REMat
ISSN 2177-5095
nº3 - 2013

REVISTA ELETRÔNICA DE MATEMÁTICA
www2.jatai.ufg.br/ojs/index.php/matematica
contato: remat.ufg@gmail.com

Haskell: uma linguagem de programação ideal para matemáticos

Thiago VedoVatto
Aluno do curso de especialização em Matemática Aplicada - CAJ/UFG
supervedovatto@yahoo.com.br

Esdras Teixeira Costa
Professor da Coordenação de Matemática - CAJ/UFG
esdras.ufg@gmail.com

Abstract

Este trabalho é uma introdução à linguagem de programação Haskell, tendo como público alvo a comunidade de professores e pesquisadores em Matemática. Inicialmente é traçado o contexto histórico do desenvolvimento da linguagem e a seguir são apresentadas opções de compiladores/interpretadores; depois mostramos alguns exemplos da abordagem funcional para estruturas de dados e recursividade, entre outros. Ao final, se conclui que a proximidade do paradigma funcional com a Matemática torna Haskell, por ser uma linguagem puramente funcional, ideal para matemáticos.

Palavras-chaves: Haskell, Matemática, linguagem de programação

Haskell: a programming language ideal for mathematicians

Abstract

This work is an introduction to the Haskell programming language, aimed to Mathematics teachers and researchers. First we show the historic context of the language development and then we present options for compilers/interpreters; after, we show some examples of the functional approach to data structures and recursion, *inter alia*. In the end, one can conclude that the proximity of the fuctional programming to Mathematics makes Haskell, by being a purely functional language, ideal for mathematicians.

Keywords: Haskell, mathematics, programming language

REMat
ISSN 2177-5095
nº3 - 2013
REVISTA ELETRÔNICA DE MATEMÁTICA
www2.jatai.ufg.br/ojs/index.php/matematica
contato: remat.ufg@gmail.com

1 Introdução

Haskell, Lisp and SQL are the only programming languages that I've seen where one spends more time thinking than typing.[1] - **Philip Greenspun**[2]

Este artigo apresenta uma modesta introdução às possibilidades e conveniências da linguagem de programação Haskell, tendo como público alvo os matemáticos e eventualmente os programadores habituados às linguagens de programação imperativas como C ou Java. Além de mostrar algumas das capacidades desta linguagem baseada no paradigma funcional de programação, também temos como objetivo explicitar similaridades entre o código escrito em Haskell e a linguagem matemática contemporânea.

Tais similaridades já começam no próprio paradigma de programação (funcional), que é bastante distindo do modelo tradicional. Mas, nas palavras de [HASKELL,2011] qualquer um que já usou uma planilha eletrônica, tem experiência em programação funcional. Em uma planilha, especifica-se o valor de cada célula, em função dos valores de outras células. O foco está em "o quê" deve ser computado, e não "como" deve ser computado. Por exemplo:

- não especifica-se a ordem em que as planilhas devem ser computadas, ao invés disso tem-se garantia que a planilha será calculada em uma ordem que respeite suas dependências.
- não se diz à planilha como alocar sua memória aloca-se na memória somente as células que estão realmente em uso.
- na maioria das vezes, avalia-se o valor das células por uma "expressão" (cujas partes podem ser avaliadas em qualquer ordem), melhor que por uma sequência de comandos" que compute seu valor.

Apesar da citada proximidade com a matemática, a linguagem Haskell é uma linguagem de propósito geral; portanto, pode-se criar qualquer tipo de programa com ela. Outras linguagens como R, Octave, MatLab, Maple e Mathematica também se aproximam muito da linguagem matemática, mas essas últimas são de propósito específico, ou seja, são direcionadas à trabalhos matemáticos, estatísticos e afins. Mas este não é o ponto no qual Haskell mais se distancia de uma linguagem procedural/imperativa: em vez de termos algoritmos expressos através de listas de intruções, como ocorre nas linguagens supracitadas e também em C ou Java, Haskell tem seus algoritmos totalmente baseados no conceito matemático de função, fato que obviamente influenciou nossa escolha em trabalhar com esta linguagem.

No ato desta escolha, também fomos influenciados por alguns autores: segundo e [SILVA,2006, p.17], esta linguagem destaca-se pela sua clareza na codificação, reuso do código e pela pouca ou quase nenhuma exigência de conhecimento prévio de programação, além de ser uma linguagem de código aberto, multiplataforma e em franca expansão de uso; outros motivos para

[1] Haskell, Lisp e SQL são as únicas linguagens de programação nas quais gasto mais tempo pensando do que digitando.

[2] Professor de Engenharia de Software do Instituto de Tecnologia de Massachusetts (M.I.T.)

Figura 1: Logomarcas da linguagem Haskell

tal escolha podem ser encontrados no discurso de [O'DONNEL,2006, p. ix], segundo o qual é interessante e prazeroso aprender lógica, teoria de conjuntos, indução e matemática discreta usando Haskell além de ajudar a desenvolver intuição e superar concepções matemáticas errôneas.

Este artigo é estruturado da seguinte forma: inicialmente encontra-se um breve histórico do desenvolvimento da linguagem Haskell, seguido de seus principais compiladores e ferramentas. Logo após, a estrutura de dados é observada, com destaque para o trabalho com listas e conjuntos. Nas linhas seguintes explana-se sobre a importância das funções dentro da linguagem e observa-se como a lógica matemática usual é perfeitamente compatível com o paradigma funcional, de modo que é possível entender os algoritmos usando a lógica matemática. Ao final são citados alguns tópicos importantes sobre Haskell que servem como sugestão para estudos posteriores.

2 História da linguagem Haskell

Um pequeno resumo do desenvolvimento histórico da programação funcional e da linguagem Haskell pode ser encontrado nos textos de [HUTTON,2006, p.6], [WIKIPEDIA,2011], [O'SULLIVAN,2008] e [FRANCISCO,2006, p.5]. Sintetizando esses autores, em 1930 os matemáticos Alonzo Church e Stephen Kleene (Figura 2) desenvolvem o λ-cálculo, uma teoria de funções que é simples mas poderosa, providencial na fundamentação teórica das linguagens de programação funcional.

Em 1950 desenvolveu-se a linguagem de programação Lisp, que foi baseada em parte no λ-cálculo; em 1960 surge a linguagem ISWIM que é a primeira linguagem puramente funcional que se baseou fortemente no λ-cálculo. De acordo com [FRANCISCO,2006, p.13] uma função é pura quando se limita a tomar parâmetros e devolver resultados sem provocar nenhum efeito colateral[3] durante sua execução.

Em 1970 é desenvolvida a FP ("Funcional Programming"), uma linguagem de programação funcional que enfatiza as funções de alto nível (*high-order* functions)[4]; mais tarde surge a

[3]Um efeito colateral é essencialmente algo que ocorre no decorrer da execução de uma função que não possui conexão com a saída produzida pela função [DAUME,2002, p.12] .

[4]Uma função de alto nível (*high-order* function) é uma função que pode receber funções como argumento

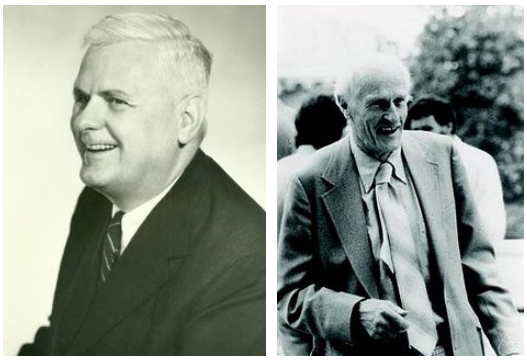

Figura 2: Alonzo Church e Stephen Kleene

Figura 3: Haskell Curry

linguagem ML ("Meta Linguagem") que introduz o conceito de polimorfismo de funções e o uso de inferência de tipos de dados; nas décadas de 70 e 80 a linguagem de programação comercial Miranda é pioneira no uso da "lazy evaluation" em suas rotinas.

No final da década de 80 as pesquisas envolvendo programação funcional resultaram na criação de mais de doze linguagens, de modo que em 1987 um comitê de pesquisadores inicia o desenvolvimento da linguagem de programação Haskell que se baseou no paradigma funcional com o uso de "lazy evaluation". A linguagem deveria ser de fácil ensino, deveria ser completamente descrita através de uma sintaxe e semântica formal, deveria estar disponível livremente. O nome da linguagem é uma homenagem ao lógico Haskell Curry (Figura 3).

Ao longo do seu desenvolvimento a linguagem Haskell passou por cinco revisões atendendo a dois propósitos principais, o primeiro deles é ser uma linguagem estável na qual se possa testar a eficiência de programas, o segundo é permitir que pesquisadores explorem técnicas de programação funcional.

No começo da década de 90 a comunidade de desenvolvimento Haskell tinha o slogan informal "evitar o sucesso a todo custo" razão pela qual a linguagem ficou restrita aos am-

ou retornar uma função como resultado. [HUTTON,2006, p.62]

```
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

Figura 4: Tela inicial, em modo texto, do compilador GHC

bientes acadêmicos. No final da década de 90, à medida em que surgiram linguagens mais dinâmicas como Python, Perl e Ruby apresentando-se como alternativas às tradicionais linguagens C e Java, a linguagem Haskell começa a sair para fora das universidades e centros de pesquisa. Em janeiro de 1999 é publicado o **Haskell 98**, a primeira versão estável da linguagem Haskell, que continua em processo de desenvolvimento.

Segundo [O'SULLIVAN,2008], Haskell é a linguagem funcional sobre a qual mais pesquisa está sendo realizada, nela tem-se a oportunidade de ver como conceitos matemáticos abstratos se relacionam diretamente com os comandos e estruturas que são construídas. Vale observar, entretanto, que a definição de programação funcional, de acordo com [HUTTON,2006, p.2], ainda não está consolidada dentro da computação, portanto assumiremos neste artigo que a programação funcional consiste na técnica de aplicar as funções como argumentos, de modo que a função é a única responsável por calcular as saídas baseando-se unicamente nos valores de entrada.

3 Compiladores

Sintetizando [HUDAK,1999], um compilador é o programa responsável por fazer a tradução de uma linguagem de alto nível, no nosso caso a linguagem Haskell, para a linguagem de máquina, que é a linguagem usada pelo computador para executar os algoritmos que criaremos. Atualmente os compiladores GHC e Hugs são os mais utilizados com a linguagem Haskell. Ao longo deste texto utilizamos o compilador GHC (Figura 4), que possui versões compatíveis com os principais sistemas operacionais, sua última versão pode ser obtida gratuitamente em [HASKELL,2011].

No site [HASKELL,2011b] há também um tutorial interativo (Figura 5) onde podemos testar pequenos comandos Haskell diretamente online, sem que nada precise ser instalado.

Ainda é possível obter em [LEKSAH,2011] o editor gráfico Leksah (Figura 6) que uma é ferramenta prática no desenvolvimento de programas Haskell.

4 Programação concisa e eficaz

[O'DONNEL,2006] mostram que a sintaxe da linguagem Haskell foi projetada para ser similar à notação matemática usual e consequentemente ser de fácil leitura; nesta mesma obra, os autores ainda chamam a atenção para o fato de que os algoritmos *devem* ser bem indentados

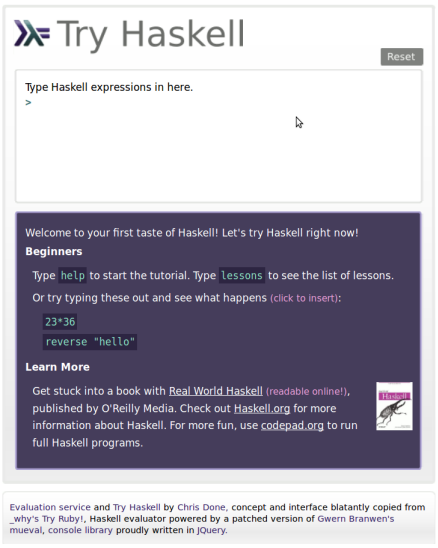

Figura 5: Try Haskell - Tutorial interativo

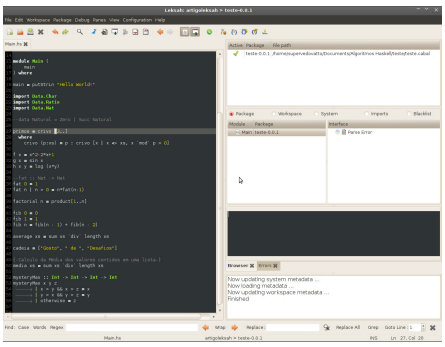

Figura 6: Leksah - IDE para Haskell

de modo a deixar clara sua estrutura. Os mesmos exemplificam que em equações que não cabem em apenas uma linha é necessário indentar as linhas subsequentes, tal disposição deixa claro tanto ao leitor como também ao compilador que as linhas subsequentes são continuação das linhas anteriores, tal procedimento também é utilizado na escrita matemática.

```
x = a + b + c + d + e y = 2 * x
```

Na visão de [HUTTON,2006, p.4] o fato de Haskell utilizar conceitos de alto nível permite que o código seja de duas a dez vezes menores do que o código de programas criados em linguagens convencionais como C e Java. Como consequência os algoritmos se tornam naturalmente elegantes. Em [HASKELL,2011] encontra-se duas implementações do programa `quicksort` que ordena uma sequência de números em ordem ascendente. O primeiro programa é escrito em Haskell e o segundo em C. Veja a implementação em Haskell:

```
 qsort [] = [] qsort (x:xs) =
qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
```

E agora compare com a mesma implementação em C:

```
void qsort(int a[], int lo, int hi)
int h, l, p, t;
if (lo < hi)  l = lo; h = hi; p = a[hi];
do while ((l < h) && (a[l] <= p)) l = l+1;
while ((h > l) && (a[h] >= p)) h = h-1;
if (l < h)  t = a[l];
a[l] = a[h];
a[h] = t;
while (l < h);
t = a[l];
a[l] = a[hi];
a[hi] = t;
qsort( a, lo, l-1 );
qsort( a, l+1, hi );
```

Fica claro que o código fonte em Haskell tende a ser muito menor que nas linguagens convencionais. Além disso, [O'SULLIVAN,2008, p. xxv] confirmam que após anos de experiência ficaram impressionados e muito felizes com a frequência com a qual os programas em Haskell simplesmente rodam na primeira tentativa, sem necessidade de corrigir erros de compilação.

5 Estruturas de dados

Haskell é uma linguagem puramente funcional, portanto, segundo [HUDAK,1999, p.2] todos os cálculos são feitos mediante avaliação de *expressões* envolvendo campos de *dados*. Como em quase qualquer outra linguagem de programação todos os dados tem um *tipo* associado. Resumindo [HUTTON,2006, p.18], em Haskell os principais tipos de dados são:

Bool Valores lógicos (`False` e `True`);

Char Caracteres isolados (`'a'`, `'A'`, `'3'` e `'_'`);

String Séries de caracteres (`''abc''`, `''1+2=3''`);

Int Inteiros entre -2^{31} e $2^{31}-1$;

Integer Inteiros quaisquer;

Float Números decimais -12.34, 1.0 e 3.14159.

[O'DONNEL,2006, p.11] mostram que as estruturas de dados mais usadas em Haskell são as tuplas e as listas. Uma tupla, segundo [HUTTON,2006, p.20], é uma sequência finita de elementos de tipos possivelmente diferentes, delimitadas por parênteses e separados por vírgulas. Escrevemos $(T_1, T_2, \ldots, T_n)$ para representar uma tupla de n elementos com o tipo T_i para i de 1 à n. As tuplas a seguir exemplificam a definição dada:

```
(False, True)::(Bool,Bool)
(False, 'a', True)::(Bool, Char, Bool)
("Yes",True,'a')::(String,Bool,Char)
```

Podemos ler as linhas acima da seguinte forma: a tupla `(False, True)` é do tipo `(Bool,Bool)`, a tupla `(False, 'a', True)` é do tipo `(Bool, Char, Bool)`, e, por fim, a tupla `("Yes",True,'a')` é do tipo `(String,Bool,Char)`. É importante ressaltar que as tuplas são ordenadas, ou seja, se promovermos uma alteração na ordem em que os elementos aparecem, a tupla obtida é diferente. A titulo de ilustração, o comando a seguir verifica se as tuplas `(1,2)` e `(2,1)` são iguais, a resposta `False` nos mostra que as tuplas são distintas.

```
 > (1,2)==(2,1)
False
```

Nesse caso pode-se interpretar tais tuplas como coordenadas de dois pontos do plano cartesiano e como já sabemos os pontos de coordenadas $(1, 2)$ e $(2, 1)$ são distintos.

Uma lista, segundo [HUTTON,2006, p.20], é uma sequência de elementos de mesmo tipo, com seus elementos delimitados por colchetes e separados por vírgulas. Em Haskell, é possível construir listas por meio de intervalos como vemos no exemplo a seguir, onde código `[2..10]` cria uma lista de inteiros de 2 a 10.

```
 > [2..10]
[2,3,4,5,6,7,8,9,10]
```

Pode-se omitir o final do intervalo, como no comando `[2..]`, que gera uma lista infinita de inteiros a partir de 2. Evidentemente não é possível exibir essa lista aqui! Observe que no último exemplo, a notação `..`, que gerou uma sequência de 2 a 10 com passo 1, no exemplo a seguir gera uma nova sequência de 2 à 10 agora com passo 2.

```
 > [2,4..10]
[2,4,6,8,10]
```

5.1 Números racionais

[O'DONNEL,2006, p.9] mostra que a linguagem Haskell suporta a aritmética dos números racionais, permitindo que você trabalhe com frações tão bem quanto com os números decimais equivalentes (aproximações). Os `Ratio Integers` são o tipo de dados dos número racionais,

no qual o numerador e o denominador são representados por meio de dois valores do tipo `Integer`.

Para que o compilador compreenda os números racionais é necessário carregar o pacote `Ratio` com o comando:

```
:module +Data.Ratio
```

Por exemplo, o inteiro $\frac{2}{3}$ pode ser representado pelos comandos:

```
> 2 % 3
2/3 :: Ratio Integer
```

Também é possível efetuar operações com `Ratio Integers`, sendo que o resultado já é dado como uma fração reduzida.

5.2 *List Comprehension*

Segundo [HUTTON,2006, p.38], na matemática a compreensão da notação pode ser usada para construir novos conjuntos numéricos a partir de conjuntos já existentes. Por exemplo, a compreensão de $\{x^2 | x \in \{1 \ldots 5\}\}$ produz o conjunto $\{1, 4, 9, 16, 25\}$ formado por todo x^2 desde que x seja um elemento do conjunto $\{1 \ldots 5\}$. A compreensão das listas (*list comprehension*), assim como a compreensão da notação matemática, permite construir listas através de conjuntos dado; o comando a seguir constrói o conjunto descrito no parágrafo anterior.

```
 > [x2 | x <- [1..5]]
[1,4,9,16,25]
```

O código `[x^2 | x <- [1..5]]` que gera o conjunto $\{1, 4, 9, 16, 25\}$ é muito semelhante à notação matemática que foi utilizada anteriormente. O código cria uma lista de elementos `x^2` a partir do gerador `[1..5]`, o símbolo `<-` representa o sinal $\in$ (pertence à). Segundo [O'DONNEL,2006, p.13], a sintaxe básica para a compreensão das listas (*list comprehension*) é: primeiro, a expressão, depois uma barra vertical, seguida do gerador:

```
[expression | generator]
```

6 Funções

Lisp was the most beautiful language in the world at least up until Haskell came along.[5] - **Larry Wall**[6]

[5] Lisp foi a linguagem mais bonita do mundo até que surge Haskell

[6] **Programador famoso por ser autor da linguagem de programação PERL**

[LEITHOLD,1994, p.909] define que uma *função de n variáves* é um conjunto de pares ordenados (P, w), onde dois pares distintos não podem ter os primeiros elementos iguais [7]. P é um ponto no espaço n-dimensional numérico e w um número real. O conjunto de todos os valores possíveis de P é chamado de *domínio* da função, enquanto que o conjunto de todos os valores possíveis de w é chamado de *imagem* da função.

[HUTTON,2006, p.1] define que, na linguagem Haskell, uma função é um mapeamento que toma um ou mais argumentos e produz um único resultado, e é definida usando uma equação que dá o nome para a função, um nome para cada um de seus argumentos e um corpo que especifica como o resultado pode ser calculado em termos dos argumentos. [THOMPSON,1999, p.3] ilustra bem esta definição para a função de adição na Figura 7, na qual, dados os valores de entrada 12 e 34, a função retorna o valor de saída 46.

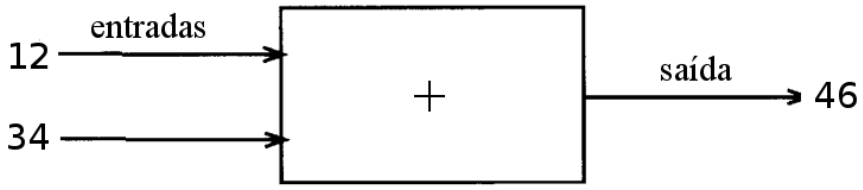

Figura 7: $12 + 34 = 46$

[THOMPSON,1999, p.3] ainda define que o processo de inserir entradas em uma função é chamado de aplicação funcional (*function aplication*). [O'DONNEL,2006, p.15] exemplificam que podemos aplicar a função `sqrt` no argumento `9` e obter o resultado `3.0`, na notação matemática usual esse comando equivale à $\sqrt{9} = 3$.

```
> sqrt 9
3.0
```

6.1 Funções no contexto da programação funcional

[O'DONNEL,2006, p.15] define que uma função que toma um argumento do tipo `a` e retorna um resultado do tipo `b` é uma função do tipo `a→b`, desde modo escrevemos `f::a→b`. Nesse momento é interessante fazer uma comparação com a definição de função $f : A \to B$ de [LIMA,2010, p.13] que diz que a mesma consta de três partes: um conjunto A, chamado o *domínio* da função, ou conjunto onde a função é definida, um conjunto B, chamado o *contradomínio* da função, ou o conjunto onde a função toma valores, e uma regra que permite associar, de modo bem determinado, a cada elemento $x \in A$, um único elemento $f(x) \in B$, chamado o valor que a função assume em x, ou no ponto x. Ao assumirmos que o conjunto A é o conjunto de todos os argumentos do tipo `a` e o conjunto B representa o conjunto de

[7] Isso significa que se (P, w_1) e (P, w_2) são dois pares de uma função, então $w_1 = w_2$, ou seja, em cada ponto P ela pode ter apenas um único valor correspondente, $w_1 = w_2$.

todos os valores do tipo `b` fica clara a proximidade dos conceitos matemáticos com a sintaxe Haskell.

[HUTTON,2006, p.13] mostra que novas funções preferencialmente não devem ser definidas no prompt de comando do GHCi (interpretador do GHC), é preferível que sejam definidas em um script, um arquivo de texto que compreende uma sequência de definições. Por convenção os scripts escritos em Haskell (*scripts Haskell*) possuem a extensão *.hs* para diferencia-los dos outros tipos de arquivos. Para testar o script *teste.hs* deve-se abrir o GHCi em um terminal e executar o comando:

```
:load teste.hs
```

[O'DONNEL,2006, p.16] completa argumentando que novas funções podem ser definidas declarando-se seu tipo seguido da definição de sua equação, conforme a sintaxe:

```
Função :: TipoArg1 -> TipoArg2 -> ... -> TipoArgn -> TipoResult
```

Como exemplo, a função $f : \mathbb{R} \to \mathbb{R}$ expressa por $f(x) = x^2 - 2x + 1$ pode ser definida em um script Haskell do seguinte modo:

```
f:: Float -> Float f x = x2-2*x+1
```

[O'SULLIVAN,2008, p.20] ressaltam que a declaração do tipo da função é opcional, pois um compilador Haskell pode automaticamente deduzir, através do processo de inferência de tipos (*type inference*), o tipo de qualquer função. A inferência de tipos ajuda a evitar uma infinidade de erros relacionados as entradas de dados. Deste modo as funções $g(x) = sin(x)$ e $h(x, y) = ln(xy)$ podem ser definidas respectivamente pelas seguintes instruções em um script Haskell:

```
 g x = sin x
h x y = log (x*y)
```

Veja como é simples realizar uma aplicação funcional com as mesmas:

```
 > f 2
1
> g (pi/2)
1.0
> h 4 2
2.0794415416798357
```

Os comandos acima equivalem respectivamente à $f(2) = 1$, $g(\frac{\pi}{2}) = 1$ e $h(4, 2) = 2.079\ldots$.

A linha a seguir mostra como é fácil trabalhar a composição de funções na linguagem Haskell. Aqui se calcula o valor de $h(f(2), g(\frac{\pi}{2}))$ com base nas funções definidas anteriormente:

```
> h (f 2) (g pi/2) -37.33188921992844
```

Nas palavras de [O'DONNEL,2006, p.19] uma função é chamada de função de primeiro nível (*first order function*) quando seus argumentos e resultados são valores de dados ordinários, e é chamada de função de alto nível (*higher order function*) se ela toma outra função como argumento ou se retorna uma função como resultado. Os mesmos autores ainda afirmam que as funções de alto nível tornam possível uma grande variedade de técnicas de programação permitindo que novas funções sejam definidas sem o uso direto das funções predefinidas da linguagem.

Como exemplo, considere a função $k(x) = f(g(x))$ definida em um algoritmo Haskell pela seguinte linha:

```
k x = f (g x)
```

A função equivale à composição de f e g. A teoria elementar das funções mostra que para calcular o valor de $f(g(x_0))$ deve-se, primeiramente, determinar o valor de $g(x_0)$, nesse ponto a função f depende do valor de outra função, isso equivale a dizer que a função f, e consequentemente a função k, recebem a função g como argumento, desde modo k é uma função de alto nível.

7 Resolução de equações (*Equational reasoning*)

[O'DONNEL,2006, p.38] afirmam categoricamente que as equações em Haskell são equações matemáticas verdadeiras e não atribuições escritas. Logo é possível "resolver" tais equações utilizando álgebra elementar. Em Haskel,l as leis de precedência são semelhantes às da álgebra. Os dois comandos a seguir são equivalentes, observe que mesmo sem o uso dos parênteses a prioridade foi da operação de multiplicação.

```
 > 1 + (4 * 4)
17
> 1 + 4 * 4
17
```

A seguir vê-se que a associatividade também é preservada, os dois comandos a seguir possuem a mesma resposta.

```
 > 2 * (3 + 4)
14
> 2 * 3 + 2 * 4
14
```

A resolução de equações em Haskell se assemelha muito a lógica de resolução de uma equação matemática convencional. Para entender bem seu funcionamento considere o exemplo de [O'DONNEL,2006, p. 39], no qual definem-se as funções f e g à seguir:

```
 f :: Integer -> Integer -> Integer
f x y = (2+x) * g y
g :: Integer -> Integer
g z = 8-z
```

Deste modo pode-se resolver manualmente a expressão `f 3 4`.

```
 f 3 4 = (2+3) * g 4 --  função f
f 3 4 = (2+3) * (8-4) --  função g
f 3 4 = 20 --  aritmética básica
```

Outro exemplo de resolução de uma equação matemática convencional é dado pela função `length` que calcula a quantidade de elementos em uma lista; o exemplo a seguir é inspirado em [THOMPSON,1999, p. 136]:

```
 length :: [a] -> Int --  Domínio e Imagem
length [] = 0 --  definição 1
length (n:ns) = 1 + length ns --  definição 2
```

A definição 1 dá o significado de `length []`, ou seja, define que o comprimento de uma lista sem elementos é zero. A definição 2 mostra que o comprimento de uma lista com n elementos será igual à `1 + length ns`, ou seja, será igual à uma unidade mais o comprimento da lista que contém todos os elementos da lista original exceto o primeiro - note também aqui a similaridade com o princípio de indução finita.

Ao executar o comando `length [2,3,1]` o processamento ocorre da seguinte forma:

```
length [2,3,1] = 1 + length [3,1] --  definição 2
length [2,3,1] = 1 + (1 + length [1]) --  definição 2
length [2,3,1] = 1 + (1 + (1 + length[])) --  definição 2
length [2,3,1] = 1 + (1 + (1 + 0)) --  definição 1
length [2,3,1] = 1 + (1 + 1) --  soma
length [2,3,1] = 1 + 2 --  soma
length [2,3,1] = 3 --  soma
```

Como se observa, todos os passos do cálculo manual de `length [2,3,1]` são justificados da maneira matemática clássica, mediante definições feitas anteriormente.

O uso da sintaxe Haskell permite deduzir propriedades sobre funções. No caso da função `length` pode-se provar que `length [x] = 1`, ou seja que o comprimento de uma lista formada por apenas um único elemento é sempre unitário qualquer que seja esse elemento. Observe como:

```
 length [x] = 1 + length [] --  definição 2
length [x] = 1 + 0 --  definição 1
length [x] = 1 --  soma
```

8 Recursividade e tópicos avançados

Muitos cálculos computacionais podem ser programados de uma forma simples e natural através de recursão ou mesmo através de uma uma função recursiva. [DAUME,2002, p.29] nos mostra que nas linguagens imperativas como C e Java, um laço (*loop*) é uma das mais básicas estruturas de controle. No entanto os laços não fazem muito sentido em Haskell, pois eles requerem atualizações destrutivas (o índice variável é constantemente atualizado). Ao invés disso, Haskell usa recursão, que segundo [O'DONNEL,2006, p.47] é um estilo de definição auto-referenciada comumente usada na matemática e nas ciências da computação.

A função fatorial é um bom exemplo de função recursiva. [IEZZI,2002, p.359] definem o fatorial de um número natural:

Proposição 1. *[Fatorial de um número natural n] Dado um número natural n, $n \geq 2$, o fatorial de n (indica-se por n!) é o produto dos n primeiros naturais positivos, escritos desde n até 1, isto é:*

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

Os mesmos autores ainda complementam com as definições especiais $0! = 1$ e $1! = 1$. É possível reduzir o cálculo de $n!$ à essas duas equações:

$$0! = 1$$

$$(n+1)! = (n+1) \times n!$$

[DAUME,2002, p.29] mostra que em uma linguagem imperativa como C e Java um protótipo dessa função seria semelhante ao script:

```
int factorial(int n)
int fact = 1;
for (int i=2;
i <= n; i++)
fact = fact * i;
return fact;
```

O algoritmo acima não possui muita semelhança com a definição matemática de fatorial dada anteriormente. Um script Haskell que define a mesma função será:

```
 fatorial :: Int -> Int fatorial 0 = 1 --  definição 1
fatorial n = n * fatorial (n - 1) --  definição 2
```

A primeira linha do algoritmo estabelece o tipo de dados da entrada e da saída (a função recebe um número `Int` e retorna um número `Int`). A segunda linha estabelece que $0! = 1$. A última linha define a função fatorial recursivamente, ou seja, o fatorial de $n + 1$ depende do fatorial de n e assim por diante recursivamente. Observe como o algoritmo trabalha para efetuar o cálculo de 4!:

```
fatorial 4 = 4 * fatorial 3 --  definição 2
fatorial 4 = 4 * 3 * fatorial 2 --  definição 2
fatorial 4 = 4 * 3 * 2 * fatorial 1 --  definição 2
fatorial 4 = 4 * 3 * 2 * 1 * fatorial 0 --  definição 2
fatorial 4 = 4 * 3 * 2 * 1 * 1 --  definição 1
fatorial 4 = 24 --  produto
```

Observe que os passos que o algoritmo executa são análogos aos utilizados para calcular com 4! com "lápis e papel". Um outro exemplo de recursão pode ser encontrado na Sequência de Fibonacci, que é uma clássica sequência da matemática, estabelecida pelas equações:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

Em mais um exemplo de como a escrita na linguagem Haskell é similar à um texto matemático, o algoritmo da Sequência de Fibonacci é escrito da seguinte forma:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib(n - 1) + fib(n - 2)
```

Observe como o algoritmo trabalha para encontrar o sexto elemento da Sucessão de Fibonacci, para tanto é necessário executar o comando `fib 5` no terminal:

```
fib 5 = fib 4 + fib 3
fib 5 = fib 3 + fib 2 + fib 2 + fib 1
fib 5 = fib 2 + fib 1 + fib 1 + fib 0 + fib 1 + fib 0 + 1
fib 5 = fib 1 + fib 0 + 1 + 1 + 0 + 1 + 0 + 1
fib 5 = 1 + 0 + 4
fib 5 = 5
```

Cabe ressaltar que os algoritmos que foram exibidos até aqui foram escolhidos com o objetivo mostrar como a linguagem Haskell se assemelha à escrita matemática, mas um fato deve ficar claro: a semelhança de um algoritmo com a escrita matemática não implica que o algoritmo seja computacionalmente eficiente. O algoritmo da Sequência de Fibonacci, por exemplo, pode ser implementado de várias formas diferentes:

```
fib = (fibs !!)
    where fibs = 0 : scanl (+) 1 fibs

fib n = fibs !! n
    where fibs = 0 : scanl (+) 1 fibs

fib n = fibs !! n
    where fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fib n = (fibs 0 1) !! n
    where fibs a b = a : fibs b (a+b)
```

Algumas das implementações acima não possuem muita semelhança, ou mesmo semelhança alguma, com a escrita matemática, mas podem apresentar uma performance melhor em termos de consumo dos recursos da máquina.

Exemplos mais avançados podem ser encontrados em programas que fazem uso excessivo de *lazy* evaluation, de modo que nenhum cálculo é executado até que seus resultados tenham sido requeridos, deste modo se evita o desperdício de processamento. Este, talvez, seja o tópico avançado mais intrigante da linguagem, juntamente com os Monads: eventualmente, é necessário se adicionar efeitos colaterais às funções Haskell enquanto o programa está rodando, como a leitura das entradas do teclado e a exibição dos resultados na tela, e isso é obtido justamente graças a implementação matemática do conceito de *Monad*. A utilização de tais ferramentas fogem ao escopo de uma pequena introdução como esta aqui apresentada, mas se colocam como excelentes opções para um aprofundamento no assunto.

9 Conclusão

O artigo apresentou uma rápida introdução à linguagem Haskell de modo a destacar a semelhança entre o código Haskell e a notação matemática usual, sem preocupações com temas mais específicos da linguagem. Vimos que esta é uma linguagem de programação altamente baseada em conceitos matemáticos e de uso geral, que pode ser usada em praticamente qualquer tipo de programa e problema. Na internet podemos encontrar vários programas e algoritmos feitos em Haskell; destaca-se aqui o site oficial da linguagem (http://www.haskell.org) que possui vasto conteúdo abrangendo tutoriais, documentação, implementações, bibliotecas, etc.

Nas Referências desse artigo o leitor encontrará várias obras fundamentais para a aprendizagem da linguagem Haskell. Aqueles que desejaram aprender técnicas de programação na linguagem Haskell recomenda-se a leitura de[HUTTON,2006, O'SULLIVAN,2008, THOMPSON,1999, SILVA,2006] e para aqueles que desejarem aprofundar mais sobre os fundamentos matemáticos da linguagem recomenda-se [O'DONNEL,2006].

Referências

[DAUME,2002] DAUMÉ, I. H. **Yet Another Haskell Tutorial**. 1. ed. http://www.haskell.org/tutorial/, 2002. 192p.

[FRANCISCO,2006] FRANCISCO, A.: **Haskell: A pure functional programming with class!** [S.l.], Abril 2006.

[HUDAK,1999] HUDAK, P.; PETERSON, J.; FASEL, J. A **gentle introduction to Haskell 98**. 1. ed. http://www.haskell.org/tutorial/, Outubro 1999.

[HUTTON,2006] HUTTON, G. **Programming in Haskell.** 1. ed. New York: Cambridge University Press, 2006. 171˜p.

[IEZZI,2002] IEZZI, G.; DOLCE, O.; PÉRIGO, D.D. ans R. **Matemática - Volume único**. 2. ed. São Paulo: Atual, 2002. 660˜p.

[LEITHOLD,1994] LEITHOLD, L. **O Cálculo com Geometria Analítica**. 3. ed. São Paulo: Harbra, 1994. 493˜p.

[LEKSAH,2011] THE LEKSAH TEAM. **Leksah - Haskell IDE in Haskell.** DisponÍvel em: http://leksah.org/. Acesso em: 12 jun. 2011.

[LIMA,2010] LIMA, E.L. **Curso de Análise**. 12. ed. Rio de Janeiro: Associação Instituto Nacional de Matemática Pura e Aplicada, 2010. 431˜p. (Projeto Euclides, v.1).

[O'DONNEL,2006] O'DONNELL, J.; HALL, C.; PAGE, R. **Discrete mathematics using a computer**. 2. ed. New York: Springer, 2006. 442˜p.

[O'SULLIVAN,2008] O'SULLIVAN, B.; GOERZEN, J.; STEWART, D. **Real World Haskell**. 1. ed. California: O'Reilly, 2008. 672˜p.

[SILVA,2006] SÁ, C.C. de; SILVA, M.F. da. **Haskell: Uma abordagem prática.** 1. ed. São Paulo: Novatec, 2006. 280˜p.

[SILVA,2003] SILVA, V.V. da.: **Números: Construção e propriedades.** 1. ed. Goiânia: Editora UFG, 2003. 292˜p. (Coleção Didática)

[HASKELL,2011] .THE HASKELL TEAM: **Haskell Programming Language**. 2011. Site oficial da linguagem Haskell. Disponível em: http://www.haskell.org/haskellwiki/Haskell. Acesso em: 12 jun. 2011.

[THOMPSON,1999] THOMPSON, S. **Haskell: The craft of functional programming.** 2. ed. New York: Addison-Wesley, 1999. 504˜p. (Pearson Education).}

[HASKELL,2011b] THE HASKELL TEAM. **Try Haskell! An interactive tutorial in your browser.** Disponível em: http://tryhaskell.org/. Acesso em: 12 jun. 2011.

[WIKIPEDIA,2011] WIKIPEDIA: The free encyclopedia. 2011. **Artigo sobre Haskell na Enciclopédia Livre Wikipédia.** Disponível em: http://en.wikipedia.org/wiki/Haskell. Acesso em: 12 jun. 2011.