Dynamic Technology Holdings

Study Programme

MEET THE λAMBDAS

A Brief Introduction to Functional Programming

What is This. Can mathematics help software developers write better programs in less time? Yes! So do we all need to be *mathemagicians*? No! This booklet details a study programme that aims to introduce the subject of computing with mathematical functions — functional programming, that is — to "real-world" software developers. So don't let the words "mathematical" and "function" scare you away! In fact, instead of studying an abstract theory of functions, we will favour a completely practical approach to gain an "operational" grasp of the theory. We will learn how to *build* programs using certain mathematical techniques (such as recursion) at our own advantage and moreover those techniques will emerge naturally from the programs we build. The programme will be carried out in a series of workshops and will require some serious homework. Read on for the gory details ...

Andrea Falconi
andrea.falconi@gmail.com

April, 2009

Programme Outline

"Programming is one of the most difficult branches of applied mathematics."

—Edsger W. Dijkstra

Objective. To introduce fundamental concepts of computing with mathematical functions and to show the practical usefulness and benefits offered by mathematical modeling in combination with a modern functional programming language.

Description. The training aims to equip attendants with an "operational" grasp of fundamental concepts of computing with mathematical functions — functional programming, that is. We will dig into types, functions, recursion, and polymorphism. In the process, we will naturally develop an ability to abstract using functions and will concentrate on modeling programs using a functional approach:

- ☐ transform the input in subsequent steps to get the desired output
- ☐ each transformation step is carried out by a function and great emphasis is put on defining the types those functions operate on
- ☐ functions are chained together by means of function composition
- ☐ often the types are recursive and polymorphic and so are the functions defined over them
- ☐ induction becomes central to prove program correctness

List processing is the simplest ground to explore these ideas, so we will start from there and then, when we are comfortable with the concepts, we will extend our reach to arbitrary data types.

Although proof (and program derivation) is a central, distinctive feature of functional programming — and indeed is what justifies the entire enterprise[1] — we will not go beyond mentioning its importance where appropriate. In the same vein, there's a fascinating interplay with Category Theory[2] that we will not be able to even touch on. We feel these topics require a certain maturity of mathematical thought and so are more appropriate for an advanced course.

Concrete coding examples and programs will be used throughout. Most of them have to do with text processing because it is a domain well understood by virtually every developer and so we can venture in non-trivial problems without the risk of obfuscating the concepts being exemplified with the complexity stemming from having to understand an unfamiliar problem domain. However, in the last workshop we will dive into a domain that has received a lot of attention by enterprise application developers, but is not necessarily familiar to many: domain specific languages. We will see how to put to good use what we've learnt to provide a simple, concise, and elegant solution to a problem that would otherwise be quite hard to crack using conventional object-oriented techniques.

[1] It is a celebrated result of 20^{th} century Logic that mathematical proofs are indeed functional programs!

[2] For example, you might have heard of Monads for functional programming.

We will develop in Haskell[5], the world's leading purely functional programming language — the result of decades of cutting-edge scientific research in the field. The choice seems natural for our purposes as the language was also designed with teaching in mind and offers a lean & mean syntax that avoids most of the baroque constructs seen in mainstream programming languages.

Prerequisites. No prior exposure to functional programming is required. However, we do assume you are already proficient in mainstream object-oriented development and are familiar with the tools of the trade — such as compilers, debuggers, revision control systems, etc.

Approach. The idea is to learn powerful mathematical concepts by developing concrete, practical functional programs. Rather than starting from the abstract theory, we will take a completely practical approach in which the abstract concept should emerge naturally after *building* many examples, i.e. with the guidance of the experience. This way, we should be able to appreciate the thought process that led to the concept and improve our creative and abstract thinking skills.

The programme is broken down into study units (detailed in the next pages) of increasing complexity; the topics of each unit are discussed in a workshop and workshops should normally be held weekly. Each unit comes with a detailed study guide you should follow to study the topics we tackled and to practice as much as possible. Also, in between workshops we can have study reviews to work out together exercises that turned out to be particularly stiff or to go over again concepts we feel need more time to be assimilated. Finally, workshop notes, examples, and programs will be posted to our wiki site.

Outcomes. At the end of the training we should be able to:

- ☐ define recursive and polymorphic types and functions
- ☐ use functional abstraction to capture patterns of computation
- ☐ devise functional designs
- ☐ write practical Haskell programs using all of the above

Literature. Our primary textbook will be "Haskell: The Craft of Functional Programming"[6]. Additionally, to cross-check our understanding or integrate additional material, we may read selected chapters from the "The Haskell School of Expression: Learning Functional Programming through Multimedia"[1] and "Real World Haskell"[4]. In the following, we will refer to these texts as, respectively, the *Craft*, the *School*, and *RwH*. Reference to further material is explicitly given in the study guides. Note that the only essential readings are the chapters from the *Craft*; all the rest is optional — peruse if you wish, but in principle sticking to the *Craft* should be enough.

Workshop I

Learning how to Compute . . . without a Computer!

"A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine."

—Alan Turing

Goals.

- ☐ To introduce informally, through concrete examples fundamental concepts of "computing with functions".
- ☐ To start developing a taste for functional abstraction.

Topics. Functions and types. Function composition. Induction and recursion. Recursive definitions of functions. Functional evaluation model. Lazy evaluation. Importance of proof.

Description. Don't let the arcane terms above scare you away! Instead of seeking precise definitions, we will examine several practical examples through which we will get an operational grasp of those concepts. We will "visualize" functions and types, come up with an easy recipe to define things in "terms of themselves", and see how this leads to an extremely simple and clean program evaluation model — so simple that in fact all we need to work out a program's output is pencil and paper! In the process, we will note that the output of a function can become the input of another and so we have a way to compose small computations into bigger ones as if by "piping"; moreover the inputs and outputs of this pipeline can be produced lazily (on-demand), which leads to an efficient and powerful modularization mechanism. Finally, we will mention the important role that proof can (actually should!) play in programming.

Study Guide.

(1) After the workshop we will post the session notes and program examples on the wiki. Review them while the concepts are still fresh in your mind. At the same time, you should read §1 from the *Craft* which covers the topics discussed in the session.
(2) Start familiarizing yourself with Haskell. Read §1 from *RwH* and use `ghci` to run the examples yourself.
(3) Read §3 from the *Craft*. It overlaps with §1 from *RwH* and should be useful to cross-check your understanding of the concepts.
(4) Do all the exercises in §3 from the *Craft*. Write down your function definitions and use pencil & paper to compute them on some test inputs. Only when you're confident a definition works, you should run your tests in `ghci`. Are you getting the same results? If not, try to understand whether your definition is incorrect or you got wrong some of the evaluation steps you simulated on paper. How could you ensure the absolute *absence* of errors in your definitions? Is testing enough?

Workshop II

Coming to Love Lists

"The conception of list processing as an abstraction created a new world in which designation and dynamic symbolic structure were the defining characteristics. The embedding of the early list processing systems in languages (the IPLs, LISP) is often decried as having been a barrier to the diffusion of list processing techniques throughout programming practice; but it was the vehicle that held the abstraction together."

—Allen Newell and Herbert Simon

Goals.

- ☐ To understand the recursive and polymorphic nature of the list type.
- ☐ To learn how to define functions over lists.

Topics. Polymorphic functions. Cons function and list type. List comprehensions. Pattern matching. Primitive and general recursion over lists. Streams.

Description. Lists are an essential weapon in the programmer's arsenal and functional programming excels at list processing by providing elegant, concise, yet extremely effective means to compute with lists. The list type is defined in terms of itself (i.e. recursively) and the definition is generic in the sense that it doesn't depend on the type of the contained elements — i.e. it applies as well to integers as to bananas. (Informally, this is what you call polymorphism.) We will construct several lists and let the recursive and polymorphic nature of the list type emerge naturally from the concrete examples. Armed with this insight, we will see how easy it becomes to define functions recursively over lists and, at the same time, keep the definition generic. These functions we will work on are going to be part of a small program that we should be able to run by the end of the session. We will also leave a few minutes to have a further glimpse into the awesome power of lists by hinting at their use for computing with infinite data sequences (aka "streams").

Study Guide.

(1) As usual we will post notes and code on the wiki. Review the material as soon as you can.
(2) Start drilling into list processing by reading §5, §6, and §7 from the *Craft*. It is also a good idea to cross-check your understanding by reading §2 from *RwH*, which covers more or less the same ground.
(3) Practice, practice, practice! Try and do as many exercises as you can from §5, §6, and §7 in the *Craft*. Again, use pencil & paper first and then verify in `ghci`. It is highly recommended that you do the exercises as you read along so that you have a chance to crystallize each concept in your mind before moving on to the next one.
(4) After completing all the reading above, you could optionally read §14 from the *School* to find out more about streams.

Workshop III

The Joys of Abstraction

"The art of doing mathematics consists in finding that special case which contains all the germs of generality."

—David Hilbert

Goals.

- ☐ To realize that functions are themselves values and so they can be arguments to other functions or returned by them.
- ☐ To be able to spot repeating patterns of computation and abstract them away using higher-order functions.

Topics. Higher-order functions. Abstracting over list computations: mapping, filtering, folding, and scanning. Lambdas. Currying and partial application. Point-free definitions.

Description. In this session we will look back at the functions we defined in the past workshops and see that many of them have "suspiciously" similar definitions. It won't be long before we realize that conceptually they do the same thing; perhaps we could classify them as those that perform the same computation on each element of a list (mapping), those that remove unwanted elements from a list (filtering), or those that reduce all elements to a single value (folding). And indeed we could be quite content with abstracting away those concepts that can be reused to design solutions to similar problems. (Yes, this is what you would call a design pattern in object-oriented land.) But how far down the rabbit hole are we prepared to go? What if we could turn those concepts into actual code? (Wow! Executable design patterns, you may wonder.) We will see that all this becomes easy if we're willing to climb the abstraction ladder further and treat functions as values. Now, if a function is a rule that turns a value into another value and functions can be values too, it stands to reason that we could define a function whose input and/or output values are themselves functions...

Study Guide.

(1) Review notes and code soon after the session.
(2) Read §9 and §10 (excluding §10.8 and §10.9) from the *Craft*. Cross-check your understanding by reading §5.1, §5.2, §5.4 and §9 from the *School*.
(3) The material may appear quite abstract on first reading, so it is *essential* that you do as many exercises as you can. Again, don't wait to finish reading a chapter, do the exercises as you go. It is recommended that you attempt at least 80% of the exercises in §9 and §10 from the *Craft* (excluding exercises following §10.8 and §10.9).

(4) §9.3 in the *Craft* hints at the fact that indeed we could capture in code the pattern of primitive recursion itself — viz the `foldr` function. This is an even more abstract standpoint, yet a "unifying" notion: it is indeed the pattern underlying all the computation patterns discussed in the session and, in fact, functions like `map` or `filter` can be readily expressed in terms of `foldr`. Those who wish to stretch their knowledge further in that direction may read: *"A tutorial on the universality and expressiveness of fold"*.[2]

Workshop IV

Face 2 Face

"FORTRAN's tragic fate has been its wide acceptance, mentally chaining thousands and thousands of programmers to our past mistakes. [. . . LISP] has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts."
—Edsger W. Dijkstra

Goals.

- ☐ To use the skills acquired so far to develop a "real-world" program.
- ☐ To compare development through formal methods to the traditional object-oriented approach.

Topics. No new topics will be introduced in this workshop.

Description. Time to roll up our sleeves and do some "real-world" development! We will put what we've learnt so far to good use to model and implement a sizable, non-trivial program in Haskell. (In the process we will also touch on how we could *prove* it correct.) Then, we will have a brief look at an object-oriented program in Java that implements *exactly* the same requirements and draw a comparison between development through formal methods and the more traditional object-oriented approach. To wet your appetite, here's a comparison chart of the two programs:

	OO + Java	**Maths + Haskell**
Lines of code	1510	48
Development effort (hrs)	42	5
Bugs	15	0
Performance (ms)	893	282

And of course, the Java program was engineered to today's highest industry standards. . .

Study Guide.

(1) The Haskell program draws heavily on the concepts tackled during the last workshop, so it's a good time to review those chapters from the *Craft* and the *School* detailed in point 2 of the last workshop's study guide.
(2) Implement all the functions whose definitions were left out during the session.
(3) Have a look at the Software Architecture Document of the Java program and then look at the code. How does the object-oriented approach to design differ from the route we took to model the Haskell program? What can you say about program verification?

Workshop V

More Fun with Types

"I still believe in abstraction, but now I know that one ends with abstraction, not starts with it. I learned that one has to adapt abstractions to reality and not the other way around."

—Alexander Stepanov

Goals.

- ☐ To be able to program with families of types and overloading.
- ☐ To extend list-processing concepts to arbitrary data types.

Topics. Overloaded functions and type classes. Type class signature and instances. Algebraic types: tupling and disjoint union. Constructors and pattern matching. Recursive and polymorphic algebraic types. Defining functions over algebraic types.

Description. So you are a caffeine addict and always sit nearby an espresso machine. However, something deeply annoys you: to deliver your espresso, all machines from CoffeeHacks require you to first press the blue button and then the red button, whereas all those from EspressoGeek work by pressing the red button first and then the green button. Although all that matters to you in the end is your cup of joy, you have to use a different procedure for different types. So you wish all espresso machines just had a plain, simple "make coffee" button. Although you have now come up with a useful abstraction, you won't have an easy time convincing CoffeeHacks and EspressoGeek — they are fierce competitors. But you may find some comfort in knowing that you could use type classes and overloading in your program to model this situation and craft the "make coffee" button yourself. This begs the question of how to represent espresso machines in our functional world. We will see that we can easily construct custom types by generalizing the mechanism used to build lists and then leverage what we've learnt about list processing to program in this more general setting.

Study Guide.

(1) Review notes and code soon after the session.
(2) Read §12.1, §12.2, §12.3, and §14, (excluding §14.6 and §14.7) from the *Craft*. You may want to integrate the material with §3 and §4 from *RwH*.
(3) Read the article on Wikipedia about Huffman coding; §15 in the *Craft* details an implementation of Huffman coding in Haskell. Read up to §15.3 and then try to implement the program yourself without reading any further.

Workshop VI
The Language Factory

"The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language."

—Donald E. Knuth

Goals.

- ☐ To introduce the concept of domain specific language (DSL).
- ☐ To be able to embed simple DSL's into a hosting functional language.

Topics. Domain specific languages. Language embedding. Combinators.

Description. Some business folks seem to have a predilection for changing the rules of the game fairly often — they call it "innovation". However fun that may be, the poor programmer tasked to encode those rules usually has a different take on this attitude — some call it business "illogic". In some cases, keeping up with the required pace of change could be a programming nightmare. Ah, if only the system could "understand" the specific language of the business domain at hand, then the business heads could "talk" directly to the system to specify their business rules. For example, we could devise a language akin to spoken *financialese* to let a financial advisor craft an investment package. However, developing a computer language from scratch is a serious undertaking and the cost may not be worth the return in flexibility. We will explore an alternative approach that will allow us to leverage what we've learnt in the past workshops to easily create a new language by embedding it into an existing one.

Study Guide.

(1) There are no new topics to study for this session — all the constructions we saw are merely using what we've learnt in the past workshops. However, as we've reached the end of the study programme, it may be a good idea to go over again all the material we've studied from start to end so that you can consolidate your knowledge. As usual, top tip: practice as much as you can!
(2) If you're interested, "Composing contracts: An adventure in financial engineering" [3] illustrates a DSL for financial contracts. (You can also download the recording of a recent talk at Ericsson given by Simon Peyton Jones on the subject — see http://ulf.wiger.net/weblog/2008/02/29/simon-peyton-jones-composing-contracts-an-adventure-in-financial-engineering/.)

(3) Also, §15 in the *School* describes a sophisticated DSL for graphical animations, whereas §19 describes one for controlling robots. The material we've learnt so far should be enough for you to venture in §15 (although it's probably a good idea to read at least §13.4 first). On the other hand, §19 requires knowledge of several topics that we haven't covered at all and so you should read §16, §17, and §18 beforehand.

References

[1] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, February 2000.

[2] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(04):355–372, 1999.

[3] S.P. Jones, J.M. Eber, and J. Seward. Composing contracts: An adventure in financial engineering functional pearl. In *5 th ACM SIGPLAN International Conference on Functional Programming(ICFP'00)*, pages 280–292, 2000.

[4] B. O'Sullivan, D.B. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, Inc., 2008.

[5] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[6] Simon Thompson. *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison Wesley, March 1999.