Programação com Tipos Dependentes em Agda

Rodrigo Ribeiro

Departamento de Computação e Sistemas — DECSI
Universidade de Federal de Ouro Preto

March 18, 2016

Motivação — (I)

- Sistemas de tipos
  - Técnica de verificação de programas mais utilizada.
  - Aplicações em segurança e concorrência.
- Porém, como garantir a correção de um programa?

Motivação — (II)

- Porém, como garantir a correção de um programa?
    - Geralmente, difícil...
    - Envolve demonstrações formais ou uso de técnicas como model checking.

Motivação — (III)

- Verificação de programas — Situação Ideal:
  - Verificação automática.
  - Idealmente, deveríamos ser capazes de especificar e programar em uma mesma linguagem.

Motivação — (IV)

- Especificar e programar em uma mesma linguagem?
  - Evita problemas de consistência entre diferentes linguagens.
- Porém, existe esta linguagem?

Motivação — (V)

- A linguagem Agda
  - Linguagem Funcional!
  - Sistema de tipos expressivo, capaz de representar especificações.
- Se tipos em Agda representam especificações, então...
  - O processo de verificação é feito pelo próprio compilador de Agda!

A Linguagem Agda — (I)

- Em Agda, não existem tipos "built-in". Tudo é definido na própria linguagem.
- Tipos em Agda são uma generalização de tipos de dados algébricos encontrados em Haskell e ML.

A Linguagem Agda — (II)

- Exemplo:

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

- Agda permite o uso de caracteres Unicode!

A Linguagem Agda — (III)

- Exemplo:

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

- A anotação `Set` especifica que $\mathbb{N}$ é um tipo.
  - O termo `Set` possui tipo $\texttt{Set}_1$.
  - Em Agda, temos uma hierarquia de tipos tal que: $\texttt{Set} : \texttt{Set}_1 : \texttt{Set}_2 \ldots$

A Linguagem Agda — (IV)

- Exemplo:

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

- O tipo $\mathbb{N}$ possui dois construtores:
  - `zero` que representa a constante 0
  - `suc` que representa a função de sucessor.

A Linguagem Agda — (V)

- Usando $\mathbb{N}$ podemos representar os números naturais como:

$$
\begin{array}{lcl}
\mathsf{zero} & \equiv & 0 \\
\mathsf{suc\ zero} & \equiv & 1 \\
\mathsf{suc\ (suc\ zero)} & \equiv & 2 \\
\ldots & \ldots & \ldots
\end{array}
$$

A Linguagem Agda — (VI)

- Adição em Agda:

$$
\begin{array}{lll}
\_+\_ & : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & \\
\mathsf{zero} & + \; n = n & \\
\mathsf{suc}\ m & + \; n = \mathsf{suc}\ (m + n) &
\end{array}
$$

A Linguagem Agda — (VII)

- Listas

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

A Linguagem Agda — (VIII)

- Funções sobre listas: tamanho

$$
\begin{array}{l}
\mathsf{length} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathbb{N} \\
\mathsf{length}\ []\ \quad = 0 \\
\mathsf{length}\ (x :: xs) = \mathsf{suc}\ (\mathsf{length}\ xs)
\end{array}
$$

A Linguagem Agda — (IX)

- Funções sobre listas: concatenação

$$
\begin{array}{l}
\_{+}{+}\_ : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A \\
[]\ {+}{+}\ ys \quad = ys \\
(x :: xs)\ {+}{+}\ ys = x :: (xs\ {+}{+}\ ys)
\end{array}
$$

A Linguagem Agda — (X)

- Podemos conjecturar que:

$$\forall\, xs\; ys . length (xs ++ ys) = length\; xs + length\; ys$$

- Isto é, o tamanho da concatenação de duas listas é a soma de seus tamanhos.

A Linguagem Agda — (XI)

- $\forall\, xs\; ys . length(xs + +ys) = length\; xs + length\; ys$
  - Teorema facilmente provado por indução sobre a estrutura de *xs*.
  - Apesar de possível provar esse fato em Agda, vamos usar o sistema de tipos de Agda para garantir que a concatenação possua essa propriedade.
- Para isso, vamos utilizar **tipos dependentes**.

Tipos Dependentes — (I)

- Dizemos que um certo tipo é dependente se este depende de um valor.
- Exemplo — Listas indexadas por seu tamanho:

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

Tipos Dependentes — (II)

- O tipo `Vec A n` é uma **família de tipos indexada** por números naturais.
  - Isto é, para cada valor `n :` $\mathbb{N}$, temos um tipo `Vec A n`
  - Note que o tipo `Vec A n` especifica o número de elementos presentes em uma lista!

Tipos Dependentes — (III)

- Mas, qual a vantagem disso?
  - Tipos mais precisos, permitem programas corretos!

Tipos Dependentes — (IV)

- Exemplo: obtendo a cabeça de uma lista.
  - Problema: E se a lista for vazia, o que devemos retornar?

Tipos Dependentes — (V)

- Primeiro, usando listas...
- O tipo `Maybe A` indica a possibilidade de retorno de um valor.

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just    : A → Maybe A
```

Tipos Dependentes — (VI)

- Usando o tipo `Maybe A`:

$$
\begin{aligned}
&\mathsf{hd} : \forall\ \{A\} \to \mathsf{List}\ A \to \mathsf{Maybe}\ A\\
&\mathsf{hd}\ []\ \quad = \mathsf{nothing}\\
&\mathsf{hd}\ (x :: xs) = \mathsf{just}\ x
\end{aligned}
$$

- Problema desta solução: Uso do tipo `Maybe A`.

Tipos Dependentes — (VII)

- Usando o tipo `Vec A n`, podemos expressar o tipo de uma lista **não vazia** por `Vec A (suc n)`.
- O compilador garante que jamais aplicaremos a função `head` a uma lista vazia!

$$\mathsf{head} : \forall \{A : \mathsf{Set}\}\{n : \mathbb{N}\} \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to A$$
$$\mathsf{head}\ (x :: xs) = x$$

Tipos Dependentes — (VIII)

- Voltando a função de concatenação...

$$
\begin{array}{l}
\_{+}{+}v\_ : \forall \{n\ m\ A\} \rightarrow \mathsf{Vec}\ A\ n \rightarrow \mathsf{Vec}\ A\ m \rightarrow \mathsf{Vec}\ A\ (n + m) \\
[]\ {+}{+}v\ ys = ys \\
(x :: xs)\ {+}{+}v\ ys = x :: (xs\ {+}{+}v\ ys)
\end{array}
$$

- Agora, a propriedade sobre o tamanho da concatenção de duas listas é verificada automaticamente pelo compilador.
- Código idêntico a concatenação de listas.

Tipos Dependentes — (IX)

- Mais um exemplo: obtendo o *n*-ésimo elemento de uma lista.

$$
\begin{array}{l}
\mathsf{lookupWeak} : \{A : \mathsf{Set}\} \to \mathbb{N} \to \mathsf{List}\ A \to \mathsf{Maybe}\ A \\
\mathsf{lookupWeak}\ n \quad [\,] \qquad\quad = \mathsf{nothing} \\
\mathsf{lookupWeak}\ 0 \quad (x :: \_) \quad = \mathsf{just}\ x \\
\mathsf{lookupWeak}\ (\mathsf{suc}\ n)\ (\_ :: xs) = \mathsf{lookupWeak}\ n\ xs
\end{array}
$$

Tipos Dependentes — (X)

- Como desenvolver uma função correta por construção para esta tarefa?
  - Primeiro, o que quer dizer "ser *n*-ésimo elemento" de uma lista?

Tipos Dependentes — (XI)

- Formalizando a noção de um elemento pertencer a uma lista:

```
data _∈_ {A : Set} : A → List A → Set where
  here  : ∀ {x xs} → x ∈ x :: xs
  there : ∀ {x y xs} → y ∈ xs → y ∈ (x :: xs)
```

Tipos Dependentes — (XII)

- Recuperando o índice de um elemento a partir de uma prova de pertinência:

$$\begin{aligned}
&\mathsf{index} : \forall\ \{A : \mathsf{Set}\}\{x : A\}\{xs : \mathsf{List}\ A\} \to x \in xs \to \mathbb{N}\\
&\mathsf{index}\ \mathsf{here}\quad\ \ = \mathsf{zero}\\
&\mathsf{index}\ (\mathsf{there}\ n) = \mathsf{suc}\ (\mathsf{index}\ n)
\end{aligned}$$

Tipos Dependentes — (XIII)

- Especificando a função para obter o *n*-ésimo elemento de uma lista:
  - Se *n* for uma posição de um elemento, então este possuirá uma prova de pertinência.
  - Se $n \geq$ `length xs`, então não existe esse elemento.

Tipos Dependentes — (XIV)

- Um tipo preciso para a função para obter o *n*-ésimo elemento de uma lista:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Lookup}\ \{A\}(xs : \mathsf{List}\ A) : \mathbb{N} \to \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{inside}\ \ : \forall\ x\ (p : x \in xs) \to \mathsf{Lookup}\ xs\ (\mathsf{index}\ p) \\
\quad \mathsf{outside} : \forall\ m \to \mathsf{Lookup}\ xs\ (\mathsf{length}\ xs + m)
\end{array}
$$

Tipos Dependentes — (XV)

- Função correta por construção:

$$
\begin{array}{l}
\text{lookup} : \{A : \text{Set}\}(xs : \text{List}\ A)(n : \mathbb{N}) \rightarrow \text{Lookup}\ xs\ n \\
\text{lookup}\ []\ n = \text{outside}\ n \\
\text{lookup}\ (x :: xs)\ \text{zero} = \text{inside}\ x\ \text{here} \\
\text{lookup}\ (x :: xs)\ (\text{suc}\ n)\ \text{with lookup}\ xs\ n \\
\text{lookup}\ (x :: xs)\ (\text{suc}\ .(\text{index}\ p))\quad \mid \text{inside}\ y\ p = \text{inside}\ y\ (\text{there}\ p) \\
\text{lookup}\ (x :: xs)\ (\text{suc}\ .(\text{length}\ xs + m)) \mid \text{outside}\ m = \text{outside}\ m
\end{array}
$$

Tipos Dependentes — (XVI)

- Casamento de padrão com tipos dependentes:
  - Se `lookup xs n = outside m`, então $n \geq \mathit{length}\ \mathit{xs}$
  - Se `lookup xs n = inside p`, então p é uma prova de que x pertence a lista xs e n é o índice obtido a partir desta prova!

Verificando Tipos — (I)

- Último e derradeiro exemplo: Algoritmo para verificar tipos correto por construção.
- O tipo do algoritmo "explica" o porquê um termo é correto.
- λ-cálculo
  - Linguagem funcional "minimalista".

Verificando Tipos — (II)

- λ-cálculo: Sintaxe de tipos

```
data Ty : Set where
  ι   : Ty
  _⇒_ : Ty → Ty → Ty

Ctx : Set
Ctx = List Ty
```

Verificando Tipos — (III)

- λ-cálculo: Sintaxe de termos

```
data Exp : Set where
  val : Exp
  var : ℕ → Exp
  abs : Ty → Exp → Exp
  app : Exp → Exp → Exp
```

Verificando Tipos — (IV)

- Sistema de tipos

$$
\begin{array}{l}
\mathsf{data}\ \_\vdash\_\ (G : \mathsf{Ctx}) : \mathsf{Ty} \to \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{tval} : G \vdash \iota \\
\quad \mathsf{tvar} : \forall\ \{t\}\ (p : t \in G) \to G \vdash t \\
\quad \mathsf{tabs} : \forall\ t\ \{t'\} \to G\ ,\ t \vdash t' \to G \vdash t \Rightarrow t' \\
\quad \mathsf{tapp} : \forall\ \{t\ t'\} \to G \vdash (t \Rightarrow t') \to G \vdash t \to G \vdash t'
\end{array}
$$

Verificando Tipos — (V)

- Obtendo uma expressão a partir de sua derivação de tipos — type erasure

$$
\begin{array}{l}
\text{erase} : \forall\ \{G\ t\} \to G \vdash t \to \text{Exp} \\
\text{erase tval} = \text{val} \\
\text{erase } (\text{tvar } p) = \text{var } (\text{index } p) \\
\text{erase } (\text{tabs } t\ p) = \text{abs } t\ (\text{erase } p) \\
\text{erase } (\text{tapp } p\ p') = \text{app } (\text{erase } p)\ (\text{erase } p')
\end{array}
$$

Verificando Tipos — (VI)

- Comparando dois tipos com respeito a igualdade

```
data TyEq : Ty → Ty → Set where
  eq  : ∀ {t} → TyEq t t
  neq : ∀ {t t'} → TyEq t t'
```

Verificando Tipos — (VII)

- Decidindo a igualdade entre tipos

$$
\begin{array}{l}
\_==\_ : (t\ t' : \mathsf{Ty}) \rightarrow \mathsf{TyEq}\ t\ t' \\
\iota == \iota = \mathsf{eq} \\
\iota == (t' \Rightarrow t'') = \mathsf{neq} \\
(t \Rightarrow t_1) == \iota = \mathsf{neq} \\
(t \Rightarrow t_1) == (t' \Rightarrow t'')\ \mathsf{with}\ t == t' \mid t_1 == t'' \\
(.t' \Rightarrow .t'') == (t' \Rightarrow t'') \mid \mathsf{eq} \mid \mathsf{eq} = \mathsf{eq} \\
(t \Rightarrow t_1) == (t' \Rightarrow t'') \mid \_ \mid \_ = \mathsf{neq}
\end{array}
$$

Verificando Tipos — (VIII)

- Especificando o comportamento do algoritmo

$$
\begin{array}{l}
\textsf{data TypeCheck } (G : \textsf{Ctx}) : \textsf{Exp} \rightarrow \textsf{Set where} \\
\quad \textsf{ok} : \forall \{t\}(d : G \vdash t) \rightarrow \textsf{TypeCheck } G \ (\textsf{erase } d) \\
\quad \textsf{bad} : \forall \{e\} \rightarrow \textsf{TypeCheck } G \ e
\end{array}
$$

Verificando Tipos — (IX)

- O algoritmo de verificação — parte 1:

$$
\begin{array}{l}
\mathsf{tc} : \forall\ G\ (e : \mathsf{Exp}) \rightarrow \mathsf{TypeCheck}\ G\ e \\
\mathsf{tc}\ G\ \mathsf{val} = \mathsf{ok}\ \mathsf{tval} \\
\mathsf{tc}\ G\ (\mathsf{var}\ x)\ \mathsf{with}\ \mathsf{lookup}\ G\ x \\
\mathsf{tc}\ G\ (\mathsf{var}\ .(\mathsf{index}\ p)) \quad \mid \mathsf{inside}\ x\ p = \mathsf{ok}\ (\mathsf{tvar}\ p) \\
\mathsf{tc}\ G\ (\mathsf{var}\ .(\mathsf{length}\ G + m)) \mid \mathsf{outside}\ m = \mathsf{bad}
\end{array}
$$

Verificando Tipos — (X)

- O algoritmo de verificação — parte 2:

```
tc G (abs t e) with tc (G , t) e
tc G (abs t .(erase d)) | ok d = ok (tabs t d)
tc G (abs t e)    | bad = bad
```

Verificando Tipos — (XI)

- O algoritmo de verificação — parte 3:

```
tc G (app e e') with tc G e | tc G e'
tc G (app ._  ._ ) | ok {t ⇒  _} d | ok {t'} d₁ with t == t'
tc G (app ._  ._)    | ok {t ⇒ z} d | ok d₁  | eq = ok (tapp d d₁)
tc G (app ._  ._)    | ok {t ⇒ z} d | ok d₁  | neq = bad
tc G (app e e')      | _     | _ = bad
```

Conclusão

- Moral da história:
    - Tipos dependentes permitem a especificação precisa do relacionamento entre parâmetros e resultados de funções.
    - Verificados automaticamente pelo compilador!
- Isso é apenas a ponta do Iceberg... Não falamos sobre:
    - Provas em Agda, Igualdade, Terminação...

Código

- Disponível no github:

  https://github.com/rodrigogribeiro/UDESC-talk-04-2014
- Código Agda dos exemplos e slides.