mypoint = new Punto_plano();

    mypoint.x = 12.0;
    mypoint.y =13.0;
    System.out.println(mypoint.x);
    System.out.println(mypoint.y);
  }
}
```

mientras que la siguiente declaración sería correcta:

```
public class PointInstance {

  public static void main(String[] args) {
    Punto_plano mypoint;

    mypoint = new Punto_plano();

    mypoint.actX(12.0);
    mypoint.actY(13.0);
    System.out.println(mypoint.getX());
    System.out.println(mypoint.getY());
  }
}
```

Como vemos en este ejemplo, la definición de un objeto de la clase Punto_plano requiere varios pasos:

- Primero tenemos que declarar el objeto indicando primero el nombre de la clase y a continuación el nombre del objeto
- A continuación tenemos que instanciar e inicializar el objeto con el operador **new**. Mediante este operador creamos espacio de memoria para el nuevo objeto, y mediante el método que le sigue (que se llama constructor y tiene el mismo nombre que la clase), inicializamos las variables de ese objeto.

Instanciación de objetos de clases predefinidas primitivas

Para definir un objeto de la clase character o number tenemos que proceder de forma similar a C. Primero tenemos que indicar el nombre de la clase y a continuación el nombre de la variable. No tenemos que realizar la instanciación con el operador new requerido para las clases definidas por el usuario.

Instanciación de objetos arrays y strings

Para definir un objeto de la clase array, tenemos que proceder de forma similar a la definición de un objeto de una clase definida por el usuario, y por tanto es diferente que en C.

Recordemos que para definir una variable de tipo tabla en C (y en C++), tenemos que definir primero un tipo tabla asignándole un nombre, y a continuación definir una variable con ese nombre de tipo. Así, para definir una variable de tipo tabla de enteros, antes del programa principal tendríamos que declarar el tipo tenteros,

typedef int tenteros[10];

y en la declaración de variables del programa principal, definiríamos la variable t de tipo tenteros:

tenteros t

En Java no existe el constructor **typedef**, pues toda declaración de tipos se tiene que realizar mediante el constructor class. Como en Java la clase array ya está definida no la tenemos que definir, pero tenemos que tener en cuenta que hay que proceder de forma diferente que en las otras clases para realizar la definición de un objeto de tipo array. El motivo básico es que para realizar la definición tenemos que especificar el tipo base de la tabla, el número de dimensiones y el número de elementos de cada dimensión. Por tanto, en Java al definir un objeto de tipo tabla primero tenemos que especificar el tipo base de la tabla y sus dimensiones en la

declaración del objeto y al instanciar el objeto tenemos que especificar el número de posiciones de cada dimensión.

Por ejemplo, para definir una variable de tipo tabla de enteros, procederíamos de la siguiente forma:

```
int[] t;
t = new int[10];
```

Como veis en esta definición no se requiere el nombre de tipo tenteros como en C (y en C++) y el tipo tabla de enteros se referencia mediante la expresión int[]. Posteriormente, al instanciar la variable de tipos tabla de enteros se tiene que indicar el número de posiciones de la tabla.

Para definir una tabla de una clase definida por el usuario se procede de forma similar. Por ejemplo, para definir una tabla de puntos procederíamos de la siguiente forma:

```
Punto_plano[] tp;
tp = new Punto_plano[10];
```

Finalmente, para definir tablas de varias dimensiones utilizamos una sintaxis similar a C. Tenemos que utilizar 2 corchetes para cada dimensión. Por ejemplo, para definir una matriz 10*20 de enteros procederíamos de la siguiente forma:

```
int[][] m;
m = new int[10][20];
```

6.2.2 Clases en Java

La declaración general de una clase es la siguiente:

public class < *nombre* > **extends** < *clase* > {

< *lista_variables* >

< *constructores* >

< *metodos* >

}

- El calificativo **public** indica que la clase puede ser utilizada por cualquier otra clase. Este calificativo en Java es opcional y se puede sustituir por otros como **abstract** o **final** que nosotros no veremos.
- < *nombre* > puede ser cualquier identificador.
- **extends** < *clase* > es una cláusula opcional y se utiliza para heredar variables y métodos de una superclase. En la introducción vimos como utilizar esta cláusula para definir el tipo dipila de enteros como subclase de pila de enteros. En la clase Punto_plano utilizamos esta cláusula con la clase Object. Esta clase esta predefinida en Java y toda clase que definamos es subclase de ésta aunque no lo indiquemos explícitamente. En este curso la cláusula **extends** no la veremos en detalle.
- < *lista_variables* > permite definir un conjunto de variables asociado a la clase. La sintaxis de una declaración es la siguiente:

< *calificativo* > < *tipo* > < *nombre* >;

El calificativo indica normalmente el grado de visibilidad de la clase (**private, protected** o **public**). Existen otros calificativos de los cuales nosotros sólo veremos final y static. El calificativo final se utiliza para indicar que el identificador es una constante. El calificativo static se utiliza para indicar que no se va a asociar a cada objeto de la clase una variable del tipo indicado. La variable será compartida por todos los objetos de esa clase. En cambio, si utilizamos los calificativos **private, protected** o **public** cada vez que instanciamos un objeto, asociamos una variable de ese tipo al objeto. Finalmente veamos lo que denotan los calificativos **private, protected** y **public**:

- **private**: Indica que la variable sólo es visible dentro de la clase.
- **protected**: Indica que la variable es visible dentro de la clase y de sus subclases.
- **public** Indica que la variable es visible en todas partes.

Nosotros para la traducción de la representación de un tad utilizaremos el calificativo **protected**. En los módulos no utilizaremos estas variables.

- $< constructor >$ El constructor es un método que tiene el mismo nombre que la clase y se utiliza para inicializar variables. Toda clase ha de tener como mínimo un constructor pero puede tener más. La definición de un constructor es igual a la de un método con el mismo nombre que la clase. Por ejemplo en la clase Punto_plano definimos el constructor Punto_plano inicializando las coordenadas con las del punto del origen.
- $< metodos >$ Los métodos se utilizan para inicializar, consultar o modificar las variables de una clase. Su sintaxis puede ser de dos formas:
  $< calificativo > < nombre > (< lista\_par\_formales >)$ {

  $< cuerpo >$

  }
  si el método es una acción, o
  $< calificativo > < tipo > < nombre > (< lista\_parametros >)$ {

  $< cuerpo >$

  }
  si el método es una función.
  $< calificativo >$ Los calificativos que nosotros utilizaremos en los métodos son **public, protected** o **private** con el mismo significado que los calificativos de las variables de la clase.
  $< lista\_parametros >$ Recordar que en Java siempre existe un parámetro implícito que es de tipo el de la clase en que se encuentra el método y que se llama **this**. Por lo tanto, nunca hemos de poner este parámetro en la lista de parámetros.
  El paso de parámetros en Java es por valor, pero es posible traducir un parámetro de salida o entrada/salida de tipo array o una clase predefinida pues el nombre de estos parámetros denotan la dirección de memoria donde se encuentra el objeto de ese tipo. Para estos tipos de parámetros, los parámetros de entrada que modificamos su

valor durante la ejecución de una acción es necesario realizar una copia de su valor inicial en una variable auxiliar para restaurarlo al finalizar la acción. Además hay que tener en cuenta que para realizar una copia de un parámetro no es suficiente realizar una asignación, sino que tenemos que copiar recursivamente todas las componentes del parámetro. En el ejemplo de la pila de enteros, tendremos que copiar todos los enteros de la pila en otra pila. Si la pila fuese de pila enteros, entonces para cada pila de enteros de la pila, tendríamos que copiar cada uno de los enteros de la pila de enteros en una pila auxiliar, y por tanto tendríamos que crear tantas pilas de enteros como pilas de enteros haya en la pila de pilas de enteros original. En la sección de ejemplos veremos como copiar una pila de enteros mediante una acción de dos formas diferentes. En ambos casos, existe un parámetro de entrada y un parámetro de salida aunque en uno de ellos el parámetro de entrada es el ímplicito

Para traducir los parámetros de salida o entrada/salida de las clases primitivas number o character tenemos que definir una clase con el conjunto de parámetros de salida y entrada/salida pues un objeto de las clases primitivas denotan el valor y no la dirección de memoria donde se encuentra dicho objeto. Por ejemplo, para definir la acción intercambiar tendríamos que definir las siguientes clases:

```
class Par_int {

  int i,j;

}
class intercambiar {

void interc(Par_int pl) {
int aux;
aux = pl.i;
pl.i = pl.j;
pl.j = aux;
}
}
```

Recordar que para intercambiar los valores de dos objetos de una clase predefinida, no podemos realizar en general la acción con tres asignaciones pues por ejemplo, al copiar primero uno de los objetos en una variable auxiliar tenemos que copiar todas las componentes de los objetos recursivamente uno por uno. Si quisiéramos intercambiar dos pilas de enteros, tendríamos que utilizar una de las dos acciones para copiar pilas de enteros que veremos en la siguiente sección.

$< cuerpo >$ El cuerpo es muy similar al cuerpo de un programa en C, pudiendo tener variables locales

Las clases en Java se corresponden con el concepto de módulo y tad en nuestra notación. Las clases en Java no pueden ser genéricas y sólo se puede heredar de una clase (herencia simple). Por otro lado, existe un tipo de módulos en Java (interfaces) a los que sí se le permite heredar de varios módulos (herencia múltiple). Nosotros no veremos este tipo de módulos.

6.2.3 Ejemplos en Java

Analicemos en detalle algunas de los tads vistos en la introducción en notación algorítmica ahora en Java.

Veamos primero la definición del tipo pila de enteros en Java:

```
class Pila_enteros extends Object {

    protected int[] pila;
    protected int altura;
    static final int MAX = 20;

    Pila_enteros() {
      pila = new int[MAX];
      altura=0;
    }
    public void empilar(int el) {
      altura=altura+1;
      pila[altura-1] = el;
    }
    public void desempilar(){
      altura=altura-1;
    }

    public int cumbre() {
      return pila[altura-1];
    }
  }

  }
```

Ahora veamos dos formas de definir la acción copiar_pila. La primera la definimos dentro de la clase, mientras que la segunda la definimos en una clase aparte. La primera definición sería así:

```
  public void copiar_pila(Pila_enteros pil) {
    int i;
    i=0;
    while ( i<this.altura) {
      pil.pila[i] = this.pila[i];
      i=i+1;
      }
```

mientras que la segunda definición sería así:

```
  class copiar {
    public void copiar_pila2 (Pila_enteros pil, Pila_enteros pil2) {
      Pila_enteros pil_aux;
      pil_aux = new Pila_enteros();
      while (pil.posicion()>0) {
        pil_aux.apilar(pil.cumbre());
        pil.desapilar();
      }
      while (pil_aux.posicion()>0) {
        pil.apilar(pil_aux.cumbre());
        pil2.apilar(pil_aux.cumbre());
        pil_aux.desapilar();
      }
```

Notar que en esta segunda definición como modificamos el parámetro de entrada, tenemos que hacer al inicio una copia en una variable auxiliar

para restaurarlo después. De todas formas, en este algoritmo la variable auxiliar también es requerida pues para realizar la copia también requerimos una variable auxiliar para almacenar primero la pila invertida.

Veamos a continuación como definir el tipo de datos dipila de enteros en Java:

```
class Dipila_enteros extends Object {

  protected int[] dp;
  protected int primero;
  protected int altura;
  static final int MAX = 20;



  Dipila_enteros() {
    dp = new int[MAX];
    altura= MAX / 2+1;
    primero = MAX / 2;
  }

  public void insertar_primero(int el) {
    primero = primero -1;
    dp[primero+1]=el;
  }

  public void suprimir_primero() {
    primero=primero+1;
  }

  public void empilar(int el) {
    altura = altura+1;
    dp[altura-1] = el;    }

  public void desempilar() {
    altura=altura-1;
  }

  public int cumbre() {
    return dp[altura-1];
  }
}
```

Y ahora veamos como definir la dipila de enteros utilizando el mecanismo de herencia simple en Java a partir de la clase pila de enteros:

```
class Dipila_enteros extends Pila_enteros {
  protected int primero;
  static final int MAX = 20;
  Dipila_enteros() {
    pila = new int[MAX];
    altura= MAX / 2 +1;
    primero = MAX / 2;
  }
  public void insertar_primero(int el) {
    primero = primero -1;
    pila[primero+1]=el;
  }
  public void suprimir_primero() {
    primero=primero+1;
  }
}
```

En Java no existe la posibilidad de definir clases genéricas, pero en [IPW01] se ha realizado una extensión de un subconjunto de Java con clases genéricas. Utilizaremos su sintaxis para definir las clases pila de elementos y dipila de elementos como subclase de pila de elementos.

La clase pila de elementos se defeniría así:

```
class Pila_el < X extends Object > extends Object {

  protected int[] pila;
  protected int altura;
  static final int MAX = 20;

  Pila_el() {
    pila = new int[MAX];
    altura=0;
  }
  public void empilar(X el) {
    altura=altura+1;
    pila[altura-1] = el;
  }
  public void desempilar(){
    altura=altura-1;
  }

  public int cumbre() {
    return pila[altura-1];
  }
}
```

Y la clase dipila de elementos como subclase de pila de elementos se definiría así:

```
class Dipila_el < X extends object > extends Pila_el< X > {
  protected int primero;
  static final int MAX = 20;
  Dipila_el() {
    pila = new int[MAX];
    altura= MAX / 2+1;
    primero = MAX / 2;
  }
  public void insertar_primero(X el) {
    primero = primero -1;
    pila[primero+1]=el;
  }
  public void suprimir_primero() {
    primero=primero+1;
  }
}
```

6.3 C++

6.3.1 Clases y objetos en C++

La declaración general de una clase en C++ es la siguiente:

```
class < nombre > : < clases > {
    private :
      < lista_variables >
      < constructores_y_metodos >
    protected :
      < lista_variables >
      < constructores_y_metodos >
    public :
      < lista_variables >
      < constructores_y_metodos >
  }
```

Las declaraciones de variables y métodos se clasifican por su ámbito de visibilidad. El significado de **private**, **protected** y **public** es el mismo que en Java.

En C++ hay herencia múltiple permitiendo que una clase pueda heredar de varias superclases.

Veamos como se trata en C++ los conflictos generados en una clase D que hereda de una misma clase A por dos clases diferentes B y C. Por ejemplo, al invocar a un método m de la clase A en la clase D se produce un error de ambigüedad. Este error se soluciona añadiendo al nombre del método la clase de la cual se hereda, por ejemplo B::m. Recordemos que en notación algorítmica no se produce un error pues una misma función sólo se puede heredar una vez.

En $< lista\_variables >$ se declaran variables como en C, pues el calificativo que determina el ámbito de visibilidad ya ha sido declarado.

En $< metodos\_y\_constructores >$ se declaran los constructores y métodos de forma muy similar a Java. Recordar que al igual que en Java siempre existe un parámetro ímplicito con tipo el de la clase y que se llama **this**. En C++ this es un puntero a la clase y para acceder a una variable de la clase se tiene que utilizar el operador $->$. En cambio, una variable de tipo una clase predefinida, accederá a una variable de esa clase con el operador . como en Java.

El paso de parámetros en C++ es diferente que en Java pues las variables de una clase predefinida no denotan una dirección de memoria a un objeto de esa clase. Por tanto los parámetros de salida y entrada/salida los traduciremos utilizando referencias. Una referencia en C++ no es más que una variable que denota la dirección de memoria de otra variable. Para traducir los parámetros de salida y entrada/salida nosotros definiremos el parámetro formal como una referencia al parámetro real. Para ello, la declaración del parámetro formal será de la siguiente forma:

$$< nombre\_tipo > \& < nombre\_par\_formal >$$

Los parámetros de entrada se pueden traducir como una declaración de una variable de la forma habitual, y en el caso en que el tipo de la variable sea de dimensión grande y el parámetro no se tenga que modificar en la acción, podremos traducirlo mediante una referencia precedido de la palabra clave **const** para controlar que no se modifique el parámetro real. En los ejemplos veremos como implementar la acción copiar_pila también de dos formas diferentes.

En C++, tenemos además la posibilidad de definir clases genéricas. Recordamos que en notación algorítmica podemos parametrizar tads y módulos con tads y modulos, mientras que en C++ podemos parametrizar las clases con una lista de tipos o clases y una lista de argumentos. La diferencia básica es que la lista de tipos o clases no ha de estar definida anteriormente.

La sintaxis general es la siguiente:

template $<< lista\_de\_tipos >, < lista\_de\_args >>$

class $< nombre >: < clases >$ {
  $< cuerpo\_clase >$
}

Las clases en C++ también se corresponden con el concepto de módulo y tad en nuestra notación.

La genericidad es distinta pudiéndose parametrizar clases únicamente por clases que no han sido predefinidas. En contrapartida, se permite tener parámetros formales similares a los subprogramas pero no hay forma de parametrizar una operación para por ejemplo definir las listas ordenadas de elementos parametrizadas por una relación de orden. En nuestra notación este tipo genérico se podría definir utilizando un módulo Elem similar al que hemos utilizado para definir la pila de elementos, añadiendo una relación de orden en la signatura. Esta diferencia en la genericidad hace que el mecanismo de herencia de clases genéricas también sea distinto al de nuestra notación. Recordamos que en C++ una clase también puede heredar de varias clases (herencia múltiple) y que la forma de tratar los conflictos en el caso en que una clase A hereda de las clases B y C y éstas dos últimas heredan de una misma clase D no es igual en C++ que en nuestra notación.

Objetos en C++

Recordamos que en Java para definir un objeto de una clase predefinida, primero tenemos que declarar el objeto y a continuación tenemos que instanciar e inicializar el objeto con el operador **new** y un constructor.

En C++ no se utiliza el operador **new**. Por tanto, podemos considerar que al declarar un objeto en C++, el compilador automáticamente reserva memoria para su ubicación. Si adicionalmente queremos inicializar sus valores de una forma determinada, es necesario utilizar un constructor.

6.3.2 Ejemplos en C++

Analicemos ahora en detalle algunas de los tads vistos en la introducción en notación algorítmica, posteriormente en Java y ahora en C++.

Veamos primero la definición del tipo pila de enteros en C++:

```
class Pila_enteros {

  protected:
    static const int MAX = 20;
    int pila[MAX];
    int altura;

  public:
    Pila_enteros() {
      altura=0;
    }
    void empilar(int el) {
      altura=altura+1;
      pila[altura-1] = el;
    }
    void desempilar(){
      altura=altura-1;
    }

    public int cumbre() {
      return pila[altura-1];
    }

}
```

Veamos a continuación como definir el tipo de datos dipila de enteros en C++ como subclase de pila de enteros:

```
class Dipila_enteros : Pila_enteros {
  protected:
    int primero;
  public:
    Dipila_enteros() {
      altura= MAX/2 +1;
      primero = MAX/2;
    }
    void insertar_primero(int el) {
      primero = primero -1;
      pila[primero+1]=el;
    }
    void suprimir_primero() {
      primero=primero+1;
    }
}
```

Veamos finalmente como definir las clases genéricas pilas de elementos y dipilas de elementos como subclase de pilas de elementos en C++.

La clase genérica pila de elementos se definiría así:

```
template < classT >
class Pila_el {

  protected:
    static const int MAX = 20;
    T pila[MAX];
    int altura;

  public:
    Pila_el() {
      altura=0;
    }
    void empilar(const T& el) {
      altura=altura+1;
      pila[altura-1] = el;
    }
    void desempilar(){
      altura=altura-1;
    }

    public T cumbre() {
      return pila[altura-1];
    }
}
```

La acción copiar_pila en la clase pila de elementos que hemos visto se definiría así:

```
void copiar (Pila_el< T >& pil) {
  int i;
  i=0;
  while (i<this- >n) {
    pil.pila[i] = this- >pila[i];
    i=i+1;
  }
pil.n = this- >n;
}
```

Y la definición de la acción copiar_pila fuera de la clase pila de elementos sería así:

```
template < classT >
class Copiar_pila {
  public:
    void copiar_pila (Pila_gen< T > pil, Pila_gen< T >& pil2) {
      Pila_gen< T > pil_aux;
      while (pil.posicion()>0) {
        pil_aux.empilar(pil.cumbre());
        pil.desempilar();
      }

      while (pil_aux.posicion()>0) {
        pil2.empilar(pil_aux.cumbre());
        pil_aux.desempilar();
      }
```

Veamos a continuación como definir el tipo de datos dipila de elementos en C++ como subclase de pila de elementos:

```
template < classT >
class Dipila_el : public Pila_el< T > {
  protected:
    int primero;
  public:
    Dipila_el() {
      altura= MAX/2 + 1;
      primero = MAX/2;
    }
    void insertar_primero(const T& el) {
      primero = primero -1;
      pila[primero+1]=el;
    }
    void suprimir_primero() {
      primero=primero+1;
    }
}
```

Referencias

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, 2000.

[Ceb97] Francisco Javier Ceballos. *Programación orientada a objetos con C++*. Editorial ra-ma, 1997.

[CJO92] Silvia Clerici, Rosa Maria Jiménez, and Fernando Orejas. Semantic constructions in the specification language Glider. *Recent Trends in Data Type Specifications*, 1992.

[CoF98] CoFi LD Task. CASL the CoFi algebraic specification language. http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary-v1.0/index.html, October 1998.

[Fra93] Xavier Franch. *Estructuras de datos: especificación, diseño e implementación*. Edicions UPC, 1993.

[GB92] Joseph A Goguen and Rod Burstall. INSTITUTIONS: Abstract model theory for specification and programming. *Journal of the Assoc. for Computing Machinery*, 39(1):95–146, 1992.

[HC01] Cay Horstmann and Gary Cornell. *Core Java. Volume I-Fundamentals*. Sun Microsystems Press, 2001.

[IPW01] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, October 2001.

[Ore89] Fernando Orejas. Tipos de datos y lenguajes de programación. *Mundo Electrónico*, 1989.

[Str02] Bjarne Stroustrup. *El lenguaje de programación C++*. Addison Wesley, 2002.

[Wir02] Niklaus Wirth. Computing science education: The road not taken. *Proceedings of the 7th Annual Conference on Innovation and Technology in CS Education*, 2002.