Modelo de slides UDESC

Rafael Castro, Renan S. Silva

rafaelcgs10@gmail.com

uber.renan@gmail.com

Departamento de Ciência da Computação
Centro de Ciências e Tecnológicas
Universidade do Estado de Santa Catarina

3 de Outubro de 2017

Sumário

- Correspondência Curry-Howard-Lambek
- Análise do Sistema de Tipos de Haskell
- Conclusões

Isomorfismo de Curry-Howard

- Há um isomorfismo entre o Fragmento da Dedução Natural Intuicionista e o Cálculo Lambda Simplesmente Tipado.

Isomorfismo de Curry-Howard

- Há um isomorfismo entre o Fragmento da Dedução Natural Intuicionista e o Cálculo Lambda Simplesmente Tipado.
- A prova de Curry-Howard é uma indução estruturada.
  - Proposições $\leftrightarrow$ tipos e termos $\leftrightarrow$ provas.
  - Normalização de termos $\leftrightarrow$ simplificação de provas.
-

Fórmulas $\leftrightarrow$ Tipos
Provas $\leftrightarrow$ Termos
Redução-β $\leftrightarrow$ Simplificação de provas
Provabilidade $\leftrightarrow$ *Inhabitation*
Teorema $\leftrightarrow$ Tipo habitado

Curry-Howard-Lambek

- Existe uma interpretação categórica para TA e ND: ambos os sistemas são categorias cartesianas fechadas.

Curry-Howard-Lambek

- Existe uma interpretação categórica para TA e ND: ambos os sistemas são categorias cartesianas fechadas.
- Em particular, é possível demonstrar que qualquer lambda-teoria determina uma CCC e, inversamente, uma CCC qualquer gera um cálculo lambda tipado.

Curry-Howard-Lambek

- Existe uma interpretação categórica para TA e ND: ambos os sistemas são categorias cartesianas fechadas.
- Em particular, é possível demonstrar que qualquer lambda-teoria determina uma CCC e, inversamente, uma CCC qualquer gera um cálculo lambda tipado.
- Dedução Natural Intuicionista é uma CCC: proposições são objetos e sequentes são morfismos.

Curry-Howard-Lambek

- Existe uma interpretação categórica para TA e ND: ambos os sistemas são categorias cartesianas fechadas.
- Em particular, é possível demonstrar que qualquer lambda-teoria determina uma CCC e, inversamente, uma CCC qualquer gera um cálculo lambda tipado.
- Dedução Natural Intuicionista é uma CCC: proposições são objetos e sequentes são morfismos.
- Cálculo Lambda Simplesmente Tipado é uma CCC: tipos são objetos e funções são morfismos.

Visão Lógica do Sistema de Tipos de Haskell

- Em Haskell existe o termo `undefined :: a`, que serve para representar uma computação com erro ou um loop infinito. Funciona como um termo "coringa", pois seu tipo unifica para qualquer outro. Qualquer tipo é habitado por ele.

Visão Lógica do Sistema de Tipos de Haskell

- Em Haskell existe o termo `undefined :: a`, que serve para representar uma computação com erro ou um loop infinito. Funciona como um termo "coringa", pois seu tipo unifica para qualquer outro. Qualquer tipo é habitado por ele.
- A lógica isomorfa ao sistema de tipos de Haskell tem todas as proposições como teoremas.

Visão Lógica do Sistema de Tipos de Haskell

- Em Haskell existe o termo `undefined :: a`, que serve para representar uma computação com erro ou um loop infinito. Funciona como um termo "coringa", pois seu tipo unifica para qualquer outro. Qualquer tipo é habitado por ele.
- A lógica isomorfa ao sistema de tipos de Haskell tem todas as proposições como teoremas.
- Haskell não tem o tipo $\perp$, pois não há tipo não habitado.

Visão Lógica do Sistema de Tipos de Haskell

- Em Haskell existe o termo `undefined :: a`, que serve para representar uma computação com erro ou um loop infinito. Funciona como um termo "coringa", pois seu tipo unifica para qualquer outro. Qualquer tipo é habitado por ele.
- A lógica isomorfa ao sistema de tipos de Haskell tem todas as proposições como teoremas.
- Haskell não tem o tipo $\perp$, pois não há tipo não habitado.
- Portanto, o sistema de tipos de Haskell é não trivial.

Theorems for free! em Haskell

- *Theorems for free!* é sobre a possibilidade de derivar teoremas a partir de funções polimórficas (paramétricas). - Também referenciado como Teorema da Abstração de Reynolds.

Theorems for free! em Haskell

- *Theorems for free!* é sobre a possibilidade de derivar teoremas a partir de funções polimórficas (paramétricas). - Também referenciado como Teorema da Abstração de Reynolds.
- Por exemplo, se fornecida uma função

  `f :: [a] -> [a]`

  deduz-se: para todos os tipos `a` e `b`, para toda lista `xs :: [a]`, e para toda função total `g :: a -> b`, tem-se a igualdade

  `map g (f xs) = f (map g xs)` ou

  `(map g) ∘ f = f ∘ (map g).`

Theorems for free! em Haskell

- Há, em Haskell, a função `seq :: a -> b -> b` que serve para a introduzir avaliação estrita: o primeiro argumento é avaliado de maneira estrita e o segundo é retornado.

```
seq ⊥ b =  ⊥
seq _ b = b
```

Theorems for free! em Haskell

- Há, em Haskell, a função `seq :: a -> b -> b` que serve para a introduzir avaliação estrita: o primeiro argumento é avaliado de maneira estrita e o segundo é retornado.

```
seq ⊥ b =  ⊥
seq _ b = b
```

- Com `seq` é possível definir a função

```
tail_seq :: [a] -> [a]
tail_seq (x:xs) = seq x xs,
```

então tomando `tail_seq` como `f` e `const 1` como g, o teorema anterior é quebrado, pois

```
map (const 1) (tail_seq [1 `div` 0]) = ⊥
tail_seq (map (const 1) [1 `div` 0]) = [].
```

Visão Categórica do Sistema de Tipos de Haskell

Segundo a *Wiki* do Haskell:

Os objetos de **Hask** são tipos de Haskell e os morfismos dos objetos `A` a `B` são funções de Haskell do tipo `A -> B`. O morfismo identidade do objeto `A` é `id :: A -> A`, e a composição dos morfismos `f` e `g` é `f . g = x -> f (g x)`. (Tradução do autor)

Visão Categórica do Sistema de Tipos de Haskell

Segundo a *Wiki* do Haskell:

Os objetos de **Hask** são tipos de Haskell e os morfismos dos objetos `A` a `B` são funções de Haskell do tipo `A -> B`. O morfismo identidade do objeto `A` é `id :: A -> A`, e a composição dos morfismos `f` e `g` é `f . g = x -> f (g x)`. (Tradução do autor)

```
undef1 :: a -> b
undef1 = undefined

undef2 :: a -> b
undef2 = \_ -> undefined
```

`seq undef1 1` = $\perp$

`seq undef2 1 = 1`

`undef1 . id = undef2` $\Longrightarrow$ `undef1 . id` $\neq$ `undef1`.

Categoria **PHask**

a. Obj_{PHask} é o conjunto de todos os tipos de Haskell.
b. Mor_{PHask} é o conjunto de todas as funções em Haskell (tipáveis) que são totais. Além disso, um morfismo `f : A` $\rightarrow$ `B` representa a classe de equivalência de funções do tipo `A` ao tipo `B` que tem o mesmo mapeamento.
c. ∂_0 e ∂_1 são as funções que levam, respectivamente, uma função ao seu tipo de origem e destino.
d. A composição é dada pela função `(.) :: (b -> c) -> (a -> b) -> a -> c` e a prova da associatividade é a mesma utilizada em **Set**.
e. O morfimso identidade é gerado pelas funções identidade `id :: a -> a`, que pelo polimorfismo garante que todo objeto (tipo) tem um morfismo (função) identidade.

Observa-se que **PHask**, diferente de **Hask**, não acontece `undef1 . id` $\neq$ `undef1`.

A categoria **PHask** é uma categoria cartesiana fechada

1. O objeto terminal é o tipo `()`.
2. Utiliza-se o dado tupla padrão: o produto é dado pelos morfismos das funções `fst :: (a, b) -> a` e `snd :: (a, b) -> b`, e pelo tipo: `data (a,b) = (,) { fst :: a, snd :: b}`. Assim, para qualquer tipo `r` e para quaisquer funções `f :: r -> a` e `g :: r -> b` deve existir uma única função `u :: r -> (a,b)` capaz de fazer o diagrama abaixo comutar.

$$\begin{tikzcd} & \mathtt{r} \arrow[dl, "\mathtt{f}"'] \arrow[d, dashed, "\mathtt{u}"] \arrow[dr, "\mathtt{g}"] & \\ \mathtt{a} & \mathtt{(a,\ b)} \arrow[l, "\mathtt{fst}"] \arrow[r, "\mathtt{snd}"'] & \mathtt{b} \end{tikzcd}$$

A melhor função `u` que garante a comutatividade desse diagrama somente pode ser:

```
u :: r -> (a, b)
u r = (f r, g r).
```

3. O objeto exponencial em Haskell é formado pelo objeto `b -> c` e pelo morfismo `eval`, onde `b` e `c` são tipos quaisquer e `eval` é gerado pela função

```
eval :: (b -> c, c) -> c
eval (f, x) = f x,
```

pois para todo tipo `a` e função `g :: (a, b) -> c`, existe um único morfismo `curry g`, onde

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f = \x -> \y -> f (x, y),
```

que faz o diagrama abaixo comutar.

$$\begin{tikzcd} \texttt{a} \arrow[d, dashed, "\texttt{curry g}"] \\ \texttt{b -> c} \end{tikzcd} \qquad \begin{tikzcd} \texttt{(a, b)} \arrow[d, "\texttt{(curry g, id)}"'] \arrow[dr, "\texttt{g}"] & \\ \texttt{(b -> c, b)} \arrow[r, "\texttt{eval}"'] & \texttt{c} \end{tikzcd}$$

Visão Categórica de Classes de Tipos

- As classes de tipos do Haskell são utilizadas para definir algumas estruturas matemáticas, como relações (por exemplo: equivalência e ordem), semigrupos, monoides, funtores e mônadas.

Visão Categórica de Classes de Tipos

- As classes de tipos do Haskell são utilizadas para definir algumas estruturas matemáticas, como relações (por exemplo: equivalência e ordem), semigrupos, monoides, funtores e mônadas.
- A classe `Functor` indica, por seu nome, representar as estruturas categóricas dos funtores, portanto deve ser um mapeamento entre categorias, o qual preserva origem e destino dos morfismos, identidade dos objetos e a composição

Visão Categórica de Classes de Tipos

- As classes de tipos do Haskell são utilizadas para definir algumas estruturas matemáticas, como relações (por exemplo: equivalência e ordem), semigrupos, monoides, funtores e mônadas.
- A classe `Functor` indica, por seu nome, representar as estruturas categóricas dos funtores, portanto deve ser um mapeamento entre categorias, o qual preserva origem e destino dos morfismos, identidade dos objetos e a composição
- Um construtor de tipos é uma função que recebe tipos como argumentos e retorna algum tipo, por exemplo o construtor `Either` recebe dois tipos `a, b` e retorna o tipo `Either a b`.
- ```
  class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b.
  ```
  ```

Visão Categórica de Classes de Tipos

- A assinatura de `fmap` garante a preservação da origem e do destino, porém a identidade e a composição não são asseguradas. Seria necessário que

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

Conclusões

- Curry-Howard-Lambek é utilizado no sistema de tipos de Haskell, não somente para definir algumas classes de tipos, mas também para garantir o bom comportamento do sistema.
- Foi identificado alguns problemas em Haskell, em particular a ausência de uma semântica operacional que seria fundamental para definir uma categoria dos tipos e funções.
- Como alternativa foi construído a categoria **PHask** a partir de um subconjunto de Haskell com apenas funções totais.