Especificación de Programas

Juan Manuel Rabasedas

Abstracciones

The key to good software design is inventing appropriate abstractions around which to structure the software. Bad programmers typically don't even try to invent abstractions. Mediocre programmers invent abstractions sufficient to solve the current problem. Great programmers invent elegant abstractions that get used again and again.
Guttag - Horning, The Larch Book

Tipos Abstractos de Datos

- Un tipo no es solamente un conjunto de valores [J. Morris, 1973]
- Un tipo de datos queda definido si damos:
  - El conjunto de valores que puede tomar.
  - Un conjunto de operaciones definidas sobre estos valores.
  - Un conjunto de propiedades que relacionan todo lo anterior.
- Por ejemplo ¿Qué es una cola?:
  - Es una estructura a la cual:
  - Podemos agregar elementos
  - Podemos obtener el primer elemento
  - Podemos quitar el primer elemento
  - Podemos preguntar si está vacía
  - Existe una relación entre el orden en que se agregan elementos y se sacan (FIFO).
- Esta descripción es abstracta porque refleja el comportamiento y no la implementación

Tipos Abstractos de Datos

- La idea de un TAD es abstraer de detalles de la implementación.
- La especificación del TAD es la descripción formal del comportamiento esperado.
- Un usuario es alguien que usa las operaciones del TADs.
- El implementador provee un código ejecutable que satisface la especificación.
- El usuario sólo puede suponer lo especificado.
- La forma usual de especificar TADs es mediante especificaciones algebraicas.

Especificación Algebraica - TADs

Para proveer un TADs debemos dar:

- Un nombre al TAD y definir el conjunto de datos.
- Sus operaciones.
- Una especificación del comportamiento:
  - vamos a describir operaciones y ecuaciones entre operaciones.

  Para definir el **tad** Cola vamos a escribir el nombre y las operaciones dando el tipo de cada una de ellas.

$$
\begin{array}{l}
\textbf{tad}\ \mathsf{Cola}\ (\mathsf{A} : \mathsf{Set})\ \textbf{where} \\
\quad vacia : \mathsf{Cola\ A} \\
\quad poner : \mathsf{A} \to \mathsf{Cola\ A} \to \mathsf{Cola\ A} \\
\quad primero : \mathsf{Cola\ A} \to \mathsf{A} \\
\quad sacar : \mathsf{Cola\ A} \to \mathsf{Cola\ A} \\
\quad esVacia : \mathsf{Cola\ A} \to \mathsf{Bool}
\end{array}
$$

Especificación Algebraica - TADs

y a continuación vamos a proveer la especificación de esas operaciones:

$$
\begin{array}{lcl}
esVacia\ vacia & = & \text{true} \\
esVacia\ (poner\ x\ q) & = & \text{false} \\
primero\ (poner\ x\ vacia) & = & x \\
primero\ (poner\ x\ (poner\ y\ q)) & = & primero\ (poner\ y\ q) \\
sacar\ (poner\ x\ vacia) & = & vacia \\
sacar\ (poner\ x\ (poner\ y\ q)) & = & poner\ x\ (sacar\ (poner\ y\ q))
\end{array}
$$

- Notar la similitud de la especificación con la implementación.
- No tenemos pater-maching.
- No tenemos un orden de evaluación.
- Cada línea en la especificación es un predicado que debe ser verdadero.
- La conjunción (and) de todas las líneas deben ser verdaderas.

Implementaciones

Un tad puede admitir varias implementaciones. Para el **tad** Cola podemos implementarlo usando listas:

$$\begin{aligned} vacia &= [\,] \\ poner &= (:) \\ primero &= last \\ sacar &= init \\ esVacia &= null \end{aligned}$$

$$\begin{aligned} vacia &= [\,] \\ poner\ x\ xs &= (xs +\!\!+ [x]) \\ primero &= head \\ sacar &= tail \\ esVacia &= null \end{aligned}$$

¿Cuál es la diferencia entre estas dos implementaciones?

- $vacia$, $poner$ y $esVacia$ son O (1)
- $primero$ y $sacar$ son O (n)

- $vacia$, $primero$,$sacar$ y $esVacia$ son O (1)
- $poner$ es O (n)

Implementaciones

- Todas las implementaciones que vimos satisfacen la especificación.
- Cada implementación pueden tener diferentes costos de operaciones.
- Es importante incluir en la especificación el costo esperado.
- Cada implementación satisface otras propiedades no especificadas.
- Es importante saber que NO deben usarse estas propiedades.
- Los lenguajes de programación proveen mecanismos de ocultamiento de información
- Módulos, funciones locales, objetos con métodos privados, compilación separada, sistemas de paquetes, ...

Implementaciones en Haskell

Una forma de implementar un TAD en Haskell es mediante una clase de tipos:

class Cola t **where**
$vacia :: t\ a$
$poner :: a \to t\ a \to t\ a$
$sacar :: t\ a \to t\ a$
$primero :: t\ a \to a$
$esVacia :: t\ a \to \mathsf{Bool}$

Una implementación es una instancia

instance Cola $[\,]$ **where**
$vacia\ \ = [\,]$
$poner\ \ = (:)$
$primero = last$
$sacar\ \ = init$
$esVacia = null$

Implementaciones en Haskell

Definimos una función para el TAD Cola

$$
\begin{array}{l}
\mathit{dividir} :: \textsf{Cola}\ t \Rightarrow t\ a \rightarrow (t\ a, t\ a) \\
\mathit{dividir}\ \mathit{cola} = \\
\textbf{case}\ (\mathit{esVacia}\ \mathit{cola})\ \textbf{of} \\
\qquad \textsf{True} \rightarrow (\mathit{vacia}, \mathit{vacia}) \\
\qquad \textsf{False} \rightarrow \textbf{case}\ (\mathit{esVacia}\ (\mathit{sacar}\ \mathit{cola}))\ \textbf{of} \\
\qquad\qquad\qquad \textsf{True} \rightarrow (\mathit{cola}, \mathit{vacia}) \\
\qquad\qquad\qquad \textsf{False} \rightarrow \textbf{let}\ (\mathit{c1}, \mathit{c2}) = \mathit{dividir}\ (\mathit{sacar}\ (\mathit{sacar}\ \mathit{cola})) \\
\qquad\qquad\qquad\qquad \textbf{in}\ (\mathit{poner}\ (\mathit{primero}\ \mathit{cola})\ \mathit{c1}, \\
\qquad\qquad\qquad\qquad\quad \mathit{poner}\ (\mathit{primero}\ (\mathit{sacar}\ \mathit{cola}))\ \mathit{c2})
\end{array}
$$

- Notar que la función dividir funciona para cualquier implementación de Cola.
- La función dividir no puede suponer nada acerca de la implementación.

Verificando la implementaciones

Dada la implementación de un TAD en Haskell, ¿Cómo sabemos que es correcta?

- El sistema de tipos asegura que los tipos de las operaciones son correctos.
- Pero la verificación respecto a la especificación la debe hacer el programador.
- ¿Cómo verificar la implementación respecto a la especificación?

Razonamiento Ecuacional

Haskell permite razonar ecuacionalmente acerca de las definiciones en forma similar al álgebra.Probar que $reverse\ [x] = [x]$

$$
\begin{array}{ll}
 & reverse\ [x] \\
 & < def(:) > \\
= & reverse\ (x : [\,]) \\
 & < defreverse.2 > \\
= & reverse\ [\,] +\!\!+ [x] \\
 & < defreverse.1 > \\
= & [\,] +\!\!+ [x] \\
 & < def(++).1 > \\
= & [x]
\end{array}
$$

Patrones Disjuntos

Considere la siguiente función:

$$
\begin{aligned}
&esCero :: \mathsf{Int} \to \mathsf{Bool} \\
&esCero\ 0 = \mathsf{True} \\
&esCero\ n = \mathsf{False}
\end{aligned}
$$

- La segunda ecuación supone $n \not\equiv 0$.
- Es más fácil razonar ecuacionalmente si los patrones son disjuntos.

$$
\begin{aligned}
&esCero' :: \mathsf{Int} \to \mathsf{Bool} \\
&esCero'\ 0 = \mathsf{True} \\
&esCero'\ n \mid n \not\equiv 0 = \mathsf{False}
\end{aligned}
$$

Si los patrones son disjuntos, no importa el orden de las ecuaciones.

Principio de Extensionalidad

Dadas dos funciones $f, g :: \mathsf{X} \rightarrow \mathsf{Y}$ ¿Cuando dos funciones son iguales?

- Si para la misma entrada obtengo la misma salida.
- La función como una caja negra.
- Sólo veo lo que entra y lo que sale.

Definición (Principio de Extensionalidad)

$$f = g \;\Leftrightarrow\; \forall\, x :: \mathsf{X} \;\cdot\; f\ x = g\ x$$

Análisis por Casos

Podemos hacer análisis por casos para probar propiedades:

$$
\begin{aligned}
&not :: \mathsf{Bool} \to \mathsf{Bool}\\
&not\ \mathsf{False} = \mathsf{True}\\
&not\ \mathsf{True} = \mathsf{False}
\end{aligned}
$$

Probamos $not\ (not\ x) = x$, por casos de $x :: \mathsf{Bool}$:

- Caso $x = \mathsf{False}$

$$
\begin{aligned}
&not\ (not\ \mathsf{False})\\
&= < not \circ 1 >\\
&not\ \mathsf{True}\\
&= < not \circ 2 >\\
&\mathsf{False}
\end{aligned}
$$

- Caso $x = \mathsf{True}$

$$
\begin{aligned}
&not\ (not\ \mathsf{True})\\
&= < not \circ 2 >\\
&not\ \mathsf{False}\\
&= < not \circ 1 >\\
&\mathsf{True}
\end{aligned}
$$

Inducción Estructural

- Los programas funcionales interesantes usan recursión.
- Para poder probar propiedades acerca de programas recursivos usualmente uno necesita usar inducción
- La inducción nos da una forma de escribir una prueba infinita de una manera finita.

Inducción Estructural

Definición (Inducción Estructural)

Dada una propiedad P *sobre un tipo de datos algebraico* T, *para probar* $\forall \ \ t ::$ T.P (t)*:*

- *probamos* P (t) *para todo* t *dado por un constructor no recursivo*
- *para todo* t *dado por un constructor con instancias recursivas* $t_1, ..., t_k$, *probamos que si* P (t_i) *para* $i = 1, \ldots, k$ *entonces* P (t).

- Podemos definir una forma adicional de inducción estructural en la que suponemos que P (t_0) para todo t_0 ::T que ocurre dentro de t.
- Podemos definir el principio de inducción para cualquier tipo algebraico.

Inducción Estructural para Listas

Definición (Inducción Estructural para Listas)

Dada una propiedad P *sobre listas, para probar* $\forall \; l :: [a].\mathsf{P}\ (l)$*:*

- *Caso* $l = [\,]$ *Probamos* $\mathsf{P}\ ([\,])$
- *Caso* $l = (x : xs)$ *Probamos que si* $\mathsf{P}\ (xs)$ *entonces* $\mathsf{P}\ (x : xs)$*, con* $x :: a$

Probemos la siguiente propiedad
$reverse\ (xs +\!\!+\ ys) = reverse\ ys +\!\!+\ reverse\ xs$ para las definiciones usales de $reverser$ y $(+\!\!+)$

Inducción Estructural para Listas

$$\begin{aligned} reverse\ [\,] \quad &= [\,] \\ reverse\ (x : xs) &= reverse\ xs +\!\!+ [x] \end{aligned} \qquad \begin{aligned} &[\,] +\!\!+ ys = ys \\ &xs +\!\!+ [\,] = xs \\ &(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys) \end{aligned}$$

Voy a demostrar que $reverse\ (xs +\!\!+ ys) = reverse\ ys +\!\!+ reverse\ xs$ usando Inducción Estructural sobre el tipo $[a]$ para $\forall\ l :: [a]$.

Sea $l = [\,]$

$$\begin{aligned} & reverse\ ([\,] +\!\!+ ys) \\ & < def(++).1 > \\ = \quad & reverse\ ys \\ & < def(++).2 > \\ = \quad & reverse\ ys +\!\!+ [\,] \\ & < defreverse.1 > \\ = \quad & reverse\ ys +\!\!+ reverse\ [\,] \end{aligned}$$

Sea $l = (x : xs)$

Supongo valida la propiedad para xs **HI**

$reverse\ (xs +\!\!+\ ys) = reverse\ ys +\!\!+\ reverse\ xs$

$$
\begin{array}{cl}
 & reverse\ ((x : xs) +\!\!+\ ys) \\
 & < def(++).2 > \\
= & reverse\ (x : (xs +\!\!+\ ys)) \\
 & < defreverse.2 > \\
= & reverse\ (xs +\!\!+\ ys) +\!\!+\ [x] \\
 & < Por\textbf{HI} > \\
= & reverse\ ys +\!\!+\ reverse\ xs +\!\!+\ [x] \\
 & < Por\ Asociarividad\ de\ (++) > \\
= & reverse\ ys +\!\!+\ (reverse\ xs +\!\!+\ [x]) \\
 & < defreverse.2 > \\
= & reverse\ ys +\!\!+\ (reverse\ (x : xs))
\end{array}
$$

Inducción Estructural para árboles binarios

Data BTree a = E | T (Btree a) a (BTree a)

Definición (Inducción Estructural para BTree)

Dada una propiedad P *sobre elementos de* BTree, *para probar* $\forall$ t :: BTree a.P (t)*:*

- *Probamos* P (E).
- *Probamos que si* P (l) *y* P (r) *entonces* P (T l x r) *con* x :: a

Referencias

- Programming in Haskell. Graham Hutton (2007)
- Introduction to Functional Programming. Richard Bird (1998)
- Foundations of Programming Languages. John C. Mitchell (1996)
- The Larch Book. John V. GuttagJames J. Horning. et al.