Functional Programming Using Haskell

Bruno Giao

a96544 AT alunos DOT uminho DOT pt

1 Introduction

1.1 Motivation

Functional Programming has always been an important paradigm in the field of Computer Science, both in it's theoretical field and in it's practical side.

No one can deny the importance Functional Programming has had in theoretical Computer Science and the advancements it has brought to the Art, from the understanding of the underlying algebraic and mathematical foundations of the Science, to the innovation of some functionalities and concepts we now take for granted. It is also an essential component of any Computer Science Academic Degree, to the point where Functional First is a well known and commonly used term in Academic Circles. This teaching philosophy originates due to Functional Programming's focus on functions, a concept that has been followed by students since elementary education, thus facilitating the understanding of some of the most important concepts in Computation.

But academic importance is not the only importance Functional Programming has, due to it's reliance on functions and safe, deterministic behaviour, there are certain areas that make its use imperative, namely scientific reliant jobs, such as mathematical reliant work and more recently quantum computing, but also facebook and other companies make notable use of Haskell to fulfill their needs.

1.2 Haskell

Haskell is the programming languange of choice for Functional Programming, this is due to its great power as a declarative, purely-functional language, and due to its stellar interpreter - the Glasgow Haskell Compiler.

Despite Lisp being also a very important language, one of the first programming languages in fact, Haskell is very commonly used to teach this paradigm due to it's intuitiveness and "syntatic sugar". As mentioned, Haskell is a declarative, purely-functional language that first appeared in 1990 and has greatly influenced the programming world with it's impact on Facebook and in Computer Science in Academia.

2 Functions as Contracts and Types

2.1 Functions as Contracts

One might consider functions as being a very rigid mathematical definition. In fact, in mathematics a function is defined as the following:

Definition 1 (Function). *A function f is a binary relation from a set X, called the domain of f, and Y, called the codomain of f such that:*

– *f is univalent:*

$$\forall x \in X, \forall y \in Y, \forall z \in Y, ((x, y) \in R \land (x, z) \in R) \implies y = z$$

– *f is total:*

$$\forall x \in X, \exists y \in Y, (x, y) \in R$$

Note 1. A function is called partial if it is only univalent and is a function if it is univalent **and** total.

However, this is too formal of a definition for what is truly needed, thus we can abstract what a function is to merely something that is given an 'input' and gives back an 'output', much like a contract where one would give a contracter money and be given back a built house.

2.2 Types

The keen reader may notice that this is not enough to define what we need, after all, when we give a contracter peas, they will surely not give us an house, but were we to give a friendly cook peas, they may gives us back baked peas, perhaps. Indeed, a clear notion of "Types" is required.

Thus, a function would be something that is given an 'input' that must be in accordance to the Function's Type, the function "build a house" may have input of the type "Currency" and output of type "House".

2.3 Inside the Black Box and Composition

Even though we are closer to a practical definition for functions, we still need to know how money is transformed into a house, or peas into baked peas. That's where we can be introduced to Composition.

Definition 2 (Composition). *Let f and g be functions and h a function obtained by composing f and g:*

$$h = f \circ g = f(g(x))$$

Theorem 1. *Most functions can be rewritten in such a way that they are the result of compositions.*
Those that cannot are called ***Initial Functions****.*

Going back to our example of a contract, we'd say the contract to build a house function is given a type of currency, and will go through the composition of the functions: BuildHouse after employBuilders and getMaterials.

In haskell we could define this as:

```
contractBuildHouse currency = BuildHouse (employBldrs currency) (getMats currency)
```

3 Real Function Types as Sequences

3.1 What is a sequence?

A sequence is nothing more than a set whose elements can be enumerated.

For the sake of simplicity we shall only consider as a sequence what is called an inductively defined set:

Definition 3 (Inductive Set - Russel). *Let L be a subset of A*, we say L is an inductive set if it is a non-empty partially ordered set, in which every element has a successor.*

This definition is considerably verbose and hard to grasp, so let us simplify it and laymen terms as, every element can be generated using either a base rule or a general rule dependant on other elements, and exemplifying it.

The case of Numbers The most well known sequence we can consider is that of $\mathbb{N}_0$.

Definition 4 (Natural Numbers definition).

- $0 \in \mathbb{N}_0$
- $n \in \mathbb{N}_0 \implies n + 1 \in \mathbb{N}_0$

This definition can even be expanded to a set most common for computer scientists - $\mathbb{Z}$:

Definition 5 (Integer).

- $n \in \mathbb{N}_0$
- $n \in \mathbb{Z} \implies n - 1 \in \mathbb{Z}$

These inductive definitions are of great importance, this because they tell us directly, when defining a function, what cases we must consider, specifically for natural numbers we need to consider $[0, \text{succ}]$, whereas in the Integers we must consider $[0, \text{succ}, \text{pred}]$, where succ is $n + 1$ and pred is $n - 1$.

The case of Lists Lists are a well known data-structure in Computer Science, they are extremely versatile and extremely efficient for data storing. However, are they such cases of "sequences"?
Yes, they are, let us define them:

Definition 6 (Lists). *Let us consider the set Lists where:*

- *nil* $\in$ *Lists*
- *list* $\in$ *Lists* $\implies$ *cons n list* $\in$ *Lists,* $\forall n \in$ *subtype of list*

Where nil is the function that creates the empty list, in Haskell written as []*, and cons is the function that adds an element n to the head (start) of a list, in Haskell written as n:list.*

Now let us look at a very common example of Lists.

The subcase of Strings Strings are no more no less than lists of characters. Thus all functions for general lists work for strings and almost all functions that work for strings work for lists as well.