Seminário de MFO
Idris

Bruno Rafael dos Santos,
Fernando Martins Hillesheim
MFO - Métodos Formais
Professor: Cristiano D. Vasconcelos

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Introdução

Idris é uma linguagem de programação funcional com tipos dependentes.

Exploraremos tipos primitivos, tipos de dados, funções, totalidade, tipos dependentes e provas.

Objetivo da Apresentação:

- Apresentar os fundamentos e características essenciais do Idris.
- Destacar como a linguagem aborda tipos dependentes, provas e teoremas.
- Demonstrar exemplos práticos de código para ilustrar os conceitos apresentados.

Sobre Idris

Origens e Desenvolvimento:

- Criada por Edwin Brady, a primeira versão foi lançada em 2011.
- Desenvolvimento contínuo pela comunidade de código aberto.

Idris em Resumo:

- **Paradigma:** Funcional com foco em tipos dependentes.
- **Objetivo:** Proporcionar alta expressividade e segurança de tipos.

Tipos Primitivos em Idris

Declaração de Variáveis

```
module Main
word : String
word = "Some string"
x, y : Int
x = 10
y = -5
value : Double
value = 0.25
letter : Char
letter = 'a'
flag : Bool
flag = False
```

Assim como em diversas outras linguagens de programação existem tipos já definidos previamente como: **Int, Double**, **Char**, **Ptr** (Ponteiros), **Bool** e **Nat**.

Para toda variável dentro do código deve ser primeiro declarado o tipo e depois seu valor nesta ordem.

Além disso a indentação importa, isto é, novas declarações devem estar no mesmo nível de indentação de sua anterior ou a anterior deve ser finalizada com ";", pois caso o contrário a declaração nova será reconhecida como continuação da anterior.

Tipos Primitivos em Idris

Informações Adicionais sobre Variáveis

Caso o arquivo deste código seja aberto com: **idris Seminario.idr**

Pode-se digitar apenas o nome da variável no terminal idris para imprimir o valor da variável ou utilizar o comando :t [variável] para imprimir o tipo da variável, como no exemplo da figura a seguir:

```
Seminario> value
0.25
Seminario> :t value
Seminario.value : Double
```

Além de comandos como estes, utilizando variáveis do tipo Bool podemos construir expressões utilizando operadores (existem muitos já definidos na biblioteca prelude como **+, -, *, /), if, then, else e etc.**; segue um exemplo:

```
Seminario> if x - y <= 10 then "Strange" else if x - y == 15 then "Ok!" else "Not ok!"
"Ok!"
```

UDESC

Tipos de Dados em Idris

- Em idris um tipo de dado é declarado com a palavra da **data**
- Todo tipo de dado deve ter nome iniciado com letra maiúscula assim como seus construtores
- Também pode-se representar os construtores por sequência de símbolos de operadores para representar um construtor e neste caso, deve-se colocar o nome do construtor dentro de parênteses.
- Uso de **infixr** e **infixl** para definir associatividade à direita e à esquerda respectivamente.
- Símbolos de operadores para serem utilizados em nome de construtores:
  :, ., +, -, =, \, /, ~, ?, |, &, >, <, @, $, %, ^

Tipos de Dados em Idris

```
data Nat_a = Z_a | S_a Nat_a

data Nat_b : Type where
    Z_b : Nat_b
    S_b : Nat_b -> Nat_b

data List : Type -> Type where
        Nil : List a
        (::) : a -> List a -> List a

infixr 10 ::

data MyList a = MyNil | (:::) a (MyList a)
```

UDESC

Funções em Idris

Construção de funções

- Os Nomes das funções não precisam iniciar com letra maiúscula.
- É obrigatório a declaração do tipo.
- funções são declaradas utilizando casamento de padrão, de modo que haja um caso base (ou mais) e uma recursão (ou mais).

```
even: Nat -> Bool
even 0 = True
even (S 0) = False
even (S (S k)) = even k

fold : (f : a -> b -> b) -> b -> List a -> b
fold f x [] = x
fold f x (y :: ys) = f y (fold f x ys)
```

UDESC

Funções em Idris

Construção de funções

- Podemos também declarar funções dentro de funções (e também tipos de dados e variáveis) utilizando como recurso a palavra **where**.
- Existem também outras palavras que podem ser usadas ao se fazer declarações de funções, das quais grande parte são demonstradas no seguinte exemplo.

```
foo : Int -> Int
foo x = case isLT of
            Yes => x*2
            No => x*4
    where
       data MyLT = Yes | No

       isLT : MyLT
       isLT = if x < 20 then Yes else No
```

Totalidade e Parada

Em idris todas as **funções devem ser totais**, isto é, para qualquer elemento possível do domínio a função deve levar a algum elemento do domínio (os casamentos de padrão devem cobrir todas as possibilidades de argumentos).

Ao tentar carregar um código com uma função não total como a seguinte:

```
fromMaybe : Maybe a -> Nat
fromMaybe (Just x) = 0
```

Idris irá gerar um erro indicando que a determinada função não é total, como mostrado abaixo:

```
Error: fromMaybe is not covering.
```

Este erro pode ser ignorado utilizando a notação **partial** da seguinte forma:

```
partial fromMaybe : Maybe a -> Nat
fromMaybe (Just x) = 0
```

Totalidade e Parada

Note que apesar da possibilidade de se utilizar partial isto não é recomendado e caso se aplique a função sobre um argumento que não se encaixa no casamento de padrão ocorrerá um erro em tempo de execução.

Outro detalhe importante é que Idris além de checar a se as funções são totais também verifica se as chamadas recursivas diminuem os argumentos de modo que se garante que a função para em algum momento.

Tipos dependentes

```
data Vect_main : Nat -> Type -> Type where
    Nil_main  : Vect_main Z a
    (:::) : a -> Vect_main k a -> Vect (S k) a

    infixr 10 :::
```

De acordo com a documentação Idris [1] sobre o tipo **Vect** conclui-se que o tipo **Vect_main** é indexado sobre **Nat** e parametrizado por **Type**.

Então executando a verificação de tipo sobre um vetor de exemplo:

: t (1 ::: (2 :: Nil_main))

tem-se a seguinte resposta no terminal:

```
Test> :t 1 ::: (2 ::: Nil_main)
1 ::: (2 ::: Nil_main) : Vect_main 2 Integer
```

Ao utilizarmos **Vect_main** como vetor a verificação de tipo no código é capaz de inferir o tamanho do vetor dado que o tipo do vetor é indexado por seu tamanho, logo pode-se notar que a verificação de tipo aplicada a tipos dependente pode ser utilizada para garantir propriedades em um programa, como por exemplo propriedades sobre o tamanho de um vetor.

Holes

Em Idris um programa pode ter **"holes"** (buracos em português) que representam partes ainda não escritas de um programa, como no seguinte exemplo sobre a função **map** (que aplica uma função sobre todos os elementos de uma lista):

```
map : (f : a -> a) -> List a -> List a
map f [] = []
map f (x :: xs) = ?map_rhs_1
```

Ao se verificar o tipo de **map_rhs_1** teremos:

```
Main> :t map_rhs_1
 0 a : Type
   x : a
   xs : List a
   f : a -> a
------------------------------
map_rhs_1 : List a
```

Note que o hole **map_rhs_1** é indicado pelo operador **?**. Estes holes são úteis na construção de programas (e provas, como demonstrado mais adiante) pois permitem escrever o programa de maneira incremental ao informar qual o tipo esperado na expressão a seguir.

PROVAS E TEOREMAS

Provar teoremas em Idris é semelhante a escrever uma função com o seguinte detalhe: a função retorna a "prova" do teorema.

Para realização de maioria das provas tem-se o tipo de dado **(=)**, declarado da seguinte forma:

```
data (=) : a -> b -> Type where
    Refl : x = x
```

Outro tipo de dado semelhante é o **Le** que é equivalente ao menor igual, descrito a seguir:

```
data Le : Nat -> Nat -> Type where
    Le_n : (n : Nat) -> Le n n
    Le_S : (n, m : Nat) -> Le n m -> Le n (S m)
```

Ter noção sobre estes tipos de dados (além de muitos outros utilizados em provas) é útil visto que em Idris, para a realização de provas, se utiliza na maioria dos casos a lógica construtivista, sendo assim, existem proposições que podem ser construídas e portanto representam algo que é verdadeiro, existem proposições que levam a algum absurdo e portanto são falsas e por último existem proposições que não podem ser provadas.

Indução e Táticas

```
plus_comm : (n, m : Nat) -> plus n m = plus m n
```

```
plus_comm : (n, m : Nat) -> plus n m = plus m n
plus_comm n m = ?plus_comm_rhs
```

```
plus_comm : (n, m : Nat) -> plus n m = plus m n
plus_comm 0 m = ?plus_comm_rhs_0
plus_comm (S k) m = ?plus_comm_rhs_1
```

Tratando inicialmente do caso base pode-se verificar o tipo do hole **plus_comm_rhs_0** que é dado como **m = plus m 0** (note que **m = plus m 0** é do tipo **Type**.

A seguir usaremos o teorema (função) abaixo:

```
plus_reduce_other_side : (n : Nat) -> n = plus n Z
```

Indução e Táticas

Independente da maneira que foi provado este teorema, pode-se verificar como é a função que lhe representa e que se fornecermos m para esta ela irá retornar **m = plus m Z** o que é justamente o que se quer para substituir o hole **plus_comm_rhs_0**.

```
plus_comm : (n, m : Nat) -> plus n m = plus m n
plus_comm 0 m = plus_reduce_other_side m
plus_comm (S k) m = ?plus_comm_rhs_1
```

Verificando o tipo do hole restante **plus_comm_rhs_1** tem-se **S(plus k m) = plus m (S k)**.

```
plus_S_r : (n, m : Nat) -> plus n (S m) = S (plus n m)
```

Esta função com argumentos **m** e **k** retorna **plus m (S k) = S (plus m k)**, e para situações com esta existe em Idris a função replace no módulo prelude cujo tipo é **(x = y) -> P x -> P y**.

Indução e Táticas

Para utilização desta função deve-se saber qual é o elemento **P** que se deseja, no entanto Idris possui um açúcar sintático para utilização de replace que é **rewrite** que calcula **P** e permite que se possa então reescrever um expressão fornecendo uma igualdade:

```
plus_comm : (n, m : Nat) -> plus n m = plus m n
plus_comm 0 m = plus_reduce_other_side m
plus_comm (S k) m = rewrite plus_S_r m k in ?plus_comm_rhs_1
```

Tendo aplicado **rewrite** o tipo de **plus_comm_rhs_1** é **agora S(plus k m) = S(plus m k)**, sabendo disso note que a igualdade **plus k m = plus m k** é o retorno da aplicação de **plus_comm** sobre **k** o que pode-se tratar como a hipótese de indução, assim dado esta igualdade fornecida por **plus_comm k** podemos novamente utilizar **rewrite**:

```
plus_comm : (n, m : Nat) -> plus n m = plus m n
plus_comm 0 m = plus_reduce_other_side m
plus_comm (S k) m = rewrite plus_S_r m k in
    rewrite plus_comm m k in ?plus_comm_rhs_1
```

Indução e Táticas

Verificando novamente o tipo do hole **plus_comm_rhs_1** tem-se **S(plus k m) = S(plus k m)**; note que agora tem-se uma igualdade e qualquer igualdade pode ser construída por meio do construto **Refl**, sendo assim basta substituir o hole completamente por **Refl**:

```
plus_comm : (n, m : Nat) -> plus n m = plus m n
plus_comm 0 m = plus_reduce_other_side m
plus_comm (S k) m = rewrite plus_S_r m k in
    rewrite plus_comm m k in Refl
```

Quantificadores

Até o agora todos os exemplos estavam relacionados ao quantificador ∀, no entanto pode se querer utilizar o quantificador ∃. Para isso, há uma maneira de o fazer apresentada por Jesse Hallett [2] utilizando o tipo **DPair** ("dependent pair") que de acordo com a documentação Idris [1] tem a seguinte definição:

```
data DPair : (a : Type) -> (P : a -> Type) -> Type where
    MkDPair : {P : a -> Type} -> (x : a) -> P x -> DPair a P
```

Esta definição indica que um elemento do tipo **DPair** é basicamente um dupla de algum elemento **x** de tipo **a** e uma proposição **P** (a declaração utilizando chaves **P : a -> Type** indica que **P** é um argumento implícito do construtor) sobre este elemento. Tal tipo funciona de maneira equivalente ao quantificador ∃ e por isso é utilizado em provas em Idris. A seguir tem-se um exemplo de declaração que equivale ao teorema:

$$\forall n \in \mathbf{N},\ even(n) = true \Rightarrow \exists m \in \mathbf{N},\ n = 2 \cdot m$$

que é:

```
even_mult_2 : (n : Nat) -> even n = True -> (m : Nat ** n = 2 * m)
```

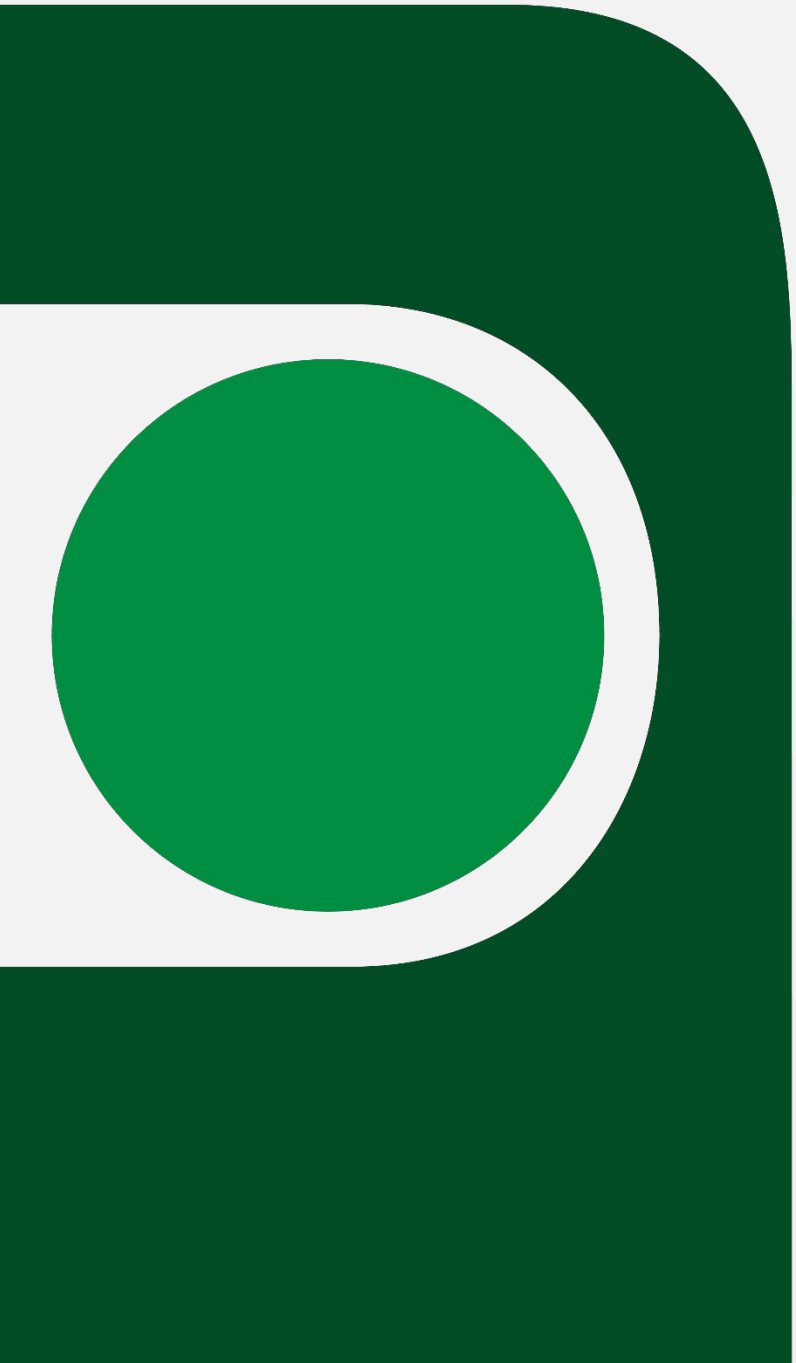
A equivalência com o quantificado ∃ pode ser notada visto que o par só pode ser construído caso exista algum m de tipo **Nat** que satisfaça a condição **n = 2 * m**.

UDESC

Referências

[1] The Idris Community. 2023. Idris. Retrieved November 22, 2023 from https://docs.idris-lang.org/en/latest/index.html

[2] Jesse Hallett. 2014. Category Theory proofs in Idris. Retrieved November 22, 2023 from https://sitr.us/2014/05/05/category-theory-proofs-in-idris.html , Vol. 1, No. 1, Article . Publication date: November 2023.



FIM