Functional Programming Design Patterns for Object Oriented Programmers - A Brief Overview

I. Introduction

The functional programming (FP) paradigm has seemed to be, for quite some time, a subject mostly reserved to academic settings and researchers interested in the intersection of mathematics and computing. Little would be talked about it among software developers, and languages would provide little to no native support for FP constructs. This scenario has been changing, and in recent years, major recognition and adoption of FP paradigm has been seen on mainstream languages and frameworks.

On Java JDK 8, released in March 2014, lambdas were introduced [1]. These are, as described in the release notes, “instances of single-method interfaces” [1]. On June 2015, ECMAScript 6, Javascript’s standard, defined arrow functions [2], allowing anonymous functions, or functions with no associated identifiers, to be passed as arguments. Although Java and Javascript might refer to these newly introduced features with jargons appropriate to the paradigms and features each already supports, both features accomplish the same, and we will go over such details in subsection III-B.

Newer languages and frameworks went beyond introducing FP language constructs and embued its paradigms in their design and native capabilities. React, for instance, is admittedly inspired by FP paradigms [3]. One clear indication of this fact is how components are decoupled from states just like functions in FP should be stateless and should not produce or be influenced by side-effects. Scala [4] and Kotlin [5], for instance, provide native support for higher-order functions (see III-C), a functional programming construct that is often not supported in “traditional” imperative languages like C.

Given these examples, it should be clear how FP has become almost ubiquitous among programming languages, and how, at some point, even a software developer that is completely oblivious to the notion of FP has already probably leveraged some of its patterns to achieve something that otherwise require a convoluted, unsafe, or inefficient solution. On the other hand, a software developer which understands FP and its patterns can benefit from its capabilities to achieve a significantly greater degree of conciseness, safety, and modularity that cannot be easily achievable otherwise.

In this document, I will start by introducing some basic theoretical foundations that should foster the reader’s intuition and allow for later conceptual connections to be made. Next, we will go over some terminology and functional patterns that should suffice for the reader to recognize functional patterns in real code. Finally, I will illustrate usage of functional patterns in real-world, non-functional code to demonstrate how its patterns can be used to improve non-functional design and how FP ultimately manifests itself as a different way of tackling computational problems.

II. An Overview on Computational Models

A. Semantic and Representation

To begin this section, I ponder a classical question trivial to computer scientists: what is a computer? Whenever I ask this question or a similar question to my computing science peers, I get “a computer is a Turing Machine (TM)” as an answer. That answer is correct, but it is not the only possible answer. As we will see, computers are ore precisely defined in terms of what they are capable of doing rather than what they are, and this might imply in vastly different approaches that attempt to capture the notion of computation. Let’s briefly explore that concept and see how that lead us into functional programming.

As stated in [6], a problem is computable iff it is Turing computable. Furthermore, a Turing computable problem is a problem that can be solved by a TM [7]. Let’s disregard the notion of efficiency here and assume we have infinite time and space for computing so we can focus on the general idea of computability. The takeaway here, whatsoever, is that computable problems is defined in terms of TMs. This is an important observation for the point I will soon make.

A TM is defined as a device that is capable of carrying out one among a finite set of instructions at a time whose instructions involve transitioning among machine states, moving right or left along an infinitely long one-dimensional tape, and reading and writing to and from that same tape. Note that this is an overly simplistic definitions for the sake of this discussion, and is based off [7], which provides a more rigorous mathematical definition. This definition does not depend on any other notion of computers or computations, and is effectively capable of providing the field of computing with the necessary axiomatic foundations for expanding our knowledge on computation. But why is a TM the defining concept of a computer?

When Turing firts defined a TM, it did not intend to satisfy any more primitive axioms or conditions which attempted to define a computer. Rather, the TM attempted to capture the primitive operations which permitted the computation of all problems deemed to be computable, and this definition has managed to withstand the trial of time. The TM is intuitive and relatable to human’s capability of computing problems. Furthermore, all problems known to be computable could be computed by TMs, and all problems known to be uncomputable could not be solved by any TM [7]. Furthermore, modern-day computers build in accordance to the von Neumann architecture since the EDVAC were inspired by the notion of finite universal TMs [8]. All these reasons seem to explain why TMs are deemed as the *de facto* definition of a computer.

Note, in the previous paragraph that the motivation behind the definition of a TM are problems deemed to be computable, and without them, the TM would make no sense. In that manner, even though the definition of a TM does not depend on any more primitive notion of computers or computation, it is relevant as effect of its ability to solve problems deemed to be computable. If the TM was not capable of solving all problems deemed to be computable, it would likely not be considered a computer at all. Note that this scenario wouldn't cause the definition of a TM to be false. Rather, its operations would only allow it to solve a subset of the problems deemed to be computable. But how good is a computer that can't solve problems that we know we can compute? Thus, a TM only makes sense given the existence of such computable problems.

Now, you might have noticed something. I have first mentioned that computable problems are defined in terms of a TM, and even though the definition of a TM precedes the definition of a computable problem, it only makes sense given the existence of computable problems. This might seem as a circular definition, but in reality, this is the duality between semantics and representation. For one to become aware of an idea, one must first capture it through some definition. At the same time, a lexical definition only makes sense in light of a semantically valid idea.

Another example where this phenomenon is clearer is when we look at Pythagora's theorem. Without some representation like Pythagora's equation, the idea captured by the mathematical equation is so vague we might not even be aware of its existence. On the other hand, if the idea was false, the formula would have no useful meaning. And even though we tend to represent Pythagora's theorem through the classical equation $c^2 = a^2 + b^2$, many other equivalent representations exist. Now, we can represent Pythagora's theorem through different representations, but what if we could grasp the notion of computability through other representations? That's where Lambda Calculus comes in.

B. *Different Representation of Computers*

Lambda Calculus was introduced in the form as we know it by Alonzo Church as a means to construct a formal system which could define and reason about mathematical objects [9]. This formal system defines every mathematical object in terms of functions, including numbers themselves. It also defines certain conversion rules (or substitution rules) which allows derivations to be performed (see section 4.2 on [7] for details on the definitions of numbers in λ-calculus and on the conversion rules).

In λ-calculus, one calculates a function on an integer by applying those conversion rules until a function equivalent to the resulting integer is produced. Thus, in some sense, if you have a λ-function, you can "effectively calculate" its result when applied on an integer by performing a finite number of conversions. Conversely, Church defines a function on positive integers to be "effectively calculable" if it can be defined as a λ-function [6], which implies that it should take a finite number of steps to arrive at its result. Furthermore, through the Church-Turing thesis, Allan Turing also proved that every "effectively caculable" function is also computable, and vice-versa [10].

Note how each of these terms were defined by their respective constructs: "Turing computable" are problems that can be computed by TMs, and "effectively calculable" are functions that can be λ-defined. In that manner, at the same time that each term is equivalent, they are independent of each other and are defined without requiring the other one to exist.

Similar as to how they are independent, they are also equivalent in that they capture the same idea, but through different representations. It could be said that Turing takes on a more "procedural" approach on computability, perhaps mimicking human's computational process, while Church takes on a more mathematical approach.

Although there have been lots of details on both computational models that have been left out, a reader interested on the topic is encouraged to look at the references in this paper look at the references of these references, and deepen their understanding about these computational models. Nonetheless, my main goal behind this theoretical background was not to thoroughly cover these computational models, but to provide the reader with a perspective on this duality which constantly manifests itself through FP. Thus, have in mind that even though we will cover applications of FP in the context of imperative OOD, you can think in terms of FP as far as possible until you reach the primitives upon which your solution depends on which are defined in terms of imperative programming.

C. *Imperative VS Declarative Languages*

Now that we covered TM and λ-calculus computational models, I invite you to think of a *procedural* language like C as a realization of a TM. Think of how it describes the *procedures* to solve a problems similar to how a TM describes the procedures to compute a problem, except that C does so at a higher abstraction level. This similarity is not a coincidence. The language is intended to replicate the computational model that inspires it, and a TM is not only a good candidate to be followed in reason of being the computational model most computing scientists are familiarized with, but also because it is the same computational model traditional computers mimick, thus being easier to implement and often having better performance as a result of a more straightforward implementation.

A *procedural* language is a type of *imperative* language. An *imperative* language imperatively describes the computer *how* to solve a problem. Object-oriented programming is another example of an imperative language, but now we have the idea of objects. On the other hand, FP is a *functional* language which *declares* the program in terms of functions and is one type of a *declarative* language. *Declarative* languages, on the other hand, *declares* what needs to be executed, but does not imperatively describe how [11]. Simiarly to how a procedural language "realizes" a TM, a functional language "realizes" λ-calculus.

The major takeaway in the definition and distinction between these language categories is what kind of syntax you should expect from each. Idiomatic FP should not be describing procedures to solve a problem, but rather a series of function applications that should take on as much a mathematical format as possible. Of course, this is not entirely possible since you might find yourself having to define some primitive language constructs in terms of imperative programming, or having to perform some computations not easily expressed mathematically, but these should be avoided as much as possible, and when cannot be avoided, be clearly compartimentalized to avoid "contaminating" functional code by causing side-effects where not expected, impairing readability or causing other undesirable effects.

III. Functional Programming Concepts

A. Immutability

Immutability is key to functional programming, and although it might be difficult to get around, it provides with several benefits which will later be discussed. Immutability, as the name says, implies in no variables being able to change values once defined.

B. Pure Function

Pure functions are functions that, 1) will produce the same output given the same input, and 2) it has no side-effects. In this way, functions will be effectively deterministic and predictable, always producing the same results for the same inputs [12].

C. Functions as First-Class Citizen

A function is a first-class citizen if it has support for operations usually associated with values, including 1) being passed to functions, 2) being returned by functions, and 3) being assigned to variables [13].

D. Higher-Order Function (HOF)

Higher-order funtions are functions take one or more functions as arguments, or return a function, or both [13].

E. Partial Application

Partial application is a concept of lambda calculus which defines a function as another functions with partially applied arguments. For instance, if we have the function `sum(a,b) := a + b`, we can define `sum2(a) := sum(a, 2)` s.t. `sum2(3) = sum(3, 2) = 3 + 2`.

F. Closure

Closure is a concept already present in procedural languages, but which must be resignified in the context of FP. Closure is the determination of a variable's value by the context of where the functions is defined. For instance, we may have something like `fooProducer(x) := foo() := x`, s.t. `fooProducer` is a one-argument function that produces a zero-argument function that returns the value passed to `fooProducer`. In this case, `foo` is able to determine the value of `x` because `foo` and `x` are both defined in the context of `fooProducer`.

G. Currying

Currying is similar to partial application in that it sets the arguments of a function, but in this case, it does so by setting one argument at a time by returning a one-argument function for each argument until all arguments are set. Here are a few examples for a one, two, and three argument functions respectively:

- `oneArg(x) := x`, then `oneArgCurry(x) := () := oneArg(x)`
- `sum(a, b) := a + b`, then `sumCurry(a) := (b) := () := sum(a, b)`
- `sumAndMult(a, b, c) := (a + b) * c`, then `sumAndMultCurry(a) := (b) := (c) := () := sumAndMult(a, b, c)`

Note that currying is only possible by leveraging closure.

H. Laziness

Laziness is the execution of a computation only when strictly necessary and not at the time of definition. For instance, if you define a variable `A` to be the result of a complex computation, that computation will not be performed until we get to a point in the program where we need to know the value of `A`, such as for printing it or writing it to the disk. On the other hand, if `A` is only used when defining another variable `B`, but for whatever reason, `B` is never needed, then `A` will never get evaluated as well [12].

I. Memoization

Is the caching of a function result for a given input so that it does not need to be recomputed for that same input. The memoized values should always be the correct ones for pure functions.

J. Tail Recursion

Tail recursion is a technique to run a call of a recursive function after returning from the calling function. This is often how imperative-programming loops and iterative functions are represented in idiomatic FP. This feature is, unfortunately, not natively supported by most imperative languages which support FP. Below is one example of a function that computes $\sum_{i=0}^{n} i$:

```
tailSum(i, n, accumulator):
  if i == n:
    return accumulator
  else:
    return tailSum(i+1, n, accumulator+i)

sum(n):
  tailSum(0, n, 0)
```

IV. Remarks on Skipped Concepts

Note that there is more to FP than the concepts here presented, even though the most used FP techniques in OOD (from my experience) have been described above. For a more comprehensive resource on functional patterns, see Greg Baker's notes on FP [14] or refer to a FP resource, such as Haskell's wiki [15].

Note as well that I have skipped discussions on type theory [16], a mathematical theory from which FP borrows many concepts to deal with types in a rigorously defined manner and cope with certain situations not clearly defined by λ-calculus alone. I believe that it would be more benefitial to deepen the discussion on FP in this document rather than introducing the idea of type theory without being able to properly expand on it. Nonetheless, I encourage interested readers to do their own research on the topic as some understanding from the topic can be very benefitial for the FP programmer.

V. Applications of FP in Haiti Hospital Project

Now, I would like to go over some FP applications in a real-world Typescript OOD project, showing how some FP techniques can be applied even in a project that follows a different programming paradigm. For the following illustrations, I will be using code from the Haiti Hospital project [17], commit `c0569944c6820b65a5326811c2d4c4345ef1e63f`. I will try to provide with some snippets of code in this document, but the commit should be referred to for further context.

A. Example 1: Partial Appliction of Functions

The Serializer is an experimental class at the time of this writing which is capable of serializing any object that is decorated with the `@serializable` decorator. A decorator is a function that takes in a constructor and performs some side-effect with it. Decorator functions are run when the class is declared, and thus should run before any usage of that class. The class is located in the `common` folder of the front-end.

The way the Serializer works is by mapping class names to a call to the object constructor, and this mapping is done by the Serializer which is invoked by the decorator. But here's the catch: the constructor the decorator function takes has no information on its own arguments. This is a problem for objects that require arguments to be constructed, and imposing restrictions on the class design is likely to lead to bad class design, and what would go against the purpose of a serializer that was meant to be flexible and generalizable.

The solution to this problem was a multi-argument decorator. The multi-argument decorator takes in a variable number of arguments and returns a function that takes in a constructor and performs a side-effect. Thus, it is a higher-order function that returns a higher-order function. By leveraging closure and partially applying the constructor, I am able to do the task. Below is the code for the decorator:

```
1  export function serializable(...args: any[]) {
2    return (constructor: Function) => {
3      let objectSerializer = ObjectSerializer.getObjectSerializer();
4      let constr = constructor.bind(null, ...args)
5      objectSerializer.registerSerializable(constructor.name, constr);
6    }
7  }
```

Note how in line 2 we begin defining an anonymous function that is going to be returned by the decorator, and in line 4, inside the anonymous function, the method `bind` is called, setting `this` to be `null` inside the constructor (there is no *this* object prior to its construction) and partially applying the multiple arguments to the constructor, which can be passed in the nested function thanks to closure. Through this partial application, a zero-arguments constructor is returned, which can be called at any moment to construct an object with the arguments partially applied. Of course, this design should only work if the arguments passed to the decorator correspond to the appropriate arguments that construct an instance of the object.

This is one example of a feat achieved by FP that probably could not be achieved otherwise. Partial application of functions allows developers to, in a way, bring down the type of arguments to a least common denominator, when one exists. In this case, the least common denominator of arguments is zero (no arguments), which can be achieved by partially applying all arguments, as it is in fact done in the code above.

B. Example 2: Higher Order Functions for Modularity

Another great usage of higher order functions is to enable behaviour modularity. In all kinds of programming paradigms, since the traditional procedural programming paradigm, we are often used to modularizing and structuring data, but when it comes to behaviour, the further the typical developer unaware of FP goes it extracting functions on procedural languages or modularizing methods according to a hierarchical structure as enabled by OOP.

While method overloading and calls to super-class methods are enabled by OOP, being able to only modularize behaviour as far as a hierarchy between types allow is rather limited. One example where this limitation was clear was in the design of what is the current class `Question`, which is found in the `common` directory of the front-end.

The `Question` class is supposed to represent a user-defined question of some type. The answer of that question must be of that type, and each particular question might have further semantic limitations to what is considered a valid answer. For instance, if the question is "what is your age?", then a negative number should not be allowed. The responsibility of making this validation could be left to the developer which would use this library, but since questions in reality do have semantical constraints, the answer validation was deemed as a part of the responsibilities of the question.

One way to solve this problem is by having `Question` be an abstract class and have a method `validate` that must be implemented. In that manner, the developer holds the responsibility to implement the class and define all validation behaviour inside that method. But I believe that this design has some issues: 1) I believe in that manner we would be over-specifying the type of the classes. That is, there would be a class implementation for each particular validation behaviour. 2) It is verbose. 3) It would tightly couple the class to the dependencies used in the validation implementation.

The discussed approach might be the correct approach in OOD, and if all the considerations I deemed as problems might be acceptable under that approach, then I cannot help but to find that such situations are clear shortcomings of OOD. My solution to this problem is to make `Question` accept one or more functions that take in the type of the answer and return a boolean representing the decision on whether the answer is valid or not. In that manner, all of the previous issues are avoided, and the developer only has to worry about defining the validation behaviour for `Question`. Below is the code for `Question`:

```
export abstract class Question<ID, T>
    extends QuestionItem<ID> {
  private readonly prompt: string;
  private answer?: T;

  private readonly validators: Array<(
      answer?: T) => ValidationResult<
      unknown>>;

  // Constructor, getters and setters

  public readonly addValidator = (
      validator: (answer?: T) =>
      ValidationResult<unknown>): void
      => {
    this.validators.push(validator);
  }

  public readonly validate = (): Array
      <ValidationResult<unknown>> => {
    return this.validators
      .map((validator: (answer?: T) =>
            ValidationResult<unknown>)
          => validator(this.answer));
  }
}
```

Although `Question` is abstract because I intend this class to have one implementation per answer and ID type, the validation behaviour is defined dynamically. Thus, if you have, for instance, a class `NumericQuestion extends Question<number, number>`, then the fact that one instance of a `NumericQuestion` determines positive integers as valid answers while the other instance only allows even answers does not imply in them being different class types.

The behaviour modularity provided by higher-order functions and proper usage of functional patterns is, perhaps, the most manifestable advantage of FP in imperative programming and possibly one of the most valuable benefits as it allows code to be abstracted to an extent not previously achievable.

C. Example 3: Declarative Syntax

In the code of the method `searchById(id: ID)` of `QuestionTable`, we have a clear example of how FP syntax should look like in OOD. The method in question searches through a two-dimensional array of `TableCells`, a generic class defined as `class TableCell<ID, T, QuestionType extends Question<ID, T>>`, and attempts to look for a `QuestionType` with the given `id`. Below is the code for the method:

```
public readonly searchById = (id: ID):
    QuestionItem<ID> | undefined => {
  return this.questionTable
    .reduce((questions1, questions2)
        => [...questions1, ...
        questions2])
    .map(questionCell => questionCell.
        getQuestion())
    .filter((questionItem) =>
        questionItem.getId() == id)[0];
}
```

Note in the code above 1) the chaining syntax (that is, a syntax like `object.procedure1(...).procedure2(...)...`) and 2) how a function is anonymously defined as the arguments for each chaining operation even though we could have instead passed pre-defined functions as arguments. The `reduce` operation defines the application of a function that iteratively takes in two elements of a list and returns a single element resulting from their computation until the entire list is *reduced* to a single element. The `map` operations *maps* every element of a list to another element (possibly of another type), and `filter` *filters* out objects that do not satisfy a given predicate (a function that decides between true or false for any given input of the specifying type).

Note how, although we still have to provide *some* description of the operations to be performed, they take a much more declarative, mathematical syntax that resembles a composition of functions rather than a step-by-step description of how to transform the input. This is the expected syntax of FP in OOP, and one of the biggest advantages is the reduced cognitive load as you abstract the computational process of manipulating the data structures and iterating through them and focus only on the behaviour while also dividing the transformation into smaller, compartimentalized steps that are much easier to comprehend than lines of code describing the *how*, but not the *what* of what you are doing.

D. Remarks on Examples

It should be noted that, due current limitations of the Typescript language and the project, other important illustrations of FP concepts cannot be here provided. For instance, Typescript does not provide native support for lazy evaluation, memoization, persistent data structures or tail recursion. Although all these features can be implemented to support FP-like programming, this implies in developers having to maintain these implementations or having to depend on modules that implement them, which are options not as attractive as relying on natively supported operations.

Functional programming also comes along with some type theory techniques not discussed here (functors, monads, etc.) which enforce rigorous definitions and type safety, and although Typescript does a really good job on statically enforcing some of these constraints, it still lacks some features, such as pattern matching or guarded expressions [14].

In light of the lack of support for these operations in imperative languages, FP cannot be thought of a way to achieve a completely different programming paradigm. Rather, it provides with means to achieve higher behaviour modularity and abstraction that can be implemented on top of imperative primitives as it has been illustrated in the examples above.

VI. Final Remarks

In this document, we have discussed the duality between semantics and representation, how the Turing computational model and λ-calculus are both ways of performing computations, and we have briefly compared and contrasted the languages representing these computational models. We went through some terminology and functional patterns and how they can be applied in the context of imperative programming with a real-world example. It has been a long discussion, but a lot of important content could not be covered.

The main goal of this document was to convince the reader that FP, although seemingly restrictive, it does not limit in any way what can be computed, rather presents a different and elegant way to compute problems. The treatment of behaviour as data allows for a level of abstraction to be reached which allows the developer to focus on *what* should be done rather than the *how*, allowing for more rigorous static checking, making the code more expressive and reducing the surface for bugs to occur.

Another important insight that is perhaps more adequate to a discussion of type theory, but is nevertheless relevant in FP and should be appreciated is that a program is safer and more readable when it is more restrictive rather than more permissive. If permissiveness is good, then strict typing is bad, and restricted control flow through `if` , `for` and `while` statements are bad and we are better off with `goto` branching. Strictness enforces clear contexts, readability, exhaustive handling of cases, and valid computational states. Having errors be statically caught is a valuable capability that can avoid hours of infuriating debugging.

If the reader is interested in expanding their knowledge on FP, I invite them to learn a purely functional programming language like Haskell [18]. Although the language is not widely used in industry, programming in Haskell will surely provide solid foundations on functional programming and ultimately challenge the reader to compute through a completely different computational model. Ultimately, even if the reader does not code a purely functional program, the novel ideas presented by this computational model should motivate the reader to think through a fundamentally different perspective from which all kinds of programming paradigms and problem solving scenarios may benefit from.

References

[1] "What's new in jdk 8," https://www.oracle.com/java/technologies/javase/8-whats-new.html.

[2] "262.ecma-international.org/6.0/," https://262.ecma-international.org/6.0.

[3] "Design principles – react," https://reactjs.org/docs/design-principles.html.

[4] "Functional programming — scala 3 — book — scala documentation," https://docs.scala-lang.org/scala3/book/fp-intro.html.

[5] "High-order functions and lambdas — kotlin," https://kotlinlang.org/docs/lambdas.html.

[6] B. J. Copeland, "The Church-Turing Thesis," in *The Stanford Encyclopedia of Philosophy*, Summer 2020 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2020.

[7] L. De Mol, "Turing Machines," in *The Stanford Encyclopedia of Philosophy*, Winter 2021 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2021.
[8] B. J. Copeland, "The Modern History of Computing," in *The Stanford Encyclopedia of Philosophy*, Winter 2020 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2020.
[9] J. Alama and J. Korbmacher, "The Lambda Calculus," in *The Stanford Encyclopedia of Philosophy*, Summer 2021 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2021.
[10] A. M. Turing *et al.*, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.
[11] M. Gabbrielli and S. Martini, *Programming languages: principles and paradigms*. Springer Science & Business Media, 2010.
[12] "Pure functions, laziness, i/o, and monads - school of haskell — school of haskell," https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io.
[13] "Clojure - higher order functions," https://clojure.org/guides/higher_order_functions.
[14] "Cmpt 383 lecture notes," https://ggbaker.ca/prog-langs/.
[15] "Haskellwiki," https://wiki.haskell.org/Haskell.
[16] T. Coquand, "Type Theory," in *The Stanford Encyclopedia of Philosophy*, Fall 2018 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2018.
[17] "drbfraser/hha-haitihospital: System for hospital departments to provide feedback and information to administrators." https://github.com/drbfraser/HHA-HaitiHospital.
[18] "Haskell language," https://www.haskell.org/.