UNIVERSIDAD CAECE

Carreras de “Ingeniería y Licenciatura en Sistemas”

Cátedra de “Algoritmos y estructuras de datos II”

Apunte “Tipos de datos de construcción”

por Sergio Amitrano

Indice

Introducción

Al utilizar la especificación de TADs con sus proyectores y operaciones en forma axiomática, obliga a definir el tipo de datos que sirve de apoyo al TAD a través de constructores. Sin embargo, esta definición de tipos de datos a través de constructores no termina siendo una verdadera definición de tipos, ya que no construye un tipo de datos verdadero, sino diversas formas de combinar en cada "situación destacada" (utilizando cada uno de los constructores sobre los que se define) expresiones de otros tipos existentes -en los casos en que se aplica a una función constructora-. Para el caso del uso de constantes constructoras, esta expresión de otro tipo existente que sirve como base no existe.
Como ejemplo de estas "situaciones destacadas" sobre el tipo de datos `List(a)`, podemos mencionar a la construcción del elemento lista vacía (con la constante constructora `EmptyList`) y a la construcción del elemento lista no vacía (con la función constructora `ConsList`, a partir de un elemento -la cabeza- y una lista -la cola o resto-, que conforman un elemento del tipo `a × List(a)`). Estas dos corresponden a las situaciones destacadas antes mencionadas sobre el tipo de datos `List(a)`.

El problema que se plantea aquí es que la especificación de tipos a través de varios constructores es algo muy propio de las especificaciones axiomáticas-funcionales, y donde no tiene una representación equivalente ni en otros tipos de especificaciones, ni a nivel de implementación. Por este motivo, se intentará "rever" la aplicación del criterio de especificación de un tipo de datos, definiéndolo solamente a partir de una única función constructora, que tome como base (dominio) una expresión de un tipo de datos, sin utilizar varios constructores que sean funciones o constantes. Ese tipo de datos que representa el dominio del ahora único constructor (función constructora) del TAD a especificar, podríamos decir que es el tipo de datos que sirve como base del TAD que se está definiendo. Esto significa que todo elemento abstracto del TAD que se está especificando, deberá mapearse a un elemento del tipo de datos que sirve como base. En el mejor de los casos, si existiera una biyección entre el tipo abstracto del TAD y el tipo de datos que sirve como base, se dice que la definición del TAD es por *equivalencia de tipos* respecto al tipo de datos que sirve como base. Esto último es deseable que ocurra, aunque no siempre se puede logar en todas las definiciones. La función constructora, al ser el único constructor, dejará de tener sentido para la especificación, pero seguirá sirviendo solamente para lograr el encapsulamiento del nuevo tipo, evitando que se utilicen las operaciones propias del tipo de datos que sirve como base como operaciones del nuevo tipo.

El criterio de definir los nuevos tipos del TAD utilizando tipos que sirven como base (logrando equivalencia de tipos o no) también ayuda a comprender cómo se llega a la representación del tipo de implementación (especialmente para los tipos recursivos), ya que en muchos casos, su implementación más simple se basa a partir de la implementación del tipo que sirve como base.

Tipos de construcción

Para poder construir tipos de datos que sirvan como base para las representaciones de los TADs, será necesario que existan ciertos mecanismos de construcción de tipos de datos que permitan "simular" la existencia de diversos tipos de constructores que se usarían para definir el tipo de datos que se necesita. La idea es que estos constructores de tipos permitan construir tipos de datos nuevos de la misma forma que los constructores crean cada una de las "situaciones destacadas" a partir de expresiones de tipos existentes.

Estos constructores de tipos corresponden a un grupo especial de tipos básicos y constructores de tipos -o tipos estructurados- que permiten la construcción de tipos de la misma forma que los constructores los construyen estableciendo estas situaciones destacadas. Podríamos decir que estos constructores de tipos (llamados *tipos de construcción*) corresponden a un criterio diferente de clasificación de tipos en básicos y estructurados, ya se basan en cómo construir tipos nuevos "en forma constructiva" a partir de tipos de datos previamente definidos, aumentando la cantidad de elementos construídos a partir de su forma de construcción.

Estos tipos de construcción se definen de una forma similar a cómo se definen constructivamente las estructuras de programación de un lenguaje estructurado clásico. Al igual que en los lenguajes estructurados, estos tipos de construcción deberían proveer el único mecanismo para construir tipos de datos. Se podrán utilizar en la construcción de nuevos tipos, otros tipos predefinidos o construídos con anterioridad, pero se asume que esos tipos ya fueron definidos en su momento a través de tipos de construcción.

El criterio de clasificación clásico de tipos de datos es el siguiente:

Tipos básicos	`bool`	Booleanos (lógicos)
	`integer`	Enteros
	`float`	Reales
	`char`	Caracteres
	`string`	Cadenas de caracteres
Tipos estructurados	`array[`n`] of T`	Arreglos
	`record` cpo_1`: T`$_1$`, …,` cpo_n`: T`$_n$ `end`	Registros
	`^T`	Punteros
	$T_1 \rightarrow T_2$	Funciones

El criterio de clasificación de los *tipos de construcción* (algo diferente del clásico) es el siguiente:

Tipos básicos		`Unit`
		Otros tipos predefinidos (básicos o estructurados)
Tipos estructurados	Secuencia	`Product(`T_1`, `T_2`)`
	Decisión	`Either(`T_1`, `T_2`)`
		`Maybe(T)`
	Iteración	*Tipos recursivos (tipos definidos recursivamente)*

Los tipos de construcción no se representan como TADs, sino como tipos de datos comunes sin encapsulamiento, ya que su representación es muy simple, posee unas pocas simples operaciones, y se permite poder acceder directamente a sus constructores (incluso como patrones dentro de otros tipos de TADs).

Cada uno de los tipos de construcción se describe a continuación:

Unit

Corresponde al único tipo de construcción básico, que se caracteriza por poseer un solo elemento. Pese a tener un solo valor como constante, el tipo está asociado al concepto de "no valor", y se lo utiliza con este fin. No tiene mucha importancia por sí solo, pero es fundamental cuando se lo combina con tipos de construcción estructurados, especialmente con `Product` y `Either`.

En varios lenguajes se lo suele asociar en cierto modo al tipo `void`, que representa el tipo de datos que representa los "no valores" (que no retornan valores).

La especificación del tipo `Unit` es la siguiente:

```
Tipo Unit
usa Bool
constructores
   Unit : Unit
operaciones
   _ = _ : Unit × Unit → Bool
axiomas
   Unit = Unit = True
```

Product(a,b)

Corresponde al tipo resultante de realizar el producto cartesiano de dos tipos `a` y `b`. Realiza la combinación de dos tipos existentes dados `a` y `b` para armar un tipo nuevo como la conjunción de ambos, conformado por elementos que representan pares de elementos, donde la primera coordenada del par es un elemento del primer tipo (`a`), y la segunda coordenada del par es un elemento del segundo tipo (`b`).

Así como existe este tipo representado como el producto cartesiano de dos tipos `a` y `b`, se podría generalizar la definición del mismo a *n* tipos.

La especificación del tipo `Product(a,b)` es la siguiente:

```
Tipo Product(a,b)
usa Bool
constructores
   Pair : a × b → Product(a,b)
proyectores
   getFirst : Product(a,b) → a
   getSecond : Product(a,b) → b
operaciones
   _ = _ : Product(a con operacion _=_, b con operacion _=_) ×
               Product(a,b) → Bool
axiomas
   getFirst(Pair(x, y)) = x
   getSecond(Pair(x, y)) = y
   Pair(x1, y1) = Pair(x2, y2) = (x1 = x2) and (y1 = y2)
```

El tipo `Product(a,b)` tiene prácticamente una definición análoga al tipo ya existente `a × b`.

Either(a,b)

Corresponde a uno de los tipos más importantes y característicos de los tipos de construcción, que representa lo que se llama la *unión disjunta* entre dos tipos existentes `a` y `b`. El resultado corresponde al equivalente resultante de unir todos los elementos de los tipos `a` y `b` (algo similar a que todos sus elementos serían o bien elementos de `a`, o bien elementos de `b` conformando algo parecido a lo que sería la disyunción de ambos tipos), pero "marcando" a cada uno de ellos con su tipo-parámetro de "origen". A los elementos provenientes de `a`, se le establecerá una "marca" que indica que proviene del primer tipo de los dos (esta marca es `Left`), y a los elementos provenientes de `b`, se le establecerá otra marca que indica que proviene del segundo tipo de los dos (esta marca es `Right`). De esta forma, si llegaran a existir elementos pertenecientes a ambos tipos (especialmente en el tipo `Either(a,a)`), cada uno existirá dos veces -porque proviene de los dos tipos-parámetro- uno con la marca `Left`, y el otro con la

marca `Right`. A estas marcas también se las conoce como *tags*, y al tipo `Either(a,b)`, al representar el tipo "unión" marcando los elementos de los tipos paramétricos con estos tags, también se lo conoce como *tagged union*.

Dicho se otra forma, se cumplen las siguientes reglas de pertenencia a este tipo:

$x \in a \Rightarrow$ `Left(`x`)` $\in$ `Either(`a`,`b`)`
$y \in b \Rightarrow$ `Right(`y`)` $\in$ `Either(`a`,`b`)`

Para este tipo existen funciones proyectoras para quitarle los tags a cada uno de sus elementos (una función proyectora que quita el tag `Left`, y otra que quita el tag `Right`). El objetivo de estos proyectores que eliminan el tag es retornar la expresión del tipo-paramétro de `Either(`a`,`b`)` para que se le puedan seguir aplicando operaciones de ese tipo origen. Sin embargo si se le aplicara a una expresión el proyector equivocado (la expresión provino del tipo a con lo cual tiene el tag `Left`, y se le aplicó la función proyectora que quita el tag `Right`, o viceversa), la evaluación provocará un error. En este caso, el sistema de tipos no puede evitar estas fallas.

Pese a la falla en el sistemas de tipos antes mencionada en relación a esta operación, la falla sería aún mayor si la operación de unión fuera sin uso de tags (lo que se denomina *untagged union*). Con este criterio, este tipo dejaría de ser monomórfico (que toda expresión tenga asociada exactamente un solo tipo de datos) para ser polimórfico, con algunas consecuencias negativas respecto al sistema de tipos de tipificación monomórfica.
Las reglas de pertenencia para el tipo *untagged union* (*`UntaggedUnion(`a`,`b`)`*) serían las siguientes:

$x \in a \Rightarrow x \in$ *`UntaggedUnion(`a`,`b`)`*
$y \in b \Rightarrow y \in$ *`UntaggedUnion(`a`,`b`)`*

Observando la definición, el tipo *`UntaggedUnion(`a`,`b`)`* también deja de ser una unión disjunta, ya que si alguna expresión perteneciera a los tipos a y b, solamente pertenecerá al tipo *`UntaggedUnion(`a`,`b`)`* solamente una vez.

Aquí se puede ver el polimorfismo de *`UntaggedUnion`*, al observar que la expresión x es tanto del tipo a como del tipo *`UntaggedUnion(`a`,`b`)`*. Lo mismo para la expresión y, que es tanto del tipo b como del tipo *`UntaggedUnion(`a`,`b`)`*. El gran problema de este polimorfismo es que las operaciones válidas para las expresiones del tipo *`UntaggedUnion(`a`,`b`)`* son todas las que se aplican para las expresiones del tipo a, más todas las operaciones que se aplican para las expresiones del tipo b. Esto es altamente peligroso, porque todas las operaciones que se apliquen sobre expresiones de tipos de datos a que participan en algún tipo *`UntaggedUnion(`a`,`b`)`* terminan siendo inseguras al poderse aplicar sobre expresiones de ese mismo tipo *`UntaggedUnion(`a`,`b`)`* (polimórfico en relación al tipo a) que realmente son del tipo b. Si el tipo unión fuera con tags (como `Either(`a`,`b`)`), las únicas operaciones que romperían el sistema de tipos son sólo sus dos proyectores que quitan los tags, y no todas las operaciones sobre a y todas las operaciones sobre b, como en el caso del tipo *`UntaggedUnion(`a`,`b`)`*.

Así como existe el tipo de la unión disjunta para dos tipos `a` y `b`, se podría generalizar la definición del mismo a *n* tipos.

La especificación del tipo `Either(a,b)` es la siguiente:

```
Tipo Either(a,b)
usa Bool
constructores
   Left : a → Either(a,b)
   Right : b → Either(a,b)
proyectores
   isLeft : Either(a,b) → Bool
   getLeft : Either(a,b) → a
   getRight : Either(a,b) → b
operaciones
   _ = _ : Either(a con operacion _=_, b con operacion _=_) ×
               Either(a,b) → Bool
axiomas
   isLeft(Left(x)) = True
   isLeft(Right(x)) = False
   getLeft(Left(x)) = x
   getLeft(Right(x)) = error
   getRight(Left(x)) = error
   getRight(Right(x)) = x
   Left(x1) = Left(x2) = (x1 = x2)
   Left(x1) = Right(y2) = False
   Right(y1) = Left(x2) = False
   Right(y1) = Right(y2) = (y1 = y2)
```

Maybe(a)

Corresponde a un tipo de construcción que representa un caso particular de `Either(a,b)`, donde un elemento de este tipo o bien pertenece al tipo-parámetro `a`, o bien corresponde a un valor constante especial (que indica un "no valor" o la no pertenencia al tipo-parámetro `a`). De ahí viene su nombre que significa "puede ser de tipo `a`". Al igual que el tipo `Either(a,b)`, las expresiones que pertenecen al tipo `a`, para que pertenezcan a `Maybe(a)` deberán contener un tag para que se diferencien de las expresiones del tipo `a` propiamente dichas, y también para evitar que existan las mismas fallas en el sistema de tipos que en *`UntaggedUnion`*`(a,b)`. Tiene el mismo problema en el sistema de tipos que en `Either(a,b)` con la función proyectora que elimina el tag, si la expresión no proviniera del tipo-parámetro `a` (que fuera la constante de "no valor").

La especificación del tipo `Maybe(a)` es la siguiente:

```
Tipo Maybe(a)
usa Bool
constructores
   Nothing : Maybe(a)
   Just : a → Maybe(a)
proyectores
   isNothing : Bool
   getValue : Maybe(a) → a
operaciones
   _ = _ : Maybe(a con operacion _=_) × Maybe(a) → Bool
axiomas
   isNothing(Nothing) = True
   isNothing(Just(x)) = False
   getValue(Nothing) = error
   getValue(Just(x)) = x
   Nothing = x = isNothing(x)
   Just(x) = Nothing = False
   Just(x) = Just(y) = (x = y)
```

Tipos recursivos

La recursión se expresa solamente en las definiciones de tipos nuevos a través de una definición de tipos por equivalencia, donde el tipo nuevo se define a partir de una expresión de tipos que tiene como subexpresión de tipos al mismo tipo que se está definiendo. Este caso representa una definición de un tipo recursivo directo. También pueden existir definiciones de tipos utilizando técnicas de recursión mutua, lo cual ambos tipos definidos son recursivos.

No existe un tipo de datos de construcción específico para los tipos recursivos, ni tienen definidas operaciones como tales. Los tipos de datos recursivos siempre se definen a partir de un tipo `Either(a,b)` o `Maybe(a)`, que determinan la existencia de un "caso base" de la definición recursiva.

Algunas formas de describir tipos recursivos (en este caso, con recursión directa -simple o múltiple-) serían las siguientes:

```
T ≡ Either(..., ...T...)
T ≡ Either(...T..., ...)
T ≡ Either(...T...T..., ...)
T ≡ Either(..., ...T...T...)
T ≡ Maybe(...T...)
T ≡ Maybe(...T...T...)
```

En el caso de la definición de un tipo recursivo `T` usando `Either`, el tipo `T` no podrá aparecer como subexpresión de tipos de *todos* los parámetros de ese `Either`, ya que alguno de sus parámetros deberá corresponder al "tipo de datos del caso base de la defínción recursiva de `T`" (a

menos que en algunos de sus tipos parámetros aparezca dentro de otra subexpresión de tipos `Either` o `Maybe`).

Como ya se mencionó antes, se permitirá utilizar los constructores de todos los tipos de datos de construcción (según lo especificado en sus tipos antes mencionados) desde las definiciones axiomáticas de otros TADs, incluso como patrones, dada la generalidad de los mismos.

Algunas definiciones y semejanzas estructurales

Se suelen definir ciertos nuevos tipos de datos en función de tipos de construcción conocidos. El objetivo de estas nuevas definiciones es extender algunas características de los tipos de datos de construcción ya conocidos.

Algunas definiciones son las siguientes:

$$\texttt{Product}(a, b) \equiv a \times b$$

Definición del tipo `Product(a, b)` como equivalente al tipo ya usado $a \times b$.

$$\texttt{Product}_2(T_1, T_2) \equiv \texttt{Product}(T_1, T_2)$$
$$\texttt{Product}_{n+1}(T_1, T_2, \ldots, T_n, T_{n+1}) \equiv \texttt{Product}(\texttt{Product}_n(T_1, T_2, \ldots, T_n), T_{n+1}) \quad (n \geq 2)$$

Definición generalizada de $\texttt{Product}_n(T_1, T_2, \ldots, T_n)$, para $n \geq 2$.

$$\texttt{Either}_2(T_1, T_2) \equiv \texttt{Either}(T_1, T_2)$$
$$\texttt{Either}_{n+1}(T_1, T_2, \ldots, T_n, T_{n+1}) \equiv \texttt{Either}(\texttt{Either}_n(T_1, T_2, \ldots, T_n), T_{n+1}) \quad (n \geq 2)$$

Definición generalizada de $\texttt{Either}_n(T_1, T_2, \ldots, T_n)$, para $n \geq 2$.

Se dice que dos tipos A y B son *estructuralmente semejantes* ($A \approx B$) si existe una correspondencia o mapeo biyectivo de cada una de las expresiones del tipo A a cada una de las expresiones del tipo B. La relación $\approx$ corresponde a una relación de equivalencia (es reflexiva, simétrica y transitiva).

La importancia de la semejanza estructural de un tipo A que se pretende definir, es que si se encontrara un tipo definido B que sea estructuralmente semejante a A, B puede ser utilizado como tipo que sirve como base para la definición de A por equivalencia de tipos.

Un buen criterio para encontrar la existencia de una biyección entre los tipos A y B es analizar la cantidad de elementos que poseen ambos tipos. Si dos tipos no recursivos (no infinitos) poseyeran la misma cantidad de elementos, entonces se puede afirmar que esa

biyección existe. O sea que con analizar la cantidad de elementos que posee un tipo de datos, basta para realizar un análisis de sus posibles semejanzas estructurales, porque no es fundamental expresar con exactitud cuál son las biyecciones con otros tipos para expresar estas semejanzas.

La cantidad de elementos de cada tipo de dato de construcción se define según lo indicado en la siguiente tabla:

CantidadElementos(`Unit`)	=	1
CantidadElementos(`Product(`T_1`, `T_2`)`)		CantidadElementos(T_1) * CantidadElementos(T_2)
CantidadElementos(`Either(`T_1`, `T_2`)`)		CantidadElementos(T_1) + CantidadElementos(T_2)
CantidadElementos(`Maybe(T)`)		1 + CantidadElementos(`T`)
CantidadElementos(*Tipo recursivo*)		∞

Algunas propiedades generales de las semejanzas estructurales son las siguientes:

$a \equiv b \Rightarrow a \approx b$

Las definiciones por equivalencia están incluídas en las semejanzas estructurales.

$a \approx a$

$a \approx b \Rightarrow b \approx a$

$a \approx b, b \approx c \Rightarrow a \approx c$

La semejanza estructural es *reflexiva* (existe el mapeo identidad entre un tipo y sí mismo, que es biyectivo), *simétrica* (si existe un mapeo biyectivo entre dos tipos, su mapeo inverso también es biyectivo) y *transitiva* (la composición de dos mapeos biyectivos da como resultado un mapeo también biyectivo). Como consecuencia, es una relación de equivalencia.

$a_1 \approx a_2, b_1 \approx b_2 \Rightarrow \texttt{Product}(a_1, b_1) \approx \texttt{Product}(a_2, b_2)$

$a_1 \approx a_2, b_1 \approx b_2 \Rightarrow \texttt{Either}(a_1, b_1) \approx \texttt{Either}(a_2, b_2)$

$a_1 \approx a_2 \Rightarrow \texttt{Maybe}(a_1) \approx \texttt{Maybe}(a_2)$

Los tipos estructurados `Product`, `Either` y `Maybe` preservan la semejanza estructural.

$\texttt{Product}(a, b) \approx \texttt{Product}(b, a)$

Conmutatividad del tipo `Product(a, b)`.

$\texttt{Product}(\texttt{Product}(a, b), c) \approx \texttt{Product}(a, \texttt{Product}(b, c))$

Asociatividad del tipo `Product(a, b)`.

```
Either(a, b) ≈ Either(b, a)
```

Conmutatividad del tipo `Either(a, b)`.

```
Either(Either(a, b), c) ≈ Either(a, Either(b, c))
```

Asociatividad del tipo `Either(a, b)`.

$\texttt{Product}_1\texttt{(a)} \approx \texttt{a}$
$\texttt{Either}_1\texttt{(a)} \approx \texttt{a}$
$\texttt{Product}_0 \approx \texttt{Unit}$
$\texttt{Either}_0 \approx \varnothing$ ($\varnothing$ es un tipo sin elementos -no es representable-)

Representaciones "extendidas" de los tipos $\texttt{Product}_n\texttt{(}T_1\texttt{, }T_2\texttt{, …, }T_n\texttt{)}$ y $\texttt{Either}_n\texttt{(}T_1\texttt{, }T_2\texttt{, …, }T_n\texttt{)}$ extendiendo el valor de sus subíndices a $0 \leq n < 2$, y preservando sus características esenciales. De todos modos, en la práctica no se usarán. Sólo se indican a efectos de mostrar los "casos base" de las definiciones de estos tipos "indexados" con un valor de *n* genérico.

Aquí se indican algunas características destacadas de las semejanzas estructurales que son de nuestro interés:

```
a ≈ Unit → a
```

Esta semejanza estructural se basa en la definición de la cantidad de elementos del tipo función, que expresa lo siguiente:

$$\text{CantidadElementos}(\texttt{a} \rightarrow \texttt{b}) = \text{CantidadElementos}(\texttt{b})^{\text{CantidadElementos}(\texttt{a})}$$

Lo importante que expresa esta semejanza estructural para nosotros es que toda constante constructora de un tipo `a` (genera un elemento constante del tipo `a`) puede sustituirse sin dificultades por una función constructora cuyo dominio sea `Unit` (tiene el mismo poder "estructural" una constante que sea de tipo `a` que una función que sea de tipo `Unit → a`).

```
Product(Unit, a) ≈ a
```

Lo importante que expresa esta semejanza estructural para nosotros es otra de las cualidades destacadas del tipo `Unit`, que es poder "eliminar" componentes variables de tipos-parámetro de un producto cartesiano, reemplazando ese tipo-parámetro variable por `Unit`.

`Either(a, a) ≈ Product(Bool, a)`

Representa una regla de reemplazo del tipo `Either(a, a)` por un tipo de datos similar donde el tipo `Either` no sea utilizado, evitando problemas de fallas en el sistema de tipos por el uso de sus proyectores que quitan los tags `Left` y `Right`.

$a \approx f(a),\ b \approx f(b) \Rightarrow a \approx b$

Representa la similitud por definición recursiva idéntica. Si dos tipos recursivos `a` y `b` se definen exactamente igual (utilizando el mismo "mecanismo de construcción" f para ambos), entonces ambos son estructuralmente semejantes.

$Set(a) \approx a \rightarrow Bool$

Representa una regla de similitud del tipo `Set(a)` (conjuntos de elementos de tipo `a`, donde no se consideran elementos repetidos y todos sus elementos no poseen una ubicación), por particularidades de los conjuntos.

Una regla importante que tiene que ver con el uso del tipo `Unit` es la siguiente:

CantidadElementos(`Either(Unit, a)`) =
CantidadElementos(`Unit`) + CantidadElementos(`a`) =
1 + CantidadElementos(`a`)

Esto muestra que el tipo `Unit` combinado con `Either` logra aumentar en un elemento al tipo original `a` en el nuevo tipo combinado a partir de estos tipos de datos.

Regla básica de conversión de constructores a tipos de construcción

Si `T` es un tipo de datos (recursivo o no recursivo) definido a través de los siguientes constructores (todos juntos):

- función constructora f_1 con dominio en T_1
- función constructora f_2 con dominio en T_2
  ...
- función constructora f_n con dominio en T_n

entonces $T \approx Either_n(T_1, T_2, \ldots, T_n)$

Por lo dicho anteriormente en las semejanzas estructurales, si alguno de los constructores de `T` fuera una constante en lugar de una función, se puede aplicar la siguiente conversión para ese constructor, para poder utilizar esta regla de conversión:

- constante constructora c_i

  $\Rightarrow$

- función constructora fc_i con dominio en `Unit`

Aplicando la regla básica de conversión de constructores a tipos de construcción, se pueden deducir las siguientes semejanzas (a partir de sus definiciones axiomáticas utilizando constantes y funciones constructoras):

```
Bool ≈ Either(Unit, Unit)

Constants_n ≈ Either_n(Unit, Unit, …, Unit)
                    (Constants_n se define con n constantes constructoras)

Maybe(a) ≈ Either(Unit, a)
```

Algunas definiciones recursivas que pueden surgir utilizando la regla básica de conversión de constructores a tipos de construcción pueden ser las siguientes:

```
List(a) ≈ Either(Unit, Product(a, List(a))) ≈
              Maybe(Product(a, List(a)))

Nat ≈ Either(Unit, Nat) ≈ Maybe(Nat) ≈ Maybe(Product(Unit, Nat))
    Nat ≈ Maybe(Product(Unit, Nat)),
    List(Unit) ≈ Maybe(Product(Unit, List(Unit)))
       ⇒
    Nat ≈ List(Unit)

BinTreeLeaf(a, b) ≈
      Either(a, Product_3(b, BinTreeLeaf(a, b), BinTreeLeaf(a, b)))
```

En la segunda definición de `Nat` se puede observar que se define a partir de un tipo de datos que no es de construcción, y que ya fue definido previamente, como el tipo `List(a)`. También se puede observar el uso del tipo `Unit` para eliminar el tipo variable paramétrico del tipo lista utilizado.
El tipo de dato `BinTreeLeaf(a, b)` corresponde al tipo de los árboles binarios con nodos binarios etiquetados con una expresión de tipo `b`, y donde los casos base representan hojas etiquetadas con nodos de tipo `a`.

Si un tipo de datos `T` se define a través de la relación de semejanza estructural $T \approx Product_n(T_1, \ldots, T_n)$, decimos que es una definición basada en atributos, donde cada uno de los tipos de datos T_1 a T_n corresponden a los tipos de cada uno de "los atributos del tipo definido `T`" (el sentido semántico y los nombres de cada uno de los atributos no quedará expresado en la definición del tipo `T`, más allá de las validaciones de los valores que poseerán los atributos luego de aplicar cada operación sobre un elemento de este tipo `T`).

Cuando no existe semejanza en la definición de un tipo

En muchos casos la semejanza estructural se puede utilizar para definir tipos nuevos por equivalencia con tipos anteriores o por equivalencia respecto a combinaciones de tipos que usan tipos de construcción. Pero en muchos casos, para definir un nuevo tipo no es necesario recurrir a la semejanza estructural con otro tipo, sino a una condición más débil, que consista en que ese otro tipo tiene que proveer medios suficientes (no los estrictamente necesarios) para la definición del tipo nuevo. Esta condición, llamada *utilización o representación de tipos* es la básica para la definición de tipos nuevos (a partir de otros existentes). El mapeo entre el tipo a definir y el tipo a utilizar para su definición dejaría de ser estrictamente biyectivo y apenas sería inyectivo (no deberá ocurrir que dos elementos distintos del mismo tipo a definir se representen con los mismos elementos del tipo utilizado). No habría que preocuparse por aquellos elementos del tipo a utilizar que no representan elementos válidos (mapeables en la representación) del tipo a definir.

Un ejemplo algo extremo sería el siguiente:

`Persona` *representado como* $\texttt{Product}_4\texttt{(Nat, String, String, Nat)}$

Si suponemos que una persona se define a partir de sus atributos correspondientes al DNI (valor de tipo `Nat`), su nombre (valor de tipo `String`), su apellido (valor también de tipo `String`) y su edad (valor de tipo `Nat`), con el tipo utilizado para su representación es suficiente, ya que el tipo a utilizar soporta la representación que se necesita, aunque sabemos que no es cierto que cualquier elemento del tipo utilizado va a corresponder a una persona válida (hay números naturales que no corresponden a ningún DNI, constantes string que no tienen forma de nombres o apellidos, y valores numéricos naturales de los que una persona jamás podrá tener como edad).

Para comprender en mayor medida este concepto de semejanza estructural (mapeo biyectivo) o utilización de representación (mapeo sólo inyectivo) entre el tipo a definir y el tipo en el que se apoya para su representación, no sólo habría que analizar el mapeo entre los dos tipos, sino también habría que hacer una comprensión semántica (de significado) de los elementos propios del tipo a utilizar en la representación en cuestión, lo que da una idea de conocimiento del dominio del modelo a representar.

Correspondencia entre los tipos de construcción y los de implementación

Existen algunos criterios para transformar tipos definidos dentro del nivel de especificación a partir de tipos de construcción, a tipos para ser representados en la implementación. Este criterio se basa en el mapeo de algunos tipos de construcción a su equivalente en implementación. Este mapeo no contempla modificaciones que podría tener la representación a nivel implementación

por cuestiones de optimización (reducción de espacio de representación de la estructura, reducción de tiempo de ejecución de operaciones sobre expresiones de esa estructura).

Un criterio de conversión simple puede ser el siguiente:

Tipo de dato de construcción (especificación)	**Tipo de dato (implementación)**
`Unit`	*Tipo con un solo elemento*

`Product`$_n$`(`T_1`, `T_2`, …, `T_n`)`	`record`
	`  cpo`$_1$`: `T_1`,`
	`  cpo`$_2$`: `T_2`,`
	`  …`
	`  cpo`$_n$`: `T_n
	`end`

`Either`$_n$`(`T_1`, `T_2`, …, `T_n`)`	`record`
	`  case (numcons: integer) of`
	`  when 1 => cpo`$_1$`: `T_1`,`
	`  when 2 => cpo`$_2$`: `T_2`,`
	`  …`
	`  when n => cpo`$_n$`: `T_n
	`  end`
	`end`

`Maybe(T)`	`^T`

El tipo de construcción `Unit` a nivel especificación puede estar definido a nivel implementación como un tipo de un solo elemento. Esto se puede realizar con el objetivo de reusar código al definir el tipo que se necesita a partir de un tipo ya existente en la implementación. De igual modo, y si se conociera la implementación de un tipo candidato a ser utilizado, se podría volver a codificar una variante de su implementación para evitar el uso del tipo `Unit` de implementación sobre ese tipo de datos ya implementado. Esta solución que podría evitar el uso del tipo `Unit` de implementación puede ser criticada por su bajo nivel de reusabilidad. Se puede también intentar eliminar su uso particular utilizando el equivalente a nivel implementación de correspondencia con tipos estructuralmente semejantes que no lo utilicen.

El tipo de construcción `Product` se puede ver fácilmente mapeado al tipo `record` de implementación (registro).

El tipo de construcción `Either` se puede ver fácilmente mapeado a una variante del tipo `record` de implementación llamado *variant record* (registro variante). Los variant records consisten en registros de implementación donde algunos campos pueden ser condicionales entre sí. Todos los campos que son condicionales entre sí deberán definirse dentro de un subbloque de la estructura de datos del registro, que es una subestructura `case`. Esta subestructura `case` tiene un campo "discriminante" que es el que se indica en el encabezado de esta subestructura con su tipo, tal que dependiendo de su valor, establece la existencia de sólo el campo condicional que coincida con su valor constante indicado en sólo una de las líneas `when`. Todos los campos condicionales que se encuentran dentro del subbloque `case` de un variant record se encuentran solapados en su representación en memoria.

Los subbloques `case` pueden aparecer en más de una oportunidad, directamente o anidados dentro de una misma estructura de registro, lo que da a entender que se pueden combinar en una misma estructura de datos de tipo registro, a campos de registros comunes y campos variantes.
Esta estructura de variant record provee las mismas fallas al sistema de tipos que provee el tipo de construcción `Either` en la especificación. Generalmente los lenguajes que poseen este tipo de estructuras poseen tipificación dinámica (el tipo de datos de las expresiones se lleva en tiempo de ejecución, para resolver las fallas en el sistema de tipos provocadas por estructuras como ésta).

El tipo de construcción `Maybe`, al representar un tipo dado `T` más el agregado de una constante que representa el "no valor", se puede mapear fácilmente al tipo `^T` (puntero a una expresión de tipo `T`).
El tipo puntero también provee fallas en el sistema de tipos, ya que puede provocar errores en tiempo de ejecución producto de direccionamientos inválidos cuando el valor del puntero es igual a `null`.

Existen otras variantes en la definición del tipo de construcción `Either` sin utilizar registros variantes. Una representación alternativa puede ser la siguiente:

```
record
   numcons: integer,
   cpo1: ^T1,
   cpo2: ^T2,
    …
   cpon: ^Tn
end
```

En esta representación se puede observar que no se usa un registro variante, sino un registro común. Observar que el "campo discriminante" se agrega como un campo común en el registro, y los otros campos "condicionales" pasaron a ser no condicionales y de tipo ^T_i en lugar del tipo original T_i. De esta forma, y dependiendo del valor que tenga el campo discriminante, sólo uno de los campos de datos tendrá un valor, y el resto de los campos, no.
En el ejemplo, si `numcons = i`, entonces sólo el campo cpo_i tendrá un valor de tipo ^T_i distinto al valor `null` ($cpo_i^$ tendrá un valor válido de tipo T_i), mientras que el resto de los campos cpo_j (para $j \neq i$) tendrá valor `null`.

Sobre la utilización del tipo puntero en la implementación

Es sabido que el tipo puntero es un tipo estructurado que puede contener una referencia a una expresión de un tipo determinado, o bien no tener ninguna referencia a una expresión de ese tipo, que significa que posee el valor `null`. Sin embargo, lo particular que tiene este tipo respecto a los otros tipos estructurados es que en su representación no "engloba" dentro de su propia estructura de representación en memoria al o los componentes a los que hace referencia, en el caso del puntero al valor referenciado. No ocurre lo mismo con el resto de los tipos estructurados (arreglos y registros), ya que los arreglos contienen dentro de su representación física a cada uno de los contenidos de sus subíndices, y los registros contienen dentro de su representación física a

cada uno de los contenidos de sus campos, y todos conviven en el mismo entorno en tiempo de ejecución.

El tamaño de representación de los tipos estructurados "englobantes" en la implementación siempre se calcula en función de los tamaños de los tipos de sus componentes a los que hace referencia. Los tamaños de los tipos estructurados son los siguientes:

Tamaño(`array[`*n*`] of T`)	=	*n* * Tamaño(`T`)
Tamaño(`record cpo`$_1$`: T`$_1$`, …, cpo`$_n$`: T`$_n$ `end`)		Tamaño(T_1) + … + Tamaño(T_n)

Tamaño(
`record`
`case (cpo`$_{dte}$`: T`$_{dte}$`) of`
`when` *cte*$_{1dte}$ `=> cpo`$_1$`: T`$_1$`,`
…
`when` *cte*$_{ndte}$ `=> cpo`$_n$`: T`$_n$
`end`
`end`
)
		Tamaño(T_{dte}) +
		+ máx(Tamaño(T_1), …, Tamaño(T_n))

Tamaño(`^T`)		4

Como el tipo puntero es el único tipo estructurado que no es "englobante" (el tamaño del tipo `^T` es siempre 4 -el tamaño de los punteros-, y es independiente del tamaño de `T`), dentro de su estructura de representación no se ubica el valor apuntado. Este valor apuntado puede estar dentro del mismo entorno en el que se encuentra el puntero, o bien en otro entorno. Este último caso puede llegar a ser peligroso, ya que el entorno donde se encuentra el valor apuntado puede ser destruído, y el puntero seguir vivo, donde a partir de ese momento apuntará a una memoria "muerta". Esto se da en los casos en que el puntero se pase por referencia de un procedimiento/función A a otro B (en el momento en que A invoca a B), y el valor apuntado aparece dentro del entorno local de B. Cuando B finalice su ejecución, todo su entorno se destruirá (incluyendo el valor apuntado por el puntero), pero el puntero seguirá vivo, ya que existe en el entorno de A, pero su valor apuntado, no.

Un caso de lo mencionado anteriormente se puede ver en el siguiente ejemplo de código:

```
(1)
procedure asignarPuntero(ref p: ^a; dato: a)
//lo que apunta el puntero retornado p es memoria semiestática (dentro
//del entorno de asignarPuntero)
begin
  p := address(dato);
  //Otra alternativa: (usando d variable declarada de tipo a)
  //  d := dato;
  //  p := address(d);
end;
```

```
(2)
procedure asignarPuntero(ref p: ^a; dato: a)
begin
  new(p);   //Creación de memoria dinámica con el puntero p apuntando a ella
  p^ := dato;
end;

procedure main()
var
  p: ^Integer;
begin
  asignarPuntero(p, 35);
  display p^;
  //Habría que desasignar explícitamente la memoria dinámica si se
  //utilizara el procedimiento (2) con la siguiente sentencia:
  //  dispose(p);
end;
```

En el procedimiento `main`, el valor de `p^` realmente es `35` usando el procedimiento `asignarPuntero` de la definición `(2)` que utiliza memoria dinámica. Sin embargo, si se utilizara el procedimiento `asignarPuntero` de la definición `(1)` que utiliza memoria semiestática, si bien el valor de `p` es retornado, el valor de `p^` representa "memoria muerta", ya que `p` apuntaba a la variable `dato` (o `d`) del entorno del procedimiento `asignarPuntero`, que fue destruído con la finalización de la ejecución del mismo. Al ser pasado como parámetro por referencia, la variable `p` no existe realmente en el procedimiento llamado (`asignarPuntero`), sino en el llamador (en este caso, `main`).

El problema que se planteó aquí es que el valor apuntado correspondía a una estructura estática, existente dentro del entorno de un procedimiento/función. Si el puntero apuntara a una estructura dinámica, este problema no ocurriría, ya que las variables dinámicas no se destruyen con el fin de ejecución de procedimientos y funciones, sino explícitamente por el programador, por pérdida de referencias o por alguna política de limpieza automática de variables dinámicas no referenciadas provistas por la implementación del lenguaje. Como ninguno de estos criterios es perjudicial, y la variable dinámica sobrevive al entorno del procedimiento/función finalizado (todas las variables dinámicas se almacenan en un entorno aparte, dentro de la zona de memoria dinámica o *heap*), se puede afirmar que el mejor criterio para utilizar variables de tipo puntero es que el valor apuntado corresponda a una variable dinámica. Sin embargo, y como es sabido, la justificación para el uso de variables dinámicas es su utilización en estructuras recursivas (tipos definidos en los que el tipo utilizado como representación del tipo a definir contiene como subexpresión de tipos al mismo tipo a definir).

El tipo puntero es la única salida para la representación de tipos recursivos, por la naturaleza "no englobante" del tipo puntero. Si fuera englobante y se tratara con tipos recursivos, dentro de su estructura debería contener estrictamente a un tipo del mismo tamaño o mayor, lo que lo haría imposible de representar. Y por lo dicho anteriormente, los usos de punteros para apuntar a estructuras estáticas pueden traer problemas de pérdidas no deseadas de referencias.

Algunos ejemplos de tipos de datos especificados con tipos de construcción

Un tipo del que se puede volver a definir su especificación utilizando tipos de datos de construcción, es el tipo `Bool`. El tipo de dato que sirve como base en la definición es el que aparece en negrita.
La definición sería la siguiente (en base a la definición utilizando constructores y manteniendo su interfaz, aunque esta vez definido como TAD en lugar de tipo para ocultar estos constructores):

```
TAD Bool
exporta true, false, not, _and_, _=_, if_then_else_, _or_, _implies_
constructores
   FromEither : Either(Unit, Unit) → Bool  //único constructor (función)
proyectores
   true : Bool
   false : Bool
   not : Bool → Bool
   _ and _ : Bool × Bool → Bool
   _ = _ : Bool × Bool → Bool
   if _ then _ else _ : Bool × a × a → a
operaciones
   _ or _ : Bool × Bool → Bool
   _ implies _ : Bool × Bool → Bool
axiomas
   true = FromEither(Left(Unit))
   false = FromEither(Right(Unit))
   not(FromEither(Left(Unit))) = FromEither(Right(Unit))
   not(FromEither(Right(Unit))) = FromEither(Left(Unit))
   FromEither(Left(Unit)) and x = x
   FromEither(Right(Unit)) and x = FromEither(Right(Unit))
   FromEither(Left(Unit)) = x = x
   FromEither(Right(Unit)) = x = not(x)
   if FromEither(Left(Unit)) then x else y = x
   if FromEither(Right(Unit)) then x else y = y
   x or y = not(not(x) and not(y))
   x implies y = not(x) or y
```

Notar que las operaciones que se definen a partir del uso de funciones proyectoras (y no utilizando matching sobre constructores) no modifican su definición.

La definición del tipo `List(a)` sería la siguiente (en base a la definición utilizando constructores y manteniendo su interfaz, y también como TAD):

```
TAD List(a)
exporta emptyList, consList, isEmptyList, head, tail, belongs, _=_,
        concat, reverse
usa Bool
constructores
   FromMaybeList : Maybe(Product(a, List(a))) → List(a)
proyectores
   emptyList : List(a)
   consList : a × List(a) → List(a)
   isEmptyList : List(a) → Bool
   head : List(a) → a
   tail : List(a) → List(a)
operaciones
   length : List(a) → Nat
   belongs : a con operaciones _=_ × List(a) → Bool
   _ = _ : List(a con operaciones _=_) × List(a) → Bool
   concat : List(a) × List(a) → List(a)
   reverse : List(a) → List(a)
   add : List(a) × a → List(a)
axiomas
   emptyList = FromMaybeList(Nothing)
   consList(x, xs) = FromMaybeList(Just(Pair(x, xs)))
   isEmptyList(FromMaybeList(Nothing)) = true
   isEmptyList(FromMaybeList(Just(Pair(x, xs)))) = false
   head(FromMaybeList(Just(Pair(x, xs)))) = x
   tail(FromMaybeList(Just(Pair(x, xs)))) = xs
```

Definiciones axiomáticas alternativas de los últimos cuatro proyectores podrían ser las siguentes:

```
isEmptyList(FromMaybeList(m)) = isNothing(m)
head(FromMaybeList(Just(p))) = getFirst(p)
tail(FromMaybeList(Just(p))) = getSecond(p)
```

También, los dos últimos proyectores podrían definirse de la siguiente manera:

```
head(FromMaybeList(m)) = getFirst(getValue(m))
tail(FromMaybeList(m)) = getSecond(getValue(m))
```

O bien, se podrían definir así:

```
head(l) = getFirst(getValue(getMaybeList(l)))
tail(l) = getSecond(getValue(getMaybeList(l)))
```

Esto último se podría aplicar a todas las operaciones que realizan pattern matching sobre la función constructora `FromMaybeList`, y siempre y cuando se defina el proyector no exportado `getMaybeList` definido axiomáticamente de la siguiente manera:

```
proyectores
   getMaybeList : List(a) → Maybe(Product(a, List(a)))
axiomas
   getMaybeList(FromMaybeList(m)) = m
```

Las definiciones axiomáticas del resto de las operaciones, al estar definidas a partir del uso de proyectores, no cambia respecto a la definición anterior.

La implementación del tipo `List(a)` a partir del tipo de dato que sirve como base -el indicado en negrita- sería el siguiente:

```
type
   List(a) = ^record
                node: a,
                sig: List(a)
             end
```

Notar que esta implementación del tipo `List(a)` es la generada utilizando los criterios de conversión del tipo de dato de construcción con que se especifica el tipo que sirve de base, al tipo de implementación. Curiosamente, el tipo de datos final de implementación es exactamente el mismo que el de la representación dinámica de una lista simplemente encadenada.

Factorización y distribución de tipos

Falta discutir un caso realmente especial de semejanza estructural entre dos tipos de construcción a nivel especificación. El caso es el siguiente:

```
Product(a, Either(b, c)) ≈ Either(Product(a, b), Product(a, c))
```

La semejanza estructural "hacia la izquierda" se llama *distribución del tipo* `a`, mientras que la semejanza estructural "hacia la derecha" se llama *factorización del tipo* `a`. Si bien es cierto que ambos tipos de datos poseen la misma cantidad de elementos (y en consecuencia existe esa biyección necesaria para afirmar que son estructuralmente semejantes), la semejanza estructural de ambos tipos está en duda.

El problema no está en la *distribución del tipo* `a` (semejanza estructural "hacia la izquierda") que siempre sería posible, sino en la *factorización del tipo* `a` (semejanza estructural "hacia la derecha"). La factorización del tipo `a` debería ser posible si los componentes de tipo `a` de ambos productos tuvieran el mismo significado semántico (que signifiquen lo mismo). De ser así, se podría factorizar, separando esta característica común o *general* a ambos productos, pero a su vez compartiéndola como una característica general separada. Si este tipo `a` tuviera un significado diferente en ambos productos, no podrá ser separado de ellos como una característica común, y deberá mantenerse en los productos originales. Esta es la primera aparición de conceptos semánticos o de significado que inciden en una definición de semejanza estructural.

Lo que queda planteado aquí es que los tipos factorizados corresponden a características comunes a varios tipos existentes, cada uno de ellos representado por los tipos-parámetro del tipo `Either` que conforma el otro componente del tipo `Product`. Esta existencia de características comunes a varios tipos y características específicas de cada uno de esos tipos sugiere una especie de jerarquía de la tipificación (jerarquía supertipo/subtipo) respecto a la generalidad/particularidad de sus características (los tipos con las características generales corresponden a los supertipos, y los que contienen las particulares corresponden a los subtipos). Esta jerarquía también puede continuar en los niveles inferiores, donde los subtipos pueden ser supertipos respecto a otros subsubtipos, y así sucesivamente.

La jerarquía de supertipo/subtipo se puede resumir con las siguientes definiciones:

- B **es subtipo de** A $\Leftrightarrow$ ($x \in B \Rightarrow x \in A$)
- $B_1, B_2, ..., B_n$ son todos los subtipos de A $\Rightarrow$ $\{B_1, B_2, ..., B_n\}$ **es una partición de** A

La segunda definición significa que todo elemento del supertipo A debe pertenecer estrictamente a sólo uno de sus subtipos B_i. Como consecuencia, toda característica (y operación que se basa en estas características) de un tipo A es *heredada* por todos sus subtipos B_i. Y a su vez, todas las características y operaciones de los tipos B_i (propias o heredadas) son heredadas por los subtipos de cada uno de ellos.

Con esta nueva concepción, aparecen tipos *polimórficos*, ya que todo elemento del tipo B (del subtipo) también es elemento del tipo A (del supertipo). Estos tipos polimórficos son seguros respecto a la tipificación (no como el polimorfismo de *untagged union*), ya que los supertipos sólo contienen características propias de ellos, y todas sus operaciones se definen en función sólo de sus características. Lo que no se permite desde un tipo es acceder a características ni operaciones exportadas propias de alguno de sus subtipos, lo que le ofrece seguridad en la tipificación. Lo que sí es permitido es acceder a todas las operaciones exportadas por su posible supertipo (aquí es donde se refleja el concepto de herencia), sean estas operaciones propias del supertipo o heredadas por él. Este uso de la herencia no provoca problemas en la tipificación.

Dicho de otra manera, cuando hasta ahora existía la siguiente definición de tipo:

Tipo A = $\texttt{Product(a, Either}_n\texttt{(}b_1\texttt{, }b_2\texttt{, …, }b_n\texttt{))}$

ahora, con la jerarquía supertipo/subtipo se reemplaza por las siguientes definiciones:

Tipo A = `a`
Tipo B_1 (subtipo de A) = b_1
Tipo B_2 (subtipo de A) = b_2
...
Tipo B_n (subtipo de A) = b_n

Notar que con el uso de jerarquías supertipo/subtipo, se deja de usar el tipo `Either`, que se reemplaza por la misma jerarquía de subtipificación con la obligatoriedad de que los subtipos conformen una partición del supertipo.

Si no existiera la jerarquía supertipo/subtipo, los tipos reales o "completos" de los mencionados arriba serían los siguientes:

Tipo A = `Product(a, Either`$_n$`(`b_1`, `b_2`, …, `b_n`))`
Tipo B_1 = `Product(a, `b_1`)`
Tipo B_2 = `Product(a, `b_2`)`
...
Tipo B_n = `Product(a, `b_n`)`

Notar que cuando no existe la jerarquía de tipos, las características del que sería el supertipo son distribuídas a cada uno de sus subtipos. Esto hace que todas las especificaciones comunes estén replicadas en cada uno de los tipos que corresponderían a los subtipos, lo se haría difícil de mantener y extender (esto último en el caso de que se defina un nuevo subtipo).

Estos conceptos aquí descriptos corresponden a la base de la *orientación a objetos*, esta vez no orientada a la programación, sino a la especificación axiomático-funcional. Las características centrales de la programación orientada a objetos no serán tratadas en el presente apunte.

Bibliografía

- *Sommerville, Ian*. **Software engineering**. 2007. Addison Wesley. [capítulo 10]

- *O'Sullivan, Bryan / Stewart, Don / Goerzen, John*. **Real World Haskell**. 2008. O'Reilly.