Programación Funcional Avanzada

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

Lenguaje Core

En lugar de ser definidos monolíticamente, muchos lenguajes de programación se construyen extendiendo un lenguaje "core"

Más simple de entender y probar propiedades

Características avanzadas pueden:

- extender el lenguaje core
- ser traducidas al lenguaje core

Haskell / GHC

El core es un cálculo lambda tipado explícitamente

- System F (FC)

Extensiones a core

- Módulos
- Tipos de datos (data)
- case
- let

Características expresadas en términos del lenguaje core

- pattern matching
- where
- type classes
- operator sections (ej. $(+2)$, (>3), etc.)

Syntactic sugar

- listas por comprensión, generadores de listas
- notación do

Sistema de tipos

Haskell soporta inferencia de tipos, mientras el lenguaje Core es tipado explícitamente

El proceso de inferencia de tipos (o reconstrucción de tipos) **elabora** el programa

El algoritmo de inferencia de tipos de Haskell está basado en el algoritmo de **Damas-Milner** para el sistema de tipos de **Hindley-Milner**

- cálculo lambda con constantes, recursión y let
- el sistema de tipos se debe extender para tipos de datos, pattern matching, type classes, etc.

La mayoría de las extensiones soportan inferencia de tipos, pero algunas sólo soportan chequeo de tipos

Type Classes

Type classes

Predicados sobre tipos (pero no tipos en sí mismos)

Proveen **polimorfismo ad-hoc** o **sobrecarga**

Extensibles o abiertas

Haskell permite solamente predicados unarios, pero permite clases sobre tipos con diferentes kinds (ej. *Show*, *Monad*)

Existen muchas extensiones

Se puede traducir al cálculo lambda polimórfico

Clases e instancias

Una declaración class define un predicado. Cada miembro de una clase soporta un cierto conjunto de **métodos**

Una declaración instance declara la pertenencia de algún tipo a la clase, proveyendo **evidencia** de esto mediante la implementación de los métodos

Las funciones que usan métodos obtienen **restricciones de clase**, que son como obligaciones de prueba

Polimorfismo paramétrico vs ad-hoc

Polimorfismo paramétrico

$$fst :: (a, b) \to a$$
$$fst\ (x, \_) = x$$

Las variables sin restricciones pueden ser instanciadas para todos los tipos. No se puede hacer ninguna asunción sobre el tipo en la definición de la función. La función trabaja de forma uniforme para todos los tipos.

Polimorfismo ad-hoc

$$maxPair :: Ord\ a \Rightarrow (a, a) \to a$$
$$maxPair\ (x, y) = \text{if } x \leqslant y \text{ then } y \text{ else } x$$

Las variables restringidas sólo pueden ser instanciadas a miembros de la clase. Dado que cada instancia es específica a un tipo, el comportamiento puede variar mucho dependiendo del tipo que se usa.

Restricciones (Haskell 98 y 2010)

1. sólo un parámetro de tipo por clase
2. sólo una instancia por tipo
3. se pueden definir superclases, pero la jerarquía de clases no puede ser cíclica
4. instancias declaradas sólo para "tipos simples": $T\ a_1 \ldots a_n$
   - T no es un sinónimo de tipo
   - $a_1, \ldots, a_n$ son variables de tipo
5. contextos de instancias o clases sólo pueden ser: $C\ a$
   - a es variable de tipo
6. contextos de funciones sólo pueden ser: $C\ (a\ t_1 \ldots t_n)$
   - a es variable de tipo
   - $t_1, \ldots, t_n$ son tipos, pudiendo contener variables

Ejemplos

instance $Eq\ a \Rightarrow Eq\ [a]$	*ok*
instance $Eq\ [a] \Rightarrow Eq\ [a]$	*ilegal* (5)
instance $Eq\ Int \Rightarrow Eq\ Bool$	*ilegal* (5)
instance $Eq\ a \Rightarrow Eq\ (a, Bool)$	*ilegal* (4)
instance $Eq\ [[a]]$	*ilegal* (4)
instance $Eq\ String$	*ilegal* (4)
$(Eq\ (f\ Int)) \Rightarrow f\ Int \rightarrow f\ Int \rightarrow Bool$	*ok*
$(Eq\ [Int]) \Rightarrow [Int] \rightarrow [Int] \rightarrow Bool$	*ilegal* (6)
$(Eq\ [a]) \Rightarrow [a] \rightarrow [a] \rightarrow Bool$	*ilegal* (6)
$(Eq\ a) \Rightarrow [a] \rightarrow [a] \rightarrow Bool$	*ok*

Las restricciones aseguran que la resolución de instancias termine y sea eficiente, y que los contextos se reduzcan lo más posible

Tipos cualificados

Los tipos con contextos también se llaman **tipos cualificados**

Mark Jones 1992 - "A Theory of Qualified Types"

- pueden ser utilizados para llevar la cuenta de propiedades de tipos
- el sistema de type classes de Haskell es una instancia específica

Entailment

Algunos contextos implican otros contextos

$$Eq\ Int \Vdash Eq\ [Int]$$
$$Eq\ Bool, Ord\ Int \Vdash Ord\ Int$$
$$\emptyset \Vdash Eq\ Int$$

La relación $\Vdash$ entre dos contextos se llama **entailment**

Reglas:

$$\frac{Q \subseteq P}{P \Vdash Q}(mono) \qquad \frac{P \Vdash Q \qquad Q \Vdash R}{P \Vdash R}(trans)$$

$$\frac{P \Vdash Q \qquad \varphi \text{ es una sustitución}}{\varphi P \Vdash \varphi Q}(clausura)$$

Entailment - Reglas Derivadas

$$\frac{}{P \Vdash P}(id) \quad \frac{}{P \Vdash \emptyset}(term)$$

$$\frac{}{P, Q \Vdash P}(fst) \quad \frac{}{P, Q \Vdash Q}(snd)$$

$$\frac{P \Vdash Q \qquad P \Vdash R}{P \Vdash Q, R}(uni)$$

Type class entailment

Reglas específicas al sistema de type classes:

$$\frac{P \Vdash \pi \qquad \textbf{class } Q \Rightarrow \pi}{P \Vdash Q}(super)$$

$$\frac{P \Vdash Q \qquad \textbf{instance } Q \Rightarrow \pi}{P \Vdash \pi}(inst)$$

Ejemplo:

$$\frac{\dfrac{}{Ord\ a \Vdash Ord\ a}(id) \qquad \text{class } Eq\ a \Rightarrow Ord\ a}{Ord\ a \Vdash Eq\ a}(super)$$

sólo si tenemos $Eq\ a$ podemos definir $Ord\ a$

Validez de instancias

Las declaraciones de instancias deben cumplir con la jerarquía de clases:

$$\frac{\textbf{class } Q \Rightarrow \pi \qquad P \Vdash \varphi\ Q}{\textbf{instance } P \Rightarrow \varphi\ \pi \text{ es válida}}(\mathit{valid})$$

Ejemplo

class $(Eq\ a, Show\ a) \Rightarrow Num\ a$
class $Foo\ a \Rightarrow Bar\ a$
class $Foo\ a$

instance $(Eq\ a, Show\ a) \Rightarrow Foo\ [a]$
instance $Num\ a \Rightarrow Bar\ [a]$

$$\frac{\dfrac{}{\text{class } Foo\ a \Rightarrow Bar\ a} \qquad \dfrac{\dots}{Num\ a \Vdash Foo\ [a]}}{\text{instance } Num\ a \Rightarrow Bar\ [a] \text{ es válida}}(valid)$$

$$\frac{\dfrac{\dfrac{}{\mathbf{c}\ (Eq\ a, Show\ a) \Rightarrow Num\ a}}{Num\ a \Vdash (Eq\ a, Show\ a)}(c) \qquad \dfrac{}{\mathbf{i}\ (Eq\ a, Show\ a) \Rightarrow Foo\ [a]}}{Num\ a \Vdash Foo\ [a]}(i)$$

Reglas de tipos

Usualmente las reglas de tipos son de la forma:

$$\Gamma \vdash e :: \tau$$

donde Γ es un ambiente que mapea identificadores a tipos, e es una expresión, y τ es un tipo (posiblemente polimórfico)

Con tipos cualificados, las reglas son de la forma:

$$P|\Gamma \vdash e :: \tau$$

donde P es un contexto representando conocimiento (local), y τ es un tipo (posiblemente polimórfico, posiblemente sobrecargado)

Reducción de contexto

$$\frac{P|\Gamma \vdash e :: \pi \Rightarrow \rho \qquad P \Vdash \pi}{P|\Gamma \vdash e :: \rho}(context - reduce)$$

Inferencia de tipos de Haskell aplica esta regla por ejemplo en:

$$(\equiv) \qquad :: (Eq\ a) \Rightarrow a \rightarrow a \rightarrow Bool$$
$$\texttt{"bla"} \equiv \texttt{"bla"} :: Bool$$

requiere $\emptyset \Vdash Eq\ String$

Introducción de contexto

$$\frac{P, \pi | \Gamma \vdash e :: \rho}{P | \Gamma \vdash e :: \pi \Rightarrow \rho}(context - intro)$$

Inferencia de tipos de Haskell aplica esta regla al generalizar en un let o en una declaración top-level:

$$maxPair\ (x, y) = \text{if } x \leqslant y \text{ then } y \text{ else } x$$

su tipo se infiere a $(Ord\ a) \Rightarrow (a, a) \rightarrow a$

Traducción de evidencia

Traducción de type classes

Las clases se pueden traducir al cálculo lambda sin utilizar clases:

- Cada declaración de clase define un tipo record (llamado también **diccionario**)
- Cada declaración de instancia define una función que resulta en el tipo diccionario
- Cada método selecciona el campo correspondiente del diccionario
- Introducción de contexto corresponde a la abstracción de argumentos de tipo diccionario
- Reducción de contexto corresponde a la construcción y aplicación implícita de un argumento diccionario

Ejemplo

```
class E a where
  e :: a → a → Bool

instance E Int where
  e = (==)

instance E a ⇒ E [a] where
  e []     []     = True
  e (x:xs) (y:ys) = e x y and e xs ys
  e _      _      = False

member :: E a ⇒ a → [a] → Bool
member _ []     = False
member a (x:xs) = e a x or member a xs

duplicates :: E a ⇒ [a] → Bool
duplicates []     = False
duplicates (x:xs) =
  member x xs or duplicates xs

is = [[], [1], [2], [1, 2], [2, 1]] :: [[Int]]

main = (duplicates is,
        duplicates (concat is))
```

```
data E a = E
  {e :: a → a → Bool}

e_Int :: E Int
e_Int = E (==)

e_List :: E a → E [a]
e_List e_a = E {e = e'} where
  e' []     []     = True
  e' (x:xs) (y:ys) = e e_a x y and e (e_List e_a) xs ys
  e' _      _      = False

member :: E a → a → [a] → Bool
member e_a _ []     = False
member e_a a (x:xs) = e e_a a x or member e_a a xs

duplicates :: E a → [a] → Bool
duplicates e_a []     = False
duplicates e_a (x:xs) =
  member e_a x xs or duplicates e_a xs

is = [[], [1], [2], [1, 2], [2, 1]] :: [[Int]]

main = (duplicates (e_List e_Int) is,
        duplicates (e_Int) (concat is))
```

Superclases

Los diccionarios contienen diccionarios de sus superclases:

$$\text{class } Eq\ a \Rightarrow Ord\ a \text{ where}$$
$$\quad compare :: a \to a \to Ordering$$
$$\quad \ldots$$

$$\text{data } Ord\ a = Ord$$
$$\quad \{ eq \qquad :: Eq\ a$$
$$\quad , compare :: a \to a \to Ordering$$
$$\quad \ldots \}$$

Métodos polimórficos

Diccionarios cuyos campos almacenan funciones polimórficas:

$$\begin{array}{l}\text{class } \mathit{Functor}\ f \text{ where}\\ \quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b\end{array}$$

$$\begin{array}{l}\text{data } \mathit{Functor}\ f = \mathit{Functor}\\ \quad \{\mathit{fmap} :: \forall\ a\ b.(a \to b) \to f\ a \to f\ b\}\end{array}$$

el constructor tiene tipo de rango 2

$$\mathit{Functor} :: (\forall\ a\ b.(a \to b) \to f\ a \to f\ b) \to \mathit{Functor}\ f$$

Traducción dirigida por tipos

Traducción de evidencia producida durante la inferencia de tipos, aumentando las reglas con los términos traducidos.

Forma de la reglas:

- $P|\Gamma \vdash e \rightsquigarrow e' :: \tau$
- $P \Vdash Q \rightsquigarrow e$

Reglas:

$$\frac{P|\Gamma \vdash e \rightsquigarrow e' :: \pi \Rightarrow \rho \qquad P \Vdash \pi \rightsquigarrow e_\pi}{P|\Gamma \vdash e \rightsquigarrow e'\ e_\pi :: \rho}(context - reduce)$$

$$\frac{P, e_\pi :: \pi|\Gamma \vdash e \rightsquigarrow e' :: \rho}{P|\Gamma \vdash e \rightsquigarrow \lambda e_\pi \rightarrow e' :: \pi \Rightarrow \rho}(context - intro)$$

Ambigüedad y coherencia

$$show \circ read$$

Expresión **ambigua**, porque tiene una restricción mencionando una variable que no ocurre en el resto del tipo:

$$(Show\ a, Read\ a) \Rightarrow String \rightarrow String$$

Los tipos ambiguos no están permitidos en Haskell.

La ambigüedad es una forma de **incoherencia** dado que múltiples traducciones de un programa pueden producir un comportamiento distinto

Especialización

Copia parcialmente evaluada de una función sobrecargada, se cambia tamaño del código por eficiencia. Pragma de GHC `SPECIALISE`

$$maxPair :: (Ord\ a) \Rightarrow (a, a) \rightarrow a$$
$$\{\text{-\# } SPECIALISE\ maxPair :: (Int, Int) \rightarrow Int\ \text{\#-}\}$$

causa que

$$maxPair_{Int} :: (Int, Int) \rightarrow Int$$
$$maxPair_{Int} = maxPair\ ord_{Int}$$

se genere y use cuando normalmente se usaría: $maxPair\ ord_{Int}$

Extensiones

Extensiones al sistema de clases

Se pueden levantar casi todas las restricciones del sistema de clases de Haskell 98

El precio:

- peores propiedades del programa
- menos predictibilidad
- peores mensajes de error
- semánticas e interacciones parcialmente no claras
- posibles bugs del compilador

Sin embargo algunas extensiones son muy útiles

Contextos e instancias flexibles

Levanta las restricciones:

- en la forma de las instancias (`-XFlexibleInstances`)

  instance $Eq\ a \Rightarrow Eq\ (a, Bool)$
  instance $Eq\ [[a]]$

- de usar sinónimos de tipos en las instancias (`-XTypeSynonymInstances`)

  instance $Eq\ String$

- en la forma de los contextos (`-XFlexibleContexts`)

  instance $Eq\ [a] \Rightarrow Eq\ [[a]]$

Instancias solapadas

Con `-XOverlappingInstances` se permiten definiciones de instancias solapadas como

instance $Foo\ a \Rightarrow Foo\ [a]$
instance $Foo\ [Int]$

Si tenemos

instance $Foo\ Int$

Hay dos posibilidades de construir $Foo\ [Int]$. Ambas posibilidades pueden tener un comportamiento distinto (incoherencia).

Se elige la instancia más específica

Instancias solapadas y reducción de contexto

Dado

$$foo :: Foo\ a \Rightarrow a \rightarrow a$$
$$test\ x\ xs = foo\ (x : xs)$$

reducir el contexto de *test* de $Foo\ [a]$ a $Foo\ a$ no permitiría elegir $Foo\ [Int]$

GHC intenta demorar la decisión, por lo que el tipo de *test* es:

$$test :: Foo\ [a] \Rightarrow a \rightarrow [a] \rightarrow [a]$$

Instancias incoherentes

Si quisieramos especificar el tipo

$$test :: Foo\ a \Rightarrow a \rightarrow [a] \rightarrow [a]$$

tendríamos un error, a menos que habilitemos instancias incoherentes (`-XIncoherentInstances`)

Con instancias incoherentes es muy difícil predecir qué instancias se van a elegir. Utilizarlas sólo si se está seguro que diferentes formas de construir una instancia tienen el mismo comportamiento.

Instancias no decidibles

Con instancias no decidibles (`-XUndecidableInstances`) se quita la restricción de que la reducción de contexto realmente reduzca el contexto.
Permite

instance $Foo\ [[a]] \Rightarrow Foo\ [a]$
instance $Foo\ Int \Rightarrow Foo\ Bool$

El chequeador de tipos podría caer en un loop infinito

Type classes multi-parámetros

Permite tener más de un parámetro

$$
\begin{array}{l}
\text{class } \mathit{Collection}\ c\ a \text{ where} \\
\quad \mathit{union} :: c\ a \to c\ a \to c\ a \\
\quad \mathit{elem} :: a \to c\ a \to \mathit{Bool} \\
\quad \emptyset :: c\ a
\end{array}
$$

¿Qué pasa con `Data.IntSet.IntSet`? Aunque soporta todos los métodos no puede ser instancia de *Collection*, ya que no tiene forma $c\ a$

class *Collection ca a* where

$$\begin{aligned} &\mathit{union} :: \mathit{ca} \to \mathit{ca} \to \mathit{ca} \\ &\mathit{elem} \;\; :: \mathit{a} \to \mathit{ca} \to \mathit{Bool} \\ &\emptyset \qquad :: \mathit{ca} \end{aligned}$$

Problema 1:

$$\emptyset :: (\mathit{Collection\ ca\ a}) \Rightarrow \mathit{ca}$$

tiene tipo ambiguo

Problema 2:

$$\mathit{test} :: (\mathit{Collection\ ca\ Bool}, \mathit{Collection\ ca\ String}) \Rightarrow \mathit{ca} \to \mathit{Bool}$$
$$\mathit{test\ coll} = \mathit{elem\ True\ coll\ and\ elem}\ \texttt{"foo"}\ \mathit{coll}$$

tiene tipo correcto, aunque intuitivamente no debería

Dependencias funcionales

class *Collection ca a* | *ca* $\rightarrow$ *a* where

...

Indica que *ca* determina *a*, restringiendo las instancias admisibles

Si tenemos

instance *Collection IntSet Int*

intentar definir luego

instance *Collection IntSet Bool*

no está permitido

Dependencias funcionales y computación a nivel de tipos

Las dependencias funcionales son muy poderosas y permiten codificar muchas computaciones

$$\begin{aligned}
&\text{data } \mathit{Zero} = \mathit{Zero}\\
&\text{data } \mathit{Succ}\ a = \mathit{Succ}\ a\\
&\text{class } \mathit{Add}\ x\ y\ z \mid x\ y \to z \text{ where}\\
&\quad \mathit{add} :: x \to y \to z\\
&\text{instance } \mathit{Add}\ x\ \mathit{Zero}\ x\\
&\text{instance } \mathit{Add}\ x\ n\ r \Rightarrow \mathit{Add}\ x\ (\mathit{Succ}\ n)\ (\mathit{Succ}\ r)
\end{aligned}$$

Si hacemos

```
Main> :t add (Succ Zero) (Succ Zero)
add (Succ Zero) (Succ Zero) :: Succ (Succ Zero)
```

la suma se realiza a nivel de tipos