Introdução à linguagem Agda

Programação Funcional

Baseado nos slides do Prof. Rodrigo Ribeiro

Objetivos

Objetivos

- Descrever o processo de instalação da linguagem Agda.
- Sintaxe da linguagem e uso do “IDE” de Agda.

Objetivos

- Descrever sobre a hierarquia de tipos de Agda.
- Apresentar as restrições de totalidade da linguagem.

Instalação

Instalação

- A linguagem Agda é desenvolvida usando Haskell.
- A maneira mais prática de instalar a linguagem em sua máquina é usando o comando:

```
stack install Agda-2.6.2.1
```

Instalação

- Editores para programação em Agda: VSCode e Emacs
  - VSCode: extensão agda-mode
  - Emacs: Agda-mode

Instalação

- Depois de instalar seu editor favorito, crie o arquivo hello.agda com o seguinte conteúdo:

```
data Greeting : Set where
  hello : Greeting

greet : Greeting
greet = hello
```

A linguagem Agda

A linguagem Agda

- Sintaxe inspirada em Haskell
- Diferenças
  - Tipagem feita usando `x : A`
  - Uso de caracteres unicode `A → B`

A linguagem Agda

- Similar a Haskell, programas Agda consistem de tipos de dados e funções definidas por casamento de padrão.

A linguagem Agda

- Ao contrário de Haskell, não há imports automáticos de bibliotecas.
- Você pode carregar módulos da biblioteca padrão ou mesmo definir tudo **do zero**.

A linguagem Agda

- Definindo números naturais

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

{-# BUILTIN NATURAL ℕ #-}
```

A linguagem Agda

- Definindo a operação de adição

```
_+_ : ℕ → ℕ → ℕ
zero    + m = m
(suc n) + m = suc (n + m)
```

Programação interativa

Programação interativa

- Vamos considerar o tipo de dados de booleanos

```
data Bool : Set where
  true false : Bool
```

Programação interativa

- Desenvolvendo a negação.
  - Vamos usar o recurso de desenvolvimento interativo.

```
not : Bool → Bool
not x = {!!}
```

Programação interativa

- Carregando um arquivo: Ctrl-c + Ctrl-l
- Definição por casos: Ctrl-c + Ctrl-c
- Apresentar valor: Ctrl-c + space

Hierarquia de tipos

Hierarquia de tipos

- Em Agda, tipos possuem tipos.
- Exemplo: Bool possui tipo Set.
  - Set possui tipo Set_1
  - Set_1 possui tipo Set_2 e assim por diante...

Hierarquia de tipos

- Usando Set, podemos implementar funções polimórficas.

```
id₁ : (A : Set) → A → A
id₁ A x = x
```

Hierarquia de tipos

- Usando chaves podemos declarar argumentos implícitos
  - Implícitos: calculados pelo compilador

```
id₂ : {A : Set} → A → A
id₂ x = x
```

Hierarquia de tipos

- Definindo if como uma função polimórfica

```
if_then_else_ : {A : Set} → Bool → A → A → A
if true  then x else _ = x
if false then _ else y = y
```

Hierarquia de tipos

- Definindo tipos polimórficos

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A → List A → List A
```

Hierarquia de tipos

- Definindo registros
  - Definem projeções para cada campo.
  - Permitem a definição de construtores.

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B
```

Totalidade

Totalidade

- Ao contrário de Haskell, Agda exige que:
  - Todas as funções façam casamento de padrão exaustivo.
  - Toda função deve terminar, isto é, não é permitido que código entre em loop.

Totalidade

- Exemplo: Casamento de padrão não exaustivo

```
f : Bool → Bool
f false = true
```

Totalidade

- Exemplo: Não terminação

```
f : Bool → Bool
f x = f x
```

Totalidade

- Essas restrições são necessárias para garantir a consistência lógica de Agda.

Totalidade

- Agda pode ser utilizada para provar resultados da matemática ou correção de software.

Exercícios

Exercícios

- Desenvolva as seguintes funções em Agda:

```
-- número de elementos em uma lista
length : {A : Set} → List A → ℕ
-- concatenação
_++_ : {A : Set} → List A → List A → List A
-- map
map : {A B : Set} → (A → B) → List A → List B
```

Tipos Dependentes

Tipos Dependentes

- Tipo dependente: tipo que referem-se a partes de um programa.
- Qual a utilidade deste conceito?

Tipos Dependentes

- Problema comum em programa: acesso a posições inválidas em arranjos.

```
public class NewClass2 {
    public static void main(String[] args)
    {
        int ar[] = { 1, 2, 3, 4, 5 };
        for (int i = 0; i <= ar.length; i++)
            System.out.println(ar[i]);
    }
}
```

Tipos Dependentes

- Seria possível um compilador capturar esses erros?
- Sim, se o tipo possuir informação sobre o tamanho de uma lista / arranjo.

Tipos Dependentes

- Exemplo: Vectors - listas indexadas por tamanho.

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A 0
  _∷_ : {n : ℕ} → A → Vec A n → Vec A (suc n)

infixr 5 _∷_
```

Tipos Dependentes

- Exemplos

```
ex1 : Vec ℕ 1
ex1 = 0 ∷ []

ex2 : Vec ℕ 3
ex2 = 1 ∷ 2 ∷ ex1
```

Tipos Dependentes

- Propriedade da concatenação de listas:

```
length (xs ++ ys) = length xs + length ys
```

Tipos Dependentes

- Usando tipos dependentes podemos garantir que essa propriedade seja atendida pela concatenação.

```
_++V_ : {A : Set}{n m : ℕ}
    → Vec A n
    → Vec A m
    → Vec A (n + m)
[]        ++V ys = ys
(x :: xs) ++V ys = x :: (xs ++V ys)
```

Tipos Dependentes

- Outro exemplo: implementar uma função para recuperar a cabeça de uma lista.
  - Em Haskell, usamos uma função parcial...

```
head : [a] -> a
head [] = error "Empty list!"
head (x : xs) = x
```

Tipos Dependentes

- Em Agda, podemos definir head de forma que seja aplicada apenas a listas não vazias.

```
head-vec : {A : Set}{n : ℕ} → Vec A (suc n) → A
head-vec (x :: _) = x
```

Tipos Dependentes

- Usando tipos dependentes, conseguimos resolver alguns problemas.
  - Concatenação correta por construção.
  - Definição de head para listas não vazias.

Tipos Dependentes

- Porém, como resolver o problema de acesso a posições inválidas?

Tipos Dependentes

- Para isso, devemos restringir os valores de possíveis posições ao tamanho da lista.

Tipos Dependentes

- Representando posições utilizando conjuntos finitos.

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Tipos Dependentes

- Usando o tipo Fin, podemos definir a função para acessar o elemento em uma posição representada por um valor do tipo Fin.

```
lookup-vec : {A : Set}{n : ℕ} → Vec A n → Fin n → A
lookup-vec (x :: _)  zero      = x
lookup-vec (_ :: xs) (suc idx) = lookup-vec xs idx
```

Propositions as Types

Propositions as Types

- Em Agda, podemos representar fórmulas da lógica como tipos da linguagem.
- Programas possuindo esses tipos consistem de provas destas fórmulas.

Propositions as Types

- Conjunção
  - Dizemos que $A \wedge B$ é verdadeiro se temos deduções de A e de B
  - Logo, representamos a conjunção por um par de deduções.

Propositions as Types

- Representando a conjunção

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B
```

Propositions as Types

- Implicações são representadas como tipos funcionais.
- Deduções de implicações são funções!

Propositions as Types

- Exemplo: $(B \to C) \to (A \to B) \to (A \to C)$

```
_∘_ : {A B C : Set} → (B → C) → (A → B) → (A → C)
f ∘ g = λ pA → f (g pA)
```

Propositions as Types

- Exemplo: provando que $A \wedge B \rightarrow B \wedge A$

```
and-comm : {A B : Set} → A × B → B × A
and-comm (pA , pB) = pB , pA
```

Propositions as Types

- Exemplo: associatividade do $\wedge$

```
and-assoc : {A B C : Set} → A × (B × C) → (A × B) × C
and-assoc (pA , (pB , pC)) = ((pA , pB) , pC)
```

Propositions as Types

- Disjunção
  - Dizemos que $A \vee B$ é verdadeiro se temos uma prova de A ou de B.

Propositions as Types

- Representando a disjunção

```
data _⊎_ (A B : Set) : Set where
  left  : A → A ⊎ B
  right : B → A ⊎ B
```

Propositions as Types

- Representando a eliminação do ∨

```
⊎-elim : {A B C : Set} → A ⊎ B → (A → C) → (B → C) → C
⊎-elim (left pA)  f _ = f pA
⊎-elim (right pB) _ g = g pB
```

Propositions as Types

- Exemplo: A ⊎ B → B ⊎ A

```
⊎-comm : {A B C : Set} → A ⊎ B → B ⊎ A
⊎-comm (left pA)  = right pA
⊎-comm (right pB) = left pB
```

Propositions as Types

- A constante verdadeiro é representada por uma tipo com único construtor.

```
data ⊤ : Set where
  tt : ⊤
```

Propositions as Types

- A constante falso é representada por um tipo sem construtores.
  - Impossível de construir uma dedução diretamente.

```
data ⊥ : Set where
```

Propositions as Types

- A partir de $\perp$, podemos deduzir qualquer proposição

```
⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

Propositions as Types

- Negação é representada em termos da implicação.

```
¬_ : Set → Set
¬ A = A → ⊥
```

Lógica de Predicados

Lógica de Predicados

- Podemos representar predicados usando tipos indutivos.

```
data Even : ℕ → Set where
  zero : Even 0
  2+_  : {n : ℕ} → Even n → Even (2 + n)
```

Lógica de Predicados

- Exemplo: demonstrando Even 8

```
8-Even : Even 8
8-Even = 2+ (2+ (2+ (2+ zero)))
```

Lógica de Predicados

- Exemplo: demonstrando que não é provavel que Even 5.

```
5-Even : ¬ Even 5
5-Even (2+ (2+ ()))
```

Lógica de Predicados

- Quantificador universal
  - Para demonstrar $\forall$ x. P(x) devemos deduzir P(v) para cada valor v.
  - Podemos fazer isso usando uma função λ v $\rightarrow$ p, em que p é uma dedução de P(v).

Lógica de Predicados

- Exemplo:

```
double : ℕ → ℕ
double zero    = zero
double (suc n) = suc (suc (double n))

doubleEven : ∀ (n : ℕ) → Even (double n)
doubleEven zero    = zero
doubleEven (suc n) = 2+ doubleEven n
```

Lógica de Predicados

- Quantificador existencial
  - Para provar $\exists$ x. P(x) precisamos de um valor v e da dedução de P(v).
  - A demonstração de um existencial consiste de um par, chamado de produto dependente.

Lógica de Predicados

- Definição de produto dependente.

```
record Σ (A : Set)(B : A → Set) : Set where
  constructor _,_
  field
    witness : A
    proof   : B witness
open Σ
```

Lógica de Predicados

- Representando o quantificador existencial.

```
∃_ : {A : Set} → (A → Set) → Set
∃_ {A} p = Σ A p
```

Lógica de Predicados

- Exemplo: Se Even n é válido então existe m tal que n $=$ 2 * m.
- Como formalizar esse resultado?

Lógica de Predicados

- Para isso, vamos precisar de um teste de igualdade para números.

```
_=N_ : ℕ → ℕ → Bool
zero  =N zero  = true
zero  =N suc m = false
suc n =N zero  = false
suc n =N suc m = n =N m
```

Lógica de Predicados

- Exemplo: predicado para garantir que um booleano é verdadeiro.

```
data IsTrue : Bool → Set where
  is-true : IsTrue true
```

Lógica de Predicados

- Exemplo:

```
_ : IsTrue (L.length (1 L.:: 2 L.:: 3 L.:: L.[]) =N 3)
_ = is-true
```

Lógica de Predicados

- Apesar de funcionar, o uso do predicado IsTrue e da função é inconveniente.
- Há uma representação melhor da igualdade?

Lógica de Predicados

- Igualdade proposicional

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x

infix 4 _≡_
```

Lógica de Predicados

- Exemplo:

```
easy : 1 + 1 ≡ 2
easy = refl

obvious : ¬ (1 ≡ 2)
obvious ()
```

Lógica de Predicados

- Propriedades da igualdade: simetria

```
sym : {A : Set}{x y : A} → x ≡ y → y ≡ x
sym refl = refl
```

Lógica de Predicados

- Propriedades da igualdade: transitividade

```
trans : {A : Set}{x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

Lógica de Predicados

- Propriedades da igualdade: congruência

```
cong : {A B : Set}{x y : A}(f : A → B)
    → x ≡ y
    → f x ≡ f y
cong f refl = refl
```

Exercícios

Exercícios

- Implemente funções Agda que provam as seguintes tautologias da lógica.
  - $(A \to B \to C) \to ((A \wedge B) \to C)$
  - $(A \wedge (B \vee C)) \to (A \wedge B) \vee (A \wedge C)$