Appreciating functional programming: A beginner's tutorial to HASKELL illustrated with applications in numerical methods

Chu Wei Lim
chuwei.lim@aostudies.com.sg

Weng Kin Ho *
wengkin.ho@nie.edu.sg
National Institute of Education
Nanyang Technological University
637616
Singapore

Abstract

This paper introduces functional programming to the numerical methods community with the aim of popularizing this programming paradigm through a deeper appreciation for function as a mathematical concept and, at the same time, for its practical benefits. The functional language HASKELL *is chosen amongst several choices because of its lazy evaluation strategy and high-performance compiler WinGHCi. We demonstrate the elegance and versatility of* HASKELL *by coding* HASKELL *programs to implement well-known numerical methods.*

1 Introduction

Functional programming is a style of programming which is an alternative to *imperative programming*; the latter being more commonly adopted in the programming community. More than just a stylistic difference, coding in a functional programming requires the programmer to put on a different mind-set. For this reason, we often refer to this new mind-set as the functional programming *paradigm*. No thinking occurs in vacuum. In line with the aims of the eJMT to focus on "all technology-based issues in all Mathematical Sciences", we introduce functional programming (the *technology* part) in close relation to numerical methods (the *mathematics* part). The main purpose of this paper is to promote functional programming paradigm to mathematicians in this community as

*Corresponding author

```
clear
n = input('Key in n: ');
value = 0;
% initialise value to 0
for i = 1:n
 value = value + i;
end
fprintf('Sum is %d. \n',value);
```

```
consecsum :: Int -> Int
consecsum 1 = 1
consecsum n = n + consecsum (n-1)
```

Figure 1: Programming styles: imperative (left) vs functional (right)

a novel way of thinking about numerical solutions of old problems. As a pleasant side effect, it is hoped that we have created here sufficient scenarios for tertiary mathematics students to explore and deepen their learning of mathematics (in the case, numerical methods) via functional programming. For this reason, our target audience would be mathematicians (and their students) who are familiar with *both* elementary numerical methods and at least one (imperative) programming language, such as MATLAB or C++.

For a quick taste of the paradigmatic difference between imperative and functional programming, let us consider the task of computing

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n.$$

Figure 1 shows how the above summation is computed in imperative style (left), and in functional style (right).

With the imperative approach, a developer writes a code that specifies the steps which the computer must take to complete the task; this often is referred to as algorithmic programming. Because of the step-by-step specification style, there is a need to track this step-to-step transition using changes in state. In the imperative program (on the left), the change in state is enacted by an increment in the counter `i`. This change in state then results in a corresponding update in the variable `value`. We say that the variable `value` is *mutable* because the updated value is stored in the same register `value` after a state change occurs in `i`.

The flow control of an imperative-style program is typically initiated by loops (e.g., for-loops `for i = 1:n ... end`, and while-loops `while (conditional) do ...`), conditionals (e.g., `if ... then ... else`), and method calls. Table 1 shows the corresponding updates in `value` in each change in state for the input `n` equals to 4.

In contrast, a functional approach involves writing the program in the form of a set of pure mathematical functions to be executed. A *pure function* (or simply, function) is just an assignment of a *unique* output to each given input. A functional programmer focuses on what information is desired and what transformations are required, and this type of programming can be said to be declarative, i.e., the programmer declares what the function is to expect as the input and what to return as the output via some assignment rule. For example, in Figure 1 the program `consecsum` is of function type