Parte II: Programación Funcional

Técnicas Básicas de la Programación funcional (usando Haskell)

P. Julián-Iranzo

Universidad de Castilla-La Mancha

Programación Declarativa

versión Abril 2014

Índice

Índice

Objetivo

- Introducir un nuevo paradigma de programación, la Programación Funcional, presentando sus principales características y mecanismos operacionales.
- En particular, introducir el lenguaje Haskell, un lenguaje con semántica no-estrícta que constituye un estándar del área.

Índice

Definición de función

Dados dos conjuntos D y E, una *relación* $R \subseteq D \times E$, es un subconjunto de pares $R = \{(d, e) \mid d \in D \wedge e \in E\}$: Si $(d, e) \in R$ escribimos $d\ R\ e$.

Definition (Función parcial)

Dados dos conjuntos D (*dominio*) y E (*codominio* o rango), una función parcial f de D en E es una relación $f \subseteq D \times E$ que verifica:

$$d\ f\ e \quad \wedge \quad d\ f\ e' \quad \Rightarrow \quad e = e'$$

Entonces escribimos $f : D \to E$ en vez de $f \subseteq D \times E$ y $f(d) = e$ en vez de $d\ f\ e$.

En otras palabras: a cada elemento del dominio le corresponde un único elemento del codominio. Si $f(d) = e$, decimos que e es la *imagen* de d por f.

Definición de función

Definition (Función total)

Una función parcial $f : D \to E$ es una función total si verifica la siguiente propiedad:

$$\forall d \in D.\ \exists e \in E.\ f(d) = e$$

Por tanto, una función total $f : D \to E$ es una función parcial donde todo elemento del dominio D tiene una imagen en E (por f).

Example

La función sign : Int $\to \{\text{neg}, \text{cero}, \text{pos}\}$ que asigna a un número entero su signo (representado por neg, cero o pos) es una función total (todo número entero tiene signo negativo, positivo o es cero).

Definición de función

- A veces se representan las funciones como 'cajas negras' con una entrada de datos (un elemento del dominio) y una salida de datos (el elemento correspondiente del codominio).
- Funciones multiargumento: $f : D_1 \times \cdots \times D_k \to E$.
  Se pueden entender directamente como funciones $f : D \to E$ de un único argumento si escogemos $D = D_1 \times \cdots \times D_k$.
- Una constante c puede entenderse como una función $f_c : \to \{c\}$ cuyo dominio es el conjunto vacío y cuyo codominio es el conjunto unitario que contiene el valor asociado a la constante.
- Composición de funciones: Dadas dos funciones $f : D \to E$ y $g : E \to F$, la composición $g \circ f$ de ambas es una función $(g \circ f) : D \to F$ definida como $(g \circ f)(d) = g(f(d))$.

Descripción de una función

Podemos distinguir dos formas básicas de describir una función $f : D \to E$:

- Por *extensión* (o extensionalmente): dando el *grafo* de la función. Por ejemplo, la función sign vendría descrita por el siguiente conjunto (infinito) de pares:

$$\{\ldots, (-2, \text{neg}), (-1, \text{neg}), (0, \text{cero}), (1, \text{pos}), (2, \text{pos}), \ldots\}$$

- Por *comprensión* (o intensionalmente): Indicando una regla que nos permita establecer las asociaciones deseadas entre los elementos del dominio y del codominio. Por ejemplo, la función anterior puede darse como sigue:

$$\text{sign}(x) = \begin{cases} \text{neg} & \text{si } x < 0 \\ \text{cero} & \text{si } x = 0 \\ \text{pos} & \text{si } x > 0 \end{cases}$$

Descripción de una función

Ambas formas de definir una función son complementarias en el sentido de que cada una de ellas sugiere distintos aspectos computacionales.

- **La definición intensional** permite poner de manifiesto los aspectos dinámicos o computacionales de la descripción de las funciones al requerir que la función sea obtenida a partir de otras funciones conocidas. Por tanto, de forma implícita, las definiciones intensionales exigen utilizar algún tipo de proceso de cómputo para poder describir una función.
- **La definición extensional** centra la atención en los aspectos estáticos o declarativos de la función, es decir en las asociaciones que ésta establece entre los elementos del dominio y el codominio. Esto es útil para discutir sobre la semántica de las funciones definidas mediante reglas aparentemente distintas.

Descripción de una función

Example

Por ejemplo, las funciones doble : Int $\to$ Int y doble$'$: Int $\to$ Int dadas por:

$$\text{doble}(x) = x + x \qquad \text{y} \qquad \text{doble}'(x) = 2 * x$$

tienen definiciones intensionales distintas. No obstante, los grafos de ambas funciones coinciden:

$$\{\ldots, (-2,-4), (-1,-2), (0,0), (1,2), (2,4), \ldots\}$$

Esto muestra que ambas funciones son idénticas (de acuerdo con el principio de extensionalidad).

Propiedades de las funciones

- Determinismo: A cada elemento del dominio de una función le corresponde un único elemento del codominio.
- Dependencia de los argumentos: El valor de una función aplicada sobre sus argumentos viene determinado exclusivamente por éstos.
- Transparencia referencial: El resultado de la evaluación de una función es independiente del momento y lugar desde el que se la referencia. Cada expresión designa un único valor independientemente del modo en que se evalúa.
- Extensionalidad: Dos funciones $f : D \to E$ y $g : D \to E$ son iguales si para cada elemento del dominio las imagenes de ambas son iguales: $(\forall d \in D.\ f(d) = g(d)) \Rightarrow f = g$

Funciones y lenguajes de programación convencionales (opacidad referencial)

Example

```
program example(output);
var flag:  boolean;
function f(x:Integer):  integer;
 begin
    IF flag THEN f:=x ELSE else f:=2*x;
    flag:=not flag;
 end;
begin
    flag:= true;
    writeln(f(1)+f(2));
    writeln(f(2)+f(1));
end.
```

Aplicación de una función

- Constituye el principio dinámico básico derivado del concepto de función.
- Una función f *aplicada* sobre un *argumento* o parámetro actual d constituye una expresión $f(d)$ cuya evaluación es el valor e del codominio de la función que es asignado por f al elemento d.
- La composición de funciones junto con la aplicación constituye el principio básico para definir las expresiones susceptibles de representar cómputos.

Observación

Distinguir entre parámero formal *(usado en la definición de la función) y* parámero actual, real *o* argumento.

Índice

Funciones y programación funcional

Del concepto de función, surge la idea de disponer de un lenguaje de programación que permita la *definición de funciones* y la *especificación de expresiones* cuya evaluación puede entenderse como la *ejecución* del programa.

- Componente estático: El perfil de una función $f : D \to E$, sugiere la necesidad de describir los dominios de definición de las funciones y las funciones en sí mismas:

$$\underbrace{f :}_{\text{Función}} \quad \overbrace{D \to E}^{\text{Dominios}}$$

  - La descripción de dominios nos lleva a la idea de *tipo*.
  - La descripción de funciones a la idea de *definición de función*.

Funciones y programación funcional

Del concepto de función, surge la idea de disponer de un lenguaje de programación que permita la definición de funciones y la especificación de expresiones cuya evaluación puede entenderse como la *ejecución* del programa.

- Componente dinámico: La noción de aplicación de una función a sus argumentos sugiere que el componente dinámico del paradigma debe dar cuenta de cómo evaluar expresiones, de acuerdo con la descripción dada para las funciones involucradas.

Observación

Seguiremos la sintaxis de Haskell en los ejemplos: La aplicación de una función `f` *sobre sus argumentos* $\texttt{e}_1,\ldots,\texttt{e}_k$ *se denota*

$$\texttt{f}\ \texttt{e}_1 \cdots\ \texttt{e}_k$$

en lugar de $\texttt{f(e}_1,\ldots,\texttt{e}_k\texttt{)}$.

Índice

Tipos: motivación e introducción

El uso de tipos en programación obedece a dos motivaciones fundamentales:

- Evitar o detectar errores de programación.
- Ayudar a definir estructuras de datos.

En un lenguaje de programación *tipificado* (*typed*), todos los valores y expresiones manejadas tienen un *tipo*.

$\Longrightarrow$ Restricción en la variedad de expresiones que podemos considerar en nuestros programas.

Sólo están permitidas las *expresiones bien tipificadas*.

Observación

En Haskell, el comando `:type` *infiere y muestra el tipo de una expresión.*

Tipos: motivación e introducción

El uso de tipos en un lenguaje de programación (funcional) supone la existencia de:

- Un lenguaje de *expresiones de tipo*.
- Un lenguaje de *expresiones de programa*.
- Un *sistema de tipificación* de las expresiones de programa que determina si una expresión está bien tipificada o no: *comprobación e inferencia de tipos*.

Example

La expresión if(cond,3,False) donde cond es una expresión booleana, no está bien tipificada.

Observación

3 *y* False *no tienen el mismo tipo, lo que imposibilita establecer un perfil consistente para la función* if.

Tipos: motivación e introducción

- En un lenguaje *fuertemente tipificado* (*strongly typed*), se realiza una comprobación de tipo de todas las expresiones que aparecen en el programa.
- Comprobación de tipos estática (*static type checking*): El tipo de cada expresión se determina antes de la ejecución del programa:
  - evita comprobaciones de tipo en tiempo de ejecución;
  - ahorra código generado y agiliza la ejecución;
  - no elimina todas las posibilidades de error en tiempo de ejecución.

Example

La expresión 1/0 está bien tipificada, pero su evaluación produce un error.

Expresiones de tipo

- Los tipos se expresan también como construcciones sintácticas producidas por una gramática.
- Existen *tipos básicos*: Bool, Char, String, Int, Integer, Float, Double.
- y operaciones de *construcción de tipos*, que combinan tipos conocidos para formar otros nuevos:
  - El operador $\rightarrow$ es el operador de construcción de tipos funcionales: Si τ_1 y τ_2 son tipos entonces $\tau_1 \rightarrow \tau_2$ es un tipo funcional.
  - El operador $\times$ es el constructor de tipos producto (tuplas): Si $\tau_1, \tau_2, \ldots, \tau_n$ son tipos entonces $\tau_1 \times \tau_2 \times \ldots \times \tau_n$ es un tipo producto.

Example

Gramática simple para expresiones de tipos *monomórficos* (*monotipos*):

$$\tau ::= \text{Bool} \mid \text{Char} \mid \text{String} \mid \cdots \mid \tau \rightarrow \tau \mid \tau \times \tau$$

Expresiones de tipo

Table: Tipos básicos en Haskell

Tipo	**Descripción**	**Valores (ejemplos)**
Bool	Valores lógicos	True y False
Char	Caracteres	'a', 'B', '3', '+'
String	Cadena de caracteres	"abc", "1 + 2 = 3"
Int	Enteros de precisión fija Enteros entre -2^{31} y $2^{31} - 1$.	123, -12
Integer	Enteros de precisión arbitraria	2032053956
Float	Reales de precisión simple (32 bits - 7 digitos)	1.2, -23.45, 45e-7
Double	Reales de precisión doble (64 bits - 16 digitos)	0.376739501953125, 45e-12

Expresiones de tipo

- La asignación de un tipo a una función se efectúa mediante el operador de *tipificación* (*typing*) '::'. $+ :: \text{Int} \times \text{Int} \rightarrow \text{Int}$
- Es posible emplear variables de tipo (α, β, γ) que expresan cualquier tipo en una expresión de tipo.
- Las expresiones de tipo que contienen variables de tipo dan lugar a sistemas de tipos *polimórficos* (*politipos*).

Example

El (poli)tipo de la función condicional if es: $\text{if} :: \text{Bool} \times \alpha \times \alpha \rightarrow \alpha$

- El polimorfismo obtenido de esta forma, se denomina *polimorfismo paramétrico*: las variables de tipo actúan como parámetros y representan cualquier tipo monomórfo.

Sobrecarga de tipos

- Debe diferenciarse entre polimorfismo y el llamado *polimorfismo ad-hoc* o *sobrecargado*.
- El *polimorfismo ad-hoc* consiste en que las variables de tipo sólo pueden adquirir ciertos monotipos seleccionados según la aplicación concreta (contexto)
- Cuando hablamos de funciones, el término de "sobrecarga" se refiere al uso del mismo nombre para diferentes funciones.

Example

Se emplea el mismo símbolo $+$ para denotar a la función suma cuando actúa sobre números enteros o reales.

$$\left(\begin{array}{l} + :: \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int} \\ + :: \mathsf{Integer} \times \mathsf{Integer} \to \mathsf{Integer} \\ + :: \mathsf{Float} \times \mathsf{Float} \to \mathsf{Float} \\ + :: \mathsf{Double} \times \mathsf{Double} \to \mathsf{Double} \end{array} \right)$$

Sobrecarga de tipos

- Esto puede expresarse asignando al operador '$+$' un politipo restringido $+ :: \alpha \times \alpha \to \alpha$, donde la variable de tipo, α, no puede ser instanciada a cualquier monotipo, sino sólo a los tipos numéricos.
- Haskell adopta esta solución: incorpora la noción de *clase* que permite agrupar tipos de datos que 'comparten' determinadas operaciones y funciones.

Example

El tipo asignado a '$+$', es:

$$+ :: \mathsf{Num}(\alpha) \Rightarrow \alpha \times \alpha \to \alpha,$$

esto es, el tipo de '$+$' es $\alpha \times \alpha \to \alpha$ restringido a los α que pertenecen a la clase Num.

Clases de tipos en Haskell

- Para introducir operadores sobrecargados, se necesita *declarar* primero una *clase de tipos*

Example

```
class Eq a where
      (==), (/=) ::  a -> a -> Bool
```

Esta declaración establece que la clase Eq contiene dos funciones miembro

- Tenemos que dar una *declaración de concrección*

Example

```
instance Eq Bool where
      (x == y) = (x && y) || (not x && not y)
      (x /= y) = not (x == y)
```

Esta declaración concreta que el tipo Boole es un tipo de la clase Eq, introduciendo definiciones específicas de las dos funciones para argumentos booleanos.

Clases de tipos en Haskell

- Los operadores de orden también están sobrecargados, siendo miembros de la clase Ord de tipos ordenados.
- Ord es una subclase de Eq: solamente se pueden dotar de un orden a los tipos con igualdad
- Se dan definiciones por defecto de los operadores $\leq$, $\geq$ y $>$, en términos de $<$, por lo que basta con definir $<$ en cada concreción.

Example

```
class Eq a => Ord a where
      (<),(<=),(>=),(>) ::  a -> a -> Bool
      (x <= y) = (x < y) || (x == y)
      (x >= y) = (x > y) || (x == y)
      (x > y) = not (x <= y)
```

Clases de tipos en Haskell

- Podemos declarar que Bool es una concreción de Ord escribiendo:

```
instance Ord Bool where
      (True < True) = False
      (True < False) = False
      (False < True) = True
      (False < False) = False
```

- Se puede establecer la igualdad y el orden entre caracteres haciendo uso de la función primitiva `ord::Char -> Int`, que asigna un número de orden a un carácter, y del hecho de que Int es un tipo con igualdad y orden:

```
instance Eq Char where
      (x == y) = (ord x == ord y)
      (x /= y) = not (ord x == ord y)
instance Ord Char where
      (x < y) = (ord x < ord y)
```

Reglas de tipificación e inferencia de tipos

- Las reglas de tipificación son las que permiten discernir cuáles son las expresiones bien tipificadas del lenguaje de programación.
- Asumiendo que el tipo de las expresiones básicas (constantes y variables) es conocido, se establecen reglas para derivar el tipo de las expresiones más complejas.

Example

Regla de tipificación para la aplicación de una función f sobre una expresión E.

$$\frac{\mathsf{f} :: \tau_1 \rightarrow \tau_2,\ \mathsf{E} :: \tau_1}{\mathsf{f(E)} :: \tau_2}$$

Reglas de tipificación e inferencia de tipos

Example

Regla de tipificación para tuplas.

$$\frac{E_1 :: \tau_1,\ E_2 :: \tau_2, \ldots,\ E_n :: \tau_n}{(E_1, E_2, \ldots, E_n) :: \tau_1 \times \tau_2 \times \ldots \times \tau_n}$$

donde $E_1, E_2, \ldots, E_n$ son expresiones.

Example

Regla de tipificación para la función if.

$$\frac{\text{cond} :: \text{Bool},\ M :: \tau,\ N :: \tau}{\text{if cond then M else N} :: \tau}$$

Reglas de tipificación e inferencia de tipos

- Muchos lenguajes de programación funcional incorporan mecanismos de inferencia de tipos.
- Éstos nos permiten omitir la presentación explícita del tipo de una función (polimórfica). En su lugar, el sistema es capaz de inferir el tipo principal de la función.
- El *tipo principal* es el tipo más general que puede asociarse a un símbolo de función (polimórfica) de modo que sea compatible con la definición dada para la misma a partir de las ecuaciones que la definen y el sistema de tipos empleado.
- El tipo de cualquier ocurrencia de una función (polimórfica) debe ser una instancia del tipo principal.

Reglas de tipificación e inferencia de tipos

- Inferencia del tipo principal de una función (polimórfica): aproximación informal.

Example

Supongamos una función min definida como

```
min(a,b) = if a<b then a else b.
```

y para la que no se ha dado su tipo.

1. min :: $(\alpha, \beta) \rightarrow \gamma$, ya que min actúa sobre una 2-tupla.
2. a :: α, b :: β, implica que $\alpha = \beta$, porque las ramas del condicional deben de ser del mismo tipo.
3. $\gamma = \alpha$, ya que el tipo del resultado de la función debe ser igual al de las ramas del condicional.

Así, el tipo principal de min es: min :: $(\alpha, \alpha) \rightarrow \alpha$, siendo α un tipo de la clase Ord.

Tipos algebraicos

- Aparte de los tipos básicos, se necesitan recursos que permitan al programador la definición de nuevos tipos.
- Los *tipos de datos algebraicos* proporcionan esos recursos para expresar un gran rango de ideas.
- Son una combinación de tipos:
  - producto (tuplas o registros): agrupaciones heterogéneas de otros tipos.
  - unión (suma o variantes): se generan mediante constructores; cada variante tiene su constructor; cada constructor etiqueta un tipo producto.
- Más precisamente, son un tipo suma de tipos producto.

Tipos algebraicos

- Haskell posee una cláusula data que en una misma declaración describe:
  - un *constructor de tipo*, que da nombre al tipo definido,
  - sus *constructores de datos*, que sirven para describir los *valores* asociados a ese tipo.

Example

```
data Color = Red | Green | Blue
```

Constructor de tipo (constante): `Color`.
Constructores de datos (constantes): `Red`, `Green` y `Blue`.

Tipos algebraicos

Example

```
    data TreeInt =
NilTree | Leaf Int | Node TreeInt Int TreeInt
```

Constructor de tipo (constante): `TreeInt`.
Constructores de datos: `NilTree` (constante), `Leaf` (unario) y `Node` (ternario) que toma como argumentos un número entero y dos árboles `TreeInt`.

Example

Un tipo polimórfico: Paramétrico con respecto al tipo `t` (arbitrario).

```
    data List t = Nil | Cons t (List t)
```

Constructor de tipo (unario): `List`.
Constructores de datos: `Nil` (constante), y `Cons` (binario) que toman como argumentos un elemento y una lista `List t`. `t` hace las veces de una variable de tipo.

Tipos algebraicos

Example

Consideremos algunos ejemplos de valores o datos de los tipos anteriores:

- `Red::Color`.
- `Node (Leaf 0) 3 (Node (Leaf 0) 2 (Leaf 1))`:: `TreeInt`.
- `Cons 0 (Cons 1 (Cons 2 Nil))`::`List Int`.
  `Cons Red (Cons Green Nil)`::`List Color`.

- Las listas se representan abreviadamente empleando los corchetes `[]` (`[]` $\equiv$ `Nil`).
- También está muy extendido el operador infijo ‘`:`’ que es equivalente a `Cons` (`Cons x xs` $\equiv$ `x:xs`).

Example

`[0,1,2]`, `[Red,Green]`, `0:1:2:[]` y `Red:Green:[]`.

Tipos algebraicos

- Es evidente que la noción de tipo algebraico está estrechamente vinculada a la de álgebra (de ahí el nombre).
- La noción de *álgebra* es simple: un conjunto de valores con una colección de funciones que operan sobre esos valores.
- Todo elemento de un tipo algebraico se puede describir como:
  - un valor básico del tipo (un constructor de datos constante); o
  - una combinación de valores básicos de ese u otros tipos mediante operaciones propias de ese tipo (los constructores de aridad no nula).
- Hay dos propiedades que distinguen a los constructores de otras funciones:
  - No tienen una definición asociada. los constructores sólo construyen.
  - Pueden aparecer como patrones en la parte izquierda de la definición de una función (ver más adelante).

Disciplina de tipos

- En la programación funcional, el universo de valores está dividido en tipos.
- Cada tipo tiene asociado un conjunto de operaciones (no significativas para otros tipos)
- Toda expresión (sintáctica) tiene asociada un tipo, que puede inferirse de sus partes constituyentes.
- Estos principios caracterizan la *Disciplina de Tipos*.

Observación

Una ventaja de la disciplina de tipos es que encamina al programador a un modo de pensamiento consistente en plantearse los tipos apropiados de los valores y funciones que esta definiendo antes de escribir las propias definiciones.

Índice

Definiciones

- En esta sección mostramos como las definiciones y las expresiones forman la base de la programación funcional y su lenguaje de *expresiones de programa*.
- Podemos ver un *programa funcional* como un conjunto de definiciones de función y nombres que permiten escribir expresiones mediante la composición de las mismas y su aplicación sobre otras expresiones.
- Un programa funcional no requiere de otros comandos o construcciones (habituales en los lenguajes convencionales).

Observación

Hay autores que ven un programa funcional *como un conjunto de ecuaciones de definiciones de función más una expresión básica (i.e., sin variables), a evaluar. La expresión a evaluar se denomina* expresión de entrada (input expression) *o* expresión inicial (initial expression) *y se identifica con el componente dinámico del programa.*

Nombres, enlaces, entornos y alcance

- Las definiciones permiten asociar un nombre con un valor. El nombre se dice que ha sido *enlazado* al valor por la definición.

Example

```
pi = 3.1416
```

En Haskell un nombre se comporta como una constante de un lenguaje convencional.

- Una colección de enlaces (de nombres a valores) se denomina *entorno*. Las expresiones se evalúan en el contexto de un entorno.
- Las funciones, como valores, también pueden enlazarse a un nombre.
- El *alcance* de un nombre es el ámbito en el que ese nombre es visible y puede ser referenciado. El alcance de un nombre es el entorno en el que se define. Por ejemplo, el alcance de una función (de primer nivel) es todo el programa.

Definición de funciones

- Las funciones de usuario se definen, por medio de ecuaciones: $l = r$
  - l se llama la *parte izquierda* (*left-hand side* o *lhs*)
  - r se llama la *parte derecha* (*right-hand side* o *rhs*).
- Definición mediante *parámetros formales*: Cuando una función `f` se describe mediante ecuaciones de la forma

$$f \ x_1 \ \cdots \ x_k \ = \ E(x_1, \ldots, x_k)$$

siendo $x_1, \ldots, x_k$ variables distintas, llamadas parámetros formales, y $E(x_1, \ldots, x_k)$ una expresión dependiente de esos parámetros formales.

Example (Definiciones de función mediante parámetros formales.)

```
doble x = x+x          --Funciones primitivas
triple x = 3*x         --Funciones primitivas
seisveces x = doble (triple x)
fact n = if n==0 then 1 else n*(fact (n-1))
```

Observad el uso de funciones primitivas (`+`, `*`, `if then else`) y recursividad.

Definición de funciones

- Definición mediante parámetros formales y *condiciones* (guardas):
  Cuando una función f se describe mediante ecuaciones de la forma

$$f \ x_1 \ \cdots \ x_k \mid c \ = \ r$$

  $x_1, \ldots, x_k$ son variables distintas entre sí y las únicas que aparecen libres en la parte condicional c y en r. La condición c es una expresión booleana.

Example (Def. de función mediante parámetros formales y condiciones guardadas.)

```
fact n | n==0 = 1
       | n>0 = n*fact (n-1)

sign x | x<0 = Neg
       | x==0 = Cero
       | x>0 = Pos
```

Definición de funciones

- Definición mediante *parámetros formales* y *expresiones case*: Cuando una función f se describe mediante ecuaciones de la forma

$$\begin{array}{lcl} f\ x_1\ \cdots\ x_k & = & \texttt{case}\ \ x_i\ \ \texttt{of} \\ & & \begin{array}{lcl} p_1 & \texttt{->} & r_1 \\ & \vdots & \\ p_n & \texttt{->} & r_n \end{array} \end{array}$$

  donde $1 \leq i \leq k$ y $p_1, \ldots, p_n$ son patrones del mismo tipo que el parámetro formal x_i escogido para realizar la distinción por casos.
- Un *patrón* (*pattern*) no es más que un término constructor, que puede incluir variables, y que representa un conjunto de valores de un tipo determinado: todos aquellos que son instancia de él.

Definición de funciones

Example (Def. mediante parámetros formales y expresiones case)

```
tail y = case y of
                [] -> []
                _:xs -> xs
length y = case y of
                [] -> 0
                x:xs -> 1 + (length xs)
fac n = case n of
                0          -> 0
                m | m>0 -> m * (fac m-1)
```

Observación

En esencia, los métodos anteriores realizaban un filtrado de los parámetros de la función, asignando el mismo tratamiento a grupos de parámetros que satisfagan ciertas condiciones. Esto corresponde a definir la función describiendo cómo actúa ésta sobre distintas partes de su dominio.

Definición de funciones mediante ajuste de patrones

- Al plantearse la descripción de funciones definidas sobre dominios representados mediante tipos de datos algebraicos resulta natural considerar el dominio de la función como la unión de los subdominios expresados por los componentes del tipo algebraico.
- Una técnica interesante para alcanzar este objetivo es la utilización de patrones.
- Al definir una función empleando *ajuste de patrones* (*pattern matching*), las ecuaciones de definición serán de la forma:

$$\begin{array}{llll} f & p_{11} & \cdots & p_{1k} & = & r_1 \\ \vdots & & & & & \\ f & p_{n1} & \cdots & p_{nk} & = & r_n \end{array}$$

  donde los patrones $p_{i1}, \ldots, p_{ik}$ no contienen (en conjunto) múltiples ocurrencias de una misma variable.
- Constituye una alternativa al uso de expresiones condicionales, cuando es necesario emplear una multiplicidad de casos al definir una función.

Definición de funciones mediante ajuste de patrones

Example

Definiendo el tipo de tipo de datos `Nat` como:

```
data Nat = Cero | S Nat
```

se puede especificar la función `first`, que obtiene los *n* primeros elementos de una lista, como sigue:

```
first Cero x = []
first (S n) (x:xs) = x:(first n xs)
```

En la definición dada para `first`:

- Cada combinación de patrones en una *lhs* corresponde a una parte del dominio global donde la función está definida.
- Las combinaciones de patrones empleadas no cubren todas las posibles combinaciones de argumentos para la función. (la llamada `first (S Cero) []` 'no tiene sentido')
- Esto significa que `first` es una función parcial.

Definición de funciones mediante ajuste de patrones

- En general, en una definición mediante ajuste de patrones podemos usar cualquier combinación de constructores, tuplas y variables en un patrón, y cualquier número de casos en la definición, pero teniendo en cuenta las siguientes restricciones:
  - Los patrones deben tener tipos compatibles y también las expresiones.
  - Si se desean funciones totales, los casos deben ser exhaustivos, con patrones que cubran todo posible argumento.
  - No debe haber ambigüedad sobre el caso que aplica a cada instancia: supondremos una ordenación de arriba abajo ("top-down"); si más de un patrón empareja con un argumento, elegimos el caso que está primero.
  - Evitar la repetición de variables en el conjunto de los patrones de un caso $\Longrightarrow$ test implícito de igualdad, que se prefiere explícito.

Observación

Le función `error` *permite el control de casos sin sentido en las definiciones de funciones parciales.*

Definición de funciones mediante ajuste de patrones

- Cuando se utiliza el ajuste de patrones para definir las funciones, *las variables* que aparecen en las partes izquierdas de las ecuaciones *no desempeñan el papel de parámetros formales* en el sentido usual.
- Sirven para etiquetar subexpresiones de las expresiones a evaluar que serán procesadas en la parte derecha de la ecuación involucrada.
- Esto se formaliza mediante la noción de sustitución de *ajuste* (*matching*).
- Una *sustitución de ajuste* (o *emparejamiento*), para dos expresiones t y s es una sustitución σ tal que $s = \sigma(t)$. Entonces decimos que s *se ajusta a* (o encaja en) t (o que s es instancia de t).

Definición de funciones mediante ajuste de patrones

Example

Si $t \equiv f(x, g(y))$ y $s \equiv f(h(b), g(a))$, $\sigma = \{x/h(b), y/a\}$ es la sustitución que empareja t con s.

- **Resumiendo**:
  - El ajuste de patrones puede fallar, tener éxito o divergir.
  - Un *emparejamiento de éxito* enlaza los parámetros formales del patrón.
  - La divergencia se produce cuando un valor que se necesita contiene un error.
  - El proceso de emparejamiento sucede de arriba abajo ("top-down") al inspeccionar las ecuaciones y de izquierda a derecha ("left-to-right") al inspeccionar la expresión que se empareja.

Ajuste de patrones y expresiones case

- La definición genérica de una función por ajuste de patrones es semánticamente equivalente a:

$$
\begin{array}{lcll}
f\ x_1\ \cdots\ x_k & = & \texttt{case}\ (x_1,\ldots,x_n) & \texttt{of} \\
 & & (p_{11}\ \cdots\ p_{1k}) & \texttt{->}\quad r_1 \\
 & & & \vdots \\
 & & (p_{n1}\ \cdots\ p_{nk}) & \texttt{->}\quad r_n
\end{array}
$$

Observación

Una expresión condicional `if cond then e1 else e2` *es un caso particular de expresión case. Se utiliza una notación especial debido a su uso tan común. Una expresión condicional se corresponde con:*

$$
\begin{array}{lll}
\texttt{case}\ \texttt{cond} & \texttt{of} & \\
\quad \texttt{True} & \texttt{->} & e_1 \\
\quad \texttt{False} & \texttt{->} & e_2
\end{array}
$$

Definiciones locales

- En ocasiones, es útil tener de un medio permitir *definiciones locales* a una expresión.
- Haskell dispone de la expresión `let` y la cláusula `where` para realizar tal fin, consiguiendo una cierta "estructura de bloques".
- Estas construcciones asocian nombres a expresiones y su sintaxis es como sigue:

```
let Def1 ...  Defn in e
l = r where Def1 ...  Defn
```

donde `Def1 ... Defn` son definiciones que:

  - asocian un nombre `vi` a una expresión `ei` (`vi = ei`);
  - declaran una función `f` (`f p1...pk = ri`);
  - el signo `=` es una igualdad matemática y no debe confundirse con la asignación destructiva de los lenguajes convencionales.
  - el orden en el que aparecen las definiciones no importa y es posible la recursión mutua.

Definiciones locales

Example (Raíces de una ec. de segundo grado: $ax^2 + bx + c = 0$)

```
roots a b c =
   let disc = b*b - 4*a*c
       a2 = 2*a
   in if disc > 0
         then ((-b + (sqrt disc)) / a2, (-b - (sqrt disc)) / a2)
         else error("discriminante negativo")
```

Example (Raíces de una ec. de segundo grado: $ax^2 + bx + c = 0$)

```
roots a b c =      if disc > 0
         then ((-b + (sqrt disc)) / a2, (-b - (sqrt disc)) / a2)
         else error("discriminante negativo")
                  where disc = b*b - 4*a*c; a2 = 2*a
```

- `let` y `where` evitan el cálculo repetido de ciertas expresiones (las enlazadas con los nombres `disc` y `a2`) $\Longrightarrow$ mejoran la eficiencia
- También mejoran la legibilidad de las definiciones de función.

Definiciones locales: alcance

Example

$$\texttt{let}\quad \begin{array}{ll} v_1 = & \boxed{e_1} \\ v_2 = & \boxed{e_2} \\ \vdots & \\ v_n = & \boxed{e_n} \end{array} \quad \texttt{in}\quad \boxed{\texttt{e}}$$

- `let` califica expresiones. La propia `let` es una expresión.
- El alcance de cada v_i es el de las expresiones recuadradas.

Example

$$\texttt{lhs = }\boxed{\texttt{rhs}}\ \texttt{where}\quad \begin{array}{ll} v_1 = & \boxed{e_1} \\ v_2 = & \boxed{e_2} \\ \vdots & \\ v_n = & \boxed{e_n} \end{array}$$

- `where` califica definiciones de función, de hecho, `where` es parte de la sintaxis para la declaración de funciones.
- El alcance de cada v_i es el de las expresiones recuadradas.

Índice

Definiciones recursivas

- Una *función recursiva* es aquella que se llama a si misma. Esto es, el nombre que se define aparece en el cuerpo (lhs) de la definición.
- En un lenguaje de programación funcional la recursión es el medio para conseguir repetición, contrariamente a lo que sucede con los lenguajes convencionales, que usan principalmente iteración sobre un estado mutable.

Example (Suma de los `n` primeros enteros positivos)

```
function sum(n:  integer):  integer;
var      counter, acc:  integer;
begin
         acc := 0;
         for counter := 1 to n do
                  acc := acc + counter;
         sum := acc
end;
```

Definiciones recursivas

- Podemos diseñar una solución recursiva, para el problema anterior, mediante el siguiente análisis:

```
(1)      sum(0)= 0
(2)      sum(1)= 1+0
(3)      sum(2)= 2+1+0
           .
           .
           .
(n)      sum(n-1)= (n-1)+...+2+1+0
(n+1)    sum(n)= n+ (n-1)+...+2+1+0
                 = n+sum(n-1)
```

- La ecuación (1) constituye el caso base y proporciona una condición de terminación.
- La ecuación (n+1) constituye el caso general.

Definiciones recursivas

- Las ecs (1) y (n+1) proporcionan una definición de `sum` mediante ajuste de patrones.

Example (Suma de los `n` primeros enteros positivos)

```
sum(0) = 0
sum(n) = n + sum(n-1)
```

- Es posible una definición mediante el empleo del la expresión condicional.

Example (Suma de los `n` primeros enteros positivos)

```
sum(n) = if n=0 the 0
                else n + sum(n-1)
```

Definiciones recursivas

- Características de la solución funcional:
  - Es una declaración del problema: pensamos en qué evaluar, más bien que en el modo en el que se evalúa.
  - No existe definida, de forma explícita, una noción de estado.
- La definición de `sum` se apoya sobre el hecho de que los objetos de su dominio (los naturales) admiten una definición inductiva.
- Una *definición inductiva (o generativa) de un conjunto de objetos* está compuesta de:
  1. Una descripción del conjunto inicial de objetos.
  2. Un mtodo para la generación de nuevos elementos, basado en el conjunto inicial de objetos.
  3. Una cláusula de terminación: solamente son elementos del conjunto, aquéllos generados a partir de (1) y (2).
- Las funciones sobre dominios definidos inductivamente, aceptan, de forma natural, definiciones recursivas.

Inducción y propiedades de programas

- Existe una estrecha relación entre las técnicas de diseño de funciones recursivas y el principio de inducción matemática, lo que facilita la prueba formal de propiedades de programas.

Definition (Inducción Matemática)

Sea $P(n)$ un enunciado sobre números naturales.

$$(\underbrace{P(0)}_{\text{Base}} \wedge (\underbrace{\forall m > 0 \text{ in } \mathbb{N}.\ \overbrace{P(m-1)}^{\text{H. Ind.}} \Rightarrow P(m)}_{\text{Paso de Inducción}})) \Rightarrow \forall n \in \mathbb{N}.\ P(n)$$

- Apliquemos el principio de inducción matemática a la demostración de corrección de algunos programas.

Example (Justificación de que la def. de `sum` se ajusta a su especificación)

El enunciado $P(n)$ sobre el que se aplica el principio de inducción matemática es la ecuación: `sum(n)` $= \sum_{i=0}^{n} i$.

Caso base ($n = 0$):

$$\sum_{i=0}^{0} i = 0 = sum(0)$$

Caso inductivo ($n = k > 0$):

$$\begin{array}{lll} \sum_{i=0}^{k} i & = k + \sum_{i=0}^{k-1} i & \text{por definición de } \Sigma \\ & = k + sum(k-1) & \text{por hipótesis de inducción} \\ & = sum(k) & \text{por definición de sum} \end{array}$$

Por consiguiente, se cumple $P(n)$ para todo $n \in \mathbb{N}$.

Inducción y propiedades de programas

Example (Justificación de que `fac` se ajusta a su especificación)

Sea la siguiente definición de una función que implementa el factorial $n!$ de un entero positivo n: `fact n = if n==0 then 1 else n*fact (n-1)`
El enunciado que expresa la corrección de esta definición (respecto a la noción matemática de factorial) es el siguiente: $P(n) \equiv$ `(fact n = n!)`.
Caso base ($n = 0$):

```
fact 0  = if 0==0 then 1 else 0 * fact (0-1)
        = if True then 1 else 0 * fact (0-1) = 1 = n!
```

Caso inductivo ($n = k > 0$):

```
fact k  = if k==0 then 1 else k*fact (k-1)   def. fun. fact
        = k * fact (k-1)                     porque k > 0
        = k *(k-1)!                          por hip. de induc.
        = k!                                 def. factorial
```

Por consiguiente, se cumple $P(n)$ para todo $n \in \mathbb{N}$.

Inducción y propiedades de programas

- Otra cuestión de interés es la de demostrar que programas distintos expresan la misma función (equivalencia semántica).
- Esto es útil cuando uno de ellos tiene mejor comportamiento operacional.

Example (La función de Fibonacci)

Consideremos el siguiente programa funcional que puede verse como una especificación ejecutable de la función de Fibonacci:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)          -- n > 1
```

Un algoritmo más eficiente vendría dado por la siguiente función `fib'`, que usa parámetros de acumulación:

```
fib' a b 0 = a
fib' a b n = fib' b (a+b) (n-1)          -- n ≠ 0
```

Example (Equivalencia semántica de `fib` y `fib'`)

Estamos interesados en establecer que:
`fib' 0 1 n = fib n, para todo n`≥ 0.
Con tal fin, debemos probar un resultado más general: $P(n, i) \equiv$ `(fib' (fib i) (fib i+1) n = fib (i+n))`, para $n \geq 0$ e $i \geq 0$.
Realizaremos la prueba por inducción.

Caso base ($n = 0$): `fib' (fib i) (fib i+1) 0 = fib i, para` $i \geq 0$
Caso inductivo ($n = k > 0$):

```
fib' (fib i) (fib i+1) k
 = fib' (fib i+1) (fib i+ fib (i+1)) (k-1)   def.  fun.  fib'
 = fib' (fib i+1) (fib i+2) (k-1)            def.  fun.  fib
 = fib (i+1+k-1) = fib (i+k)                 por hip.  induc.
```

Por consiguiente, se cumple $P(n)$ para todo $n \in \mathbb{N}$ e $i \geq 0$. Ahora, utilizando este resultado con $i = 0$:

`fib' 0 1 n = fib n, para todo n`≥ 0.

Inducción y propiedades de programas

- Cuando una función puede definirse mediante un computo, éste es preferible a una definición recursiva (o incluso iterativa).

Example (Cómputos frente a repetición)

Recordemos que la especificación de `sum(n)`=$\sum_{i=0}^{n} i$, por tanto:

```
  sum(n) = 0 +    1  +    2  + ..... + (n-1) + n
  sum(n) = n + (n-1) + (n-2) + ..... +    1  + 0
-----------------------------------------------------
2 sum(n) = n +    n  +    n  + ..... +    n  + n =n(n+1)
```

Lo que indica que es preferible definir `sum`, como sigue:

```
sum(n) = n(n+1)/2
```

que es la definición óptima.

Inducción y propiedades de programas

- También podemos emplear inducción matemática para comprobar que la nueva definición de `sum` se ajusta a su especificación

Example (Justificación de que `sum` se ajusta a su especificación)

En este caso, el enunciado $P(n)$ sobre el que se aplica el principio de inducción matemática es la ecuación: $\sum_{i=0}^{n} i = n(n+1)/2$.

Caso base ($n = 0$):

$$\sum_{i=0}^{0} i = 0 = 0(0+1)/2$$

Caso inductivo ($n = k > 0$):

$$\begin{array}{lll} \sum_{i=0}^{k} i & = k + \sum_{i=0}^{k-1} i & \texttt{por definición de } \Sigma \\ & = k + (k-1)k/2 & \texttt{por hipótesis de inducción} \\ & = k(k+1)/2 & \texttt{operando} \end{array}$$

Por consiguiente, se cumple $P(n)$ para todo $n \in \mathbb{N}$.

Inducción y propiedades de programas

- En programación funcional también se emplean con frecuencia los principios de *inducción completa* e *inducción estructural*, que generalizan el principio *inducción matemática*.

Definition (Inducción completa)

Sea $(A, <)$ un orden parcial estricto bien fundado y $P(x)$ un enunciado sobre elementos de A.

$$\forall x \in A.\ P(x) \text{ si y sólo si } \forall x \in A.([\forall y < x.\ P(y)] \Rightarrow P(x))$$

Observación

Sea un conjunto A sobre el que se define una relación binaria $< \subseteq A \times A$ transitiva e irreflexiva. $(A, <)$ es un orden parcial estricto.
Si, además, para todo par de elementos distintos $x, y \in A$ o bien $x < y$ o bien $y < x$, entonces decimos que $(A, <)$ es un orden total estricto.
Un orden parcial estricto $(A, <)$ es un orden bien fundado si no hay cadenas decrecientes infinitas.

Inducción y propiedades de programas

- El método de inducción estructural nos permite establecer propiedades acerca de los valores de un tipo de datos algebraico.

Definition (Inducción estructural)

Sea $P(x)$ un enunciado sobre valores x del tipo τ. Entonces, $P(x)$ es cierto para todo x de tipo τ si y sólo si puede probarse que:

1. **Casos base**:
   1. $P(C)$ se cumple para todo símbolo constructor constante C del tipo τ,
   2. $P(C(x_1, \ldots, x_k))$ se cumple para todo símbolo constructor k-ario C del tipo τ cuyos componentes no contienen elementos del tipo τ,
2. **Casos inductivos**: para todo constructor k-ario C del tipo τ, cuyos argumentos son de tipo $\tau_1, \ldots, \tau_k$, entonces siempre se sigue $P(C(x_1, \ldots, x_k))$ si se toman como ciertas $P(y_1), \ldots, P(y_n)$ para los miembros $y_1, \ldots, y_n$ de tipo τ en los valores $x_1, \ldots, x_k$ de los tipos $\tau_1, \ldots, \tau_k$ respectivamente (hipótesis de inducción).

Inducción y propiedades de programas

- Es una forma de razonamiento basada en la estructura de los datos definidos en los tipos de datos algebraicos.
- Resulta especialmente útil para demostrar propiedades de funciones definidas mediante ajuste de patrones.

Example

Consideremos el tipo de datos

```
data Tree t = Nil | Leaf t | Node (Tree t) t (Tree t)
```

y la función `reflect` definida sobre el tipo anterior mediante las siguientes ecuaciones:

```
reflect Nil            = Nil
reflect Leaf x         = Leaf x
reflect (Node l x r) = Node (reflect r) x (reflect l)
```

Deseamos demostrar la siguiente propiedad:

$P(a)$= '(Para todo a::Tree t), reflect (reflect a) = a'.

Inducción y propiedades de programas

Example (Cont.)

Realizamos la demostración por inducción estructural:

1. Caso base (`a = Nil`):
   `reflect (reflect Nil) = reflect Nil = Nil`.
2. Caso base (`a = Leaf x`):
   `reflect (reflect Leaf x) = reflect Leaf x = Leaf x`.
3. Caso inductivo (`a = Node l x r`):

```
reflect (reflect Node l x r)
    = reflect (Node (reflect r) x (reflect l))                (p
    = Node (reflect (reflect l)) x (reflect (reflect r))      (p
    = Node (l x r)                                            (p
```

   Por tanto, por inducción estructural, $P(a)$, para `todo a::Tree t`.