Pro Scala: Monadic Design Patterns for the Web

L.G. Meredith

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 or 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 a model of computation, like the actor model or the process calculi that are based on a 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

1.1.3 Advances in functional programming, monads and the awkward squad

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

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.

$$\begin{array}{ll} \text{MENTION} & \text{ABSTRACTION} \\ M, N ::= x & \mid \lambda x.M \end{array}$$

$$\begin{array}{l} \text{APPLICATION} \\ \mid MN \end{array}$$

To this core let us add some syntactic sugar.

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

LET
$\mid let x = M$

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

Given that we have a type constructor, T

In Haskell a monad is presented in terms of the following data

subject to the following laws:

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

subject to the following laws:

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

Unlike Haskell, Scala did not reify the notion of monad under the language's equivalent of the Haskell type class. Instead the systematic means of de-sugaring for-notation and polymorphic interpretations of flatMap, etc are the effective definitions of monads in Scala.

If one were to reify the notion in Scala there are several design choices – all of which endure some desiderata. Following category theory, however, has some crucial advantages: namely, the constraints on any given monad candidate are well factored into 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.

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