Implementação de Verificação de Tipos em Haskell

Luisa Sinzker Fantin, 14/0151893
João Pedro Silva Sousa, 15/0038381
Rafael Oliveira de Souza, 15/0081537

6 de julho 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.

O objetivo da implementação de tipos consiste em identificar erros a nível sintático (não erros de sintaxe) na linguagem, utilizando o interpretador. Algumas computações inválidas podem ser identificadas antes mesmo de executá-las, como por exemplo uma divisão por zero.

Caso tentássemos avaliar uma divisão por zero no interpretador, seria obtida a saía *infinity*, resultado de uma computação realizada pelo interpretador da linguagem Haskell, não do interpretador construído. Ou seja, se esperássemos um erro e o GHCi não o identificasse, ou o GHCi marcasse um erro que não estamos esperando, o interpretador não seria confiável [1].

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 o tipo verificado pelo interpretador seja um tipo `a` válido, é retornado um `Just a`, 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`.

$$verificarTipos\ (ValorI\ n)\ \_ = return\ TInt$$

$$verificarTipos\ (ValorB\ b)\ \_ = return\ TBool$$

$$verificarTipos\ (Lambda\ (v, t1)\ t2\ exp)\ g = return\ (TFuncao\ t1\ t2)$$

Os casos mostrados são a base dos tipos da linguagem, pois todas as construções devem ser reduzidas a um dos casos triviais.

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:

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

que pode ser traduzida para Haskell como:

$$
\begin{array}{l}
verificarTipos\ (Soma\ l\ r)\ gamma = \\
\quad verificarTipos\ l\ gamma \gg\!= \lambda lt \rightarrow \\
\quad verificarTipos\ r\ gamma \gg\!= \lambda rt \rightarrow \\
\quad \textbf{if}\ lt \equiv TInt \wedge rt \equiv TInt \\
\quad\quad \textbf{then}\ return\ TInt \\
\quad\quad \textbf{else}\ Nothing
\end{array}
$$

Analogamente, as operações de subtração, multiplicação e divisão possuem árvores de derivação parecidas:

- Subtração:

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

- Multiplicação:

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

Apenas uma mudança na verificação de tipos da divisão: caso seja identificado um denominador nulo (igual a zero) na divisão, retorna-se `Nothing`, caso contrário, avalia-se o tipo das expressões algébricas normalmente.

$$\frac{\Gamma \vdash lhs : \texttt{TInt} \qquad \Gamma \vdash rhs : \texttt{TInt} \qquad rhs \rightarrow \neg(ValorI0)}{\Gamma \vdash divisao(lhs, rhs) : \texttt{TInt}}$$

Em Haskell:

$verificarTipos\ (Divisao\ l\ r)\ gamma =$
$\quad \textbf{if}\ r \equiv (ValorI\ 0)$
$\quad\quad \textbf{then}\ Nothing$
$\quad\quad \textbf{else}\ verificarTipos\ l\ gamma \ggg\!= \lambda lt \rightarrow$
$\quad\quad\quad verificarTipos\ r\ gamma \ggg\!= \lambda rt \rightarrow$
$\quad\quad\quad\quad \textbf{if}\ lt \equiv TInt \land rt \equiv TInt$
$\quad\quad\quad\quad\quad \textbf{then}\ return\ TInt$
$\quad\quad\quad\quad\quad \textbf{else}\ Nothing$

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 nomeada (`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, teremos:

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

Em Haskell:

$verificarTipos\ (Let\ v\ e\ c)\ gamma =$
$\quad verificarTipos\ e\ gamma \ggg\!= \lambda lt \rightarrow$
$\quad verificarTipos\ c\ gamma' \ggg\!= \lambda rt \rightarrow$
$\quad\quad \textbf{if}\ lt \equiv rt$
$\quad\quad\quad \textbf{then}\ return\ rt$
$\quad\quad\quad \textbf{else}\ Nothing$
$\quad \textbf{where}$
$\quad\quad gamma' = incrementaAmb\ v\ (verificarTipos\ e\ gamma)\ gamma$

A função `incrementaAmb` enriquece o ambiente na mudança de escopo das expressões `Let`.

A verificação do tipo de uma referência é bastante simples, requer apenas pesquisar o identificador no ambiente de mapeamento Γ, para isso, utilizamos a função pesquisar:

$verificarTipos\ (Ref\ v)\ gamma = pesquisar\ v\ gamma$

Caso o identificador não seja encontrado no ambiente de mapeamento, é retornado um erro.

Uma expressão `If Expressao Expressao Expressao` é formada por três partes: a condição de avaliação; uma cláusula `then` e uma cláusula `else`. A cláusula `then` é executada se a expressão de condição for avaliada verdadeira e, a cláusula `else`, caso contrário. A determinação do tipo de uma expressão `If` é um processo em duas etapas:

1. Verificar se o tipo da expressão de condição é um booleano (`TBool`);
2. Verificar se os tipos das expressões associadas às cláusulas `then` e `else` são iguais.

O interpretador será como apresentado:

$$
\begin{array}{l}
\textit{verificarTipos}\ (\textit{If test pass fail})\ \textit{gamma} = \\
\quad \textit{verificarTipos test gamma} \gg\!= \lambda t \rightarrow \\
\quad \textbf{if}\ t \equiv \textit{TBool} \\
\quad\quad \textbf{then}\ \textit{verificarTipos pass gamma} \gg\!= \lambda p \rightarrow \\
\quad\quad\quad \textit{verificarTipos fail gamma} \gg\!= \lambda f \rightarrow \\
\quad\quad\quad \textbf{if}\ p \equiv f \\
\quad\quad\quad\quad \textbf{then}\ \textit{return}\ p \\
\quad\quad\quad\quad \textbf{else}\ \textit{Nothing} \\
\quad\quad \textbf{else}\ \textit{Nothing}
\end{array}
$$

Conclui-se então que, a regra para definir uma expressão `If` é:

$$
\frac{\Gamma \vdash \textit{cond} : \texttt{boolean} \qquad \Gamma \vdash \textit{then} : \tau \qquad \Gamma \vdash \textit{else} : \tau}{\Gamma \vdash \{\texttt{if}\ \textit{cond then else}\} : \tau}
$$

A necessidade das cláusulas `then` e `else` possuirem o mesmo tipo (no caso da linguagem pode ser inteiro ou booleano) permite que possamos atribuir um tipo específico para expressões `If` e que não dependem de uma avaliação em tempo de execução.

Para definir o tipo de uma aplicação de função, serão necessárias mais verificações. Uma aplicação é definida como `Aplicacao Expressao Expressao`, na qual a primeira expressão é a definição da função e a segunda representa o argumento. O processo de verificaçã para uma `Aplicacao def arg` é o seguinte:

1. Verificar se a expressão que define a função (`def`) é uma expressão *lambda*; caso seja, ela automaticamente possui o tipo `TFuncao tId tExp`;

2. Verificar o tipo do argumento passado para a aplicação (`arg`) no contexto Γ (denominado `tArg`);

3. Comparar `tArg` com `tId`:

   - Se forem iguais, então o parâmetro passado na aplicação pode ser associado ao identificador da expressão *lambda*;
   - Se forem diferentes, então o tipo do parâmetro difere do tipo do argumento e pode ser retornado `Nothing`;

4. O tipo da aplicação será o tipo da expressão associada na expressão *lambda* verificado no ambiente Γ' composto pela tupla `(id, tipo_do_id)`;

Em Haskell:

```
verificarTipos (Aplicacao def arg) gamma =
  case def of
    (Lambda (v, tId) tExp exp) ->
      verificarTipos arg gamma >>= \a ->
        if a == tId
          then verificarTipos exp gamma'
          else Nothing
        where
          gamma' = [(v, tId)]
    otherwise -> error ("Aplicacao de funcao nao anonima")
```

Conclui-se então que a regra que define uma aplicação de função pode ser representada pela árvore a seguir:

$$\frac{\Gamma \vdash def : (\texttt{TFuncao}\ \tau_1\ \tau_2) \qquad \Gamma \vdash arg : \tau_1}{\Gamma \vdash \{\texttt{Aplicacao}\ def\ arg\} : \tau_2}$$

```
incrementaAmb :: Id -> Maybe Tipo -> Gamma -> Gamma
incrementaAmb n Nothing [] = []
incrementaAmb n Nothing ((i, e) : xs) = ((i, e) : xs)
incrementaAmb n (Just v) [] = [(n, v)]
incrementaAmb n (Just v) ((i, e) : xs)
  | n == i = incrementaAmb n (return v) []
  | otherwise = incrementaAmb n (return v) xs
pesquisar :: Id -> Gamma -> Maybe Tipo
```

$pesquisar\ v\ [\,] = error$ `"Variavel nao declarada."`
$pesquisar\ v\ ((i, e) : xs)$
$\quad |\ v \equiv i = return\ e$
$\quad |\ otherwise = pesquisar\ v\ xs$

Referências

[1] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Brown University, 2006.