Pro Scala: Monadic Design Patterns for the Web

L.G. Meredith

© *Draft date February 18, 2010*

Contents

List of Figures

List of Tables

Preface

The book you hold in your hands, Dear Reader, is not at all what you expected...

Chapter 1

Motivation and Background

Where are we; how did we get here; and where are we going?

TBD

1.1 Where are we

1.1.1 The concurrency squeeze: from the hardware up, from the web down

It used to be fashionable in academic papers or think tank reports to predict and then bemoan the imminent demise of Moore's law, to wax on about the need to "go sideways" in hardware design from the number of cores per die to the number of processors per box. Those days of polite conversation about the on-coming storm are definitely in our rear view mirror. Today's developer knows that if her program is commercially interesting at all then it needs to be web-accessible on a 24x7 basis; and if it's going to be commercially significant it will need to support at least 100's if not thousands of concurrent accesses to its features and functions. Her application is most likely hosted by some commercial outfit, a Joyent or an EngineYard or an Amazon EC3 or ... who are deploying her code over multiple servers each of which is in turn multi-processor with multiple cores. This means that from the hardware up and from the web down today's intrepid developer is dealing with parallelism, concurrency and distribution.

Unfortunately, the methods available in in mainstream programming languages of dealing with these different aspects of simultaneous execution are not up to the task of supporting development at this scale. The core issue is complexity. The

modern application developer is faced with a huge range of concurrency and concurrency control models, from transactions in the database to message-passing between server components. Whether to partition her data is no longer an option, she's thinking hard about *how* to partition her data and whether or not this "eventual consistency" thing is going to liberate her or bring on a new host of programming nightmares. By comparison threads packages seem like quaint relics from a time when concurrent programming was a little hobby project she did after hours. The modern programmer needs to simplify her life in order to maintain a competitive level of productivity.

Functional programming provides a sort of transition technology. On the one hand, it's not that much of a radical departure from mainstream programming like Java. On the other it offers simple, uniform model that introduces a number of key features that considerably improve productivity and maintainability. Java brought the C/C++ programmer several steps closer to a functional paradigm, introducing garbage collection, type abstractions such as generics and other niceties. Languages like `OCaml`, `F#` and `Scala` go a step further, bringing the modern developer into contact with higher order functions, the relationship between types and pattern matching and powerful abstractions like monads. Yet, functional programming does not embrace concurrency and distribution in its foundations. It is not based on a model of computation, like the actor model or the process calculi, in which the notion of execution that is fundamentally concurrent. That said, it meshes nicely with a variety of concurrency programming models. In particular, the combination of higher order functions (with the ability to pass functions as arguments and return functions as values) together with the structuring techniques of monads make models such as software transactional memory or data flow parallelism quite easy to integrate, while pattern-matching additionally makes message-passing style easier to incorporate.

1.1.2 Ubiquity of robust, high-performance virtual machines

Another reality of the modern programmer's life is the ubiquity of robust, high-performance virtual machines. Both the `Java` Virtual Machine (`JVM`) and the Common Language Runtime (`CLR`) provide manage code execution environments that are not just competitive with their unmanaged counterparts (such as `C` and `C++`), but actually the dominant choice for many applications. This has two effects that are playing themselves out in terms of industry trends. Firstly, it provides some level of insulation between changes in hardware design (from single core per die to multi-core, for example) that impacts execution model and language level interface. To illustrate the point note that these changes in hardware have impacted memory models. This has a much greater impact on the `C/C++` family of languages than on

Java because the latter is built on an abstract machine that hides the underlying memory model. One may, in fact, contemplate an ironic future in which this abstraction alone causes managed code to outperform C/C++ code because of C/C++'s faulty assumptions about best use of memory. Secondly, it completely changes the landscape for language development. By providing a much higher level and more uniform target for language execution semantics it lowers the barrier to entry for contending language designs. It is not surprising, therefore, that we have seen an explosion in language proposals in the last several years, including Clojure, Fortress, Scala, F# and many others. It should not escape notice that all of the languages in that list and the majority of the proposals coming out are either functional languages, object-functional languages, or heavily influenced by functional language design concepts.

1.1.3 Advances in functional programming, monads and the awkward squad

One of the reasons that language design proposals have been so heavily influenced by functional language design principles is that functional language design has made demonstrable progress. Since the '80's when Lisp and it's progeny were thrown out of the industry for performance failures a lot of excellent work has gone on that has rectified many of the problems those languages faced. In particular, while Lisp implementations tried to take a practical approach to certain aspects of computation, chiefly having to do with side-effecting operations and i/o, the underlying semantic model did not seem well-suited to address those kinds of computations. And yet, not only are side-effecting computations and especially i/o ubiquitous, using them led (at least initially) to considerably better performance. Avoiding those operations (sometimes called functional purity) seemed to be an academic exercise not well suited to writing "real world" applications.

However, while many industry shops were throwing out functional languages, except for niche applications, work was going on that would reverse this trend. One of the key developments in this was an early bifurcation of functional language designs at a fairly fundamental level. The Lisp family of languages are untyped and dynamic. In the modern world the lack of typing might seem egregiously unmaintainable, but by comparison to C it was more than made up for by the kind of dynamic meta-programming that these languages made possible. Programmers enjoyed a certain kind of productivity because they could "go meta" – writing programs to write programs (even dynamically modify them on the fly) – in a uniform manner. This sort of feature has become mainstream, as found in Ruby or even Java's reflection API, precisely because it is so extremely useful. Unfortunately, the productivity gains of meta-programming were not enough to offset the performance

shortfalls at the time.

There was, however, a statically typed branch of functional programming that began to have traction in certain academic circles with the development of the `ML` family of languages – which today includes `OCaml`, which can be considered the direct ancestor of `Scala`. One of the very first developments in that line of investigation was the recognition that data description came in not just one but *two* flavors: types and *patterns*. The two flavors, it was recognized, are dual. Types tell the program how data was built up from its components while patterns tell a program how to take data apart in terms of its components. These are the origins of elements in `Scala`'s design like **case class**es and the **match** construct. The `ML` family of languages also gave us the first workable parametric polymorphism (that's generic types to you).

Still these languages suffered when it came to a compelling and uniform treatment of side-effecting computations. That all changed with Haskell. In the mid-80's a young researcher by the name of Eugenio Moggi observed that an idea previously discovered in a then obscure branch of mathematics (called category theory) offered a way to *structure* functional programs to allow them to deal with side-effecting computations in uniform and compelling manner. Essentially, the notion of a monad (as it was called in the category theory literature) provided a language level abstraction for structuring side-effecting computations in a functional setting. In today's parlance, he found a domain specific language, a DSL, for organizing side-effecting computations in an ambient (or hosting) functional language. Once Moggi made this discovery another researcher, Phil Wadler, realized that this DSL had a couple of different "presentations" that were almost immediately understandable by the average programmer. One presentation, called comprehensions (after it's counter part in set theory), could be understood directly in terms of a very familiar construct **SELECT** ... **FROM** ... **WHERE** ...; while the other, dubbed **do**-notation by the `Haskell` community, provided operations that behaved remarkably like sequencing and assignment. `Haskell` offers syntactic sugar to support the latter while the former has been adopted in both `XQuery`'s FLWOR-expressions and Microsoft's `LINQ`

1.2 Where are we going

1.2.1 A functional web

1.2.2 DSL-based design

1.3 How are we going to get there

1.3.1 Leading by example

The principal technique throughout this book is leading by example. What this means in this case is that the ideas are presented primarily in terms of a coherent collection of examples that work together to do something. Namely, these examples function together to provide a prototypical web-based application with a feature set that resonates with what application developers are building today and contemplating building tomorrow.

Let's illustrate this in more detail by telling a story. We imagine a cloud-based editor for a simple programming language, not unlike `Mozilla's bespin` . A user can register with the service and then create an application project which allows them

- to write code in a structured editor that understands the language;
- manage files in the application project;
- compile the application;
- run the application

Our toy language

For our example we'll need a toy language. Fittingly for a book about `Scala` we'll use the λ-calculus as our toy language. The core *abstract* syntax of the lambda calculus is given by the following *EBNF* grammar.

MENTION	ABSTRACTION
$M, N ::= x$	$\mid \lambda x.M$

APPLICATION

$\mid MN$

To this core let us add some syntactic sugar.

PREVIOUS
$M, N ::= ...$

LET
$\mid letx = M$

SEQ
$\mid M; N$

Now let's wrap this up in concrete syntax.

MENTION
$M, N ::= x$

ABSTRACTION
$\mid$ `(`x_1`,...,`x_k`)=>`M

APPLICATION
$\mid M$`(`N_1`,...,`N_k`)`

LET
$\mid$ `val`x`=`M

SEQ
$\mid M$`;`N

GROUP
$\mid$ `M`

Chapter 2

Toolbox

Notation and terminology

TBD

2.1 Introduction to notation and terminology

While we defer to the rich and growing body of literature on Scala to provide a more complete set of references for basic Scala notation, to be somewhat self-contained in this section we review the notation and terminology we will need for this book.

2.2 Introduction to core design patterns

2.3 Variations in presentation

2.3.1 A little history

Haskell was the first programming language to popularize the notion of monad as a structuring technique for functional programming. There were several key ideas that went into the `Haskell` packaging of the idea. One was to treat the core elements that make up a monad more or less *directly* without appeal to category theory – which was where the notion originated. This is considerably easier to do in a functional programming language because the ambient language can be thought of as a category; thus, for the average programmer there is no need to refer to

categories, in general, but only to the "universe" of programs that can be written in the language at hand. Then, because Haskell already has a notion of parametric polymorphism (which we can think of as as generics in the context of this discussion), a monad's most central piece of data is a parametric type constructor, say T.

Given such a type constructor, you only need a pair of maps (one of which is higher order). Thus, in `Haskell` a monad is presented in terms of the following data

- a parametric type constructor, T a
- a **return** map enjoying the signature **return** :: a −> T a
- a bind map enjoying the signature bind : T a −> (a −> T b) −> T b

In `Haskell` these elements can be collected inside a typeclass. Resulting in a declaration of the form

```
typeclass Monad T a where
 return :: a -> T a
 bind :: T a -> (a -> T b ) -> T b
```

Listing 2.1: monad typeclass

Now, it's not enough to simply have this collection of pieces. The pieces have to fit together in a certain way; that is, they are subject to the following laws:

- **return** (bind a f) ≡ f a
- bind m **return** ≡ m
- bind (bind m f) g ≡ bind m (\ x −> bind (f x) g)

Unlike `Haskell`, `Scala` did not reify the notion of monad under a **trait**, the language's equivalent of `Haskell`'s typeclass. Instead the systematic means of desugaring **for**-notation and polymorphic interpretations of flatMap, etc are the effective definitions of the notion in `Scala`.

2.3.2 A little more history

If one were to reify the notion in `Scala` there are several design choices – all of which endure some desiderata. Following the original presentation developed in category theory, however, has some crucial advantages:

- intuition
- correspondence to previously existing structures
- decomposition of the requirements

which we explore in some detail here.

Intuition: Monad as container

As we will see the notion of monad maps nicely onto an appropriately parametric notion of container. From this point of view we can imagine a container "API" that has three basic operations.

Shape of the container The first of these is a *parametric* specification of the *shape* of the container. Examples of container shapes include: List[A], Set[A], Tree[A], etc. At the outset we remain uncommitted to the particular shape. The API just demands that there is some shape, say S[A].

Putting things into the container The next operation is very basic, it says how to put things into the container. To align with a very long history, we will refer to this operation by the name unit. Since the operation is supposed to allow us to put elements of type A into containers of shape S[A], we expect the signature of this operation to be unit : A =>S[A].

Flattening nested containers Finally, we want a generic way to flatten nested containers. Just like there's something fundamentally the same about the obvious way to flatten nested lists and nested sets, we ask that the container API provide a canonical way to flatten nested containers. If you think about it for a moment, if a container is of shape, S[A], then a nested container will be of shape, S[S[A]]. If history demands that we call our flattening operation mult, then our generic flatten operation will have signature, mult : S[S[A]] =>S[A].

Preserving connection to existing structure: Monad as generalization of monoid

Programmers are very aware of data structures that support a kind of concatenation operation. The data type of String is a perfect example. Every programmer expects that the concatenation of a given String, say s, with the empty String, "" will

return a result string equal to the original. In code, s.equals(s + "") ==**true** . Likewise, string concatenation is insensitive to the order of operation. Again, in code, ((s + t) + u).equals(s + (t + u)) ==**true** .

Most programmers have noticed that these very same laws survive polymorphic interpretations of +, equals and the "empty" element. For example, if we substituted the data type Integer as the base type and used integer addition, integer equality, and 0 as the empty element, these same code snippets (amounting assertions) would still work.

Many programmers are aware that there is a very generic underlying data type, historically referred to as a *monoid* defined by these operations and laws. In code, we can imagine defining a **trait** in `Scala` something like

Decomposition of monad requirements

The constraints on any given monad candidate are well factored into three different kinds of requirements – operating at different levels of the "API": functoriality, naturality and coherence. Often these can be mechanically verified, and when they can't there are natural ways to generate spot-checks that fit well with tools such as `ScalaCheck`.

A categorical way to look at monads

In category theory the monad is presented in terms of the following data

- a "unit" map enjoying the signature $unit : A \to T[A]$
- a "mult" map enjoying the signature $mult : T[T[A]] \to T[A]$

subject to the following laws:

- $T(id_A) = id_{T[A]}$
- – $unit(f) \circ unit_A = unit_B \circ unit(B)$
  – $mult(f) \circ mult_{T[A]} = mult_{T[B]} \circ mult(B)$

Coherence
- – $mult \circ Tmult = mult \circ multT$
  – $mult \circ Tunit = multunitT$

These two definitions are interchangeable. That's what makes them "presentations" of the same underlying idea.

One of the reasons for the difference in presentation is that `Haskell` doesn't treat the Monad type class as a Functor. The refactoring of the *mult* map into the *bind* map is that it builds functoriality into definition. The other reason is the do notation.

`Scala` follows `Haskell`, though the same notation is called for-notation. The basic construct looks like

and the de-sugaring looks like

This means, therefore, that we have the following correspondence

Chapter 3

An IO-monad for http streams

Code first; questions later

TBD

3.1 Code first, questions later

TBD

3.1.1 An HTTP-request processor

3.1.2 What we did

3.2 Synchrony, asynchrony and buffering

TBD

3.3 State, statelessness and continuations

TBD

Chapter 4

Parsing requests, monadically

How to get from the obligatory to the well formed

TBD

4.1 Obligatory parsing monad

TDB

4.2 Your parser combinators are showing

TBD

4.3 EBNF and why higher levels of abstraction are better

TBD

Chapter 5

The domain model as abstract syntax

In which Pooh and Piglet understand the value of pipelines

TBD

5.1 Our abstract syntax

TBD

5.2 Our application domain model

TBD

5.3 A transform pipeline

TBD

Chapter 6

Zippers and contexts and URI's, oh my!

Zippers are not just for Bruno anymore

TBD

6.1 Zippers are not just for Bruno anymore

TBD

6.2 Constructing contexts and zippers from data types

TBD

6.3 Mapping URIs to zipper-based paths and back

TBD

Chapter 7

A review of collections as monads

Where are we; how did we get here; and where are we going?

TBD

7.1 Monad as container

TBD

7.2 Monads and take-out

TBD

Chapter 8

Domain model, storage and state

Mapping to the backend

TBD

8.1 Mapping our domain model to storage

TBD

8.2 Storage and language-integrated query

TBD

8.3 Continuations revisited

TBD

Chapter 9

Putting it all together

The application as a whole

TBD

9.1 Our web application end-to-end

TBD

9.2 Deploying our application

9.2.1 Why we are not deploying on GAE

9.3 From one web application to web framework

TBD

Chapter 10

The semantic web

Where are we; how did we get here; and where are we going?

TBD

10.1 How our web framework enables different kinds of application queries

TBD

10.2 Searching for programs

TBD