Seminário de MFO

BRUNO RAFAEL DOS SANTOS, Universidade do Estado de Santa Catarina, BR
FERNANDO MARTINS HILLESHEIM, Universidade do Estado de Santa Catarina, BR

Este trabalho busca apresentar conteúdos introdutórios sobre a linguagem Idris, passando por representação de dados (e tipos) até criação de funções e teoremas. Idris é uma linguagem funcional de propósito geral e além disso possui tipos dependentes.

Additional Key Words and Phrases: Idris, Programação funcional, Tipos dependentes, Teoremas

ACM Reference Format:
Bruno Rafael dos Santos and Fernando Martins Hillesheim. 2023. Seminário de MFO. 1, 1 (November 2023), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUÇÃO

Idris é uma linguagem funcional com tipos dependentes que serve tanto para provar teoremas quanto para propósitos gerais e em que é possível compilar arquivos em executáveis. O fato de ser uma linguagem funcional implica que esta faz parte das linguangens de programação consideradas como tendo um idioma funcional, o que significa que na construção de seus programas busca realizar computação sem efeitos colaterais (IO, alterar variáveis ou ponteiros e etc). Além disso, Idris tem funções como "cidadãos de primeira classe", isto significa que estas são como quaisquer outros dados podendo ser passadas como argumentos ou retornos de outras funções. A linguagem Idris também possui uma ferramenta de extrema utilidade que são tipos independentes, estes como veremos a seguir permitem que programas escritos nesta linguagem tenham maior confiabilidade.

2 INSTALAÇÃO

2.1 Linux

Para instalar o Idris no Linux, é necessário ter o Cabal instalado. Se você ainda não o possui, pode instalá-lo executando os seguintes comandos: **`sudo apt-get update`**; **`sudo apt-get install cabal-install`**. Para verificar se o Cabal foi instalado com sucesso, digite **`cabal –version`**.

Antes de prosseguir com a instalação do Idris, certifique-se de que o diretório do Cabal está incluído no seu **`$PATH`**. Caso não esteja, adicione **`/.cabal/bin`** ao seu **`$PATH`**.

Com o Cabal devidamente instalado e configurado, você pode prosseguir com a instalação do Idris usando o comando cabal update; cabal install idris. Para verificar se o Idris foi instalado corretamente, utilize o comando **`idris –version`**.

Authors' addresses: Bruno Rafael dos Santos, Universidade do Estado de Santa Catarina, Joinville, Santa Catarina, BR; Fernando Martins Hillesheim, Universidade do Estado de Santa Catarina, Joinville, Santa Catarina, BR.

ACM XXXX-XXXX/2023/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.2 Mac OS

Para instalar o Idris no MAc OS, é necessário ter o Cabal instalado. Se você ainda não o possui, pode instalá-lo executando os seguintes comandos: **`brew update`**; **`brew install cabal-install`**. Para verificar se o Cabal foi instalado com sucesso, digite **`cabal -version`**.

Antes de prosseguir com a instalação do Idris, certifique-se de que o diretório do Cabal está incluído no seu **`$PATH`**. Caso não esteja, adicione **`$PATH`** ao seu **`/Library/Haskell/bin`**.

Com o Cabal devidamente instalado e configurado, você pode prosseguir com a instalação do Idris usando o comando **`cabal update`**; **`cabal install idris`**. Para verificar se o Idris foi instalado corretamente, utilize o comando .

2.3 Windows

Para instalar o Idris no Windows, é necessário ter o Cabal instalado. Se você ainda não o possui, no Windows, o Cabal geralmente é incluído com a Haskell Platform. Você pode instalar a Haskell Platform, que inclui o GHC (Compilador Glasgow Haskell) e o Cabal, baixando-a no site oficial do Haskell. Para verificar se o Cabal foi instalado com sucesso, digite **`cabal -version.`** Antes de prosseguir com a instalação do Idris, certifique-se de que o diretório do Cabal está incluído no seu **`$PATH`**. No Windowns o Cabal geralmente instala programas em **`%HOME%\AppData\Roaming\cabal\bin`**.

Com o Cabal devidamente instalado e configurado, você pode prosseguir com a instalação do Idris usando o comando **`cabal update`**; **`cabal install idris`**. Para verificar se o Idris foi instalado corretamente, utilize o comando **`idris -version`**.

3 SINTAXE

A sintaxe é semelhante a sintaxe de Haskell; para mais fácil explicação tem-se os itens a seguir.

3.1 Exemplo de arquivo e comandos no terminal

Os arquivos em Idris tem exetensão ".idr" e no inicio do programa deve haver uma declaração da forma **`Module NomeDoArquivo`** (sendo o arquivo **`NomeDoArquivo.idr`**). Considerando um arquivo com o seguinte código:

```
module Main
main : IO ()
main = putStrLn "Hello World"
```

Neste código main é um objeto (que é também semelhante a uma função) de tipo **`IO ()`** (entrada e saída) e é igual ao resultada da função putStrLn "Hello World". Este código pode ser compilado através do seguinte comando no terminal:

```
idris Main.idr -o Main
```

O executável **`Main`** criado será colocado dentro de um diretório **`/build/exec/`**, portanto poderá se executar o arquivo pelo comando:

```
./build/exec/hello
```

3.2 Tipos primitivos

Assim como em diversas outras linguagens de programção existem tipos já definidos previamente (no caso da linguagem Idris a biblioteca `prelude` é importada automaticamente) 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 e 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.

No exemplo a seguir tem-se um código idris com algumas variáveis declaradas:

```
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
```

Caso o arquivo desse 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
```

Fig. 1. Verificação do valor e tipo de uma variável, via autores.

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!"
```

Fig. 2. Construção de uma expressão com tipos booleanos, via autores.

3.3 Tipos de dados

Em Idris um tipo de dado é declarado com a palavra da **data** e existem diversas maneiras de declarar seus construtores, o que em geral funciona de modo semelhante a proposições indutivas em Coq, porém todo tipo de dado deve ter nome inciado com letra maiúscula assim como seus construtores caso o construtor tenha nome com letras, pois pode ser que se deseje utilizar uma sequência de símbolos de operadores para representar um construtor e neste caso, deve-se colocar o nome do construtor dentro de parênteses. Além disso os construtores podem ser escritos de maneira separa

por "|" ou por linhas e não é necessária a declaração de seu tipo (todo tipo tem tipo **Type**), mas caso seja feita a declaração é necessário associar o tipo aos seus construtores com a palavra "where", caso contrário se utiliza o símbolo "=". Por fim, os argumentos de construtores tais como os argumentos do próprio tipo podem ser instanciados (caso o tipo não seja definido o Idris irá inferir o tipo pelo contexto, como no tipo de dado **MyList** apresentado adiante). No exemplo a seguir tem-se 2 declarações diferentes de tipos de números naturais e 2 declarações diferentes de tipos de lista:

```
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)
```

Os símbolos para operadores (quais podem ser utilizados na representação de construtores) são: :, ., +, -, =, \, /, ˜, ?, |, &, >, <, @, $, %, ˆ, no entanto deve-se observar que algumas combinações destes símbolos são palavras reservadas, quais são: "%", "\", ":", "=", "|", "|||", "<-", "->", "=>", "?", "!", "&", "**", "..".

Sobre o uso da palavra **infixr** na declaração de **List**, caso houvesse apenas a declaração do operador (construtor) **(::)** dentro da função este funcionaria apenas de maneira prefixa mas com a declaração **infixr** 10 **::** o operador pode ser usado de maneira infixa com associatividade à direita (de nível 10) ao ser declarado sem parênteses. Para caso se deseje inferir associatividade a esquerda em um operador pode-se utilizar de maneira semelhante **infixl**.

Quanto ao polimorfismo no tipo **MyList** e **List** tem-se exemplos disso e para tal ambos são definidos de modo que recebem um argumento de tipo **Type** (tipo dos tipos). Além disso deve-se destacar que em geral podem ser utilizados construtores com nomes iguais para diferentes tipos de dados se estes estiverem em diferentes módulos, pois Idris normalmente é capaz de inferir a qual tipo pertence o construtor por meio do contexto em que ele é utilizado (caso se importem módulos que levem a esta situação).

3.4 Funções

As funções em Idris também são muitos semlhantes aos Fixpoints em Coq e também as declarações de função em Haskell no entanto, se comparando se com o que se tem em Haskell existem diferenças significativas visto que os nomes não precisam iniciar com letra maiúscula e é obrigatório a declaração do tipo. De restante vale destacar que semelhantemente as outras linguagens citadas as funções são declaradas utilizando casamento de padrão, de modo que haja um caso base (ou mais) e uma recursão (ou mais). A seguir tem-se como exemplos as funções `even` e `fold`:

```
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)
```

Podemos também declarar funções dentro de funções (e também tipos de dados e variáveis) utilizando como recurso a palavra **where**, como no seguinte exemplo da função `lenght` (que calcula o tamanho de uma lista):

```
length : List a -> Nat
length [] = 0
length (x :: xs) = fold count 0 (x :: xs) where
    count : a -> Nat -> Nat
    count n m = plus 1 m
```

Existem também outras palavras que podem ser usadas ao se fazer declarações de funções, das quais grande parte são demonstradas no seguinte exemplo proveniente da documentação oficial da linguagem:

```
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
```

Pode-se verificar que na definição de `foo` há dentro o uso de uma estrutura de seleção **case** semelhante ao **switch** na linguagem C e há a definição de um tipo de dado **MyLT** e uma função `isLT` qual utiliza a variável `x` sem recebe-la com argumento, o que pode ser feito pois a variável existe no contexto da função `foo` (é o argumento de `foo`).

3.5 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). Por padrão ao tentar se carregar um código com um função não total o Idris irá gerar um erro indicando que a determinada função não é total. A seguir tem-se um exemplo sobre a função fromMaybe inspirada em outro exemplo presente na documentação [1]:

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

Ao se tentar carregar um código contendo esta definição é apresentado o seguinte erro:

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

Fig. 3. Erro por definição de função não total, via autores.

Este erro pode ser ignorado utilizando a notação **partial**, conforme demosntrado na documentação Idris [1], da seguinte forma:

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

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.

3.6 Tipos dependentes

Um destaque desta linguagem é o fato de possuir tipos serem dependentes, isto é, seus tipos são tratados como construtores de primeira classe, assim podem ser manipulados como quaisquer outros valores. Um exemplo disto é o tipo dado **Vect_main** com sua declaração em código (semelhante ao tipo de dado **Vect** presente na documentação Idris [1]) demonstrada a seguir:

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

    infixr 10 :::
```

Note que neste código a construção de um tipo depende de um número natural assim como de um tipo. De maneira mais detalhada o construtor **Nil_main** não recebe nenhum argumento e devolve o construtor **Vect_main** aplicado sobre o **Nat** que é `0` e um **Type** a (**Type** é o tipos dos tipos), o que portanto é um **Type**, já o construtor **(::)** recebe um **Type** a e um **Vect_main** k a e então constrói um elemento de tipo **Vect_main (S** k**)** a. 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**.

A vantagem na utilização de tipos dependentes pode ser demonstrada claramente através deste exemplo se observando o seguinte: 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 que nesse caso é o natural utilizado em sua construção, sendo assim um vetor vazio tem como tipo **Vect_main** `0` a mas caso se concatene um elemento de tipo x a um vetor vazio se terá um vetor com tipo **Vect_main** `1` x e assim por diante, portanto a verificação de tipo no código ajuda a

garantir que o código tenha um determinado resultado. Como exemplo ao abrir um arquivo Idris que contenha a definição apresentada de **Vect_main** com o comando:

```
idris NomeDoArquivo.idr
```

e então executando a verificação de tipo sobre um vetor de exemplo:

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

tem-se a seguinte resposta no terminal:

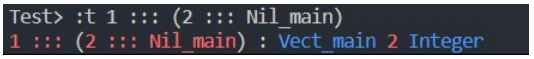

Fig. 4. Verificação do tipo de um elemento **Vect_main**, via autores.

Após os dois pontos tem-se o tipo do elemento declarado e neste pode-se verificar qual o tamanho do vetor (2) pois o tipo depende do tamanho do mesmo.

3.7 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 um 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:

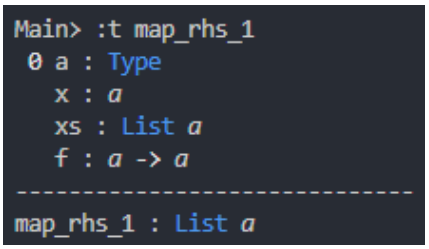

Fig. 5. Verificação do tipo do hole `map_rhs_1`, via autores.

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.

4 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 **(=)**, que de acordo com a documentação Idris [1] é 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, não se trabalha com provas por meio da lógica booleana da qual a maioria das pessoas está acostumada, pelo contrário a lógica que se utiliza na maioria dos casos em Idris é a 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.

4.1 Indução e Táticas

Para apresentar a maneira de se realizar provas se apresenta a seguir uma prova por indução em que tem-se a seguinte proposição (função):

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

Escrever a prova desta proposição é semelhante a escrever um função, e aliás pode-se dizer que `plus_comm` é uma função que recebe dois naturais n e m e constrói uma prova de que `plus n m = plus m n`. Sendo assim deve-se tratar todos os casos possíveis de n e m (dado que função tem que ser total) utilizando casamento de padrão, vale por isso destacar que existem uma série de ferramentas interativas que permitem adicionar clausulas de maneira automaticamente para funções e que também permitem gerar novas clausulas de acordo com os casos possíveis de um dos argumentos. Tem-se utilizando a adição automática de clausulas:

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

E gerando clausulas para todos os casos de `n` tem-se:

```
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`, aliás, assim sempre que se escreve uma proposição se gera um novo tipo). Para resolver isto, no presente trabalho, foi provado o seguinte teorema qual diz que $\forall n \in \mathbf{N}, n = n + 0$:

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

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`, e ao se substituir pode-se carregar o código para se verificar que não ocorreu nenhum erro e portanto tal substituição é válida.

```
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)`, mas note que não há como retorna este dado por meio de outra função já definida, no entanto em outra prova feita para o presente trabalho, chamada `plus_S_r` tem-se:

```
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`; de acordo com a documentação Idris (e o que pode ser notado por seu tipo) `replace` recebe um igualdade entre dois valores `x` e `y` e um proposição sobre `x` e retorna a mesma proposição sobre `y`. 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 disto 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
```

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 construtor **`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
```

Assim está feita a prova pois não há nenhum hole em `plus_comm` e caso se carregue o programa não ocorrerá nenhum erro mesmo sem a existência de um hole na função, o que significa que o tipo de retorno condiz com o que foi definido na declaração de tipo da função (primeira linha).

4.2 Quantificadores

Na prova apresentada na sub-sessão anterior se provou que $\forall n, m \in \mathbb{N}, n + m = m + n$, ou seja, de maneira implícita fez-se uso de um quantificador $\forall$, o que é simples de se notar pois a função `plus_comm` aceita quaisquer argumentos `n` e `m` de tipo **`Nat`**, entretanto em alguns caso pode-se desejar provar alguma proposição que envolva o quantificador $\exists$; para isso existe 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 $\exists$ e por isso é utilizado em provas em Idris. A seguir tem-se um exemplo de declaração que equivale ao teorema:

$$\forall n \in \mathbb{N}, even(n) = true \Longrightarrow \exists m \in \mathbb{N}, n = 2 \cdot m \tag{1}$$

que é:

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

A equivalência com o quantificado $\exists$ 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`.

REFERENCES

[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