Lomonosov Moscow State University
Faculty of Computational Mathematics and Cybernetics

Essay

«Haskell functional programming language»

Gordeev Mikhail
student 518/1 group

Moscow, 2016

1 Introduction

Haskell is a general-purpose purely functional programming language.

Historically it became so that today the most popular approach to programming is imperative. With such approach a program is a sequence of instructions, which should be executed exactly in the specified order. Furthermore, imperative programming imply the existence of an assignment operator, as a programmer often changes the state of many variables.

However, there is a completely different approach to programming, namely declarative. With such approach the program is a set of declarations describing the result that should be achieved. Functional programming is one of the incarnations of declarative approach. Moreover, Haskell has no assignment operator and all variables are not variables at all, they are nothing but constants.

The key features of Haskell are:

1. existence of pure functions,
2. separation of pure functions and functions with side-effects,
3. lazy evaluation,
4. algebraic data types.

2 History

Following the release of Miranda by Research Software Ltd, in 1985, interest in lazy functional languages grew: by 1987, more than a dozen non-strict, purely functional programming languages existed. Of these, Miranda was the most widely used, but it was proprietary software. At the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, a meeting was held during which participants formed a strong consensus that a committee should be formed to define an open standard for such languages. The committee's purpose was to consolidate the existing functional languages into a common one that would serve as a basis for future research in functional-language design.

The first version of Haskell ("Haskell 1.0") was defined in 1990.In late 1997, the series culminated in Haskell 98, intended to specify a stable, minimal, portable version of the language and an accompanying standard library for teaching, and as a base for future extensions. The committee expressly welcomed the creation of extensions and variants of Haskell 98 via adding and incorporating experimental features. In February 1999, the Haskell 98 language standard was originally published as "The Haskell 98 Report". In January 2003, a revised version was published as "Haskell 98 Language and Libraries: The Revised Report". The language continues to evolve rapidly, with the Glasgow Haskell Compiler (GHC) implementation representing the current de facto standard.

3 Three foundations of typing

Haskell has a serious relationship with types. His type system is based on three foundations:

1. static checking
2. strictness
3. automatic inference

3.1 Foundation one

Static type checking is a checking of each expression, which is performed during compilation stage. If the compiler doesn't like something in some expression's type, the compilation will fail.

Respectively, if the compilation of Haskell code succeeded, we can say that everything is fine with types, because we have a second foundation.

3.2 Foundation two

Type strictness is a requirement of a correspondence between something we expect and something we get.

For example we can write the following function in C:

```c
int coefficient() {
    return 12.9;
}
```

This is an example of implicit type conversion. We expect a value of type int but in fact we get a value of type double. However the compiler will allow this and truncate the fractional part of the return value, because this expression's type will be implicitly converted to int.

There is no chance for this code to compile in Haskell, because there is no implicit type conversion there: if we expect an integer - you need to provide exactly an integer.

However, explicit type conversion is also very limited. In C++ we can write something like:

```
int main() {
    std::cout << (int)'1' << std::endl;
}
```

We took a value of type char and remade it into a value of type int. The compiler keeps silence. The consequences of such errors are well known.

In Haskell we can explicitly name the type of some value, but only if the type is associated with the value. Like if it is the number 1, we can only explicitly name a "numerical" type (like Integer or Double). Tricks with converting a symbol to an integer, like the one shown above, are not possible in Haskell.

3.3 Foundation three

Automatic type inference is a compiler's ability to understand an expression's type by the expression itself.

For example, in C we need to declare the type explicitly:

```
double i = 10.34;
```

We do not need this in Haskell. We just write:

```
i = 10.34;
```

The compiler will analyze the value 10.34 and will understand, that type of i is Double. However, as it was already said, we can explicitly name the type of the expression (and sometimes we must do it).

4 Immutability

One of the fundamental properties of Haskell is an absence of an assignment operator.

When I first heard about this property, I couldn't believe it. How is it possible to code without an assignment operator? And how will we change the state of our variables? My surprise was understandable: when writing C++ code I often use the assignment operator.

To understand, consider the following expression:

```
a = 123
```

This construction means assignment in imperative languages. In this case, we say: "Take the sequence of bytes, corresponding to the value 123, and change the sequence of bytes, stored in the variable a, with it." Thus, the overwriting of the old value with the new value occurs.

But in pure functional language this instruction means the same that it does in maths, namely the equality. In this case we declare: "The value of a is equal to 123."

What's the difference, you may ask. In any case we get a variable a with the value 123. The difference is that the assignment of the same variable can occur many times, but at the same time the equality declaration can occur only once. So if we declared, that the value of a is 123, then it will be so forever. That's why Haskell has neither the concept of "variable", nor the const keyword, because all values are constant in it.

You might be interested how to add an element to a collection if everything is constant? The answer is: you can't do it. We can't change the value, we can only create a new value from it. Do not worry about the memory: it will be allocated and freed automatically.

5 Laziness and infinite lists

5.1 Infinite lists

Another interesting feature of Haskell is infinite lists.

Infinite lists provide a way to define mathematical series in a neat manner. Consider a definition such as `[1,3..]` for example. As you might have guessed this gives you an ascending series with numbers 1, 3, 5 , 7 and so on. You can do this the other way too. `[-1,-3..]` gives you the same in negative.

The same idea works with characters too. `['a','b'..]` gives you "abcd..."and some weirdness after you reach the end of alphabet.

To give an example of this, consider the following:

```
take 30 ['a','b'..]
```

Result:

```
"abcdefghijklmnopqrstuvwxyz{|}~"
```

5.2 Laziness

Haskell is lazy programming language, but what it means?

In programming language theory, lazy evaluation, or call-by-need is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing). The sharing can reduce the running time of certain functions by an exponential factor over other non-strict evaluation strategies, such as call-by-name.

The benefits of lazy evaluation include:

- Performance increases by avoiding needless calculations, and error conditions in evaluating compound expressions
- The ability to construct potentially infinite data structures
- The ability to define control flow (structures) as abstractions instead of primitives

Lazy evaluation is often combined with memoization, as described in Jon Bentley's Writing Efficient Programs. After a function's value is computed for that parameter or set of parameters, the result is stored in a lookup table that is indexed by the values of those parameters; the next time the function is called, the table is consulted to determine whether the result for that combination of parameter values is already available. If so, the stored result is simply returned. If not, the function is evaluated and another entry is added to the lookup table for reuse.

Lazy evaluation can lead to reduction in memory footprint, since values are created when needed. However, lazy evaluation is difficult to combine with imperative features such as exception handling and input/output, because the order of operations becomes indeterminate. Lazy evaluation can introduce space leaks.

Delayed evaluation has the advantage of being able to create calculable infinite lists without infinite loops or size matters interfering in computation. For example, one could create a function that creates an infinite list (often called a stream) of Fibonacci numbers. The calculation of the n-th Fibonacci number would be merely the extraction of that element from the infinite list, forcing the evaluation of only the first n members of the list.

In the Haskell programming language, the list of all Fibonacci numbers can be written as:

```haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

6 Pure functions

Since Haskell is a purely functional language, let's discuss pure functions as one of the cornerstones of the language.

To begin with, let's recall some school math and formulate the basic definition of a function. Pure functions in Haskell are functions in mathematical sense themselves. They are descriptions of how the input value define the output value.

Hence follows an important feature of pure functions, precisely the lack of side-effects. Input value of a pure function entirely and completely defines its output value. Therefore if we pass same value as input million times, we are guaranteed that on the output we'll get the same result million times.

6.1 Declaring

Same as in other programming languages, function should first be declared. Let us do that.

```
simpleSum :: Int -> Int
```

Before the symbol :: we declared the name of the function, then follows its type.

Consider the declaration of this type:

```
Int -> Int
```

Notice an arrow. This arrow itself means that we face the pure function. The type of the only argument (in this case it's the standard type Int) is on the left side of arrow, on the right side we have an output type (same Int). The arrow may be considered as a "mental designation" to the stream of information which pass through the function from input to output, left to right.

I should remind you that pure function must have at least one argument and return something, cause that is the point of mathematical function.

Concerning amount of arguments. Of course function may take several arguments. Here is a type of function that takes three arguments.

```
Int -> Int -> Int -> Int
```

It was type signature of the function of three arguments.

6.2 Defining

Now the function needs to be defined. By the way, the function must be defined. For example in the C or C++ language we may declare the function, but not define it (provided that it is never called) and stay calm. Haskell has a stricter approach: if you declared the function, please be so kind to define it, otherwise the compiler will express it's categorical discontent.

That's why right after the declaration we write the definition:

```
simpleSum :: Int -> Int
simpleSum value = value + value
```

6.3 Calling

Now our function may be called. Let us do that with argument 4, or as we say in world of FP, apply the function to the argument 4:

```
main = putStrLn (show (simpleSum 4))
```

Result:

```
8
```

7 Algebraic data type

Haskell has one of the best implementations of algebraic data type(ADT) concept.

7.1 Definition

In computer programming, particularly functional programming and type theory, an algebraic data type is a kind of composite type, i.e. a type formed by combining other types. Two common classes of algebraic type are product types—i.e. tuples and records—and sum types, also called tagged or disjoint unions or variant types.

The values of a product type typically contain several values, called fields. All values of that type have the same combination of field types. The set of all possible values of a product type is the set-theoretical product of the sets of all possible values of its field types.

The values of a sum type are typically grouped into several classes, called variants. A value of a variant type is usually created with a quasi-functional entity called a constructor. Each variant has its own constructor, which takes a specified number of arguments with specified types. The set of all possible values of a sum type is the set-theoretical sum, i.e. the disjoint union, of the sets of all possible values of its variants. Enumerated types are a special case of sum types in which the constructors take no arguments, as exactly one value is defined for each type.

Values of algebraic types are analyzed with pattern matching, which identifies a value by its constructor or field names and extracts the data it contains.

7.2 Examples

One of the most common examples of an algebraic data type is the singly linked list. A list type is a sum type with two variants, Nil for an empty list and Cons x xs for the combination of a new element x with a list xs to create a new list:

```
data List a = Nil | Cons a (List a)
```

Cons is an abbreviation of construct. Many languages have special syntax for lists. For example, Haskell use `[]` for Nil, `:` or `::` for `Cons`, and square brackets for entire lists. So `Cons 1 (Cons 2 (Cons 3 Nil))` would normally be written as `1:2:3:[]` or `[1,2,3]` in Haskell.

For another example, in Haskell we can define a new algebraic data type, Tree:

```
data Tree = Empty
          | Leaf Int
          | Node Tree Tree
```

Here, `Empty` represents an empty tree, `Leaf` contains a piece of data, and `Node` organizes the data into branches.

8 Applications of Haskell

We have seen many powerful features of Haskell, but what we can use it for?

I consider the Computer Algebra System(CAS), that I write. The CAS works with polynomial forms, but the general purpose is work with polynomial normal forms (PNF).

Main ADT is `Formula`:

```
data Formula = C Int
             | V String
             | Add Formula Formula
             | Mul Formula Formula
             | Exp Formula Int
             deriving (Eq, Ord)
```

We have some operations for Formulas:

- arithmetic operations such as +, -, *, ^
- toPolynomial – function to conversion any polynomial form to polynomial normal form
- simplify – simplify polynomial form
- polarize – polarize PNF
- collect – collect summands and multipliers
- mapFormula – apply function to any component of thr formula
- expand – open the brackets and replace powers with multiplication