Lenguajes de Programación, 2019-I
Nota de clase 8: Tipos de Datos Estructurados*

Favio E. Miranda Perea Lourdes del Carmen González Huesca
Facultad de Ciencias UNAM

1 de octubre de 2018

Resulta conveniente agregar a nuestro lenguaje nucleo funcional MinHs algunos tipos adicionales como tuplas, listas ó árboles. Estas estructuras o tipos de datos se pueden clasificar en tres categorías.

- Tipos producto: son aquellos cuyos valores son tuplas de valores donde cada componente es de un tipo específico. El caso base es el de tuplas con 2 elementos o pares ordenados.
- Tipos suma: también conocidos como uniones (disjuntas), son aquellos cuyos valores son valores de alguno de n tipos específicos con una etiqueta explícita indicando de cual de estos n tipos se trata. El caso base es el de una unión disjunta de dos tipos.
- Tipos recursivos: estos son tipos auto-referenciales en el sentido de que sus valores pueden definirse mediante otros valores del mismo tipo recursivo. Por ejemplo las listas y los árboles.

En nuestro curso nos dedicaremos a las sumas y los productos que proporcionan estructuras fundamentales básicas para definir tipos registro (records) y tipos variante (opciones). Ambas nociones guardan una dualidad encontrada en diversas áreas. Por ejemplo:

- Lógica: conjunción y disyunción.
- Lenguajes formales: concatenación y unión.
- Programación: secuencias y alternativas en instrucciones.
- Definiciones inductivas: varias premisas y varias reglas.

Iniciamos con los tipos producto.

1. Productos

Con frecuencia un lenguaje de programación ofrece una manera de fusionar elementos de dos o más tipos mediante el uso de un nuevo tipos. Por ejemplo dos valores flotantes representan un punto en el espacio, etc.

En Haskell por ejemplo tenemos:

*Estas notas se basan en los libros de Harper y Pierce y en material de Frank Pfenning, Ralf Hinze y Gerwin Klein.

- Tuplas:
  - Declaración: `type Punto = (Float,Float)`
  - Uso: `let x = (1.75,1.21) in fst x`
- Tuplas con campos nombrados:
  - Declaración: `data Punto = Punto x::Float, y::Float`
  - Uso: `let p=Point 5.0 10.1 in...`, `let p = Point x=5.0 y=10.1 in...` se generan funciones selectoras `x,y:: Punto ->Float` de manera automática, por ejemplo `xp` devuelve el valor de `x` en el punto `p`.

Similarmente en C podemos definir:

```
struct point {
   float x;
   float y;
};

struct point p;
 p.x = 10.1;
```

Mientras que en Java podemos definir clases degeneradas que no contienen métodos:

```
class Point {
 public float x;
 public float y;
}
```

O, como es más usual, proteger los campos de datos y dando métodos para generar, accesar y alterar estructuras:

```
class Point {
 private float x;
 private float y;

 public Point (float x, float y) {
   this.x = x;
   this.y = y;
 }

 public float getX () {return x;}
 public float getY () {return y;}

 public void setX (float x) {this.x = x;}
 public void setY (float y) {this.y = x;}
}
```

1.1. Tipos producto y sus constructores

Los tipos producto son nuestro primer ejemplo de un tipo estructurado. Un tipo producto $\mathsf{T} \times \mathsf{S}$ representa al tipo de parejas de valores cuyo primer componente es de tipo T y el segundo de tipo S. Como ya es costumbre presentamos aquí únicamente los nuevos constructores del lenguaje.
Los dos niveles de sintaxis se resumen como sigue:

Sintaxis concreta	Sintaxis abstracta
Void, 1, ()	`Void`
void,()	`void`
$\mathsf{T} \times \mathsf{S}$	$\texttt{prod}(\mathsf{T}, \mathsf{S})$
(e_1, e_2)	$\texttt{pair}(t_1, t_2)$
$\mathsf{fst}\, e$	$\texttt{fst}(t)$
$\mathsf{snd}\, e$	$\texttt{snd}(t)$

Distinguimos tres categorias:

- Tipos: `Void`, 1 o () para tuplas sin elementos y $\mathsf{T} \times \mathsf{S}$ para pares ordenados.
- Constructores: () o `void` es la única tupla sin elementos y (e_1, e_2) es un par ordenado.
- Selectores o destructores: las proyecciones $\mathsf{fst}\, e, \mathsf{snd}\, e$ devuelven el primer y segundo elemento del par ordenado e respectivamente.

Es importante aclarar que el nombre void es usual en lenguajes de programación aunque void significa vacío y este tipo no es vacío sino que tiene un único elemento.

La semántica estática está dada por las siguientes reglas:

$$\frac{}{\Gamma \vdash () : \texttt{Void}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T} \quad \Gamma \vdash e_2 : \mathsf{S}}{\Gamma \vdash \texttt{pair}(e_1, e_2) : \texttt{prod}(\mathsf{T}, \mathsf{S})}$$

$$\frac{\Gamma \vdash e : \texttt{prod}(\mathsf{T}, \mathsf{S})}{\Gamma \vdash \texttt{fst}(e) : \mathsf{T}} \qquad \frac{\Gamma \vdash e : \texttt{prod}(\mathsf{T}, \mathsf{S})}{\Gamma \vdash \texttt{snd}(e) : \mathsf{S}}$$

Como ya es usual usaremos con frecuencia la sintaxis concreta, por ejemplo usando la regla para tipar pares como sigue:

$$\frac{\Gamma \vdash e_1 : \mathsf{T} \quad \Gamma \vdash e_2 : \mathsf{S}}{\Gamma \vdash (e_1, e_2) : \mathsf{T} \times \mathsf{S}}$$

Para la estrategia de evaluación de la semántica dinámica definimos los siguientes nuevos valores:

$$v ::= \ldots \mid () \mid (v_1, v_2)$$

Es decir, la tupla vacía () es un valor y cualquier par ordenado de valores es un valor.

La semántica dinámica se especifica mediante las siguientes reglas, nuevamente en sintaxis concreta, obsérvese que estas reglas modelan la estrategia de evaluación ansiosa:

$$\frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)} \qquad \frac{e_2 \to e_2'}{(v, e_2) \to (v, e_2')}$$

$$\frac{e \to e'}{\mathsf{fst}\, e \to \mathsf{fst}\, e'} \qquad \frac{e \to e'}{\mathsf{snd}\, e \to \mathsf{snd}\, e'}$$

$$\frac{}{\mathsf{fst}(v_1, v_2) \to v_1} \qquad \frac{}{\mathsf{snd}(v_1, v_2) \to v_2}$$

El sistema es seguro, es decir se cumple lo siguiente:

Proposición 1 (Seguridad) *Se cumplen las siguientes propiedades:*

- *Si* $\Gamma \vdash e : \mathsf{T}$ *y* $e \to e'$ *entonces* $\Gamma \vdash e' : \mathsf{T}$.
- *Si* $\vdash e : \mathsf{T}$ *entonces* e *es un valor o existe* e' *tal que* $e \to e'$.

1.2. Las operaciones *Curry* y *Uncurry*

La relación entre las funciones de orden superior y las funciones de primer orden con múltiples argumentos es una pregunta de interés práctico en el cálculo lambda. Por lo general estamos acostumbrados a usar y definir funciones de múltiples argumentos, sin embargo en el cálculo lambda con tipos simples todas las funciones son de un argumento. Si tenemos una función de dos argumentos, por ejemplo la suma $suma(x : \mathsf{Nat}, y : \mathsf{Nat}) = x + y$ generalmente la pensamos como una función cuyo dominio son los pares ordenados de $\mathsf{Nat} \times \mathsf{Nat}$, es decir el tipo en que estamos pensando es $suma : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$. Al tener disponibles ahora los tipos producto podemos escribir una versión de esta función como sigue:

$$suma = \lambda p : \mathsf{Nat} \times \mathsf{Nat} \,.\, \mathsf{fst}\, p + \mathsf{snd}\, p$$

En el cálculo lambda sin productos esta función no puede escribirse, pero sí la siguiente función relacionada a la anterior.

$$sumaCurry = \lambda x : \mathsf{Nat}\, \lambda y : \mathsf{Nat} \,.\, x + y$$

la cual devuelve también la suma pero recibe a sus argumentos uno por uno, por ejemplo la función $sumaDos\ x = 2 + x$ puede definirse como $sumaDos\ x = suma\langle 2, x\rangle$ o bien como $sumaDos = sumaCurry\ 2$. Obsérvese que la segunda definición no requiere de un argumento x para definirse, lo cual es obligatorio en la primera. Funciones como $sumaCurry$ se llaman funciones *Curryficadas* debido a que fue el lógico Haskell Curry quién las popularizo aunque realmente fue Schönfinkel quien descubrio su definición y relación con las funciones de múltiples argumentos. De hecho, en el lenguaje Haskell existe la función de orden superior `curry` que transforma una función de dos argumentos en una función que toma a sus argumentos de uno en uno, definida como `curry` $f\ x\ y = f\ \langle x, y\rangle$. Análogamente existe la función inversa `uncurry` que transforma una función de orden superior de un argumento en una función que toma a sus dos argumentos a la vez, definida como `uncurry` $f\ \langle x, y\rangle = f\ x\ y$.

1.3. El tipo unitario

El tipo unitario () o `Void` aparentemente no es de gran utilidad puesto que sólo tiene un valor, a saber (). Sin embargo su utilidad surge cuando se necesita devolver un resultado irrelevante como veremos más adelante. Dada su simplicidad no hay ninguna operación interesante sobre este tipo por lo cual no figura en la semántica dinámica.

1.4. Tuplas

Podemos generalizar el constructor producto a un número arbitrario de tipos generando tuplas. La sintaxis es:

Sintaxis concreta	Sintaxis abstracta
$\mathsf{T}_1 \times \mathsf{T}_2 \times \ldots \times \mathsf{T}_k$	$\texttt{tpl}(\mathsf{T}_1, \ldots \mathsf{T}_k)$
$(e_1, e_2, \ldots, e_k)$	$\texttt{tpl}(t_1, t_2, \ldots, t_k)$
$t.i$	$\texttt{prj}[i](t)$

Obsérvese que las funciones selectoras se indican por posición usando una notación de punto $t.i$ que indica a la i-ésima posición de la tupla t. Esta notación no debe confundirse con la notación punto para ligado utilizada en la sintaxis abstracta.

No daremos más detalles de esta extensión, en su lugar tratamos una generalización más útil, los tipos registro.

1.5. Tipos Registro

Los tipos producto pueden usarse para definir tipos de tuplas con nombre llamados tipos registro o *records* cuyas componentes se accesan mediante nombres simbólicos en lugar de por posición. De esta forma un *record* modela a una colección de tuplas de diversos tipos con una operación al nivel de valores para extraer los elementos de esta familia por su nombre.

La sintaxis para los tipos registro es:

Concreta	Abstracta
$(\ell_1 : \mathsf{T}_1, \ell_2 : \mathsf{T}_2, \ldots, \ell_k : \mathsf{T}_k)$	$\texttt{rcd}(\ell_1 : \mathsf{T}_1, \ell_2 : \mathsf{T}_2 \ldots, \ell_k : \mathsf{T}_k)$
$\{\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_k = e_k\}$	$\texttt{rcd}(\ell_1 = t_1, \ell_2 = t_2, \ldots, \ell_k = t_k)$
$t.\ell$	$\texttt{prj}[\ell](t)$

Aquí se introduce una nueva categoría de etiquetas ℓ los cuales son identificadores pero no variables. En un registro $\{\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_k = e_k\}$ todas las etiquetas deben ser distintas para evitar confusiones entre los campos del registro.

La semántica estática se rige por las siguientes reglas:

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \ \ldots \ \Gamma \vdash e_k : \mathsf{T}_k}{\Gamma \vdash \texttt{rcd}(\ell_1 = e_1, \ldots, \ell_k = e_k) : \texttt{rcd}(\ell_1 : \mathsf{T}_1, \ldots, \ell_k : \mathsf{T}_k)}$$

$$\frac{\Gamma \vdash e : \mathtt{rcd}(\ell_1 : \mathsf{T}_1, \ldots, \ell_j : \mathsf{T}_j, \ldots, \ell_k : \mathsf{T}_k))}{\Gamma \vdash \mathtt{prj}[\ell_j](e) : \mathsf{T}_j}$$

A la categoría de valores agregamos un registro de valores:

$$v ::= \ldots \mid \mathtt{rcd}(\ell_1 = v_1, \ldots, \ell_k = v_k)$$

Finalmente la semántica dinámica para la estrategia de evaluación ansiosa es:

$$\frac{e_j \to e'_j}{\mathtt{rcd}((\ell_i = v_i)_{i=1}^{j-1}, \ell_j = e_j, (\ell_k = e_k)_{k=j+1}^{n}) \to \mathtt{rcd}((\ell_i = v_i)_{i=1}^{j-1}, \ell_j = e'_j, (\ell_k = e_k)_{k=j+1}^{n})}$$

La notación $(\ell_i = v_i)_{i=1}^{j-1}$ abrevia a $(\ell_1 = v_1, \ldots, \ell_{j-1} = v_{j-1})$.

$$\frac{e \to e'}{\mathtt{prj}[\ell](e) \to \mathtt{prj}[\ell](e')}$$

$$\frac{}{\mathtt{prj}[\ell_j](\mathtt{rcd}(\ell_1 = v_1, \ldots, \ell_j = v_j, \ldots, \ell_k = v_k)) \to v_j}$$

2. Sumas

Los tipos suma proporcionan una manera para fusionar en un mismo tipo elementos de tipos distintos. Un elemento del tipo suma $\mathsf{T} + \mathsf{S}$ puede pensarse como un elemento de T o bien como un elemento de S los cuales se encuentran etiquetados para denotar su origen.

Sintaxis concreta	Sintaxis abstracta
Empty, $\bot$, 0	$\mathtt{empty}$
$\mathsf{abort}_{\mathsf{T}}(e)$	$\mathtt{abort}[\mathsf{T}](e)$
$\mathsf{T} + \mathsf{S}$	$\mathtt{suma}(\mathsf{T}, \mathsf{S})$
$\mathsf{inl}_{\mathsf{T}}\, e$	$\mathtt{inl}[\mathsf{T}](e)$
$\mathsf{inr}_{\mathsf{T}}\, e$	$\mathtt{inr}[\mathsf{T}](e)$
$\mathsf{case}\ e\ \mathsf{of}\ \{\mathsf{inl}\ x\ \Rightarrow\ e_1 \mid \mathsf{inr}\ y\ \Rightarrow\ e_2\}$	$\mathtt{case}[\mathsf{T}, \mathsf{S}](e, x.e_1, y.e_2)$

Nuevamente distinguimos tres categorías:

- Tipos: `Empty` es el tipo vacío, que representa una elección entre cero alternativas. Por supuesto no hay valores que habiten este tipo. El tipo $\mathsf{T} + \mathsf{S}$ es la suma o unión disjunta de dos tipos.

- Constructores: no existen constructores para `Empty`. Los constructores para una suma binaria se conocen como inyecciones. Se denotan con inl, inr e indican a cual de las dos posibles elecciones para el tipo suma nos referimos, al tipo de la izquierda o al de la derecha.
- Selectores o destructores: el destructor $\mathsf{abort}_{\mathsf{T}} e$ aborta el cómputo de un programa que intentó evaluar una expresión de tipo `empty` lo cual es imposible. El selector `case` hace un análisis de casos de acuerdo a la etiqueta de un valor de tipo suma.

La semántica estática es:

$$\frac{\Gamma \vdash e : \mathsf{Empty}}{\Gamma \vdash \mathtt{abort}[\mathsf{T}](e) : \mathsf{T}}$$

$$\frac{\Gamma \vdash e : \mathsf{T}}{\Gamma \vdash \mathsf{inl}[\mathsf{S}](e) : \mathsf{T} + \mathsf{S}} \qquad \frac{\Gamma \vdash e : \mathsf{S}}{\Gamma \vdash \mathsf{inr}[\mathsf{T}](e) : \mathsf{T} + \mathsf{S}}$$

$$\frac{\Gamma \vdash e : \mathsf{T} + \mathsf{S} \quad \Gamma, x : \mathsf{T} \vdash e_1 : \mathsf{R} \quad \Gamma, y : \mathsf{S} \vdash e_2 : \mathsf{R}}{\Gamma \vdash \mathtt{case}[\mathsf{T}, \mathsf{S}](e,\ x.e_1,\ y.e_2) : \mathsf{R}}$$

Los nuevos valores son:

$$v ::= \ldots \mid \mathsf{inl}[\mathsf{T}](v) \mid \mathsf{inr}[\mathsf{T}](v)$$

La semántica dinámica para la evaluación ansiosa es:

$$\frac{e \to e'}{\mathsf{inl}[\mathsf{T}](e) \to \mathsf{inl}[\mathsf{T}](e')} \qquad \frac{e \to e'}{\mathsf{inr}[\mathsf{T}](e) \to \mathsf{inr}[\mathsf{T}](e')}$$

$$\frac{e \to e'}{\mathtt{case}[\mathsf{T}, \mathsf{S}](e,\ x.e_1,\ y.e_2) \to \mathtt{case}[\mathsf{T}, \mathsf{S}](e',\ x.e_1,\ y.e_2)}$$

$$\frac{}{\mathtt{case}[\mathsf{T}, \mathsf{S}](\mathsf{inl}[\mathsf{S}](v),\ x.e_1,\ y.e_2) \to e_1[x := v]}$$

$$\frac{}{\mathtt{case}[\mathsf{T}, \mathsf{S}](\mathsf{inr}[\mathsf{T}](v),\ x.e_1,\ y.e_2) \to e_2[y := v]}$$

Nuevamente esta extensión con tipos suma es segura. Los detalles se dejan al lector.

2.1. Ejemplos del uso del tipo suma

2.1.1. Booleanos

El tipo booleano puede definirse mediante azucar sintáctica como una suma, en la presencia adicional del tipo unitario como sigue:

$$\mathsf{Bool} =_{def} \mathtt{Void} + \mathtt{Void}$$

$$\mathsf{true} =_{def} \mathsf{inl}_{\mathtt{Void}}() \qquad\qquad \mathsf{false} =_{def} \mathsf{inr}_{\mathtt{Void}}()$$

$$\mathsf{if}\ e_{\mathtt{Void}}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \quad =_{def} \quad \mathsf{case}\ e_{\mathtt{Void}}\ \mathsf{of}\ \{\mathsf{inl}\ x\ \Rightarrow\ e_2 \mid \mathsf{inr}\ y\ \Rightarrow\ e_3\}\ \text{donde}\ x \notin FV(e_2),\ y \notin FV(e_3)$$

2.1.2. Tipos Enumerados

Los tipos enumerados son aquellos con un número finito de valores los cuales se pueden enumerar directamente. Estos tipos son definibles mediantes sumas del tipo unitario, por ejemplo en Haskell el tipo

```
data Color = Verde | Blanco | Rojo
```

puede definirse como

$$\begin{aligned} \mathsf{Color} &=_{def} \mathtt{Void} + (\mathtt{Void} + \mathtt{Void}) \\ \mathsf{Verde} &=_{def} \mathsf{inl}_{\mathtt{Void}+\mathtt{Void}}() \\ \mathsf{Blanco} &=_{def} \mathsf{inr}_{\mathtt{Void}}(\mathsf{inl}_{\mathtt{Void}}()) \\ \mathsf{Rojo} &=_{def} \mathsf{inr}_{\mathtt{Void}}(\mathsf{inr}_{\mathtt{Void}}()) \end{aligned}$$

2.1.3. Tipos Opción

Un tipo de gran interés y utilidad es el tipo opción, llamado `Maybe` en Haskell, definido como

```
data Maybe a = Just a | Nothing
```

el cual puede definirse como

$$\begin{aligned} \mathsf{MaybeT} &=_{def} \mathtt{Void} + \mathsf{T} \\ \mathsf{Just}\, e &=_{def} \mathsf{inr}_{\mathtt{Void}}\, e \\ \mathsf{Nothing} &=_{def} \mathsf{inl}_{\mathsf{T}}() \end{aligned}$$

Este tipo es de gran importancia para un manejo de errores en ciertos casos, por ejemplo usando Maybe Float podemos devolver el valor Nothing al tratar de dividir entre cero.

Existen otros tipos de datos con elecciones como

```
data Nat = Cero | Suc Nat
```

Este no es un tipo suma propiamente dicho puesto que, visto como suma, su definición es $\mathtt{Nat} = \mathtt{Void} + \mathtt{Nat}$ pero esta no es una suma válida pues implica recursión en tipos. De manera que `Nat` es un tipo recursivo, cuya definición formal no trataremos.

2.2. Tipos Variante

Considérese el siguiente tipo en Haskell

```
data Expr = LitN Int | LitF Float | LitA Char | Ident String| Error
```

Este tipo puede representarse como una suma $Int + (Float + (Char + (String + \texttt{Void})))$ sin embargo el acceso a cada componente particular resulta complicado al tener que componer inyecciones, por ejemplo un elemento de tipo `Char` debe etiquetarse mediante $\mathsf{inr}(\mathsf{inr}(\mathsf{inl}))$. Es más fácil manejar directamente las etiquetas `LitN,LitF,LitA,Ident,Error` que utilizar la referencias a su posición en la suma mediante combinaciones de etiquetas $\mathsf{inl}, \mathsf{inr}$. Este mecanismo genera los tipos variantes o sumas etiquetadas.
La sintaxis es:

Sintaxis concreta	Sintaxis abstracta
$[\ell_1 : \mathsf{T}_1, \ell_2 : \mathsf{T}_2, \ldots, \ell_k : \mathsf{T}_k]$	$\mathtt{var}(\ell_1 : \mathsf{T}_1, \ell_2 : \mathsf{T}_2, \ldots, \ell_k : \mathsf{T}_k)$
$\ell\ e$	$\mathtt{in}[\ell](e)$
$\mathsf{case}\ e\ \mathsf{of}\ \{\ell_1\ x_1 \Rightarrow e_1 \mid \ldots \mid \ell_n\ x_n \Rightarrow e_n\}$	$\mathtt{case}[\ell_1 : \mathsf{T}_1, \ell_2 : \mathsf{T}_2, \ldots, \ell_k : \mathsf{T}_k](e, x_1.e_1, \ldots, x_n.e_n)$

La semántica estática se define como sigue:

$$\frac{\Gamma \vdash e : \mathsf{T}_i}{\Gamma \vdash \mathtt{in}[\ell_i](e) : \mathtt{var}(\ell_1 : \mathsf{T}_1, \ldots, \ell_i : \mathsf{T}_i, \ldots, \ell_k : \mathsf{T}_k)}$$

Por otro lado la regla para tipar una expresión case es:

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathtt{var}(\ell_1 : \mathsf{T}_1, \ldots, \ell_k : \mathsf{T}_k) \\ \Gamma, x_1 : \mathsf{T}_1 \vdash e_1 : \mathsf{T} \\ \vdots \\ \Gamma, x_n : \mathsf{T}_n \vdash e_n : \mathsf{T}\end{array}}{\Gamma \vdash \mathtt{case}[\ell_1 : \mathsf{T}_1, \ldots, \ell_k : \mathsf{T}_k](e, x_1.e_1, \ldots, x_n.e_n) : \mathsf{T}}$$

Se le deja al lector la labor de definir la semántica dinámica de esta extensión.
Veamos cómo se ve el tipo de Haskell del ejemplo inicial en nuestra extensión.

```
data Expr = LitN Int | LitF Float | LitA Char | Ident String| Error
```

Las etiquetas son `LitN, LitF, LitA, Ident, Error`.

El tipo `Expr` es entonces una variante con cinco alternativas, definido como

$$\mathtt{var}(\mathtt{LitN} : \mathsf{Int}, \mathtt{LitF} : \mathsf{Float}, \mathtt{LitA} : \mathsf{Char}, \mathtt{Ident} : \mathsf{String}, \mathtt{Error} : \mathsf{Void})$$

Obsérvese que la etiqueta `Error` que no tiene un tipo asociado en Haskell debe declarse aquí con el tipo unitario o bien podemos convenir, mediante azucar sintáctica, que una entrada que sólo tenga una etiqueta ℓ en un tipo variante, significa realmente $\ell : \mathsf{void}$.