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

Weng Kin Ho
wengkin.ho@nie.edu.sg
Mathematics and Mathematics Education
National Institute of Education
Singapore 637616
Singapore

Abstract

This paper introduces functional programming to the ATCM 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. Within the functional programming paradigm, data manipulations are performed via the use of expressions that do not contain side effects. This is achieved by the use of *immutable* data (i.e., whose values cannot change once they are created) – a characteristic feature of functional languages.

As illustration, let us consider the task of computing $\sum_{i=1}^{n} i := 1 + 2 + 3 + \ldots + n$. In an imperative language (such as MATLAB), one might code it as follows:

```
clear
n = input('Give a value for n: ');
value = 0; % initialise value to 0
for i = 1:n
 value = value + i;
end
fprintf('The sum is %d. \n',value);
```

The above program contains a *mutable* variable, namely `value`; its value gets updated once a state change occurs. In addition, the code includes some instructions for the computer in the form of loops (e.g., for-loops), initialisation and increment of variables (e.g., `value`), and updating of states (e.g., `i`).

In a functional language (such as HASKELL), the same task can be accomplished simply by the *function-like* program:

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

The key difference here with the functional approach is that we make use of expressions to manipulate data. Crucially, we are expressing what we want to be done via expression transformations, much like the use of rewriting rules in a rewriting system. For instance, when `consecsum` operates on the input `4`, the above code instructs the computer to perform the following expression transformations:

```
consecsum 4 = 4 + consecsum 3
            = 4 + 3 + consecsum 2
            = 4 + 3 + 2 + consecsum 1
            = 4 + 3 + 2 + 1
```

Many popular articles and web-sites that favour functional programming over imperative programming usually advertise the following advantages: (1) immutability, (2 explicit state management, (3) side effect programming through data transformation, (4) expressions versus statements, and (5) higher level functions (i.e., those that take function as argument to transform data). However, these technical jargons are not immediately understood by people who have little or no experience in programming, let alone functional programming.

The purpose of this paper is to introduce functional programming to a certain target reader – one who possesses some basic knowledge about functions as a mathematical concept but no prior experience in programming. To do so, we emphasise that functions are *first class citizens* in functional programming. The functional language we choose here is HASKELL in which syntactic representation of programs bears strong resemblance to that of a function. More precisely, the code which implements a functional program `f` that takes in as input element from the data set `A` and returns an output element in the data set `B` always begins with the 'domain-codomain' type declaration:

```
f :: A -> B
```

Drawing on the reader's familiarity with functions, we argue that the cognitive overhead for acquiring a functional programming language such as HASKELL is relatively light compared to acquiring an imperative one.

Additionally, we feature some attractive benefits offered by functional programming paradigm:

1. Code is easier to read and write.
2. Code is easier to test when most of the programs involved are just 'small' functions.
3. Better programming discipline is enforced by the functional programming paradigm.

To advertise the aforementioned advantages by coding familiar numerical methods, such as root-finding and quadratures, in the functional language HASKELL. We urge readers who compare these functional programs (see Section 4) with their imperative counterparts.

As this paper is meant to be an introductory tutorial, it is far from being comprehensive. For a systematic introduction to HASKELL, we refer the reader to [2], and for a discussion which is focused more on the functional programming paradigm itself, [3]. Here are some technical reminders: (a) Install WinGHCi (Glasgow Haskell Compiler's interactive environment in Windows) from the official HASKELL platform (`https://www.haskell.org/platform/`), (b) write the codes using NOTEPAD and save them in the HASKELL format with the `.hs` extension, (c) compile these `Haskell` files using WinGHCi, and (d) run the program by applying the function on valid arguments.

2 Functions

2.1 As a mathematical concept

In mathematics, the notion of a function allows one to formalize the assignment of an output to a given input. Indeed this assignment is a special kind of relation between the domain and the co-domain that one have in mind.

Definition 1 (Relation) *A relation R between sets A and B is just a subset R of the cartesian product of A and B, i.e., $R \subseteq A \times B$. An instance of the relation R is denoted by $a\ R\ b$, which reads "$a \in A$ is R-related to $b \in B$".*

A function is a special type of relation. Precisely, we have that:

Definition 2 (Function) *A function f from the set A to the set B, denoted by $f : A \longrightarrow B$, is just a relation between A and B such that for each $a \in A$, there exists a unique $b \in B$ such that $(a, b) \in f$. In this case, an instance of $(a, b) \in f$ is denoted by $f(a) = b$.*

The fundamental idea of a function, f, is that of a *black-box* which assigns to a given input x a unique output $f(x)$ (see Figure 1). Of course, this is just the *extensional* view of functions,

Figure 1: Black-box view of a function

i.e., one can only observe how a function behaves by inspecting *externally* what comes out as output given its input. You are *not* allowed to open up the black-box to inspect what goes on in there. But there is also the other view – the *intensional* one – which pays attention to the *internal* process of how one produces the required output given its input.

The extensional perspective is crucial, for instance, in the definition of equality of functions.

Definition 3 (Equality of functions) *Two functions $f,\ g : A \longrightarrow B$ are* equal *if for all $a \in A$, it holds that $f(a) = g(a)$.*

So in particular, the constant function $f : \mathbb{R} \longrightarrow \mathbb{R}$ that assigns 1 constantly to any input x must therefore be equal to the function $g : \mathbb{R} \longrightarrow \mathbb{R},\ x \mapsto \sin^2 x + \cos^2 x$ because they agree on all arguments, despite their very different-looking intensional definitions.

2.2 As a computing concept

Almost naturally and intrinsically, the notion of a function is analogous to that of a machine which takes in some inputs and produces some outputs. In a more restrictive case, we can imagine that this machine performs some computation or treatment of input data and returns some data as output. Thus, the functional concept lies in the heart of computing itself in that it illustrates the fundamental role a *computer* performs. In an even more restrictive sense, we can think of a computer program or algorithm as a function that is devised to meet a certain input-output specification.

In object-oriented languages, this input-output view manifests itself as object-method relationships, i.e., a method is regarded as a procedure associated with a message and an object, where an object comprises data and behaviour, forming the interface that an object presents to the outside world. In particular, data is represented as properties of the object and behaviour as methods. For imperative languages, the program flow is controlled by changing states, e.g., to assign new value to a local variable once a state undergoes change.

In functional programming, functions are first class citizens. Not only can programs can be viewed as functions which receive input and return output, but programs themselves can be presented as inputs to other programs of higher-order function types and be used to produce meaningful outputs.

3 Functional programming with HASKELL

3.1 What's HASKELL

The programming language HASKELL is a *purely functional language*, and so all computations are done via *evaluation* of *expressions* (also, called *syntactic terms*) to *values* (i.e., special expressions that we can regard as 'answers'). Every expression (and hence every value) is assigned a (data) *type*; types can be thought of as sets of expressions.

3.2 Values, expressions and types

In this paper, we only make use of the following *ground types*:

1. `Int`, the integer types containing the integers $\ldots, -2, -1, 0, 1, 2, \ldots$;

2. `Bool`, the boolean types containing the boolean values `true` and `false`;

3. `Double`, the double precision floating point numbers.

Associating of a value with its type is called *t*yping. Some instances of typing given below:

```
 1   :: Int  true :: Bool  0.1  :: Double
```

In each of the above instances, we read "`::`" as "is of type". Of course, expressions in general can be assigned their types once the constituent terms in these expressions have been assigned their respective types. For instance, if we have the *typing assignment*

```
n :: Int
```

then the expression `n + 1` can be assigned the valid type:

```
n + 1 :: Int
```

Another example is the conditional expression "`if ... then ... else ...`". If we assign the types below:

```
c :: Bool  t1, t2 :: A
```

then the following assignment is valid:

```
if c then t1 else t2 :: A
```

3.3 Building types

We can build new types out of basic ground types. In this fragment of HASKELL we are working with, the data types that can be built from the basic ground types using a finite number of formation rules are presented in the *Backus-Naur form* (BNF, for short) below:

```
A := Int | Bool | Double | (A,A) | A -> A | [A]
```

What the above BNF means is that a data type `A` is exactly one of these:

1. an integer type, or
2. a boolean type, or
3. a double precision floating point number, or
4. a *product* type, or
5. a *function* type, or
6. a *list* type

We now explain which values belong to the product, the function and the list types respectively by using examples.

Example 4 (Product type) *The type constructor* `(-,-)` *takes a pair of data types* `A` *and* `B` *to form the* product type `(A,B)`*. This is very similar to the case of näive set theory where two sets A and B together form the cartesian product, $A \times B$. As an example, we take the product type* `(Int,Bool)` *which contains as data ordered pairs whose first component is of type* `Int` *and whose second component of type* `Bool`*. A typical instance of the product type* `(Int,Bool)` *is given by the ordered pair:*

```
(2,True) :: (Int,Bool)
```

Example 5 (Function type) *Central to the functional programming languages is the facility to produce function types. More precisely, given two data types* `A` *and* `B` *one forms the* function type `A -> B`*. The data which inhabit the function type* `A -> B` *are programs that take in an input of type* `A` *and return an output of type* `B`*. For instance, the first projection map* `fst` *that takes a data pair,* `(x,y)`*, of type* `(X,Y)` *and returns the first component* `x` *of type* `X` *can be implemented by the following self-explanatory script:*

```
fst :: (X,Y) -> X
fst (x,y) = x
```

It is possible to build higher order functionals using the function type constructor `->`*. For instance, the addition operation* `add` *that takes in two summands of integers to produce their sum can be implemented using the following code:*

```
addint :: Int -> Int -> Int
addint x y = x+y
```

Here the higher-order function type `Int -> Int -> Int`*, under the convention of right associativity, is the same as* `Int -> (Int -> Int)`*. Indeed given an input of* `x :: Int` *the function* `addint` *returns as output a program, i.e.,* `addint x`*, of function type* `Int -> Int`*. This is because* `addint x` *is waiting to accept an integer argument,* `y::Int`*, and returns as output* `x+y` *which is of type* `Int`*.*

To understand the higher-order function type better, here is another illustration. Consider the evaluation map which takes in as first argument a function `f::Int -> Int`*, followed by a second argument an integer* `n :: Int`*, and evaluates the function* `f` *on* `n`*. Then we can implement the evaluation map* `eval` *as follows:*

```
eval :: (Int -> Int) -> Int -> Int
eval f n = f n
```

Example 6 (List type) *A very versatile data type constructed out of a given type* `A` *is the* list type, `[A]`*. An instance of a datum of type* `[A]` *is a list, i.e., sequence of elements each of type* `A`*. We can think of the list type* `[A]` *as one that is defined recursively by* `Null | (A,[A])`*, where* `Null` *is the null data type that contains only non-termination. This means that an instance of* `[A]` *is* either *the empty list* `[ ]` *that contains nothing* or *a list,* `(a:as)`*, which is headed by the datum* `a::A` *and tailed by another list* `as::A` *(read as a's). Because of the recursive nature of list types, programs that manipulate lists can often be coded by recursion. For instance, the program* `intl` *which takes in two lists, e.g.,* `[x0,x1,x2,x3,...]` *and* `[y0,y1,y2,y3,...]`*, and interleaves their elements to produce* `[x0,y0,x1,y1,x2,y2,...]` *can be defined recursively as follows:*

```
intl :: [A] -> [A] -> [A]
intl [] l = l
intl l [] = l
intl (x:xs) (y:ys) = x:y: intl xs ys
```

Simply put, the program `intl` *takes the heads of the two input lists and interleaves them in the expected manner and then repeats the procedure to the tails of the two input lists.*

Apart from product, function and list constructors, HASKELL has the facility to define user-declared data types. There are essentially two ways to do so.

Firstly, we can create *type synonyms*, i.e., names given conveniently for a constructed type. For instance, we may use the type synonym `Pairint` to denote the constructed product type of a pair of `Int` types by declaring that:

```
type Pairint = (Int,Int)
```

Subsequently, one may then use the name `Pairint` in place of `(Int,Int)` within the same environment of this type declaration. For instance, we can define the first projection map as follows:

Example 7 (Type synonym) `fst :: Pairint -> Int`

```
fst (x,y) = x
```

Secondly, we may construct new types by using a *recursive type equation.* In general, a recursive type equation is of the form:

```
data A = F(A)
```

where `F` is a type expression that involves existing type constructors such as product, function and list constructors. Notice that the data type `A` appears on both sides of the equation and is to be interpreted as the data structure which satisfies the recursive type equation. Here are two examples.

Example 8 (Type equation: Day type) *We can define a new data type whose values are the seven days of a week.*

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
deriving (Show)
```

`Day` *is defined using the separated sum constructor* `|`*, which can be taken to mean "or" for data types. In this definition,* `Day` *is* not *defined using recursion since the type expression on the right hand side does not involve* `Day`*. Note that the command* `deriving (Show)` *allows for appearance of constant* HASKELL *expressions of that declared type, i.e., a value instance of* `Day` *appears as a printable string value.*

Example 9 (Type equation: Tree type) *A tree is a data type commonly encountered in programming contexts. The data type* `Tree` *of binary branching trees can be defined by the following recursive type equation:*

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
deriving (Show)
```

Here `a` *stands for a given data type. So, a binary branching tree associated with the type* `a` *is either a leaf whose tag is an instance of type* `a` *or branching into a left tree structure and a right tree structure.*

Clearly, this recursive definition of `Tree` *entails that the collection of binary branching tree structures associated to type* `a` *to be an inductive set, i.e., reasoning about trees relies on structural induction as defined by the above recursive type equation. For example, we can script a program* `ht` *that calculates the height of a binary branching tree, i.e., the number of edges on the longest path between the root node and a leaf in the given tree. The height program is given below:*

```
ht :: Tree -> Int
ht Leaf a = 0
ht Branch (Tree a) (Tree a) = 1 + max(ht Tree a,ht Tree b)
```

4 Implementing numerical methods

It may appear that the computational power of the small fragment of HASKELL (which we described in the preceding section) is very limited since it merely consists of three ground types (`Int, Double, Bool`) and three type constructors (`(-,-), - -> -, [-]`). We now demonstrate that despite its structural leanness this fragment of the language is surprisingly expressive. In particular, we show that within this small fragment of HASKELL we already can perform most of the useful real-number computations that one usually encounters in an introductory course on numerical methods. The reader is encouraged to compare our functional programs that implement these numerical methods with their imperative counterparts. It is hoped that through this comparison, readers can decide for themselves the merits of functional programming as an alternative programming paradigm to imperative programming. Readers who need to refresh their memory concerning numerical methods may refer to [1] – this book also contains imperative MATLAB codes for implementing all the numerical methods mentioned herein.

Since we do not want to deal with exact real arithmetic (i.e., real number computation with arbitrary precision) in this paper, we conveniently use `Double` for real number data type (double-precision) to implement our calculations of real numbers. Readers interested in exact real arithmetic may refer to [4].

For instance, here are some basic arithmetic operations:

```
add :: Double -> Double -> Double
add x y = x + y

sub :: Double -> Double -> Double
sub x y = x - y

mult :: Double -> Double -> Double
mult x y = x*y

divd :: Double -> Double -> Double
divd x y = x/y
```

```
powr :: Double -> Double -> Double
powr x y = x**y
```

Every closed bounded interval $[a, b]$ can be coded as a pair of double-precision floating point numbers. In HASKELL, we thus define the `Interval` data type as the *Cartesian product* of `Double` with itself:

```
type Interval = (Double,Double)
```

A closed interval $[a, b]$ can be coded as a pair of real numbers `(a,b)` – the parentheses used here can be confusing; it denotes an instance of a pair of real numbers rather than an open interval.

Recall that programs of function type are first-class citizens in functional programming. Our ensuing discussion only concerns the set, $\mathcal{C}(\mathbb{R})$, of real-valued functions of a single real variable. The data type we use to code this set is `Function` which is defined to be the function space from `Double` to itself, i.e.,

```
type Function = (Double -> Double)
```

4.1 Point approximation

One of the ways to compute a real number, α, is by producing a sequence of real numbers, $\{a_n\}_{n=0}^{\infty}$, that converges to it. We refer to this computational approach as the *point approximation* approach. The main idea here is to code a sequence of (distinct) real numbers as a list of double-precision floating numbers, `[Double]`. Given a list of `Double` data

```
a0, a1, ..., an, ...
```

which represents a convergent sequence $\{a_n\}_{n=0}^{\infty}$, we output the first element `an` if the next element is sufficiently close to it in a relative sense, i.e.,

$$\left| \frac{a_n}{a_{n+1}} - 1 \right| < \epsilon,$$

where $\epsilon > 0$ is the required precision. This type of precision requirement has the advantage over that of taking difference of consecutive terms, i.e.,

$$|a_n - a_{n+1}| < \epsilon,$$

when the magnitudes of a_n and a_{n+1} are very small to begin with.

The following program can be employed to display the first element of the list which satisfies the precision requirement `eps`:

```
relerr :: Double -> [Double] -> Double
relerr eps (a:b:xs)
 | abs(a/b - 1) < eps = a
 | otherwise          = relerr eps (b:xs)
```

We illustrate this point approximation approach using four well-known numerical methods, namely, (1) the fixed point iteration method, (2) the Newton-Raphson's method, (3) numerical differentiation, and (4) numerical integration.

4.1.1 Fixed point iteration

Let f be a continuous function defined on some closed bounded interval I and $x_0 \in I$ be given. Then applying f repeatedly yields the following trajectory of iterates:

$$x_0, f(x_0), f^2(x_0), \ldots, f^n(x_0), f^{n+1}(x_0), \ldots$$

Such a trajectory can be produced by the following program:

```
iterates :: Double -> Function -> [Double]
iterates a f = a : (iterates (f a) f)
```

Suppose further that the sequence $\{f^n(x_0)\}_{n=0}^{\infty}$ converges, i.e., $\lim_{n\to\infty} f^n(x_0) = \alpha$. Then by the continuity of f and the fact that a convergent sequence and any of its subsequence share the same limit, we have

$$f(\alpha) = f\left(\lim_{n\to\infty} f^n(x_0)\right) = \lim_{n\to\infty} f^{n+1}(x_0) = \lim_{n\to\infty} f^n(x_0) = \alpha.$$

In other words, α is a *fixed point* of f.

Certain natural conditions are sufficient to guarantee that the trajectory of f, starting with the seed x_0, is convergent. Simple topological arguments justify that the continuous image, $f(I)$, of a closed bounded interval, I, is again a closed bounded interval. Now suppose that $f(I) \subseteq I$. Repeated applications of f then yield the following descending family of closed bounded intervals, together with the membership of each iterate:

$$\begin{array}{ccccccccccccc}
I & \supseteq & f(I) & \supseteq & f^2(I) & \supseteq & \ldots & \supseteq & f^n(I) & \supseteq & f^{n+1}(I) & \supseteq & \ldots \\
\rotatebox{90}{$\in$} & & \rotatebox{90}{$\in$} & & \rotatebox{90}{$\in$} & & & & \rotatebox{90}{$\in$} & & \rotatebox{90}{$\in$} & & \\
x_0 & \supseteq & f(x_0) & \supseteq & f^2(x_0) & \supseteq & \ldots & \supseteq & f^n(x_0) & \supseteq & f^{n+1}(x_0) & \supseteq & \ldots
\end{array}$$

Suppose that the length of these intervals tend to 0, i.e.,

$$\lim_{n\to\infty} \ell(f^n(I)) = 0.$$

Then by the Heine-Borel Theorem, it follows that there is a unique real number α such that

$$\alpha \in \bigcap_{n=0}^{\infty} f^n(I),$$

and thus $\lim_{n\to\infty} f^n(x_0) = \alpha$.

One such instance for which the length of the subintervals $f^n(I)$ converges to 0 occurs when f is a *contractive mapping*, i.e., there exists a constant $c \in [0, 1)$ such that

$$\forall x,\ y \in \mathbb{R}.\ |f(x) - f(y)| < c|x - y|.$$

Given sufficient conditions that ensure the convergence of the trajectory of f hold, with seed value x_0, we can implement the fixed point iteration via the following algorithm:

```
fpi :: Double -> Double -> Function -> Double
fpi eps a f = relerr eps (iterates a f)
```

4.1.2 Newton-Rapshon method

A special instance of fixed point iteration is the famous Newton-Raphson iteration scheme for solving numerically the equation

$$f(x) = 0,$$

where f is a differentiable function. The iteration scheme, involving the successive constructions of tangents to the curve $y = f(x)$ and their intersections with the x-axis, is given by

$$x_{n+1} = g(x_n), \qquad n = 0, 1, 2, \ldots,$$

where $g(x) = x - \frac{f(x)}{f'(x)}$. Denote the derivative of the given function `f` by `df`. Given a non-stationary seed value `x`, the next iterate may be computed by using the following program:

```
newtonit :: Function -> Function -> Double -> Double
newtonit f df x = x - (f x)/(df x)
```

Making use of the `fpi` program, we implement the Newton-Raphson method as follows:

```
newton :: Double -> Double -> Function -> Function -> Double
newton eps a f df = fpi eps a (newtonit f df)
```

4.1.3 Numerical differentiation

A preliminary estimation of the gradient of the tangent to the curve $y = f(x)$ at the point x is the first order quotient given by the gradient of the secant line joining the points $(x, f(x))$ and $(x+h, f(x+h))$, i.e.,

$$\frac{f(x+h) - f(x)}{h},$$

where h is a 'small enough' quantity such that the function f is defined over the interval $[x, x+h]$. This is easily calculated by the following program:

```
secant :: Double -> Function -> Double -> Double
secant h f x = (f (x+h) - f x)/h
```

Now limiting h to 0 by consecutively halving the initial difference of h, and producing the corresponding secant value:

```
secantseq :: Double -> Function -> Double -> [Double]
secantseq h f x = (secant h f x) : secantseq (h/2) f x
```

To calculate the derivative of f at x, it suffices to numerically demand that the consecutive secant values are relatively close by applying the program below:

```
easydiff :: Double -> Function -> Double -> Double
easydiff eps f x = relerr eps (secantseq 1 f x)
```

Note that we have set the initial value of h to be 1 as above.

4.1.4 Numerical integral

The (trapezoidal) area under the line segment joining $(a, f(a))$ and $(b, f(b))$ provides a first approximation of the definite integral of the continuous function f over the interval $[a, b]$. This common numerical method is called the *simple trapezoidal rule.* For instance, if $f(x) = x^2$ for $x \in [1, 2]$, then the trapezoidal area under the line segment $y = 3x - 2$ over $[1, 2]$ estimates the value of the definite integral $\displaystyle\int_1^2 x^2\ dx$ (see Figure 2).

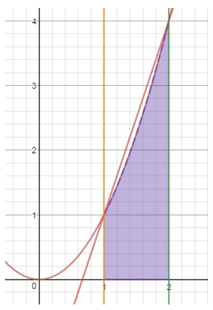

Figure 2: Simple trapezoidal rule for estimating $\displaystyle\int_1^2 x^2\ dx$

This first estimate, $\frac{1}{2}(b-a)(f(a)+f(b))$, produced by the simple trapezoidal rule can be calculated using `trap`:

```
trap :: Function -> Interval -> Double
trap f (a,b) = (f(a)+f(b))*(b-a)/2
```

The idea for calculating the definite integral is to see it as the limit of the sum of trapezia obtained by a certain refinement procedure which we illustrate below. From the first estimation of $\displaystyle\int_a^b f(x)\ dx$ produced by the simple trapezoidal rule on $[a, b]$, we may then produce the second estimation of the definite integral by applying the following procedure:

1. Bisect the interval $I_0 := [a, b]$ into two equal sub-intervals $I_{10} := [a, c]$ and $I_{11} := [c, b]$, where $c = \frac{a+b}{2}$.
2. Apply the simple trapezoidal rule to estimate:
   (i) $\int_a^c f(x)\ dx$ over the interval I_{10}; and
   (ii) $\int_c^b f(x)\ dx$ over the interval I_{11}.
3. Add the two estimates in (2) to obtain the second estimation of $\displaystyle\int_a^b f(x)\ dx$.

We can think of the first application of Step (2) as a branching at Level 1 that yields leaves, i.e., $\int_a^c f(x)\ dx$ arising from the left half interval and $\int_c^b f(x)\ dx$ arising from the right half interval respectively. To yield the third estimate, we apply the above procedure, i.e., Steps (1)–(3), to estimate each of the aforementioned two definite integrals. More precisely, there are

2^2 trapezium each estimating the definite integrals over the 2^2 definite integrals at Level 2. In general, the nth estimate is given by the total area of all the 2^n trapezia arising from iteratively applying Step (2) at Level n.

In order to total up the area of these 2^n trapezia, the following componentwise-addition of two lists of real numbers comes in handy:

```
addlist :: [Double] -> [Double] -> [Double]
addlist (x0:xs) (y0:ys) = (x0 + y0): addlist xs ys
```

The iterative procedure we have illustrated yields a sequence of approximations (making use of more and more trapezia), and this sequence can be produced by the following program:

```
integralseq :: Function -> Interval -> [Double]
integralseq f (a,b) = (trap f (a,b)):
                       addlist (integralseq f (a,c))
                               (integralseq f (c,b))
                                where c = (a+b)/2
```

Lastly, we rely on the usual technique of picking up the first term of the above sequence which is relatively close to the next one, up to the given precision requirement.

```
integrate :: Double -> Function -> Interval -> Double
integrate eps f (a,b) = relerr eps (integralseq f (a,b))
```

4.2 Interval approximation

Another approach of computing a real number, α, is by computing an decreasing family, $\mathcal{C}$, of closed intervals $\{I_n\}_{n=0}^{\infty}$, i.e.,

$$I_0 \supseteq I_1 \supseteq I_2 \supseteq I_3 \supseteq \ldots I_n \supseteq I_{n+1} \supseteq \ldots$$

such that the lengths of the intervals converge to 0 and $\alpha \in \bigcap_{n=0}^{\infty} I_n$.

The key idea here is to set a precision requirement on the calculation of α based on the length of the approximating interval. Given a descending family of closed intervals $\{[a_n, b_n]\}_{n=0}^{\infty}$ coded as

`[(a0,b0),(a1,b1),(a2,b2),...]`

select the first interval `(an,bn)` such that the endpoints are close to each other in the usual relative sense, i.e.,

$$|a_n/b_n - 1| < \epsilon,$$

where the positive quantity, ϵ, is the user-set precision requirement. This can be achieved by the program below:

```
relerrint :: Double -> [Interval] -> Double
relerrint eps ((a,b):xs)
 | abs(a/b - 1) < eps = a
 | otherwise          = relerrint eps xs
```

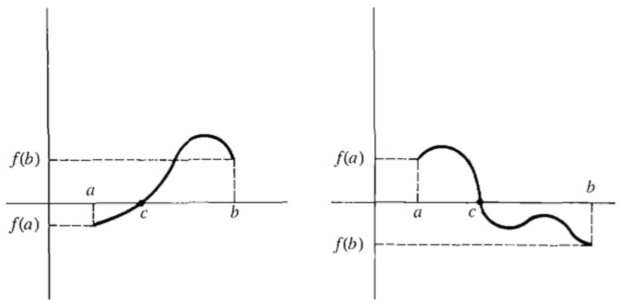

Figure 3: $[a,b]$ is f-good

We deal with numerical solution of equations here. For convenience, we assume that f is a continuous function defined on $[a,b]$ such that f has exactly one real zero, α, in $[a,b]$.

Definition 10 (f-good interval) *An interval $I=[a,b]$ is f-*good *if f experiences a change of sign over it, i.e., $f(a)f(b)\leq 0$.*

We can implement f-goodness of an interval as follows:

```
good :: Function -> Interval -> Bool
good f (a,b) = ((f a)*(f b) <= 0)
```

4.2.1 Bisection method

Consider any descending chain, $\mathcal{C}$, of f-good intervals ordered by reverse inclusion:

$$[a_0,b_0]\supseteq[a_1,b_1]\supseteq[a_2,b_2]\supseteq\ldots[a_k,b_k]\supseteq[a_{k+1},b_{k+1}]\supseteq\ldots,$$

where $\lim_{k\to\infty}[a_k,b_k]=0$. By the Heine-Borel theorem, $\bigcap_{i=0}^{\infty}[a_i,b_i]=\{\alpha\}$. In other words, $\{\alpha\}$ is the supremum of the descending chain $(\mathcal{C},\supseteq)$.

For each given f-good interval $[a,b]$, its left (respectively, right) half sub-interval is $[a,c]$ (respectively, $[c,b]$), where $c=\frac{a+b}{2}$. The bisection method selects the f-good subinterval of these two sub-intervals; in the event when both sub-intervals are f-good, then the left one is always selected.

```
goodhalf :: Function -> Interval -> Interval
goodhalf f (a,b) = let c = (a+b)/2 in
                   if (good f (a,c)) then (a,c) else (c,b)
```

Based on the bisection method, we output the desired descending chain of f-good subintervals in the form of a list of intervals, beginning with the input f-good interval $[a_0,b_0]$ using the algorithm below:

```
bisectseq :: Function -> Interval -> [Interval]
bisectseq f (a,b) = (a,b): bisectseq f (goodhalf f (a,b))
```

Lastly, we set an error of tolerance `eps` such that as soon as the relative ratio, a/b of the endpoints of an approximating interval $[a,b]$ differs from 1 by this set error then the right endpoint of this interval is displayed.

```
bisect :: Double ->  Function -> Interval -> Double
bisect eps f (a,b) = relerrint eps (bisectseq f (a,b))
```

4.2.2 Linear interpolation

For each given f-good interval $[a, b]$, the line which interpolates between the points $(a, f(a))$ and $(b, f(b))$ cuts the x-axis to form a better estimate, c, for α, the zero of f, where

$$c = \frac{af(b) - b(a)}{f(b) - f(a)}.$$

The HASKELL version of the saying the above is:

```
linecut :: Function -> Interval ->  Double
linecut f (a,b) = (a*f(b)-b*f(a))/(f(b)-f(a))
```

This 'cut-point' c creates the left 'half' subinterval (a, c) and right 'half' subinterval (c, b). Like in the bisection method, we can select the f-good 'half' subinterval of these two:

```
linehalf :: Function -> Interval -> Interval
linehalf f (a,b) = let c = linecut f (a,b) in
                   if (good f (a,c)) then (a,c) else (c,b)
```

Again, we create a descending family of f-good intervals relying on the linear interpolation method.

```
linehalfseq :: Function -> Interval -> [Interval]
linehalfseq f (a,b) = (a,b) : linehalfseq f (linehalf f (a,b))
```

Setting the precision requirement yields the desired algorithm:

```
linint :: Double -> Function -> Interval -> Double
linint eps f (a,b) = relerrint eps (linehalfseq f (a,b))
```

5 Conclusion

This paper is a beginner's tutorial to functional programming via HASKELL. We illustrate the simplicity of codes and the versatility of the functional style of programming by implementing well-known numerical methods in HASKELL.

References

[1] Ang, K. C. *Numerical Mathematics with MATLAB*, Prentice-Hall, 2009.

[2] Bird, B. *Introduction to Functional Programming using Haskell*, Prentice-Hall Series in Computer Science, Prentice-Hall, 1998.

[3] Bird, B. *Thinking Functionally with Haskell*, Cambridge University Press, October 2014.

[4] Ho, W. K. Exact Real Calculator for Everyone. In W.-C. Yang , M. Majweski, T. Alwis, and I. K. Rana, (Eds.) *Proceedings of the 18th Asian Technology conference in Mathematics* (pp. 1-15). Bombay, India: ATCM, December 2013.