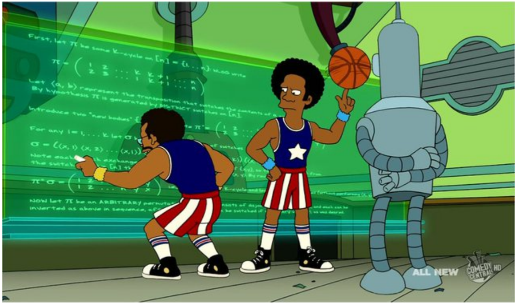
Razonando con programas

Mauro Jaskelioff

27/04/2020



Verificando la especificación

- Dado una implementación 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.

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

Probar que

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

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

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

Patrones disjuntos

- Considere la siguiente función:

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

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

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

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

Extensionalidad

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

$$¿\text{Cómo probar que } f = g?$$

- 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:

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

- Probamos $\mathit{not}\ (\mathit{not}\ x) = x$, por casos de x:

- Caso $x = \mathit{False}$

$$
\begin{aligned}
&\ \mathit{not}\ (\mathit{not}\ \mathit{False}) \\
= &\quad \{\mathit{not}\ .\ 1\} \\
&\ \mathit{not}\ \mathit{True} \\
= &\quad \{\mathit{not}\ .\ 2\} \\
&\ \mathit{False}
\end{aligned}
$$

- Caso $x = \mathit{True}$

$$
\begin{aligned}
&\ \mathit{not}\ (\mathit{not}\ \mathit{True}) \\
= &\quad \{\mathit{not}\ .\ 2\} \\
&\ \mathit{not}\ \mathit{False} \\
= &\quad \{\mathit{not}\ .\ 1\} \\
&\ \mathit{True}
\end{aligned}
$$

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*.

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

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

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

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

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

Prueba del ejemplo

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

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

- 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 $height\ t = n$ entonces podemos calcular

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

- 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 $\forall t :: T \;.\;\; P\ (t)$:

- *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 $reverse\ (xs +\!\!+\ ys) = reverse\ ys +\!\!+\ reverse\ xs$.

Ejercicio

Exprese la inducción estructural para el tipo Nat

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

Ejemplo: Compilador correcto

- Dado un lenguaje aritmético simple, cuyo AST es:

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

- Su semántica denotacional está dada por el siguiente evaluador

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

Máquina virtual

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

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

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

Compilador

- Definimos un compilador

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

$e = \mathit{Add}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))\ (\mathit{Val}\ 4)$

$> \mathit{eval}\ e$

9

$> \mathit{comp}\ e$

$[\mathit{PUSH}\ 2, \mathit{PUSH}\ 3, \mathit{ADD}, \mathit{PUSH}\ 4, \mathit{ADD}]$

$> \mathit{exec}\ (\mathit{comp}\ e)\ [\,]$

$[9]$

¿Es correcto el compilador?

- El compilador es correcto si

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

- 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 $P\ (Add\ e\ e')$.*

Prueba: el compilador es correcto

Queremos probar:

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

Lo hacemos por inducción estructural sobre e.

Caso *Val* $(e = Val\ n)$

$$
\begin{array}{lll}
 & exec\ (comp\ (Val\ n))\ [\,] & \\
= & \quad \{comp.1\} & comp\ (Val\ n) = [PUSH\ n] \\
 & exec\ [PUSH\ n]\ [\,] & \\
= & \quad \{exec.2\} & exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s) \\
 & exec\ [\,]\ [n] & \\
= & \quad \{exec.1\} & exec\ [\,]\ s = s \\
 & [n] & \\
= & \quad \{eval.1\} & eval\ (Val\ n) = n \\
 & [eval\ (Val\ n)] &
\end{array}
$$

Caso Add $(e = Add\ e_1\ e_2)$

Hipótesis Inductiva: $\begin{array}{ll} HI.1 & exec\ (comp\ e_1)\ [\,] = [eval\ e_1] \\ HI.2 & exec\ (comp\ e_2)\ [\,] = [eval\ e_2] \end{array}$

$$
\begin{array}{ll}
 & exec\ (comp\ (Add\ e_1\ e_2))\ [\,] \\
= & \{comp.2\} \qquad comp\ (Add\ x\ y) = comp\ x +\!\!+\ comp\ y +\!\!+ [ADD] \\
 & exec\ (comp\ e_1 +\!\!+\ comp\ e_2 +\!\!+ [ADD])\ [\,] \\
= & \{Lema\ 1 : \forall c, d, s.\ exec\ (c +\!\!+ d)\ s = exec\ d\ (exec\ c\ s)\} \\
 & exec\ (comp\ e_2 +\!\!+ [ADD])\ (exec\ (comp\ e_1)\ [\,]) \\
= & \{Lema\ 1 : \forall c, d, s.\ exec\ (c +\!\!+ d)\ s = exec\ d\ (exec\ c\ s)\} \\
 & exec\ [ADD]\ (exec\ (comp\ e_2)\ (exec\ (comp\ e_1)\ [\,])) \\
= & \{HI\} \qquad exec\ (comp\ e_1)\ [\,] = [eval\ e_1] \\
 & exec\ [ADD]\ (exec\ (comp\ e_2)\ [eval\ e_1]) \\
= & \{?\}
\end{array}
$$

No puedo seguir. Necesito **generalizar la hipótesis inductiva**!

Reforzando la hipótesis inductiva

- Probamos una propiedad más fuerte.
  - La hipótesis inductiva será mas fuerte.
- Generalizamos

$$
\begin{array}{lllll}
\forall e. & exec\ (comp\ e)\ [\,] & = & [eval\ e] \\
\forall e, s. & exec\ (comp\ e)\ s & = & (eval\ e) : s
\end{array}
$$

- La prueba original es un caso particular de la general (cuando $s = [\,]$).

Prueba: el compilador es correcto (prop. general)

Queremos probar:

$$\forall e, s. \quad exec\ (comp\ e)\ s \quad = \quad eval\ e : s$$

Lo hacemos por inducción estructural sobre e.

Caso *Val* $(e = Val\ n)$

$$
\begin{array}{lll}
 & exec\ (comp\ (Val\ n))\ s & \\
= & \quad \{comp.1\} & comp\ (Val\ n) = [PUSH\ n] \\
 & exec\ [PUSH\ n]\ s & \\
= & \quad \{exec.2\} & exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s) \\
 & exec\ []\ (n : s) & \\
= & \quad \{exec.1\} & exec\ []\ s = s \\
 & n : s & \\
= & \quad \{eval.1\} & eval\ (Val\ n) = n \\
 & eval\ (Val\ n) : s &
\end{array}
$$

Caso Add $(e = Add\ e_1\ e_2)$

Hipótesis Inductiva:

$$HI.1 \quad \forall s.\ exec\ (comp\ e_1)\ s = eval\ e_1 : s$$
$$HI.2 \quad \forall s.\ exec\ (comp\ e_2)\ s = eval\ e_2 : s$$

$$
\begin{array}{ll}
 & exec\ (comp\ (Add\ e_1\ e_2))\ s \\
= & \{comp.2\} \qquad comp\ (Add\ x\ y) = comp\ x +\!\!+\ comp\ y +\!\!+\ [ADD] \\
 & exec\ (comp\ e_1 +\!\!+\ comp\ e_2 +\!\!+\ [ADD])\ s \\
= & \{Lema\ 1 : \forall c, d, s.\ exec\ (c +\!\!+\ d)\ s = exec\ d\ (exec\ c\ s)\} \\
 & exec\ (comp\ e_2 +\!\!+\ [ADD])\ (exec\ (comp\ e_1)\ s) \\
= & \{Lema\ 1 : \forall c, d, s.\ exec\ (c +\!\!+\ d)\ s = exec\ d\ (exec\ c\ s)\} \\
 & exec\ [ADD]\ (exec\ (comp\ e_2)\ (exec\ (comp\ e_1)\ s)) \\
= & \{HI.1\} \qquad \forall s\ .\ exec\ (comp\ e_1)\ s = eval\ e_1 : s \\
 & exec\ [ADD]\ (exec\ (comp\ e_2)\ (eval\ e_1 : s)) \\
= & \{HI.2\} \qquad \forall s\ .\ exec\ (comp\ e_2)\ s = eval\ e_2 : s \\
 & exec\ [ADD]\ (eval\ e_2 : eval\ e_1 : s)
\end{array}
$$

Ahora podemos finalizar la prueba:

$$
\begin{array}{ll}
& exec\ [ADD]\ (eval\ e_2 : eval\ e_1 : s) \\
= & \{exec.3\} \qquad exec\ (ADD : c)\ (m : n : s) = exec\ c\ (n + m : s) \\
& exec\ []\ (eval\ e_1 + eval\ e_2 : s) \\
= & \{exec.1\} \qquad] \qquad exec\ []\ s = s \\
& (eval\ e_1 + eval\ e_2 : s) \\
= & \{eval.2\} \qquad eval\ (Add\ x\ y) = eval\ x + eval\ y \\
& eval\ (Add\ e_1\ e_2) : s
\end{array}
$$

- Nos queda probar el lema auxiliar:

  $$Lema\ 1 : \forall c, d, s.\ exec\ (c +\!\!+\ d)\ s = exec\ d\ (exec\ c\ s)$$

  donde para probar esto debemos asumir que el stack está bien formado, (para todo *ADD* hay al menos dos elementos en el stack).

Conclusiones

- 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)