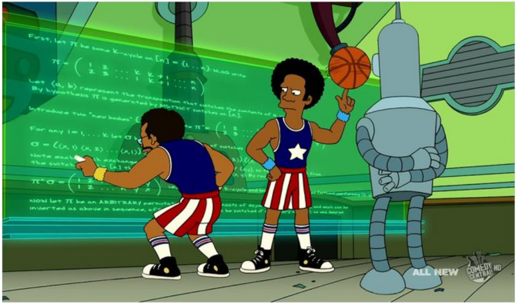
Especificando y razonando acerca de programas

Mauro Jaskelioff

18/04/2016



Colas

- ¿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 tipo abstracto de datos es abstraer detalles de implementación.
- Un usuario es alguien que simplemente usa la abstracción.
- El implementador provee un implementación que se ajusta al comportamiento esperado.
- El usuario sólo puede suponer el comportamiento descripto.
- Podemos ser más precisos sobre el comportamiento de las colas mediante una *especificación*.

TADs

Un TAD consiste de:

1. Un nombre de tipo. Ej . *Cola*
2. Operaciones.

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

3. Especificación del comportamiento
   - Especificación algebraica
     - Se describen operaciones y ecuaciones entre operaciones
   - Modelos
     - Se describen operaciones y cómo se interpretan en un modelo matemático

Especificaciones Algebraicas

- Especificación algebraica para colas

$$
\begin{aligned}
\mathit{esVacia}\ \mathit{vacia} &= \mathit{True} \\
\mathit{esVacia}\ (\mathit{poner}\ x\ q) &= \mathit{False} \\
\mathit{primero}\ (\mathit{poner}\ x\ \mathit{vacia}) &= x \\
\mathit{primero}\ (\mathit{poner}\ x\ (\mathit{poner}\ y\ q)) &= \mathit{primero}\ (\mathit{poner}\ y\ q) \\
\mathit{sacar}\ (\mathit{poner}\ x\ \mathit{vacia}) &= \mathit{vacia} \\
\mathit{sacar}\ (\mathit{poner}\ x\ (\mathit{poner}\ y\ q)) &= \mathit{poner}\ x\ (\mathit{sacar}\ (\mathit{poner}\ y\ q))
\end{aligned}
$$

- No confundir especificación con implementación!

Modelos

- Como modelo de colas tomamos las secuencias $\langle x_1, x_2, \ldots, x_n \rangle$
- Para cada operación damos una función equivalente sobre modelos:

$$
\begin{array}{lll}
\mathit{vacia} & = \langle\rangle & \\
\mathit{poner}\ x\ \langle x_1, x_2\ \ \ldots, x_n \rangle & = \langle x, x_1, x_2, \ldots, x_n \rangle & \\
\mathit{sacar}\ \langle x_1, x_2, \ldots, x_n \rangle & = \langle x_1, x_2, \ldots, x_{n-1} \rangle & \\
\mathit{primero}\ \langle x_1, x_2, \ldots, x_n \rangle & = x_n & \\
\mathit{esVacia}\ \langle x_1, x_2, \ldots, x_n \rangle & = \mathit{True} & \mathit{si}\ \ n = 0 \\
\mathit{esVacia}\ \langle x_1, x_2, \ldots, x_n \rangle & = \mathit{False} & \mathit{en\ otro\ caso}
\end{array}
$$

Implementaciones

- Cada TAD admite diferentes implementaciones
- Ejercicio:
  a) Implementar en Haskell el TAD de colas usando listas.

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

  b) ¿Cuál es el trabajo de las operaciones?

Otra implementación de Colas

- Implementemos colas usando un par de listas (xs, ys) tal que los elementos en orden sean $xs +\!\!+ \; reverse\ ys$
- Invariante de la implementación: Si xs es vacía, entonces ys también (las operaciones deben conservar este invariante...)

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{type}\ Cola\ a = ([a],[a])} \\
vacia & = ([\,],[\,]) \\
poner\ x\ (ys, zs) & = validar\ (ys, x : zs) \\
prim\ (x : xs, ys) & = x \\
sacar\ (x : xs, ys) & = validar\ (xs, ys) \\
esvacia\ (xs, ys) & = null\ xs \\
validar\ (xs, ys) & = \textbf{if}\ null\ xs\ \textbf{then}\ (reverse\ ys, [\,]) \\
 & \phantom{= \textbf{if}\ null\ xs\ }\textbf{else}\ \ (xs, ys)
\end{array}
$$

Especificación de costo

- Cada TAD admite diferentes implementaciones
- En cada implementación las operaciones pueden tener diferentes costos (trabajo, profundidad).
- Dependiendo del *uso* de la estructura, puede convenir una implementación u otra.
- Por lo tanto es importante tener una *especificación de costo* de cada implementación.
  - Ejemplo: Para la 2da implementación de colas

$$W_{vacia} \in O(1) \qquad W_{poner}(x, xs) \in O(1) \qquad W_{primero}(xs) \in O(1)$$

$$W_{sacar}(x, xs) \in O(|xs|), O(1)_{\text{(amortizado)}} \qquad W_{esVacia}(x, xs) \in O(1)$$

TADs en Haskell

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

$$
\begin{array}{l}
\textbf{class}\ \mathit{Cola}\ t\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{vacia} & :: t\ a \\
\mathit{poner} & :: a \rightarrow t\ a \rightarrow t\ a \\
\mathit{sacar} & :: t\ a \rightarrow t\ a \\
\mathit{primero} & :: t\ a \rightarrow a \\
\mathit{esVacia} & :: t\ a \rightarrow \mathit{Bool}
\end{array}
\end{array}
$$

- Una implementación es una instancia

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Cola}\ [\,]\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{vacia} & = [\,] \\
\mathit{poner}\ x\ \mathit{xs} & = x : \mathit{xs} \\
\mathit{sacar}\ \mathit{xs} & = \mathit{init}\ \mathit{xs} \\
\mathit{primero} & = \mathit{last} \\
\mathit{esVacia}\ \mathit{xs} & = \mathit{null}\ \mathit{xs}
\end{array}
\end{array}
$$

Usando TADs en Haskell

- Usamos un TAD con una función polimórfica en el TAD

$$
\begin{aligned}
&\mathit{ciclar} && :: \mathit{Cola}\ t \Rightarrow \mathit{Int} \rightarrow t\ a \rightarrow t\ a \\
&\mathit{ciclar}\ 0\ \mathit{cola} && = \mathit{cola} \\
&\mathit{ciclar}\ n\ \mathit{cola} && = \mathit{ciclar}\ (n-1)\ (\mathit{poner}\ (\mathit{primero}\ \mathit{cola}) \\
& && \qquad\qquad\qquad\qquad\qquad (\mathit{sacar}\ \mathit{cola}))
\end{aligned}
$$

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

TAD

- **Especificación:** Qué operaciones tiene el TAD y cómo se comportan. Es Única.
- **Implementación:** Cómo se realizan las operaciones y cuánto cuestan. Puede haber varias implementaciones (con diferentes costos). Todas deben garantizar el comportamiento dado por la especificación.
- **Uso:** Sólo puede suponer el comportamiento dado por la especificación. Se elije implementación de acuerdo al uso (menor costo para un determinado uso).

Verificando la especificación

- Dado una implementacíon TAD, ¿cómo sabemos que es correcta?
  - Implementa las operaciones.
  - Estas operaciones verifican la especificación.
- Dada una implementación en Haskell
  - El sistema de tipos asegura que los tipos de las operaciones son correctos.
  - Pero la verificación de la especificación la debe hacer el programador.
- Pero, ¿cómo verificar la especificación?

Razonamiento Ecuacional

- Haskell permite razonar ecuacionalmente acerca de las definiciones en forma similar al álgebra.

$$
\begin{array}{ll}
\mathit{reverse} & :: [a] \to [a] \\
\mathit{reverse}\ [\,] & = [\,] \\
\mathit{reverse}\ (x : xs) & = \mathit{reverse}\ xs +\!\!+ [x]
\end{array}
$$

Probar que

$$\mathit{reverse}\ [x] = [x]$$

$$
\begin{array}{ll}
 & \mathit{reverse}\ [x] \\
= & \quad \{\mathit{Listas}\} \\
 & \mathit{reverse}\ (x : [\,]) \\
= & \quad \{\mathit{reverse}\ .\ 2\} \\
 & \mathit{reverse}\ [\,] +\!\!+ [x] \\
= & \quad \{\mathit{reverse}\ .\ 1\} \\
 & [\,] +\!\!+ [x] \\
= & \quad \{(+\!\!+)\ .\ 1\} \\
 & [x]
\end{array}
$$

- Notar que en las ecuaciones usamos $=$ y no **==**.

Patrones disjuntos

- Considere la siguiente función:

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

- La 2da ecuación es aplicable sólo si $n \not\equiv 0$.
- Es más fácil razonar ecuacionalmente si los patrones son *disjuntos*.

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

- Patrones disjuntos $\Rightarrow$ no hace falta tener en cuenta el orden de las ecuaciones

Extensionalidad

- Dadas dos funciones $f, g :: A \to B$

$$\text{¿Cómo probar que } f = g\text{?}$$

- Tomamos una visión de *caja negra* sobre las funciones.
  - Sólo podemos evaluar el comportamiento de una función, i.e. cómo se comporta al aplicarle argumentos.
- Principio de Extensionalidad:

$$f = g \qquad \Leftrightarrow \qquad \forall x :: A.\ f\ x = g\ x$$

Análisis por Casos

- Podemos hacer análisis por casos para probar propiedades:

$$
\begin{aligned}
\neg &:: \mathit{Bool} \to \mathit{Bool} \\
\neg\ \mathit{False} &= \mathit{True} \\
\neg\ \mathit{True} &= \mathit{False}
\end{aligned}
$$

- Probamos $\neg\ (\neg\ x) = x$, por casos de x:

- Caso $x = \mathit{False}$

$$
\begin{aligned}
&\neg\ (\neg\ \mathit{False}) \\
= &\quad \{\neg\ .\ 1\} \\
&\neg\ \mathit{True} \\
= &\quad \{\neg\ .\ 2\} \\
&\mathit{False}
\end{aligned}
$$

- Caso $x = \mathit{True}$

$$
\begin{aligned}
&\neg\ (\neg\ \mathit{True}) \\
= &\quad \{\neg\ .\ 2\} \\
&\neg\ \mathit{False} \\
= &\quad \{\neg\ .\ 1\} \\
&\mathit{True}
\end{aligned}
$$

Razonando con programas recursivos

- Los programas funcionales interesantes usan *recursión*.
- Para poder probar propiedades acerca de programas recursivos usualmente uno necesita usar *inducción*

Inducción

- La inducción nos da una forma de escribir una prueba infinita de una manera finita.
- Queremos probar una propiedad P para todo número natural.
  - Por ej: $P(n) = n$ es par o impar.
- Con un papel infinito e infinito tiempo podríamos probar $P(0)$, luego probar $P(1)$, luego $P(2)$, etc.
- La inducción es una forma de probar que con papel infinito e infinito tiempo podríamos completar la prueba.

Inducción sobre $\mathbb{N}$: Primera forma

Definición

Para probar $P(n)$ para todo $n \in \mathbb{N}$, probamos $P(0)$ y probamos que para cualquier m, si $P(m)$ entonces $P(m+1)$.

- La prueba $P(0)$ es lo que llamamos *caso base*.
- La prueba de que $P(m) \to P(m+1)$ es el *paso inductivo*.
- El suponer $P(m)$ verdadero es la *hipótesis de inducción*.
- Ej: Probar $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

Inducción sobre $\mathbb{N}$: Segunda forma

Definición

Para probar $P(n)$ para todo $n \in \mathbb{N}$, probamos que para cualquier m, si $P(i)$ para todo $i < m$, entonces $P(m)$.

- No hay caso base
- Suponemos $P(i)$ para todo $i < m$ (hipótesis de inducción)
- Esta forma es llamada a veces *inducción completa* o *inducción fuerte*.
- En realidad, es tan completa o fuerte como la anterior.

Inducción sobre otros conjuntos

- Podemos usar la inducción sobre los naturales para obtener inducción sobre otros conjuntos.
- Por ejemplo, podemos hacer inducción sobre la altura de un árbol o la longitud de una lista.
- En gral, dada una función $f : A \to \mathbb{N}$, y una propiedad P sobre elementos de A, podemos definir:

$$Q\ (n) = \forall a :: A\ .\quad f\ a = n \quad \Rightarrow \quad P\ (a)$$

- Transformamos una propiedad sobre A en una sobre $\mathbb{N}$.

Ejemplo

$$\textbf{data}\ \mathit{Bin} = \mathit{Null} \mid \mathit{Leaf} \mid \mathit{Node}\ \mathit{Bin}\ \mathit{Bin}$$

- Probar $\forall t :: \mathit{Bin}\ .\quad \mathit{cantleaf}\ t \leqslant \mathit{cantnode}\ t + 1$

$$\begin{aligned}
&\mathit{cantleaf} && :: \mathit{Bin} \to \mathit{Int} \\
&\mathit{cantleaf}\ \mathit{Null} && = 0 \\
&\mathit{cantleaf}\ \mathit{Leaf} && = 1 \\
&\mathit{cantleaf}\ (\mathit{Node}\ t\ u) && = \mathit{cantleaf}\ t + \mathit{cantleaf}\ u
\end{aligned}$$

$$\begin{aligned}
&\mathit{cantnode} && :: \mathit{Bin} \to \mathit{Int} \\
&\mathit{cantnode}\ (\mathit{Node}\ t\ u) && = 1 + \mathit{cantnode}\ t + \mathit{cantnode}\ u \\
&\mathit{cantnode}\ _ && = 0
\end{aligned}$$

Prueba del ejemplo

$$Q\ (n) = \forall t :: Bin\ .\quad height\ (t) = n \Rightarrow cantleaf\ \ t \leqslant cantnode\ t + 1$$

$$\begin{array}{ll} height & :: Bin \rightarrow Int \\ height\ (Node\ t\ u) & = 1 + max\ (height\ t)\ (height\ u) \\ height\ \_ & = 0 \end{array}$$

- Usamos la 2da forma de inducción y suponemos que $\forall i < n$, si $height\ (t) = i$ entonces $cantleaf\ \ t \leqslant cantnode\ t + 1$.
- Hacemos un análisis por casos de n
  - Si $n = 0$, entonces la HI no se aplica y debemos probar directamente

$$height\ (t) = 0 \quad \Rightarrow \quad cantleaf\ \ t \leqslant cantnode\ t + 1$$

  (¡fácil!)

Prueba del ejemplo (cont.)

- Si $n > 0$ y $\mathit{height}\ t = n$ entonces podemos calcular

$$
\begin{array}{ll}
 & \mathit{cantleaf}\ t \\
= & \quad \{\mathit{height}\ t > 0\} \\
 & \mathit{cantleaf}\ (\mathit{Node}\ u\ v) \\
= & \quad \{\mathit{cantleaf}\ .\ 3\} \\
 & \mathit{cantleaf}\ u + \mathit{cantleaf}\ v \\
\leqslant & \quad \{\mathit{HI}\ (\mathit{height}\ u < n) \wedge (\mathit{height}\ v < n)\} \\
 & \mathit{cantnode}\ u + 1 + \mathit{cantnode}\ v + 1 \\
= & \quad \{\mathit{cantnode}\ .\ 1\} \\
 & \mathit{cantnode}\ (\mathit{Node}\ u\ v) + 1
\end{array}
$$

- Pudimos probar una propiedad sobre árboles usando inducción sobre naturales.

Inducción Estructural

- Es más práctico hacer inducción directamente sobre la estructura del árbol.

Definición (Inducción estructural)

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

- *probamos $P\ (t)$ para todo t dado por un constructor no recursivo*
- *para todo t dado por un constructor con instancias recursivas $t_1, \ldots, t_k$, 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 $P\ (t')$ para *todo* $t' :: T$ que ocurre dentro de t.

Ejemplo: Inducción Estructural para *Bin*

$\textbf{data}\ \mathit{Bin} = \mathit{Null} \mid \mathit{Leaf} \mid \mathit{Node}\ \mathit{Bin}\ \mathit{Bin}$

Definición (Inducción estructural para *Bin*)

Dada una propiedad P sobre elementos de Bin, para probar $\forall t :: \mathit{Bin}\ .\quad P\ (t)$*:*

- *probamos* $P\ (\mathit{Null})$ *y* $P\ (\mathit{Leaf})$.
- *probamos que si* $P\ (u)$ *y* $P\ (v)$ *entonces* $P\ (\mathit{Node}\ u\ v)$.

- Probamos $\forall t :: \mathit{Bin}\ .\quad \mathit{cantleaf}\ \ t \leqslant \mathit{cantnode}\ t + 1$.

Prueba usando inducción estructural

- Caso *Null*:
  $\mathit{cantleaf}\ \mathit{Null} = 0 \leqslant 1 = 0 + 1 = \mathit{cantnode}\ \mathit{Null} + 1$
- Caso *Leaf*:
  $\mathit{cantleaf}\ \mathit{Leaf} = 1 \leqslant 1 = 0 + 1 = \mathit{cantnode}\ \mathit{Leaf} + 1$
- Caso *Node u v*:

La hipótesis inductiva es:

$$\mathit{cantleaf}\ u \leqslant \mathit{cantnode}\ u + 1$$
$$\mathit{cantleaf}\ v \leqslant \mathit{cantnode}\ v + 1$$

$$\begin{array}{cl}
 & \mathit{cantleaf}\ (\mathit{Node}\ u\ v) \\
= & \quad \{\mathit{cantleaf}\ .\ 3\} \\
 & \mathit{cantleaf}\ u + \mathit{cantleaf}\ v \\
\leqslant & \quad \{\mathit{HI}\} \\
 & \mathit{cantnode}\ u + 1 + \mathit{cantnode}\ v + 1 \\
= & \quad \{\mathit{cantnode}\ .\ 1\} \\
 & \mathit{cantnode}\ (\mathit{Node}\ u\ v) + 1
\end{array}$$

Ejemplo: Inducción Estructural para Listas

Definición (Inducción estructural para listas)

Dada una propiedad P sobre listas, para probar $\forall xs :: [a]\ .\quad P\ (xs)$:

- *probamos $P\ ([\,])$.*
- *probamos que si $P\ (xs)$ entonces $P\ (x : xs)$.*

Ejercicio

Probar $reverse\ (xs \mathbin{+\!\!+} ys) = reverse\ ys \mathbin{+\!\!+} reverse\ xs$.

Ejercicio

Exprese la inducción estructural para el tipo Nat

$$\textbf{data}\ Nat = Zero \mid Succ\ Nat$$

Ejemplo: Compilador correcto

- Dado un lenguaje aritmético simple, cuyo AST es:

$$\mathbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

- Su semántica denotacional está dada por el siguiente evaluador

$$\begin{aligned}
&\mathit{eval} && :: \mathit{Expr} \rightarrow \mathit{Int} \\
&\mathit{eval}\ (\mathit{Val}\ n) && = n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) && = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}$$

Máquina virtual

- Queremos compilar el lenguaje a la siguiente máquina de stack:

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Stack} & = & [\mathit{Int}] \\
\textbf{type}\ \mathit{Code} & = & [\mathit{Op}] \\
\textbf{data}\ \mathit{Op} & = & \mathit{PUSH}\ \mathit{Int} \mid \mathit{ADD}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{exec} & :: \mathit{Code} \to \mathit{Stack} \to \mathit{Stack} \\
\mathit{exec}\ [\,]\ s & = s \\
\mathit{exec}\ (\mathit{PUSH}\ n : c)\ s & = \mathit{exec}\ c\ (n : s) \\
\mathit{exec}\ (\mathit{ADD} : c)\ (m : n : s) & = \mathit{exec}\ c\ (n + m : s)
\end{array}
$$

Compilador

- Definimos un compilador

$$
\begin{array}{ll}
comp & :: Expr \to Code \\
comp\ (Val\ n) & = [PUSH\ n] \\
comp\ (Add\ x\ y) & = comp\ x \mathbin{+\!\!+} comp\ y \mathbin{+\!\!+} [ADD]
\end{array}
$$

$e = Add\ (Add\ (Val\ 2)\ (Val\ 3))\ (Val\ 4)$

$> eval\ e$

9

$> comp\ e$

$[PUSH\ 2, PUSH\ 3, ADD, PUSH\ 4, ADD]$

$> exec\ (comp\ e)\ [\,]$

$[9]$

¿Es correcto el compilador?

- El compilador es correcto si

$$exec\ (comp\ e)\ [\,] \quad = \quad [eval\ e]$$

- Lo probamos por inducción estructural de *Expr*

Definición (Inducción estructural para *Expr*)

Dada una propiedad P sobre elementos de Expr, para probar $\forall e :: Expr\ .\quad P\ (e)$*:*

- *probamos* $P\ (Val\ n)$ *para todo* n.
- *probamos que si* $P\ (e)$ *y* $P\ (e')$ *entonces* $P\ (Add\ e\ e')$.

- La prueba, en el pizarrón.

Conclusiones

- Tipos Abstractos de Datos
  - Ocultan detalles de implementación.
  - El comportamiento se describe algebraicamente o proveyendo un modelo.
  - Cada implementación debe tener una especificación de costo.
- Pruebas de propiedades
  - Probar una propiedad más general hace más fuerte la hipótesis de inducción.
  - ¡A veces es más fácil probar propiedades más generales!
  - Conviene estructurar las pruebas en lemas.

Referencias

- Programming in Haskell. Graham Hutton (2007)
- Introduction to Functional Programming. Richard Bird (1998)
- Foundations of Programming Languages. John C. Mitchell (1996)