



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



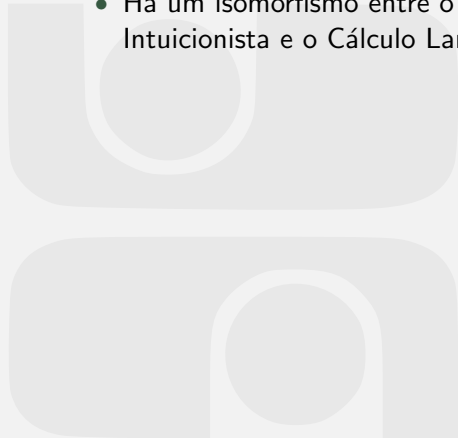
Sumário

- 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

$f :: [a] \rightarrow [a]$

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

$\text{map } g (f \text{ } xs) = f (\text{map } g \text{ } xs)$ ou
 $(\text{map } g) \circ f = f \circ (\text{map } g).$



Theorems for free! em Haskell

- Há, em Haskell, a função $\text{seq} :: a \rightarrow b \rightarrow b$ que serve para a introduzir avaliação estrita: o primeiro argumento é avaliado de maneira estrita e o segundo é retornado.

$$\text{seq } \perp \ b = \ \perp$$

$$\text{seq } _ \ b = b$$



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.

$$\text{seq } \perp \ b = \ \perp$$
$$\text{seq } _ \ b = b$$

- Com `seq` é possível definir a função

$$\text{tail_seq} :: [a] \rightarrow [a]$$
$$\text{tail_seq } (x:xs) = \text{seq } x \ xs,$$

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

$$\text{map } (\text{const } 1) \ (\text{tail_seq } [1 \ \backslash \text{div} \ 0]) = \perp$$
$$\text{tail_seq } (\text{map } (\text{const } 1) \ [1 \ \backslash \text{div} \ 0]) = [].$$



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 $A \rightarrow B$. O morfismo identidade do objeto A é $\text{id} :: A \rightarrow A$, e a composição dos morfismos f e g é $f \cdot g = x \rightarrow f (g x)$. (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 $A \rightarrow B$. O morfismo identidade do objeto A é $\text{id} :: A \rightarrow A$, e a composição dos morfismos f e g é $f \cdot g = x \rightarrow f (g x)$. (Tradução do autor)

```
undef1 :: a -> b
```

```
undef1 = undefined
```

```
undef2 :: a -> b
```

```
undef2 = \_ -> undefined
```

```
seq undef1 1 = ⊥
```

```
seq undef2 1 = 1
```

$\text{undef1} \cdot \text{id} = \text{undef2} \implies \text{undef1} \cdot \text{id} \neq \text{undef1}$.



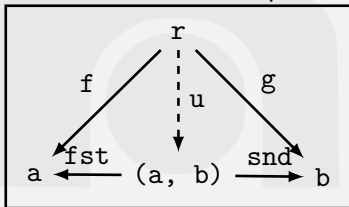
Categoria **PHask**

- 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 $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$ e a prova da associatividade é a mesma utilizada em **Set**.
- e O morfismo identidade é gerado pelas funções identidade $id :: a \rightarrow a$, 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 **P**Hask 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 $\text{fst} :: (a, b) \rightarrow a$ e $\text{snd} :: (a, b) \rightarrow b$, e pelo tipo:
 $\text{data } (a,b) = (,) \{ \text{fst} :: a, \text{snd} :: b \}$. Assim, para qualquer tipo r e para quaisquer funções $f :: r \rightarrow a$ e $g :: r \rightarrow b$ deve existir uma única função $u :: r \rightarrow (a,b)$ capaz de fazer o diagrama abaixo comutar.



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

$$u :: r \rightarrow (a, b)$$
$$u \, r = (f \, r, g \, r).$$

- ③ O objeto exponencial em Haskell é formado pelo objeto $b \rightarrow c$ e pelo morfismo eval , onde b e c são tipos quaisquer e eval é gerado pela função

$$\text{eval} :: (b \rightarrow c, c) \rightarrow c$$

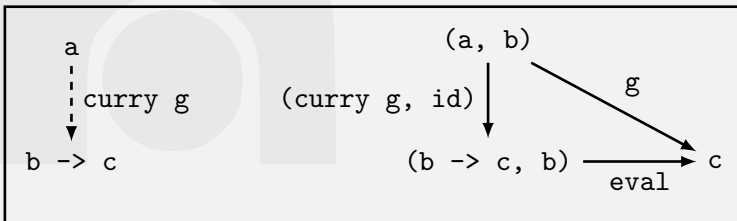
$$\text{eval} (f, x) = f \ x,$$

pois para todo tipo a e função $g :: (a, b) \rightarrow c$, existe um único morfismo $\text{curry } g$, onde

$$\text{curry} :: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$$

$$\text{curry } f = \lambda x \rightarrow \lambda y \rightarrow f (x, y),$$

que faz o diagrama abaixo comutar.





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

$$\text{fmap id} = \text{id}$$
$$\text{fmap (f . g)} = \text{fmap f} . \text{fmap g}$$



# 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.