Apuntes de Programación y estructuras de datos. Abstracción y modularidad

Nikos Mylonakis

nicos@lsi.upc.edu

Dept. Llenguatges i Sistemes Informátics Universitat Politécnica de Catalunya

Barcelona

- Para resolver problemas de cierta envergadura se utiliza normalmente la técnica de diseño que se conoce con el nombre de diseño modular.
- La idea básica es descomponer el problema en subproblemas que se puedan resolver con un conjunto de subprogramas que tengan una cierta cohesión y que sean más tratables que el problema original
- Al conjunto de subprogramas que resuelve un subproblema se le llama módulo funcional.
- Un módulo puede hacer uso de un subprograma de otro módulo si éste aparece en su interfaz

- Un interfaz es un conjunto de cabeceras de subprogramas que pueden ser utilizados por otros módulos
- Los interfaces aparecieron para minimizar el acoplamiento entre los módulos. Se dice que es conveniente un diseño con acoplamiento débil entre módulos.
- Con este fin, en un interfaz sólo tienen que aparecer los subprogramas principales del módulo y no los subprogramas que son utilizados para implementar los subprogramas principales
- Esta descomposición se conoce con el nombre de descomposición funcional

- Más adelante apareció un nuevo tipo de descomposición o abstracción que se conoce con el nombre de abstracción de datos
- La idea básica es determinar los tipos abstractos de datos que se requieren para resolver el problema
- Estos tipos de datos serán utillizadas por diferentes módulos funcionales
- La novedad principal de este mecanismo de abstracción es que no permite acceder directamente a la representación de la estructura de datos.

- De nuevo sólo podemos acceder mediante las operaciones principales declaradas en el interfaz. Esto garantiza el acoplamiento débil entre los módulos
- Esta técnica de diseño permite especificar primero todos los módulos funcionales y de datos y después implementarlos con las modificaciones oportunas o primero especificar e implementar unos módulos funcionales y después especificar e implementar el resto

- Las estructuras de datos se suelen implementar al final cuando ya se sabe con seguridad cuáles son las operaciones que interesan que sean más eficientes. Si es posible éstas han de poder ser ejecutables en tiempo constante.
- Veamos como ejemplo el cálculo de la frecuencia de aparición de cada letra minúscula en un texto dado

  accion frecuencia (**sal** f: tabla_frec, **ent** s: sec de caracter)
  { Pre: La secuencia s está cerrada }
  { Post: f contiene la frecuencia de las letras minúsculas de s }

- El módulo tendría adicionalmente una función auxiliar que determinaría si una letra es minúscula
- La estructura de datos tabla_frec estaría definida en otro módulo con las siguientes operaciones en el interfaz:

  accion inicializar_cero(**sal** f: tabla_frec)
  {Pre: Cierto }
  {Post: Inicializamos la tabla de frecuencia a cero }

accion incrementar(**ent/sal** f: tabla_frec, **ent** c:caracter)
{Pre: Cierto }
{Post: Incrementamos la frecuencia del carácter c en f
}

- En este caso la representación es muy sencilla mediante una tabla de carácteres con rango 1..27
- Si cambiamos levemente el problema con el fin de determinar la frecuencia de las palabras la estructura de datos se complica si queremos obtener una solución eficiente

- El módulo principal se simplifica pero ahora tenemos dos estructuras de datos: una para las palabras y otra para almacenar la frecuencia de las palabras
- Una posible representación de la estructura para almacenar la frecuencia de las palabras sería mediante una lista ordenada de pares (palabra,frecuencia)
- Esta solución hace que la operación incrementar tenga un coste lineal
- Este curso veremos una estructura de datos que mejora el coste de esta operación haciéndolo prácticamente constante

Sintaxis abstracta del lenguaje de módulos

Ya hemos visto varias implementaciones de módulos. Veamos ahora su sintaxis abstracta:

modulo $< nombre >$

  usa $< lista\_modulos >$

  ops $< operaciones >$

implementacion

  $< representacion >$

  $< subprogramas >$

fmodulo

- El nombre del módulo coincide con el nuevo tipo que estamos definiendo
- **usa** $< modulos >$
  Al incluir el nombre de un módulo o tad en esta lista nos permite utilizar cualquier elemento de su interfaz en el módulo que estamos definiendo.

- $< operaciones >$ Aquí definimos un conjunto de operaciones.
- Toda operación que no aparezca en la cláusula **ops** pero esté definida dentro del módulo, no podrá ser importada 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.
- Uno de los tipos ha de tener el nombre del nombre del módulo.

- $< subprogramas >$ Aquí se definen las operaciones del módulo utilizando como parámetros formales la representación del nuevo tipo definido en el módulo.
- Para cada operación, tenemos que definir su cabecera,a continuación su especificación indicando su precondición y postcondición y finalmente el código.

Especificación algebraica

- Hasta ahora hemos visto especificación pre/post de subprogramas
- Para el caso de las estructuras de datos definíamos una representación de la estructura que se utilizaba para la especificación pre/post de las operaciones.
- En este curso vamos a ver una nueva forma de especificar estructuras de datos independientemente de la representación de la estructura que se conoce como especificación algebraica de tipos abstractos de datos (tads).

- Una especificación algebraica de un tad consiste en un conjunto de géneros, una signatura con la aridad de las operaciones del tad y un conjunto de ecuaciones entre términos generados por las operaciones más un conjunto de variables asociados a cada género.
- Un término sin variables denota un valor del tad mientras que un término con variables denota un conjunto de valores no necesariamente acotado del tad
- Los términos con variables se utilizan en las ecuaciones para expresar propiedades de las operaciones de los tads

- Veamos como primer ejemplo la especificación algebraica del tad pila de enteros
- Los géneros del tad son $pila\_ent$ y $entero$
- La signatura contiene las operaciones siguientes con su correspondiente aridad

$$pila\_vacia :\rightarrow pila\_ent$$
$$empilar : entero \times pila\_ent \rightarrow pila\_ent$$
$$desempilar : pila\_ent \rightarrow pila\_ent \ \mathbf{parcial}$$
$$cima : pila\_ent \rightarrow entero \ \mathbf{parcial}$$

- Cualquier valor de una pila es expresable únicamente con las operaciones $pila\_vacia$ y $empilar$.
- Por ejemplo la pila con valores 2 4 y 6 en la cima se expresaría mediante el término $empilar(6, empilar(4, empilar(2, pila\_vacia)))$
- Pero tenemos adicionalmente una operación $desempilar$ que modifica el estado de una pila y la operación $cima$ que consulta el último elemento insertado en la pila.
- La especificación de su funcionamiento ha de ser mediante propiedades expresadas en forma de ecuaciones.

- Una propiedad de la operación desempilar es que al desempilar cualquier pila p al que se le ha empilado i obtenemos la misma pila p
- Esto se expresa mediante la ecuación $desempilar(empilar(i, p)) = p$ donde p es una variable del género $pila\_ent$ e i una variable del género $entero$.
- Otra propiedad de la operación cima es que la cima de cualquier pila p a la que se le empila i es i.
- Esto se expresa mediante la ecuación $cima(empilar(i, p)) = i$

- Estas dos ecuaciones no completan la especificación de $Pila\_ent$
- Adicionalmente tenemos que expresar que no podemos aplicar las operaciones $desempilar$ y $cima$ a la $pila\_vacia$.
- Esto se expresa mediante términos seguidos de la flecha $\uparrow$. En nuestro caso $desempilar(pila\_vacia) \uparrow$ y $cima(pila\_vacia) \uparrow$.
- Esto completa la especificación algebraica del tad Pila_ent

- Veamos ahora la especificación algebraica del tad tabla de frecuencia de palabras ($Tabla\_frec\_pal$)

especificacion $Tabla\_frec\_pal$

usa $palabra, natural$

ops

$inicializar :\longrightarrow tabla\_frec\_pal$

$incrementar : tabla\_frec\_pal \times palabra \longrightarrow tabla\_frec\_pal$

$frecuencia : tabla\_frec\_pal \times palabra \longrightarrow natural$

axiomas

$$incrementar(incrementar(tfp, p), q)$$
$$= incrementar(incrementar(tfp, q), p)$$
$$frecuencia(inicializar, p) = 0$$
$$frecuencia(incrementar(tfp, p), p)$$
$$= frecuencia(tfp, p) + 1$$
$$p \neq q \Rightarrow frecuencia(incrementar(tfp, p), q)$$
$$= frecuencia(tfp, q)$$

fespecificacion

- En este caso mediante las operaciones inicializar e incrementar_frec podemos generar todos las posibles tabla de frecuencia
- Para definir las propiedades de la operación consultora frecuencia requerimos de una ecuación condicional
- Esta ecuación condicional tiene como premisa una operación booleana sobre las palabras
- En general podremos tener un conjunto de conjunciones con ecuaciones y expresiones booleanas

Sintaxis abstracta

especificacion $< nombre >$ [$< par\_formal >$]

usa $< lista\_modulos >$

$< signatura >$

axiomas

$< ecuaciones >$

fespecificacion

- El interfaz de las especificaciones es igual que el de los módulos pudiendo tener especificaciones genéricas con la excepción que en la signatura de las operaciones sólo se tiene que expresar su aridad en el formato descrito en los ejemplos
- Las ecuaciones pueden ser ecuaciones simples de términos con variables, ecuaciones condicionales o un término seguido de $\uparrow$ que indica que ese término o conjunto de términos no está definido
- La interpretación de una ecuación t1=t2 si t1 y t2 no tienen variables es que t1 y t2 están definidos y denotan el mismo valor.

- Una substitución σ es una asociación de variables a términos sin variables
- La interpretación de una ecuación t1 = t2 si t1 y t2 tienen variables es que para toda substitución σ de las variables por términos definidos $\sigma\, t1$ y $\sigma\, t2$ están definidos y denotan el mismo valor
- La interpretación de una ecuación condicional es que para toda substitucion σ que haga que las premisas sean ciertas entonces $\sigma\, t1$ y $\sigma\, t2$ están definidos y denotan el mismo valor

Semántica

- Para determinar si una especificación es correcta es conveniente saber exactamente que denota y para ello hay que definir una semántica.
- Veamos un ejemplo

  especificacion treselem

  ops

  a:treselem;

  b:treselem;

  c:treselem;

  s:treselem $\rightarrow$ treselem;

axiomas

s(a)=b;

s(b)=c;

s(c)=a

- A una misma especificación se le pueden dar diferentes semánticas
- Las diferentes semánticas asocian una clase de álgebras a cada especificación

- Un álgebra consiste en un conjunto de valores para cada género y una función para cada operación de la signatura
- Mediante una semántica laxa asociamos una clase de álgebras que pueden ser muy diferentes entre sí.
- Ejemplos de estas álgebras para nuestro ejemplo son:
- $A1_{treselem} = \{1, 2, 3\},\ a_{A1} = 1,\ b_{A1} = 2,\ c_{A1} = 3,$ $s_{A1}(1) = 2,\ s_{A1}(2) = 3,\ s_{A1}(3) = 1$

- $A2_{treselem} = \{1, 2, 3\},\ a_{A2} = 2,\ b_{A2} = 1,\ c_{A2} = 3,$
  $s_{A2}(1) = 3,\ s_{A2}(2) = 1,\ s_{A2}(3) = 2$
- $A3_{treselem} = \{4, 5, 6\},\ a_{A3} = 4,\ b_{A3} = 5,\ c_{A3} = 6,$
  $s_{A3}(4) = 5,\ s_{A3}(5) = 6,\ s_{A3}(6) = 4$
- $B_{treselem} = \{1, 2, 3, 4, 5, 6\},\ a_B = 1,\ b_B = 2,\ c_C = 3,$
  $s_B(1) = 2,\ s_B(2) = 3,\ s_B(3) = 1,\ s_B(4) = 5,\ s_B(5) =$
  $5,\ s_B(6) = 5$
- $C_{treselem} = \{2\},\ a_C = 2,\ b_C = 2,\ c_C = 2,$
  $s_C(2) = 2$

- Un álgebra satisface una especificación si satisface todas sus ecuaciones.
- Podemos comprobar que las 5 álgebras dadas satisfacen la especificación
- De éstas se dice que las 3 primeras son isomorfas pues existe una biyección entre ellas
- Para dar semántica a TADS a nosotros no nos interesan las 2 últimas álgebras

- Como ahora veremos la semántica inicial no incluirá estas 2 últimas álgebras y definirá una clase isomorfa de álgebras como las 3 primeras de nuestro ejemplo
- Para ello incluye 2 condiciones adicionales además de que se tienen que satisfacer todas las ecuaciones de la especificación
- La primera condición es ausencia de elementos extraños. Es decir ausencia de valores del álgebra que no tengan un término que denote ese valor

- La segunda condición es ausencia de confusión. Es decir que no haya dos términos que denoten el mismo valor si no se puede deducir de las ecuaciones de la especificación.
- Se puede comprobar fácilmente que en nuestro ejemplo el álgebra B tiene elementos extraños y el álgebra C tiene confusión.
- En cambio las tres primeras álgebras Ai no tienen ni elementos extraños ni confusión y se puede demostrar que son isomorfas entre sí

- Se puede demostrar formalmente que si se imponen estas dos condiciones la semántica obtenida es una única clase isomorfa de álgebras.
- A esta semántica se le llama semantica inicial y es la semántica que utilizaremos para nuestra especificación algebraica de TADS
- Por tanto para determinar si un TAD T satisface una especificación tendremos que comprobar estas dos condiciones (ausencia de elementos extraños y ausencia de confusión) además de satisfacer todas las ecuaciones

- Ahora vamos a ver una metodología para definir especificaciones algebraicas de TADS
- Esta metodología se verá con el ejemplo de la especificación de los conjuntos de enteros
- La primera cuestión a resolver es determinar las operaciones a incluir. En un diseño modular queda determinado por las necesidades de las abstracciones funcionales pero en este ejemplo podemos incluir inicialmente las usuales

- Estas serían:

$$\emptyset :\rightarrow conjunto\_ent;$$
$$union : conjunto\_ent \times conjunto\_ent \rightarrow conjunto\_ent;$$
$$borrar\_ent : entero \times conjunto\_ent \rightarrow conjunto\_ent;$$
$$pert : entero \times conjunto\_ent \rightarrow booleano$$

- La segunda cuestión es determinar si con las operaciones podemos generar todos los posibles valores del TAD. En nuestro caso la respuesta es negativa y necesitamos añadir una nueva operación

- Podemos añadir una operación que genere un conjunto de un elemento o una operacion que añada un entero a un conjunto. Nosotros decidimos incorporar esta última con aridad

$$ampliar : entero \times conjunto\_ent \rightarrow conjunto\_ent$$

- Con estas operaciones podemos generar conjuntos no acotados de enteros aunque no infinitos

- La tercera cuestión es determinar un conjunto mínimo con en el cual podemos generar todo el conjunto de valores
- En nuestro caso escogemos $\emptyset$ y la operación $ampliar$.
- La cuarta cuestión es determinar si existen pares de términos generados por constructoras que han de denotar el mismo valor.
- Si es así tenemos que añadir ecuaciones que caractericen esto

- En nuestro ejemplo tenemos que añadir los siguientes axiomas:

$$ampliar(i, ampliar(j, c)) = ampliar(j, ampliar(i, c));$$
$$ampliar(i, ampliar(i, c)) = ampliar(i, c)$$

- Ahora es conveniente determinar una forma estandar o canónica de denotar todos los posibles valores de un género y comprobar que mediante nuestras ecuaciones podemos convertir cualquier término en término canónico

- En nuestro caso una forma canónica de representar un conjunto {i1,. . . ,in} es añadiendo ordenadamente los elementos mediante la operación ampliar
- Es fácil comprobar que cualquier término que añade elementos de forma arbitraria repitiendo inserciones, mediante las 2 ecuaciones descritas podemos convertirlo a un término canónico
- La ultima cuestión consiste en definir las operaciones no constructoras. Una forma usual de proceder es tratar en diferentes ecuaciones las diferentes operaciones constructoras para los argumentos del género que tratamos.

- Esto a veces no es necesario y a veces se requiere un mayor análisis de caso que depende del problema
- Los axiomas requeridos serían los siguientes:

$$union(\emptyset, s) = s;$$
$$union(ampliar(v, s1), s2) = ampliar(v, union(s1, s2))$$
$$borrar\_ent(i, \emptyset) = \emptyset$$
$$borrar\_ent(i, ampliar(i, s)) = borrar\_ent(i, s)$$
$$i \neq j \Rightarrow borrar\_ent(i, ampliar(j, s)) = ampliar(j, borrar\_ent(i, s))$$

$$pert(i, \emptyset) = falso;$$
$$pert(i, ampliar(j, s)) = (i = j) \vee pert(i, s)$$

Esto concluye la especificación del TAD conjunto de enteros