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?

If on a winter's night a programmer (with apologies to Italo Calvino) You've just picked up the new book by Greg Meredith, Pro Scala. Perhaps you've heard about it on one of the mailing lists or seen it advertised on the `Scala` site or at Amazon. You're wondering if it's for you. Maybe you've been programming in functional languages or even `Scala` for as long as you can remember. Or maybe you've been a professional programmer for quite some time. Or maybe you're a manager of programmers, now and you're trying to stay abreast of the latest technology. Or, maybe you're a futurologist who looks at technology trends to get a sense of where things are heading. Whoever you are, if you're like most people, this book is going to make a lot more sense to you if you've already got about five to ten thousand hours of either `Scala` or some other functional language programming under your belt [1]. There may be nuggets in here that provide some useful insights for people with a different kind of experience; and, of course, there are those who just take to the ideas without needing to put in the same number of hours; but, for most, that's probably the simplest gauge of whether this book is going to make sense to you at first reading.

On the other hand, just because you've got that sort of experience under your belt still doesn't mean this book is for you. Maybe you're just looking for a few tips and tricks to make `Scala` do what you want for the program you're writing right

[1] Now, i've been told that this is too much to expect of a would-be reader; but, when i whip out my calculator, i see that (5000 hrs / 25 hrs/wk) / 52 wks/yr) = 3.84615384615 years. That means that if you've put in under four years at a hobbyist level, you've met this requirement. Alternatively, if you've put in less than two years as a professional working solely in functional languages, you've met the requirement. Honestly, we don't have to give in to inflationary trends in the meanings of terms. If we say something is aimed at a pro, we could mean what we say.

now. Or maybe you've got a nasty perf issue you want to address and are looking here for a resolution. If that's the case, then maybe this book isn't for you because this book is really about a point of view, a way of looking at programming and computation. In some sense this book is all about programming and complexity management because that's really the issue that the professional programmer is up against, today. On average the modern programmer building an Internet-based application is dealing with no less than a dozen technologies. They are attempting to build applications with nearly continuous operation, 24x7 availability servicing 100's to 1000's of concurrent requests. They are overwhelmed by complexity. What the professional programmer really needs are tools for complexity management. The principle aim of this book is to serve that need in that community.

The design patterns expressed in this book have been developed for nearly *fifty* years to address exactly those concerns. Since `Scala` isn't nearly fifty years old you can guess that they have origins in older technologies, but `Scala`, it turns out, is an ideal framework in which both to realize them and to talk about their ins and outs and pros and cons. However, since they don't originate in `Scala`, you can also guess that they have some significant applicability to the other eleven technologies the modern professional programmer is juggling.

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 managed 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 hardware 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 not only hides the underlying hardware memory model, but more importantly can hide changes to the 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 that percolate all through application code. 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 are either functional languages, object-functional languages, and the majority of the proposals coming out are either functional, object-functional or heavily influenced by functional language design concepts.

1.1.3 Advances in functional programming, monads and the awkward squad

Perhaps chief among the reasons for the popularity of developing a language design based on functional concepts is that the core of the functional model is inherently simple. The rules governing the execution of functional programs (the basis of an abstract evaluator) can be stated in half a page. In some sense functional language design is a "path of least resistance" approach. A deeper reason for adoption of functional language design is that the core model is *compositional*. Enrichment of the execution semantics amounts to enrichment of the components of the semantics. Much more can be said about this, but needs to be deferred to a point where more context has been developed. Deep simplicity and compositionality are properties and principles that take quite some time to appreciate while some of the practical reasons that recent language design proposals have been so heavily influenced by functional language design principles is easily understood by even the most impatient of pragmatic of programmers: functional language design has made significant and demonstrable progress addressing performance issues that plagued it at the beginning. Moreover, these developments have significant applicability to the situation related to concurrent execution that the modern programmer finds herself now.

Since the mid '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 available in `Lisp` and it's derivatives 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`, the language that can be considered the direct ancestor of both `Scala` and `F#`. 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 is built up from its components while patterns tell a program how to take data apart in terms of its components. The crucial point is that these two notions are just two sides of the same coin and can be made to work together and support each other in the structuring and execution of programs. In this sense the development – while an enrichment of the language features – is a reduction in the complexity of concepts. Both language designer and programmer think in terms of one thing, description of data, while recognizing that such descriptions have uses for structuring and de-structuring data. 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 robust instantiations of parametric polymorphism. The widespread adoption of generics in `C/C++`, `Java` and `C#` say much more about the importance of this feature than any impoverished account the author can conjure here. Again, though, the moral of the story is that this represents a significant reduction in complexity. Common container patterns, for example, can be separated from the types they contain, allowing for programming

that is considerably DRYer. [2]

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" (different concrete syntaxes for the same underlying abstract syntax) 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`.

Of course, to say that `Haskell` offers syntactic sugar hides the true nature of how monads are supported in the language. There are actually three elements that come together to make this work. First, expressing the pattern at all requires support for parametric polymorphism, generics-style type abstraction. Second, another mechanism, `Haskell`'s typeclass mechanism (the `Haskell` equivalent to `Scala`'s **trait**) is required to make the pattern itself polymorphic. Then there is the **do**-notation itself and the syntax-driven translation from that to `Haskell`'s core syntax. Taken together, these features allow the compiler to work out which interpretations of sequencing, assignment and return are in play – without type annotations. The simplicity of the design sometimes makes it difficult to appreciate the subtlety, or the impact it has had on modern language design, but this was the blueprint for the way `Scala`'s **for**-comprehensions work.

With this structuring technique (and others like it) in hand it becomes a lot easier to spot (often by type analysis alone) situations where programs can be rewritten to equivalent programs that execute much better on existing hardware. This is one of the central benefits of the monad abstraction, and these sorts of powerful abstractions are among the primary reasons why functional programming has made

[2] DRY is the pop culture term for the 'Do not Repeat Yourself'. Don't make me say it again.

such progress in the area of performance. As an example, not only can `LINQ`-based expressions be retargeted to different storage models (from relational database to `XML` database) they can be rewritten to execute in a data parallel fashion. Results of this type suggest that we are really just at the beginning of understanding the kinds of performance optimizations available through the use of monadic programming structuring techniques.

It turns out that side-effecting computations are right at the nub of strategies for using concurrency as a means to scale up performance and availability. In some sense a side-effect really represents an interaction between two systems (one of which is viewed as "on the side" of the other, i.e. at the boundary of some central locus of computation). Such an interaction, say between a program in memory and the I/O subsystem, entails some sort of synchronization. Synchronization constraints are the central concerns in using concurrency to scale up both performance and availability. Analogies to traffic illustrate the point. It's easy to see the difference in traffic flow if two major thoroughfares can run side-by-side versus when they intersect and have to use some synchronization mechanism like a traffic light or a stop sign. So, in a concurrent world, functional purity – which insists on no side-effects, i.e. no synchronization – is no longer an academic exercise with unrealistic performance characteristics. Instead computation which can proceed without synchronization, including side-effect-free code, becomes the gold standard. Of course, it is not realistic to expect computation never to synchronize, but now this is seen in a different light, and is perhaps the most stark way to illustrate the promise of monadic structuring techniques in the concurrent world programmers find themselves. They allow us to write in a language that is at least notionally familiar to most programmers and yet analyze what's written and retarget it for the concurrent setting.

In summary, functional language design improved in terms of

- extending the underlying mechanism at work in how types work on data exposing the duality between type conformance and pattern-matching
- extending the reach of types to parametric polymorphism
- providing a framework for cleaning up the semantics of side-effecting or stateful computations and generalizing them

Taken together with the inherent simplicity of functional language design and its compositional nature we have the makings of a revolution in complexity management. This is the real dominating trend in the industry. Once `Java` was within 1.4X the speed of `C/C++` the game was over because `Java` offered such a significant reduction in application development complexity which turned into gains in both

productivity and manageability. Likewise, the complexity of `Java` [3] development and especially `Java` development on Internet-based applications has become nearly prohibitive. Functional languages, especially languages like `Scala` which run on the `JVM` and have excellent interoperability with the extensive `Java` legacy, and have performance on par with `Java` are poised to do to `Java` what `Java` did to `C/C++`.

1.2 Where are we going

With a preamble like that it doesn't take much to guess where all this is heading. More and more we are looking at trends that lead toward more functional and functionally-based web applications. We need not look to the growing popularity of cutting-edge frameworks like `Lift` to see this trend. Both Javascript (with it's origins in Self) and Rails must be counted amongst the functionally influenced.

1.2.1 A functional web

Because there are already plenty of excellent functional web frameworks in the open source community our aim is not to build another. Rather our aim is to supply a set of design patterns that will work with most – in fact are already implicitly at work in many – but that when used correctly will reduce complexity.

Specifically, we will look at the organization of the pipeline of a web-application from the pipeline of HTTP requests through the application logic to the store and back. We will see how in each case judicious use of the monadic design pattern provides for significant leverage in structuring code, making it both simpler, more maintainable and more robust in the face of change.

To that end we will be looking at

- processing HTTP-streams using *delimited* continuations to allow for a sophisticated state management
- parser combinators for parsing HTTP-requests and higher-level application protocols using HTTP as a transport
- application domain model as an abstract syntax
- zippers as a means of automatically generating navigation
- collections and containers in memory

[3] and here we are not picking on `Java`, specifically, the same could be said of `C#`

- storage, including a new way to approach query and search

In each case there is an underlying organization to the computation that solves the problem. In each case we find an instance of the monadic design pattern. Whether this apparent universal applicability is an instance of finding a hammer that turns everything it encounters into nails or that structuring computation in terms of monads has a genuine depth remains to be seen. What can be said even at this early stage of the game is that object-oriented design patterns were certainly proposed for each of these situations and many others. It was commonly held that such techniques were not merely universally applicable, but of genuine utility in every domain of application. The failure of object-oriented design methods to make good on these claims might be an argument for caution. Sober assessment of the situation, however, gives cause for hope.

Unlike the notion monad, objects began as "folk" tradition. It was many years into proposals for object-oriented design methods before there were commonly accepted formal or mathematical accounts. By contrast monads began as a mathematical entity. Sequestered away in category theory the idea was one of a whole zoology of generalizations of common mathematical entities. It took some time to understand that both set comprehensions and algebraic data types were instances monads and that the former was a universal language for the notion. It took even more time to see the application to structuring computations. Progress was slow and steady and built from a solid foundation. This gave the notion an unprecedented level of quality assurance testing. The category theoretic definition is nearly fifty years old. If we include the investigation of set comprehensions as a part of the QA process we add another one hundred years. If we include the forty years of vigorous use of relational databases and the **SELECT−FROM−WHERE** construct in the industry, we see that this was hardly just an academic exercise.

Perhaps more importantly than any of those is the fact that while object-oriented techniques *as realized in mainstream language designs* [4] ultimately failed to be compositional in any useful way – inheritance, in fact, being positively at odds with concurrent composition – the notion of monad is actually an attempt to capture the meaning of composition. As we will see in the upcoming sections, it defines an powerful notion of parametric composition. This is crucial because in the real world *composition is the primary means to scaling* – both in the sense of performance and in the sense of complexity. As pragmatic engineers we manage complexity of scale by building larger systems out of smaller ones. As pragmatic engineers we understand that each time components are required to interface or synchronize we have the potential for introducing performance concerns. The parametric form of composition

[4] To be clear, message-passing and delegation are certainly compositional. Very few mainstream languages support these concepts directly

encapsulated in the notion of monad gives us a language for talking about both kinds of scaling and connecting the two ideas. It provides a language for talking about the interplay between the composition of structure and the composition of the flow of control. It encapsulates stateful computation. It encapsulates data structure. In this sense the notion of monad is poised to be the rational reconstruction of the notion of object. Telling this story was my motivation for writing this book.

1.2.2 DSL-based design

It has become buzz-word du jour to talk about DSL-based design. So much so that it's becoming hard to understand what the term means. In the functional setting the meaning is really quite clear and since the writing of the Structure and Interpretation of Computer Programs (one of the seminal texts of functional programming and one of the first to pioneer the idea of DSL-based design) the meaning has gotten considerably clearer. In a typed functional setting the design of a collection of types tailor-made to model and address the operations of some domain is the basis is effectively the design of an abstract syntax of a language for computing over the domain.

To see why this must be so, let's begin from the basics. Informally, DSL-based design means we express our design in terms of a little mini-language, tailor-made for our application domain. When push comes to shove, though, if we want to know what DSL-based design means in practical terms, eventually we have to ask what goes into the specification of a language. The commonly received wisdom is that a language is comprised of a *syntax* and a *semantics*. The syntax carries the structure of the expressions of the language while the semantics says how to evaluate those expressions to achieve a result – typically either to derive a meaning for the expression (such as this expression denotes that value) or perform an action or computation indicated by the expression (such as print this string on the console). Focusing, for the moment, on syntax as the more concrete of the two elements, we note that syntax is governed by *grammar*. Whether we're building a concrete syntax, like the `ASCII` strings one types to communicate `Scala` expressions to the compiler or building an abstract syntax, like the expression trees of `LINQ`, syntax is governed by grammar.

What we really want to call out in this discussion is that a collection of types forming a model of some domain is actually a grammar for an abstract syntax. This is most readily seen by comparing the core of the type definition language of modern functional languages with something like `EBNF`, the most prevalent language for defining context-free grammars. At their heart the two structures are nearly the same. When one is defining a grammar one is defining a collection of types that

model some domain and vice versa. This is blindingly obvious in `Haskell`, and is the essence of techniques like the application of two-level type decomposition to model grammars. Moreover, while a little harder to see in `Scala` it is still there. It is in this sense that typed functional languages like `Scala` are very well suited for DSL-based design. To the extent that the use of `Scala` relies on the functional core of the language (not the object-oriented bits) virtually every domain model is already a kind of DSL in that it's types define a kind of abstract syntax.

Taking this idea a step further, in most cases such collections of types are actually representable as a monad. Monads effectively encapsulate the notion of an algebra – which in this context is a category theorist's way of saying a certain kind of collection of types. If you are at all familiar with parser combinators and perhaps have heard that these too are facilitated with monadic composition then the suggestion that there is a deeper link between parsing, grammars, types and monads might make some sense. On the other hand, if this seems a little too abstract it will be made much more concrete in the following sections. For now, we are simply planting the seed of the idea that monads are not just for structuring side-effecting computations.

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, rendered as `Scala` code, 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

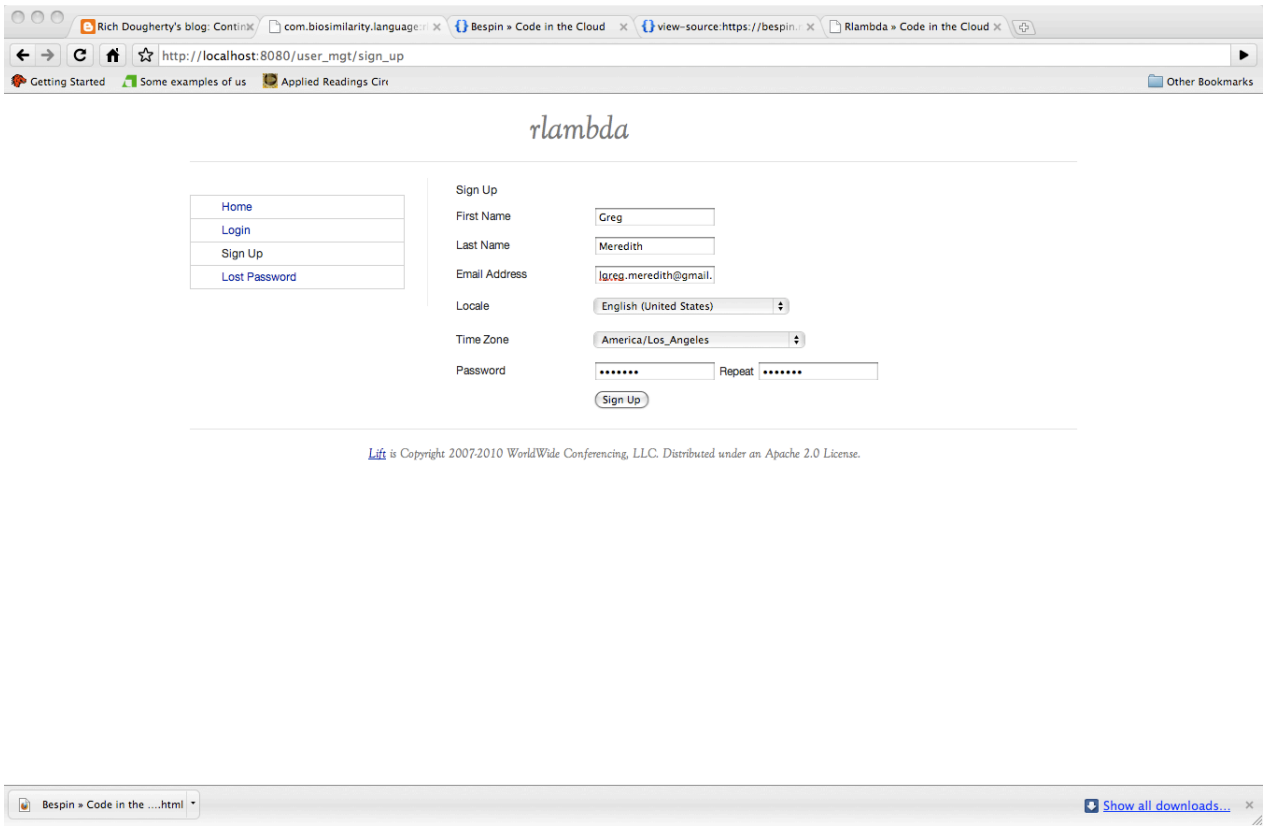


Figure 1.1: Example sign up page

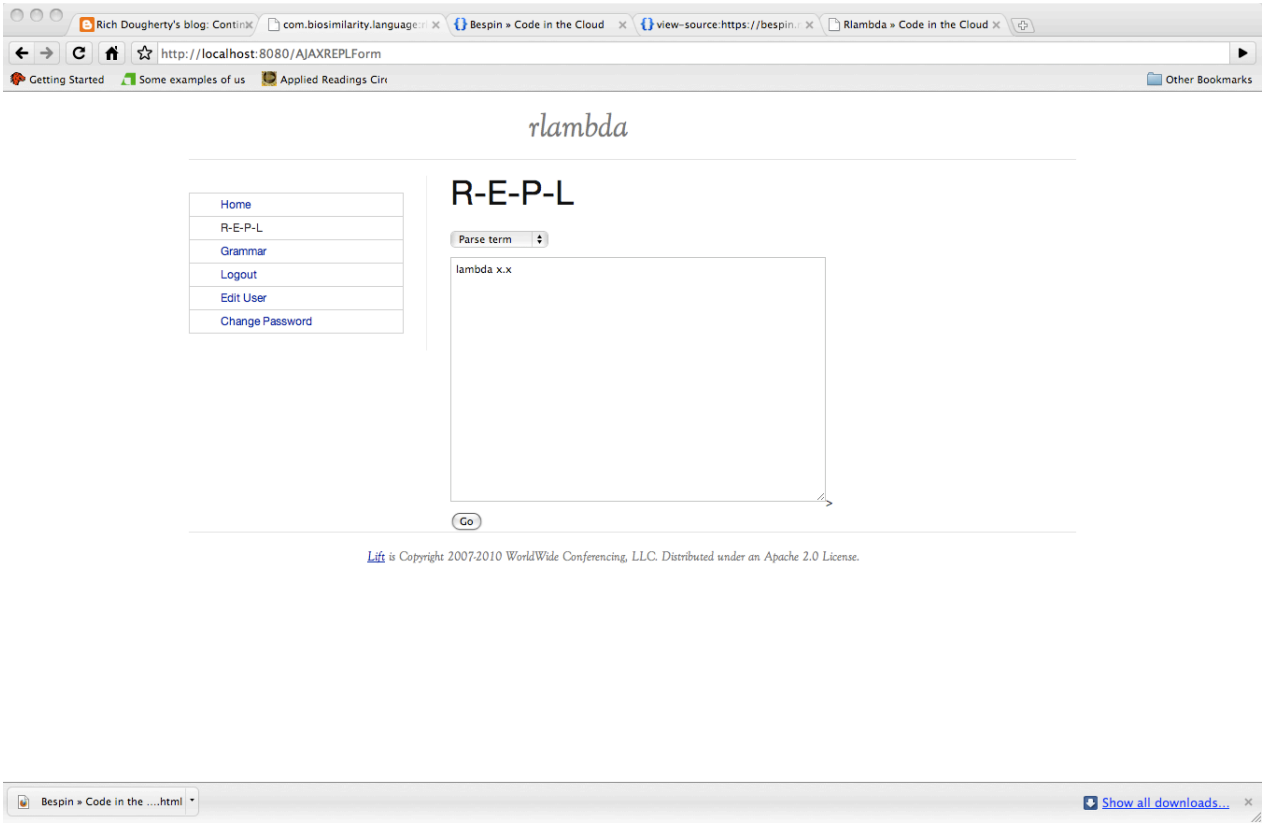


Figure 1.2: Example REPL page

These core capabilities wrap around our little toy programming language in much the same way a modern IDE might wrap around development in a more robust, full-featured language. Hence, we want the capabilities of the application to be partially driven from the specification of our toy language. For example, if we support some syntax-highlighting, or syntax-validation on the client, we want that to be driven from that language spec to the extent that changes to the language spec ought to result in changes to the behavior of the highlighting and validation. Thus, at the center of our application is the specification of our toy language.

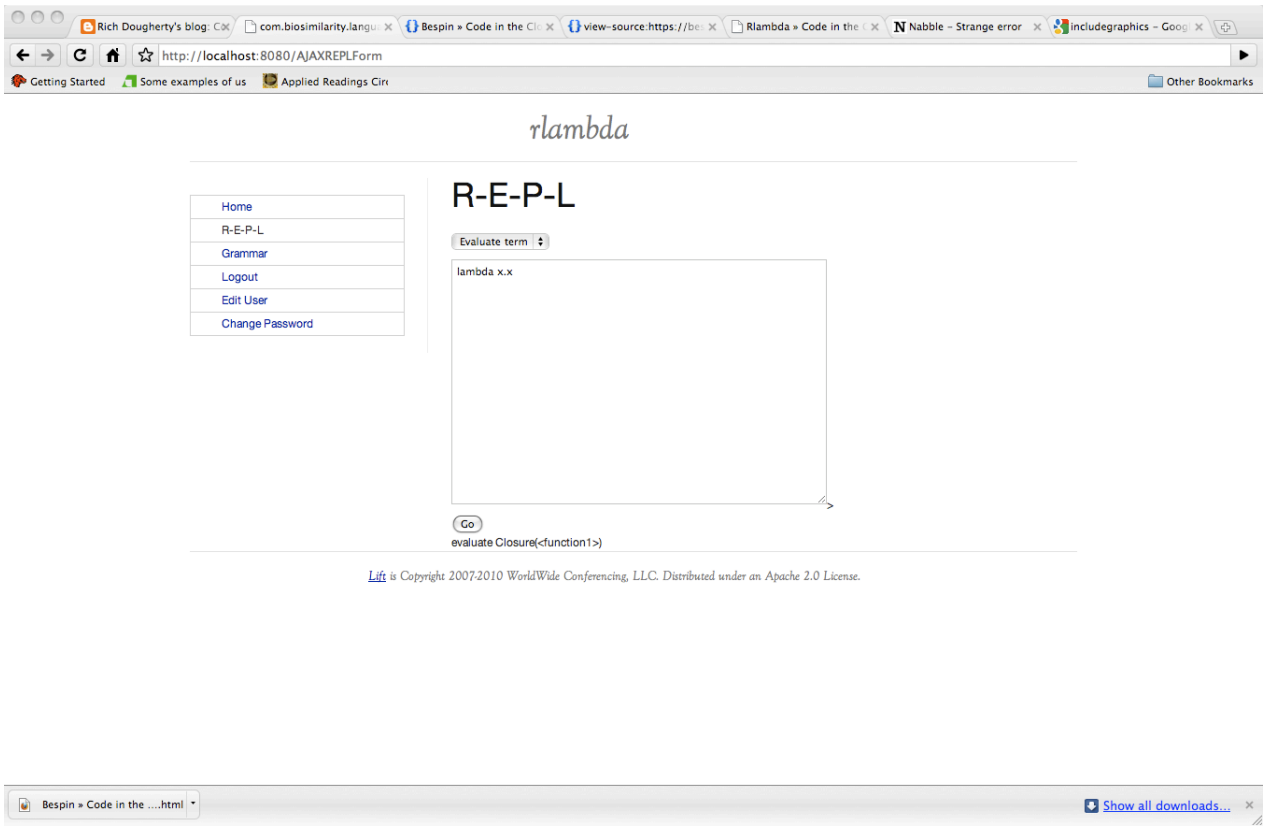


Figure 1.3: Example evaluation result page

Our toy language

Abstract syntax Fittingly for a book about `Scala` we'll use the λ-calculus as our toy language. [5] The core *abstract* syntax of the lambda calculus is given by the following *EBNF* grammar.

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

Informally, this is really a language of pure variable management. For example, if the expression M mentions x, then $\lambda x.M$ turns x into a variable in M and provides a means to substitute values into M, via application. Thus, $(\lambda x.M)N$ will result in a new term, sometimes written $M[N/x]$, in which every occurrence of x has been replaced by an occurrence of N. Thus, $(\lambda x.x)M$ yields M, illustrating the implementation in the λ-calculus of the identity function. It turns out to be quite remarkable what you can do with pure variable management.

Concrete syntax We'll wrap this up in concrete syntax.

EXPRESSION	MENTION	ABSTRACTION	APPLICATION
$M, N ::=$	x	$\mid$ `(`x_1`,...,`x_k`) =>` M	$\mid M$`(`N_1`,...,`N_k`)`

LET	SEQ	GROUP
$\mid$ `val` x `=` M`;`N	$\mid M$`;`N	$\mid$ `{` M `}`

It doesn't take much squinting to see that this looks a lot like a subset of `Scala`, and that's because – of course! – functional languages like `Scala` all share a common core that is essentially the λ-calculus. Once you familiarize yourself with the λ-calculus as a kind of design pattern you'll see it poking out everywhere: in `Clojure` and `OCaml` and `F#` and `Scala`. In fact, as we'll see later, just about any DSL you design that needs a notion of variables could do worse than simply to crib from this existing and well understood design pattern.

[5] A word to the wise: even if you are an old hand at programming language semantics, even if you know the λ-calculus like the back of your hand, you are likely to be surprised by some of the things you see in the next few sections. Just to make sure that everyone gets a chance to look at the formalism as if it were brand new, a few recent theoretical developments have been thrown in. So, watch out!

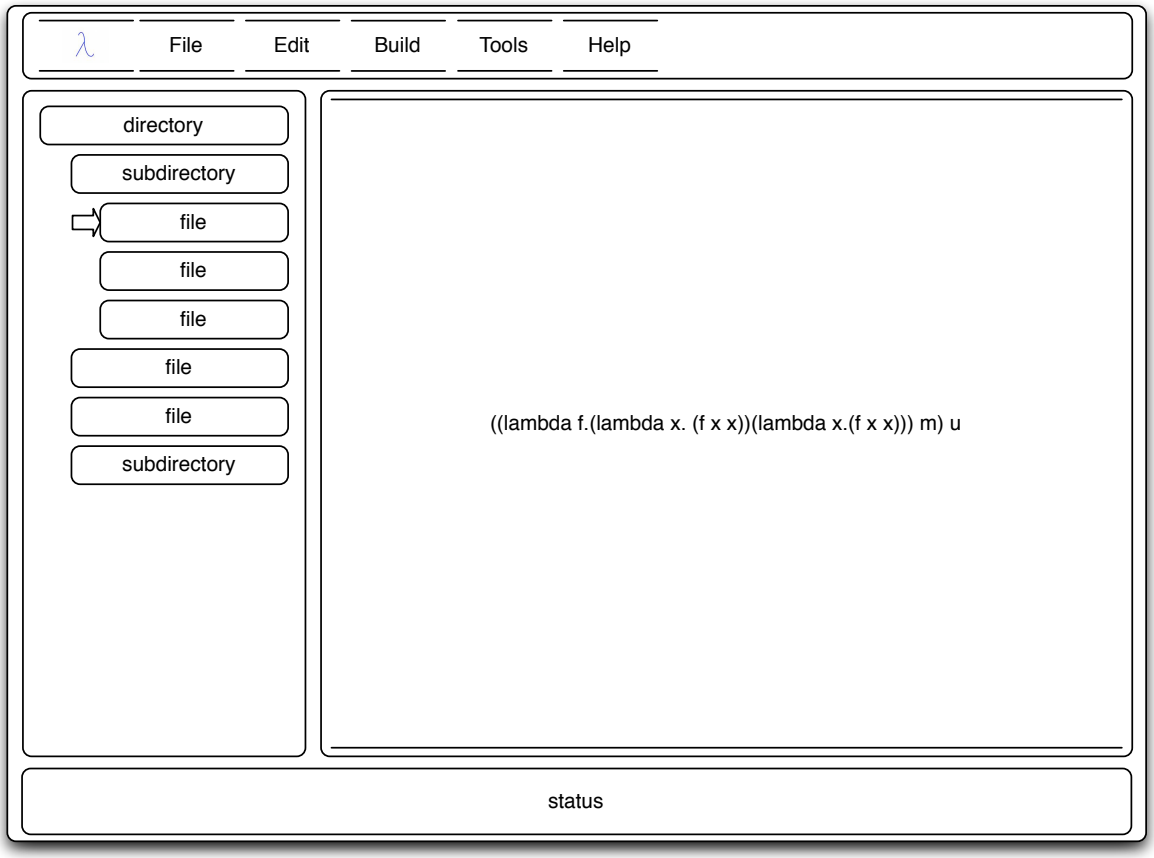


Figure 1.4: Project and code editor

Code editor

Project editor

Advanced features

1.3.2 Chapter map

Taking a step back from the technical discussion let's recall what we plan to cover and how we plan to cover it. Essentially, the book is organized to follow the processing of `HTTP` requests from the browser through the server and application code out to the store and back.

- Chapter two introduces terminology, notation and concepts necessary for the rest of the book.
- Chapter three looks at the organization of an `HTTP` server.
- Chapter four investigates parsing the transport and application level requests.
- Chapter five focuses on the application domain model.
- Chapter six addresses at the navigation model.
- Chapter seven reviews collections.
- Chapter eight looks at the storage model.
- Chapter nine investigates deployment of the application.
- Chapter ten addresses new foundations for semantic query.

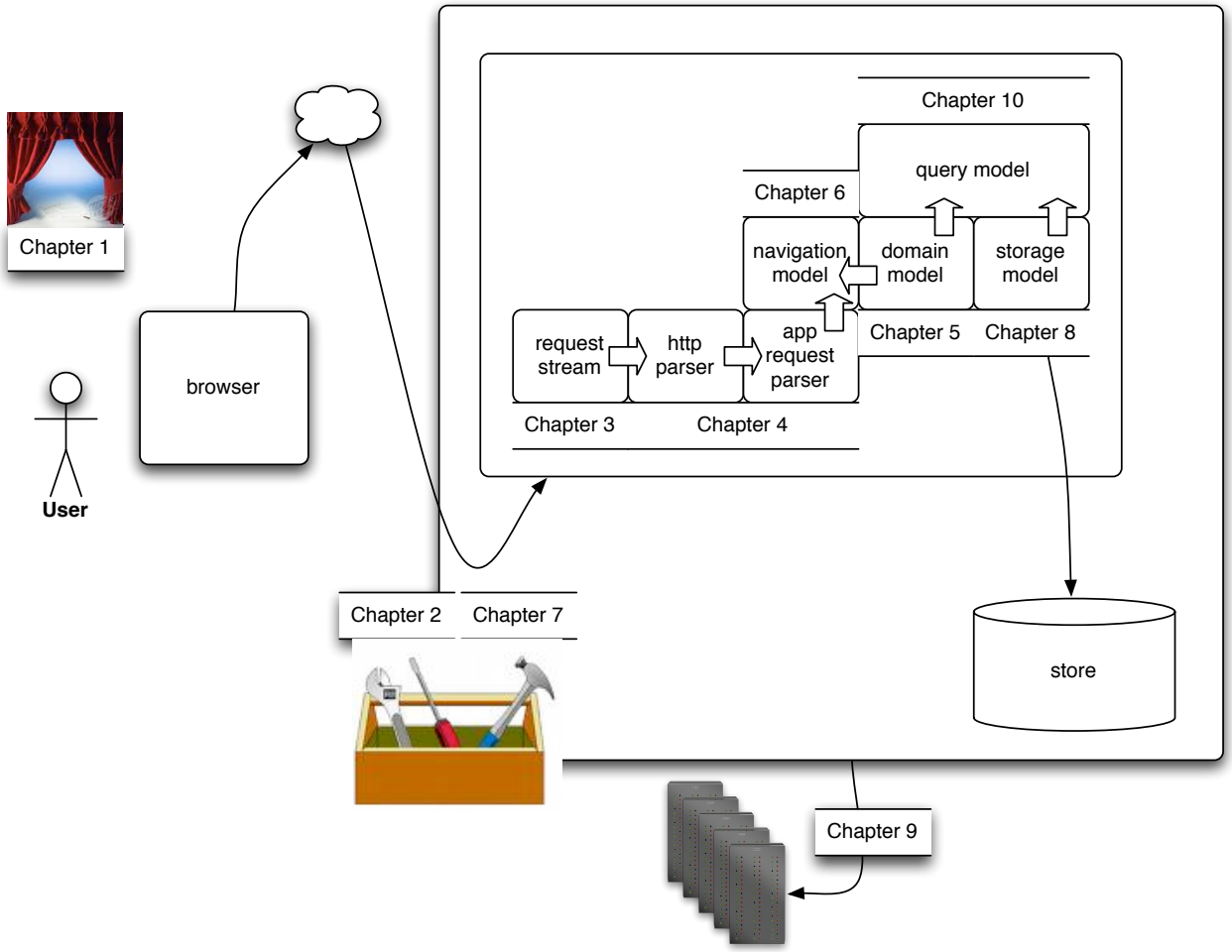


Figure 1.5: Chapter map

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.

2.1.1 Scala

2.1.2 Maths

2.2 Introduction to core design patterns

2.2.1 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 – the branch of mathematics 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, a monad's most central piece of data is a parametric type constructor, say T.

Haskell's monad API 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)

Do-notation One of the driving motivations for this particular formulation of the concept is that it makes it very easy to host a little DSL inside the language. The syntax and semantics of the DSL is simultaneously given by the following procedure for de-sugaring, i.e. translating expressions in the DSL back to core `Haskell`.

```
do { x } = x

do { x ; <stmts> }
= bind x (\_ -> do { <stmts> })

do { v <- x ; <stmts> }
= bind x (\v -> do { <stmts> })

do { let <decls> ; <stmts> }
= let <decls> in do { <stmts> }
```

Listing 2.2: do-notation de-sugaring

The assignment-like operation extends to full pattern matching with

```
do { p <- x ; <stmts> }
= let f p = do { <stmts> }
            f _     = fail "..."
  in bind x f
```

On the face of it, the notation provides both a syntax and a semantics reminiscent of the standard side-effecting operations of mainstream imperative languages. In presence of polymorphism, however, these instruments are much more powerful. These operations can be *systematically* "overloaded" (meaning the overloaded definitions satisfy the laws above). This allows to systematically use the notation for a wide variety of computations that all have some underlying commonality. Typical examples include I/O, state management, control flow (all three of which all bundle up in parsing), and also container navigation and manipulation. It gets better for many of the tools of mathematics that are regularly the subject of computer programs such probability distributions, integration, etc., also have presentations as monads. Thus, innocent examples like this one

```
do { putStrLn "Enter a line of text:";
     x <- getLine;
     putStrLn ("you wrote: " ++ x) }
```

as might be found in some on-line tutorial on monads belie the potency of this combination of ideas.

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

The basic `Scala` construct looks like

```
for( p <- e [; p <- e] [p = e] [if t] ) yield { e }
```

and the de-sugaring looks like

```
for( x <- expr1 ; y <- expr2 ; <stmts> )
yield expr3
=
expr1 flatMap(
           x => for( y <- expr2; <stmts> ) yield expr3
         )

for( x <- expr1 ; y = expr2 ; <stmts> )
yield expr3
=
for( ( x, y ) <- for ( x <- expr1 ) yield ( x, expr2 );
     <stmts> )
yield expr3

for( x <- expr1 if pred ) yield expr2
=
expr1 filter ( x => pred ) map ( x => expr2 )
```

Listing 2.3: for-comprehension de-sugaring

Again, general pattern matching is supported in assignment-like statements.

```
for( p <- expr1 ; <stmts> ) yield expr2

=

expr1 filter {
  case p => true
  case _ => false
} flatMap {
  p => for( <stmts> ) yield expr2
}
```

This means, therefore, inside the appropriate code context (i.e., a **do**-block or a **for**-comprehension, respectively) we have the following correspondence

```
ptn <- expr          ptn <- expr
return expr          yield expr
```

with a kind of spiritual kinship between expr_1 << expr_2 and expr_1 ; expr_2.

2.3 Variations in presentation

2.3.1 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

```
trait Monoid {
  def unit : Monoid
  def mult( that : Monoid )
}
```

This might allow *views* of Int as a monoid as in

```
class MMultInt extends Int with Monoid {
  override def unit = 1
  override def mult( that : Monoid ) = this * that
  }
```

except for the small problem that Int is **final** (illustrating an important difference between the adhoc polymorphism of `Haskell`'s typeclass and `Scala`'s **trait**).

Any solution will depend on type parametrization. For example

```
trait Monoid[Element] {
  def unit : Element
  def mult( a : Element, b : Element )
}
```

and corresponding view of Int as a monoid.

```
class MMultInt extends Monoid[Int] {
  override def unit : Int = 1
```

```
  override def mult( a : Int , b : Int ) = a * b
}
```

This parametric way of viewing some underlying data structure is natural both to the modern programmer and the modern mathematician. Both are quite familiar with and make extensive use of overloading of this kind. Both are very happy to find higher levels of abstraction that allow them to remain DRY when the programming demands might cause some perspiration. One of the obvious places where repetition is happening is in the construction of view. Consider another view of Int

```
class MAddInt extends Monoid[Int] {
  override def unit : Int = 0
  override def mult( a : Int , b : Int ) = a + b
}
```

It turns out that there is a lot of machinery that is common to defining a view like this for any given data type. Category theorists realized this and recognized that you could reify the *view* which not only provides a place to refactor the common machinery, but also to give it another level of polymorphism. Thus, a category theorist's view of the monad API might look something like this.

```
trait Monad[Element,M[_]] {
  def unit( e : Element ) : M[Element]
  def mult( mme : M[M[Element]] ) : M[Element]
}
```

The family resemblance to the Monoid API is not accidental. The trick is to bring syntax back into the picture. Here's an example.

```
case class MonoidExpr[Element]( val e : List[Element] )
class MMInt extends Monad[Int,MonoidExpr] {
  override def unit( e : Int ) = MonoidExpr( List( e ) )
  override def mult( mme : MonoidExpr[MonoidExpr[Int]] ) =
  mme match {
    case MonoidExpr( Nil ) =>
      MonoidExpr( Nil )
    case MonoidExpr( mes ) =>
      MonoidExpr(
        ( Nil /: mes)(
          { ( acc, me ) => me match {
              case MonoidExpr( es ) => acc +++ es
            }
          }
        )
```

```
            )
    }
}
```

While it's clear that unit turns Ints into integer expressions, what the operation mult is doing is canonically flattening nested expressions in a way the exactly parallels the flattening of nest arithmetic addition expressions. For a broad class of monads, this is the paradigmatic behavior of mult. The fact that monads are characterized by a generic interpretation of flattening of nested structure, by the way, makes the choice of the term flatMap particularly appropriate.

Associativity as flattening Looking at it from the other way around, one of the properties of a monoid is that it's binary operation, its mult, is associative. The actual content of the notion of associativity is that order of grouping doesn't make any difference. In symbols, a binary operation, $*$, is associative when $a * (b * c) = (a * b) * c$. This fact gives us the right to erase the parens and simply write $a * b * c$. In other words, associativity is flattening. A similar connection can be made for unit and the identity of a monoid. One quick and dirty way to see this is that since we know that $a * e = a$ (when e is the unit of the monoid) then the expression $a * e$ effectively nests a in a MonoidExpr. That's the "moral" content of the connection between the two notions of unit.

Bracing for `XML` In this connection it is useful to make yet another connection to a ubiquitous technology, namely `XML`. As a segue, notice that we can always write a binary operation in prefix notation as well as infix. That is, whatever we could write at $a * b$ we could just as easily write as $*(a, b)$. The flattening property of associativity says we can drop nesting such as $*(a, *(b, c))$ in favor of $*(a, b, c)$. In this sense, the syntax of braces is a kind of generic syntax for monoids and monads. If we introduce the notion of "colored" braces, this becomes even more clear at the lexicographic or notational level. So, instead of $*(a, b, c)$ we'll mark the "color" of the braces like so: $(*|...|*)$, where $*$ can be any color. Then, at the level of monoid the unit is the empty braces, $(*||*)$, while at the level of the monad the unit places the element, say a, in between the braces: $(*|a|*)$. The conceptual connection between the two variations of the operation now becomes clear: writing $a * e$ is the same as writing $*(a, e)$ which is the same as writing $(*|a, (*||*)|*)$, which canonically flattens into $(*|a|*)$.

Now, anyone who's spent any time around `XML` can see where this is headed. At a purely syntactic, lexicographic level we replace round brackets with angle brackets and we have exactly `XML` notation for elements. In this sense, `XML` is a kind of universal notation for monads. The only thing missing from the framework is a means

to associate operations to unit and mult, i.e. to inserting content into elements and flattening nested elements. `Scala`'s specific support for `XML` puts it in an interesting position to rectify this situation.

The connection with set-comprehensions Finally, since we've gone this far into it, we might as well make the connection to comprehensions. Again, let's let notation support our intuitions. The above discussion should make it clear that its not the particular shape of the brace that matters, but the action of "embracing" a collection of elements that lies at the heart of the notion. So, it's fine if we shift to curly braces to be suggestive. Thus, we are looking at a formalism that allows us to polymorphically "collect" elements between braces, like $\{*|a, b, c|*\}$.

This is fine for finite collections, but what about infinitary collections or collections of elements selected programmatically, rather than given explicitly. The set theoretic notation was designed specifically for this purpose. When we have an extant set of elements that we can give explicitly, we simply write $\{a_1, a_2, a_3, ...\}$. When we have a potentially infinitary collection of elements, or elements that are selected on the basis of a condition, then we write $\{pattern \in S \mid condition\}$. The idea of monad as comprehension recognizes that these operations of collecting, pattern matching and selection on the basis of a condition can be made *polymorphic* using monads. Notationally, we can denote the different polymorphic interpretations by the "color" of the brace. In other words, we are looking at a shift of the form

- $\{a_1, a_2, a_3, ...\} \mapsto \{*|\ a_1, a_2, a_3, ...\ |*\}$
- $\{pattern\ \in\ S \mid condition\} \mapsto \{*|\ pattern \in S \mid condition\ |*\}$

to build into our notation an explicit representation of the fact that the operation of collection, pattern matching and filtering on the basis of predicate are polymorphic. [1]

Often times, good mathematics, like good programming is really about the design of good notation – it's about DSLs! In this case, the notation is particularly useful *because* it begs the question of the language of patterns and the language of conditions – something that Wadler's original paper on monads as generalized comprehensions did not address. This is a theme to which we will return at the end

[1] This demarcation between extensionally and intensionally given expressions is also reflected in the notation used for arithmetic or monoids, more generally. When we have a finite number and/or explicitly given set of operands, we can write expressions like $a_1 + a_2 + ... + a_n$, but when we have an infinite expression (like and infinite series) or an expression whose operands are given programmatically we write expressions like $\sum_{i \in S} e(i)$.

of the book when we address search on a semantics basis. For now, the central point is to understand how monad as container and monad as generalization of monoid are actually views of the same underlying idea.

Now, just to make sure the connection is absolutely explicit, there is a one-for-one correspondence between the polymorphic set-comprehension notation and the **for**-comprehension notation of `Scala`. The correspondence takes $\{*|pattern \in S \mid condition|*\}$ to

```
for( x <- S if condition ) yield {
  x match { case pattern => x }
}
```

As the `Scala` type checker will explain, this translation is only approximate. If the pattern is refutable, then we need to handle the **case** when the match is not possible. Obviously, we just want to throw those away, so a fold might be a better a better choice, but then that obscures the correspondence.

Syntax and containers The crucial point in all of this is that *syntax is the only container we have for computation.* What do we mean by this? Back when Moggi was crafting his story about the application of the notion of monad to computing he referred to monads as "notions of computation". What he meant by that was that monads reify computation (such as I/O or flow of control or constructing data structures) into "objects". Computation as a phenomenon, however, is both dynamic and (potentially) infinitary. At least as we understand it today, it's not in the category of widgets we can hold in our hand like an apple or an Apple ™ computer. All we can do is *point* to it, indicate it in some way. Syntax, it turns out, is our primary means of signifying computation. That's why many monads factor out as a reification of syntax, and why they are so key to DSL-based design.

Decomposition of monad requirements

In the presentation of the monad API that we've discussed here the constraints on any given monad candidate are well factored into three different kinds of requirements – operating at different levels of the "API", dubbed in order of abstraction: 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

One of the principle challenges of presenting the categorical view of monads is the dependencies on the ambient theory. In some sense the categorical view of the monad API is like a useful piece of software that drags in a bunch of other libraries. A complete specification of monad from the categorical point of view requires providing definitions for

- category
- functor
- natural transformation

This book is not intended to be a tutorial on category theory. There are lots of those and Google and Wikipedia are your friends. Rather, this book is about a certain design pattern that can be expressed, and originally was expressed within that theory, but is to a great extent an independent notion. [2] On the other hand, for the diligent and/or curious reader a pointer to that body of work has the potential to be quite rewarding. There are many treasures there waiting to be found. For our purposes, we strike a compromise. We take the notion of category to be given in terms of the definable types within the `Scala` type system and the definable programs (sometimes called maps) between those types. Then a functor, say F, is a pair consisting of a parametric type constructor, F_T, together with a corresponding action, say F_M, on programs, that respects certain invariants. Specifically,

- A functor must preserve identity. That is, for any type, A, we can define an identity map, given canonically by the program (x : A) =>x. Then F_M((x : A) => x) = (x : F_T[A]) => x
- A functor must preserve composition. That is, given two programs, f : A =>B and g : B =>C, F_M(f ∘ g) = F_M(f) ∘ F_M(g) where (f ∘ g)(x) = g(f(x))

In `Scala`-land this is what it means for F =(F_T,F_M) to be *functorial*. The constraint itself is called *functoriality*. Sometimes we will refer to the tuple (F_T,F_M) just by F when there is little risk of confusion.

From these operational definitions, it follows that a natural transformation is map between functors! We expect it to be given in terms of component maps. That

[2] In point of fact, at present writing, i suspect that there is a way to turn category theory on its head and make the notion of monad as the fundamental building block out of which the rest of category theory may be defined.

is, at a type, say A, a natural transformation, n from a functor F to a functor G should have a map $n_A : F_T[A] => G_T[A]$. These component maps need to satisfy some constraints. To wit,

- Suppose we have a map f : A =>B. Then we want $n_A \circ G_M(f) = F_M(f) \circ n_B$.

As you might have guessed, this constraint is dubbed *naturality*. Category theorists have developed a nice methodology for reasoning about such constraints. They draw them as diagrams. For example, the diagram below represents the naturality equation.

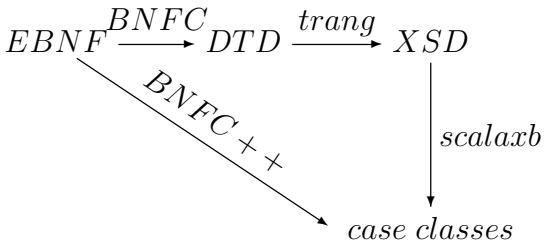
$$\begin{array}{ccc} F_T[A] & \xrightarrow{n_A} & G_T[A] \\ {\scriptstyle F_M(f)}\downarrow & & \downarrow{\scriptstyle G_M(f)} \\ F_T[B] & \xrightarrow[n_B]{} & G_T[B] \end{array}$$

You can read the diagram as stating that the two paths from the upper left corner of the diagram to the lower right corner (one along the top and down the right and the other down the left and along the bottom) must be equal as functions. In general, when all the path between two vertices in such a diagram are equal as functions, the diagram is said to commute. This sort of tool is really invaluable for people doing systems-level design.

Quick digression about design by diagram-chasing In preparation for this book i was looking at the open source `Scala` library, `kiama`. This package provides support for rewrite systems, making it possible to give a specification of systems like the *lambda*-calculus at a level more closely resembling its original specification. That system makes the choice of using `Scala` **case** classes as the input language for the terms in rewrite clauses. i wanted to use it over the `Java` classes generated from a parser written in `BNFC`. This meant i needed a **case** class shadow of the `BNFC`-generated `Java`-based abstract syntax model. It just turns out that `BNFC` also generates a `DTD`-driven `XML` parser for element structure that is isomorphic to the input grammar. There are open source tools that will generate an `XSD` schema from the `DTD`. This can then be fed into the open source tool `scalaxb` which will generate **case** classes. In pictures

$$EBNF \xrightarrow{BNFC} DTD \xrightarrow{trang} XSD \xrightarrow{scalaxb} case\ classes$$

Chaining through the open source components (maps in our category) to find a way to wire in the `kiama` functionality is a lot like diagram chasing, which feels like it was made for an open source world. Moreover, when `BNFC` eventually targets `Scala` directly, we have a quality assurance constraint. Up to some accepted variance in output format we want



Often developers will draw similar diagrams, intuitively attempting to convey similar information; but, just as often, because of their informality, these diagrams are just as much a source of mis-communication. It is possible, however, in a language such as `Scala` to get both more formal and more intuitive (by appealing to a higher level of abstraction, like diagram-chasing) at the same time.

Monads are triples Returning to the topic at hand, a monad is really given by a triple[3], (S, unit, mult) where

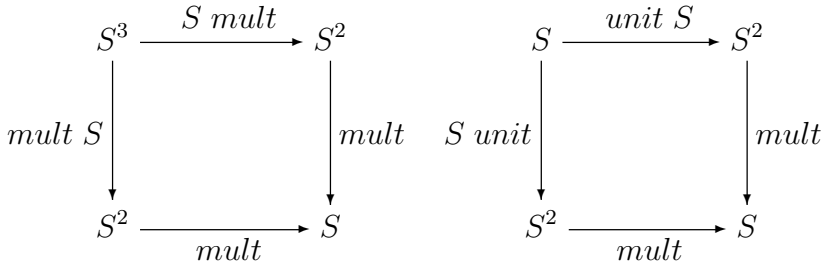
- S is a functor,
- unit is a natural transformation from the identity functor to S,
- mult is a natural transformation from S^2 to S.

subject to the following constraints.

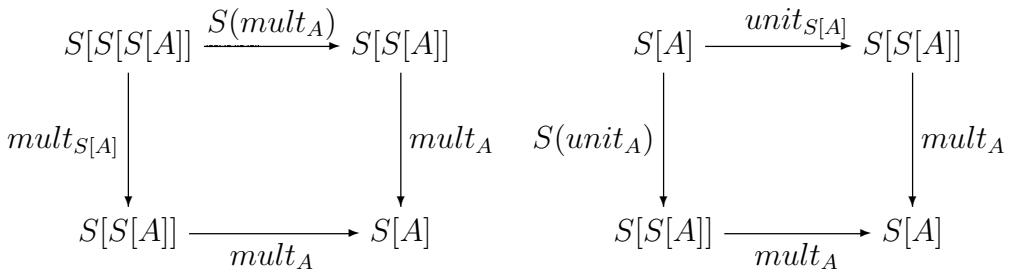
- mult ∘ S mult = mult ∘ mult S
- mult ∘ S unit = mult ∘ unit S

[3]In fact, in the early days of category theory they were actually give the imaginative moniker: triple.

Or in pictures



which are really shorthand for



These constraints are called coherence constraints because they ensure that unit and mult interact coherently (and, in fact, that mult interacts coherently with itself).

`Scala` programmers can certainly understand these laws. Once you observe that nesting of containers corresponds to iterated invocation of the functor associated with a monad, then it's easy to see that the first diagram merely expresses that there is a canonical way to flatten nested monadic structure. The second diagram says that whichever way you try to nest with the unit transform, applying the mult after results in the same flattened structure.

Despite the apparent complexity, the presentation has a certain organization to it that is very natural once it becomes clear. There are actually three different levels in operation here, following a kind of food-chain.

- At the level of `Scala`, which – if you recall – is our ambient category, we find types and maps between them.
- Though this is harder to see because we have restricted our view to just *one* category, at the level of functors, *categories* play in the role of types, while *functors* play in the role of maps between them.
- At the level of natural transformations, functors play in the role of types while natural transformations play in the role of maps between them.

Correspondingly, we have three different levels of constraints.

- functoriality
- naturality
- coherence

Monads bring all three levels together into one package. Monads operate on a category via a functor and pair of natural transformations that interact coherently. This food chain arrangement points the way toward an extremely promising reconstruction of the notion of interface. One way to think about it is in terms of the recent trend away from inheritance and towards composition. In this trend the notion of interface is still widely supported, but it really begs the question: what is an interface? What makes a collection of functions cohere enough to be tied together under an interface?

One way to go about answering that question is to assume there's nothing but the interface name that collects the functions it gathers together. In that case, how many interfaces are there? One way to see that is just to consider all the sub interfaces of a single interface with n methods on it: that's 2^n interfaces. That's a lot. Does that give us any confidence that any one way of carving up functionality via interfaces is going to be sane? Further, in practice, do we see random distribution through this very large space?

What we see over and over again in practice is that the answer to the latter question is "no!" Good programmers invariably pick out just a few factorizations of possible interfaces – from the giant sea of factorizations. That means that there is something in the mind of a good programmer that binds a collection of methods together. What might that something be? i submit that in their minds there are some constraints they know or at least intuit must hold across these functions. The evidence from category theory is that these are not just arbitrary constraints, but that the space of constraints that bind together well factored interfaces is organized along the lines of functoriality, naturality and coherence. There may yet be higher-order levels of organization beyond that, but these – at least – provide a well-vetted and practical approach to addressing the question of what makes a good interface. If monad is the new object, then these sorts of categorical situations (of which monad is but one instance) are the basis for a re-thinking what we mean when we say "interface".

All of this is discussion leads up to the context in which to understand the correspondence between the `Haskell` variation of the monad laws and their original presentation.

```
trait Monad[M[_]] {
   // map part of M
   // part of the requirement of M's functoriality
   // M : Scala ⇒ Scala
   def map[A,B]( a2b : A ⇒ B ) : M[A] ⇒ M[B]
   // the unit natural transformation, unit : Identity ⇒ M[A]
   def unit[A]( a : A ) : M[A]
   // the mult natural transformation, mult : M[M[A]] ⇒ M[A]
   def mult[A]( mma : M[M[A]] ) : M[A]

   // flatMap, aka bind is a derived notion
   def flatMap[A,B]( ma : M[A], a2mb : A ⇒ M[B] ) : M[B] = {
      mult( map( a2mb )( ma ) )
   }
}
```

Listing 2.4: categorical presentation of monad as Scala trait

Chapter 3

An IO-monad for http streams

Code first; questions later

The following code is adapted from Tiark Rompf's work using delimited continuations for handling HTTP streams.

3.1 Code first, questions later

```
import scala.continuations._
import scala.continuations.ControlContext._

import scala.concurrent._
import scala.concurrent.cpsops._


import java.net.InetSocketAddress
import java.net.InetAddress

import java.nio.channels.SelectionKey
import java.nio.channels.Selector
import java.nio.channels.SelectableChannel
import java.nio.channels.ServerSocketChannel
import java.nio.channels.SocketChannel
import java.nio.channels.spi.SelectorProvider

import java.nio.ByteBuffer
import java.nio.CharBuffer
```

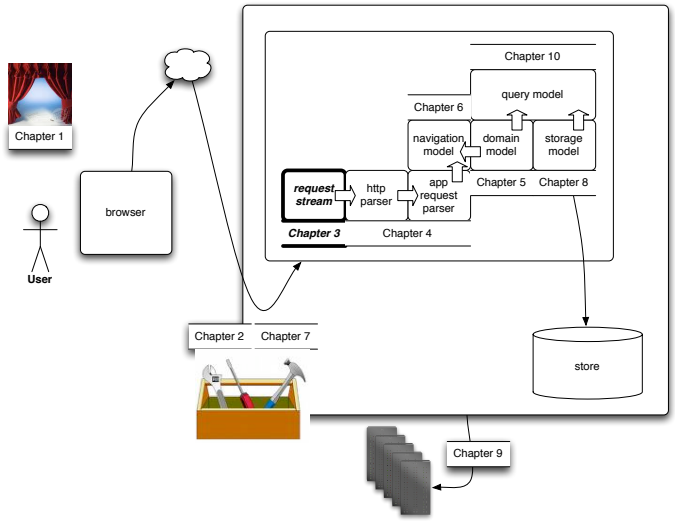


Figure 3.1: Chapter map

```
import java.nio.charset.Charset
import java.nio.charset.CharsetDecoder
import java.nio.charset.CharsetEncoder

import java.util.regex.Pattern
import java.util.regex.Matcher

import java.util.Set

import scala.collection.JavaConversions._

// adapted from http://vodka.nachtlicht-media.de/tutHttpd.html

object DCWebserver
 extends FJTaskRunners {

  case class Generator[+A,-B,+C](val fun: (A => (B @cps[Any,Any])) => (C @c

    def copy = null // FIXME: workaround for named/default params bug

    final def foreach(f: (A => B @cps[Any,Any])): C @cps[Any,Any] = {
      fun(f)
    }
```

```
  }

  def selections( selector: Selector )
  : ControlContext[Set[SelectionKey], Unit, Unit] =
  shiftR {
    k: (Set[SelectionKey] => Any) =>

    println("inside select")

      while(true) { // problem???
        val count = selector.selectNow()
        if (count > 0)
          k(selector.selectedKeys())
      }
  }

  def createAsyncSelector() = {
    val selector = SelectorProvider.provider().openSelector()

    // TODO: this should run in its own thread, so select can block safel
    spawn {
      selections(selector).fun {
        keySet =>
          for (key <- keySet) {
            println("Select: " + key)
            val handler = key.attachment().asInstanceOf[(SelectionKey =>

            println("handling: " + handler)
            handler(key)
          }
        keySet.clear()
      }
    }

    selector
  }


  def callbacks(channel: SelectableChannel, selector: Selector, ops: Int)
    Generator {
      k: (SelectionKey => Unit @cps[Any,Any]) =>
```

```
        println("level 1 callbacks")

      shift {
        outerk: (Unit => Any) =>

          def callback(key: SelectionKey) = {
            key.interestOps(0)

            spawn {
              println("before continuation in callback")

              k(key)

              println("after continuation in callback")

              if (key.isValid()) {
                key.interestOps(ops)
                selector.wakeup()
              } else {
                outerk()
                //returnto.gen();
              }
            }
          }

        println("before registering callback")

        val selectionKey = channel.register(selector, ops, callback _)

        println("after registering callback")
        // stop
        ()
      }
    }

  def acceptConnections(selector: Selector, port: Int) =
    Generator {
      k: (SocketChannel => Unit @cps[Any,Any]) =>
        val serverSocketChannel = ServerSocketChannel.open()
        serverSocketChannel.configureBlocking(false)
```

```
      val isa = new InetSocketAddress(port)

      serverSocketChannel.socket().bind(isa)

      for (
        key <-
        callbacks( serverSocketChannel, selector, SelectionKey.OP_ACCEP
      ) {

        val serverSocketChannel =
          key.channel().asInstanceOf[ServerSocketChannel]

        val socketChannel = serverSocketChannel.accept()
        socketChannel.configureBlocking(false)

        k(socketChannel)
      }

    println("accept returning")
  }


  def readBytes(selector: Selector, socketChannel: SocketChannel) =
    Generator {
      k: (ByteBuffer => Unit @cps[Any,Any]) =>
        shift {
          outerk: (Unit => Any) =>
            reset {
              val bufSize = 4096 // for example...
              val buffer = ByteBuffer.allocateDirect(bufSize)

              println("about to read")

              for (
                key
                <- callbacks(
                  socketChannel, selector, SelectionKey.OP_READ
                )
              ) {

                println("about to actually read")
```

```
                val count = socketChannel.read(buffer)

                if (count < 0) {
                  println("should close connection")
                  socketChannel.close()

                  println("result of outerk " + outerk())
                  //returnto.gen() should cancel here!
                } else {

                  buffer.flip()

                  println("about to call read cont")

                  k(buffer)

                  buffer.clear()
                  shift { k: (Unit=>Any) => k() }
                }
              }

              println("readBytes returning")
              outerk()
            }
        }
    }


  def readRequests(selector: Selector, socketChannel: SocketChannel) =
    Generator {
      k: (String => Unit @cps[Any,Any]) =>

        var s: String = "";

        for ( buf <- readBytes( selector, socketChannel ) ) {
          k("read: " + buf)
        }
    }
```

```
  def writeResponse(
    selector: Selector,
    socketChannel: SocketChannel,
    res: String
  ) = {
    val reply = res

    val charset = Charset.forName("ISO-8859-1")
    val encoder = charset.newEncoder()

    socketChannel.write(encoder.encode(CharBuffer.wrap(reply)))
  }


  def handleRequest(req: String) = req


  def test() = {

    val sel = createAsyncSelector()

    println("http daemon running...")

    for (socketChannel <- acceptConnections(sel, 8080)) {

      spawn {
        println("Connect: " + socketChannel)

        for (req <- readRequests(sel, socketChannel)) {

          val res = handleRequest(req)

          writeResponse(sel, socketChannel, res)

          shift { k: (Unit => Any) => k() } // FIXME: shouldn't be needed
        }

        println("Disconnect: " + socketChannel)
      }

      shift { k: (Unit => Any) => k() } // FIXME: shouldn't be needed
```

```
    }

  }


  // def main(args: Array[String]) = {

//      reset(test())

//      Thread.sleep(1000*60*60)   // 1h!
//      //      test.mainTaskRunner.waitUntilFinished()

//    }

}
```

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

4.3.1 Different platforms, different parsers

4.3.2 Different performance constraints, different parsers

4.3.3 Maintainability

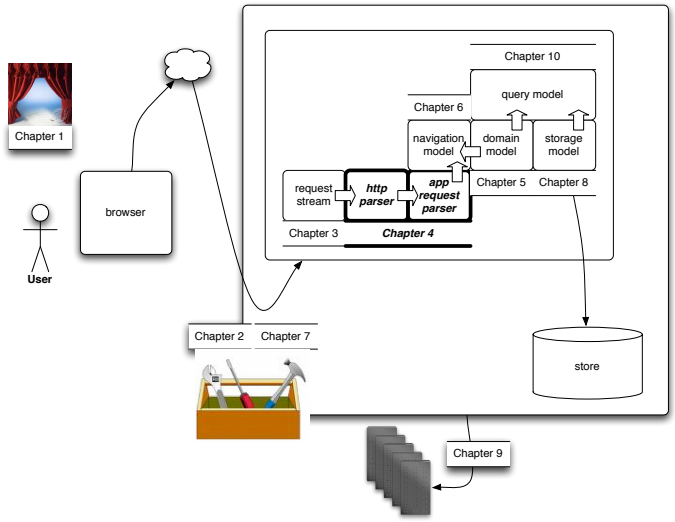


Figure 4.1: Chapter map

```
# line endings
  CRLF = "\r\n";

# character types
  CTL = (cntrl | 127);
  safe = ("$" | "-" | "_" | ".");
  extra = ("!" | "*" | "'" | "(" | ")" | ",");
  reserved = (";" | "/" | "?" | ":" | "@" | "&" | "=" | "+");
  sorta_safe = ("\"" | "<" | ">");
  unsafe = (CTL | " " | "#" | "%" | sorta_safe);
  national = any -- (alpha | digit | reserved | extra | safe | unsafe);
  unreserved = (alpha | digit | safe | extra | national);
  escape = ("%" xdigit xdigit);
  uchar = (unreserved | escape | sorta_safe);
  pchar = (uchar | ":" | "@" | "&" | "=" | "+");
  tspecials = ("(" | ")" | "<" | ">" | "@" | "," | ";" | ":" | "\\" | "\"" |

# elements
  token = (ascii -- (CTL | tspecials));

# URI schemes and absolute paths
  scheme = ( alpha | digit | "+" | "-" | "." )* ;
  absolute_uri = (scheme ":" (uchar | reserved )*);
```

```
  path = ( pchar+ ( "/" pchar* )* ) ;
  query = ( uchar | reserved )* %query_string ;
  param = ( pchar | "/" )* ;
  params = ( param ( ";" param )* ) ;
  rel_path = ( path? %request_path (";" params)? ) ("?" %start_query quer
  absolute_path = ( "/"+ rel_path );

  Request_URI = ( "*" | absolute_uri | absolute_path ) >mark %request_uri
  Fragment = ( uchar | reserved )* >mark %fragment;
  Method = ( upper | digit | safe ){1,20} >mark %request_method;

  http_number = ( digit+ "." digit+ ) ;
  HTTP_Version = ( "HTTP/" http_number ) >mark %http_version ;
  Request_Line = ( Method " " Request_URI ("#" Fragment){0,1} " " HTTP_V

  field_name = ( token -- ":" )+ >start_field $snake_upcase_field %write_

  field_value = any* >start_value %write_value;

  message_header = field_name ":" " "* field_value :> CRLF;

  Request = Request_Line ( message_header )* ( CRLF @done );

main := Request;
```

Chapter 5

The domain model as abstract syntax

In which Pooh and Piglet understand the value of pipelines

TBD

5.1 Our abstract syntax

Abstract syntax Fittingly for a book about `Scala` we'll use the λ-calculus as our toy language. [1] The core *abstract* syntax of the lambda calculus is given by the following *EBNF* grammar.

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

Informally, this is really a language of pure variable management. For example, if the expression M mentions x, then $\lambda x.M$ turns x into a variable in M and provides a means to substitute values into M, via application. Thus, $(\lambda x.M)N$ will result in a new term, sometimes written $M[N/x]$, in which every occurrence of x has been replaced by an occurrence of N. Thus, $(\lambda x.x)M$ yields M, illustrating the

[1] A word to the wise: even if you are an old hand at programming language semantics, even if you know the λ-calculus like the back of your hand, you are likely to be surprised by some of the things you see in the next few sections. Just to make sure that everyone gets a chance to look at the formalism as if it were brand new, a few recent theoretical developments have been thrown in. So, watch out!

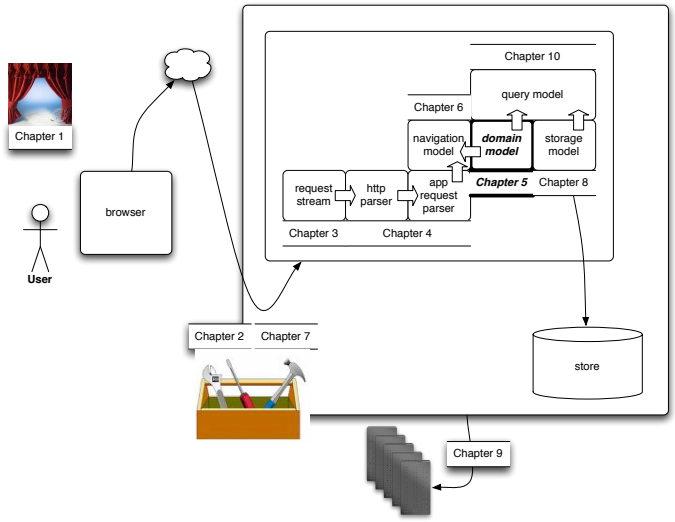


Figure 5.1: Chapter map

implementation in the λ-calculus of the identity function. It turns out to be quite remarkable what you can do with pure variable management.

5.2 Our application domain model

Our toy language

A simple-minded representation At a syntactic level this has a direct representation as the following `Scala` code.

```
trait Expressions {
  type Nominal
  // M, N ::=
  abstract class Expression

  // x
  case class Mention( reference : Nominal )
     extends Expression

  // λ x_1, ..., x_n.M
  case class Abstraction(
     formals : List[Nominal],
     body : Expression
  ) extends Expression

  // MN_1...N_n
  case class Application(
     operation : Expression,
     actuals : List[Expression]
  ) extends Expression
}
```

In this representation each *syntactic category*, EXPRESSION, MENTION, ABSTRACTION and APPLICATION, is represented by a **trait** or **case class**. EXPRESSION's are **trait**'s because they are pure placeholders. The other categories elaborate the syntactic form, and the elaboration is matched by the **case class** structure. Thus, for example, an ABSTRACTION is modeled by an instance of the **case class** called Abstraction having members formal for the formal parameter of the abstraction, and body for the λ-term under the abstraction that might make use of the parameter. Similarly, an APPLICATION is modeled by an instance of the **case class** of the same name having members operation for the expression that will be applied to the actual parameter called (not surprisingly) actual.

Currying The attentive reader will have noticed that there's a difference between the abstract syntax and our `Scala` model. The abstract syntax only supports a single formal parameter under λ-abstraction, while the `Scala` model declares the formals to be of type List[Nominal]. The model anticipates the encoding $\lambda\ x\ y.M \stackrel{def}{=} \lambda\, x.\lambda\, y.M$. Given that abstractions are first-class values, in the sense that they can be returned as values and passed as parameters, this is a fairly intuitive encoding. It has some pleasant knock-on effects. For example, when there is an arity shortfall, i.e. the number of actual parameters is less than the number of formal parameters,

then it is both natural and useful simply to return an abstraction. Thus, $(\lambda xy.fxy)u$ can be evaluated to return $(\lambda y.fuy)$. This is an extremely convenient mechanism to support partial evaluation.

Type parametrization and quotation One key aspect of this representation is that we acknowledge that the abstract syntax is strangely silent on what the *terminals* are. It doesn't actually say what x's are. Often implementations of the λ-calculus will make some choice, such as Strings or Integers or some other representation. With `Scala`'s type parametrization we can defer this choice. In fact, to foreshadow some of what's to come, we illustrate that we never actually have to go outside of the basic grammar definition to come up with a supply of identifiers.

In the code above we have deferred the choice of identifier. In the code below we provide several different kinds of identifiers (the term of art in this context is "name"), but defer the notion of an expression by the same trick used to defer the choice of identifiers.

```
trait Nominals {
  type Term
  abstract class Name
  case class Transcription( expression : Term )
     extends Name
  case class StringLiteral( str : String )
     extends Name
  case class DeBruijn( outerIndex : Int, innerIndex : Int )
     extends Name
  case class URLLiteral( url : java.net.URL )
     extends Name
}
```

Now we wire the two types together.

```
trait ReflectiveGenerators
extends Expressions with Nominals {
  type Nominal = Name
  type Term = Expression
}
```

This allows us to use *quoted* terms as variables in *lambda*-terms! The idea is very rich as it begs the question of whether such variables can be *unquoted* and what that means for evaluation. Thus, `Scala`'s type system is already leading to some pretty interesting places! In fact, this is an instance of a much deeper design

principle lurking here, called two-level type decomposition, that is enabled by type-level parametricity. We'll talk more about this in upcoming chapters, but just want to put it on the backlog.

Some syntactic sugar To this core let us add some syntactic sugar.

PREVIOUS	LET	SEQ
$M, N ::= ...$	$\mid let\ x = M\ in\ N$	$\mid M; N$

This is sugar because we can reduce $let\ x\ =\ M\ in\ N$ to $(\lambda x.N)M$ and $M; N$ to $let\ x\ =\ M\ in\ N$ with x not occurring in N.

Digression: complexity management principle In terms of our implementation, the existence of this reduction means that we can choose to have explicit representation of these syntactic categories or not. This choice is one of a those design situations that's of significant interest if our concern is complexity management. [Note: brief discussion of the relevance of super combinators.]

Concrete syntax Now let's wrap this up in concrete syntax.

EXPRESSION	MENTION	ABSTRACTION	APPLICATION
$M, N ::=$	x	$\mid$ `(`x_1`,...,`x_k`) =>` M	$\mid M$`(`N_1`,...,`N_k`)`

LET	SEQ	GROUP
$\mid$ `val` x `=` M`;`N	$\mid M$`;`N	$\mid$ `{` M `}`

It doesn't take much squinting to see that this looks a lot like a subset of `Scala`, and that's because – of course! – functional languages like `Scala` all share a common core that is essentially the λ-calculus. Once you familiarize yourself with the λ-calculus as a kind of design pattern you'll see it poking out everywhere: in `Clojure` and `OCaml` and `F#` and `Scala`. In fact, as we'll see later, just about any DSL you design that needs a notion of variables could do worse than simply to crib from this existing and well understood design pattern.

If you've been following along so far, however, you will spot that something is actually wrong with this grammar. We still don't have an actual terminal! *Concrete* syntax is what "users" type, so as soon as we get to concrete syntax we can no longer defer our choices about identifiers. Let's leave open the door for both ordinary

identifiers – such as we see in `Scala` – and our funny quoted terms. This means we need to add the following productions to our grammar.

IDENTIFIER	STRING-ID	QUOTATION
$x, y ::=$	$\mid String$	$\mid$ `@<`M`>`

(The reason we use the `@` for quotation – as will become clear later – is that when we have both quote and dequote, the former functions a lot like asking for a *pointer* to a term while the latter is a lot like dereferencing the pointer.)

Translating concrete syntax to abstract syntax The translation from the concrete syntax to the abstract syntax is compactly expressed as follows. Even if the form of the translation is unfamiliar, it should still leave you with the impression that some core of `Scala` is really the λ-calculus.

$$
\begin{array}{l}
[\![\ \mathrm{x}\ ]\!] = x \\
[\![\ (\mathrm{x})\ \Rightarrow\ \mathrm{expr}\ ]\!] = \lambda x.[\![\ \mathrm{expr}\ ]\!] \\
[\![\ \mathrm{expr}(\ \mathrm{expr}_1\ ,\ \ldots\ ,\ \mathrm{expr}_n\ )\ ]\!] \\
\qquad = [\![\ \mathrm{expr}\ ]\!]\ [\![\ \mathrm{expr}_1\ ]\!]\ \ldots\ [\![\ \mathrm{expr}_n\ ]\!] \\
[\![\ \mathbf{val}\ \mathrm{x} = \mathrm{expr}_1\ ;\ \mathrm{expr}_2\ ]\!] = \mathit{let}\ [\![\ \mathrm{x}\ ]\!] = [\![\ \mathrm{expr}_1\ ]\!]\ \mathit{in}\ [\![\ \mathrm{expr}_2\ ]\!] \\
[\![\ \mathrm{expr}_1\ ;\ \mathrm{expr}_2\ ]\!] = [\![\ \mathrm{expr}_1\ ]\!]\ ;\ [\![\ \mathrm{expr}_2\ ]\!] \\
[\![\ \{\ \mathrm{expr}\ \}\ ]\!] = (\ [\![\ \mathrm{expr}\ ]\!]\ )
\end{array}
$$

Listing 5.1: translating concrete to abstract syntax

Further, the value of the explicit representation of sugar in terms of structuring the translation should be clear. Of course, in a book entitled *Pro Scala* the best way to unpack this presentation is in terms of a `Scala` implementation.

```
trait Compiler extends Expressions with Nominals {
  // Abstract away interning variables
  type Internist =
  {def intern( varExpr : Absyn.VariableExpr ) : Nominal}
  def internist() : Internist

  def intern( varExpr : Absyn.VariableExpr )
  : Nominal = { internist().intern( varExpr ) }
  def compileExpr( numericExpr : Absyn.Numeric )
  : Expression = {
    new IntegerExpression(
        numericExpr.integer_.asInstanceOf[Int]
    )
```

```
}

// ⟦ x ⟧ = x
def compileExpr( mentionExpr : Absyn.Mention )
: Expression = {
  new Mention( intern( mentionExpr.variableexpr_ ) )
}
// ⟦ (x) ⇒ expr ⟧ = λx.⟦ expr ⟧
def compileExpr( abstractionExpr : Absyn.Abstraction )
: Expression = {
  val fmls : List[Nominal] =
  abstractionExpr.listvariableexpr_.map(
  { ( vExpr : Absyn.VariableExpr ) ⇒ intern( vExpr ) }
  ).toList
  new Abstraction( fmls, compile( abstractionExpr.expression_ ) )
}
// ⟦ expr( expr_1, ..., expr_n ) ⟧
// = ⟦ expr ⟧ ⟦ expr_1 ⟧ ... ⟦ exprn ⟧
def compileExpr( applicationExpr : Absyn.Application )
: Expression = {
  new Application(
     compile( applicationExpr.expression_1 ),
     List( compile( applicationExpr.expression_2 ) )
  )
}

// ⟦ − ⟧ : Mini−Scala ⇒ λ−calculus
def compile( expr : Absyn.Expression )
: Expression = {
  expr match {
    case value : Absyn.Value ⇒ {
      value.valueexpr_ match {
        case numericExpr : Absyn.Numeric ⇒
        compileExpr( numericExpr )
      }
    }
    case numericExpr : Absyn.Numeric ⇒ {
      compileExpr( numericExpr )
    }
    case mentionExpr : Absyn.Mention ⇒ {
      compileExpr( mentionExpr )
```

```
        }
        case abstractionExpr : Absyn.Abstraction => {
          compileExpr( abstractionExpr )
        }
        case applicationExpr : Absyn.Application => {
          compileExpr( applicationExpr )
        }
      }
    }

    def parse( str : String ) : Absyn.Expression = {
      (new parser(
           new Yylex( new StringReader( str ) )
           ) ).pExpression()
    }

    def compile( str : String ) : Expression = {
      try {
        compile( parse( str ) )
      }
      catch {
        case e => { // log error
          throw e
        }
      }
    }
  }
```

The first thing to notice about this translation is how faithfully it follows the equational specification. This aspect of functional programming in general and `Scala` in particular is one of the things that sets it apart. In a development culture where AGILE methodologies rightfully demand a justification thread running from feature to line of code, a means of tracing specification to implementation is of practical importance. Of course, rarely do today's SCRUM meetings result in equational specifications; however, they might result in diagrammatic specification which, as we will see in subsequent sections, can be given equational interpretations that then guide functional implementation. Of equal importance: it cannot have escaped notice how much more compact the notations we have used for specification actually are. In a context where brevity and complexity management are paramount, tools – such as these specification techniques – that help us gain a higher vantage point ought to carry some weight. This is another aim of this book, to provide at least

some exposure to these higher-level techniques. One of the central points to be made is that if she's not already using them, the *pro* `Scala` programmer is primed and ready to take advantage of them.

Structural equivalence and Relations or What makes abstract syntax abstract Apart from the fact that concrete syntax forces commitment to explicit representation of terminals, you might be wondering if there are any other differences between concrete and abstract syntax. It turns out there are. One of the key properties of abstract syntax is that it encodes a notion of equality of terms that is not generally represented in concrete syntax.

It's easier to illustrate the idea in terms of our example. We know that programs that differ only by a change of bound variable are essentially the same. Concretely, the program (x) =>x + 5 is essentially the same as the program (y) =>y + 5. By "essentially the same" we mean that in every evaluation context where we might put the former if we substitute the latter we will get the same answer.

However, this sort of equivalence doesn't have to be all intertwined with evaluation to be expressed. A little forethought shows we can achieve some separation of concerns by separating out certain kinds of *structural* equivalences. Abstract syntax is where we express structural equivalence (often written using $\equiv$, for example $M \equiv N$). In terms of our example we can actually calculate when two λ-terms differ only by a change of *bound* variable, where by bound variable we just mean a variable mention in a term also using the variable as formal parameter of an abstraction.

Since we'll need that notion to express this structural equivalence, let's write some code to clarify the idea, but because it will be more convenient, let's calculate the variables not occurring bound, i.e. the *free* variables of a λ-term.

```
def freeVariables( term : Expression ) : Set[Nominal] = {
  term match {
    case Mention( reference ) => Set( reference )
    case Abstraction( formals, body ) =>
       freeVariables( body ) &~ formals.toSet
    case Application( operation, actuals ) =>
       ( freeVariables( operation ) /: actuals )(
       { ( acc, elem ) => acc ++ freeVariables( elem ) }
       )
  }
}
```

In addition to this idea we'll need to represent exchanging bound variables.

A traditional way to approach this is in terms of substituting a term (including variables) for a variable. A crucial point is that we need to avoid unwanted variable capture. For example, suppose we need to substitute y for x in a term of the form $\lambda y.(xy)$. Doing so blindly will result in a term of the form $\lambda y.(yy)$; but, now the first y is bound by the abstraction – probably not what we want. To avoid this – using structural equivalence! – we can move the bound variable "out of the way". That is, we can first change the term to an equivalent one, say $\lambda u.(xu)$. Now, we can make the substitution, resulting in $\lambda u.(yu)$. This is what's called capture-avoiding substitution. Central to this trick is the ability to come up with a fresh variable, one that doesn't occur in a term. Obviously, any implementation of this trick is going to depend implicitly on the internal structure of names. Until we have such a structure in hand we have to defer the implementation, but we mark it with a placeholder.

```
def fresh( terms : List[Expression] ) : Nominal
```

Now we can write in `Scala` our definition of substitution.

```
  def substitute(
  term : Expression,
  actuals : List[Expression], formals : List[Nominal]
) : Expression = {
  term match {
    case Mention( ref ) => {
      formals.indexOf( ref ) match {
        case -1 => term
        case i => actuals( i )
      }
    }
    case Abstraction( fmls, body ) => {
      val fmlsN = fmls.map(
        {
          ( fml ) => {
            formals.indexOf( fml ) match {
              case -1 => fml
              case i => fresh( List( body ) )
            }
          }
        }
      )
      val bodyN =
        substitute(
          body,
          fmlsN.map( _ => Mention( _ ) ),
          fmlsN
        )
      Abstraction(
        fmlsN,
        substitute( bodyN, actuals, formals )
      )
    }
    case Application( op, actls ) => {
      Application(
        substitute( op, actuals, formals ),
        actls.map( _ => substitute( _, actuals, formals ) )
      )
    }
  }
}
```

With this code in hand we have what we need to express the structural equivalence of terms.

```
def `=a=`(
      term1 : Expression, term2 : Expression
  ) : Boolean = {
  ( term1, term2 ) match {
      case (
        Mention( ref1 ),
        Mention( ref2 )
      ) => {
        ref1 == ref2
      }
      case (
        Abstraction( fmls1, body1 ), Abstraction( fmls2, body2 )
      ) => {
        if ( fmls1.length == fmls2.length ) {
          val freshFmls =
            fmls1.map(
              { ( fml ) => Mention( fresh( List( body1, body2 ) ) ) }
            )
          `=a=`(
            substitute( body1, freshFmls, fmls1 ),
            substitute( body2, freshFmls, fmls2 )
          )
        }
        else false
      }
      case (
        Application( op1, actls1 ),
        Application( op2, actls2 )
      ) => {
        ( `=a=`( op1, op2 ) /: actls1.zip actls2 )(
          { ( acc, actlPair ) =>
            acc && `=a=`( actlPair._1, actlPair._2 )
          }
        )
      }
  }
}
```

This is actually some significant piece of machinery just to be able to calculate

what we mean when we write $M[N/x]$ and $M \equiv N$. People have wondered if this sort of machinery could be reasonably factored so that it could be mixed into a variety of variable-binding capabilities. It turns out that this is possible and is at the root of a whole family of language design proposals that began with Jamie Gabbay's `FreshML`.

Beyond this separation of concerns the introduction of abstract syntax affords another kind of functionality. While we will look at this in much more detail in subsequent chapters, and especially the final chapter of the book, it is worthwhile setting up the discussion at the outset. A computationally effective notion of structural equivalence enables programmatic investigation of structure. In the context of our story, users not only write programs, but store them and expect to retrieve them later for further editing. In such a system it is easy to imagine they might want to search for structurally equivalent programs. In looking for patterns in their own code they might want to abstract it is easy to imagine them searching for programs structurally equivalent to one they've found themselves writing for the umpteenth time. Further, structural equivalence is one of the pillars of a system that supports automated refactoring.

Digression: the internal structure of the type of variables At this point the attentive reader may be bothered by something going on in the calculation of freeVariables. To actually perform the remove or the union we have to have equality defined on variables. Now, this works fine for Strings, but what about Quotations?

The question reveals something quite startling about the types[2] of variables. Clearly, the type has to include a definition of equality. Now, if we want to have an inexhaustible supply of variables, then the definition of equality of variables must make use of the "internal" structure of the variables. For example, checking equality of Strings means checking the equality of the respective sequences of characters. There are a finite set of characters out of which all Strings are built and so eventually the calculation of equality grounds out. The same would be true if we used Integers as "variables". If our type of variables didn't have some evident internal structure (like a String has characters or an Integer has arithmetic structure) and yet it was to provide an endless supply of variables, then the equality check could only be an *infinite* table – which wouldn't fit inside a computer. So, the type of variables must also support some internal structure, i.e. it must be a container type!

Fortunately, our Quotations are containers, by definition. However, they face another challenge: are the Quotations of two structurally equivalent terms equal as variables? If they are then there is a mutual recursion! Equivalence of terms

[2]Note that here we mean the type of the entity in the model that represents variables – not a typing for variables in the language we're modeling.

depends on equality of Quotations which depends on equivalence of terms. It turns out that we have cleverly arranged things so that this recursion will bottom out. The key property of the structure of the abstract syntax that makes this work is that there is an alternation: quotation, term constructor, quotation, ... Each recursive call will lower the number of quotes, until we reach 0.

Evaluation – aka operational semantics The computational engine of the λ-calculus is an operation called β-reduction. It's a very simple operation in principle. Whenever an *abstraction* occurs in the *operation* position of an *application* the application as a whole reduces to the *body* of the abstraction in which every occurrence of the formal parameter(s) of the abstraction has been replaced with the actual parameter(s) of the application. As an example, in the application $(\lambda u.(uu))(\lambda x.x)$, $\lambda u.(uu)$ is in operation position. Replacing in the body of the abstraction each occurrence of the formal parameter, u, with the actual parameter, $\lambda x.x$, of the application reduces the application to $(\lambda x.x)(\lambda x.x)$ which when we repeat the action of β-reduction reduces in turn to $\lambda x.x$.

In symbols we write beta reduction like this

$$\underset{\beta\text{-REDUCTION}}{(\lambda x.M)N \rightarrow M[N/x]}$$

In terms of our concrete syntax this means that we can expect expressions of the form $((\mathrm{x}_1, ..., \mathrm{x}_n) =>\mathrm{e})\mathrm{e}_1 \ ... \ \mathrm{e}_n$ to evaluate to $\mathrm{e}[\mathrm{e}_1/\mathrm{x}_1 \ ... \ \mathrm{e}_n/\mathrm{x}_n]$.

It is perhaps this last expression that brings home a point: we need to manage variable bindings, called environments in this discussion. The lambda calculus is silent on *how* this is done. There are a variety of strategies for implementing environments.

Ordinary maps

DeBruijn notation

The `Scala` implementation

Here's the code

```
trait Values {
  type Environment
```

```
    type Expression
    abstract class Value
    case class Closure(
       fn : List[Value] => Value
    ) extends Value
    case class Quantity( quantity : Int )
      extends Value
}

trait Reduction extends Expressions with Values {
  type Dereferencer = {def apply( m : Mention ) : Value }
  type Expansionist =
  {def extend(
    fmls : List[Mention],
    actls : List[Value]
    ) : Dereferencer}
  type Environment <: (Dereferencer with Expansionist)
  type Applicator = Expression => List[Value] => Value

  val initialApplicator : Applicator =
  { ( xpr : Expression ) => {
      ( actls : List[Value] ) => {
        xpr match {
          case IntegerExpression( i ) => Quantity( i )
          case _ => throw new Exception( "why are we here?" )
        }
      }
    }
  }

  def reduce(
     applicator : Applicator,
     environment : Environment
  ) : Expression => Value = {
    case IntegerExpression( i ) => Quantity( i )
    case Mention( v ) => environment( Mention( v ) )
    case Abstraction( fmls, body ) =>
    Closure(
       { ( actuals : List[Value] ) => {
           val keys : List[Mention] =
           fmls.map( { ( fml : Nominal ) => Mention( fml ) });
```

```
            reduce(
                applicator,
                environment.extend(
                    keys,
                    actuals ).asInstanceOf[Environment]
            )( body )
          }
        }
      )
      case Application(
          operator : Expression,
          actuals : List[Expression]
      ) => {
          reduce( applicator, environment )( operator ) match {
            case Closure( fn ) => {
              fn.apply(
                (actuals
                 map
                 {( actual : Expression) =>
                   (reduce( applicator, environment ))( actual )})
              )
            }
            case _ =>
               throw new Exception( "attempt to apply non function" )
          }
       }
    case _ => throw new Exception( "not implemented, yet" )
  }
}
```

What goes into a language definition

Before moving to the next chapter it is important to digest what we've done here. Since we've called out DSL-based design as a methodology worthy of attention, what does our little foray into defining a language tell us about language definition? It turns out that this is really part of folk lore in the programming language semantics community. At this point in time one of the commonly accepted presentations of a language definition has three components:

- syntax – usually given in terms of some variant of BNF

- structural equivalence – usually given in terms of a set of relations
- operational semantics – usually given as a set of conditional rewrite rules, such as might be expressed in SOS format.

That's exactly what we see here. Our toy language can be completely characterized by the following aforementioned half-page specification.

Syntax

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

Structural equivalence

$$\alpha\text{-EQUIVALENCE}\ \frac{y \notin \mathcal{FN}(M)}{\lambda x.M = \lambda y.M[y/x]}$$

where

$$\mathcal{FN}(x) = x \qquad \mathcal{FN}(\lambda x.M) = \mathcal{FN}(M) \backslash \{x\} \qquad \mathcal{FN}(MN) = \mathcal{FN}(M) \cup \mathcal{FN}(N)$$

and we write $M[y/x]$ to mean substitute a y for every occurrence of x in M.

Operational semantics

$$\beta\text{-REDUCTION}\ (\lambda x.M)N \to M[N/y] \qquad \text{STRUCT}\ \frac{M \equiv M', M' \to N', N' \equiv N}{M \to N}$$

Discussion This specification leaves open some questions regarding order or evaluation. In this sense it's a kind of proto-specification. For example, to get a left-most evaluation order you could add the rule

$$\text{LEFTMOST}\ \frac{M \to M'}{MN \to M'N}$$

The `Scala` code we wrote in the preceding sections is essentially an elaboration of this specification. While this notation is clearly more compact, it is not hard to recognize that the code follows this structure very closely.

5.3 The project model

Recalling chapter one, the aim is not just to implement the λ-calculus, but to implement an editor and project management capability. One of the key points of this book is that DSL-based design really does work. So, the basic methods for specifying and implementing the toy language *also* apply to specifying and implementing these features of our application as well.

5.3.1 Abstract syntax

5.3.2 Concrete syntax – and presentation layer

5.3.3 Domain model

5.4 A transform pipeline

TBD

Chapter 6

Zippers and contexts and URI's, oh my!

Zippers are not just for Bruno anymore

6.1 Zippers are not just for Bruno anymore

6.1.1 The history of the zipper

The origin of the zipper rests in the desire to provide an efficient functional representation of a "structure" editor. For example, we might consider navigation and destructive modification of a tree. In a functional representation destructive operations need to be replaced by copying. Done naively, this can be very expensive.

Huet's zipper

In his functional pearl Huet describes a generic approach to the problem of an applicative structure editor. He dubbed it the zipper. The key idea is to denote the location of a position in a tree by splitting it into two pieces: the subtree of focus and the context in which it appears.

To render this idea in `Scala` suppose that we have modeled the type of a tree as

```
trait Tree[A]
// Leaf
class TreeItem[A]( val item : A ) extends Tree[A]
```

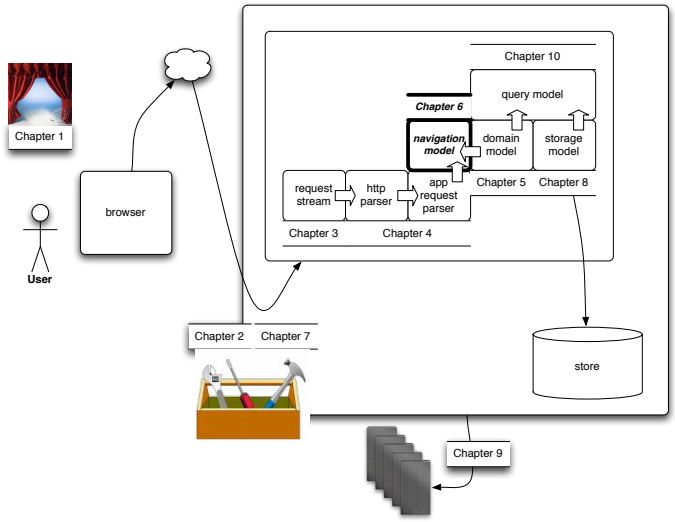


Figure 6.1: Chapter map

```
// Branches
class TreeSection[A](
  val section: List[Tree[A]]
) extends Tree[A]
```

with corresponding companion objects for easy construction and deconstruction. (We'd make these **case class**es, but then we couldn't use inheritance.)

```
object TreeItem {
  def apply[A]( item : A ) = { new TreeItem( item ) }
  def unapply[A]( tree : TreeItem[A] )
  : Option[( A )] = {
    Some( ( tree.item ) )
  }
}
object TreeSection {
  def apply[A]( section : List[Tree[A]] ) = {
    new TreeSection( section )
  }
  def unapply[A]( tree : TreeSection[A] )
  : Option[( List[Tree[A]] )] = {
    Some( ( tree.section ) )
  }
}
```

then we would model a context in the tree as

```
trait Context[A]
case class Top[A]( ) extends Context[A]
class TreeContext[A](
  val left : List[Tree[A]],
  val ctxt : Context[A],
  val right : List[Tree[A]]
) extends Context[A]
```

Essentially, a Context denotes a place where we might "plugin" a subtree. Thus, it identifies the branches to the left, the branches to the right and a "path" to a "hole".

Of course, we have the obligatory companion object.

```
object TreeContext {
  def apply[A](
    left : List[Tree[A]],
    ctxt : Context[A],
    right : List[Tree[A]] ) = {
      new TreeContext( left, ctxt, right )
    }
    def unapply[A]( ctxt : TreeContext[A] )
    : Option[( List[Tree[A]], Context[A], List[Tree[A]] )] = {
      Some( ( ctxt.left, ctxt.ctxt, ctxt.right ) )
    }
}
```

Since it is clear how this boilerplate is made, we will dispense with it in subsequent discussion; but note that the cost in boilerplate may not have been worth deprecating inheritance in **case class**es.

Now, we have the types necessary to model our intuitions as to what a location is. It's a pair of a context and a tree that plugs into the context. Note that neither of these datum are suffient in an of themselves to identify a location in a tree. The subtree could occur in any number of trees. Likewise, the context could be filled with any number of subtrees. It takes the pair to identify a location in a tree. For those with some experience in mathematics, this idea is strongly reminiscent of both Dedekind cuts and Conway's models of games as numbers.

```
class Location[A](
  val tree : Tree[A],
  val ctxt : Context[A]
)
```

As a paradigmatic example consider (a crude model of) the syntax tree of an arithmetic expression. (Now, the decision to model a tree as a **class** becomes clear.)

```
case class Token[A](
 override item : A
) extends TreeItem[A]( item )
case class AST[A](
  override section : List[Tree[A]]
) extends TreeSection[A]( section )
```

Then an instance might look like

```
AST[String](
  List(
    AST[String](
      List(
        Token[String]( "a" ),
        Token[String]( "*" ),
        Token[String]( "b" )
      )
    ),
    Token[String]( "+" ),
    AST[String](
      List(
        Token[String]( "c" ),
        Token[String]( "*" ),
        Token[String]( "d" )
      )
    )
  )
)
```

Then the location of the second multiplication sign is:

```
Location[String](
  Token[String]( "*" ),
  TreeContext[String](
    List( Token[String]( "c" ) ),
    TreeContext[String](
      List(
        Token[String]( "+" ),
        AST[String](
          List(
            Token[String]( "a" ),
```

```
            Token[String]( "*" ),
            Token[String]( "b" )
            )
          )
        ),
        Top( ),
        List( )
        ),
      List( Token[String]( "d" ) )
    )
  )
```

The navigation functions With this structure we can define generic navigation functions.

```
trait ZipperNavigation[A] {
  def left( location : Location[A] ) : Location[A] = {
    location match {
      case Location( _, Top ) => {
        throw new Exception( "left of top" )
      }
      case Location( t, TreeContext( l :: left, up, right ) ) => {
        Location( l, TreeContext( left, up, t :: right ) )
      }
      case Location( t, TreeContext( Nil, up, right ) ) => {
        throw new Exception( "left of first" )
      }
    }
  }
  def right( location : Location[A] ) : Location[A] = {
    location match {
      case Location( _, Top ) => {
        throw new Exception( "right of top" )
      }
      case Location( t, TreeContext( left, up, r :: right ) ) => {
        Location( r, TreeContext( t :: left, up, right ) )
      }
      case Location( t, _ ) => {
        throw new Exception( "right of last" )
      }
    }
```

```
  }
  def up( location : Location[A] ) : Location[A] = {
    location match {
      case Location( _, Top ) => {
        throw new Exception( "up of top" )
      }
      case Location( t, TreeContext( left, up, right ) ) => {
        Location( TreeSection[A]( left.reverse ::: ( t :: right ) ),
                  up )
      }
    }
  }
  def down( location : Location[A] ) : Location[A] = {
    location match {
      case Location( TreeItem( _ ), _ ) => {
        throw new Exception( "down of item" )
      }
      case Location( TreeSection( u :: trees ), ctxt ) => {
        Location( u, Context( Nil, ctxt, trees ) )
      }
    }
  }
}
```

Exercising the zipper We can exercise the zipper navigation functions using the two examples from above.

```
  object Exercise extends ZipperNavigation[String] {
    val arithmeticExpr1 = ...

    val locationOf2ndMult = ...

     def show( depth : Int )( tree : Tree[String] ) : Unit = {
       tree match {
         case TreeItem( item : String ) => {
           val indent =
           ( "" /: (1 to depth) )( { ( acc, d ) => acc + " " } )
           println( indent + "Leaf : " + item )
         }
         case TreeSection( section : List[Tree[String]] ) => {
           for( t <- section ){ show( depth + 2 )( t ) }
```

```
        }
      }
    }
  }

scala> import Exercise._
import Exercise._
import Exercise._

scala> show( 0 )( arithmeticExpr1 )
show( 0 )( arithmeticExpr1 )
  Leaf : a
  Leaf : *
  Leaf : b
Leaf : +
  Leaf : c
  Leaf : *
  Leaf : d

scala> show( 0 )( locationOf2ndMult.tree )
show( 0 )( locationOf2ndMult.tree )
Leaf : *

scala> show( 0 )( up( locationOf2ndMult ).tree )
show( 0 )( up( locationOf2ndMult ).tree )
Leaf : c
Leaf : *
Leaf : d

scala> show( 0 )( up( up( locationOf2ndMult ) ).tree )
show( 0 )( up( up( locationOf2ndMult ) ).tree )
  Leaf : a
  Leaf : *
  Leaf : b
Leaf : +
  Leaf : c
  Leaf : *
  Leaf : d

scala> show( 0 )( up( up( up( locationOf2ndMult ) ) ).tree )
show( 0 )( up( up( up( locationOf2ndMult ) ) ).tree )
```

```
java.lang.Exception: up of top
        ...
scala>
```

Of course, the real desiderata are the mutation functions.

```
trait ZipperMutation[A] {
  def update(
  location : Location[A], tree : Tree[A]
  ) : Location[A] = {
    location match {
      case Location( _, ctxt ) =>
        Location( tree, ctxt )
      }
    }
  def insertRight(
    location : Location[A], tree : Tree[A]
  ) : Location[A] = {
    location match {
      case Location( _, Top( ) ) => {
        throw new Exception( "insert of top" )
      }
      case Location(
        curr, TreeContext( left, up, right )
      ) => {
        Location(
          curr, TreeContext( left, up, tree :: right )
        )
      }
    }
  }
  def insertLeft(
    location : Location[A], tree : Tree[A]
  ) : Location[A] = {
    location match {
      case Location( _, Top( ) ) => {
        throw new Exception( "insert of top" )
      }
      case Location(
        curr, TreeContext( left, up, right )
      ) => {
        Location(
```

```
          curr, TreeContext( tree :: left, up, right )
        )
      }
    }
  }
  def insertDown(
    location : Location[A], tree : Tree[A]
  ) : Location[A] = {
    location match {
      case Location( TreeItem( _ ), _ ) => {
        throw new Exception( "down of item" )
      }
      case Location(
        TreeSection( progeny ), ctxt
      ) => {
        Location(
          tree, TreeContext( Nil, ctxt, progeny )
        )
      }
    }
  }
  def delete(
    location : Location[A], tree : Tree[A]
  ) : Location[A] = {
    location match {
      case Location( _, Top( ) ) => {
        throw new Exception( "delete of top" )
      }
      case Location(
        _, TreeContext( left, up, r :: right )
      ) => {
        Location(
          r, TreeContext( left, up, right )
        )
      }
      case Location(
        _, TreeContext( l :: left, up, Nil )
      ) => {
        Location(
          l, TreeContext( left, up, Nil )
        )
```

```
        }
        case Location(
          _, TreeContext( Nil, up, Nil )
        ) => {
          Location( TreeSection( Nil ), up )
        }
      }
    }
  }
```

Zippers generically

Two kinds of genericity It turns out that Huet's discovery can be made to work on a much wider class of structures than "just" trees. Intuitively speaking, if their type arguments are "zippable", then virtually all of the common functional data type constructors, including sequencing constructors like product, and branching constructors, like summation or "casing", result in "zippable" types. That is, there are procedures for deriving a notion of zipper capable of traversing and mutating the structure. Essentially, there are two strategies to achieve this genericity: one is based on structural genericity and the other on procedural genericity.

Genericity of structure The former approach relies on being able to define a notion of context for any "reasonable" data structure. Not surprisingly, it turns out that we can give a good definition of "reasonable". What is surprising is that the resulting definition is amenable to an operation that perfectly mimics the notion of derivative from Newton's calculus. The operation is an operation on *types*. This allows us to give a type-level definition of the notion of location – just as we did with trees, but now for any type.

We can use `Scala`'s type notation to see where the new genericity has been added. The type of trees in the example is already polymorphic: Tree[A]. That's what having that type parameter A means. The navigation trait is therefore also parametric in A. The navigation trait, however, is hardcoded in the container type, Tree[A]. When we add this second level of genericity, the navigation trait will have to take a second, *higher-kinded* type parameter for the container because it will work on any container within a range of reasonably defined containers.

The use case we have been considering – navigating and mutating an in-memory representation of a tree – is then extended to navigating and mutating an in-memory representation of an arbitrary data structure. Moreover, the code is purely functional – with all of the attendant advantages of purely functional code

we have been observing since Chapter 1. Obviously, in the context of the web, this particular use case is of considerable interest. Nearly, every web application is of this form: navigating a tree or graph of pages. Usually, that graph of pages is somehow homomorphic, i.e. an image of, the graph of some underlying domain data structure, like the data structures of employee records in a payroll system, or the social graph of a social media application like Twitter. Many web applications, such as so-called content management systems, also support the mutation of the graph of pages. So, having a method of generating this functionality from the types of the underlying data domain, be they web pages, or some other domain data type, is clearly pertinent to the most focused of application developers.

And yet, the notion of a *derivative of data types* is irresistably intriguing. It's not simply that it has many other applications besides web navigation and update. That a calculational device that an Englishman discovered some 400+ years ago in his investigations for providing a mathematical framework for gravitation and other physical phenomena should be applicable to structuring computer programs is as surprising as it is elegant and that makes it *cool*.

Genericity of control The latter approach to generically constructing zippers is just as rich in terms of the world of ideas it opens up as it is in the imminent practicality of its immediate applications. The key insight is to abstract on control, rather than on form. Not surprisingly, then the central tool is the (delimited) continuation. To be clear, in this approach, originally developed by Oleg Kiselyov, navigation is reifed as a function and supplied as a parameter. In this sense, it is not automagically deriving mechanism for navigation, as does the structural approach. The semantics of mutation, on the other hand, is provided with a powerful generative mechanism. More specifically, a dial is provided for the visibility of mutation with respect to different threads of control. In other words, fine-grained constrol on the *transactional semantics* of mutating the data structure is provided. This is exceptionally powerful because, as we have mentioned since Chapter 1, the transactional semantics is one of the principal places on which performance of a system – especially a high-volume system – hinges; but, by being based on a form of monad, namely delimited continuations, the abstraction gets the compiler involved. This has the effect of enlisting the compiler in maintaining discipline and sanity on transaction semantics – which is vitally important when supplying a fine-grained control on something as performance-critical as the semantics and visibility of update.

6.2 Zipper and one-holed contexts

6.3 Differentiation and contexts

6.3.1 Regular types

6.3.2 Container types

6.4 Generic zipper – differentiating navigation

In this section we build a bridge between Oleg Kiselyov's Haskell implementation of the generic zipper. This is a transliteration of his original code. As such, we provide a veneer over `Scala`'s native delimited continuation library. Then we use this veneer to express a direct translation of Oleg's code.

```
object MonadDefns {
  type MonadLike = {
    def map[A,B]( f : A => B )
    def flatMap[M[_],A,B]( f : A => M[B] )
    def filter[A]( p : A => Boolean )
  }
  type MonadXFormLike = {
    def lift[ML[_],MU[_],A]( m : ML[A] ) : MU[A]
  }
}

trait StateT[S,M[_],A]{
  def runState( s : S ) : M[(A,S)]
  def evalState( s : S ) : M[A]
  def get : StateT[S,M,S]
  def put( s : S ) : StateT[S,M,Unit]

  def map[B]( f : A => B )
  def flatMap[B]( f : A => StateT[S,M,B] )
  def filter( p : A => Boolean )

  def lift( c : M[A] ) : StateT[S,M,A]
}

trait CC[R,M[_],A] {
```

```
    def k2P : K[R,M,A,_] => StateT[Int,M,A]
  }

  trait Prompt[R,A] {
    def level : Int
  }

  class CPrompt[R,A](
    override val level : Int
  ) extends Prompt[R,A] {
  }

  trait P[R,M[_],A] {
    self : StateT[Int,M,A] =>
    def stateT : StateT[Int,M,A]
    def runP() : M[(Int,A)]
    def newPrompt() = {
      for( n <- get ) yield{ put( n+1 ); new CPrompt( n ) }
    }
  }

  trait Frame[M[_],R,A,B]{
    def a2CC : A => CC[R,M,B]
  }

  trait K[R,M[_],A,B]{
    def frame : Frame[M,R,A,B]
    def r : R
    def a : A
    def b : B

    def map[C]( f : A => C )
    def flatMap[C]( f : A => K[R,M,A,C] )
    def filter( p : A => Boolean )

    def lift( m : M[A] ) : CC[R,M,A]
  }

  trait SubK[R,M[_],A,B] extends K[R,M,A,B]{
  }
```

```
  trait ControlOps[R,M[_],A] {
    def appk[B]( k : K[R,M,A,B], a : A ) : StateT[Int,M,A]
    def runCC( cc : CC[R,M,A] ) : M[A]
    def newPrompt( ) : CC[R,M,Prompt[R,A]]
    def pushPrompt(
      prompt : Prompt[R,A], cc : CC[R,M,A]
    ) : CC[R,M,A]
    def letSubK[B](
      prompt : Prompt[R,B],
      subk : SubK[R,M,A,B] => CC[R,M,B]
    ) : CC[R,M,A]
    def pushSubK[B](
      prompt : Prompt[R,B],
      subk : CC[R,M,A]
    ) : CC[R,M,B]
    def promptP( f : Prompt[R,A] => CC[R,M,A] ) : CC[R,M,A]
    def shiftP[B](
      p : Prompt[R,B],
      f : (CC[R,M,A] => CC[R,M,B]) => CC[R,M,B]
    ) : CC[R,M,A]
}
```

Essentially, a zipper in this new style wraps a term. It may also contain a traversal function.

```
  trait Zipper[R,M[_],T,D] {
    def term : T
  }

  class DCZipper[R,M[_],T,D](
    override val term : T,
    val traversal : CC[R,M,(Option[T],D)] => CC[R,M,Zipper[R,M,T,D]]
  ) extends Zipper[R,M,T,D]

  class ZipDone[R,M[_],T,D](
    override val term : T
  ) extends Zipper[R,M,T,D]
```

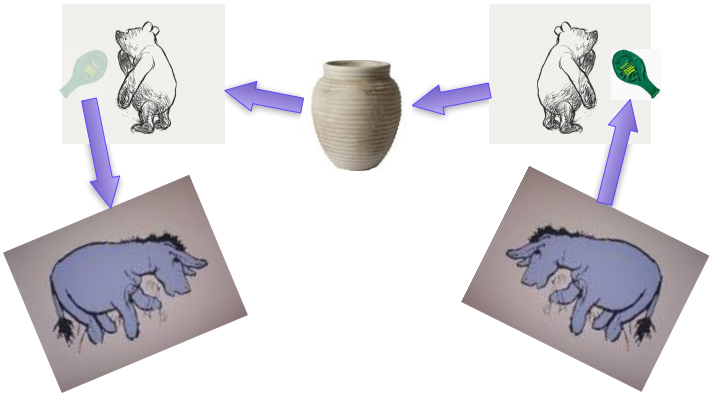


Figure 6.2: delimited continuations and synchronized exchange

We then provide basic factory mechanism for constructing zippers and then using them.

```
trait ZipperOps[R,M[_],T,D] {
  def zipTerm(
    traversal
    : ( ( T ⇒ CC[R,M,( Option[T], D )] ), T )
      ⇒ CC[R,M,T],
    term : T
  ) : CC[R,M, Zipper[R,M,T,D]]
  def zipThrough( zipper : Zipper[R,M,T,D] ) : Unit
}
```

6.4.1 Delimited continuations

In previous sections we have used the analogy of monads as maintaining a discipline for "putting things in a box"; similarly, comonads provide a discipline for "taking things out of the box". There is a connection between this and delimited continuations. To see the connection, we might imagine a picture like

One way to implement this is that the "daemon", Pooh, is really just the act of wrapping either client's access to the box in code that grabs the current continuation, call it kg (or kt, respectively), and then does the following.

- Giver side:
  - check to see if there is a matching taker, kt, (in a queue of taker requests packaged as continuations).

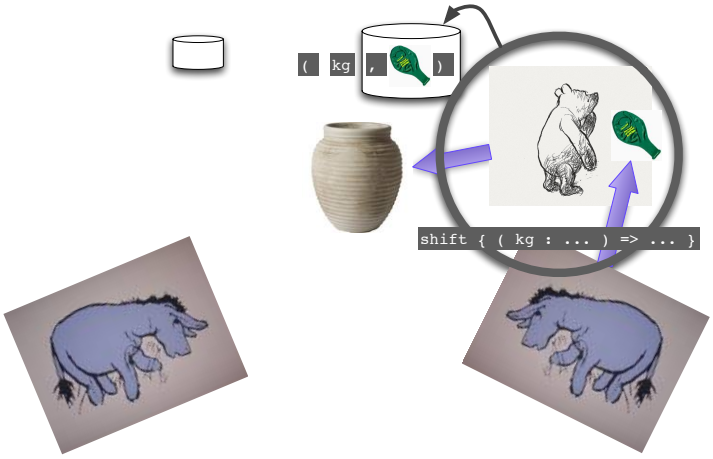


Figure 6.3: Giver's side

  - If there is, invoke (kt v) passing it the value, v, that came in on the giver's call, and invoke (kg unit), passing it unit.
  - Otherwise, queue (v,kg) in a giver's queue.

- Taker's side:
  - check to see if there is a matching giver, (v,kg), (in a queue of giver requests packages as continuations).
  - If there is, invoke (kt v), passing v to the taker's continuation, and (kg unit) passing unit to the giver's continuation.
  - Otherwise, queue kt in a taker's queue.

If these look strangely like the put and get operations of the State monad – they's because they are. They've been coordinated around a state cell that is "located" at a rendezvous point for a pair of coroutines to exchange data.

For the adventurous, it is possible to develop a further connection to Milner's π-calculus. Roughly speaking, this is the way to implement synchronous-IO-style in the π-calculus, spelling out a specific relationship between delimited continuations and π-calculus-style communication.

If you see a further connection between this pattern and tuple spaces, that's because it's the basic mechanism for implementing tuple spaces.

Summarizing, monads like IO that are forever sticky, are one-way monads. Like the roach motel, or Hotel California, things go in, but don't come out. Monads that are really containers are "intuitionistic". That is, you know that if you put

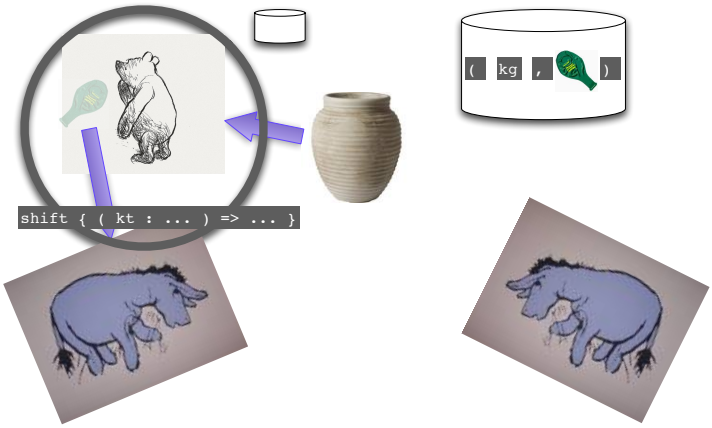


Figure 6.4: Taker's side

something in, you can get it out; but, if you receive a container, you don't know if it has anything in it until you open the lid. They have a relationship with a comonad that is "intuitionistically" disciplined. Finally, there are monad-comonad pairs that enjoy a linear discipline. This linear discipline matches every "gozinta" with a "gozouta" and vice versa. That discipline may be implemented by delimited continuations. This implementation strategy, by the way, also connects delimited continuations to the other generic zipper, discovered by Oleg.

6.5 Species of Structure

6.6 Constructing contexts and zippers from data types

The key intuition is that a zipper is a "focus" on a subterm of a term. The data needed to capture this idea is a pair, (T,∂), the subterm itself, and the context in which it occurs. Using types to guide our intuition we see that the subterm must have the same type as a term while the type of a context is determined by a calculation that perfectly matches a version of the derivative one might have learned in high school calculus – but applied to data structures.

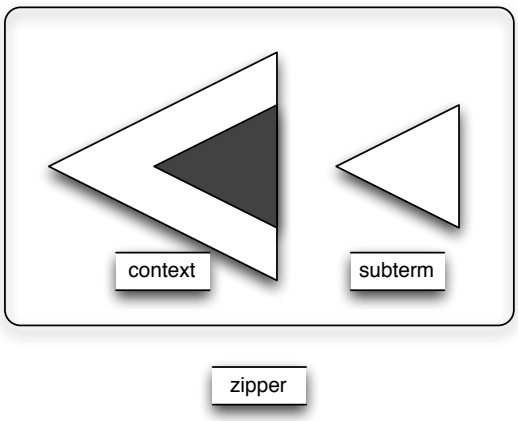


Figure 6.5: Context and subterm

6.6.1 Contexts

$$\partial Const_A = 0$$

$$\partial Id = 0$$

$$\partial F + G = \partial F + \partial G$$

$$\partial F \times G = F \times \partial G + \partial F \times G$$

$$\partial F \circ G = \partial F \circ G \times G$$

6.6.2 Zippers

```
case class Context[Name, NSeq <: NmSeq[Name]](
    override val self : RegularType[Name,NSeq]
)
extends RegularType[Name, NSeq] with Proxy {
  override def support = self.support
}

trait Contextual[Name, NSeq <: NmSeq[Name]]
extends Differential[Name,NSeq]  {
  def holePunch( support : NSeq )(
      x : Name, regularType : RegularType[Name,NSeq]
```

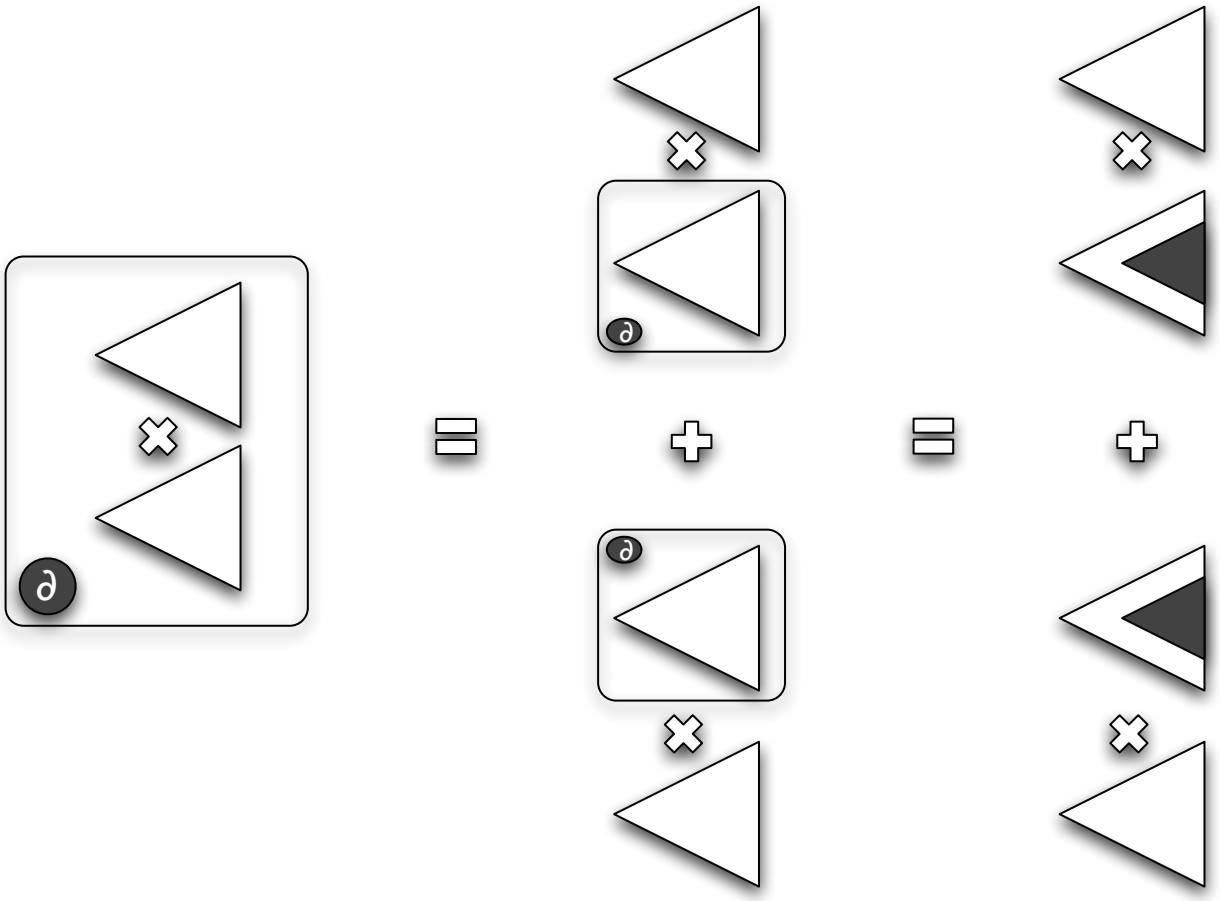


Figure 6.6: Context and subterm

```
) : Context[Name,NSeq] = {
  fresh match {
    case None ⇒ throw new Exception( "out␣of␣names" )
    case Some( cX ) ⇒ {
      val fixRT =
      RegularFixPt[Name,NSeq](
          (fresh match {
            case None ⇒
               throw new Exception( "out␣of␣names" )
            case Some( fX ) ⇒ fX
          }),
          regularType,
          support
      )
      Context[Name,NSeq](
          RegularFixPt[Name,NSeq](
             cX,
          RegularSum[Name,NSeq](
              List(
                  RegularUnity[Name,NSeq]( support ),
                  RegularProduct[Name,NSeq](
                      List(
                          RegularFPEnv[Name,NSeq](
                          x,
                          partial( x, regularType ),
                          fixRT,
                          support
                          ),
                          RegularMention[Name,NSeq](
                             cX,
                             support
                          )
                      ),
                      support
                  )
              ),
              support
          ),
          support
       )
    )
```

```
      }
    }
  }
}

  trait Differential[Name, NSeq <: NmSeq[Name]]
  extends NmSeqOps[Name,NSeq] {
    def regularNull( supp : NSeq ) : RegularNullity[Name,NSeq]
    def regularUnit( supp : NSeq ) : RegularUnity[Name,NSeq]
    def partial( x : Name, rtype : RegularType[Name, NSeq] )
    : RegularType[Name, NSeq] = {
      rtype match {
        case RegularMention( y, supp ) => {
          if ( x == y ) {
            regularUnit( supp )
          }
          else {
            regularNull( supp )
          }
        }
        case RegularNullity( supp ) => regularNull( supp )
        case RegularUnity( supp ) => regularNull( supp )
        case RegularSum( s, supp ) => {
          RegularSum(
              s.map(
                {
                  ( rt : RegularType[Name,NSeq]) => {
                    partial( x, rt )
                  }
                }
              ),
              supp
          )
        }
        case RegularProduct( s, supp ) => {
          val right = s.dropRight( 1 )
          RegularSum[Name,NSeq](
              List(
                 RegularProduct[Name,NSeq](
                    List(
                       partial( x, s( 0 ) ),
```

```
                    RegularProduct[Name,NSeq](
                        right,
                        supp
                    )
                ),
                supp
            ),
            RegularProduct[Name,NSeq](
                List(
                    s(0),
                    partial(
                        x,
                        RegularProduct[Name,NSeq](right, supp)
                    )
                ),
                supp
            )
        ),
        supp
    )
}
case RegularFixPt(v, e, supp) => {
  val z = fresh match {
    case None => throw new Exception("out of names")
    case Some(fn) => fn
  }
  RegularSum[Name,NSeq](
      List(
          RegularFixPt(
              z,
              partial(
                  x,
                  RegularWeakening(
                      z,
                      RegularFPEnv(v, e, rtype, supp),
                      supp
                  )
              ),
              supp
          ),
          RegularProduct(
```

```
                List(
                    partial(
                        v,
                        RegularFPEnv(
                            v,
                            e,
                            rtype,
                            supp
                        )
                    ),
                    RegularMention( z, supp )
                ),
                supp
            )
        ),
        supp
    )
}
case RegularFPEnv( v, e, s, supp ) => {
  RegularSum(
      List(
          RegularFPEnv(
              v,
              partial( x, e ),
              s,
              supp
          ),
          // BUGBUG -- lgm -- have i got the association correct
          RegularProduct(
              List(
                  RegularFPEnv(
                      v,
                      partial( v, e ),
                      s,
                      supp
                  ),
                  partial( x, s )
              ),
              supp
          )
      ),
```

```
            supp
          )
        }
        case RegularWeakening( v, e, supp ) => {
          if ( x == v ) {
            regularNull( supp )
          }
          else {
            RegularWeakening( v, partial( x, e ), supp )
          }
        }
      }
    }
  }
```

6.7 Mapping URIs to zipper-based paths and back

6.7.1 Path and context

6.7.2 Homomorphisms and obfuscation

6.8 Applying zippers to our project

6.8.1 Navigating and editing terms

6.8.2 Navigating and editing projects

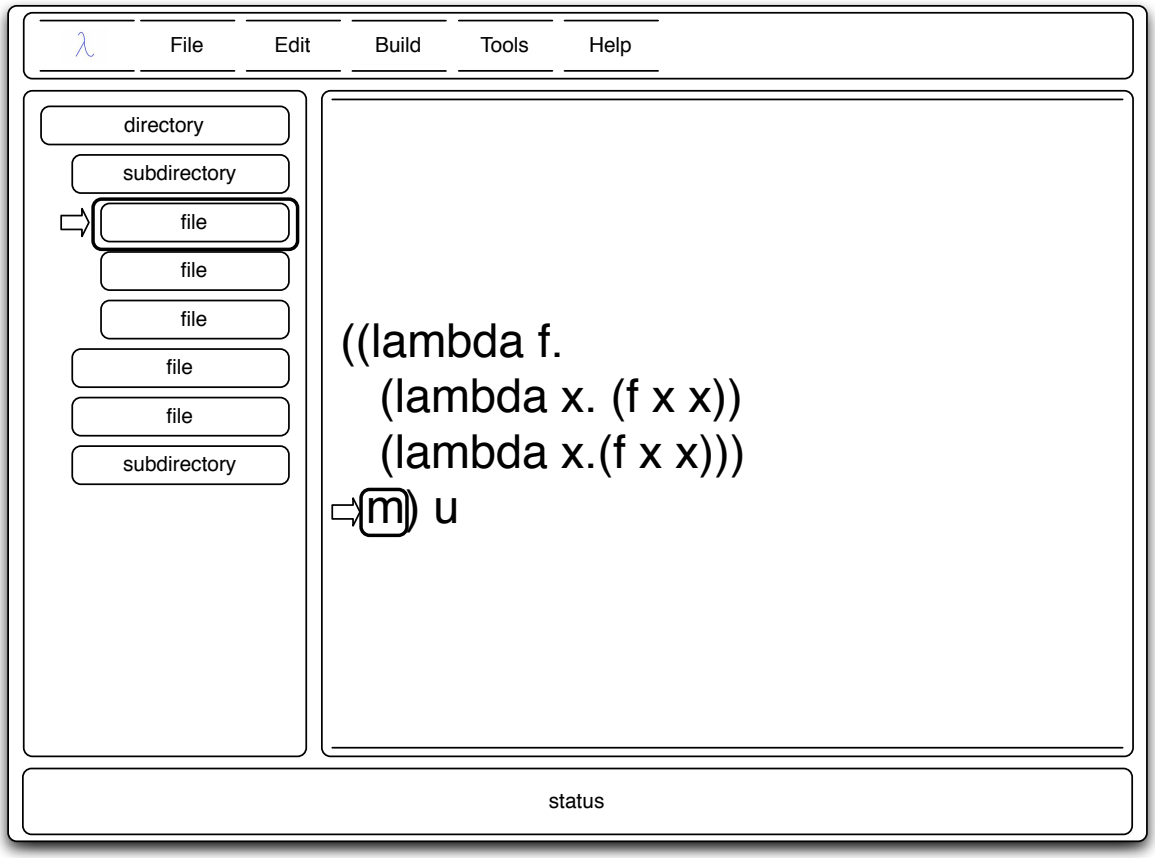


Figure 6.7: Zippers and editors

Chapter 7

A review of collections as monads

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

7.1 Sets, Lists and Languages

As we saw in chapter two, one role of monad is to provide the bridge between "flattenable" collections and the models of binary operators, investigating two paradigmatic kinds of collections and, more importantly, their interaction, exposes some of the necessary interior structure of a wide range of species of monad. It also prepares us for an investigation of the new `Scala` collections library. Hence, in this section we investigate, in detail, the Set and List monads as well as their combinations.

7.1.1 Witnessing Sets and Lists monadicity

Recalling our basic encapsulation of the core of the monad structure in `Scala`

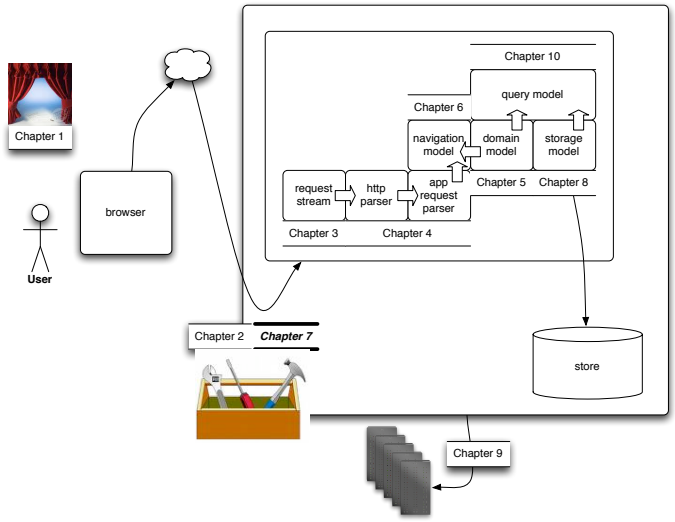


Figure 7.1: Chapter map

```
trait Monad[M[_]] {
    // map part of the functor M
    def map[A,B]( a2b : A => B ) : M[A] => M[B]
    // the unit natural transformation, unit: Identity => M[A]
    def unit[A]( a : A ) : M[A]
    // the mult natural transformation, mult: M[M[A]] => M[A]
    def mult[A]( mma : M[M[A]] ) : M[A]

    // derived
    def flatMap[A,B]( ma : M[A], a2mb : A => M[B] ) : M[B] = {
        mult( map( a2mb )( ma ) )
    }
}
```

We instantiate it for List by extending Monad[List] in order to provide canonical implementations of the operations map, unit and mult.

```
trait ListM extends Monad[List] {
   // map part of the List functor
   override def map[A,B]( a2b : A => B ) = {
       ( sa : List[A] ) => sa map a2b
   }
   // the unit natural transformation of the List monad
   override def unit[A]( a : A ) = List( a )
   // the mult natural transformation of the List monad
   override def mult[A]( mma : List[List[A]] ) =
   (( List( ) : List[A] ) /: mma )(
      { ( acc : List[A], elem : List[A] ) => acc ++ elem }
   )
}
```

The definition suggests we have named map well: our map means `Scala`'s map. This is a fairly general recipe: in a preponderance of cases lifting a function, say f : A =>B, to a function, M[f] : M[A] =>M[B], means calculating the function on each of the "elements" of M[A] and collecting the results in an M-like collection, namely M[B]. In the case above, M just happens to be Set.

In a similar manner, the recipe for the implementation of unit is ... well... paradigmatic. If the meaning of unit is the construction of a container embracing a single element, say a, then calling the constructor of the M collection feels like a natural choice. This is yet another view on the discussion in chapter 2 on monads as a kind of generic brace notation. If that was the syntactic view, this is the semantic view of the very same concept.

Finally, while there are several ways to implement mult we choose fold because the genericity of this implementation is a quick and dirty demonstration of the universality of fold. In some very real sense, all "flattening" of structure is representable as a fold.

To illustrate the genericity of these definitions, we compare them with a simple implementation of the Set monad. The implementations are nearly identical, begging the question of a DRYer expression of these instantiations, which we defer to a later section.

```
trait SetM extends Monad[Set] {
    // map part of the Set functor
    def map[A,B]( a2b : A => B ) = {
        ( sa : Set[A] ) => sa map a2b
    }
    // the unit natural transformation of the Set monad
    def unit[A]( a : A ) = Set( a )
    // the mult natural transformation of the Set monad
    def mult[A]( mma : Set[Set[A]] ) =
    (( Set( ) : Set[A] ) /: mma )(
       { ( acc : Set[A], elem : Set[A] ) => acc ++ elem }
    )
}
```

They illustrate another point that bears investigation. What distinguishes Sets from Lists is that the latter remembers both order and multiplicity. Not to put to fine a point on it, we expect that Set(1)++Set(1)==Set(1) while List(1)++List(1)==List(1, 1). In a similar manner, Set(1, 2) ==Set(2, 1) while List(1, 2)≠List(2, 1). As the code should make clear, when encountering these two species of collection in the wild, the notion of monad is indifferent to their distinguishing markings. It will assimilate either of them in exactly the same manner. At least as used in this particular way, monad is *not* where we encode order information. Likewise, it is not where we encode characteristics like the *idempotency* of operations like the idempotency of the ++ operation on Sets.

Recalling the summary of what goes into a language definition at the end of chapter six, notice that there are – at a minimum – two components: the grammar generating terms in the language and the relations saying when two terms in the language may be considered equal despite surface syntactic differences. The point of contact between monads and languages, as we will see in the upcoming sections, is that monads assimilate and encode the *grammar* part of a language *without* the relations part. A language that is pure grammar, with no additional identification of terms, i.e. no relation component, is called *free*. The above code is an encoding of the *proof* that List and Set stand in the same relation to some underlying "free" structure. That is, there is a language the terms of which stand in one-to-one correspondence with syntactic representations of Sets and Lists. The difference between the two structures lies "above" this underlying syntactic representation, in the relations component of a purely syntactic presentation of either data structure. That is why the monadic view of these data structures is identical.

It is also worth noting that while List records more information about order and multiplicity of the elements of a collection inhabiting the type, that corresponds

to *fewer* constraints on the operation ++. Inversely, Set records *less* information about order and multiplicity of the elements inhabiting the type; yet, this corresponds to *more* properties imposed on the operation ++. To wit, on the data type ++, the operation is required to be commutative, i.e. if s_1:Set[A] and s_2:Set[A], then $(s_1{+}{+}s_2){=}{=}(s_2{+}{+}s_1)$. Likewise, if s : Set[A], then (s++s)==s.

This is a general principle worth internalizing. When the *operations* associated with a collection acquire more structure, i.e. enjoy more *properties*, the collection remembers less information about the individual inhabitants of the type, precisely because the operation associated with "collecting" *identifies* more inhabitants of the type. In some sense the the assumption of properties drops a kind of veil down over individual structure. Contraposatively, "freedom" means that individual structure is the only carrier of information, or that all inhabitants of the type are "perfectly" individuated.

As seen below, the structure underlying the monadic view of List and Set is the data type we called a Monoid in chapter two. More specifically, it is the *free* monoid. It turns out that List is really just another syntax for the free monoid, while Set is a characterization of the smallest version of the monoid where the binary operation is commutative and idempotent. For those in the know, this means that Set is model of Boolean algebra. In terms of our discussion of DSLs, this means that there is an isomorphism between the DSL of Boolean algebra and the data type Set.

Why go to such lengths to expose truths that most programmers know in their bones, even if they don't know that they know them? We return to our aim: complexity management. What we have seen is that there is a deep simplicity, in fact one common structure, underlying these data types. Moreover, the notation of monad provides a specific framework for factoring this common structure in a way that both aligns with the principles of DSL-based design and with mathematical wisdom now vetted over 50 years. Looked at from another point of view, it provides justification for the intuitions guiding proposals for DSL-based design. Language-oriented design hooks into and makes available a wide range of tools that actually can simplify code and encourage reuse.

Moreover, like the language design view, the categorical view also provides a factorization of the free structure, aka the grammar, and the identities on terms, aka the relations. In categorical language the addition of identities takes place in what's called the Eilenberg-Moore algebras of the the monad. As we will see below, in a computatonal universe such as `Scala` this is just a four syllable name for the action of pairing the grammar with the relations. As we will see in the last chapter, on semantic search, holding a place for the relations widens the scope of the applicability of this technology. Specifically, it provides a unified framework for constraint-based programming, significantly expanding the scope of reach of `LINQ`-like technologies.

7.1.2 Languages and Sets of Words

Kleene star

I am not a number, I am a free monoid

```
type SetList[X] = Set[List[X]]
trait SetListM extends Monad[SetList] {
   // map part of the Set functor
   def map( a2b : A ⇒ B ) = {
       ( sa : Set[List[A]] ) ⇒ sa map a2b
   }
   // the unit natural transformation of the Set monad
   def unit( a : A ) = Set( List( a ) )
   // the mult natural transformation of the Set monad
   def mult( mma : Set[List[Set[List[A]]]] ) =
   (( Set( ) : Set[A] ) /: mma )(
      { ( acc : Set[List[A]], elem : Set[List[A]] ) ⇒ ... }
   )
}
```

7.1.3 Of lenses and bananas

7.2 Containers and syntax

7.2.1 The algebra of Sets

EXPRESSION	ADDITIVE IDENTITY	GENERATORS	COMPLEMENT	ADDITION
$m, n ::=$	T	$\mid g_1 \mid ... \mid g_n$	$\mid \neg m$	$\mid m \& n$

7.2.2 The algebra of Lists

EXPRESSION	MULTIPLICATIVE IDENTITY	GENERATORS	MULTIPLICATION
$m, n ::=$	1	$\mid g_1 \mid ... \mid g_n$	$\mid m * n$

7.2.3 The algebra of Sets of Words

Often we want to compose different kinds of collections. Languages offer a good example. Languages are Sets of words – which, as we saw above, can be identified with Lists. That is, languages are Sets of Lists. Just like Sets and Lists the composite also has an algebra, known in the literature as a *quantale*. The free quantale is legitimately identified with the Set of all Sets of Lists of some finite enumeration. Presented as a algebra this looks like

EXPRESSION	ADDITIVE IDENTITY	MULTIPLICATIVE IDENTITY	GENERATORS
$m, n ::=$	T	1	$\mid g_1 \mid ... \mid g_n$

ADDITION	MULTIPLICATION
$\mid m\&n$	$\mid m * n$

7.3 Algebras

7.3.1 Kleisli

7.3.2 Eilenberg-Moore

7.4 Monad as container

TBD

7.5 Monads and take-out

7.5.1 Option as container

7.5.2 I/O monad for contrast

7.5.3 Matching gazintas and gazoutas

Intuitionistic discipline

Linear discipline

7.6 Co-monad and take-out

7.7 Hopf structure

7.8 Container and control

7.8.1 Delimited continuations reconsidered

Chapter 8

Domain model, storage and state

Mapping to the backend

TBD

8.1 Mapping our domain model to storage

8.1.1 Functional and relational models

8.1.2 Functional and XML models

8.1.3 ORM

8.2 Storage and language-integrated query

8.2.1 LINQ and for-comprehensions

Open source implementations

ScalaQuery

Squeryl

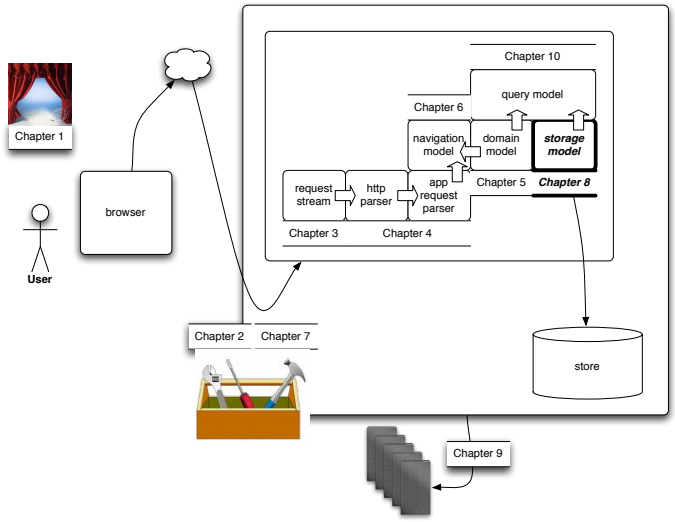


Figure 8.1: Chapter map

8.3 Continuations revisited

8.3.1 Stored state

8.3.2 Transactions

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

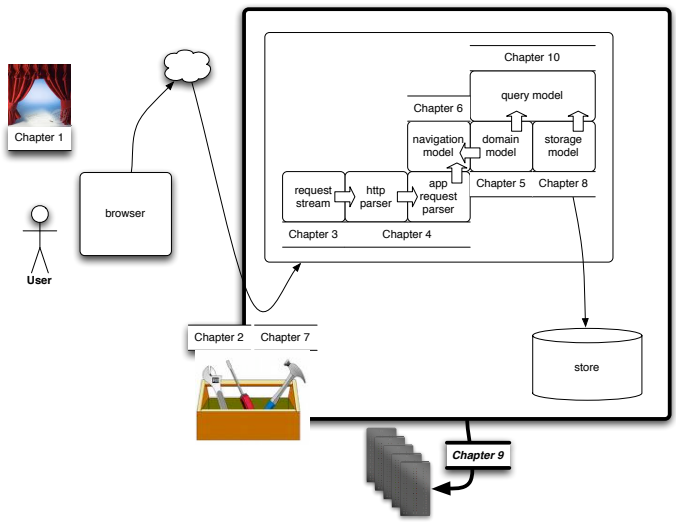


Figure 9.1: Chapter map

Chapter 10

The semantic web

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

10.1 Referential transparency

In the interest of complete transparency, it is important for me to be clear about my position on the current approach to the semantic web. As early as 2004 i appeared in print as stating a complete lack of confidence regarding meta-data, tags and ontology-based approaches. Despite the attention and intense efforts around technologies like `OWL`, i am unaware of major success stories. The funny thing is, the same could be said of similar sorts of efforts underway two decades before that, such as `KIF`, and those two decades before that. i realize this is a controversial position. However, since i worked one floor above Doug Lenat's team at MCC, i feel i have a particular vantage point some 30 years on to ask, so what has CyC done for you lately? In my humble opinion, the theory of programming language semantics, especially compositional accounts as found in λ-calculus and π-calculus, is currently the best foundation we have for a theory we have of semantics, period. As such it constitutes the most sound basis for a good account of *knowledge representation*.

To make good on this claim, i want to illustrate how the monadic techniques provide a new foundation for search on a semantic basis. In particular, what we will see in the following sections of the concluding chapter is how to use monads to search for programs in our toy language on the basis of their structure and their *behavior*! Despite the fact that the open source movement has created such a demand for higher-level techniques to search code repositories, at present writing, i am unaware of any system, not `Hoogle`, not `Merobase`, not `Google Codebase`, nor any of the other of several dozen efforts in this direction, that offer this feature. Yet, the

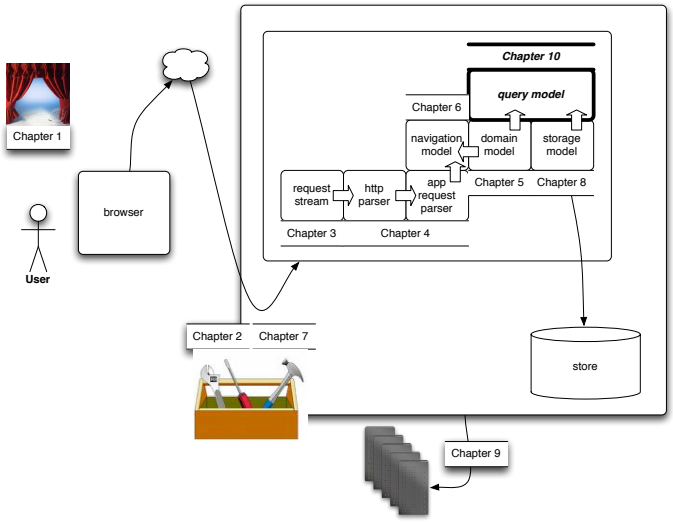


Figure 10.1: Chapter map

monadic design pattern not only makes it clear that such a feature is a possibility, it makes the organization of the code to do it perfectly tractable. i cannot imagine a more powerful argument for the efficacy of this technique for structuring functional programs.

A little motivation The next couple of sections will introduce some a little more apparatus. Hopefully, by now, the reader is convinced of the value of the more standard theoretical presentations of this kind of material if for no other reason than the evident compression it affords. That said, we recognize the need to ground the introduction of new apparatus in good use cases. The discussion above can be turned directly into a use case. The central point of this chapter is to develop a query language for searching for programs in our toy language. Following the analogy we established at the outset of this book between **select** ... **from** ... **where** ... and **for**-comprehensions, this query language will allow users to write queries of the form

```
for ( p <- d if c ) yield e
```

where p is a pattern, d is an interface to a data source and c is a predicate constraining the structure and behavior of the program. We will show how to programmatically derive the language of patterns and the language of constraints from our toy language.

The first new piece of machinery we need to introduce is how to compose monads.

10.2 Composing monads

In all of the preceding chapters we deferred one of the most important questions: do monads themselves compose? After all, if monad is the proposal to replace the notion of object, and the primary criticism of the notion of object is its lack of support for composition, hadn't we better check that monads compose?

Intriguingly, monads do not automatically compose. That is, if F = (F, unit_F, mult_F) and G = (G, unit_G, mult_G) are monads it does not necessarily follow that

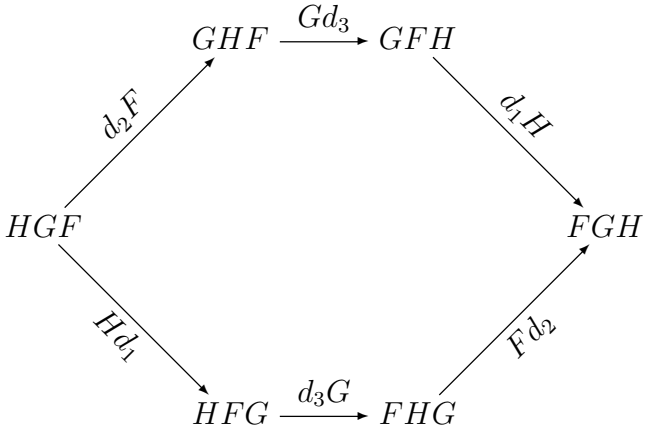
$$\text{F} \circ \text{G} \stackrel{def}{=} (\text{F} \circ \text{G},\ \text{unit}_F \circ \text{unit}_G,\ \text{mult}_f \circ \text{mult}_G)$$

(which we'll write simply as F G going forward) is a monad. In `Haskell` this is one of the purposes of monad transformers, to sketch out a compositional model for monads. Here, we follow a different route. The internal structure of a monad nearly dictates the simplest conditions under which F G forms a monad. Consider the requirement of having a mult for F G. We need a natural transformation from mult: F G F G =>F G.

The components we have to build this mult are primarily mult_F and mult_G. These act to take F F =>G and G G =>G, yet we have F G F G as our initial type. Notice that if we had a way of swapping the interior G F to make it F G, that is, we had a map of the form d : G F =>F G (d for distributive because it distributes F across G), then we could chain up like so

$$FGFG \xrightarrow{F\ d\ G} FFGG \xrightarrow{mult_F\ mult_G} FG$$

It is natural therefore, to require a map like d in order to compose monads. We can investigate whether this proposal scales by looking at how it fairs when we have three monads, F, G and H. We insist on being supplied with distributive maps d_1 : G F => F G, d_2 : H G => G H and, for good measure, d_3 : H F => F H. These will give canonical monads (F G)H and F (G H), but we cannot ensure their equality. That is, we cannot ensure the higher level of associativity. To get this we need to impose an additional set of requirements. These requirements come down to making the following diagram commute.



They are the coherence conditions, the conditions of good interaction amongst the distributive maps. In fact, this is sufficient to scale out to arbitrary collections of monads. That is, if for any pair of monads in the collection we have a distributive map, and for any three we have the switching condition above, then composition is completely coherent and well defined. To illustrate that this is not just some abstract mathematical gadget lets put it to work.

Preliminary

First we will consider a single distributive map. We will look at this in terms of two extremely simple monads, a DSL for forming arithmetic expressions involving only addition, i.e. a monoid, and a monad for collection, in this case Set.

```
case class MonoidExpr[Element]( val e : List[Element] )
class MMExpr[A] extends Monad[A, MonoidExpr] {
  override def unit( e : A ) = MonoidExpr( List( e ) )
  override def mult( mme : MonoidExpr[MonoidExpr[A]] ) =
  mme match {
    case MonoidExpr( Nil ) =>
      MonoidExpr( Nil )
    case MonoidExpr( mes ) =>
      MonoidExpr(
        ( Nil /: mes)(
          { ( acc , me ) => me match {
              case MonoidExpr( es ) => acc +++ es
            }
          }
        )
```

```
        )
    }
}
```

Now, what we need to construct is a map d that takes elements inhabiting the type MMExpr[Set[A]] to elements inhabiting the type Set[MMExpr[A]].

The primary technique is what's called point-wise lifting of operations. Consider a simple example, such as the element

e =MMExpr(List(Set(a_1, a_2), Set(b_1, b_2, b_3))).

This element represents the composition of two sets. We can turn this into a set of compositions, by considering pairs of a's with b's. That is,

```
e match {
  case MMExpr( s1 :: s2 :: Nil ) =>
     Set(
        for( a <- s1; b <- s2 )
        yield { MMExpr( List( a, b ) ) }
     )
  case ...
  }
```

This is exactly the type we want.

10.3 Semantic application queries

An alternative presentation

If you recall, there's an alternative way to present monads that are algebras, like our monoid monad. Algebras are presented in terms of generators and relations. In our case the generators presentation is really just a grammar for monoid expressions.

EXPRESSION	IDENTITY ELEMENT	GENERATORS	MONOID-MULTIPLICATION
$m, n ::=$	e	$\mid g_1 \mid ... \mid g_n$	$\mid m * n$

This is subject to the following constraints, meaning that we will treat syntactic expressions of certain forms as denoting the same element of the monoid. To emphasize the nearly purely syntactic role of these constraints we will use a different symbol for the constraints. We also use the same symbol, $\equiv$, for the smallest equivalence relation respecting these constraints.

IDENTITY LAWS	ASSOCIATIVITY
$m * e \equiv m \equiv e * m$	$m_1 * (m_2 * m_3) \equiv (m_1 * m_2) * m_3$

Logic: the set monad as an algebra In a similar manner, there is a language associated with the monad of sets *considered as an algebra*. This language is very familiar to most programmers.

EXPRESSION	IDENTITY VERITY	NEGATION	CONJUNCTION
$c, d ::=$	$true$	$\mid \neg c$	$\mid c\&d$

Now, if we had a specific set in hand, say L (which we'll call a universe in the sequel), we can interpret the expressions in the this language, aka formulae, in terms of operations on subsets of that set. As with our compiler for the concrete syntax of the *lambda*-calculus in chapter 1, we can express this translation very compactly as

$$[\![true]\!] = L \qquad [\![\neg c]\!] = L \backslash c \qquad [\![c\&d]\!] = [\![c]\!] \cap [\![d]\!]$$

Now, what's happening when we pull the monoid monad through the set monad via a distributive map is this. First, the monoid monad furnishes the universe, L, as the set of expressions generated by the grammar. We'll denote this by $L(m)$. Then, we enrich the set of formulae by the operations of the monoid *acting on sets*.

EXPRESSION	IDENTITY VERITY	NEGATION	CONJUNCTION
$c, d ::=$	$true$	$\mid \neg c$	$\mid c\&d$

IDENTITY VERITY	NEGATION	CONJUNCTION
$\mathbf{e}$	$\mid \mathbf{g_1} \mid ... \mid \mathbf{g_n}$	$\mid c * d$

The identity element, e and the generators of the monoid, g_1, ..., g_n, can be considered 0-ary operations in the same way that we usually consider constants as 0-ary operations. To avoid confusion between these elements and the *logical formulae* that pick them out of the crowd, we write the logical formulae in **boldface**.

Now, we can write our distributive map. Surprisingly, it is exactly a meaning for our logic!

$$[\![true]\!] = L(m) \qquad [\![\neg c]\!] = L(m) \backslash c \qquad [\![c \& d]\!] = [\![c]\!] \cap [\![d]\!]$$

$$[\![\mathbf{e}]\!] = \{\mathbf{m} \in \mathbf{L(m)} \mid \mathbf{m} \equiv \mathbf{e}\} \qquad [\![\mathbf{g_i}]\!] = \{\mathbf{m} \in \mathbf{L(m)} \mid \mathbf{m} \equiv \mathbf{g_i}\}$$

$$[\![c * d]\!] = \{m \in L(m) \mid m \equiv m_1 * m_2, m_1 \in [\![c]\!], m_2 \in [\![d]\!]\}$$

Primes: an application Before going any further, let's look at an example of how to use these new operators. Suppose we wanted to pick out all the elements of the monoid that were not expressible as a composition of other elements. Obviously, for monoids with a finite set of generators, this is exactly just the generators, so we could write $\mathbf{g_1}||...||\mathbf{g_n}$[1]. However, when the set of generators is not finite, as it is when the monoid is the integers under multiplication, we need another way to write this down. That's where our other operators come in handy. A moment's thought suggests that we could say that since *true* denotes any possible element in the monoid, an element is not a composition using negation plus our composition formula, i.e. $\neg(true * true)$. This is a little overkill, however. We just want to eliminate non-trivial compositions. We know how to express the identity element, that's $\mathbf{e}$, so we are interested in those elements that are not the identity, i.e. $\neg\mathbf{e}$. Then a formula that eliminates compositions of non-trivial elements is spelled out $\neg(\neg e * \neg e)$ [2]. Finally, we want to eliminate the identity as a solution. So, we arrive at $\neg(\neg e * \neg e)\&\neg e$. There, that formula picks out the *primes* of *any* monoid.

Summary What have we done? We've illustrated a specific distributive map, one that pulls the set monad through the monoid monad. We've shown that this particular distributive map coincides with giving a semantics to a particular logic, one whose structure is derived solely from the shape of the collection monad, i.e. set, and the shape of the term language, in this case monoid.

The observation that the distributive map is also a semantics for a logic comes about through a kind of factoring. We note that there is a language, the language of Boolean algebra, that takes its meaning in the set monad. As with the monoid monad, the *syntax* of Boolean algebra is given by a monad. The semantics of Boolean algebra can expressed in terms of sets. That is, one can find models for the syntax in terms of sets. In some sense, the distributive map is the unique extension of that semantics map to an enrichment of the syntax with the constructors of the monoid term language.

---

[1] We get the disjunction, $||$, by the usual DeMorgan translation: $c||d \stackrel{def}{=} \neg(\neg c \& \neg d)$

[2] Note the similarity of this construction to the DeMorgan construction of Boolean disjunction. This is, in fact, another kind of disjunction.

Patterns

The constructions of a language of patterns for our monoid expressions is also completely determined by monadic structure. All we are really doing is constructing the data type of 1-holed contexts. In chapter 6 we showed how the derivative of a given regular data type is exactly the 1-holed contexts for the data type. This provides our first example of how to calculate the pattern language for our **for**-comprehensions. After calculation we arrive at

EXPRESSION	HOLE	IDENTITY	GENERATORS	MULTIPLICATION
$m, n ::=$	x	$\mid e$	$\mid g_1 \mid ... \mid g_n$	$\mid m * n$

In some sense, the story here, much like the Sherlock Holmes story, is that the dog didn't bark. The patterns we calculate from our term language are precisely the sorts of patterns we expect if we modeled our term language via `Scala` **case** classes.

A first mini-query language

We can now use these pieces to flesh out some examples of the kinds of queries we might build. The expression

for (x <– d **if** $\neg(\neg e * \neg e)\&\neg e$) **yield** x

will result in a collection of primes residing in the data source d.

for (x <– d **if** $(\neg e * g)$) **yield** x

will result in a collection of expressions residing in the data source d having g as a factor in a non-trivial composition.

Iterating the design pattern

The whole point of working in this manner is that by virtue of its compositional structure it provides a much higher level of abstraction and greater opportunities for reuse. To illustrate the point, we will now iterate the construction using our toy language, the *lambda*-calculus, as the term language. As we saw in chapter 1, the *lambda*-calculus also has a generators and relations presentation. Unlike a monoid, however, the lambda calculus has another piece of machinery: reduction! In addition to structural equivalence of terms (which is a bi-directional relation) there is the *beta*-reduction rule that captures the *behavioral* aspect of the lambda calculus.

It is key to understand this underlying structure of language definitions. In essence, when a DSL is purely about structure it is presented entirely in terms of generators (read: a grammar) and relations (like the monoid laws). When the DSL is also about behavior, i.e. the terms in the language somehow express some kind of computation, then the language has a third component, some kind of reduction relation. [3] This organization, this common factoring of the specification of a language, makes it possible to factor code that handles a wide range of semantic features. The logic we derive below provides a great example.

A spatial-behavioral-style logic for λ-calculus

EXPRESSION	IDENTITY VERITY	NEGATION	CONJUNCTION
$c, d ::=$	$true$	$\mid \neg c$	$\mid c \& d$

MENTION	ABSTRACTION	APPLICATION
$\mid \mathbf{x}$	$\mid (\mathbf{x_1}, \ldots, \mathbf{x_k}) \Rightarrow \mathbf{c}$	$\mid c(c_1, \ldots, c_k)$

LET	SEQ	GROUP
$\mid \mathbf{val\ x = c; d}$	$\mid c; d$	$\mid \{\ c\ \}$

PROBE

$\mid \langle d \rangle c$

The first category of formulae, included for completeness, is again, just the language of Boolean algebra we get because our collection monad is Set. The next category comes directly from the abstract syntax of the λ-calculus. The next group is of interest because it shows that the construction faithfully supports syntactic sugar. The semantics of the "sugar" formulae is the semantics of desugaring factored through our distributive map. These latter two categories allow us to investigate the structure of terms. The final category of formulae, which has only one entry, $PROBE$, is the means of investigating behavior of terms.

Examples Before we get to the formal specification of the semantics of our logic, let's exercise intuition via a few examples.

[3] In some sense this is one of the central contributions of the theory of computation back to mathematics. Algebraists have known for a long time about generators and relations presentations of algebraic structures (of which algebraic data types are a subset). This collective wisdom is studied, for example, in the field of universal algebra. Computational models like the *lambda*-calculus and more recently the process calculi, like Milner's π-calculus or Cardelli and Gordon's ambient calculus, take this presentation one step further and add a set of conditional rewrite rules to express the computational content of the model. It was Milner who first recognized this particular decomposition of language definitions in his seminal paper, Functions as Processes, where he reformulated the presentation π-calculus along these lines.

- **for**(fn(_, ..., _) <- d **if** $true(c_1, ..., c_n)$) **yield** fn

- ```
  for( _( fixpt ) <- d
       if ((f) => ((x) => f(x(x)))((x) => f(x(x))))(true) )
  yield fixpt
  ```

```
for( a <- d if ⟨(x) => ((Yf)x)⟩ a )
yield a
```

The first of these will return the expressions in "function" position applied the actual parameters meeting the conditions c_i respectively. The second will return all actual parameters of expressions that calculate fixpoints. Both of these examples are representative common code optimization schemes that are usually carefully hand-coded. The third example finds all elements in d that are already fixpoints of a given function, f.

Logical semantics

EXPRESSION $c, d ::=$ | IDENTITY VERITY $[\![true]\!] = L(m)$ | NEGATION $| [\![\neg c]\!] = L(m) \backslash [\![c]\!]$ | CONJUNCTION $| c\&d = [\![c]\!] \cap [\![d]\!]$

MENTION
$| \mathbf{x} = \{\mathbf{m} \in \mathbf{L(m)} \mid \mathbf{m} \equiv \mathbf{x}\}$

ABSTRACTION
$| [\![\mathbf{(x_1, ..., x_k)\ =>\ c}]\!] = \{\mathbf{m} \in \mathbf{L(m)} \mid \mathbf{m} \equiv (\mathbf{x_1}, ..., \mathbf{x_k}) => \mathbf{m'}, \mathbf{m'} \in [\![\mathbf{c}]\!]\}$

APPLICATION
$| [\![c(c_1, ..., c_k)]\!] = \{m \in L(m) \mid m \equiv m'(m_1, ..., m_n), m' \in [\![c]\!], m_i \in [\![c_i]\!]\}$

LET $| \mathbf{val\ x = c; d}$ | SEQ $| c; d$ | GROUP $|$ { c }

PROBE
$| [\![\langle d \rangle c]\!] = \{m \in L(m) \mid \exists m' \in [\![d]\!].m'(m) \rightarrow m'', m'' \in [\![c]\!]\}$

Other collection monads, other logics

Stateful collections
```

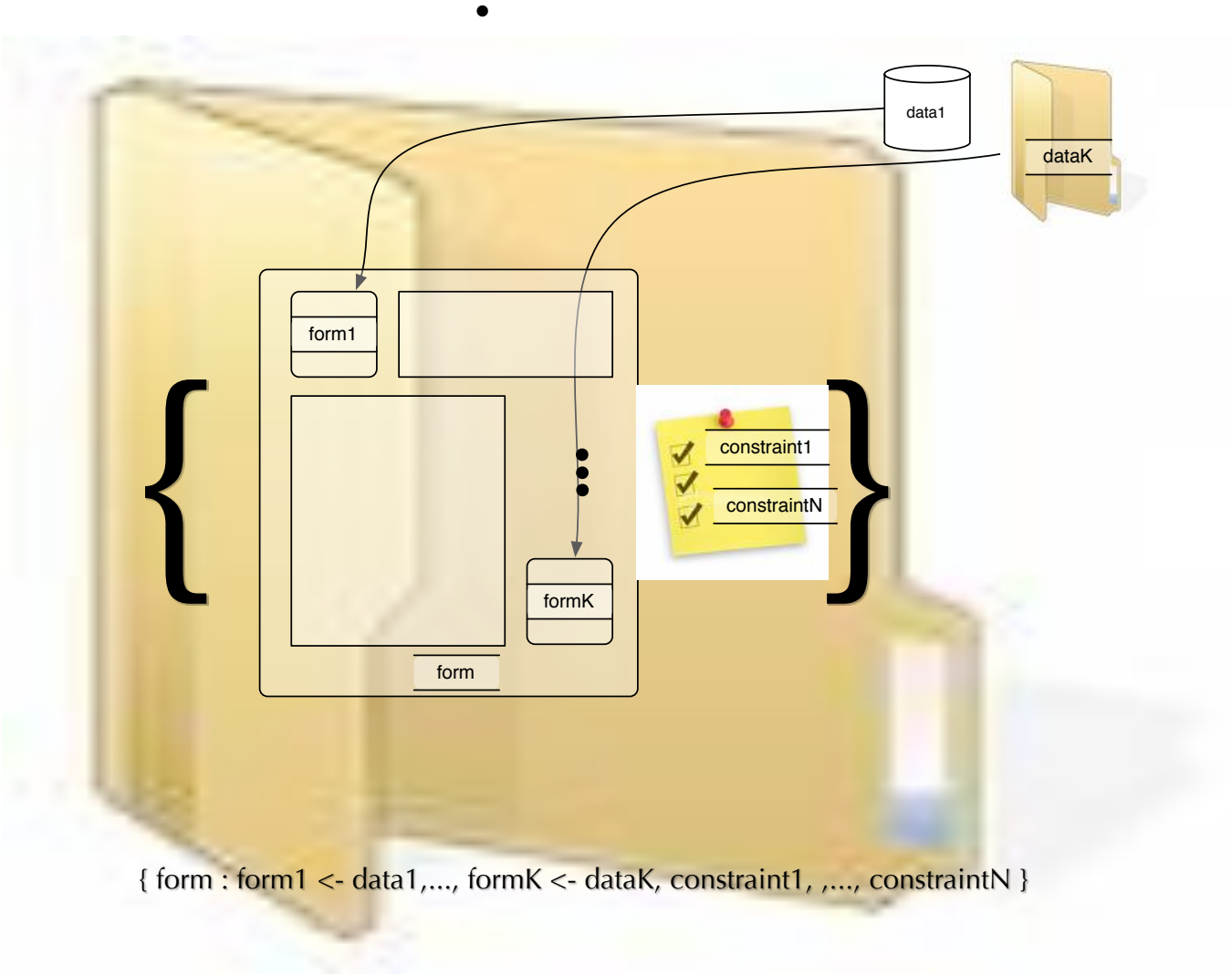


Figure 10.2: Comprehensions and distributive maps

10.3.1 Other logical operations

EXPRESSION	PREVIOUS	QUANTIFICATION	FIXPT DEFN	FIXPT MENTION
$c, d ::=$	$\mid ...$	$\mid \forall v.c$	$\mid rec\ X.c$	$\mid X$

10.4 Searching for programs

10.4.1 A new foundation for search

Monad composition via distributive laws

10.4.2 Examples