A Linguagem Funcional Haskell

Márcio M. Piffer , Rafael A. Kroth

Ciências da Computação 3ª fase, 2002
Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC), Brasil, 88040-900
Fone (0XX48)333-9999, Fax (0XX48)333-9999
m_piffer@inf.ufsc.br, coca@inf.ufsc.br

RESUMO

Este estudo tem como principal objetivo fazer uma breve introdução a linguagem de programação Haskell, sendo abordado em primeira instância alguns dos princípios básico da linguagem, ou seja, onde esta foi fundamentada. Programar em uma linguagem funcional significa basicamente definir funções e utilizar o computador para avaliar expressões. Falando desta maneira não imaginamos como a mais ascendente das linguagens funcionais do momento o Haskell pode ser utilizado. Após a leitura deste artigo espera-se que o leitor no mínimo compreenda conceitos básicos, vantagens, e como a linguagem é utilizada.

Palavras-chave: Haskell, Funcional, Lambda.

ABSTRACT

This study has as main objective to do a brief introduction the programming language Haskell, being approached in first instance some of the basic of the language beginnings, that is to say, where this was based. To program in a functional language means basically to define functions and to use the computer to evaluate expressions. Speaking this way didn't imagine as the more ascendancy of the functional languages of the moment Haskell can be used. After the reading of this article it is waited that the reader at least understands basic concepts, advantages, and as the language it is used.

Key-words: Haskell, Funcional, Lambda.

Introdução

Existem diversos paradigmas de programação e diversas linguagens associadas a estes paradigmas, e este trabalho irá enfatizar o paradigma de programação funcional e a linguagem a ser abordada será o Haskell. O GHC (Glasgow Haskell Compiler) foi desenvolvido na universidade de Glasgow na Escócia, ou seja, seu nascimento por assim dizer foi dado lá. Sua última versão é a 1.4 e foi desenvolvida por um grupo internacional até março de 1997, e está sendo administrada, ou aperfeiçoada na universidade de Yale.

Princípios Básicos

A linguagem a ser estudada é baseada quase que totalmente no cálculo lambda (λ), sendo assim fica claro e evidente que aqueles que já possuírem

um conhecimento prévio desta área terão uma maior facilidade no entendimento do artigo do que aqueles que são leigos no assunto citado.

Outra questão interessante e que não pode deixar de ser dita é a de que quase todos os problemas que são implementados em Haskell já tiveram uma resolução indutiva construída anteriormente.

Conforme visto em **[SCH 92]** a indução é uma ferramenta muito poderosa no desenvolvimento de algoritmos e esta geralmente traz uma resolução elegante aos problemas apresentados. A Tese de Church, que não será abordada aqui, comprova que todo e qualquer problema resolvido imperativamente possui um resolução indutiva, ficando claro então que todos aqueles que quiserem se aventurar e conhecer o Haskell mais a fundo deveriam primeiramente possuir um conhecimento prévio destas duas áreas, cálculo lambda e indução.

Tendo consciência também de que esta é uma linguagem extremamente tipada, torna-se sabido que para aqueles que já tiveram algum contato como linguagens não tipadas como Perl, Tcl, ou Scheme, infelizmente terão uma dificuldade ainda maior para com o entendimento da linguagem.

Já aqueles que possuem algum tipo de experiência com linguagens tipadas como Java, C, Modula, ou até mesmo ML, irão ter maior facilidade no entendimento quanto ao sistema de tipos da linguagem apresentada neste trabalho.

Haskell também é conhecido como calculadora funcional e a forma de avaliação de expressões é de extrema importância, pois esta é que não deixará que certas expressões resultem em não-determinismo. Todas as expressões são reduzidas tendo como base o cálculo lambda e maiores aprofundamentos sobre este podem ser encontrados em **[THO 87]**.

Toda e qualquer expressão em Haskell é avaliada como sendo uma equação matemática, ou seja, está pode ser construída, avaliada e resolvida utilizando-se sempre de leis algébricas.

Qualquer expressão apresentada ao computador será entendida como sendo um valor, sendo assim o papel do computador torna-se obter a forma normal ou mais reduzida possível deste valor e isto é feito através de b - reduções. Deduz-se então que o significado de uma expressão é seu valor e a tarefa do computador é encontra - lo.

Sessões e Scripts

Ao passarmos uma expressão para esta ser avaliada pelo computador o processo de avaliação e evolução é chamado **sessão**, sendo assim tendo como prompt o ponto de interrogação (**?**). Alguns exemplos de sessão são apresentados abaixo.

Exemplo 1:
? 64 expressão apresentada
64 valor de retorno apresentado ao usuário

Exemplo 2:
? 8*8 expressão apresentada
64 valor de retorno apresentado ao usuário

Note que nos dois exemplos acima são passados expressões, a primeira já esta em sua forma normal, logo o computador não pode fazer mais avaliações, então ele somente retorna o valor a apresentado, no entanto a segunda expressão poder ser avaliada e evoluída então ele, o computador, faz a avaliação e evolução da expressão, até chegar a uma forma normal e daí então retorna o valor encontrado. Concluímos então que uma sessão é iniciada com a passagem de uma expressão para o computador e só será terminada quando este retornar algum valor.

O nome Script é dado a área de declaração de funções e tipos, e alguns exemplos de declaração de tipos e funções serão dados logo a seguir, por esta razão não serão dados exemplos por enquanto.

Valores e Tipos

Segundo **[HUD 97]** expressões denotam valores e tipos de expressões são apenas termos sintáticos que denotam tipos de valores ou só tipos. Todo valor possui um tipo associado, e intuitivamente podemos pensar que tipos são definições de valores.

Valores em Haskell são considerados tipos primários e podem ser passados para funções como argumentos, e estes irão retornar como respostas colocadas em estruturas de dados.

Tipos por outro lado não são considerados primários e trazem como sentido principal a descrição de um valor. A associação de um tipo a um valor é chamada ***"typing"***.

Usando tipos e valores no exemplo abaixo escrevemos alguns *typings*.

5 :: int

'a' :: char

O símbolo "::" deve ser lido como "tem tipo".

Funções em Haskell geralmente são definidos por uma série de equações, mas existe um tipo especial chamado "type signature declaration", que declara um tipo explicito para a função.

Um exemplo segue abaixo.

inc :: int → int "type signature declaration".

As "type signature declaration" não são de uso obrigatório, mas regras geralmente são utilizadas pelo programador para que este saiba que tipo possui cada função.

Quando queremos demonstrar que uma expressão e_1 é avaliada ou "reduzida" para uma outra expressão e_2 , nós indicamos desta maneira que se segue:

$e_1 \Rightarrow e_2$

Por exemplo note que:

inc (inc 3) ⇒ 5

Tipos Polimórficos

Tipos monomórficos são encontramos com muita frequência em diversas linguagens de programação, mas estes restrigem de certa forma a capacidade do programador.

Haskell também incorpora tipos polimórficos - tipos que são universalmente quantificados em algum lugar e referem-se a todos os outros tipos.

Expressões de tipos polimórficos essencialmente descrevem famílias de tipos. Por exemplo, (∀ *a*) [*a*] é a família de tipos consistindo de, para todo tipo *a*, o tipo de listas de *a*.

Listas de inteiros [1, 2, 3], listas de caracteres ['h', 'e', 'l' , 'l', 'o'], e até mesmo listas de listas de inteiros[[2], [4], [6]]. Todos são membros desta família, ou seja, no momento que usarmos a variável *a* está pode ser de qualquer tipo sendo desde que todos sejam do mesmo tipo. Haskell não irá admitir algo como ['b' , 2]. Nota-se então que através da definição de uma função que utiliza tipos polimórficos construímos, ou temos a oportunidade, de construir uma função genérica, onde esta poderá ser utilizada para qualquer família de tipos.

Para fazermos a declaração de um tipo polimórfico não necessitamos explicitar por escrito o símbolo do quantificação universal (∀). Em outras palavras todos os tipos de variáveis estão implicitamente quantificados.

Listas são muito utilizados em linguagens funcionais e são um ótimo veículo para que possamos demonstrar os princípios do polimorfismo.

A lista [1, 2, 3] é um resumo da lista 1: (2: (3: [])), onde [] é lista vazia e : é o operador de infixação, é ele que adiciona o primeiro argumento para a frente do segundo e assim por diante, e desta maneira conseguimos definir um lista.

Ao definirmos uma função que conta o número de elementos de uma lista, notamos as facilidades oferecidas por este sistema de tipos. Veja o exemplo abaixo:

length :: [a] → int

length [] = 0

length (x : xs) = 1 + length (xs)

Se prestarmos atenção a definição é quase que auto - explicativa. Nós podemos ler a função como sendo "O tamanho da lista vazia é zero, e o tamanho da lista cujo primeiro elemento é **x** e o restante e da lista é **xs** é 1 mais o tamanho de **xs**, passado novamente a esta função.

A função length é um exemplo de função polimórfica e sua "type signature declaration" demonstra isto. Sua grande vantagem vem quando depois de definida esta pode ser aplicada a qualquer tipo de lista e é isto que notamos no exemplo abaixo, pois são passados para a mesma função diversos tipos de listas entre elas listas de tipo [Int], Char], e até mesmo, [[Int]] lista de lista de inteiros.

length [1, 2, 3] ⇒ 3

length ['a', 'b'] ⇒ 2

length [[1], [2], [3], [4]] ⇒ 4

Com tipos polimórficos, achamos que alguns tipos estão em um senso extremamente mais geral que outros. Por exemplo o tipo [a] é mais geral do que o tipo [Char].

O Sistema de tipos Haskell possui duas importantes propriedades: Primeira, toda expressão bem tipada é garantida para ter um único tipo principal (explicado abaixo), e a Segunda, é que o tipo principal pode ser inferido automaticamente.

Uma expressão ou função tipo principal é o tipo menos geral, que, intuitivamente, "Contém todas as instâncias da expressão". Por exemplo, o tipo principal da função length é expressado como [*a*] → *a*; os tipos [b] → a, a → a , ou até mesmo de *a* são muito gerais, considerando que algum tipo como [Int] → Int é mais especifico. A existência de

tipos principais únicos é a característica mínima do sistema de tipos Hindley-Milner, sendo que este forma a base do sistema de tipos Haskell, ML, Miranda e a maioria das linguagens funcionais hoje conhecidas.

Em comparação com linguagens de tipos monomórficos como C, o leitor irá notar que o polimorfismo desenvolve expressões, e a inferência diminui a carga de tipos sobre os programadores.

User-Defined Types

Para definirmos nossos próprios tipos em Haskell usa-se a declaração ***data***, ao qual este trabalho tenta exemplificar de forma a tornar maior a compreensão do leitor.

Um importante tipo que já esta predefinido em Haskell, é a definição de tipos boleanos, mas só para exemplificar iremos fazer a sua criação tomando como se este não existisse:

data Bool = False | True

O tipo demonstrado acima é Bool e possui exatamente dois valores: Verdadeiro e Falso. O tipo Bool em Haskell possui a denominação de **tipo construtor**, e True e False são chamados **construtores de dado**, ou só construtores.

Podemos assim definir diversos tipos. Similarmente podemos definir o tipo Cor:

data Cor = Vermelho | Verde | Azul | Violeta

Mas se pararmos e prestarmos atenção estas duas definições de tipos são limitadas, ou seja, são tipos enumerados, pois possuem um número finito de construtores de dado.

Abaixo temos um exemplo com um único **construtor de dados**.

data Point = Pt a a

Definindo o tipo de dado Point não limitamos o usuário, pois este poderá definir qualquer tipo de ponto.

Um tipo como Point é freqüentemente chamado de *tuple type*, sendo que neste caso este é só um produto cartesiano (neste caso binário) de outros tipos. Tipos multi-construtor, como Bool e Cor, são chamados (disjunto) união ou tipos soma, em contraste a tipos mono construtor.

Mais importante, no entanto, é lembrar que Point também é um exemplo de tipo polimórfico: para qualquer tipo *t*, haverá a definição de pontos cartesianos que usam *t* como o tipo coordenada.

Note que o tipo de binário Pt é a → a → Point *a*, ou seja os tipos abaixo são válidos:

Pt 2.0 3.0 :: Point Float
Pt ‘a’ ‘b’ :: Point Char
Pt True False :: Point Bool

Não podemos esquecer de ressaltar que algo como **Pt ‘a’ 1** é sem tipo pois **‘a’** e **1** possuem tipos diferentes sendo assim o próprio sistema de tipos Haskell durante a compilação irá acusar erro.

É de extrema importância saber distinguir a aplicação entre construtor de dados para construir um valor e a aplicação de um tipo construtor para construir um tipo. O primeiro formador acontece em tempo de execução e é como nós computamos coisas em Haskell, já o segundo acontece em tempo de compilação e é parte do processo do sistema de tipos para assegurar tipos seguros.

Existe também a possibilidade de construirmos tipos utilizando o mesmo nome para o tipo construtor e para o construtor de dados, como vemos a seguir.

data Point a = Point a a

E já que estamos falando do sistema de tipos Haskell não podemos deixar de abranger tipos sinônimos, sendo que estes são utilizados para definir um mesmo tipo com outro nome, mas não novos tipos. Portanto tipos sinônimos criam um novo tipo a partir de um antigo só que com outro nome. Para que possamos criar tipos desta maneira utilizamos a declaração ***type***, como no exemplo abaixo.

type String = [Char]
type Name = String

Funções

Como toda e qualquer linguagem Haskell oferece funções pré-definidas, mas seu maior potencial está destinado ao programador, ou seja, a linguagem oferece muitos recursos a este, quanto a criação de novas funções. Alguns deste recursos já foram demonstrados no sistema de tipos, agora iremos abordar a parte funcional da linguagem.

Não iremos abordar funções oferecidas pela linguagem, pois nossa intenção não é ensinar a programar em Haskell e tão pouco ensinar a utilizar funções pré-definidas ou oferecidas pela linguagem.

Geralmente quando definimos uma função em Haskell utilizamos um *"type signature"* em primeiro lugar e na próxima linha iniciamos a declaração da função, como no exemplo que segue:

```
add :: Int → Int → Int -- type signature
add x y = x + y -- Corpo da função
```

Através deste exemplo temos conhecimento de como são feitos comentários de linha. Utilizamos dois sinais de menos (- -) consecutivos e tudo que for escrito após não será reconhecido pelo compilador.

Após este breve comentário continuamos falando sobre funções e por intermédio do mesmo exemplo, que é chamado de função de *curried*. Uma aplicação de **add** tem a forma **add** e_1 e_2, e isso eqüivale a (**add** e_1) e_2, desde que a associação da aplicação da função seja para a esquerda. Em outras palavras, aplicando **add** a um argumento, isto produzirá uma nova função ao qual é então aplicado o segundo argumento. Isto é consistente com o tipo de **add; Int → Int → Int**.

Sendo assim nós podemos redefinir a função **inc** de um modo diferente da que demonstramos anteriormente, agora podemos utilizar a função **add**. Esta maneira que apresentamos agora o é uma aplicação parcial de uma função *curried*.

Inc = add 1

Sendo a linguagem Haskell baseada quase que totalmente no Cálculo Lambda, podemos definir as mesmas funções de maneira muito mais reduzida e simplificada, mas não iremos demonstrar e ou entrar em detalhes pois exigiria, como já falamos antes, um conhecimento prévio do leitor sobre o cálculo lambda para que este conseguisse entender.

Podemos acompanhar o comportamento da função entendo o próximo exemplo.

Add (add 2 3) 5 ⇒ 10

O exemplo apresentado acima é praticamente auto-explicável, pois ao lermos ele entendemos que "A função add recebe dois argumentos um é uma função e o outro um valor sendo que este já se encontra na sua forma normal, então basta ser feita a resolução do primeiro argumento sendo que este dever retornar um valor e é isto que irá acontecer, sendo assim o próximo passo é a resolução da expressão". Repare que se o primeiro argumento retorna-se um caracter ‘a’ por exemplo este erro seria encontrado, mas somente em tempo de execução não de compilação.

Aplicações

Segue ainda uma listagem das áreas onde tem-se sido desenvolvido software utilizando a linguagem já referida.

Fran (animation)
Haskore (music)
CGI programming in Haskell
Happy (Parser generator)
Derive (Automatic derivation of classes from data declarations)
Tk Gofer (the Tk GUI library ported to Gofer, a language very similar to Haskell).

Conclusão

Concluímos após um ano de estudo que Haskell além dos inúmeros recursos que traz embutido consigo e dos demais que oferece, consegue hoje um crescimento extraordinário no mundo científico, sendo esta hoje a linguagem funcional mais ascendente do momento, apesar de não ser a mais utilizada ainda.

Foi descoberto também através deste estudo que um dos grandes anseios da população da área de informática atualmente a portabilidade, é muito bem proporcionada pela linguagem pois esta trabalha em diversos tipos de arquiteturas e sistemas operacionais. Interpretadores ou compiladores Haskell "rodam" em quase todo hardware ou sistema operacional existente hoje.

Deduz-se também que o estudo de outras áreas, afins é claro, torna-se necessário para uma maior compreensão de alguns problemas, e dentre estas áreas podemos citar a indução matemática, sendo que esta é de extrema importância não só para o aprendizado do Haskell mas em diversas outras áreas. Esta afirmação é feita após a descoberta de que existem estruturas na informática que possuem uma resolução extremamente elegante através da indução, entre eles estão árvores, listas, strings, e o mais clássico deles o das Torres de Hannoi.

Mais uma dica para quem quiser aprender esta nova linguagem é utilizar o Hugs um interpretador pequeno e de grande portabilidade. Este interpretador é um excelente veículo para aprender se aprender Haskell.

Nota Bibliográfica

[HUD 97] HUDAK, Paul e PETERSON, John. **A Gentle Intoduction to Haskell**. Los Alamos National Laboratory, 1997.

[SCH 92] SCHINEIDER, Gerardo. **Uso de la Inducción para el Diseño de Algoritmos**. UTN., Facultad Regional Concepión del Uruguay, Argentina. 1992.

[THO 87] THOMPSON, Simon. **An Introduction to Type Theory and Constructive Mathematics**. Canterbury, Kent, U.K. 1987.
http://www.haskell.org/