# Implementação de Verificação de Tipos em Haskell

Rodrigo Bonifácio

26 de junho de 2017

## 1 Introdução

Esse documento apresenta uma implementação, em *literate Haskell*, do mecanismo de verificação de tipos de uma linguagem de programação funcional minimalista. Os alunos da disciplina Linguagens de Programação devem extender essa implementação de tal forma que todos os elementos sintáticos possuam a verificação de tipos implementada.

## 2 Visão geral da linguagem

A linguagem LFCF suporta tanto expressões identificadas (LET) quanto identificadores e funções de alta ordem (com o mecanismo de expressoes lambda). O foco é na verificação de tipos, então não estão implementadas funções voltadas para a avaliação das expressões.

## 3 Definição da Árvore Sintática Abstrata

A implementação consiste na definição de um módulo Haskell mais alguns tipos auxiliares, como `Id` (um tipo sinônimo para uma `string`) e `Gamma`, que corresponde a um mapeamento de identificadores em tipos.

```
module LFCFDTypes where
type Id = String
```

```
type Gamma = [(Id, Tipo)]
```

Os tipos válidos são definidos com o tipo algébrico `Tipo`, que pode ser um tipo inteiro, um tipo booleano e um tipo função. O tipo função deve expressar tanto o tipo do argumento quanto o tipo do retorno. As expressões, conforme mencionado anteriormente, envolvem tanto valores inteiros quanto booleanos, bem como expressões binárias (soma, subtração, etc.), expressões `let`, `lambda`, aplicação de funções e `if-then-else`

```
data Tipo = TInt | TBool | TFuncao Tipo Tipo
deriving (Show, Eq)
data Expressao = ValorI Int
    | ValorB Bool
    | Soma Expressao Expressao
    | Subtracao Expressao Expressao
    | Multiplicacao Expressao Expressao
    | Divisao Expressao Expressao
    | Let Id Expressao Expressao
    | Ref Id
    | Lambda (Id, Tipo) Tipo Expressao
    | Aplicacao Expressao Expressao
    | If Expressao Expressao Expressao
deriving (Show, Eq)
```

A função que realiza a verificação de tipos recebe uma expressão, um ambiente `Gamma` e possivelmente retorna um tipo válido (por isso o retorno `Maybe Tipo`). Caso algum erro ocorra no sistema de tipos, essa função deve retornar `Nothing`. Isso permite o uso de uma notação baseada em monadas.

```
verificarTipos :: Expressao → Gamma → Maybe Tipo
```

Para alguns casos, a verificação de tipos é bem trivial, particularmente a verificação de tipos de expressões envolvendo valores inteiros, valores booleanos e expressões `lambda`

```
verificarTipos (ValorI n) _ = return TInt
verificarTipos (ValorB b) _ = return TBool
verificarTipos (Lambda (v, t1) t2 exp) g = return (TFuncao t1 t2)
```

Para outros casos, a verificação de tipos requer um certo grau de indução (seguindo as regras de derivação vistas em sala de aula). Para a soma, temos a seguinte regra de derivação:

$$\frac{\Gamma \vdash lhs : TInt \qquad \Gamma \vdash rhs : TInt}{\Gamma \vdash soma(lhs, rhs) : TInt}$$

que pode ser traduzida para Haskell como:

```
verificarTipos (Soma l r) gamma =
  verificarTipos l gamma >>= λlt →
  verificarTipos r gamma >>= λrt →
  if lt ≡ TInt ∧ rt ≡ TInt then return TInt else Nothing
```

Similarmente, a verificação de expressões do tipo `let` requer um grau de indução. Supondo uma expressão `let v = e in c`, primeiro verificamos o tipo da expressão nomeda (`e`) é bem tipada com tipo `t`, adicionamos uma associação `(v, t)` no ambiente `Gamma` original e computamos o tipo de `c` no novo ambiente. Em termos de regras de derivação, terámos:

$$\frac{\Gamma \vdash e : \tau_1 \qquad (x, \tau_1)\Gamma \vdash c : \tau_2}{\Gamma \vdash let(v, e, c) : \tau_2}$$

Em Haskell:

```
verificarTipos (Let v e c) gamma =
  verificarTipos e gamma >>= λt →
  verificarTipos c ((v, t) : gamma)
```

# 4 Trabalho

Essa atividade do projeto final envolve verificar os tipos das demais expressoes e escrever casos de teste para verificar se esta tudo ok. Para simplificar, escolha a estrategia de escopo (dinamico ou estatico).