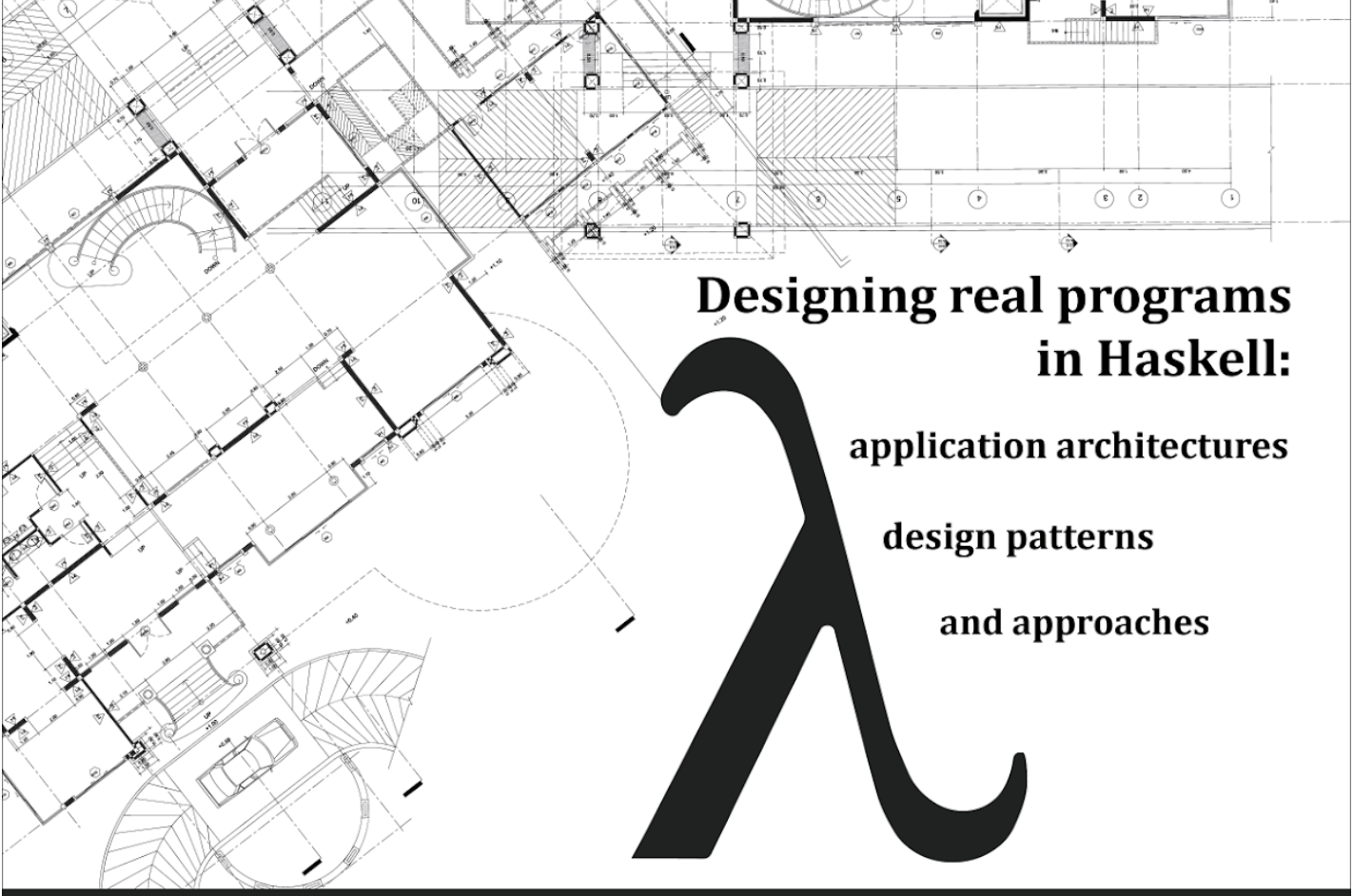


Functional Design and Architecture

Alexander Granin

Functional Design and Architecture

v.1.0.2f

Alexander Granin
graninas@gmail.com

Author's foreword

Hello dear friend,

Thank you for purchasing this book, *Functional Design and Architecture*. This book is about Software Design, Software Engineering in Functional Programming, and Haskell.

The main focus is to provide a comprehensive source of knowledge about Software Design in Haskell: application architectures, best practices, design patterns and approaches — all the needed information on how to build real software with good quality. By reading this book, you'll learn many useful techniques organized into a complete methodology, from requirement analysis to particular subsystems such as database support, multithreading, and logging. You'll learn about design principles, application layers, functional interfaces, Inversion of Control, and Dependency Injection in Haskell.

This book is very practical. It will be useful for software engineers who want to improve their Software Design skills with a strong understanding of many advanced Haskell concepts.

The book requires a certain level of Haskell knowledge, up to intermediate. In fact, *Functional Design and Architecture* can be your second or third Haskell book. If you don't know Haskell, you'll probably find it a bit difficult, although I did my best to provide a good explanation of the Haskell concepts we'll use.

The book is based on two projects:

> **LINK** Andromeda, a simulator of a SCADA system for a spaceship
> https://github.com/graninas/Andromeda
>
> **LINK** Hydra, a comprehensive framework for creating web and console applications
> https://github.com/graninas/Hydra

The code of these projects will help you to better understand the main points of Software Design in Haskell. Moreover, the ideas from this book have been successfully used in several commercial companies, and the methodology the book provides is already proven to be working.

This is the most developed methodology in Haskell today, and I hope you'll enjoy reading the book.

Best wishes,

Alexander Granin

Acknowledgments

Acknowledgments

I'm very thankful to the community and all those who gave light to this project:

- My wife
- Reviewers and readers who made a lot of suggestions on how to improve the book
- Manning Publications for substantial work done for this book in its early stages
- All my patrons and supporters
- All contributors to the book and the related projects

I'm very grateful to top supporters who helped me to make this book even better:

- Special thanks to Vimal Kumar, my first top Patreon supporter, who believed in this project and who believed in me. He did a lot for me and this book, and he gave me the opportunity to try these ideas in his company, Juspay. I'm proud that the knowledge I'm building in the book helped Juspay to become a successful Haskell company!

- Special thanks to Victor Savkov, my second top Patreon supporter, who made a big contribution to the book. He's the first person who offered help when I lost hope of ever finishing it.

- Special thanks to Grammarly, my third top Patreon supporter. It's so cool to get this great service as a sponsor!

Personal acknowledgments to the book's greatest supporters

- I'm very thankful to Nicole Miya Sylvester, who made a good contribution to the Hydra framework!

- Big thanks to Rebecca Deuel-Gallegos, who helped me to improve the text and style!

- Special thanks to these Patrons who made significant contributions to the project:

  Alexander Kucherenko
  Andrei Dziahel
  Andrei Orlov
  Angel Vanegas
  Artur Yushchenko
  Florian Pieper
  Gabriele Lana
  Grammarly
  Henry Laxen
  Ilia Rodionov
  Ivan Zemlyachenko
  James Bartholomew
  John Fast
  John Walker
  Sandy Maguire
  Sergey Homa
  SleepingCat
  Vadim Sotnikov
  Vasily Gorev
  Victor Savkov
  Vimal Kumar
  Yaroslav Voloshchuk
  Yuriy Syrovetskiy

Table of Contents

Part I

Introduction to Functional Declarative Design

Chapter 1

What is software design?

This chapter covers

- An introduction to software design
- Principles and patterns of software design
- Software design and the functional paradigm

Our world is a complex, strange place that can be described using physical math, at least to some degree. The deeper we look into space, time, and matter, the more complex such mathematical formulas become — formulas we must use in order to explain the facts we observe. Finding better abstractions for natural phenomena lets us predict the behavior of the systems involved. We can build wiser and more intelligent things, thus changing everything around us, from quality of life, culture, and technology to the way we think. *Homo sapiens* have come a long way to the present time by climbing the ladder of progress.

If you think about it, you'll see that this ability to describe the Universe using mathematical language isn't an obvious one. There's no intrinsic reason why our world should obey laws developed by physicists and verified by natural experiments. However, it's true: given the same conditions, you can expect the same results. The determinism of physical laws seems to be an unavoidable property of the Universe. Math is a suitable way to explain this determinism.

You may wonder why I'm talking about the Universe in a programming book. Besides the fact that it is an intriguing start for any text, it's also a good

metaphor for the central theme of this book: functional programming from the software design point of view. *Functional Design and Architecture* presents the most interesting ideas about software design and architecture we've discovered thus far in functional programming. You may be asking, why break the status quo — why stray from plain old techniques the imperative world elaborated for us years ago? Good question. I could answer that functional programming techniques can make your code safer, shorter, and better in general. I could also say that some problems are much easier to approach within the functional paradigm. Moreover, I could argue that the functional paradigm is no doubt just as deserving as others. But these are just words — not that convincing and lacking strong facts.

I'll present facts later on in the book. For now, there is a simple, excellent motivation to read this book. Perhaps the main reason I argue for functional programming is that it brings a lot of fun to all aspects of development, including the hardest ones. You've probably heard that parallel and concurrent code is where functional approaches shine. This is true, but it's not the only benefit. In fact, the real power of functional programming is in its ability to make complicated things much simpler and more enjoyable because functional tools are highly consistent and elaborated thanks to their mathematical nature. Considering this, many problems you might face in imperative programming are made more simple or even eliminated in functional programming. Certainly, functional programming has its own problems and drawbacks, but learning new ideas is always profitable because it gives you more opportunity to find better techniques or ways of reasoning.

You're probably looking for new insights into what you already know about software design. The book has three parts:

- Introduction to Functional Declarative Design (chapters 1-2),
- Domain Driven Design (chapters 3-5),
- Designing real world software (chapters 6-9).

You can start from either of the parts, but I recommend reading the chapters in order because they complement each other and help you to build a complete picture of Software Design in Haskell. The first part introduces the discipline of Software Engineering and prepares a ground for a deep discussion on how we do things in functional languages. Two other parts are project-based. The two projects have a slightly different architecture but share some common ideas.

This will help to look onto the same concepts from many points of view and get a better understanding when to apply them and how.

The first project is a software to control simple spaceships.This field is known as *supervisory control and data acquisition software* (SCADA), and it's a rather big and rich one. We certainly can't build a real SCADA application, but we'll try to create a kind of simulator in order to demonstrate the ideas of DDD. We'll track all the stages of software design from requirements gathering to a possibly incomplete but working application. In other words, we'll follow a whole cycle of software creation processes. You don't have to be proficient in SCADA, because I'll be giving you all the information necessary to solve this task. Writing such software requires utilizing many concepts, so it's a good example for showing different sides of functional programming.

The second project represents a framework for building web services, backends, console applications. We'll talk about design patterns, design approaches and practices which help to structure our code properly, to make it less risky and more simple. We'll see how to build layered applications and how to write a testable, maintainable and well-organized code. While building a framework and some demo applications, you'll deal with many challenges that you might expect to meet in the real world: relational and key-value database access, logging, state handling, multithreading and concurrency. What's else important, the ideas presented in this part, are not only theoretical reasoning. They have been successfully tested in production.

In this chapter, you'll find a definition of software design, a description of software complexity, and an overview of known practices. The terms I introduce in this chapter will show that you may already be using some common approaches, and, if not, you'll get a bird's-eye view of how to approach design thinking in three main paradigms: imperative, object-oriented, and functional. It is important to understand when functional programming (sometimes called FP) is better than object-oriented programming (OOP) and when it's not. We'll look at the pros and cons of traditional design methodologies and then see how to build our own.

1.1 Software design

When constructing programs, we want to obey certain requirements in order to make the program's behavior correct. But every time we deal with our complex world, we experience the difficulties of describing the world in terms of code. We can just continue developing, but at some point, we will suddenly realize that we can't go further because of overcomplicated code. Sooner or later, we'll get stuck and fail to meet the requirements. There seems to be a general law of code complexity that symmetrically reflects the phenomenon of entropy:

> Any big, complex system tends to become bigger and more complex.

But if we try to change some parts of such a system, we'll encounter another problem that is very similar to mass in physics:

> Any big, complex system resists our attempts to change it.

Software complexity is the main problem developers deal with. Fortunately, we've found many techniques that help decrease this problem's acuteness. To keep a big program maintainable, correct, and clear, we have to structure it in a particular way. First, the system's behavior should be deterministic because we can't manage chaos. Second, the code should be as simple and clear as possible because we can't maintain Klingon manuscripts.

You might say that many successful systems have an unjustifiably complex structure. True, but would you be happy to support code like that? How much time can you endure working on complex code that you know could be designed better? You can try: the "FizzBuzzEnterpriseEdition" project has an enormous number of Java classes to solve the classic problem FizzBuzz.

LINK *Fizz Buzz Enterprise Edition*
https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

A small portion of these classes, interfaces, and dependencies is presented in the following figure 1.1. Imagine how much weird code there is!

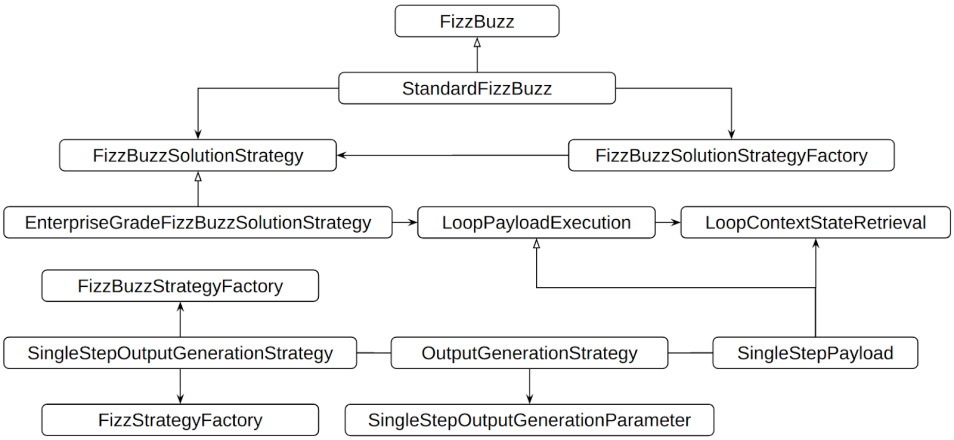


Figure 1.1 FizzBuzz Enterprise Edition class diagram (an excerpt)

So does going functional mean you're guaranteed to write simple and maintainable code? No — like many tools, functional programming can be dangerous when used incorrectly. Consider as evidence: in the following paper, the same FizzBuzz problem is solved in a functional yet mind-blowing manner: “FizzBuzz in Haskell by Embedding a Domain-Specific Language.”

> **LINK** *FizzBuzz in Haskell by Embedding a Domain-Specific Language*
> https://themonadreader.files.wordpress.com/2014/04/fizzbuzz.pdf

That's why software design is important even in Haskell or Scala. But before you design something, you need to understand your goals, limitations, and requirements. Let's examine this now.

1.1.1 Requirements, goals, and simplicity

Imagine you are a budding software architect with a small but ambitious team. One day, a man knocks at your office door and comes in. He introduces himself as a director of Space Z Corporation. He says that they have started a big space project recently and need some spaceship management software. What a wonderful career opportunity for you! You decide to contribute to this project. After discussing some details, you sign an agreement, and now your team is an official contractor of Space Z Corporation. You agree to develop a prototype for date one, to release version 1.0 by date two, and to deliver major update 1.5 by

date three. The director gives you a thick stack of technical documents and contact details for his engineers and other responsible people, so you can explore the construction of the spaceship. You say goodbye, and he leaves. You quickly form a roadmap to understand your future plans. The roadmap — a path of what to do and when — is presented in figure 1.2.

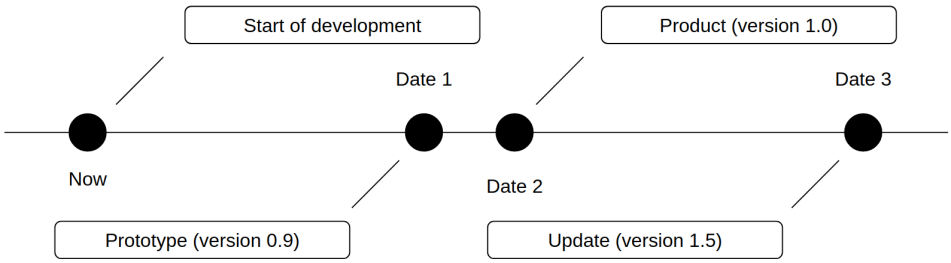


Figure 1.2 Roadmap of the development process

To cut a long story short, you read the documentation inside and out and gather a bunch of requirements for how the spaceship software should work. At this point, you are able to enter the software design phase.

As the space domain dictates, you have to create a robust, fault-tolerant program that works correctly all the time, around the clock. The program should be easy to operate, secure, and compatible with a wide component spectrum. These software property expectations are known as *nonfunctional requirements*. Also, the program should do what it is supposed to do: allow an astronaut to control the ship's systems in manual mode in addition to the fully automatic mode. These expectations are known as *functional requirements*.

DEFINITION *Functional requirements* are the application's requirements for functionality. In other words, functional requirements describe a full set of things the application should do to allow its users to complete their tasks.

DEFINITION *Nonfunctional requirements* are requirements for the application's general properties: performance, stability, extensibility, availability, amounts of data it should be able to process, latency, and so on.

You have to create a program that will meet the requirements and will not necessitate rewriting from scratch — a quite challenging task, with deadlines approaching. Fortunately, you understand the risks. One of them is overcomplicated code, and you would like to avoid this problem. Your goal is not only to create the software on time, but to update it on time too; therefore, you should still be comfortable with the code after a few months.

Designing simple yet powerful code takes time, and it often involves compromises. You will have to maneuver between these three success factors (there are other approaches to this classic problem, but let's consider this one):

- *Goals accomplished*. Your main goal is to deliver the system when it's needed, and it must meet your customer’s expectations: quality, budget, deadlines, support, and so on. There is also a goal to keep risks low, and to be able to handle problems when they arise.
- *Compliant with requirements*. The system must have all the agreed-on functions and properties. It should work correctly.
- *Constant simplicity*. A simple system is maintainable and understandable; simple code allows you to find and fix bugs easily. Newcomers can quickly drop into the project and start modifying the code.

Although fully satisfying each factor is your primary meta-goal, it is often an unattainable ideal in our imperfect world. This might sound fatalistic, but it actually gives you additional possibilities to explore, like factor execution gaps. For example, you might want to focus on some aspects of fault tolerance, even if it means exceeding a deadline by a little. Or you may decide to ignore some spaceship equipment that you know will be out of production soon. The compromises themselves can be represented by a radar chart (see figure 1.3).

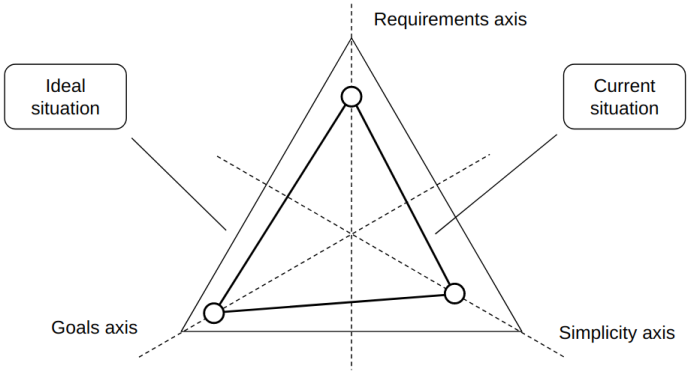


Figure 1.3 Compromising between simplicity, goals, and requirements

Software design is a risk management process. Risks affect our design decisions and may force us to use tools and practices we don't like. We say risk is low when the cost of solving problems is low. We can list the typical risks that any software architect deals with:

- *Low budget.* If we can't hire a good software architect, we can't expect the software to be of production quality.
- *Changing requirements.* Suppose we've finished a system that can serve a thousand clients. For some reason, our system becomes popular, and more and more clients are coming. If our requirement was to serve a thousand clients, we'll face problems when there are millions of clients.
- *Misunderstood requirements.* The feature we have been building over the last six months was described poorly. As a result, we've created a kind of fifth wheel and lost time. When the requirements were clarified, we were forced to start over again.
- *New requirements.* We created a wonderful hammer with nice features like a nail puller, a ruler, pliers, and electrical insulation. What a drama it will be someday to redesign our hammer in order to give it a striking surface.
- *Lack of time.* Lack of time can force us to write quick and dirty code with no thought for design or for the future. It leads to code we're likely to throw in the trash soon.
- *Overcomplicated code.* With code that's difficult to read and maintain, we

lose time trying to understand how it works and how to avoid breaking everything with a small change.

- *Invalid tools and approaches.* We thought using our favorite dynamic language would boost the development significantly, but when we needed to increase performance, we realized it has insuperable disadvantages compared to static languages.

At the beginning of a project, it's important to choose the right tools and approaches for your program's design and architecture. Carefully evaluated and chosen technologies and techniques can make you confident of success later. Making the right decisions now leads to good code in the future. Why should you care? Why not just use mainstream technologies like C++ or Java? Why pay attention to the new fashion today for learning strange things like functional programming? The answer is simple: parallelism, correctness, determinism, and simplicity. Note that I didn't say *easiness*, but *simplicity*. With the functional paradigm comes simplicity of reasoning about parallelism and correctness. That's a significant mental shift.

> **NOTE** To better understand the difference between easiness and simplicity, I recommend watching the talk "Simple Made Easy" (or "Simplicity Matters") by Rich Hickey, the creator of the functional language Clojure and a great functional developer. In his presentation, he speaks about the difference between "simple" and "easy" and how this affects whether we write good or bad code. He shows that we all need to seek simplicity, which can be hard, but is definitely much more beneficial than the easy paths we usually like to follow. This talk is useful not only for functional developers; it is a mind-expanding speech of value to every professional developer, without exception. Sometimes we don't understand how bad we are at making programming decisions.

You'll be dealing with these challenges every day, but what tools do you have to make these risks lower? In general, software design is that tool: you want to create an application, but you also want to decrease any potential problems in the future. Let's continue walking in the mythical architect's shoes and see what software design is.

1.1.2 Defining software design

You are meditating over the documentation. After a while, you end up with a set of diagrams. These diagrams show actors, actions, and the context of those actions. Actors — stick figures in the pictures — evaluate actions. For example, an astronaut starts and stops the engine in the context of the control subsystem. These kinds of diagrams — *use case diagrams* — come from the Unified Modeling Language (UML), and you've decided to use them to organize your requirements in a traditional way. One of the use case diagrams is presented in figure 1.4.

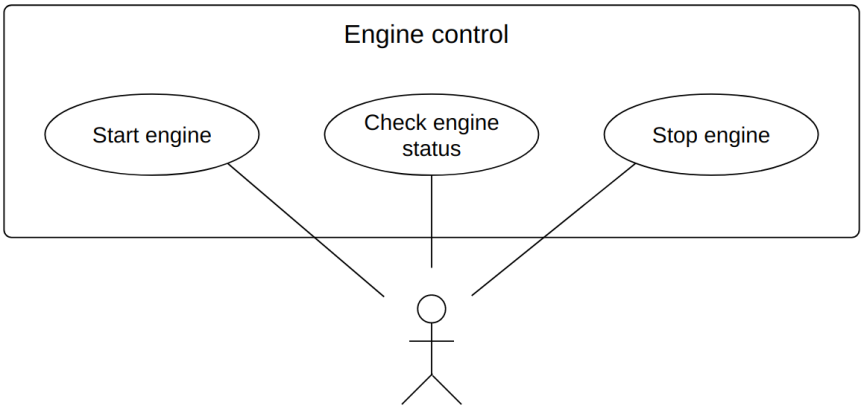


Figure 1.4 Use case diagram for the engine control subsystem

> **NOTE** Use case diagrams are a part of UML, which is primarily used for object-oriented design. But looking at the diagram, can you say how they are related to OOP? In fact, use case diagrams are paradigm-agnostic, so they can be used to express requirements regardless of the implementation stack. However, we will see how some UML diagrams lead to imperative thinking and can't be used directly in functional declarative design.

Thinking about the program's architecture, you notice that the diagrams are complex, dense, and highly detailed. The list of subsystems the astronaut will work with is huge, and there are two or three instances of many of those subsystems. Duplication of critical units should prevent the ship's loss in case of

disaster or technical failure. Communication protocols between subsystems are developed in the same vein of fault tolerance, and every command carries a recovery code. The whole scheme looks very sophisticated, and there is no way to simplify it or ignore any of these issues. You must support all of the required features because this complexity is an inherent property of the spaceship control software. This type of unavoidable complexity has a special name: *essential complexity*. The integral properties every big system has make our solutions big and heavy too.

The technical documentation contains a long list of subsystem commands. An excerpt is shown in table 1.1.

Table 1.1 A small portion of the imaginary reference of subsystem commands

Command	**Native API function**
Start boosters	`int send(BOOSTERS, START, 0)`
Stop boosters	`int send(BOOSTERS, STOP, 0)`
Start rotary engine	`core::request::result request_start(core::RotaryEngine)`
Stop rotary engine	`core::request::result request_stop(core::RotaryEngine)`

Mixing components' manufacturers makes the API too messy. This is your reality, and you can do nothing to change it. These functions have to be called somewhere in your program. Your task is to hide native calls behind an abstraction, which will keep your program concise, clean, and testable. After meditating over the list of commands, you write down some possible solutions that come to mind:

- No abstractions. Native calls only.
- Create a runtime mapping between native functions and higher-level functions.

- Create a compile-time mapping (side note: how should this work?).
- Wrap every native command in a polymorphic object (Command pattern).
- Wrap the native API with a higher-level API with the interfaces and syntax unified.
- Create a unified embedded domain-specific language (DSL).
- Create a unified external DSL.

Without going into detail, it's easy to see that all the solutions are very different. Aside from architectural advantages and disadvantages, every solution has its own complexity depending on many factors. Thanks to your position, you can weigh the pros and cons and choose the best one. Your decisions affect a type of complexity known as *accidental complexity*. Accidental complexity is not a property of the system; it didn't exist before you created the code itself. When you write unreasonably tricky code, you increase the accidental complexity.

We reject the idea of abstracting the native calls — that would decrease the code's maintainability and increase the accidental complexity. We don't think about overdesigning while making new levels of abstractions — that would have extremely bad effects on accidental complexity too.

Figure 1.5 compares the factors in two solutions that affect accidental and essential complexity.

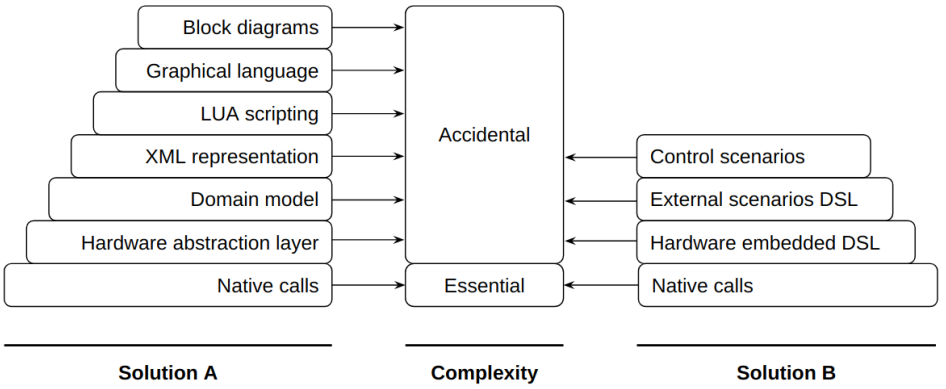


Figure 1.5 Accidental and essential complexity of two solutions

Software design is a creative activity in the sense that there is no general path to the perfect solution. Maybe the perfect solution doesn't even exist. All the time we're designing, we will have to balance controversial options. That's why we want to know software design best practices and patterns: our predecessors have already encountered such problems and invented handy solutions that we can use too.

Now we are able to formulate what software design is and the main task of this activity.

> **DEFINITION** *Software design* is the process of implementing the domain model and requirements in high-level code composition. It's aimed at accomplishing goals. The result of software design can be represented as software design documents, high-level code structures, diagrams, or other software artifacts. The main task of software design is to keep the accidental complexity as low as possible, but not at the expense of other factors.

An example of object-oriented design (OOD) is presented in figure 1.6.

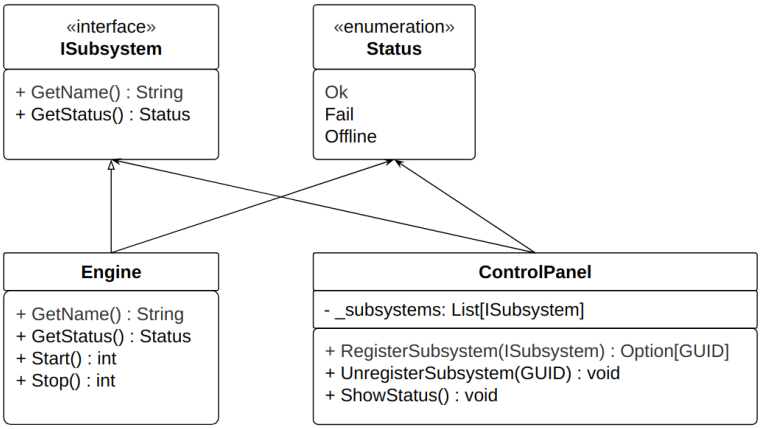


Figure 1.6 OOD class diagram for the engine control subsystem

Here, you can see a class diagram that describes the high-level organization of a small part of the domain model. Class diagrams may be the best-known part of UML, which has been widely used in OOD recently. Class diagrams help object-oriented developers communicate with each other and express their ideas

before coding. An interesting question here is how applicable UML is to functional programming. We traditionally don't have objects and state in functional programming — does that really mean we can't use UML diagrams? I'll answer this question in the following chapters.

Someone could ask: why not skip object-oriented concepts? We are functional developers, after all. The answer is quite simple: many object-oriented practices lead to functional code! How so? See the next chapter — we'll discuss why classic design patterns try to overcome the lack of functional programming in languages. Here, we will take only a brief tour of some major design principles (not patterns!): low coupling and high cohesion. This is all about keeping complexity manageable in OOD and, in fact, in other methodologies.

1.1.3 Low coupling, high cohesion

As team leader, you want the code your team produces to be of good quality, suitable for the space field. You've just finished reviewing some of your developers' code and you are in a bad mood. The task was extremely simple: read data from a thermometer, transform it into an internal representation, and send it to the remote server. But you've seen some unsatisfactory code in one class. The following listing in Scala shows the relevant part of it.

Listing 1.1 Highly coupled object-oriented code

```
object Observer {
  def readAndSendTemperature() {
    def toCelsius(data: native.core.Temperature) : Float =
      data match {
        case native.core.Kelvin(v)  => 273.15f - v
        case native.core.Celsius(v) => v
      }

      val received = native.core.thermometer.getData()
      val inCelsius = toCelsius(received)
      val corrected = inCelsius - 12.5f                    #A
      server.connection
            .send("temperature", "T-201A", corrected)
  }
}
```

#A Defective device!

Look at the code. The transformation algorithm hasn’t been tested at all! Why? Because there is no way to test this code in laboratory conditions. You need a real thermometer connected and a real server online to evaluate all the commands. You can't do this in tests. As a result, the code contains an error in converting from Kelvin to Celsius that might have gone undetected. The right formula should be `v - 273.15f`. Also, this code has magic constants and secret knowledge about a manufacturing defect in the thermometer.

The class is highly coupled with the outer systems, which makes it unpredictable. It would not be an exaggeration to say we don't know if this code will even work. Also, the code violates the Single Responsibility Principle (SRP): it does too much, so it has low cohesion. Finally, it's bad because the logic we embedded into this class is untestable because we can’t access these subsystems in tests.

Solving these problems requires introducing new levels of abstraction. You need interfaces to hide native functions and side effects to have these responsibilities separated from each other. You probably want an interface for the transformation algorithm itself. After refactoring, your code could look like this.

Listing 1.2 Loosely coupled object-oriented code

```
trait ISensor {
  def getData() : Float
  def getName() : String
  def getDataType() : String
}

trait IConnection {
  def send(name: String, dataType: String, v: Float)
}

final class Observer (val sensor: ISensor,
                      val connection: IConnection) {
  def readAndSendData() {
    val data = sensor.getData()
    val sensorName = sensor.getName()
    val dataType = sensor.getDataType()
    connection.send(sensorName, dataType, data)
}}
```

Here, the `ISensor` interface represents a general sensor device, and you don't need to know too much about that device. It may be defective, but your code isn't responsible for fixing defects; that should be done in the concrete implementations of `ISensor`. `IConnection` has a small method to send data to a destination: it can be a remote server, a database, or something else. It doesn't matter to your code what implementation is used behind the interface. A class diagram of this simple code is shown in figure 1.7.

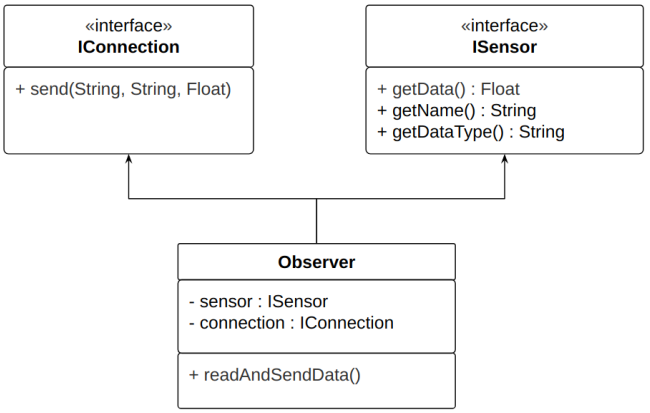


Figure 1.7. Class diagram of listing 1.2

Achieving low coupling and high cohesion is a general principle of software design. Do you think this principle is applicable to functional programming? Can functional code be highly coupled or loosely coupled? Both answers are "yes." Consider the code in listing 1.3: it's functional (because of Haskell!) but has exactly the same issues as the code in listing 1.1.

Listing 1.3 Highly coupled functional code

```
import qualified Native.Core.Thermometer as T
import qualified ServerContext.Connection as C

readThermometer :: String -> IO T.Temperature              #A
readThermometer name = T.read name

sendTemperature :: String -> Float -> IO ()                #B
sendTemperature name t = C.send "temperature" name t
```

```
readTemperature :: IO Float                                    #C
readTemperature = do
  t1 <- readThermometer "T-201A"
  return $ case t1 of
    T.Kelvin  v -> 273.15 – v
    T.Celsius v -> v

readAndSend :: IO ()                                           #D
readAndSend = do
  t1 <- readTemperature
  let t2 = t1 - 12.5              -- defect device!
  sendTemperature "T-201A" t2

#A Native impure call to thermometer
#B Server impure call
#C Impure function that depends on native call
#D Highly coupled impure function with a lot of dependencies
```

NOTE We'll be discussing this code more closely in chapters 3 and 4.

We call the functions `read` and `send` *impure*. These are functions that work with the native device and remote server. The problem here is finding a straightforward approach to dealing with side effects. There are good solutions in the object-oriented world that help to keep code loosely coupled. The functional paradigm tries to handle this problem in another way. For example, the code in listing 1.3 can be made less tightly coupled by introducing a DSL for native calls. We can build a scenario using this DSL, so the client code will only work with the DSL, and its dependency on native calls will be eliminated. We then have two options: first, we can use a native translator for the DSL that converts high-level commands to native functions; second, we can test our scenario separately by inventing some testing interpreter. Listing 1.4 shows an example of how this can be done. The DSL `ActionDsl` shown here is not ideal and has some disadvantages, but we'll ignore those details for now.

Listing 1.4 Loosely coupled functional code

```
type DeviceName = String
type DataType = String
type TransformF a = Float -> ActionDsl a

data ActionDsl a                                          #A
  = ReadDevice DeviceName (a -> ActionDsl a)
  | Transform (a -> Float) a (TransformF a)
  | Correct (Float -> Float) Float (TransformF a)
  | Send DataType DeviceName Float

transform (T.Kelvin  v) = v - 273.15                      #B
transform (T.Celsius v) = v
correction v = v - 12.5
therm = Thermometer "T-800"                               #C

scenario :: ActionDsl T.Temperature                       #D
scenario =
  ReadDevice therm (\v ->
  Transform transform v (\v1 ->
  Correct correction v1 (\v2 ->
  Send temp therm v2)))

interpret :: ActionDsl T.Temperature -> IO ()             #E
interpret (ReadDevice n a) = do
  v <- T.read n
  interpret (a v)
interpret (Transform f v a) = interpret (a (f v))
interpret (Correct f v a)   = interpret (a (f v))
interpret (Send t n v)      = C.send t n v

readAndSend :: IO ()
readAndSend = interpret scenario
```

#A Embedded DSL for observing scenarios
#B Pure auxiliary functions
#C Some hardcoded thermometer identifier
#D Straightforward pure scenario of reading and sending data
#E Impure scenario interpreter that uses native functions

NOTE We'll be discussing this code more closely in chapters 3 and 4.

By having the DSL in between the native calls and our program code, we achieve loose coupling and less dependency from a low level. The idea of DSLs in functional programming is so common and natural that we can find it everywhere. Most functional programs are built of many small internal DSLs addressing different domain parts. We will construct many DSLs for different tasks in this book.

There are other brilliant patterns and idioms in functional programming. I've said that no one concept gives you a silver bullet, but the functional paradigm seems to be a really, really, really good try. I'll discuss it more in the following chapters.

1.1.4 Interfaces, Inversion of Control, and Modularity

Functional programming provides new methods of software design, but does it invent any design principles? Let's deal with this. Look at the solutions in listings 1.1 and 1.2. We separate interface from implementation. Separating parts from each other to make them easy to maintain rises to a well-known general principle, "divide and conquer." Its realization may vary depending on the paradigm and concrete language features. As we know, this idea has come to us from ancient times, where politicians used it to rule disunited nations, and it works very well today — no matter what area of engineering you have chosen.

Interfaces in object-oriented languages like Scala, C#, or Java are a form of this principle too. An object-oriented interface declares an abstract way of communicating with the underlying subsystem without knowing much about its internal structure. Client code depends on abstraction and sees no more than it should: a little set of methods and properties. The client code knows nothing about the concrete implementation it works with. It's also possible to substitute one implementation for another, and the client code will stay the same. A set of such interfaces forms an *application programming interface* (API).

> **DEFINITION** "In computer programming, an application programming interface (API) is a set of routines, protocols, and tools for building software and applications. An API expresses a software component in terms of its operations, inputs, outputs, and underlying types, defining functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface. A good API makes it easier to develop a

program by providing all the building blocks, which are then put together by the programmer" (see Wikipedia: "Application Programming Interface").

LINK Wikipedia: *Application Programming Interface*
https://en.wikipedia.org/wiki/Application_programming_interface

Passing the implementation behind the interface to the client code is known as *Inversion of Control (IoC)*. With IoC, we make our code depend on the abstraction, not on the implementation, which leads to loosely coupled code. An example of this is shown in listing 1.5. This code complements the code in listing 1.2.

Listing 1.5 Interfaces and inversion of control

```
final class Receiver extends IConnection {
  def send(name: String, dataType: String, v: Float) =
    server.connection.send(name, dataType, v)
}

final class Thermometer extends ISensor {
  val correction = -12.5f
  def transform(data: native.core.Temperature) : Float =
    toCelsius(data) + correction

  def getName() : String = "T-201A"
  def getDataType() : String = "temperature"
  def getData() : Float = {
    val data = native.core.thermometer.getData()
    transform(data)
  }
}

object Worker {
  def observeThermometerData() {
    val t = new Thermometer()
    val r = new Receiver()
    val observer = new Observer(t, r)
    observer.readAndSendData()
  }
}
```

The full class diagram of listings 1.2 and 1.5 is presented in figure 1.8.

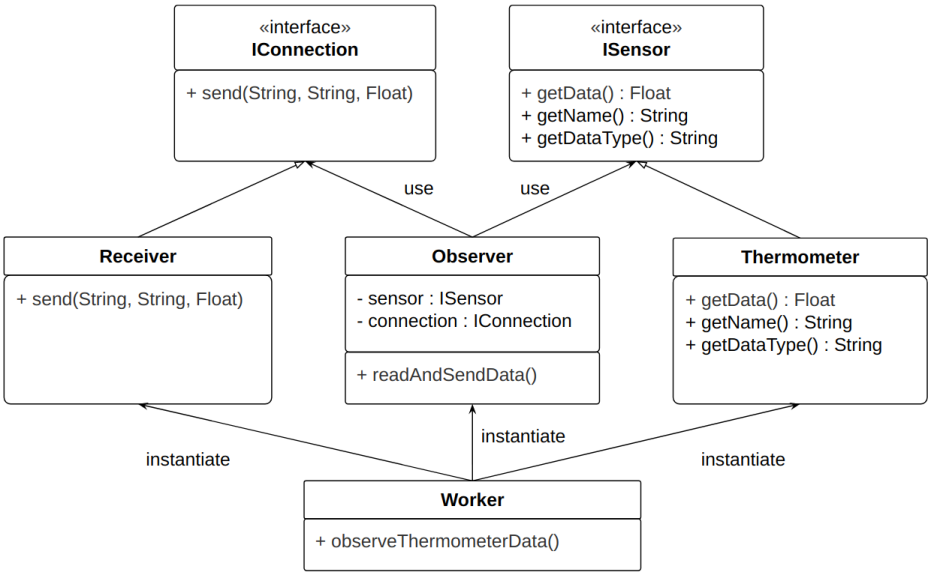


Figure 1.8 Full class diagram of listings 1.2 and 1.5

Now, we are going to do one more simple step. Usually, you have a bunch of object-oriented interfaces related to a few aspects of the domain. To keep your code well organized and maintainable, you may want to group your functionality into packages, services, libraries, or subsystems. We say a program has a *modular* structure if it's divided into independent parts somehow. We can conclude that such design principles as Modularity, IoC, and Interfaces help us to achieve our goal of low software complexity.

Fine. We've discussed OOD in short. But what about functional design? Any time we read articles on OOD, we ask ourselves the following: Is the functional paradigm good for software design too? What are the principles of functional design, and how are they related to object-oriented principles? For example, can we have interfaces in functional code? Yes, we can. Does that mean that we have IoC out of the box? The answer is "yes" again, although our functional interfaces are somewhat different because "functional" is not "object-oriented," obviously. A functional interface for communication between two subsystems can be implemented as an algebraic data type and an interpreter. Or it can be encoded as a state machine. Or it can be monadic. Or it could be built on top of

lenses …. In functional programming, there are many interesting possibilities that are much better and wiser than what an object-oriented paradigm provides. OOP is good, but has to do a lot to keep complexity low. As we will see in the following chapters, functional programming does this much more elegantly.

There is another argument for why the functional paradigm is better: we do have one new principle of software design. This principle can be formulated like this: “The nature of the domain model is often something mathematical. We define what the concept is in the essence, and we get the correct behavior of the model as a consequence.”

When designing a program in the functional paradigm, we must investigate our domain model, its properties, and its nature. This allows us to generalize the properties to functional idioms (for example, functor, monad, or zipper). The right generalization gives us additional tools specific to those concrete functional idioms and already defined in base libraries. This dramatically increases the power of code. For example, if any functional list is a functor, an applicative functor, and a monad, then we can use monadic list comprehensions and automatic parallel computations for free. Wow! We just came to parallelism by knowing a simple fact about the nature of a list. It sounds so amazing — and maybe unclear — and we have to learn more. We will do so in the next chapters. For now, you can just accept that functional programming really comes with new design principles.

Our brief tour of software design has been a bit abstract and general so far. In the rest of this chapter, I’ll discuss software design from three points of view: imperative, object-oriented, and, finally, functional. We want to understand the relations between these paradigms better so that we can operate by the terms consciously.

1.2 Imperative design

In the early computer era (roughly 1950–1990), imperative programming was a dominant paradigm. Almost all big programs were written in C, FORTRAN, COBOL, Ada, or another well-used language. Imperative programming is still the most popular paradigm today, for two reasons: first, many complex systems (like operating system kernels) are idiomatically imperative; second, the widely spread object-oriented paradigm is imperative under the hood. The term *imperative programming* denotes a program control flow in which any data

mutations can happen, and any side effects are allowed. Code usually contains instructions on how to change a variable step-by-step. We can freely use imperative techniques such as loops, mutable plain old data structures, pointers, procedures, and eager computations. So, imperative programming here means *procedural* or *structured programming*.

On the other hand, the term *imperative design* can be understood as a way of program structuring that applies methods like unsafe type casting, variable destructive mutation, or using side effects to get desired low-level properties of the code (for example, maximal CPU cache utilization and avoiding cache misses).

Has the long history of the imperative paradigm produced any design practices and patterns? Definitely. Have we seen these patterns described as much as the object-oriented patterns? It seems we haven't. Despite the fact that OOD is much younger than bare imperative design, it has been much better described. But if you ask system-level developers about the design of imperative code, they will probably name techniques like modularity, polymorphism, and opaque pointers. These terms may sound strange, but there’s nothing new here. In fact, we already discussed these concepts earlier:

- *Modularity* is what allows us to divide a large program into small parts. We use modules to group behavioral meaning in one place. In imperative design, it is a common thing to divide a program into separate parts.
- *Opaque data types* are what allow a subsystem to be divided into two parts: an unstable private implementation and a stable public interface. Hiding the implementation behind the interface is a common idea of good design. Client code can safely use the interface, and it never breaks, even if the implementation changes someday.
- *Polymorphism* is the way to vary implementations under the unifying interface. Polymorphism in an imperative language often simulates an ad hoc polymorphism from OOP.

For example, in the imperative language C, an interface is represented by a public opaque type and the procedures it is used in. The following code is taken from the Linux kernel file as an example of an opaque type.

Listing 1.6 Opaque data type from Linux kernel source code

```
/* These are opaque structures to users.
 * Fields are declared only in the implementation .c files.
 */
typedef struct MYPROCOBJECT_Tag MYPROCOBJECT;
typedef struct MYPROCTYPE_Tag   MYPROCTYPE;

MYPROCOBJECT *visor_proc_CreateObject(MYPROCTYPE *type,
                                      const char *name,
                                      void *context);
void          visor_proc_DestroyObject(MYPROCOBJECT *obj);
```

Low-level imperative language C provides full control over the computer. High-level dynamic imperative language PHP provides full control over the data and types. But having full control over the system can be risky. Developers have less motivation to express their ideas in design because they always have a short path to their goal. It's possible to hack something in code — reinterpret the type of a value, cast a pointer even though there is no information about the needed type, use some language-specific tricks, and so on. Sometimes it's fine, sometimes it's not, but it's definitely not safe and robust. This freedom requires good developers to be disciplined and pushes them to write tests. Limiting the ways a developer could occasionally break something may produce new problems in software design. Despite this, the benefits you gain, such as low risk and good quality of code, can be much more important than any inconveniences that emerge. Let's see how OOD deals with lowering the risks.

1.3 Object-oriented design

In this section, I'll discuss what object-oriented concepts exist, how functional programming reflects them, and why functional programming is gaining huge popularity nowadays.

1.3.1 Object-oriented design patterns

What is OOD? In short, it is software design using object-oriented languages, concepts, patterns, and ideas. Also, OOD is a well-investigated field of knowledge on how to construct big applications with low risk. OOD is focused on the idea of "divide and conquer" in different forms. OOD patterns are

intended to solve common problems in a general, language-agnostic manner. This means you can take a formal, language-agnostic definition of the pattern and translate it into your favorite object-oriented language. For example, the *Adapter* pattern shown here allows you to adapt a mismatched interface to the interface you need in your code.

Listing 1.7 Adapter pattern

```
final class HighAccuracyThermometer {
  def name() : String = "HAT-53-2"
  def getKelvin() : Float = {
      native.core.highAccuracyThermometer.getData()
  }
}

final class HAThermometerAdapter (
    thermometer: HighAccuracyThermometer)
  extends ISensor {
    val t = thermometer

    def getData() : Float = {
        val data = t.getKelvin()
        native.core.utils.toCelsius(data)
    }
    def getName() : String = t.name()
    def getDataType() : String = "temperature"
}
```

The de facto standard for general description of patterns is UML. We are familiar with the case diagrams and class diagrams already, so let's see one more usage of the latter. Figure 1.9 shows the Adapter design pattern structure as it is presented in the classic “Gang of Four” book.

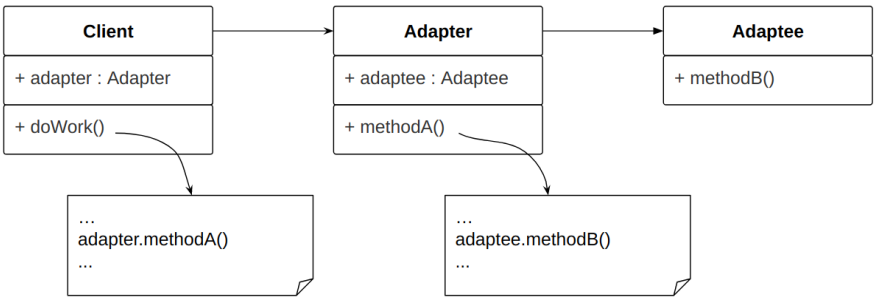


Figure 1.9 The Adapter design pattern

> **NOTE** You can find hundreds of books describing patterns that apply to almost any object-oriented language we have. The largest and most influential work is the book *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1994), which is informally called the "Gang of Four" or just "GoF" book. The two dozen general patterns it introduces have detailed descriptions and explanations of how and when to use them. This book has a systematic approach to solving common design problems in object-oriented languages.

The knowledge of object-oriented patterns is a must for any good developer today. But it seems the features coming to object-oriented languages from a functional paradigm can solve problems better than particular object-oriented patterns. Some patterns (such as Command or Strategy) have a synthetic structure with complex hierarchies involving tons of classes; however, you can replicate the functionality of the patterns with only high-order functions, lambdas, and closures. Functional solutions will be less wordy and will have better maintainability and readability because their parts are very small functions that compose well. I can even say that many object-oriented patterns bypass the limitations of object-oriented languages no matter these patterns' actual purpose.

> **NOTE** As proof of these words, consider some external resources. The article "Design Patterns in Haskell" by Edward Z. Yang will tell you how some design patterns can be rethought using functional concepts. Also, there is notable discussion in StackOverflow in the question "Does Functional Programming Replace GoF Design Patterns?" You can also

> find many different articles that try to comprehend object-oriented patterns from a functional perspective. This is a really hot topic today.

So, we can define object-oriented patterns as well-known solutions to common design problems. But what if you encounter a problem no one pattern can solve? In real development, this dark situation dominates over the light one. The patterns themselves are not the key thing in software design, as you might be thinking. Note that all the patterns use Interfaces and IoC. These are the key things: IoC, Modularity, and Interfaces. And, of course, design principles.

1.3.2 Object-oriented design principles

Let's consider an example. Our spaceship is equipped with smart lamps with program switchers. Every cabin has two daylight lamps on the ceiling and one utility lamp over the table. Both kinds of lamps have a unified API to switch them on and off. The manufacturer of the ship provided sample code for how to use the lamps' API, and we created one general program switcher for convenient electricity management. Our code is very simple:

```
trait ILampSwitcher {
  def switch(onOff: bool)
}

class DaylightLamp extends ILampSwitcher
class TableLamp extends ILampSwitcher

def turnAllOff(lamps: List[ILampSwitcher]) {
  lamps.foreach(_.switch(false))
}
```

What do we see in this listing? Client code can switch off any lamps with the interface `ILampSwitcher`. The interface has a `switch()` method for this. Let's test it! We turn our general switcher off, passing all the existing lamps to it … and a strange thing happens: only one lamp goes dark, and the other lamps stay on. We try again, and the same thing happens. We are facing a problem somewhere in the code — in the native code, to be precise, because our code is extremely simple and clearly has no bugs. The only option we have to solve the problem is to understand what the native code does. Consider the following listing.

Listing 1.8 Concrete lamp code

```
class DaylightLamp (n: String, v: Int, onOff: Boolean)
extends ILampSwitcher {
  var isOn: Boolean = onOff
  var value: Int    = v
  val name: String  = n
  def switch(onOff: Boolean) = {
    isOn = onOff
  }
}

class TableLamp (n: String, onOff: Boolean)
extends ILampSwitcher {
  var isOn: Boolean = onOff
  val name: String  = n
  def switch(onOff: Boolean) = {
    isOn = onOff
    // Debug: will remove it later!
    throw new Exception("switched")
  }
}
```

Stop! There are some circumstances here we have to take into consideration. The manufacturer's programmer forgot to remove the debug code from the method `TableLamp.switch()`. In our code, we assume that the native code will not throw any exceptions or do any other strange things. Why should we be ready for unspecified behavior when the interface `ILampSwitcher` tells us the lamps will be switched on or off and nothing more?

The guarantees that the `ILampSwitcher` interface provides are called a *behavior contract*. We use this contract when we design our code. In this particular situation, we see the violation of the contract by the class `TableLamp`. That's why our client code can be easily broken by any instance of `ILampSwitcher`. This doesn't only happen with the assistance of exceptions. Mutating of global state, reading of an absent file, working with memory — all these things can potentially fail, but the contract doesn't define this behavior explicitly. Violation of an established contract of the subsystem we try to use always makes us think that something is badly implemented. The contracts have to be followed by implementation, otherwise it becomes really hard to predict our program's behavior. This is why so-called *contract*

programming was introduced. It brings some special tools into software design. These tools allow to express the contracts explicitly and to check whether the implementation code violates these contracts or is fine.

Let's show how the contract violation occurs in a class diagram (figure 1.10).

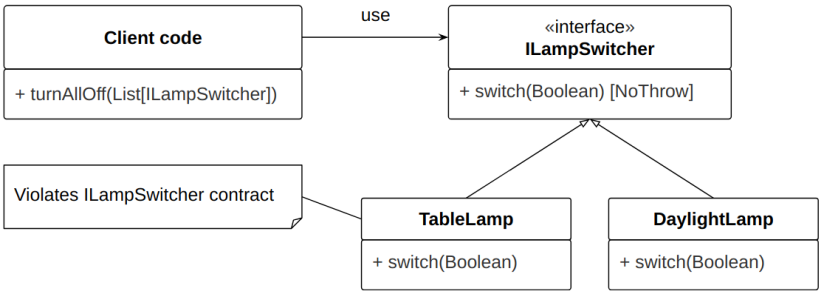


Figure 1.10 Class diagram for listing 1.8 illustrating contract violation by TableLamp

When you use a language that is unable to prevent undesirable things, the only option you have is to establish special rules that all developers must comply with. And once someone has violated a rule, they must fix the mistake. Object-oriented languages are impure and imperative by nature, so developers have invented a few rules, called “object-oriented principles” that should always be followed in order to improve the maintainability and reusability of object-oriented code. You may know them as the *SOLID* principles.

> **NOTE** Robert C. Martin first described the SOLID principles in the early 2000s. SOLID principles allow programmers to create code that is easy to understand and maintain because every part of it has one responsibility, hidden by abstractions, and respects the contracts.

In SOLID, the “L” stands for the Liskov Substitution Principle (LSP). This rule prohibits situations like the one described here. LSP states that if you use `ILampSwitcher`, then the substitution of `ILampSwitcher` by the concrete object `TableLamp` or `DaylightLamp` must be transparent to your code (in other words, your code correctness shouldn’t be specially updated for this substitution), and it shouldn't affect the program’s correctness.

`TableLamp` obviously violates this principle because it throws an unexpected exception and breaks the client code.

Besides the LSP, SOLID contains four more principles of OOP. The components of the acronym are presented in table 1.2.

Table 1.2 SOLID principles

Initial	Stands for	Concept
S	SRP	Single Responsibility Principle
O	OCP	Open/Closed Principle
L	LSP	Liskov Substitution Principle
I	ISP	Interface Segregation Principle
D	DIP	Dependency Inversion Principle

We will return to SOLID principles in the next chapters. For now, I note only that DIP, ISP, and SRP correspond to the ideas mentioned in section 1.1.4: Modularity, Interfaces, and IoC, respectively. That's why SOLID principles are applicable to imperative and functional design, too, and why we should be comfortable with them.

> **NOTE** We also know another design principle set, called GRASP (General Responsibility Assignment Software Patterns). We talked about low coupling, high cohesion, and polymorphism earlier, and those are among the GRASP patterns. GRASP incorporates other OOD patterns too, but they aren't so interesting to us from a functional programming point of view. If you want to learn more about OOD, you can read a comprehensive guide by Craig Larman, *Applying UML and Patterns*, (3rd edition, Prentice Hall, 2004).

1.3.3 Shifting to functional programming

We finished our discussion of imperative design by talking about freedom. Although OOP introduces new ways to express ideas in design (classes, object-oriented interfaces, encapsulation, and so on), it also tries to restrict the absolute freedom of the imperative approach. Have you used a global mutable state, which is considered harmful in most cases? Do you prefer a static cast checked in compile-time or a hard unmanaged cast, the validity of which is unpredictable for the compiler? Then you certainly know how hard it is sometimes to understand why the program crashes and where the bug is. In imperative and object-oriented programming, it's common to debug a program step-by-step with a debugger. Sometimes this is the only debugging technique that can give you an answer about what's happening in the program.

But lowering the need of a debugger (and thus limiting ourselves to other debug techniques) has positive consequences for program design. Instead of investigating its behavior, you are encoding the behavior explicitly, with guarantees of correctness. In fact, step-by-step debugging can be avoided completely. Step-by-step debugging makes a developer lazy in his intentions. He tends to not think about the code behavior and relies on the results of its investigation. And while the developer doesn't need to plan the behavior (he can adjust it while debugging), he will most likely ignore the idea of software design. Unfortunately, it's rather hard to maintain the code without a design. It often looks like a developer's mind dump, and has a significantly increased accidental complexity.

So how could we replace step-by-step debugging?

A greater help to a developer in this will be a compiler, which we can teach to handle many classes of errors; the more we go into the type level, the more errors can be eliminated at the design phase. To make this possible, the type system should be static (compile-time-checkable) and strong (with the minimum of imperative freedom).

Classes and interfaces in an object-oriented language are the elements of its type system. Using the information about the types of objects, the compiler verifies the casting correctness and ensures that you work with objects correctly, in contrast to the ability to cast any pointer to any type freely in imperative programming. This is a good shift from bare imperative freedom to

object-oriented shackles. However, object-oriented languages are still imperative, and consequently have a relatively weak type system. Consider the following code: it does something really bad while it converts temperature values to Celsius:

```
def toCelsius(data: native.core.Temperature) : Float = {
  launchMissile()
  data match {
    case native.core.Kelvin(v)  => toCelsius(v)
    case native.core.Celsius(v) => v
  }
}
```

Do you want to know a curious fact? OOP is used to reduce complexity, but it does nothing about determinism! The compiler can't punish us for this trick because there are no restrictions on the imperative code inside. We are free to do any madness we want, to create any side effects and surprise data mutations. It seems that OOP is stuck in its evolution, and that's why functional programming is waiting for us. Functional programming offers promising ideas for how to handle side effects; how to express a domain model in a wise, composable way; and even how to write parallel code painlessly.

Let's go on to functional programming now.

1.4 Functional declarative design

The first functional language was born in 1958, when John McCarthy invented Lisp. For 50 years, functional programming lived in academia, with functional languages primarily used in scientific research and small niches of business. With Haskell, functional programming was significantly rethought. Haskell (created in 1990) was intended to research the idea of laziness and issues of strong type systems in programming languages. But it also introduced functional idioms and highly mathematical and abstract concepts in the '90s and early 2000s that became a calling card of the whole functional paradigm. I mean, of course, monads. No one imagined that pure functional programming would arouse interest in mainstream programming. But programmers were beginning to realize that the imperative approach is quite deficient in controlling side effects and handling state, and so makes parallel and distributed programming painful.

The time of the functional paradigm had come. Immutability, purity, and wrapping side effects into a safe representation opened doors to parallel programming heaven. Functional programming began to conquer the programming world. You can see a growing number of books on functional programming, and all the mainstream languages have adopted functional programming techniques such as lambdas, closures, first-class functions, immutability, and purity. At a higher level, functional programming has brilliant ideas for software design. Let me give you some quick examples:

- *Functional reactive programming (FRP)* has been used successfully in web development in the form of reactive libraries. FRP is not an easy topic, and adopting it incorrectly may send a project into chaos. Still, FRP shows good potential and attracts more interest nowadays.
- *LINQ* in C#, *streams* in Java, and even *ranges* in C++ are examples of functional approach to data processing.
- *Monads*. This concept deserves its own mention because it can reduce the complexity of some code — for instance, eliminate callback hell or make parsers quite handy.
- *Lenses*. This is an idea in which data structures are handled in a combinatorial way without knowing much about their internals.
- *Functional (monadic) Software Transactional Memory (STM)*. This approach to concurrency is based on a small set of concepts that are being used to handle a concurrent state and do not produce extra accidental complexity. In contrast, raw threads and manual synchronization with mutexes, semaphores, and so on usually turn the code into an unmanageable, mind-blowing puzzle.

Functional developers have researched these and other techniques a lot. They've also found analogues to Interfaces and IoC in the functional world. They did all that was necessary to launch the functional paradigm into the mainstream. But there is still one obstacle remaining. We lack the answer to one important question: how can we tie together all the concepts from the functional programming world to design our software? Is it possible to have an entire application built in a functional language and not sacrifice maintainability, testability, simplicity, and other important characteristics of the code?

This book provides the answer. It's here to create a new field of knowledge. Let's call it *functional declarative design* (FDD). Functional programming is a

subset of the declarative approach, but it is possible to write imperatively in any functional language. Lisp, Scala, and even pure functional Haskell — all these languages have syntactic or conceptual features for true imperative programming. That's why I say “declarative” in my definition of FDD: we will put imperative and object-oriented paradigms away and will strive to achieve declarative thinking. One might wonder if functional programming is really so peculiar in its new way of thinking. Yes, definitely. Functional programming is not just about lambdas, higher-order functions, and closures. It's also about composition, declarative design, and functional idioms. In learning FDD, we will dive into genuine idiomatic functional code.

Let's sow the seeds of FDD.

1.4.1 Immutability, purity, and determinism in FDD

In functional programming, we love immutability. We create bindings of variables, not assignments. When we bind a variable to an expression, it's immutable and just a declaration of the fact that the expression and the variable are equal, interchangeable. We can use either the short name of the variable or the expression itself with no difference.

Assignment operation is destructive by nature: we destroy an old value and replace it with a new one. It is a fact that shared mutable state is the main cause of bugs in parallel or concurrent code. In functional programming, we restrict our freedom by prohibiting data mutations and shared state, so we don't have this class of parallel bugs at all. Of course, we can do destructive assignments if we want: Scala has the `var` keyword, Haskell has the `IO` type and the `IORef` type — but using these imperatively is considered bad practice. It's not functional programming; it's the tempting path to nondeterminism. Sometimes it’s necessary, but more often the mutable state should be avoided.

In functional programming, we love pure functions. A pure function doesn't have side effects. It uses arguments to produce the result and doesn't mutate any state or data. A pure function represents deterministic computation: every time we call a pure function with the same arguments, we get the same result. The combination of two pure functions gives a pure function again. If we have a “pyramid” made of such functions, we have a guarantee that the pyramid behaves predictably on each level. We can illustrate this by code:

```
def max(a: Float, b: Float) = {
  math.max(a, b)
}

def calc(a: Int, b: Int, c: Float) : Float = {
  val sum = a + b
  val average = sum / 2
  max(average, c)
}
```

Also, it is convenient to support a pyramidal functional code: it always has a clear evaluation flow, as in the diagram in figure 1.11.

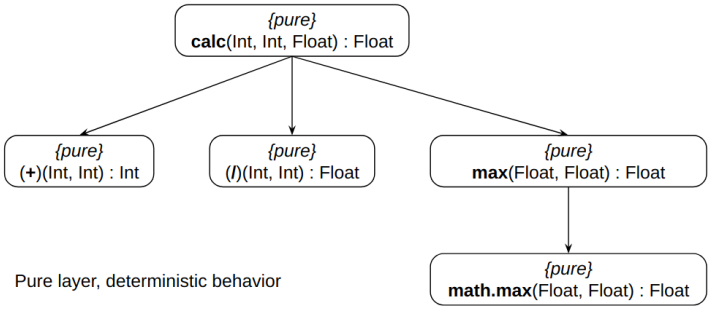


Figure 1.11 Pure pyramidal functional code

We give arguments `a`, `b`, and `c`, and the top function returns the `max` of `average` and `c`. If a year later, we give the same arguments to the function, we will receive the same result.

Unfortunately, only a few languages provide the concept of pure computations. Most languages lack this feature and allow a developer to perform any side effects anywhere in the program. Namely, the `max` function can suddenly write into a file or do something else, and the compiler will be humbly silent about this, as follows:

```
def max(a: Float, b: Float) = {
  launchMissile()
  math.max(a, b)
}
```

And that's the problem. We have to be careful and self-disciplined with our own and third-party code. Code designed to be pure is still vulnerable to nondeterminism if someone breaks its idea of purity. Writing supposedly pure code that can produce side effects is definitely not functional programming.

A modified picture of the impure code is shown in figure 1.12.

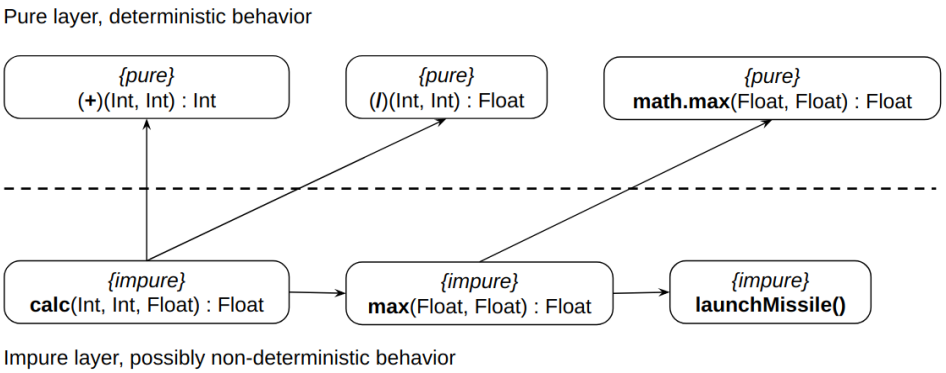


Figure 1.12 Impure pyramidal functional code

Note that there is no way to call impure layer functions from the pure layer: any impure call pollutes our function and moves it into the impure layer. Functional programming forces us to focus on pure functions and decrease the impure layer to the minimum.

Calculation logic like math can be made pure easily. But most data comes from the unsafe, impure world. How to deal with it? Should we stay in that impure layer? The general advice from functional programming says that we still need to separate the two layers. Obviously, we can do this by collecting the data in the impure layer and then calling pure functions. But you might ask what the difference is. Simple: in the impure layer, you are allowed to use destructive assignments and mutable variables. So, you might want to collect the data into a mutable array. After that, you'd better pass this data to the pure layer. There are very few reasons in which you'd like to stay on the impure layer. Maybe, the performance reasons. Anyway, being pure is good!

Let’s consider an example. Suppose we need to calculate the average from a thermometer for one hour with one-minute discretization. We can't avoid using an impure function to get the thermometer readings, but we can pass the math calculations into the pure layer. (Another option would be to use a pure DSL and then interpret it somehow; we did this already in listing 1.4.) Consider the following code in Haskell, which does so:

```
calculateAverage :: [Float] -> Float                    #A
calculateAverage values = ...

observeTemperatureDuring :: Seconds -> IO [Float]       #B
observeTemperatureDuring secs = …

getAverageTemperature :: IO Float
getAverageTemperature = do
  values <- observeTemperatureDuring 60
  return $ calculateAverage values
```

#A Pure calculations
#B Impure data collecting

This design technique — dividing logic into pure and impure parts — is very natural in functional programming. But sometimes it’s hard to design the code so that these two layers don’t interleave occasionally.

> **NOTE** What languages support the purity mechanism? The D programming language has the special declaration `pure`, and Haskell and Clean are pure by default. Rust has some separation of the safe and unsafe worlds. C++ supports pure logic using templates and `constexpr`. The compiler should be able to distinguish side effects in code from pure computations. It does so by analyzing types and code. When the compiler sees a pure function, it then checks whether all internal expressions are pure too. If not, a compile-time error occurs: we must fix the problem.

At the end of this discussion, we will be very demanding. Previously, determinism was denoted implicitly through purity and immutability. Let us demand it from a language compiler explicitly. We need a declaration of determinism in order to make the code self-explanatory. You are reading the code, and you are sure it works as you want it to. With a declaration of

determinism, nothing unpredictable can happen; otherwise, the compilation should fail. With this feature, designing programs with separation of deterministic parts (which always work) and nondeterministic parts (where the side effects can break everything) is extremely desirable. There is no reason to disallow side effects completely; after all, we need to operate databases, filesystems, network, memory, and so on. But isolating this type of nondeterminism sounds promising. Is it possible, or are we asking too much? It's time to talk about strong static type systems.

1.4.2 Strong static type systems in FDD

We have come to the end of our path of self-restraint. I said that expressing determinism can help in reasoning about code. But what exactly does this mean? In FDD, it means that we define deterministic behavior through the type of function. We describe what a function is permitted to do in its type declaration. In doing so, we don't need to know how the function works — what happens in its body. We have the type, and it says enough for us to reason about the code.

Let's write some code. Here, we will use Haskell because its type system has the notion to express impurity; also, it's very mathematical. The following code shows the simplest case: an explicit type cast. Our intention to convert data from one type to another is declared by the function's body. Its type says that we do a conversion from `Int` to `Float`:

```
toFloat :: Int -> Float
toFloat value = fromIntegral value
```

In Haskell, we can't create any side effects here because the return type of the function doesn't support such declarations. This function is pure and deterministic. But what if we want to use some side effects? In Haskell, we should declare it in the return type explicitly. For example, suppose we want to write data into a file; that's a side effect that can fail if something goes wrong with the filesystem. We use a special type to clarify our intent: the return type `IO ()`. Because we only want to write data and don't expect any information back, we use the "unit" type (`()`) after the effect (`IO`). The code may look like the following:

```
writeFloat :: Float -> IO ()
writeFloat value = writeFile "value.txt" (show value)

toFloatAndWrite :: Int -> IO ()
toFloatAndWrite value = let
  value = toFloat 42
  in writeFloat value
```

In Haskell, every function with return type `IO` may do impure calls; as a result, this function isn't pure,[1] and all the applications of this function give impure code. Impurity infects all code, layer by layer. The opposite is also true: all functions without the return type `IO` are pure, and it's impossible to call, for example, `writeFile` or `getDate` from such a function. Why is this important? Let's return to the code in listing 1.4. Function definitions give us all the necessary background on what's going on in the code:

```
scenario :: ActionDsl Temperature
interpret :: ActionDsl Temperature -> IO ()
```

We see a pure function that returns the scenario in `ActionDsl`, and the interpreter takes that scenario to evaluate the impure actions the scenario describes. We get all this information just from the types. Actually, we just implemented the “divide and conquer” rule for a strong static type system. We separate code with side effects from pure code with the help of type declarations. This leads us to a technique of designing software against the types. We define the types of top-level functions and reflect the behavior in them. If we want the code to be extremely safe, we can lift our behavior to the types, which forces the compiler to check the correctness of the logic. This approach, known as *type-level design*, uses such concepts as *type-level calculations*, *advanced types*, and *dependent types*. You may want to use this interesting (but not so easy) design technique if your domain requires absolute correctness of the code. In this book, I’ll discuss a bit of type-level design too.

1.4.3 Functional patterns, idioms, and thinking

While software design is an expensive business, we would like to cut corners where we can by adjusting ready-to-use solutions for our tasks and adopting some design principles to lower risks. Functional programming isn’t something

[1] That’s not entirely true. In Haskell, every function with return type IO () can be considered pure because it only declares the effect and does not evaluate it. Evaluation will happen when the main function is called.

special, and we already know that interesting functional solutions exist. Monads are an example. In Haskell, monads are everywhere. You can do many things with monads: layering, separating side effects, mutating state, handling errors, and so on. Obviously, any book about functional programming must contain a chapter on monads. Monads are so amazing that we must equate them to functional programming! But that's not the goal of this section. We will discuss monads in upcoming chapters, but for now, let's focus on terminology and the place of monads in FDD, irrespective of what they actually do and how they can be used.

What is a monad? A design pattern or a functional idiom? Or both? Can we say patterns and idioms are the same things? To answer these questions, we need to define these terms.

> **DEFINITION** A *design pattern* is the "external" solution to certain types of problems. A pattern is an auxiliary compound mechanism that helps to solve a problem in an abstract, generic way. Design patterns describe how the system *should* work. In particular, OOD patterns address objects and mutable interactions between them. An OOD pattern is constructed by using classes, interfaces, inheritance, and encapsulation.

> **DEFINITION** A *functional idiom* is the internal solution to certain types of problems. It addresses the natural properties of the domain and immutable transformations of those properties. The idiom describes what the domain is and what inseparable mathematical properties it has. Functional idioms introduce new meanings and operations for domain data types.

In the definition of "functional idiom," what properties are we talking about? For example, if you have a functional list, then it is a monad, whether you know this fact or not. Monadic is a mathematical property of the functional list. This is an argument in favor of "monad" being a functional idiom. But from another perspective, it's a design pattern too, because the monadic mechanism is built somewhere "outside" the problem (in monadic libraries, to be precise).

If you feel this introduction to FDD is a bit abstract and lacking in detail, you're completely right. I'm discussing terms and attempting to reveal meaning just by looking at the logical shape of the statements. Do you feel like a scientist? That's

the point! Why? I'll maintain the intrigue but explain it soon. For now, let's consider some code:

```
getUserInitials :: Int -> Maybe Char
getUserInitials key =
  case getUser key users of
    Nothing -> Nothing
    Just user -> case getUserName user of
      Nothing -> Nothing
      Just name -> Just (head name)
```

Here, we can see boilerplate for checking the return values in `case … of` blocks. Let's see if we can write this better using the monadic property of the `Maybe` type:

```
getUserInitials' u = do
  user <- getUser u users
  name <- getUserName user
  Just (head name)
```

Here, we refactored in terms of the results' mathematical meaning. We don't care what the functions `getUser`, `getUserName`, and `head` do, or how they do it; it's not important at all. But we see these functions return a value of the `Maybe` type (because the two alternatives are `Nothing` and `Just`), which is a monadic thing. In the do-block, we've used the generic properties of these monadic functions to bind them monadically and get rid of long if-then-else cascades.

The whole process of functional design looks like this. We are researching the properties of the domain model in order to relate them to functional idioms. When we succeed, we have all the machinery written for the concrete idiom in our toolbox. Functors, applicative functors, monads, monoids, comonads, zippers … that's a lot of tools! This activity turns us into software development scientists, and is what can be called "functional thinking."

Throughout this book, you will learn how to use patterns and idioms in functional programming. Returning to the general principles (Modularity, IoC, and Interfaces), you will see how functional idioms help you to design good-quality functional code.

1.5 Summary

We learned a lot in this chapter. We talked about software design, but only briefly — it is a huge field of knowledge. In addition to OOD, we introduced FDD, denoting the key ideas it exposes. Let's revise the foundations of software design.

Software design is the process of composing application structure. It begins when the requirements are complete; our goal is to implement these requirements in high-level code structures. The result of design can be represented as diagrams (in OOD, usually UML diagrams), high-level function declarations, or even an informal description of application parts. The code we write can be considered a design artifact too.

In software design, we apply object-oriented patterns or reveal functional idioms. Any well-described solution helps us to represent behavior in a better, shorter, clearer way, and thus keep the code maintainable.

FDD is a new field of knowledge. The growing interest in functional programming has generated a lot of research into how to build big applications using functional ideas. We are about to consolidate this knowledge in FDD. FDD will be useful to functional developers, but not only to them: the ideas of functional programming can offer many insights to object-oriented developers in their work.

We also learned about general design principles:

- Modularity
- IoC
- Interfaces

The implementation of these principles may vary in OOP and functional programming, but the ideas are the same. We use software design principles to separate big, complex domains into smaller, less complex parts. We also want to achieve low coupling between the parts and high cohesion within each part. This helps us to pursue the main goal of software design, which is to keep accidental software complexity at a low level.

We are now ready to design something using FDD.

Chapter 2

Application architecture

This chapter covers

- Approaches to collecting requirements
- Top-down architecture design
- Model-driven development of DSLs

As humans, we believe that we have the freedom to make choices. This means we control our decisions, and we have to take responsibility for the results of those choices. In the opposite scenario — in an imaginary universe where everything is predestined, including all of our choices — we wouldn’t be responsible because we wouldn’t be able to change any of these decisions. It would be boring if all events were predefined and arranged in the timeline of the universe. Everyone would be like a robot, without will and mind, born according to one's program and with the sole purpose of evaluating another program, planned many years ago, before dying at the destined time and place. Time would tick by, old robots would disappear, new robots would take vacant places, the Turing machine tape would go to the next round, and everything would repeat again.

We also believe we aren't robots here, in our cozy world. We see the advantages of the fully deterministic model only when we design software. In our programs, we create systems in which behavior is completely defined and well described, similar to the imaginary universe we just talked about. If our programs are similar to that fatalistic universe, then the robots we mentioned are the domain

model entities, with their own lifetimes and purposes; the timeline of events reflects the very natural notion of event-driven design; and there are cycles, scripts, and the construction and destruction of objects when needed. In short, we are demiurges of our local universes.

Where freedom exists, responsibility exists too. In our architectural decisions, we express the ideas of top-level code structure, and any mistakes will hit us in the future. Consider the diagram in figure 2.1.

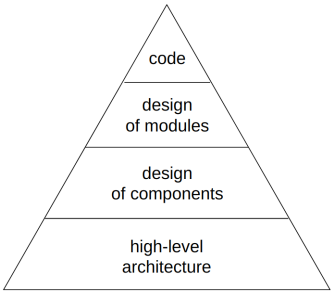


Figure 2.1 Levels of software design

The lower the level where we make decisions, the higher the cost of possible mistakes. Invalid decisions made about architecture are a widespread cause of project deaths. It’s sometimes unclear whether the architectural solution is good enough or not. To understand this, we need to remember three things: there are goals to be accomplished, requirements to be satisfied, and complexity to be eliminated. Good architecture should address these characteristics well. In this chapter, we’ll learn how to design the architecture of a functional application and what the metrics of quality here are.

We'll start working on the SCADA-like system mentioned in chapter 1. Let’s call it “Andromeda.” This project aims to develop a spaceship control and simulation software that the engineers from Space Z Corporation can use in their work. It has many parts and components, all with their own requirements and desired properties. Here, you'll be collecting requirements and designing a high-level project architecture. By the end of the chapter, it will be clear what the scope of development includes and what features can be delayed or even rejected. You'll get a methodology on how to reason about an application from a functional developer's viewpoint, and you'll have tools in your toolbox — design

diagrams that are very useful for carving out a general shape of the future application.

2.1 Defining software architecture

If you ask a nonprogrammer what a particular desktop application consists of, they'll probably answer that the program is something with buttons, text editing boxes, labels, pictures, menus, lists, and folders. This is what's important to users because they can solve some task with the help of this application by interacting with these interface elements. They probably can't, however, say how the program works; it just works, that's all.

Now let's imagine we're talking with someone who knows programming in depth. Even if they've never seen this particular program, they can make assumptions about the largest concepts comprising the program. As an example, take Minesweeper. This application has a graphic user interface (GUI), a field of bombs and numbers, some rules for bombs, and a score dashboard. We might assume that the application has a monolithic structure, with bare WinAPI calls for drawing and controlling the GUI, in which the logic is encoded as transform functions over a two-dimensional array of predefined values ("Bomb", "Blank", and numbers from 1 to 8). Figure 2.2 shows this structure.

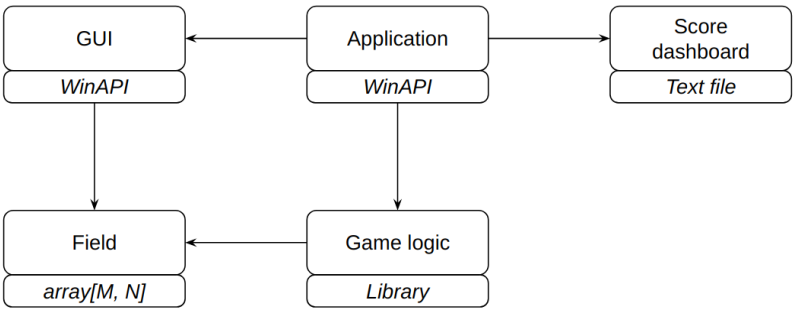


Figure 2.2 Architecture of the Minesweeper game

Another example is Minecraft. It's a little more challenging to say what the game code looks like because it's more complex, but we can make some assumptions because we can see external symptoms of the internal mechanisms. The game has a GUI, server and client parts, networking code, 3D graphics logic, a domain model, an AI solver, a database with game objects,

world-generation logic, game mechanics, and lots and lots of fans. Each of the parts (except the fans, of course) is implemented in the code somehow. If we imagine we are Minecraft developers, we can

- implement 3D graphics using OpenGL,
- use relational database SQLite,
- have a Lua scripting layer for the AI,
- encode game logic in pure functions, and
- mix it all together with some spaghetti code in Java.

Clearly, we won't get into the implementation details; instead, let's look at the top level of code structure only, which we can visualize in a diagram (figure 2.3).

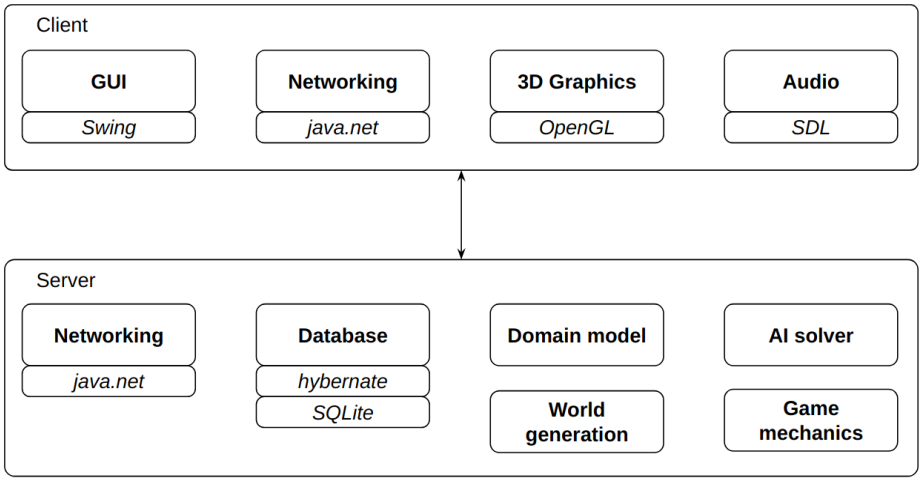


Figure 2.3 Possible architecture of Minecraft

We call this the application's *architecture*. So, architecture is the high-level structure of an application.

> **DEFINITION** Software architecture is a high-level, fundamental description of a software system: a description of its structure, behavior, and properties, as well as the motivation behind the decisions that have led to this architecture.

When newcomers join our team, we give them a diagram of the architecture. They see the big picture and can quickly understand our project. That's how architecture diagrams solve the problem of knowledge transfer. Architecture diagrams are a language every developer should speak to understand the top-level code.

The architectural decisions we make define all aspects of our program, and the cost of mistakes is high because it's often impossible to change an architectural choice. Wrong choices can bury the project easily. To minimize this risk, we elaborate the application's architecture before we start designing low-level code and structure. Good architecture allows us to rework the parts independently, without breaking the rest of the application. Unfortunately, there are no holy tablets with step-by-step guides for how to create a good architecture, but there are some common patterns and practices that help us obey the software requirements (see the "Software architecture" article in Wikipedia). By following these practices, it's possible to make a raw sketch of an architecture, but every project has its own unique characteristics, and so will every architecture.

> **LINK** Wikipedia: *Software architecture*
> https://en.wikipedia.org/wiki/Software_architecture

In this chapter, we'll create the initial architecture of the spaceship control software. It may be imperfect, but it should show how to reason about the domain from a top-level perspective using functional architectural patterns and approaches. What patterns? The next section discusses this question to establish the intuition needed to make architectural decisions.

2.1.1 Software architecture in FDD

To some degree, you may consider that functions (either first-class or higher-order) are the only abstraction all functional languages have in common. Combining functions isn't a big deal: you just apply one to another while building a pyramid of functions. Not so much, you say? But this is a quite sharp

and robust tool that has expressiveness as powerful as the built-in features in other languages. Moreover, you can solve any problem by combining more functions, whereas combining more language features can be painful because of spooky syntax or undefined behavior (example: templates in C++). If a language feature can't solve a problem directly, it pushes a developer to construct a house of cards, stacking more and more features. By doing this, the developer wants to achieve an abstraction that solves the problem, but the taller the house, the more likely it will crash. Abstractions over abstractions are overengineering, and they tend to leak.

You probably know what a *leaky abstraction* is, but let me give you a definition.

> **DEFINITION** A leaky abstraction is an abstraction that reveals implementation details instead of hiding them. The leaky abstraction solves part of the problem while resisting changes that are supposed to solve the whole problem. Also, a leaky abstraction reduces complexity less than what it brings to the code.

Which is worse: building a pyramid of functions or a card house of language features? Can the functional approach lead to leaky abstractions too? Let's see. If you want to use two language features together, you'll have to consult with a language specification. In contrast, all you need to know to combine functions is their type definitions. Functions in functional language are based on a strict mathematical foundation — the lambda calculus — which makes functional abstractions less “leakable” because math is a well-constructed field.

Typical concepts in functional languages include lambdas, regular functions, higher-order functions, immutability, purity, currying, composition, separation of model and behavior, and separation of pure and impure computations — these and other concepts are what functional programming is all about. But at the design level, you would rather abstract from bolts and nuts to more general instruments — to advanced functional idioms. Haskell, for example, suggests a set of concepts derived from category theory, and these abstractions can't leak because they are proven to be consistent.[2] Additionally, you can refer to a few functional patterns that are intended to solve a particular architectural problem in

[2] Note that you can still break these abstractions by violating the laws of the abstractions — for example, the laws of monads.

a functional manner. In practice, they can leak, but this wouldn't usually be the case if they’re used right.

So, what are these idioms and patterns? You'll find informal definitions of them right after the schematic view in figure 2.4.

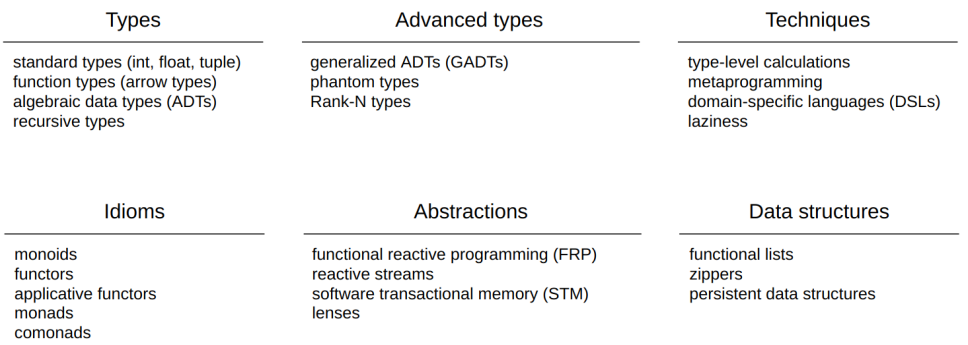


Figure 2.4 Classification of functional ideas

You'll encounter many of these important terms and functional pearls, and the following definitions should help you form a kind of intuition needed to design functionally:

- *Standard types.* These include int, float, tuple, function types (also known as types of functions or arrow types), and so on. In functional programming, almost every design starts from constructing a type that reflects the domain. When you have a type of something, you know how it can be used and what properties it has.
- *Recursion and recursive data types.* Recursion, while being the only way to iterate over things in pure functional programming, also permeates many functional idioms. Interestingly, without recursion in the level of types, it would be hard to represent, for example, a list data type in a static type system.
- *Meta-programming.* Meta-programming in functional languages is often an abstract syntax tree (AST) modification of the code for reducing boilerplate. It may be a powerful tool for design too.
- *DSLs*. A DSL represents the logic of a particular domain by defining its structure, behavior, naming, semantics, or possibly syntax. The benefits of

a DSL are reliability, reducing the complexity of the domain, clearness of code to nonprogrammers, ease of getting things right, and difficulty of getting things wrong. But a DSL requires that we maintain compliance with current requirements.

- *Algebraic data types (ADTs).* An ADT is a composite type consisting of sum types (also known as *variant types*), and every sum type can be composed with a product type (also known as a *tuple* or *record*). ADTs are widely used in functional programming for designing domain models, DSLs, or any kind of data structure.
- *Functional idioms: monoids, functors, applicative functors, monads, comonads, arrows, and others.* These idioms are intrinsic mathematical properties of some data types. Once we've revealed a property of our data type (for example, our ADT is a functor), we gain the next level of abstraction for it. Now our type belongs to the corresponding class of types, and all the library functions defined for that class will work for our type too. We can also create our own library functions for general classes of types. And, of course, there are monads. We'll meet many monads in this book: `State`, `Free`, `Reader`, `Writer`, `Maybe`, `List`, `Either`, `Par`, `IO` — plenty of them. Perhaps no functional designer can avoid inventing monads irrespective of his intentions to do or not do so.
- *Nonstandard complex types: existential types, phantom types, and Rank-N types.* We won't define these here, but you should know that sometimes it's possible to enforce your code by enclosing additional information into your types so the compiler will be checking code validity all the time you are compiling the code.
- *Generalized algebraic data types (GADTs).* The GADT is an extension of the ADT in which nonstandard complex types are allowed. GADTs allow us to lift some logic and behavior to the type level and also solve some problems in a more generic way.
- *Type-level calculations and logic.* Type-level calculations are types too. Why? This is the further development of the idea of getting correctness from types. Correct behavior can be achieved either by testing code and debugging it or by lifting the logic to the type level and proving its correctness via the compiler. Haskell and Scala have many features for type-level calculations: type classes, phantom types, Rank-N types, type families, GADTs, recursive types, meta-programming, and so on.

Learning such unusual concepts for design can be a mind-blowing experience!

- *Laziness*. In the hands of a master, laziness can be a powerful design technique. With laziness, for example, it's possible to transform one data structure of a graph to another effectively. How does this work? You might write a code that looks like it transforms a big structure but actually doesn't. After that, you can compose such "heavy" transformations, but, in fact, when they finally go to evaluation, only a small part of the chained transformations will be evaluated: no more and no less than you need.

These are all functional techniques for the design of code. But functional programming also provides some abstractions and approaches for architecture design:

- *FRP*. FRP is a style of reactive programming merged with functional ideas. In FRP, the notion of time-varying values is introduced. If some other value depends on the first one, it will be automatically updated. These values have a special name: *signals* (or *behaviors*). Every event is mapped to a form of signal, as it can occur, change, repeat, and disappear — it's a time-dependent value. Any logic can then be bound to the signal as a reaction. By having a reactive model composed of signals (often called a *reactive network*) and actions, it's possible to support GUIs and complex business logic in functional programs.
- *Functional reactive streams*. This abstraction addresses similar purposes as FRP. A reactive stream is a continuous stream of events arranged by time. Streams (and thus the events that have occurred) can be mapped, merged, divided, and zipped, forming new types of streams. Every change or absence in a stream can be bound to an action. The stream model implements the data flow concept.
- *STM*. STM represents a model of concurrent state with safe transactions for its modification. Unlike concurrent data structures, STM represents a combinatorial approach, so combining models gives us another STM model. One of the biggest advantages of STM is that it can be implemented as a monad, so you can construct your model monadically and even embed STM into other monads.
- *Lenses*. When you have a deep, immutable data structure, you can't avoid unrolling it (accessing its layers one by one), even for a single change of

the internal value. Then you roll your structures back. The mess that all code turns into after rolling and unrolling a complex structure can dramatically impact its readability. Lenses help keep the code concise by introducing an abstraction for modifying combinatorial values.

Each of the listed patterns has some mathematical base, so you can be sure these abstractions don't leak. I've reviewed the most important of them, but I'll leave detailed descriptions and use cases for other chapters. Unfortunately, this book isn't big enough to contain full information about all the patterns, but I believe specialized books can teach you well.

2.1.2 Top-down development process

In chapter 1, we discussed the phases of the development process, which follow one another until we reach a ready application. We started with requirements analysis. This is the process in which we try to understand the domain we want to program. We collect issues, facts, and notions by reading documentation and talking with engineers. We want to know what we should create. After that comes the software design phase. We translate requirements into a form suitable for developers to begin writing code. In this phase, we design the software architecture, composing it from big independent parts and high-level design patterns. When the big picture is done, we implement details in code. That's why we call this approach *top-down*: we start from top-level concepts of the domain and descend to the details of low-level code. The whole process is described in figure 2.5.

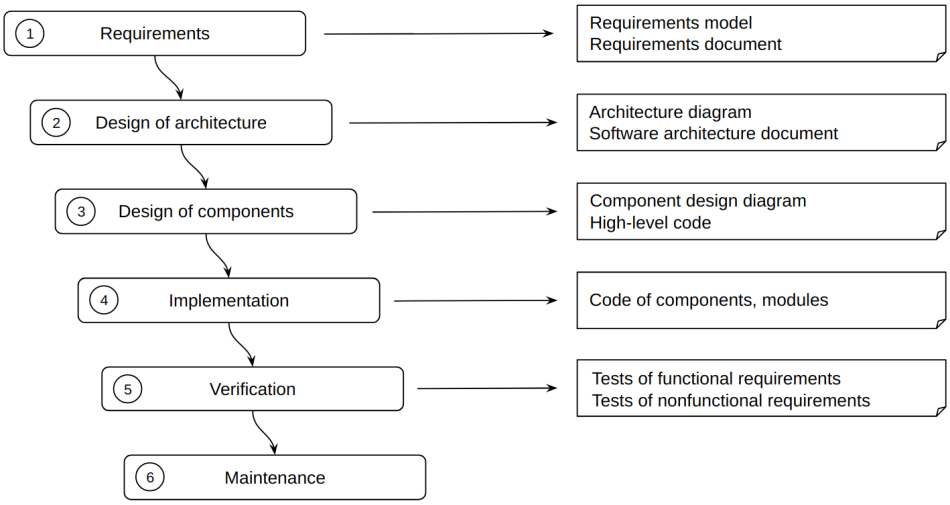


Figure 2.5 The waterfall model of software development

While in this model, we can't return to the previous phase; this flow is an idealization of the development process. It has a well-known name: *waterfall model*. In real life, we'll never be so smart as to foresee all the hidden pitfalls and prevent them before the next phase starts. For example, we may face problems in phases 2, 3, and 4 if we miss something in the initial requirements phase. For this exact reason, we wouldn't follow the waterfall model in this book. We'll adopt iterative and incremental models with cycles rather than this rigid, single-directional one; however, consider the waterfall model as a reference to what activities exist in the software development process.

> **NOTE** In its way, the waterfall model has undergone many editions aimed to make it less resistant to eventual switches between phases. You need to know the waterfall model or its modifications to understand what you're doing now, not to waste time while trying hard to fit your activities into some blocks on a piece of paper. There are other methodologies you might want to learn, so consider the relevant Wikipedia articles and books mentioned there.

The better models allow you to return to the previous phases iteratively, so you can fix any problems that may have occurred. In functional programming, the

iterative top-down approach seems the best choice for initial design. You start from a high level — for example, a function:

```
solveProblem :: Problem -> Solution
```

and then descend into the implementation. Before attacking the problem in code, you need to collect requirements and design the architecture and separate subsystems. Make yourself comfortable, and grab a pencil and paper: we'll be drawing some diagrams to better understand the domain and design options we have.

> **NOTE** Before we continue, let's agree that irrespective of the methodology this book suggests, it does not matter what diagrams you'll be creating. Drawing pictures that illustrate the design of a code seems essential to software developers who want to describe their ideas to others well. And there are far more informal ways of structuring diagrams than formal ones (like UML or like this book suggests). However, in this book, we'll not only design an application in diagrams, but we'll also be collecting and analyzing requirements.

> **NOTE** Also, it's completely fine to just skip diagrams and proceed with the code. There are methodologies that work well and can be as effective as others to get a good project architecture. For example, the test-driven development (TDD) methodology can be used to write a functional programming code pretty easily because testability is one of the features in which functional programming shines. But designing the architecture and parts of the application with TDD requires a bit more knowledge of different approaches. TDD is impossible without the ability to separate an application into independent parts, so we'll need something for our three key techniques: Interfaces, IoC, and Modularity. The next chapters aim to cover the most suitable implementations of these techniques in functional programming.

Designing with diagrams in FDD tries to reveal points of interest, see hidden pitfalls, and elaborate a rough vision of how to overcome problems. Returning to diagrams at any stage of development is normal just because something will always need to be specified or changed. Figure 2.6 shows the whole design process.

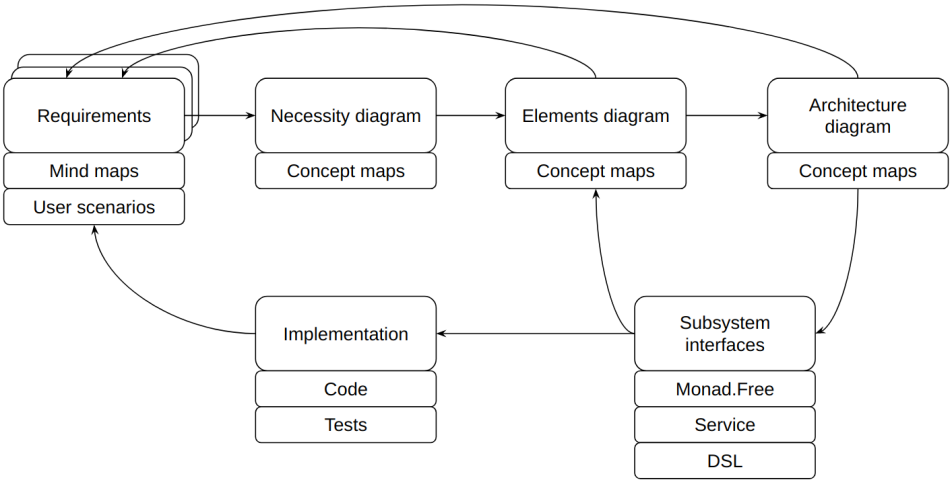


Figure 2.6 Iterative architecture design process in FDD

The diagram describes an iterative process in which every block has a concrete goal.

- *Requirements*. Every development starts from collecting requirements. In FDD, it's recommended that you keep the requirements model in mind maps and user scenarios.
- *Necessity diagram*. This is a concept map containing the application's main parts. It shows the front of the future work and basic dependencies between blocks. It can also be just a list of the big parts of the application.
- *Elements diagram*. This is a set of concept maps that fixate the results of brainstorming about application architecture. The elements diagram may be unstructured and rough, but it helps you estimate every design approach you might think about.
- *Architecture diagram*. This kind of diagram represents the architecture of an application in a structured way. It keeps your design decisions and shows different subsystems with their relations.
- *Subsystem interfaces*. By modeling interfaces between subsystems, you'll better understand many things: how to separate responsibilities, how to

make a subsystem convenient to use, what is the minimal information the client code should know about the subsystem, what is its lifetime, and so on.

- *Implementation*. The implementation phase follows the design phase, but you'll be returning to the design phase all the time because new information is revealed, and requirements are changing.

2.1.3 Collecting requirements

One part of the Andromeda Control Software is the simulator. With it, engineers can test spaceship systems for reliability or check other characteristics before the real spaceship is launched into space. Unfortunately, we can't support many actual SCADA requirements, including a hard real-time environment and minimal latency. To support hard real-timeness, another set of tools should be chosen: a system-level language with embedding possibilities, some specific operating system, programmable hardware (for example, field-programmable gate arrays [FPGAs] or microcontrollers), and so on. But let's consider our program as a kind of training simulator, because we don't need to create a complex, ready-to-use program. We will try, however, to identify as many real requirements as we can to make our development process as close as possible to reality.

In general, we want a comprehensive, consistent, and clear description of the task we want to solve. This will serve as a feature reference for developers and, much more importantly, an agreement with the customer as to what properties the system should have. Let's call this description a *requirements model*.

> **DEFINITION** A requirements model is a set of detailed, clear, and well-structured descriptions of what properties an upcoming product should have. The process of requirements modeling aims to collect the requirements and represent them in a form accessible to all project participants.

While creating the requirements model, you may feel like Sherlock Holmes. You’re trying to extract actual knowledge from your customer, from experts, from the documentation provided, from examples, and from any other sources of information available. You have to be discerning because the facts you’re getting are often very unclear; some of them will be contrary to each other and even to common sense. This is normal because we’re humans, and we make mistakes.

Don't hesitate to ask questions when you need clarification. What's worse: spending more time on requirements analysis or wasting that time creating something vague and useless?

There are many possibilities for creating a requirements model. When you're done, you will have a software requirements document or, less formally, a set of well-formed requirements available to both developers and the customer. The most important ways to gather requirements are as follows:

- *Questions and answers*. This is the best method of investigating what the customer wants done. Questions can be varying: about the purpose of the software, domain terms and their meaning, expectations regarding functionality, and so on. The main disadvantage of questions and answers is the lack of a convenient information structure.
- *Use case diagrams*. A use case diagram shows what actions the actor can do with a subsystem within some logical boundary. An actor can be a user or another subsystem. As mentioned in chapter 1, use case diagrams are part of UML, a standard that suggests a strong structure for the diagrams. Use case diagrams can be used as the agreement between you and your customer about what functionality the program should have. For our intention to design software using a functional language, use case diagrams don't help much because they don't allow us to reveal the domain's functional properties. In order to try our hand, though, we'll create a few of them.
- *User scenarios*. These scenarios describe a user solving a particular problem step-by-step with the help of the application we're constructing. Scenarios may have alternative paths — input conditions and output conditions the system should meet before and after the scenario, respectively. User scenarios can follow use case diagrams or be an independent part of the requirements model. They are written informally, so you and the customer can verify that you understand each other. User scenarios have a good and focused structure and are concrete and comprehensive. We'll write some user scenarios in the next sections.
- *Associative mind maps*. A mind map is a tree-like diagram with notions connected associatively. Mind maps are also called *intellect maps*. The term "mind" or "intellect" here means that you dump your thoughts according to the theme of the diagram. A mind map is an informal tree of ideas, terms, descriptions, comments, pictures, or other forms of

information that look important. There are no standards or recommendations for how to dump your mind in a diagram because this is a highly personal process. You might know the game of associations: a player receives a starting word and must list as many associations as possible. That's exactly what a mind map is like, but, in our case, it shouldn't be so chaotic because we want to use it as a reference for the domain concepts. The associative nature of mind maps makes the brainstorming process simple: we don't care about standardizing our thoughts; we just extract associations regarding a concrete notion and write them down as part of the diagram. We can then reorganize it if we want more order. You will see that mind maps are very useful in software analysis, and they help us understand requirements from an FDD perspective.

NOTE We need to touch on the software we can use for collecting requirements. In chapter 1, we discussed use case diagrams. The requirements model formed by use case diagrams can take up a huge amount of space. To help developers maintain this model easily, specialized computer-aided software engineering (CASE) tools were invented. You can find a lot of free, open-source tools as well as paid proprietary solutions for all the phases of software development: collecting requirements, modeling architecture, designing subsystems, and forming project structure. For mind maps you may want to try Freemind, a free cross-platform tool with a lot of features. With Freemind, you'll be able to create a complex of nice-looking maps connected by hyperlinks. You can break a big diagram into small parts and work with them separately, so this will be a well-organized requirements model. Additionally, if you're an advocate of web applications, there are online tools. But there's nothing better than a good old pencil and piece of paper!

You may use any of these four tools for domain analysis. Questions and answers, user stories, and use case diagrams are well-known, so we'll skip explanations here to have a deeper look at mind maps. As said earlier, associative diagrams seem to be the most suitable tool for brainstorming, but they aren't the only one. You'll see how information can be organized in the mind map model, where each diagram reflects a part of the domain with a granularity that can be adapted to your needs.

2.1.4 Requirements in mind maps

In FDD methodology, brainstorming is the preferred method for gaining a deep understanding of things. For example, the architecture of the SCADA application presented at the end of this chapter is built with huge help from brainstorming. Moreover, you'll learn some design diagrams that force you to think associatively and, I believe, lead you to functional solutions for architecture problems. But first, you need to prepare for this journey, because you should understand your domain before you can construct the architecture. Welcome to the world of requirements analysis with mind maps.

Brainstorming is very unordered; that's why the results often have a free form. Mind maps are not so chaotic, and yet are not as rigid as user stories or use case diagrams. We want to keep our diagrams focused on the goal: a description of a small part of the domain that’s short but not so short as to miss important points that can affect the design.

So what is a mind map, and how do you create one? That's simple. A mind map is a tree-like diagram with one notion as the root and many branches that detail this notion to a necessary degree. To create a mind map, take a piece of blank paper and a pencil. In the center of the paper, write a word describing the domain you want to brainstorm about. Then circle it, or make it bold, or highlight it with a bright color — no matter how, make this notion significant. Then focus on this word, and write whatever comes to your mind. Connect these new details with your root. Now you have a small mind map, like in figure 2.7.

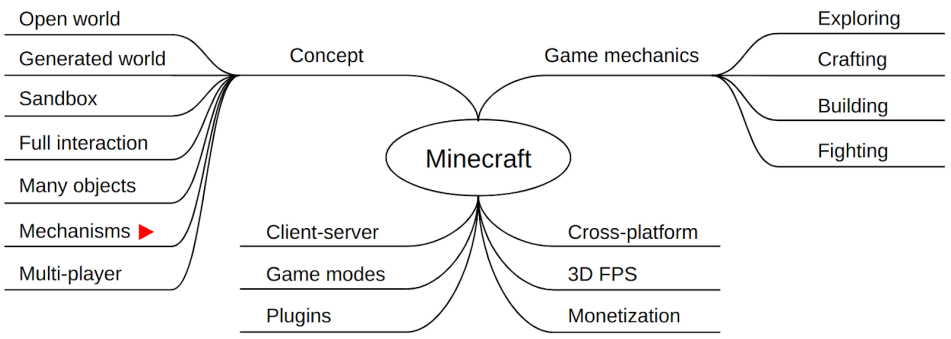


Figure 2.7 Level 1 mind map

The rest of the space on the paper is your limited resource — use it wisely. Repeat the procedure with new words. Do this until you get a full description of the first word. Then take a new piece of paper, and rearrange everything you've written, possibly with additions or deletions. Now you have a partial domain model in the form of a mind map. If it's not enough, feel free to create mind maps for other notions you haven't investigated yet.

This is simple, yet powerful enough to find hidden pitfalls. Mind maps are trees, but cross-relations aren't prohibited — connect elements by secondary links if you see that it's necessary and helpful. Unlike with UML diagrams, there are no rigorous rules for creating good and bad mind maps, and you aren't limited by someone else's vision of exactly how you should do it. Computer-powered mind maps can contain pictures, numbered and bulleted lists, side notes, links to external resources, and even embedded sounds and movies. Good mind map applications can manage a set of linked mind maps, so you can zoom in on a notion by opening the corresponding diagram via a link. Navigation over a mind map model is very similar to navigation over web pages, and this is the key to the convenience of this knowledge system. All the properties of the mind map make this tool very popular for domain analysis. It's also nice that in creating a mind map, we can connect functional ideas and idioms with the elements and notions.

The diagram in figure 2.7 doesn't contain that much information, only large themes. We need to go deeper. How deep? You decide for yourself. If you see that it's enough to list only key functionalities, you stop at level 1, the level of components and general requirements. Or, you might want to decompose these further, so you draw a mind map of the second level, the level of services and specific requirements. You can leave gaps and blanks because it's never bad to return and improve something in diagrams and therefore in your understanding of the domain. If something seems too broad, you have the third level to back it with user scenarios or use cases (see figures 2.8 and 2.9).

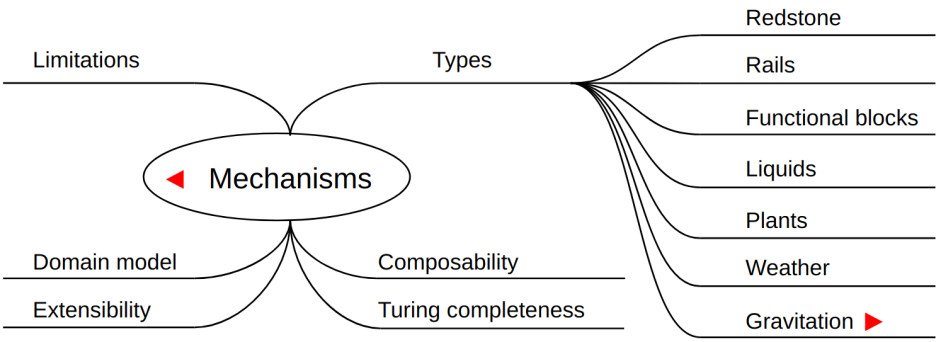


Figure 2.8 Level 2 mind map

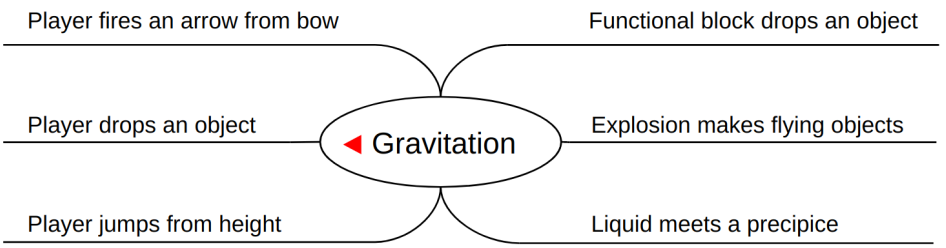


Figure 2.9 Level 3 mind map

Let's investigate the theme “Andromeda Control Software.” What do we know about it? It's SCADA software, it's a GUI application, it connects to hardware controllers, and it's for astronauts and engineers. We've also collected some functional and nonfunctional requirements already. Let's organize this knowledge from the perspective of development, not of the SCADA theory. Figure 2.10 illustrates this information in the level 1 mind map.

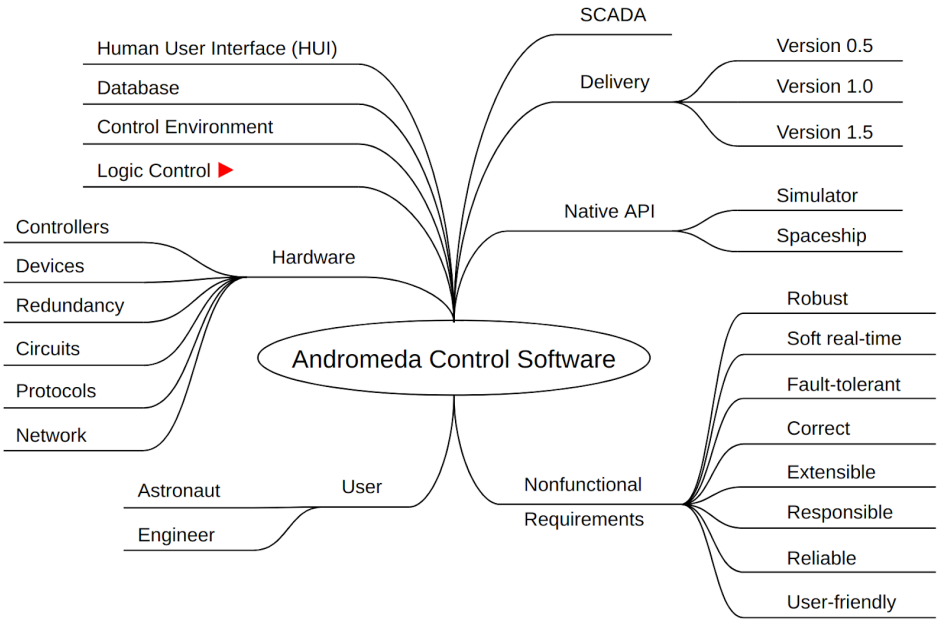


Figure 2.10 Andromeda Control Software mind map

The diagram in figure 2.10 invites us to descend one level down for the following components at least: Hardware, Logic Control, Database, and Human User Interface. We'll be analyzing the notion of “Logic Control” as the central notion of the next chapters. It’s marked with a triangular arrow.

What is Logic Control? In SCADA, it's a subsystem that evaluates the control of hardware: it sends commands, reads measurements, and calculates parameters. A programmable autonomous logic then utilizes the operational data to continuously correct spaceship orientation and movement. This is an important fact because one of the main responsibilities of engineers is to create this kind of logic during the shipbuilding process. That means we should provide either an interface to an external programming language or a custom DSL. Or both. We don't need to think about how we can do this, just point out the fact that we should. Why? Because we’re working on the requirements, and this corresponds to the question of “what?” We’ll deal with the question of “how?” (which is about software design) in the next sections.

Figure 2.11 shows the Logic Control diagram of level 2.

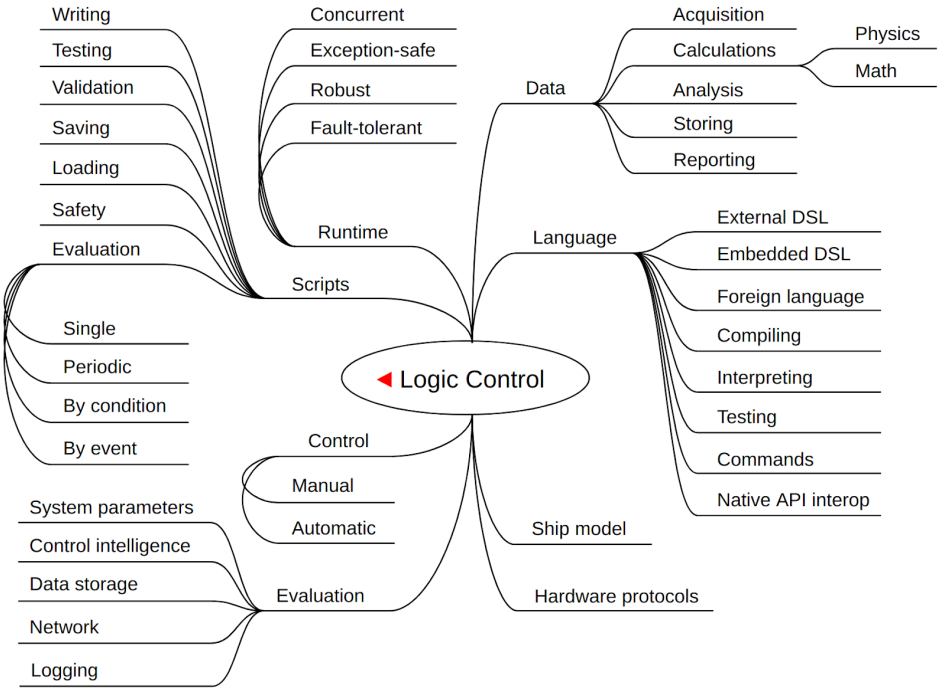


Figure 2.11 Logic Control mind map

The most interesting part of this is the “Language” branch. Every time we are going to implement a domain model (in this case, the domain model of Logic Control), we should create a specific language (an *embedded DSL*, or eDSL) as the formal representation of the subsystem. The design of such a language is highly situational, but often a type model made of algebraic and regular data types is enough for small modules or services that are stable and not extension points. As for big subsystems, you might want to model more complex language using higher-order functions, advanced type-level features, and complex algebraic types. Anyway, every code that reflects a subsystem can be considered as the formal language of that domain.

2.2 Architecture layering and modularization

Software design as a discipline has more terms and notions that we'll need to know to proceed with the design. Let's improve our vocabulary of terms.

We need a notion of a *layer*. In OOD, a layer is a set of classes that are grouped together by architectural purpose. We can also say a layer is a slice of the architecture that unites components having one behavioral idea. The second definition looks better than the first because it's paradigm-agnostic, so it's suitable for FDD too.

Another term is *module*. A module is a unit of decomposition that serves one concrete problem. Every module has an interface, which is the only way to interact with the hidden (encapsulated) module internals. In OOP, a module can be a class or a set of classes that are related to a particular task. In functional programming, a module is a set of functions, types, and abstractions that serves a particular problem. We decompose our application into small modules in order to reduce the code's complexity.

This section gives a short, theoretical overview of such big themes as architecture layers and modularization.

2.2.1 Architecture layers

What layers do we know of? There are several in OOD, and FDD suggests a few new ones. Let's revise the OOD layers:

- *Application layer.* This layer is concerned with application functioning and configuration — for example, command-line arguments, logging, saving, loading, environment initialization, and application settings. The application layer may also be responsible for threading and memory management.
- *Service layer.* In some literature, "service layer" is a synonym for "application layer." In FDD, it's a layer where all the service interfaces are available. This is just a slice that's less materialistic than other layers.
- *Persistence (data access) layer*. This layer provides abstractions for storing data in permanent storage; it includes object relational mappers (ORMs), data transfer objects (DTOs), data storage abstractions, and so on. In FDD, this layer may be pure or impure depending on the level of

abstraction you want in your application. Purity means this layer's services should have functional interfaces and that there should be declarative descriptions of data storage. The actual work with real data storage will be done in the interoperability layer or in the business logic layer.

- *Presentation (view) layer*. This layer provides mechanisms for building GUIs and other presentations of the application — for example, a console or web UI. It also handles input from the user (mouse or keyboard, for example), from graphic and audio subsystems, and possibly from the network.
- *Domain model layer*. This layer represents a domain model in data types and DSLs, and provides logic to work with. In FDD, it's a completely pure layer. Cases when it's impure should be considered harmful.
- *Interoperability layer*. This layer was born from the idea of providing one common bus that all the layers should use for communications without revealing the details. It often manages events (don't confuse these with OS messages), or it can be just impure procedures that call a native API. In FDD, this layer can be represented either by reactive code (FRP or reactive streams) or pipelined impure code (streams, pipelines, or compositions of impure, low-level functions, for example).
- *Business logic layer*. This layer should be considered as a superstructure over the interoperability layer and other layers. It consists of behavioral code that connects different parts of the application together. In FDD, the business logic layer is about scenarios and scripts. Business logic is aware of interfaces and DSLs that the subsystems provide, and it may be pure. Business logic scenarios can be translated to impure code that should work over the interoperability layer.

Figure 2.12 brings everything into one schematic view.

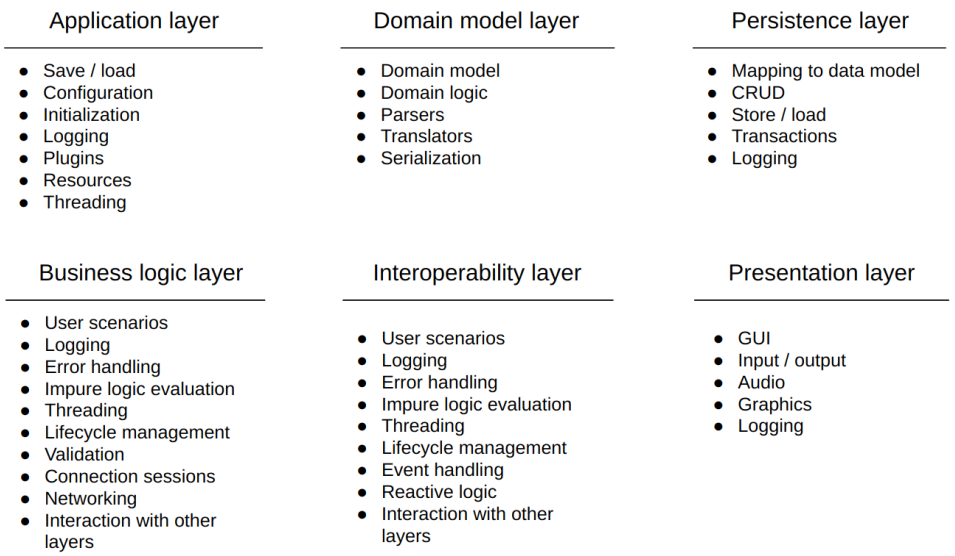


Figure 2.12 Layers: descriptions and responsibilities

At the same time, the idea of purity in FDD gives us two other layers:

- *Pure layer*. This layer contains pure and deterministic code that declares behavior but never evaluates it in an impure environment.
- *Impure layer*. In this layer, any impure actions are allowed. It has access to the outer world and also has means to evaluate logic declared in the pure layer. The impure layer should be as small as possible.

How can we shrink the impure layer? One option is to have pure lazy declarations of impure actions. For example, every Haskell function that has the return type `IO ()` is an impure action that can be stored for further evaluation when it's actually needed. It becomes impure at the moment of evaluation but not at the moment it’s composed with outer `IO` functions. We can compose our logic with lazily delayed impure actions, and the logic will be pure. Unfortunately, this doesn't protect us from mistakes in the logic itself.

Another option is to have a pure language that represents impure actions. You compose some code in this language: you declare what impure calls to do, but you don't evaluate them immediately. When you really need this code to be

evaluated, you should call a specialized interpreter that will map the elements of the language to the impure actions. We'll develop this idea in the next chapters; in particular, you'll know how to adopt the `Free` monad for making your own languages. For now consider the simple illustration in the following listing (listing 2.1).

Listing 2.1 Encoding of impure logic by pure language

```
data Language = ImpureAction1                    #A
              | ImpureAction2
              | ImpureAction3

type Interpreter = Language -> IO ()             #B

simpleInterpreter :: Interpreter                 #C
mockInterpreter :: Interpreter
whateverInterpreter :: Interpreter

simpleInterpreter ImpureAction1 = impureCall1
simpleInterpreter ImpureAction2 = impureCall2
simpleInterpreter ImpureAction3 = impureCall3

mockInterpreter ImpureAction1 = doNothing
mockInterpreter ImpureAction2 = doNothing
mockInterpreter ImpureAction3 = doNothing

-- Run language against the interpreter
run :: Language -> Interpreter -> IO ()
run script interpreter = interpreter script
```

#A Pure language encoding impure actions
#B Type alias for interpreters
#C Different interpreters

Here you see three interpreters that are evaluating the elements of language and acting differently. The `simpleInterpreter` function just maps pure language to real impure actions; the `mockInterpreter` function can be used in tests because it does nothing and so mocks the impure subsystem that the `Language` data type represents. The `run` function is the entry point to the subsystem. By taking the interpreter as the parameter, it doesn't care what the interpreter will actually do:

```
run simpleInterpreter boostersHeatingUp
run mockInterpreter boostersHeatingUp
run whateverInterpreter boostersHeatingUp
```

Note that the pair of types `Language` and `Interpreter` becomes a functional interface to the impure subsystem — you can either substitute the implementation by passing another interpreter or mock it by passing a mocking interpreter. In other words, this approach can be considered a kind of IoC in functional languages.

By introducing the `Language` DSL, we gained another level of abstraction over the logic. We can interpret it by an impure interpreter, but we can also translate it into another DSL that serves other goals, for example, adds a logging approach to every action. We then provide interpreters for this new language. The main benefit of such multiple translation is the ability to handle different problems on different levels. Figure 2.13 shows the intermediate pure languages before they're interpreted against an impure environment. The first language is the domain model, the second language has authorization possibilities, and the third language adds automatic log messages.

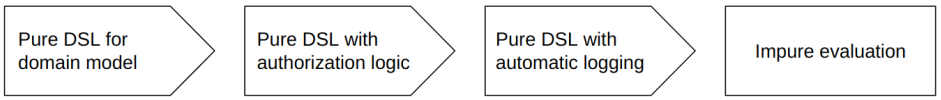


Figure 2.13 Translation from one language to another

The idea of translation (data transformation, in general) can be found in every functional idiom and pattern. In the next chapters, we'll see how it's useful for the design of DSLs. I've discussed some of the aspects of functional architecture layering, but as you can see, I've said nothing about implementation of the particular layers. We'll descend to this level of design in the next chapters.

2.2.2 Modularization of applications

First, we want our code to be loosely coupled. A general principle of design is that we need to divide logic and be happy with smaller parts. In FDD, we can achieve this by introducing functional interfaces that are primarily eDSLs. So, we can conclude that every subsystem communicates with the outer world

through a DSL with the help of interpreters. This gives us the notion of *subsystem boundaries*, illustrated in figure 2.14.

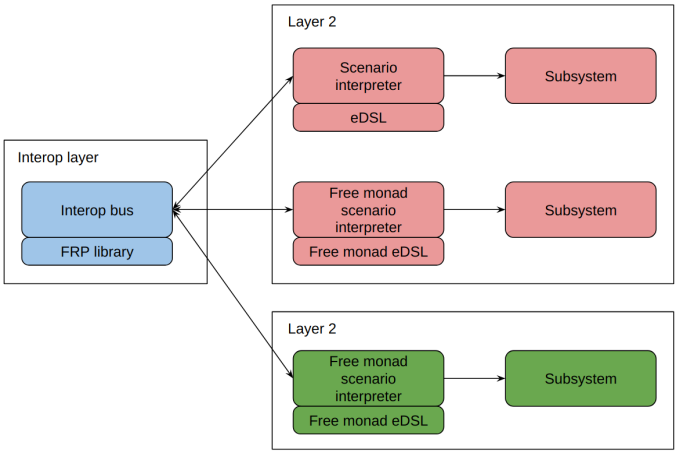


Figure 2.14 Subsystem boundaries

In the picture, you see the Interop layer and two types of eDSLs with interpreters: Free monad wrapped and not. A Free monad interface to a DSL doesn't change the DSL's behavior, but makes it possible to enhance scenarios using monadic libraries and syntactic sugar, like the `do` notation in Haskell and `for` comprehensions in Scala. If you have a DSL, the cost of wrapping it in a Free monad may be completely paid off by the benefits you get for free. Chapter 3 gives more information on this.

FDD (in its current state) suggests that you use this technique as a primary method of modularization in a functional way. In this book, we'll also learn other approaches to layering, like the monad stack. For now, this is enough to start modeling the architecture of the SCADA software for the Andromeda spaceship. The last part of this chapter offers you a lot of practice. It also presents some theoretical concepts that I've omitted so far.

2.3 Modeling the architecture

Have you ever met a developer who doesn't like diagrams? Coding is everything they have, and everything they want to have. The code they write is just mysterious: there's no graphical documentation on it; nevertheless, you can't deny that the code works. No one knows why. They don't like our children's games with colorful diagrams, they prefer pure code instead. The lack of color, though, makes them sad from time to time.

In this journey, we are about to draw diagrams — tens of diagrams. We can spend hours aligning rectangles and circles, selecting fonts and colors, and writing the accompanying text. Unfortunately, we might forget about the code itself while playing with our diagrams. Where the pure code programmer has a working prototype, we have only abstract figures, arrows, and pictures — nothing we can run and try. Even though designing with diagrams gives us great pleasure, we must stop ourselves at some point and learn from that pure code programmer. We should return to our main destiny — writing code. Thankfully, we can continue designing our application in high-level code too, along with our diagrams.

In functional programming, it's good to have architecture in two forms:

- *Topology of layers, modules, and subsystems*. This can be either described by diagrams or implemented in the project structure. The hints for how to define functionality for modules lie in our requirements model, and we will extract them from mind maps directly.
- *DSL interfaces for domain model, subsystems, and services*. A set of DSLs on top of a subsystem can be thought of as the functional interface of that system. We can use model-driven development to build DSLs for our domain model.

Imagine developers who know nothing about your architecture. Can they easily understand its underlying ideas? Will they have trouble using the DSLs? Can they break something? If they know your home address, do you expect them to visit you sometimes in the dead of night? How can you be sure the types you have elaborated are good enough? And what is a "good enough" architecture? Let's think. The architecture of the functional application is good if it is at least

- *Modular.* The architecture is separated into many parts (for example, subsystems, libraries, frameworks, or services), and every part has a small

set of strongly motivated responsibilities.

- *Simple*. All parts have clear and simple interfaces. You don't need much time to understand how a specific part behaves.
- *Consistent*. There is nothing contradictory in the architectural decisions.
- *Expressive*. The architecture can support all of the known requirements and many of those that will be discovered later.
- *Robust*. Breaking one separate part shouldn't lead to crashes of any other.
- *Extensible*. The architecture should allow you to add new behaviors easily if the domain requires it.

In this section, we'll create the architecture of the control software while learning how to convert our mind maps into architecture diagrams, modules, layers, data types, and DSLs. This is a methodology of iterative, top-down design that is highly suitable for functional architecture.

2.3.1 Defining architecture components

"If one does not know to which port one is sailing, no wind is favorable," said one thinker. This holds true for us: if we don't have a bird's-eye view of the software that we're going to create, how can we make architectural decisions?

The architecture of an application is made up of interacting components. A *component* is a big thing that can be implemented independently, and it's a good candidate to be a separate library or project. A component is not a layer, because one component can work on several layers, and one layer can unify many components interacting. A layer is a logical notion, while a component is a physical notion: it's a programmable unit with an interface. Also, a component is not a module, but it can be composed from modules. A component does one concrete thing. Traditional examples of components include the GUI, network, database, ORM, and logger. We're accustomed to understanding them in an object-oriented context, but in a functional context, they are the same. The difference is not in the components but in the interaction and implementation.

How would we organize the components of our software? A *necessity map*, introduced here, helps with that. It's very simple: it's just an informal diagram, a kind of concept map, illustrating a few big parts of the application. All the parts should be placed into rectangles and arranged in the most obvious way. Some of them may be connected together by arrows, while some may be independent,

and it's possible to show the part–whole relations. What components should our spaceship control software have? The requirements in our mind maps can tell us. Here's the list of components extracted from figures 2.10 and 2.11:

- Logic Control
- Hardware protocols
- Simulator or real spaceship
- Database
- Scripts
- GUI
- Hardware interface
- Network

Other components may reveal themselves during development, but this tends to be an exceptional situation.

All the components in the necessity map are necessary for the application — that's the reason for the diagram's name. You can see our necessity diagram in figure 2.15.

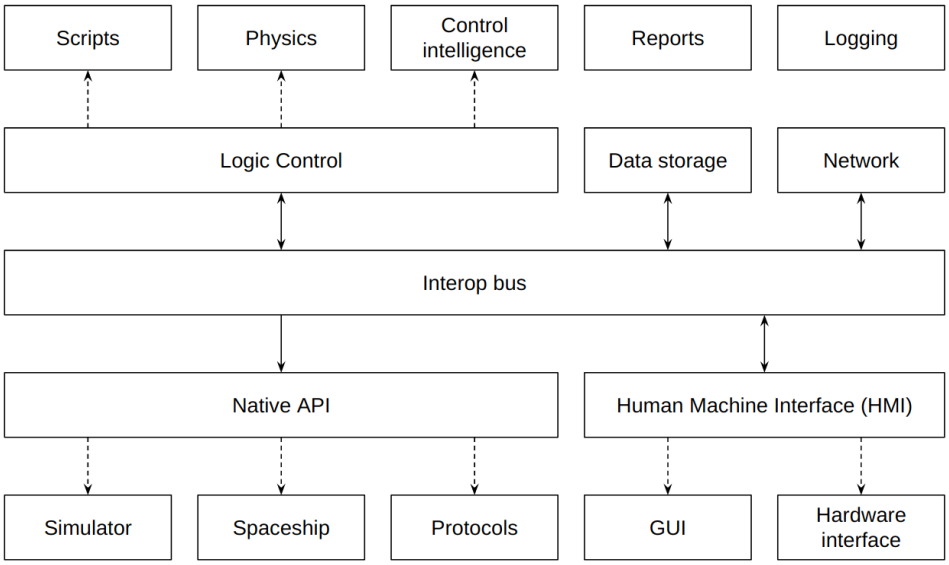


Figure 2.15 Necessity diagram of Andromeda Control Software

In the middle of the diagram, you see the *Interop Bus*. This was not mentioned before; where did it come from? Well, remember what we talked about in chapter 1. Simplicity and reducing accidental complexity are the main themes of that chapter. What would the diagram look like without this “rectangle of interaction”? Figure 2.16 shows it perfectly.

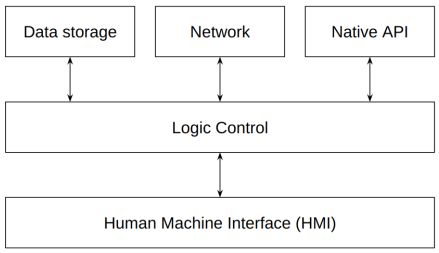


Figure 2.16 Necessity diagram without Interop Bus

Without the Interop Bus, the Logic Control component would operate with other components directly, and this would add a huge weight to the natural

responsibility of controlling the ship. It would mean the Logic Control component can't be made pure. The Interop Bus, as it’s called here, is an abstraction of interaction logic that’s intended to decouple components from each other. So, the Interop Bus is an architectural solution. Its implementations differ in philosophy, purpose, scale, and complexity. Most of them are built around a message-passing concept. Some of the implementations — event aggregator, event queue, or message brokers, for example — have come from the mainstream. In functional programming, we would like to use reactive streams and FRP, the next level of abstraction over the interaction logic.

2.3.2 Defining modules, subsystems, and relations

The next step of defining the architecture logically follows from the previous one. The necessity diagram stores information about what to do, and now it’s time to figure out how to do it. Again, we’ll use the brainstorming method to solve this problem. Our goal is to elaborate one or more elements diagrams as an intermediate step before we form the whole architecture. This is a concept map too, but, in this case, it’s much closer to a mind map because it doesn't dictate any structure. Just place your elements somewhere and continue thinking. The reasoning is the same: while brainstorming, we shouldn't waste our attention on formalities. But unlike with mind maps, in the elements diagram, there is no one central object. All elements are equal.

The necessity diagram will be a starting point. Take a block from it, and try to understand what else there is to say about this topic. Or you can take elements from mind maps — why not? All means are good here. If you think unstructured diagrams are ugly, that's normal! After this phase, we’ll be able to create a beautiful architecture diagram that will please your inner perfectionist.

As an example, I've taken the block “Logic Control” and expanded it as shown in figure 2.17.

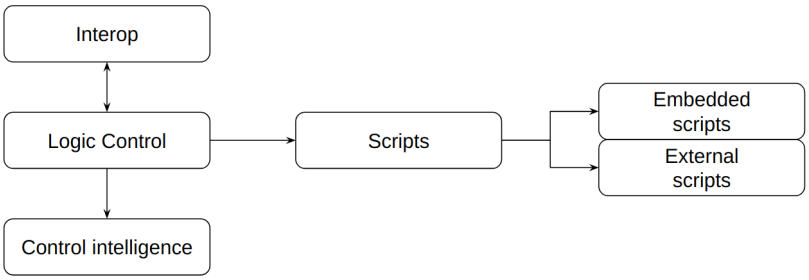


Figure 2.17 Simple elements diagram

What elements can be in the same diagram? There are no special limitations here. If you aren't sure whether a notion is suitable for an elements diagram, the answer is always "yes." You are free to refer to any concepts, domain terms, notions, libraries, layers, and objects, even if they have little chance of appearing in the software. There are no rules on how to connect the elements; you may even leave some elements unconnected. It's not important. Focus on the essence. A set of typical relations you may use is presented in, but not limited by, the following list:

- A is part of B
- A contains B
- A uses B
- A implements B
- A is B
- A is related to B
- A is made of B
- A interacts with B

For example, scripts use a scripting language; data storage can be implemented as a relational database; the interoperability layer works with the network, database, and GUI. To illustrate how these might look, consider the following two elements diagrams, both developed out of the diagram in figure 2.15. Figure 2.18 shows the Logic Control elements diagram, and figure 2.19 shows the diagram for Interoperability.

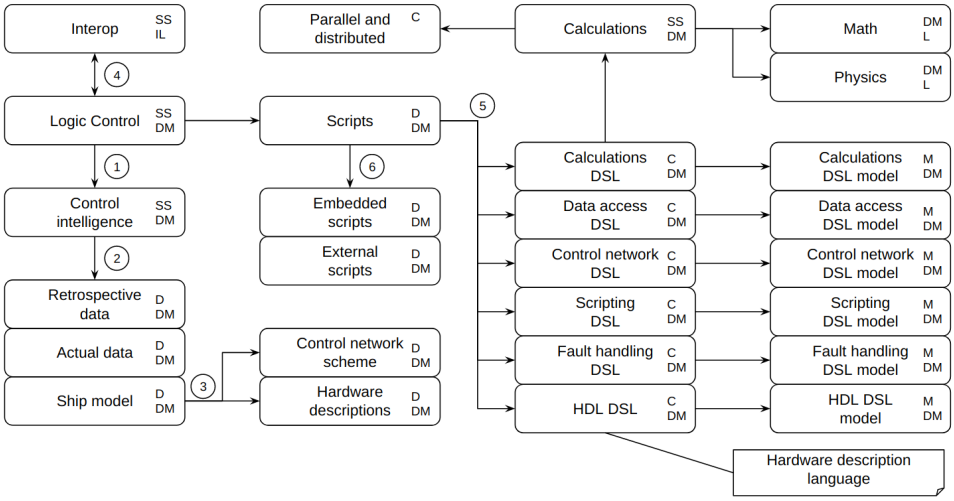


Figure 2.18 Elements diagram — Logic Control

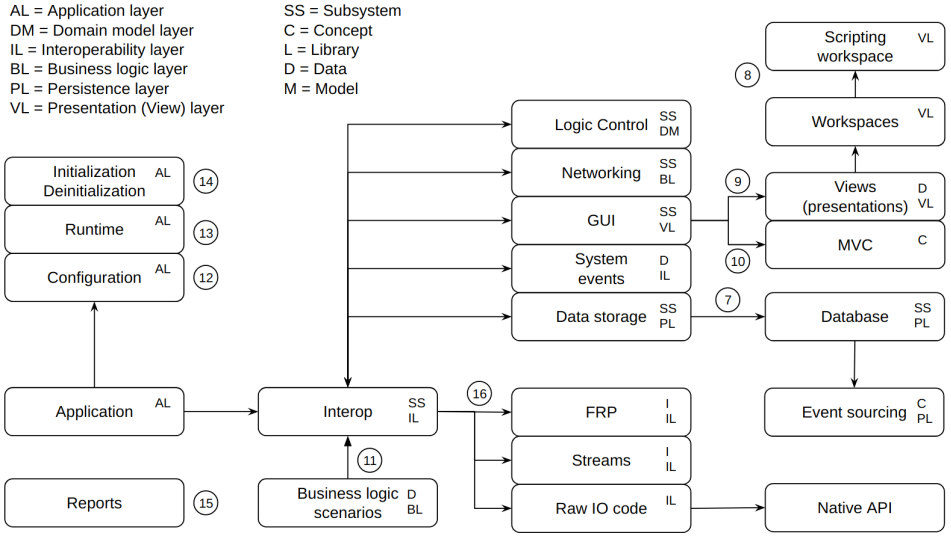


Figure 2.19 Elements diagram — Interoperability

Numbers placed on the diagrams describe the ideas behind the numbered elements. The following transcript can be considered a part of the requirements discovered during design:

1. Control Intelligence is the logic of automatic control of the ship: for example, correction of rotations, control of the life support system, and so on.
2. Control Intelligence uses Retrospective Data, Actual Data, and Ship Model.
3. Ship Model includes Control Network Scheme and Hardware Descriptors.
4. Logic Control interacts with other subsystems through Interop.
5. Scripts are written using many separate DSLs.
6. Scripts should be embedded and external.
7. Data Storage is an abstraction over some Database.
8. Scripting Workspace provides visual tools for writing Scripts.
9. GUI consists of many Views (Presentations).
10. GUI may be implemented with the MVC pattern.
11. Business Logic Scenarios compiles to Interop logic.
12. Application has Runtime.
13. Application manages Configuration.
14. Application does Initializations and Deinitializations of subsystems.
15. There should be reports. This theme has not been researched yet.
16. Interop includes these approaches:
    - FRP
    - Streams
    - *Raw IO* Code

Additionally, elements may have special labels. Every time you label an element, you ascertain whether the element is in the right place. The labels used in the preceding diagrams are

- Library (L) — element is an external library, or can be.
- Subsystem (SS) — element is a subsystem, module, or service.
- Concept (C) — element represents some general concept of software engineering.
- Data (D) — element represents some data.

- Model (M) — element is a model of some domain part.

If you want, you may use other labels. Since these diagrams are informal, you are the boss; feel free to modify them at your own discretion. There is only one thing that matters: adding elements to the diagram should give you new requirements and ideas for how to organize the application's entire architecture.

2.3.3 Defining architecture

The elements diagram doesn't reveal all secrets of the domain, certainly. But we rejected the idea of the waterfall development process and agreed that the iterative process is much better, and we can return to the elements diagram and improve it, or maybe create another one. After that, it becomes possible to elaborate the last architecture diagram suggested by FDD: the *architecture diagram*.

An architecture diagram is much more formal. It is a concept map with a tree-like structure; no cycles are allowed, and separate elements are prohibited. An architecture diagram should have all the important subsystems and the relations among them. It also describes concrete architectural decisions, like which libraries, interfaces, modules, and so on are used. An architecture diagram can be made from elements diagrams, but it's not necessary to take all the elements from there.

In the architecture diagram, you show a component and its implementation by a "burger" block. Figure 2.20 says that the Interoperability Bus should be implemented using the `reactive-banana` FRP library. It's also impure because its implementation has a `Monad.IO` block and consists of two parts: accessors to the database and GUI.

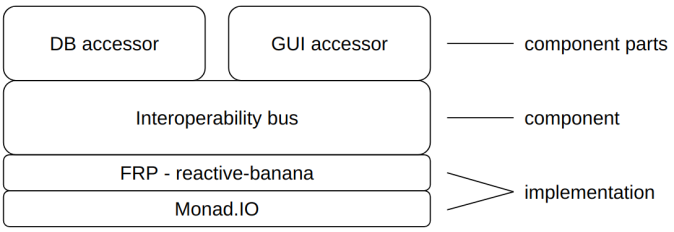


Figure 2.20 Component description block.

All components that your application should contain will be connected by relations “interacts with” (bidirectional) or “uses” (one-directional). You can associate layers with colors to make the diagram more expressive. See figure 2.21.

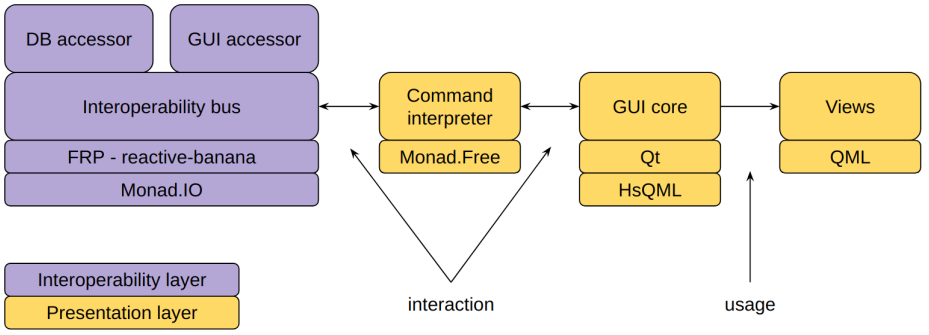


Figure 2.21 Relations between blocks

The whole architecture diagram can tell you many important facts about the architectural decisions, but it certainly has limited tools to describe all ideas of the application structure. So, consider this diagram as a graphical roadmap for the path you’re going to take. It's a bird's-eye view of your application, and you can change it whenever you need to.

Let me show you the architecture diagram for the Andromeda Control Software (see figure 2.22).

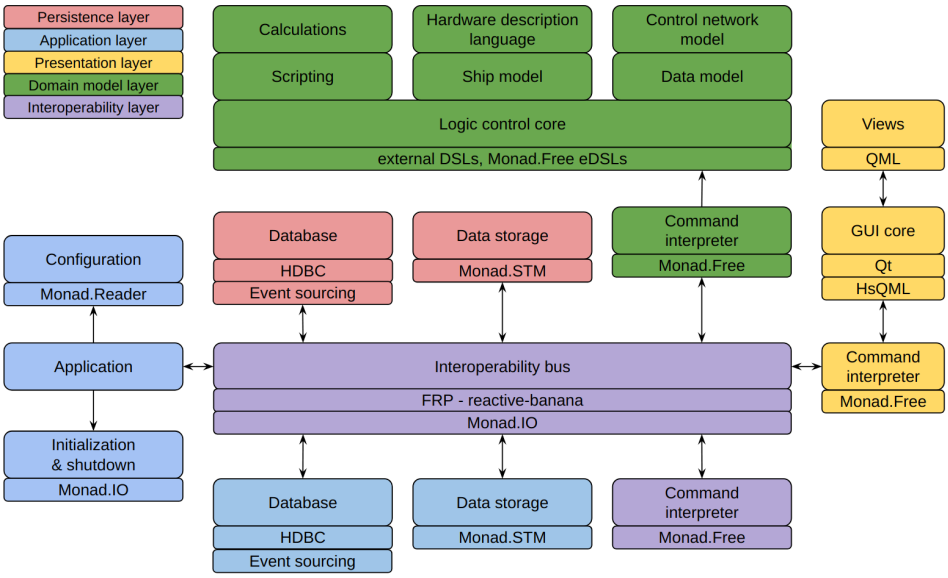


Figure 2.22 Architecture diagram for the Andromeda Control Software

That's it. I've shown you many pictures and written many words. We've designed the overall structure of the Andromeda Control Software. But don't think it's the final version of the architecture; of course it's not! The architecture diagram gives you a direction for further development that will help in the design of types and subsystem interfaces. You may wonder, why not create another type of diagram that includes type declarations and patterns like monad stacks, for example? Well, if you have plenty of time, do it, and if you succeed, please be so kind as to share how it's done! But from my point of view, it seems near impossible. Moreover, what are the benefits of such a "types diagram" to types in code?

I bet your programmer's addiction requires injection of code. Mine does. In the next chapters, we'll be coding a lot and trying to implement everything shown here.

2.4 Summary

This chapter is very important. It covers some of the fundamental questions of software architecture in functional programming — some, but not all of them. The theoretical foundations we've discussed are similar in OOD and FDD, but the implementation differs. First of all, architecture layers: as we now know, a layer unifies functionality that serves one architectural purpose. For example, the presentation layer unifies components that are related to the GUI; the layer that includes a domain model has the same name, and so on. We also denoted two meta-layers: pure and impure. In chapter 1, I described purity of domain and the advantages and disadvantages of being pure and impure. That directly maps to what we studied here. Just to recap: purity gives determinism, determinism simplifies reasoning about the code, and all together this reduces software complexity.

Another big thing we learned about is requirements collecting. Why should we care about this? Because without requirements, we can't be sure we're doing the right things. Furthermore, the requirements model can be converted into architecture and high-level code. We've learned four models for defining requirements: questions and answers, use case diagrams, user scenarios, and, finally, mind maps. If you browse the Internet, you'll find mind maps a popular tool for requirements analysis by brainstorming. We also saw how model-driven development can be used to create a DSL that describes a part of the domain. We constructed the DSL for control logic and two different interpreters. We showed how DSLs provide safety of evaluation, where nothing unpredictable can happen. The technique we used is extremely common in functional programming. The DSL is represented by an ADT, and the interpreter is just a pattern matcher that converts algebraic alternatives into real commands. Last but not least, we created a few diagrams and came to our application's architecture. The process of development in which we start from big parts and descend to small ones is known as top-down design. FDD suggests three types of diagrams: the necessity diagram, with big application components; the elements diagram, which is unstructured but highly informative; and the architecture diagram, which represents the architecture at large.

We also wrote high-level code to illustrate the design with types and functional interfaces. This is the next step of design we need to go through, so let's move on to the next chapter.

Part II

Domain Driven Design

Chapter 3

Subsystems and services

This chapter covers

- Module design and allocation of responsibilities
- Different approaches to functional interfaces
- How to use monads in general and Free monads in particular
- A little on monad transformers as architecture layers
- Design of functional services

Imagine that our galaxy, the Milky Way, had a solar system with a planet that was very similar to Earth. It had two oceans, three big continents, a warm equator, and cold poles. The people living there looked like us: two legs, two arms, torso, and head. They walked, ate, worked, and played just like us. One day, we decided to visit their planet. When our spaceship arrived, it landed on the first continent. We finally met our brothers in mind! And we realized they spoke in a language completely incomprehensible to us. Fortunately, our supercomputer was able to interpret this foreign language after analyzing thousands of books in their libraries. It successfully composed a word vocabulary for us, so we could speak freely. We were happy to get familiar with a great culture and the true masterpieces of literature our brothers had been collecting for many years. Then we went to the second continent. There lived happy people who communicated through a gestural language; they were great mime artists, but they had never known writing or verbal communication. Our supercomputer helped us again by analyzing their dances and rituals. After

interpreting their body language, it successfully composed a gesture vocabulary; we had a great time dancing and playing games. Finally, on the third continent, we met people who used colors to communicate instead of words or gestures. Our smart supercomputer was successful this time too; it analyzed and interpreted all the colorful signals around: signs, banners, news, and artwork. A vocabulary of colors helped us to see the beauty of this amazing planet.

This story is not so far removed from software development as it might first appear. People use different languages to communicate with each other, and the same principles hold if we replace humans with software. Every subsystem should interact with an outer world, and there's no other way to do this but to define a common language of data transformation and exchange. As in real life, the words of such a language mean different things depending on the context in which they occur. Subsystems that have received a "text" should interpret it and evaluate the tasks it codes. Many communication protocols have cryptic contents, so you can't understand what's happening just by reading the message: you need special tools to decrypt it into a human-readable form. But when we develop software, we invent many small DSLs with a certain purpose: all interactions between two subsystems, between application and user, and between application and computer must be understandable and clear. As we discussed earlier, this helps to keep the accidental complexity low.

In this chapter, we'll learn some design practices for functional subsystems, communication languages, and separation of responsibility that will help us to follow such principles as the Single Responsibility Principle (SRP), the Interface Segregation Principle (ISP), and the Liskov Substitution Principle (LSP).

3.1 Functional subsystems

Relax — you've just read one of those funny introductory texts that often precede a hard-to-learn theme. Soon, you'll be captured in a hurricane of meanings, a whirlwind of code, and a storm of weird concepts, but I'll try to provide some flashes of lightning to illuminate your path. We'll learn about mysterious monads and tricky monad transformers, and we'll discuss how our subsystems can be comprehended from a monadic point of view. We'll see why, we'll know how, we'll define when. For now, unwind and prepare your brain for this voyage.

The application architecture we created in chapter 2 is rather abstract and schematic. According to the top–bottom design process, we should next descend to the design of the components we specified in the architecture diagrams. Now we're interested in how to delineate the responsibilities of related subsystems. We'll study modules as the main tool for encapsulation in functional languages, but this is a syntactic approach that just helps you to organize your project and declare responsibilities. The most interesting design techniques come with functional idioms and patterns such as monads and monad transformers. These notions operate on the semantics of your code, and the behavior they introduce can be programmed even if your language doesn't support them in its syntax.

We'll continue to work with our sample, a SCADA-like system called the Andromeda Control Software. Next, we'll focus on the design of the Hardware subsystem.

3.1.1 Modules and libraries

In OOP, the notion of encapsulation plays a leading role. Along with the other three pillars of OOP — abstraction, polymorphism, and inheritance — encapsulation simplifies the interaction between classes by preventing casual access to the internal state. You can't corrupt an object's state in a way unexpected by the interface of a class. But a badly designed or inconsistent interface can be a source of bugs, thus invalidating internal state. We can roughly say that encapsulation of the internal state is another type of code contract, and once it's been violated, we encounter bugs and instability. A widespread example of contract violation is unsafe mutable code that's used in a concurrent environment. Without encapsulation, the state of an object is open for modification outside of the class; it's public, and there are no formal contracts in code that can prevent unwanted modifications. If there are public members of the class, then there are certain to be lazy programmers who will change them directly, even when you're completely sure they won't. You know the rest of the story: there will be blood, sweat, and tears when you have to redesign your code.

Encapsulation in functional programming is fairly safe. We usually operate by using pure functions that don't have an internal state that we could accidentally corrupt when complex logic is involved. Functional code tends to be divided into tiny pure functions, each addressing one small problem well, and the composition of these functions solves bigger problems. Still, if you have pure

composable functions, it neither prevents a contract violation nor gives a guaranteed stable interface for users. Consider this code:

```
data Value = BoolValue Bool
           | IntValue Int
           | FloatValue Float
           | StringValue String
```

Suppose you left the value constructors public. This means you don't establish any contracts that a code user must comply with. They are free to pattern match over the value constructors `BoolValue`, `IntValue`, `FloatValue`, and `StringValue`, and this operation is considered valid by design. But when you decide to rename some constructors or add a new one, you put their code out of action. Obviously, there should be a mechanism for encapsulating internal changes and hiding the actual data structure from external users. You do this in two steps: first, you introduce *smart constructors* for your ADT; second, you make your type *abstract* by placing it into a *module* and providing both smart constructors and useful functions. It will be better if your language can export the ADT without its value constructors, as this forces hiding the type on the language level. Many languages that have modules have an explicit exporting mechanism, or a way to simulate somehow. Let's deal.

> **DEFINITION** A *smart constructor* is a function that constructs a value of some type without revealing the actual nature of the type. Smart constructors may even transform arguments intellectually to construct complex values when needed.

> **DEFINITION** A *module* is the unit of encapsulation that unifies a set of related functions and also hides implementation details, providing a public interface.

> **TIP** Modules in Scala are called objects because they have a multipurpose use: objects in OOP and modules in functional programming. Modules in Haskell have syntax to control the export of certain things: types, functions, type classes, or ADTs with or without value constructors.

The next example introduces a widespread pattern of encapsulation — smart constructors for the ADT `Value`, which is still visible to the client code:

```
boolValue :: Bool -> Value
boolValue b = BoolValue b

stringValue :: String -> Value
stringValue s = StringValue s

intValue :: Int -> Value
intValue i = IntValue i

floatValue :: Float -> Value
floatValue f = FloatValue f
```

We've supported the `Value` type by using smart functions, but is it enough? The value constructors are still naked and visible to external code. While this is so, lazy programmers will certainly ignore smart constructors because they can. We might want something else, another mechanism of encapsulation to completely hide value constructors from prying eyes. We need to put this value into a module. This feature makes it possible to create *abstract data types* — data types you know how to operate with but whose internals are hidden. The interface of an abstract data type usually consists of pure functions for creating, transforming, and reading parts, combining with other data types, pretty printing, and other functionality. Let's place the `Value` type into a module `Andromeda.Common.Value` and define what to export:

```
module Andromeda.Common.Value -- hierarchical module name
  ( -- export list:
    Value,            -- type without value constructors
    boolValue,        -- smart constructors
    stringValue,
    intValue,
    floatValue
  ) where

data Value = BoolValue Bool
           | IntValue Int
           | FloatValue Float
           | StringValue String

boolValue :: Bool -> Value
boolValue b = BoolValue b

stringValue :: String -> Value
```

```
stringValue s = StringValue s

intValue :: Int -> Value
intValue i = IntValue i

floatValue :: Float -> Value
floatValue f = FloatValue f
```

If we wanted to export value constructors, we could write `Value(..)` in the export list, but we don't, so the export record is just `Value`.

An excellent example of an abstract type in Haskell is `Data.Map`. If you open the source code, you'll see that it's an algebraic data type with two constructors:

```
data Map k a = Bin Size k a (Map k a) (Map k a)
             | Tip
```

You can't pattern match over the constructors because the module `Data.Map` doesn't provide them in the interface. It gives you the `Map k a` type only. Also, the module `Data.Map` publishes many useful stateless and pure functions for the `Map` data type:

```
null :: Map k a -> Bool
size :: Map k a -> Int
lookup :: Ord k => k -> Map k a -> Maybe a
```

The base Haskell library provides several modules with almost the same operations over the `Map` data type. What's the difference? Just read the names of the modules, and you'll see: `Data.Map.Strict, Data.Map.Lazy,` and `Data.IntMap`. The first module is for a strict data type, the second is for a lazy one, and `Data.IntMap` is a special, efficient implementation of maps from integer keys to values. A set of functions related to one data type and organized in modules represents a library. Interfaces of these modules have the same functions with the same types. The function `null`, for instance, has type `Map k a -> Bool` regardless of whether the type `Map` is strict or lazy. This allows you to switch between implementations of the type `Map` without refactoring the code, just by importing a particular module you need:

```
-- Import strict or lazy Map:
import qualified Data.Map.Lazy as M
-- This code won't change:
abcMap = M.fromList [(1, 'a'), (2, 'b'), (3, 'c')]
```

So, modules represent another incarnation of ISP. It's easy to see that SRP is also applicable to modules: a module should have a single theme of content (well, responsibility).

From a design viewpoint, modules are the main tool for logical organization of the code. Modules can be arranged hierarchically, and there is no reason not to take advantage of them to reduce a project's complexity. You grow your project structure according to your taste to achieve better readability and maintainability — but design diagrams wouldn't be so useful if they didn't give any hints. By reading the elements diagrams in figures 2.13 and 2.14, and the architecture diagram in figure 2.15, we can elaborate something like what listing 3.1 presents. Here, directories are marked by backslashes, while modules are formatted in italics.

Listing 3.1 Structure of Andromeda Control Software

```
src\
    Andromeda                   #A
    Andromeda\                  #B
        Assets                  #C
        Calculations            #C
        Common                  #C
        Hardware                #C
        LogicControl            #C
        Simulator               #C
        Assets\                 #D
            Hardware\           #D
                Components      #D
        Calculations\           #E
            Math\               #E
            Physics\            #E
        Common\                 #E
            Value               #E
        Hardware\               #F
            HDL                 #F
            HNDL                #F
            Device              #F
```

```
        Runtime                 #F
    LogicControl\               #G
        Language                #G
    Simulator\                  #H
        Language                #H
        Runtime                 #H
```

#A The module that exports the whole project as a library
#B Root of the project – will contain all source code
#C Top modules – external code such as test suites should depend on these modules but not on the internal ones
#D Predefined data, default values, definitions, and scenarios
#E Separate libraries for math, physics, and other stuff – we can use an external library instead of these modules when it is reasonable
#F Hardware description language (HDL), hardware network description language (HNDL), runtime, and other data types used to define the hardware, network, and devices
#G Logic Control eDSL and other modules to define Logic Control operations
#H Simulator subsystem – in the future, this should be a separate project with its own structure and tests

One important thing: the top modules we place into the root directory (here you see six of them) shouldn't contain any logic but should re-export the logic from submodules, like so:

```
-- file src\Andromeda.hs:
module Andromeda (
    module Andromeda.Simulator,
    module Andromeda.Common
  ) where

import Andromeda.Simulator
import Andromeda.Common

-- file src\Andromeda\Simulator.hs:
module Andromeda.Simulator
    ( module Andromeda.Simulator.SimulationModel
    , module Andromeda.Simulator.Simulation
    , module Andromeda.Simulator.Actions
  ) where
```

```
import Andromeda.Simulator.SimulationModel
import Andromeda.Simulator.Simulation
import Andromeda.Simulator.Actions
```

For example, you should import the `Andromeda` module in tests, or, if you want more granularity, you may import any subsystem by referencing its top module:

```
module TestLogicControl where
import Andromeda.LogicControl

testCase = error “Test is not implemented.”
```

External code that depends on the module `Andromeda.LogicControl` will never be broken if the internals of the library change. With this approach, we decrease the coupling of modules and provide a kind of facade to our subsystem. So, all this is about separation of responsibilities. Functional programming has very nice techniques for it. But we’ve just gotten started. In the next section, we’ll discuss a truly brilliant aspect of functional programming.

3.1.2 Monads as subsystems

Finally, monads! A famous invention of category theorists that’s responsible for a true revolution in functional programming. What makes monads so powerful, so intriguing, and, to be honest, so weird a concept? The awesomeness of monads comes from their ability to combine different actions that you might think can't be combined at all. What kind of magic is it when you compose two parallel computations with the `Par` monad, and the code works in parallel without any problems? Why does the `List` monad behave like a generator, iterating over all possible combinations of items? And how do you recover from the shock the `State` monad plunges you into by demonstrating the state in pure stateless code?

I'm so sorry — we won't be discussing all these exciting questions. But don't be disappointed: we are going to discuss monads in the context of design. Monads, as chains of computation, will be required when we want to bind some actions in a sequence. Monadic languages are combinatorial, and all monadic functions are combinators.

> **DEFINITION** Wikipedia defines a *combinator* as “a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments.”

According to this definition, all pure functions are combinators, for example:

```
even :: Int -> Bool
map :: (a -> b) -> [a] -> [b]
(\x -> x + 1) :: Int -> Int
(+1) :: Int -> Int
```

In this book, we'll narrow the definition of a combinator to the following.

> **DEFINITION** A *combinator of combinatorial language* is a function that has a unified functional interface that allows us to compose combinators, resulting in a bigger combinator that again has this same interface. A combinator is a part of a combinatorial library related to a specific domain.

Given that we're talking about functional interfaces, you should know that there aren't so many of them. Functors, applicatives, monads, and arrows are the mechanisms we usually use to construct such a combinatorial language. It’s very beneficial, then, to represent an interface to a subsystem as a monadic combinatorial eDSL, the combinators of which can be bound together, resulting in another monadic combinator. Again, this brand-new combinator fits into the language perfectly because it has the same monadic type. This concept of the monadic eDSL has glorified the *parsec* library of combinatorial parsers; other examples include the `STM` monad and `Par` monad. Combinatorial monadic eDSLs have great complexity-beating power, and all functional developers should be fluent in designing using monads. But let's postpone this talk until I’ve introduced the concept of monads themselves.

> **TIP** We need to refresh our knowledge of monads to speak freely about many of them; if you feel a lack of understanding here, then you could probably try some of those monad tutorials you left for later. But the best way to learn monads, in my opinion, is to meet them face to face in code when trying to program something interesting — a game, for instance.

As you know, dealing with the impure world in a pure environment can be done with the so-called monad `IO` — this is what the story of monads starts from. The `IO` monad represents some glue for binding impure actions together. The following code demonstrates the `do` notation, which is really a syntactic sugar for monadic chains. You might read it as an imperative program at first, but after practice, you'll see that this understanding isn't really accurate: monads aren't imperative but are able to simulate this well:

```
askAndPrint :: IO ()
askAndPrint = do
  putStrLn "Type something:"
  line <- getLine
  putStrLn "You typed:"
  putStrLn line
```

Possible output:

```
> askAndPrint
Type something:
faddfaf
You typed:
faddfaf
```

This code has four `IO` sequential actions (`putStrLn “Type something:”`, `getLine`, `putStrLn “You typed:”`, and `putStrLn line`) chained into the one higher-level `IO` action `askAndPrint`. Every monadic action is a function with a monadic return type. Here, the return type `IO a` defines that all the functions in a `do` block should be in the `IO` monad (should return this type), as should the whole monadic computation composed of them. The last instruction in the `do` block defines the computation output. The function `getLine` has type `IO String`: we bind a typed string with the `line` variable, so we can print it later with the `putStrLn` function. The latter is interesting: I said every function in the `IO` monad returns a value of this type. What is the type of the function `putStrLn`? Look:

```
putStrLn :: String -> IO ()
putStrLn "Type something:" :: IO ()
askAndPrint :: IO ()
getLine :: IO String
```

Although we could do it, we don't bind the result from the function `putStrLn` with any variable:

```
res <- putStrLn "Type something:"
```

This is because we discard the only value of the unit type `()`, which is always the same: `()`. In the preceding code, we aren't interested in this boring value. It can be discarded explicitly, but this looks strange and is unnecessary:

```
discard = do
  _ <- putStrLn "Discard return value explicitly."
  putStrLn "Discard return value implicitly"
```

Then, the function `askAndPrint` has the return type `IO ()` — this makes it a full-fledged member of `IO` monad computations. We can use it similarly to the standard functions `putStrLn` and `readLine`:

```
askAndPrintTwice :: IO ()
askAndPrintTwice = do
  putStrLn "1st try:"
  askAndPrint
  putStrLn "2nd try:"
  askAndPrint
```

Figure 3.1 may help you understand the monadic sequencing.

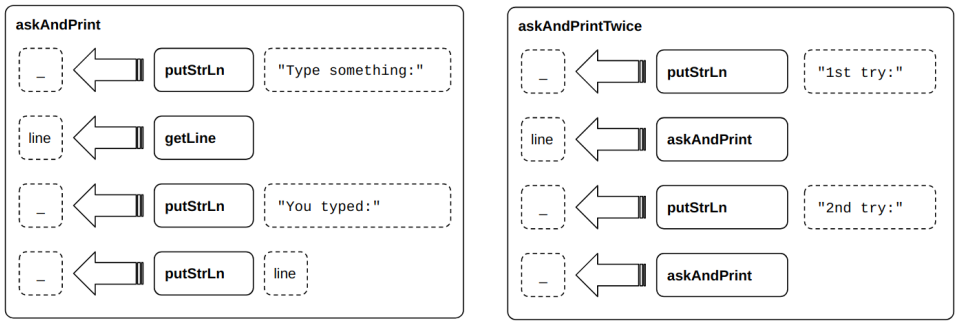


Figure 3.1 Monadic sequencing of actions in the IO monad

We can also force functions to return any value we want. In this case, we should use the `return` standard function as a wrapper of the pure value into a monadic value, as `getLine` does (see listing 3.2).

Listing 3.2 Monadic code in the IO monad

```
ask :: IO ()
ask = putStrLn "Print something:"

quote :: String -> String                #A
quote line = "'" ++ line ++ "'"

askAndQuote :: IO String
askAndQuote = do
  _ <- ask                               #B
  line <- getLine
  return (quote line)

askQuoteAndPrint :: IO ()
askQuoteAndPrint = do
  val <- askAndQuote                     #C
  putStrLn val
```

#A Pure function over the String value
#B Explicitly discarded value ()
#C Binding of quoted line with the val variable;
val variable has String type

This code produces the following output, if you enter `abcd`:

```
> askQuoteAndPrint
Print something:
abcd
'abcd'
```

In Haskell, the `do` notation opens a monadic chain of computations. In Scala, the analogue of the `do` notation is a `for` comprehension. Due to general impurity, Scala doesn't need the `IO` monad, but the `scalaz` library has the `IO` monad as well as many others. Also, some other libraries have emerged from the community recently, and their purpose is very similar to the IO monad. For

example, ZIO. This is a whole ecosystem built on top of an idea of wrapping around the IO effect.

This is what the near side of the “Moonad” looks like. You see just a chain of separate monadic functions, and it's hard to see any evidence that a far side — an undercover mechanism that makes the “Moonad” magic work — is hidden there. It can be mysterious, but every two neighboring monadic functions in a `do` block are tightly bound together even if you don't see any special syntax between them. Different monads implement this binding in their own way, but the general scheme is common: function `X` returns the result, and the monad makes its own transformations of the result and feeds function `Y`. All this stuff happens in the background, behind the syntactic sugar of `do` notations or `for` comprehensions. Figure 3.2 illustrates two sides of some monad.

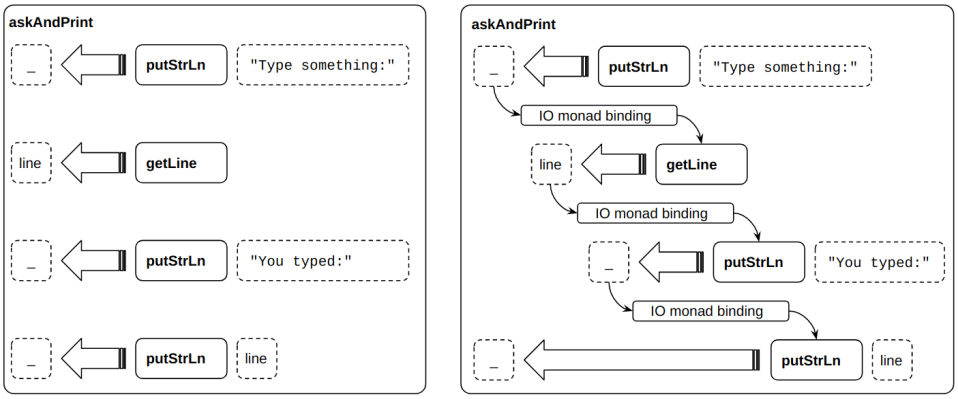


Figure 3.2 Two sides of a monad

If you want to create a monad, all you need to do is reinvent your own binding mechanism. You can steal the `bind` function from the `State` monad and say it’s yours, but with that, you also steal the idea of the `State` monad it carries. So, your monad becomes the `State` monad. If you cut off parts of the stolen `bind`, you'll probably get a kind of either `Reader` or `Writer` monad. Experiments with the stolen `bind` might end sadly for you. By stealing and cutting, it's more likely you'll break the monadic laws and your “monad” will behave unpredictably. Being a mathematical conception, monads follow several rules (for example, associativity). Think twice before doing this. If you really want to create a monad, you have to invent your own binding mechanism with

an original idea or composition that obeys the monadic laws. You should also learn applicatives and functors because every monad is an applicative functor and every applicative functor is a functor, all having their own properties and laws. Creating monads requires a lot of scientific work. Maybe it's better to learn existing monads? There are plenty of them already invented for you! They are described in table 3.1.

Table 3.1 Important monads

Monad	**Description**
IO	Used for impure computations. Any side effect is allowed: direct access to memory, mutable variables, the network, native OS calls, threads, objects of synchronization, bindings to foreign language libraries, and so on.
State	Used for emulation of stateful computations. Behaves like there is mutable state, but in reality, it's just an argument-passed state. Also, the lens conception may greatly work inside the `State` monad, which really makes sense when dealing with a very complex data structure such as the state.
Reader	Provides a context that can be read during computations. Any information useful for the computation can be placed into the context rather than passed in many arguments. You “poll” data from the context when you need it.
Writer	Allows you to have a write-only context to which you push monadic values (that can be added). Often used to abstract logging or debug printing.
RWS	`Reader`, `Writer` and `State` monad all at once. Useful when you need immutable environment data to hold useful information, stateful computations for operational needs, and write-only context for pushing values into it.

Free	Wraps a specially formed algebra into a monadic form, allowing you to create a monadic interface to a subsystem. Abstracts an underlying algebra, which helps to create more safe and convenient DSLs. One of the possible applications considered to be a functional analogue of OOP interfaces.
Either	Used as an error-handling monad, it can split computation into success and failure scenarios, where the error path is additionally described by an error type you supply to the monad. Due to its monadic nature, the code will stay brief and manageable.
Maybe	Can be used as an error-handling monad without error information attached. That is, the `Maybe` monad represents a weaker form of the `Either` monad. Also, this monad can be used to generalize computations where the absent result is a valid case.
Par	Monad for data parallelism. With the primitives it has, you define an oriented graph of data transformations. When a Par monad code is evaluating, independent branches of the graph may be parallelized automatically.
ST	Strict state threads. Naturally, a real mutable state that is embedded locally into your pure function. You can create and delete variables, arrays, access their internals, and change values. The state is local, - meaning, it won't be seen from the outside of the ST monad block. The state is mutable, but it doesn't affect the external immutable world. A stateful code in the ST monad can be much more efficient than the analogue in the State monad.
STM	Software Transactional Memory monad. Provides a safely concurrent state with transactional updates. Due to monadic composability, you may build a complex mutable computation over your data structure and then run it safely in a concurrent environment.

We'll see many applications of the `State` monad in this book, so it would be nice to consider an example of it. Let's say we want to calculate a factorial. The shortest solution in Haskell follows the factorial definition: factorial of natural number n (n > 0) is the product of all numbers in the sequence [1..n]. In Haskell, it's pretty straightforward (we'll ignore problems with a possible negative input):

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

Let's say we investigate how good imperative programming is in pure functional code. We want the factorial to be calculated by the classic algorithm with a counter and accumulation of partial products. It's not that beautiful if we try an argument-passing style to keep these partial products:

```
factorial :: Integer -> Integer
factorial n = let (fact, _) = go (1, n)
              in fact
  where
    go (part, 0)       = (part, 0)
    go (part, counter) = go (part * counter, counter - 1)

-- running:
> factorial 10
3628800
```

While doing exactly the same argument-passing work behind the scenes, the `State` monad frees you from managing the state manually. It gives you two primitives for working with the state indirectly: the `get` function that extracts a value from the state context and the `put` function that puts a new value into the state context. This monad's type is defined in the `Control.Monad.State` library:

```
State s a
```

It has two type arguments: `s` to hold the state and `a` for the return value. We'll specialize these type arguments by the `Integer` type and the unit type `()`, respectively. The integer state will represent the accumulator of the factorial calculation. See the following code:

```
factorialStateful :: Integer -> State Integer ()
factorialStateful 0 = return ()
factorialStateful n = do
  part <- get
  put (part * n)
  factorialStateful (n - 1)
```

The code became less wordy, but now it's a monadic function, and you should run the `State` monad with one of three possible functions:

```
> runState (factorialStateful 10) 1
((),3628800)
> execState (factorialStateful 10) 1
3628800
> evalState (factorialStateful 10) 1
()
```

All three are pure, all three are running the stateful computation (`factorialStateful 10`), all three are taking the initial state "1", and all three return either the state (`3628800`) or the value from the monadic function (`()`), or both paired (`((), 3628800)`).

The last thing that we should learn is true for all monads. We said the main reason to state our code by monads is to make it more composable. Every monadic function is a good combinator that perfectly matches with other combinators of the same monad. But what exactly does this mean? In general, two monadic functions combined share the effect the monad expresses. If we take the `IO` monad, all nested monadic functions may do impure things. No matter how deep we nest the functions, they all work with the same effect:

```
ask :: IO ()
ask = putStrLn "Print something:"

askAndGetLine :: IO String
askAndGetLine = do
  ask
  line <- getLine
  return line

main :: IO ()
main = do
  line <- askAndGetLine
```

```
  putStrLn ("You printed: " ++ line)
```

If we take the `State` monad, the state is shared between all the functions in this calculation tree:

```
multiply :: Integer -> State Integer ()
multiply n = do
  value <- get
  put (value * n)

factorialStateful :: Integer -> State Integer ()
factorialStateful 0 = return ()
factorialStateful n = do
  multiply n
  factorialStateful (n - 1)

printFactorial :: Integer -> IO ()
printFactorial n = do
  print (execState (factorialStateful n) 1)
```

Both `multiply` and `factorialStateful` functions have the same state in the `State's` context. You might guess what will share two monadic functions in the `Reader` and `Writer` monads, but what about other monads? Every monad has its own meaning of composition of two monadic functions. Technically, the nesting of monadic functions that you see in the previous examples is no different from the plain chain of computations; breaking the code to the named functions makes it easy to reuse these parts in different computations. But it's important to understand that effects can't be shared between independent computations. In the following code, you see two different computations in the `State` monad, each of which has its own state value:

```
squareStateful :: Integer -> State Integer ()
squareStateful n = do
  put n
  multiply n

printValues :: IO ()
printValues = do
  print (execState (factorialStateful 10) 1)
  print (execState (squareStateful 5) 0)
```

This program prints:

```
> printValues
3628800
25
```

In fact, every monadic function is just a declaration of what to do with the effect. The multiply function is a declaration that determines to take a value from the state, multiply it to some n passed, and give the result back to the state. It doesn't know whether a value in the state will be a partial product of the factorial or something else. It just knows what to do when the whole monadic computation is run.

That's all I wanted to tell you about the origins of monads; this section is too short to contain more information. We'll now discuss an interesting theme: monads as subsystems.

Every monadic computation is the function returning a monadic type `m a`, where `m` is a type constructor of the monad, and the type variable `a` generalizes a type of value to return. Almost all monads work with some metadata that you should specify in the monad's type. The `State` monad keeps a state (`State Parameters a`), the `Writer` monad holds a write-only collection of values (`Writer [Event] a`), and the `Reader` monad holds the environment for calculation (`Reader Environment a`). In listing 3.3, you can see definitions of monadic functions. Note the types of the functions.

Listing 3.3 Definitions of monadic functions

```
------------ Monad State --------------

type Parameters = (Integer, Integer)
type FactorialStateMonad a = State Parameters a

calcFactorial :: FactorialStateMonad Integer

------------ Monad Reader -------------

data Environment = Env
  { sqlDefinition :: SqlDefinition
  , optimizeQuery :: Bool
```

```
  , prettyPrint   :: Bool
  , useJoins      :: Bool
  }

type SqlQueryGenMonad a = Reader Environment a

sqlScriptGen :: SqlQueryGenMonad String

------------ Monad Writer -------------

data Event = NewValue Int Value
           | ModifyValue Int Value Value
           | DeleteValue Int
type EventSourcingMonad a = Writer [Event] a

valuesToString :: [Value] -> EventSourcingMonad String
```

The common pattern of all monads is “running.” To start calculations in the monad and get the final result, we evaluate a function `runX`, passing to it our calculations and possibly additional arguments. The `runX` function is specific for each monad: `runState`, `runWriter`, `runReader`, and so on. The code in listing 3.4 shows how we run the monadic functions defined in listing 3.3.

Listing 3.4 Running of monadic functions

```
sqlDef = SqlDef (Select "field")
                (From   "table")
                (Where  "id > 10")

sqlGenSettings = Env sqlDef True True True

values = [ BoolValue True
         , FloatValue 10.3
         , FloatValue 522.643]

valuesToStringCalc :: EventSourcingMonad String
valuesToStringCalc = valuesToString values

calculateStuff :: String                                    #A
calculateStuff = let
    (fact, _) = runState  calcFactorial (10, 1)
```

```
    sql       = runReader sqlScriptGen sqlGenSettings
    (s, es)   = runWriter valuesToStringCalc

    in "\nfact: " ++ show fact ++
       "\nsql: " ++ show sql ++
       "\nvalues string: " ++ s ++
       "\nevents: " ++ show es

#A Pure function that runs different monads
   and gets results back
```

The result of evaluating the `calculateStuff` function may look like the following:

```
> putStrLn calculateStuff

fact: 3628800
sql: "<<optimized sql>>"
values string: True 10.3 522.643
events: [ NewValue 1 (BoolValue True)
        , NewValue 2 (FloatValue 10.3)
        , NewValue 3 (FloatValue 522.643)]
```

Here, you see the monadic functions (`calcFactorialStateful`, `sqlScriptGen`, and `valuesToStringCalc`), starting values for the `State` and `Reader` monads (`initialPi` and `sqlGenSettings`, respectively), and the `runX` library functions (`runState`, `runReader`, and `runWriter`). Note how we run all three monadic computations from the pure `calculateStuff`. All these calculations are pure, even though they are monadic. We can conclude that all monads are pure too.

> **NOTE** You may have heard that program state is inevitably impure because of its mutability. This isn't true. The preceding code proves the contrary. Strictly speaking, the `State` monad doesn't mutate any variables during evaluation of the monadic chain, so it isn't a real imperative state that everybody is aware of. The `State` monad just imitates the modification of variables (passing state as an argument between monadic actions in the background), and often it's enough to do very complex imperative-looking calculations. There are some controversial aspects of `IO` monad purity that are nothing special or

> language-hacked in Haskell but have to do with the incarnation of the `State` monad, but we won't touch on this polemical theme in the book.

You can treat a monad as a subsystem with some additional effect provided. The “run” function then becomes an entry point for that subsystem. You can configure your effect; for example, setting environment data for the computation in the `Reader` monad. You also expect that the evaluation of the monadic computation will return a result. The monadic calculation itself can be infinitely complex and heavy; it can operate with megabytes of data, but in the end, you want some artifact to be calculated. As an example, consider our Logic Control subsystem. It operates with hardware sensors, it reads and writes data from and to Data Storage, it does math and physics computing — in fact, this is the most important subsystem of the application. But we can't just implement it, because we don’t want the code to be a mess. You see what the effects should be in every part of your big system and associate these effects with a certain monad. You then divide the subsystem into smaller ones according to the SRP:

- Data Storage and the `STM` monad (because data storage should be modified concurrently — this is what the `STM` monad is responsible for)
- Logic Control and the Free eDSL monad (because Logic Control has many internal languages, and we'll better make them interpretable and monadic — this is the responsibility of the `Free` monad)
- Hardware description language (HDL) and, probably, the two monads `Reader` and `Free` (the `Reader` monad may be used to hold operational information about the current device we’re composing).
- Native API and the `IO` monad (because we deal with an impure world)
- Logging and the `Writer` monad (because we push log messages into the write-only context)

I mentioned the `IO` monad because it’s no different from the others. Moreover, the `IO` monad has the same property of being a subsystem, but, in this case, we’re talking about a program-wide subsystem. In Haskell, the `main` function has type `IO ()`, and is the entry point to your application’s impure subsystem. It even has an environmental configuration — the command-line arguments your application started with.

In imperative programming, we call subsystems from other subsystems on a regular basis. Monads are even better, for several reasons. First, we do this already, starting from the impure IO subsystem and digging deeper and deeper into the pure monadic subsystems. Second, monadic computations are composable. You often can't compose an imperative subsystem, but you can compose the monadic functions of one monad, as we did in listings 3.2 and 3.3. Third, you can *mix* several subsystems (monads) and get all the effects together, while they remain composable and neat. This operation doesn't have any direct analogues in the imperative world. You may argue that multiple inheritance exists, but your class will be gorging more and more with every new interface that's implemented — nevermind the problems the inheritance can cause.

So, how does mixing subsystems (monads) work? Let's find out.

3.1.3 Layering subsystems with the monad stack

A common case when mixing monads is when we need impure computations (`IO`) here and now, but we also want to use another monadic effect at the same time. For a simple start, let's say our subsystem should be impure (the `IO` monad) and should work with state (the `State` monad), yet it shouldn't lose composability. The following code shows two functions in these monads. The `factorialStateful` function calculates factorial. It obtains the current value from the state context, multiplies it by the `counter`, and puts the result back. The `printFactorial` function runs this stateful computation and prints the result of the factorial of 10, which is 3628800.

```
factorialStateful :: Integer -> State Integer ()
factorialStateful 0 = return ()
factorialStateful n = do
  part <- get
  put (part * n)
  factorialStateful (n - 1)

printFactorial :: Integer -> IO ()
printFactorial n = do
  let fact = execState (factorialStateful 10) 1
  print fact
```

What if we want to print all the partial products too? We can collect them along with the result and then print:

```
factorialProducts :: Integer -> State [Integer] ()
factorialProducts 0 = return ()
factorialProducts counter = do
  (prevPart:parts) <- get
  let nextPart = prevPart * counter
  put (nextPart : prevPart : parts)
  factorialProducts (counter - 1)

printFactorialProducts :: Integer -> IO ()
printFactorialProducts n = do
  let products = execState (factorialProducts n) [1]
  print products

-- running:
> printFactorialProducts 10
[3628800,3628800,1814400,604800,151200,30240,5040,720,90,10,1]
```

It does what we want, and you can stop here if you're pleased. I believe this code isn't as simple as it can be. All this ugly list work ... ugh. Why not just print every immediate part at the moment when it's calculated, like so:

```
factorialStateful :: Integer -> State Integer ()
factorialStateful 0 = return ()
factorialStateful n = do
  part <- get
  print part         -- this won't compile!
  put (part * n)
  factorialStateful (n - 1)
```

Unfortunately, this code won't compile because the compiler is pretty sure the function `factorialStateful` is pure and can't do printing to the console. The compiler knows this because of the type `State Integer ()`, which doesn't deal with the `IO` monad.

> **NOTE** I said that Haskell is pure; that's why the compiler will raise a compilation error on the preceding code. In Scala, side effects are imperatively allowed, so the analogue code will work. But we know imperative thinking and bare freedom can easily break our functional

> design, and we have to uphold the pure functional view of monads. For this reason, let's agree to avoid Scala's impurity and learn how to use the `IO` monad, even if it's not strictly necessary.

We need to combine the `IO` and `State` monads into a third "StateIO" monad somehow. Smart mathematicians have found the appropriate concept for us: it's called *monad transformers*. When you have a monad transformer for a monad, you can combine it with any other monad. The result will have all the properties of the underlying monads, and it will be a monad too. For instance, the `StateT` transformer can be combined with `IO` this way ("T" in "StateT" stands for "transformer"):

```
-- StateT transformer is defined here
import Control.Monad.Trans.State

-- Monadic type with State and IO effects mixed:
type StateIO state returnval = StateT state IO returnval
```

Now it's possible to print values inside the `StateIO` monad. We should introduce one more little thing for this, namely, the `lift` function. It takes a function from the underlying monad (here it's the `print` function that works in the `IO` monad) and adapts this function to run inside the mixed monad. Let's rewrite our example:

```
type StateIO state returnval = StateT state IO returnval

factorialStateful :: Integer -> StateIO Integer ()
factorialStateful 0 = return ()
factorialStateful n = do
  part <- get
  lift (print part)        -- this will now compile as desired
  put (part * n)
  factorialStateful (n - 1)
```

The logical consequence we can see from looking at the type of the `factorialStateful` function is that it's no longer pure. The functions `runState`, `evalState`, and `execState` are pure and can't be used to run this new monad computation. The `StateT` transformer is accompanied by the special version of these functions: `runStateT`, `evalStateT`, and `execStateT`. We can run them inside the `IO` monad like a regular `IO`

action; for example, the `evalStateT` that evaluates the computation and returns a value of this computation:

```
printFactorial :: Integer -> IO ()
printFactorial n = evalStateT (factorialStateful n) 1
```

Result:

```
> printFactorial 10
1
10
90
720
5040
30240
151200
604800
1814400
3628800
```

In Haskell libraries and in the `scalaz` library, there are transformers for almost all standard monads. We don't have to reinvent the wheel.

> **NOTE** While imperative programmers may grab their heads and scream after seeing our abstractions (totally useless for them, probably), I maintain that composability and separation of effects gives us a truly powerful weapon against complexity. We'll learn how to use monad transformers, but we won't open Pandora's box and try to figure out how the mechanism works. You can do it yourself, and you'll see that the mathematicians were extremely smart.

By now, you probably have a rough idea of why we want to deal with the `State` and the `IO` monads in the pure functional code. I hope the purpose of the `StateT` monad became a little clearer. If not, we'll try to come at it from the other side. Remember that we associated monads with subsystems? I said that when we see that a particular subsystem should have some effect (impurity, state, parallelism, and so on), it can be implemented inside the monad that simulates this effect. So, if you think about monads as subsystems, you may conclude that it's sometimes beneficial to have a subsystem that can do several effects in one computation. Perhaps your subsystem should be able to read clock

time (the `IO` monad) to benchmark parallel computations (the `Par` monad). Or your subsystem works with a database (the `IO` monad) and should be fault-tolerant, so you should construct a convenient API to handle errors (the `Either` monad). Or else it may be a subsystem that implements an imperative algorithm over mutable data structures (the `ST` monad), and this algorithm may return anything or nothing (the `Maybe` monad). This can all be done without monads, but monads give you a better way to reason about the problem, namely, a combinatorial language. You pay a little cost when learning monads and monad transformers but gain a magically powerful tool to decrease code entropy.

> **TIP** If you don't have that much experience with monad stacks and monad transformers, consider to read Appendix A "Word statistics example with monad transformers".

In this section, we learned about monads and monad stacks and saw how monads can be subsystems. The monad stack itself can be a subsystem: every monad in it represents a layer of abstraction having some specific properties. And every monad stack is a monad too. You imagine what your subsystem should do and pick an appropriate monad for it. Then you arrange the monads in a stack and enjoy. Doing so will get you a composable and finely controllable effect. But this section has all been theory. Important theory? Yes. Still, I hear your doubts through space and time. You need practice. Let it be so.

3.2 Designing the Hardware subsystem

Upon returning to the office, you get a letter: your employer, Space Z Corporation, has proposed a new requirement. They want you to create a language for hardware description, another HDL to add to all those that have already been invented. HDLs come in different forms: standardized and ad hoc, proprietary and free, popular and unknown. Space Z Corporation wants its own. The language should have external representation, should be able to describe a wide set of measuring devices and terminal units, should aim to be simple rather than universal, and should abstract the nuts and bolts of native APIs. At the end of the letter, you see a remark: "We expect to get some prototype to be sure the design is good enough and the architectural solutions are appropriate." Good. Designing a language is your favorite part of work. It shouldn't be delayed indefinitely.

The model-driven design of the Hardware subsystem would be the best place to start from, but you decide to clarify the requirements first. According to SRP and ISP, this subsystem should be divided into small parts:

- *HDL embedded DSL* — the base HDL. These data structures and helper functions are used to define the spaceship's control hardware: engines, fuel tanks, solar panels, remote terminal units, and so on. These devices consist of components, such as sensors and terminal units, that read measurements and evaluate commands. One HDL script should represent a single device with many components inside. Many HDL scripts form a set of available devices to make a spaceship control network. The language will be used as a tool for declaring controllable objects in the network. The definitions aren't devices but rather stateless declarations (templates, if you wish) of devices that should be instantiated and compiled into internal representation.[3] There can be many internal representations of devices — for example, one simplified representation for tests and one for actual code. HDL should allow extending the set of supportable devices.
- *HDL external DSL* — a special format of HDL that engineers will use to describe devices in separate files and load the descriptions into the program. External definitions of hardware will be translated into HDL by parsing and translation services.
- *Runtime*. These data structures and services operate on instances of hardware at runtime: they read measurements, transmit commands and results, and check availability and state. These structures are what the HDL should be compiled to. The Logic Control subsystem evaluates the control of devices using hardware runtime services.

Figure 3.3 shows the Hardware subsystem design.

[3] This is very similar to classes and objects in OOP, but the definitions are still functional because they will be stateless and declarative.

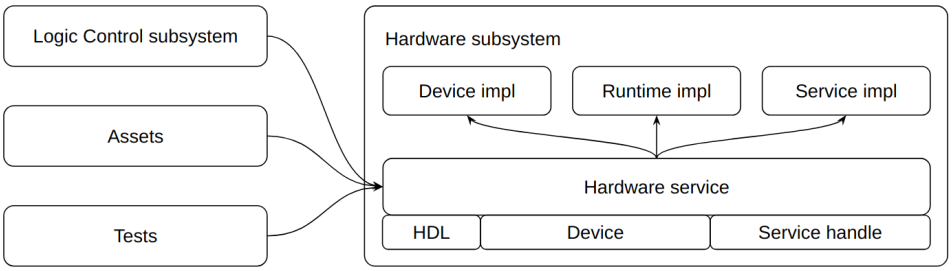


Figure 3.3 Design of the Hardware subsystem

3.2.1 Requirements

According to the documentation, every device has a passport. This thick pack of papers contains schemes, hardware interface descriptions, installation notes, and other technical information. Unfortunately, we can't encode all this stuff for every engine, every solar panel, and every fuel tank that has ever been produced; this would be the work of Sisyphus. But we know devices have common properties and abilities: sensors to measure parameters and intellectual terminal units to evaluate commands. Table 3.2 presents several imaginary devices and components.

Table 3.2 Imaginary devices for the spaceship

Device	**Manufacturer's identifier**	**Components**
Temperature sensor	AAA-T-25	None
Pressure sensor	AAA-P-02	None
Controller	AAA-C-86	None
Booster	AAA-BS-1756	Sensor AAA-T-25, “nozzle1-t” Sensor AAA-P-02, “nozzle1-p”

		Sensor AAA-T-25, “nozzle2-t” Sensor AAA-P-02, “nozzle2-p” Controller AAA-C-86
Rotary engine	AAA-RE-68	Sensor AAA-P-02, “nozzle-p” Sensor AAA-T-25, “nozzle-t” Controller AAA-C-86
Fuel tank	AAA-FT-17	4 propellant-depletion sensors for fuel 4 propellant-depletion sensors for oxidizer Controller AAA-C-86

The following short descriptions could be also found in the documentation, but they don’t play that much role for our tasks.

- *Temperature sensor*. A sensor for measuring temperatures. Widely used in space devices. Has good measurement characteristics.
- *Pressure sensor*. A sensor widely used in space devices. Has good measurement characteristics.
- *Controller*. A simple microcontroller suitable to work in the space conditions.
- *Booster*. The main engine of a spaceship, used for movement in space. This modification has built-in pressure and temperature sensors in each of two nozzles.
- *Rotary engine*. A small engine controlling rotations of a spaceship. A rotary engine has one nozzle and two sensors: temperature and pressure.
- *Fuel tank*. A fuel-supplying device that has one tank for fuel and one tank for the oxidizer.

Because of the fundamental role of the HDL eDSL in this subsystem, it's wise to start the design by defining the domain model in ADTs. This will be the raw, naive approach to the Hardware subsystem interface.

3.2.2 Algebraic data type interface to HDL

The HDL should be divided into two parts: a human-readable descriptive language to describe devices and the runtime structures the first part should interpret to. Runtime data types should be abstract. This means we have to develop interface functions and types with the implementation hidden. Why do we need a two-part architecture like this? Let's list the reasons for a separate descriptive HDL:

- We can design this language to be human-readable, which is often not appropriate for runtime operations.
- A separate language for HDL definitions doesn't depend on a runtime subsystem.
- Compilation of HDL definitions can be conditional. We can interpret HDL definitions either to runtime structures or to mock structures for tests.
- Sometimes we'll create parsers and translators for external HDL into HDL definitions, or maybe into runtime structures, depending on the case we want to support.
- Runtime data structures can hold additional information needed for calculations.
- Runtime values should have a defined lifetime, while device definitions can be in a script that exists outside the evaluation flow.

Let the first design of our HDL be simple. Listing 3.5 shows the type `Hdl`, which is just a list of `Component`'s.

Listing 3.5 HDL for device definitions

```
module Andromeda.Hardware.Hdl where

type Guid = String
data Parameter = Temperature | Pressure
data ComponentClass = Sensors | Controllers
type ComponentIndex = String
```

```
data ComponentDef = ComponentDef                #A
  { componentClass :: ComponentClass
  , componentGuid :: Guid
  , componentManufacturer :: String
  , componentName :: String
  }

data Component
  = Sensor ComponentDef ComponentIndex Parameter
  | Controller ComponentDef ComponentIndex

type Hdl = [Component]

sensor = Sensor                                 #B
controller = Controller
temperature = Temperature
pressure = Pressure
```

#A Description of component: manufacturer, serial number, class, and name
#B Constructors for convenience

In listing 3.6, you can see sample definitions of hardware using lists of components.

Listing 3.6 Sample definitions of controlled hardware

```
aaa_p_02 = ComponentDef Sensors     guid1 "AAA Inc." "AAA-P-02"
aaa_t_25 = ComponentDef Sensors     guid2 "AAA Inc." "AAA-T-25"
aaa_c_86 = ComponentDef Controllers guid3 "AAA Inc." "AAA-C-86"

boostersDef :: Hdl
boostersDef =
  [ sensor aaa_t_25 "nozzle1-t" temperature
  , sensor aaa_p_02 "nozzle1-p" pressure
  , sensor aaa_t_25 "nozzle2-t" temperature
  , sensor aaa_p_02 "nozzle2-P" pressure
  , controller aaa_c_86 "controller"
  ]
```

Remember when we designed the Logic Control eDSL in the previous chapter? We stopped developing our eDSL with lists and ADTs and said we would return

to advanced DSL design later. Whereas the design using primitive and ADTs works, it's not so enjoyable for the end user because the list syntax for composed data structures is too awkward. Even worse, it reveals the details of the `Hdl` type implementation. We already said that a naked implementation means high coupling of code, where changing one part will break the parts depending on it. If we decide to change the `Hdl` type to make it, say, an ADT, the code based on the value constructors will become invalid, as will any interpreters we may write:

```
type Device = ()       -- A dummy type. Design it later!

interpret :: Hdl -> IO Device
interpret [] = putStrLn "Finished."
interpret (c:cs) = case c of
  Sensor _ _ _   -> putStrLn "Interpret: Sensor"
  Controller _ _ -> putStrLn "Interpret: Controller"
```

So, what eDSL design options do we have to solve the problem of high coupling and revealing the implementation details? Great news, everyone: our simple HDL can be easily modified into a monadic form that abstracts the way we compose values of the type `Hdl`. We'll see how a monadic interface to our HDL eDSL is better than an algebraic one, due to hiding HDL details but preserving the ability to process the internal structure. But we'll delay this talk a bit because we need to define the runtime part of the Hardware subsystem, so we don’t quit with the `Device` type undefined.

3.2.3 Functional interface to the Hardware subsystem

Runtime instances of devices serve a similar purpose to file handles in an operating system: the instance knows how to connect to the device, read measurements, and evaluate commands. So, any value of the type `Device` will be an instance we can use for dealing with engines, fuel tanks, lights, and other spaceship devices. But we don’t want to decompose this type into subtypes to access the needed sensor or controller. We shouldn't even know what the internal type structure is to keep coupling low, but without the possibility of decomposing the `Device` type, how can we actually use it? With the *abstract* (or *opaque)* data type `Device`, the Hardware subsystem must envisage a functional interface to it. This is the only way to interact with the `Device` type outside the subsystem.

> **DEFINITION** A *functional interface* to a subsystem is a set of abstract types accompanied by functions that work with those types without revealing the implementation details.

A value of the `Device` type represents an instance of the real device in the hardware network. We need a way to create the value with only a single HDL device definition. This means there should be a function with the following signature:

```
makeDevice :: Hdl -> Device
```

It's easy to see that the function is actually an interpreter from the `Hdl` type to the `Device` type; we'll build it later, as well as the `Device` type itself. Using it is rather artless:

```
let boosters = makeDevice boostersDef
```

Now we want to do something with it. For example, to read data from the sensor with index "nozzle1-t," the appropriate function will be

```
type Measurement = ()    -- A dummy type. Design it later!
readMeasurement :: ComponentIndex -> Device
                -> Maybe Measurement
```

Oops, this isn't obvious. What is the type `Measurement` for, and why is the return type `Maybe Measurement`? Let's see:

- The `Measurement` data type carries a value with a measurement unit attached to it. The ability to define the measurement parameter we provided for a sensor in HDL would help us make the physical calculations safe. The type `Measurement` should not allow improper calculations at the type level. This is an interesting design task involving some type-level tricks (phantom types) — an excellent theme for us to learn advanced type system patterns. Intriguing? Good, but let's do things in the right order: we'll take a look at type-level design in the corresponding chapter.
- The type `Maybe a` is used when there are failure and success alternatives for calculations. Here, the index passed into the function `readMeasurement` can point to a nonexistent device component. In

this case, the return value will be `Nothing`; otherwise, it should be `Just value`.

- The function `readMeasurement` is pure. Pure means it can't have side effects ...

Let's figure out how the `Device` type can be built. Simple: it consists of a set of components, each having an index uniquely identifying the component inside a device. Consequently, it's just a map from component indexes to components:

```
module Andromeda.Hardware.Device where

type Measurement = ()     -- A dummy type. Design it later!
data DeviceComponent = Sensor Measurement Guid
                     | Controller Guid

newtype Device = DeviceImpl (Map ComponentIndex DeviceComponent)
```

Note that the type `Component` defined earlier has almost the same value constructors as the type `DeviceComponent`, but the purposes of the types differ. This is just a naming issue; you can name your structures differently — for example, `SensorInstance` and `ControllerInstance` — but there's nothing bad about having the same names for value constructors. You just put your types in different modules, and this is enough to solve the name clash problem. When you need both types `DeviceComponent` and `Component` in one place, you just import the modules qualified. The typical usage of this can appear in tests:

```
-- file /test/HardwareTest.hs:
module HardwareTest where
import Andromeda.Hardware.Hdl
import qualified Andromeda.Hardware.Device as D

sensorDefinition = Sensor
sensorInstance = D.Sensor     -- Module Device will be used
```

I said there's nothing bad about having the same names, and it's true. But it is bad to provide too much information about the internals of our runtime-aimed type `Device`. For instance, we know the constructor `D.Sensor` exists, as it's shown in the preceding code. As a result, the two modules are highly coupled. If we want the module `HardwareTest` to be independent from

eventual changes to the Hardware subsystem implementation, we should first make the `Device` type abstract, and then define more public functions, thus forming the interface to the Hardware subsystem. The complete code will look like listing 3.7.

Listing 3.7 Interface to the Hardware runtime data structures

```
module Andromeda.Hardware.Device (
    Device,                    #A
    DeviceComponent,           #A
    blankDevice,               #B
    addSensor,                 #B
    addController,             #B
    getComponent,              #B
    updateComponent,           #B
    setMeasurement,            #B
    readMeasurement            #B
 ) where

data DeviceComponent = Sensor Measurement Guid
                     | Controller Guid
newtype Device = DeviceImpl (Map ComponentIndex DeviceComponent)

blankDevice :: Device
addSensor :: ComponentIndex -> Parameter
          -> ComponentDef -> Device -> Device
addController :: ComponentIndex -> ComponentDef
              -> Device -> Device
getComponent :: ComponentIndex -> Device
             -> Maybe DeviceComponent
updateComponent :: ComponentIndex -> DeviceComponent
                -> Device -> Maybe Device
setMeasurement :: ComponentIndex -> Measurement
               -> Device -> Maybe Device
readMeasurement :: ComponentIndex
                -> Device -> Maybe Measurement
```

#A Abstract types with the implementation hidden
#B Functional interface to this part of the Hardware subsystem. Can be extended as needed

Wow, there's a lot going on here! This is actually a small part of the interface that will be extended as needed. You can meditate on it, guessing what these functions do and how to implement them. In particular, consider the `makeDevice` function. I mentioned earlier that it is an interpreter from the `Hdl` type to the `Device` type. We also can say it is a translator. In the following code, you can see a possible implementation of it (not minimal, really):

```
makeDevice :: Hdl -> Device
makeDevice hdl = makeDevice' hdl blankDevice
  where
    makeDevice' [] d = d
    makeDevice' (c:cs) d = makeDevice' cs (add' c d)
    add' (Sensor c idx p)   = addSensor idx p c
    add' (Controller c idx) = addController idx c
```

Here, we use recursion to traverse the list of component definitions and pattern matching for composing the `Device` type. This code doesn't know too much about the `Device` type: we use interface functions only. This is very important because interpreting the HDL is the responsibility of the module `Andromeda.Hardware.Device`. Unfortunately, the `makeDevice` function is fragile because it depends on the value constructors. We still risk this code being broken if something changes in the `Hdl` type. Remember, we were developing an interpreter for the Logic Control subsystem embedded language, and said that lists and ADTs aren't the best design choice for embedded languages. Now imagine a drumroll and fanfare: the Free monad comes onto the scene.

3.2.4 Free monad interface to HDL

The Free monad, which comes from the depths of category theory, is used to wrap some (but not all) types into a monad. Applications of the Free monad usually live in the academic part of functional programming, but there is one important functional pattern we can adopt in practical development. The Free monad pattern helps you to separate subsystem interface from subsystem implementation and provide an eDSL that your clients can use to describe scenarios for your subsystem without knowing the implementation details. An eDSL you provide is a Free monad language that reflects the logic of your subsystem in a safe, concise way. A free interpreter you also provide translates

eDSL scripts into internal representation. It takes each step of the scenario and converts it into real subsystem actions. There can be many interpreters for different purposes. Figure 3.4 shows the Free monad pattern.

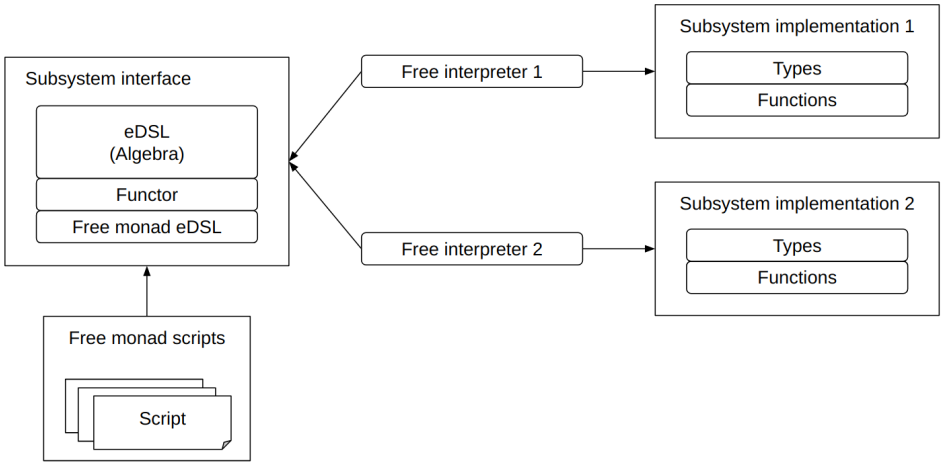


Figure 3.4 The Free monad pattern

As applied to the Hardware subsystem, this design pattern turns into the structure presented in figure 3.5.

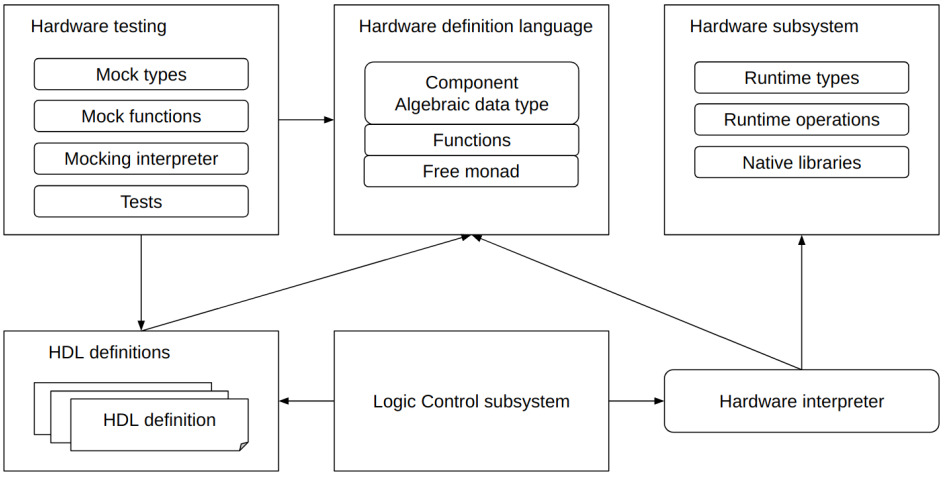


Figure 3.5 Hardware subsystem design

Note that the Logic Control subsystem doesn't need to know about Hardware subsystem runtime types. The Hardware Interpreter connects the HDL language actions and runtime operations of the Hardware subsystem. It’s also possible to test HDL scripts separately from native functions by mocking them in the Mocking Interpreter. This design makes Hardware and Logic Control subsystems low-coupled and independent. Now it's time to reinforce the `Hdl` type. Listing 3.8 demonstrates the free monadic `Hdl` type and a typical script in it.

Listing 3.8 Reinforced Hdl type and Hdl script

```
module Andromeda.Hardware.Hdl where

import Control.Monad.Free

type Guid = String                                #A
data Parameter = Temperature | Pressure           #A
data ComponentClass = Sensors | Controllers       #A
type ComponentIndex = String                      #A
                                                  #A
temperature = Temperature                         #A
pressure = Pressure                               #A

data ComponentDef = ComponentDef                                  #A
  { componentClass :: ComponentClass                              #A
  , componentGuid :: Guid                                         #A
  , componentManufacturer :: String                               #A
  , componentName :: String                                       #A
  }

aaa_p_02 = ComponentDef Sensors     guid1 "AAA Inc." "AAA-P-02"
aaa_t_25 = ComponentDef Sensors     guid2 "AAA Inc." "AAA-T-25"
aaa_c_86 = ComponentDef Controllers guid3 "AAA Inc." "AAA-C-86"

-- Free monadic eDSL

data Component a                                                  #1
  = SensorDef ComponentDef ComponentIndex Parameter a
  | ControllerDef ComponentDef ComponentIndex a                   #B

instance Functor Component where                                  #2
```

```
  fmap f (SensorDef cd idx p a)   = SensorDef cd idx p (f a)
  fmap f (ControllerDef cd idx a) = ControllerDef cd idx (f a)

type Hdl a = Free Component a                                    #3

-- Smart constructors
sensor :: ComponentDef
       -> ComponentIndex
       -> Parameter
       -> Hdl ()
sensor c idx p = Free (SensorDef c idx p (Pure ()))              #4

controller :: ComponentDef -> ComponentIndex -> Hdl ()
controller c idx = Free (ControllerDef c idx (Pure ()))

-- Free monadic scripting

boostersDef :: Hdl ()                                            #5
boostersDef = do
    sensor aaa_t_25 "nozzle1-t" temperature
    sensor aaa_p_02 "nozzle1-p" pressure
    sensor aaa_t_25 "nozzle2-t" temperature
    sensor aaa_p_02 "nozzle2-P" pressure
    controller aaa_c_86 "controller"

#A All this stuff remains unchanged
#B Actions of eDSL
#1 An updated algebra of eDSL
#2 Functor for algebra
#3 Free monad eDSL
#4 Smart constructors for actions (monadic)
#5 Free monad eDSL script
```

We should discuss the new concepts this listing introduces. Technically speaking, the Free monad is a monad. It can be built over any type that is a `Functor` in a mathematical sense. If you have such a type that's an algebra of your domain, all you should do to make it monadic is declare a `Functor` for it and then declare a Free monad over that functor. In listing 3.8, the parameterized type `Component a` #1 is the algebra; it also has the `Functor` instance #2, and #3 is a Free monad wrapped over the `Component a` functor. So, we need to clarify all these concepts in detail. Let's start with `Functor`.

> **TIP** This section introduces the Free monad, with some implementation details described to give you a basic knowledge of what's inside. To get a deeper explanation, you might want to read some of the papers and tutorials available on the Internet. Consider starting with the excellent tutorial "Why Free Monads Matter," by Gabriel Gonzalez. You may also find good references listed in the bibliography of this book.

Type is a `Functor` when you can map some function `f` over its internals without changing the data structure. Plain list is a functor because you can map many functions over its internals (items), while the structure of the list remains the same. For example,

```
oldList :: [Int]
oldList = [1, 2, 3, 4]

newList :: [String]
newList = map show list

-- newList: ["1", "2", "3", "4"]
```

When you call the `map` function, you actually use the list type as a `Functor`. You may use the `fmap` function instead — the only method of the `Functor` type class:

```
newList' = fmap show list
```

This is the same. Consider the type class `Functor` and its instance for the list type:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map
```

Here, `f` is `[]` (list), the `a` type variable is `Int`, and the `b` type variable is `String`:

```
fmap :: (Int -> String) -> [Int] -> [String]
```

By mapping the (`show :: Int -> String`) function over the list, we get a list of the same size. The structure of the data doesn't change, but the internal type may be different. Both `oldList` and `newList` are lists, but they carry items of different types: `Int` and `String`.

Now we can declare the `Functor` instance for the `Component` type. The `fmap` function will be specified like this:

```
fmap :: (a -> b) -> Component a -> Component b
```

Consequently, our type should have a type parameter for some purpose. Why? The internal values having this type will hold *continuations* for the Free monad (whatever that means). The algebra `Component a` #1 has two value constructors: `SensorDef` and `ControllerDef`. Both have fields with parameter type `a,` both are mappable. This is not a must for any `Functor`, but it is a must for the Free monad pattern. Every action of the Free eDSL should be mappable because every action should be able to hold continuations (we'll see it later). When the `Functor` instance is declared (see #2), we can map any appropriate function over the `Component a` type:

```
sensorValue :: Component Int
sensorValue = SensorDef aaa_t_25 "nozzle1-t" temperature 1000

mappedValue :: Component String
mappedValue = fmap show sensorValue

expectedValue :: Component String
expectedValue = SensorDef aaa_t_25 "nozzle1-t" temperature
"1000"

-- expectedValue and mappedValue are equal:
> mappedValue == expectedValue
True
```

Next, we'll see what the `Free` type is. If you check the `Hdl` type #3, it's not just a list anymore but something else:

```
-- Old:
type Hdl = [Component]
boostersDef :: Hdl
```

```
-- New:
type Hdl a = Free Component a
boostersDef :: Hdl ()
```

Here, the type `Free f a` is the Haskell standard implementation of the `Free` concept (think carefully about which occurrence of the word “Free” here is a type constructor and which is a value constructor):

```
data Free f a = Pure a
              | Free (f (Free f a))
```

`Pure` and `Free` are value constructors with a special meaning:

- `Pure`. This (monadic) value finalizes the chain of computations. It also keeps some value of type `a` to be returned after the interpretation is over.
- `Free`. Action (monadic) that should be able to keep another action of type `Free a` inside. We call the other action a *continuation* because when the outer action is interpreted, the process of interpretation should be continued with the inner action.

The `f` type variable is the type for which the `Functor` instance exists. In our case, it's the `Component` type. The `a` type variable you see in definition #3 specializes both the `Free` type and the `Component` type. The record

```
Free Component a
```

means the `Component` type will share exactly the same type (`a`) variable. We can rewrite the definition of the `Free` type like so:

```
Free Component a = Pure a
                 | Free (Component (Free Component a))
```

Consider the type `(Component (Free Component a))` that the second value constructor has: it's our `SensorDef` or `ControllerDef` value with the continuation field that has the monadic type `(Free Component a)` again. The type `Hdl` now represents a type synonym to the `Free Component` type, which means you can write either `Hdl a` or

`Free Component a`. For example, you may construct these values of the `Free Component a` type:

```
val1 :: Free Component Float
val1 = Pure 10.0

val2 :: Free Component Float
val2 = Free (SensorDef aaa_t_25 "nozzle1-t" temperature val1)

val3 :: Hdl Float
val3 = Free (ControllerDef aaa_t_25 "nozzle1-t" val2)

val4 :: Free Component ()
val4 = Free (ControllerDef aaa_t_25 "nozzle1-t" (Pure ()))
```

Note that we put `val1` into the continuation field of `val2`, and then we put `val2` into the continuation field of `val3`. You may read this as “Define a controller with these parameters. Then define a sensor. Then return 10.0.” The `Pure` value stops this recursive nesting process and declares the final value to be returned by the whole computation `val3`. That's why the `Float` return type of `val1` is “inherited” by `val2` and then by `val3`. This component of the `Free` type shows what type the whole computation should return when it's interpreted. If we don't want to return something useful, we put (`Pure ()`) with the unit value.

Smart constructors #4 allow you to construct values much easier:

```
sensor :: ComponentDef -> ComponentIndex -> Parameter -> Hdl ()
sensor c idx p = Free (SensorDef c idx p (Pure ()))

controller :: ComponentDef -> ComponentIndex -> Hdl ()
controller c idx = Free (ControllerDef c idx (Pure ()))

val5 = Pure "ABC"

val6 :: Hdl ()
val6 = sensor aaa_t_25 "X" temperature
```

But now it's not obvious how to place `val5` into `val6`. What's important is that all of these values are monadic because they are instances of the `Free` type, which is a monad. We actually don't have to nest them by hand as we did

with `val1`, `val2`, and `val3`. The monadic mechanism of the Free monad will do it for us in the background of the `do` notation. Consider the definition of boosters #5: it's an example of monadic actions bound into one big monadic computation. Remember, a monadic type should be represented by some monadic type constructor `m` and a return value `a`:

```
monadicValue :: m a
```

Here, the `m` type equals `Free Component`, and the `a` type may vary depending on what we put into the continuation field (in the previous examples, `a` is `String`, `Float`, and `()`). It's not visible from the code, but monadic binding uses the `fmap` function to nest values and push the bottom `Pure` value deeper in the recursive chain. The following pseudocode demonstrates what will happen if you combine `val5` and `val6` monadically:

```
bind val5 val6 =>

bind (Pure "ABC")
  (Free (SensorDef aaa_t_25 "X" temperature (Pure ()))) =>

  (Free (SensorDef aaa_t_25 "X" temperature (Pure "ABC")))
```

The idea of the Free monad pattern is that you can chain the actions of your algebra, turning them into monadic functions that are composable due to their monadic nature. Actually, there’s one only fundamental way to chain computations in functional programming, upon which all other concepts are based. Not continuations. Not monads. *Recursion*. It doesn't matter whether we mean recursive functions or data types, continuations, or another thing that has a self-repeating property. Remember when we utilized a functional list earlier to stack the actions of our eDSL? Again: any functional list is a recursive data structure because it has a head and tail of the same list type. The Free monad behaves very similarly to the functional list, but whereas the list stores items in a head and tail, the Free monad stores actions in actions themselves — that's what fields of type `a` are intended for. In other words, to chain some actions of type `Component a`, we put one action inside another:

```
data Component a
  = SensorDef ComponentDef ComponentIndex Parameter a
  | ControllerDef ComponentDef ComponentIndex a

a1 = ControllerDef definition1 index1 ()
a2 = SensorDef definition2 index2 par1 a1
a3 = SensorDef definition3 index3 par2 a2
```

Unfortunately, this code has a big problem — the more actions we insert, the bigger the type of the action becomes:

```
a1 :: Component ()
a2 :: Component (Component ())
a3 :: Component (Component (Component ()))
```

That is, all three variables `a1`, `a2`, and `a3` have different types, and we can't use them uniformly. The `Free` type solves this problem by wrapping our enhanced type with its own recursive structures. Because of this, the nesting of values of type `Free Component a` instead of just `Component a` gives a unified type for the whole chain regardless of its length:

```
fa1 = Free (ControllerDef definition1 index1 (Pure ()))
fa2 = Free (SensorDef definition2 index2 par1 fa1)
fa3 = Free (SensorDef definition3 index3 par2 fa2)
fa1 :: Free Component ()
fa2 :: Free Component ()
fa3 :: Free Component ()
```

Notice how the `Free` actions reflect the idea of a Russian nesting doll. We'll see more interesting features of the Free monad we didn’t discuss here — in particular, how to handle return values your domain actions may have. All actions of the `Component` type are declared as nonreturning, so they don't pass results further to the nested actions. We'll continue developing our knowledge of the Free monad pattern in the next chapter.

Let's summarize what we learned into the “free monadizing” algorithm:

1. Declare a domain-specific algebra, for example, an ADT with value constructors representing domain actions.
2. Make the algebra a `Functor`, as the monadic nature of the Free monad requires. To do this, create an instance of the `Functor` type class by

defining the `fmap` function. The algebra should be a type that's parametrized by a type variable. Also, all the value constructors should have a field of this parameterized type to hold nested actions.

3. Define a monadic type for your functor.
4. Create smart constructors for every value constructor of your algebra.
5. Create interpreters for concrete needs.

Our eDSL now has a monadic interface. The scripts you can write using this monadic language are just declarations of actions to be evaluated. For instance, the `boostersDef` script is not a boosters device itself but a template we can translate into many boosters devices that we operate at runtime. We still need something to translate the `Hdl` language into the code that does the actual work. This is what free interpreters do. We can't hold any operational information in the `boostersDef` script, but the specific runtime type `Device` that we'll translate the script to can store component state, uptime, error logs, and other important stuff. We should design such a runtime type and find a way to instantiate our device definitions.

3.2.5 Interpreter for monadic HDL

What result should the interpreting process end with? It depends on your tasks. These may be

- *Interpreting (compiling) to runtime types and functions*. This is a common use case. The Free monad language is just a definition of logic — a definition of actions an eDSL has — not the acting logic itself. Having a Free monad eDSL, we define the data it transforms and declare how it should work after it's interpreted, but we don't dictate what real operations the client code should do when it interprets our eDSL action definitions.
- *Interpreting to mocking types in tests*. Instead of real actions, test interpreters can do the same work as popular mocking libraries (like Google Test and Google Mock): counting calls, returning stubs, simulating a subsystem, and so on.
- *Translating to another eDSL*. In doing this, we adapt one interface to another. This is the way to implement a functional version of the Adapter OO pattern.
- *Interpreting to a printable version*. This can be used for logging if a

source type of a Free language can't be printed in the original form.

- *Serializing*. With an interpreter traversing every item, it's not so difficult to transform an item into either a raw string or JSON, or XML, or even a custom format.

When the HDL language was just a list of algebraic data types, we eventually translated it into the `Device` type:

```
data Component
    = Sensor ComponentDef ComponentIndex Parameter
    | Controller ComponentDef ComponentIndex

type Hdl = [Component]

data Device = ...

makeDevice :: Hdl -> Device
```

But now the `Hdl` type is a monadic type, the internal structure of which is very different. We made the new version of the `Hdl` type a Free monad, so the previous `makeDevice` function can't be used anymore. We're going to translate our monadic language using a new version of the `makeDevice` function, and the `Device` type will be the target runtime data structure:

```
data Component next
    = SensorDef ComponentDef ComponentIndex Parameter next
    | ControllerDef ComponentDef ComponentIndex next

type Hdl a = Free Component a

makeDevice :: Hdl () -> Device
```

Note that the `Component` type was improved to take a new type variable (in this case, it's parametrized by the unit type `()`), and also the `Component` type that has received one more field with the `next` type. Keep in mind that every value constructor (action) stores the action to be evaluated next. This is the structure of the Free monad pattern. Next we should create an interpreter for these actions. Where should this code be placed? In other words, what module is responsible for creating runtime "devices"? Let's think:

- The module `Andromeda.Hardware.HDL` has definitions of HDL eDSL. It knows nothing about `Device`.
- The module `Andromeda.Hardware.Device` holds the abstract type `Device`. It knows nothing about `Hdl ()`.

Putting the interpreter into one of these modules will break the SRP. But the module for runtime operations looks like a good candidate for the function `makeDevice`.

How should this function work? Remember that the type `Hdl ()` has been exported with the value constructors. The fact that our language is a monad doesn't mean it isn't still an algebraic data type. So, we pattern match over it. The complete code of the interpreter is presented in the following listing.

Listing 3.9 Interpreting the Hdl () type

```
module Andromeda.Hardware.Runtime where

import Andromeda.Hardware.HDL
import Andromeda.Hardware.Device
  ( Device, blankDevice, addSensor, addController )

import Control.Monad.Free

makeDevice :: Hdl () -> Device                          #1
makeDevice hdl = interpretHdl blankDevice hdl

interpretHdl :: Device -> Hdl () -> Device              #2
interpretHdl device (Pure _)    = device
interpretHdl device (Free comp) = interpretComponent device comp

interpretComponent :: Device -> Component (Hdl ()) -> Device
interpretComponent device (SensorDef c idx par next) =
  let device' = addSensor idx par c device
  in interpretHdl device' next
interpretComponent device (ControllerDef c idx next) =
  let device' = addController idx c device
  in interpretHdl device' next

#1 Interface function that just calls the interpreter
#2 Interpreter translates Hdl to Device
```

The following explanation should help you understand the tangled interpreting process:

- `makeDevice` (#1) is a pure function that initiates the interpreting process with a blank device created by the library function `blankDevice`.
- The `interpretHdl` (#2) function pattern matches over the `Free` algebraic data type, which has two constructors. We aren't interested in a `Pure` value (yet), so this function body just returns a composed device. Another function body deconstructs the `Free` value constructor and calls the interpreter for a component definition value.
- The `interpretComponent` function deconstructs a value of the `Component` type and modifies the device value using the library functions `addSensor` and `addController`.
- Remember, all value constructors (actions) of the type `Component` carry the continuation actions in their latest fields. When we pattern match over the `SensorDef` and the `ComponentDef` values, we put these continuations into the `next` value. Again, this `next` value has the `Hdl ()` type (which is a synonym of the `Free` type), so we should continue interpreting this nested `Hdl ()` script. We do so by calling the `interpretHdl` function recursively. The process repeats, with a modified device value passed in the arguments, until the end of a `Free` chain is reached. The end of a `Free` chain is always `Pure` with some value — in our case, the unit value `()`.

> **TIP** If it's not clear what's happening in these examples, consider reading additional sources discussing the Free monad, functors, algebraic data types, and DSLs in functional programming.

Using the code in listing 3.9 as an example, you can write your own free language interpreter. I suggest that you write an interpreter that serializes `Hdl` definitions into JSON. In other words, the following function should be implemented (this will be good practice for you):

```
toJSON :: Hdl () -> String
```

3.2.6 Advantages and disadvantages of a free language

What else should we know about the Free monad pattern? The following list summarizes the pattern's advantageous properties:

- *Monadic*. A monadic interface to an embedded language is much more convenient than just a functional one, especially for sequential and mutually dependent actions. Besides, you gain the power of standard monadic libraries, the functions of which will considerably enhance your monadic language.
- *Readable and clean*. A language that has a Free monad interface is usually self-descriptive. Monadic actions that the language is turned to will be obvious in usage.
- *Safely composable and type-safe*. Two monadic actions can be monadically composed; they have the same Free monad type. The base bricks of this monadic language are defined by your free language. In the next chapter we'll see another good example of safe composability when we talk about monadic actions returning some useful result. In that case, we'll be able to define return types of monadic actions explicitly in the value constructors of your domain algebra. This will add more static checking of the correctness of your free scripts.
- *Interpretable*. The Free monad pattern provides an easy way to interpret declarative definitions of domain entities into evaluable actions or into another language. Different interpreters serve different needs.
- *Abstract*. A Free monad interface abstracts the underlying language. Scripts don't depend on the implementation details. It's even possible to completely replace the implementation of a language: all code contracts and invariants will be saved. The Free monad mechanism is an analogue of the OOP interface in this sense.
- *Small adopting cost*. To "free monadize" your language, which is probably an ADT, you make your language a functor and wrap it in the `Free` type. Unless you examine what's inside a Free monad library, it stays simple.
- *Small maintaining cost*. Modifying both a language and a monadic interface isn't a big deal. You also need to keep the interpreters consistent, which can break some external code if the monadic interface isn't stable, but this is a common problem of interfaces regardless of whether they are object-oriented or functional.

As you can see, the Free monad pattern has a lot of advantages over the algebraic interfaces we were creating earlier. But there are some disadvantages too:

- Because it's a monad, the user should know monads well to use your language.
- The Free monad requires that the language be a functor. Although it's an easy transformation for your language, you need to understand what it's for.
- The theory behind the Free monad is complex. For example, the `Free` type has a recursive value constructor that's kind of the fix point notion (the mathematical concept of recursive operation). Also, converting a language into the Free monad form looks like a magical hack. But it's not magic, it's science. The good thing is that you can use the Free monad in practice without knowing the theory at all.
- It's hard (sometimes impossible) to convert your language into the Free monad form when it's made of advanced functional concepts: GADTs, phantom types, type-level logic, and so on. This probably means you need another approach to the problem.
- Some implementations of the Free monad can be slow as they approximate O(n^2) in the binding mechanism. But there are other Free monads, so this is not a problem.

TIP The "no remorse" Free monad from PureScript has a fairly fast implementation. It's called so because it's based on the paper "Reflection without remorse". Another option is the Church-encoded Free monad, the `F` type from the same package as the Free monad. This one is really fast. Going ahead, its performance is on par with Final Tagless. We'll be discussing these things in chapters 6-9, but for now our goal is to learn how Free monads can be useful in Domain-Driven Design.

LINK *Reflection without remorse*
http://okmij.org/ftp/Haskell/zseq.pdf

TIP The Hydra project (a project for the third part of this book) has two engines: based on the normal Free monad and based on the

Church-encoded Free monad. You can compare the performance of the two by auditing the same application written on top of them.

LINK The Hydra project
https://github.com/graninas/Hydra

At this point, you should be comfortable with simple eDSLs. You know how to convert a simple, list-based language into a monadic language, for which you should implement a `Functor` instance to wrap your type into the Free monad. But we've just begun exploring this big theme; in fact, there is much more to learn about Free monadic languages. Remember the Logic Control eDSL we worked out in the previous chapter? It has actions `ReadTemperature`, `AskStatus`, and `InitBoosters`, and they all return some values when they're evaluated! Can the Free monad handle this? Completely, and as you'll see, it's brilliant. So, we want to revisit the Logic Control eDSL, and we'll do that in the next chapter. There's a lot of work ahead!

3.3 Functional services

I would like to begin with an apology. This section does not discuss authentic services, as you might expect. Unlike the famous gurus of software development, I am not brave or competent enough to delve into the theory of, you know, service-oriented architectures, CORBA, microservices, REST APIs, and so on. I'll stay on the practical path of the functional developer who solves a problem by just writing more functions, where others may prefer to invent a brand-new killer technology. We'll adopt a simple definition of a functional service so as not to take the bread out of the gurus' mouths.

DEFINITION A *functional service* is a single-task behavior hidden by a public functional interface that allows us to substitute the code at runtime transparently to the client code.

This definition doesn't say anything about the remoteness of a functional service, so it doesn't matter whether the service works in the same process or is forked into a separate one. The definition requires only that the functional service have a functional interface: one of the ones we discussed previously (ADT interface, interface that is made of functions, or a Free monad interface) or another we haven't touched on yet. Functional services can be pure or impure, local or remote — all these properties are representable in functional interfaces and

types. For the sake of clarity, we'll look deeply at local services and lightly touch on remote ones.

3.3.1 Pure service

Suppose we know that the Hardware subsystem will be replaced by an improved version in the future, but today we can't do this easily because it has too many external relations. For now, it's supposed to be used as shown in figure 3.6.

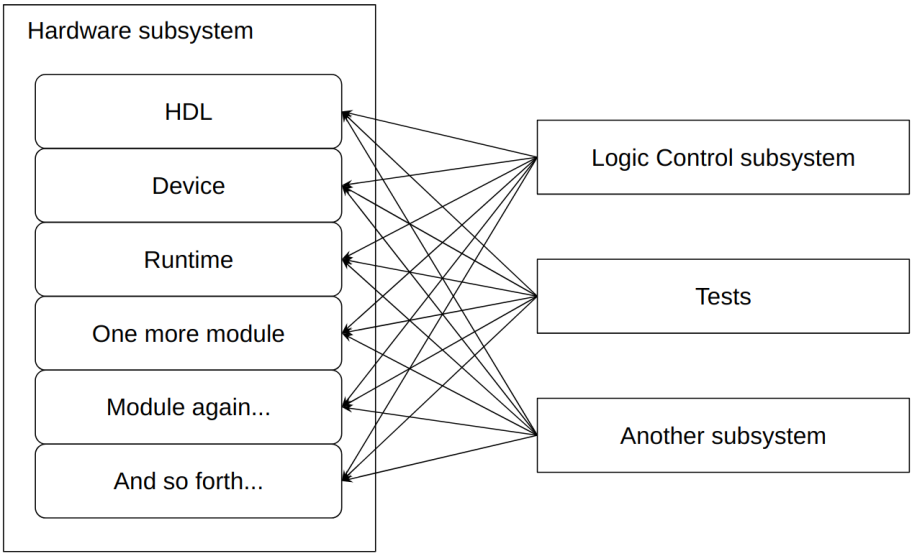


Figure 3.6 Complex relations between subsystems

Some other subsystems depend on the Hardware subsystem, and we don't want this dependency to be strong. The functional interfaces we developed earlier definitely decrease coupling between subsystems, but these interfaces provide too much functionality and too many small pieces. The more relations there are, the more difficult the refactoring will be if we need to update the subsystem. We might want a simple lever that just does the whole task of replacing an old version with a new one. In other words, it would seem to be a good thing to have just one relation to the Hardware subsystem. Consider the code in listing 3.10.

Listing 3.10 Pure service for the Hardware subsystem

```
module Andromeda.Hardware.Service where
import Andromeda.Hardware.HDL
import Andromeda.Hardware.Device
import Andromeda.Hardware.Runtime

data Handle = Handle {                          #A
  createDevice :: Hdl () -> Device,
  getBlankDevice :: Device
  }

newHandle :: (Hdl () -> Device) -> Device -> Handle
newHandle mkDeviceF getBlankF = Handle mkDeviceF getBlankF

defaultHandle :: Handle
defaultHandle = newHandle makeDevice blankDevice
```

#A Handle for the service of the Hardware subsystem;
should contain the methods available

The code speaks for itself. The type `Handle` will be our service, which has a few methods to interact with the subsystem: we have one focused service rather than many direct calls. Every field here represents a pure method that the Hardware subsystem provides via the service. The real methods of the subsystem can be placed into this handle, but that's not all. It's possible to place mocks instead of real methods: all we need to do is initialize a new `Handle` with mock functions. This is shown in listing 3.11.

Listing 3.11 Mocking the subsystem

```
testDevice :: Handle -> IO ()                           #A
testDevice h  = do
  let blank    = getBlankDevice h
  let notBlank = createDevice h boostersDef
  if (blank == notBlank)
    then putStrLn "FAILED"
    else putStrLn "passed"

makeDeviceMock _ = blankDevice                          #B
mockedHandle = newHandle makeDeviceMock blankDevice
```

```
test = do
  putStr "With real service: "
  testDevice defaultHandle

  putStrLn "With mocked service: "
  testDevice mockedHandle

-- Output:
-- > test
-- With real service: passed
-- With mocked service: FAILED

#A This code uses the service by calling functions
   stored in the handle
#B Mocked service; does nothing but return blank devices
```

Notice that the function `testDevice` hasn't been changed; it works fine with both real and mocked services, which means it depends on the Hardware subsystem interface (types) rather than the implementation.

It’s not rocket science, really. We’ve introduced a pure service that can be considered a boundary between subsystems. It's pure because we store pure methods in the `Handle`, and it’s also stateless because we don't keep any state behind. You might ask what the benefits of the design are. The service approach reduces complexity by weakening the coupling of the application. We can also say that the service and its core type, `Handle`, represent a new kind of functional interface to the Hardware subsystem. The diagram in figure 3.7 illustrates the whole approach.

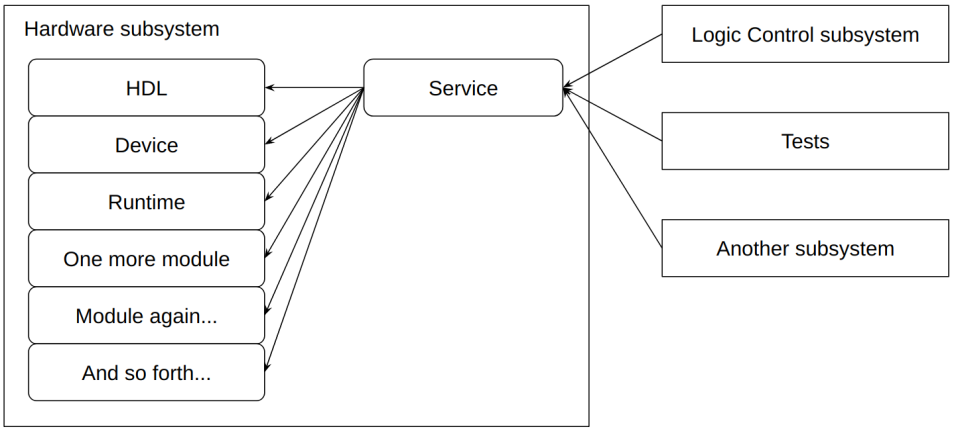


Figure 3.7 Simple relations between subsystems

Despite the fact that we prefer pure subsystems as much as possible, reality often pushes us into situations in which we have to depart from this rule. The obvious problem with this service is that it's pure, and can't be made remote when we suddenly decide it should be. Additionally, we might want to evaluate some impure actions while doing stuff with the service, such as writing logs. So, we need to look at impure services too.

3.3.2 Impure service

What parts of the service could be impure to allow, for example, writing a message to the console when some method is called? Look at the service's `Handle` type again:

```
data Handle = Handle
  { createDevice   :: Hdl () -> Device
  , getBlankDevice :: Device
  }
```

The algebraic data type `Handle` is nothing but a container. Fields it contains represent the methods of the subsystem: client code takes a handle and calls these functions, and each call should be logged before the actual method is invoked. Consequently, fields of the `Handle` type should be impure. Let's wrap them into the `IO` type:

```
data Handle = Handle
  { createDevice :: Hdl () -> IO Device
  , getBlankDevice :: IO Device
  }

newHandle :: (Hdl () -> IO Device) -> IO Device -> Handle
newHandle mkDeviceF getBlankF = Handle mkDeviceF getBlankF
```

After this, some changes in the `testDevice` function should be made. First, we need to replace pure calls with impure ones:

```
testDevice :: Handle -> IO ()
testDevice h  = do
  blank    <- getBlankDevice h               -- Impure!
  notBlank <- createDevice h boostersDef     -- Impure!
  if (blank == notBlank)
    then putStrLn "FAILED"
    else putStrLn "passed"
```

Next, consider the most interesting part of this redesign. We can no longer put the functions `makeDevice` and `blankDevice` into the `Handle` because their types don't match with the impure types of the fields:

```
makeDevice :: Hdl () -> Device      -- Pure!
blankDevice :: Device               -- Pure!
```

But now it's possible to create arbitrary impure actions instead of just calls to the subsystem (see listing 3.12).

Listing 3.12 Default and mocked services with impure logging

```
defaultHandle :: Handle
defaultHandle = newHandle mkDeviceF' getBlankF'
  where
    mkDeviceF' ::  Hdl () -> IO Device                #A
    mkDeviceF' hdl = do
      putStr " (createDevice defaultHandle) "
      return (makeDevice hdl)
    getBlankF' :: IO Device
    getBlankF' = do
      putStr " (getBlankDevice defaultHandle) "
      return blankDevice
```

```
mockedHandle = newHandle mock1 mock2                    #B
  where
    mock1 _ = do
      putStr " (createDevice mockedHandle) "
      return blankDevice
    mock2 = do
      putStr " (getBlankDevice mockedHandle) "
      return blankDevice

#A These functions will do real work:
   translate Hdl to Device and return a real blank device
#B These functions will do nothing but return a blank device
```

The function `test`, which hasn't changed, produces the following output:

```
With real service:
  (getBlankDevice defaultHandle) (createDevice defaultHandle)
passed

With mocked service:
  (getBlankDevice mockedHandle) (createDevice mockedHandle)
FAILED
```

We're now armed with a design pattern that allows us to create impure services. In fact, this exact `Handle` type can access the remote service because it's possible to put any logic for remote communication (for example, sockets or HTTP) into the impure fields `createDevice` and `getBlankDevice`. In the next section, we will study one way to implement a remote service working in a separate thread. It will utilize the MVar request–response communication pattern.

3.3.3 The MVar request–response pattern

To prepare the ground for introducing a remote impure service, we'll learn about the MVar request–response pattern. The idea behind a concurrently mutable variable (or, for short, MVar) is pretty simple. The variable of the `MVar a` type may hold one value of type `a` or hold nothing. `MVar a` is a variable of some type `a` that can be changed. The mutation of MVar is a real mutation in the sense of imperative programming, but what's important is that it's thread-safe. Since

the notion of the MVar involves mutation, it works on the `IO` layer. The MVar interface consists of the creation functions

```
newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
```

and the blocking functions for reading and writing contents

```
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
```

A value of type `MVar a` can be in two states: either empty or full. When it's empty, every thread that calls `takeMVar` will wait until it's full. Conversely, when the MVar is full, a thread that calls `putMVar` will wait until it's empty. Concurrency means that threads can't access the MVar simultaneously; they must wait their turn (first in, first out), and that's why the MVar can't have an invalid state. But classic deadlocks are still possible because of the imperative nature of the MVar's interface.

> **TIP** MVars have other behavioral issues you might want to know about. To learn more about these, consult other resources.

With MVars, it's possible to construct a remote service that works in a separate thread, waits until a request is pushed, processes it, and pushes the result back. This is known as the MVar request–response pattern. Let's take a look at its structure.

The MVar request–response pattern operates with two MVars: one for the request and one for the response. These MVars together represent a communication channel that can abstract any kind of interaction between threads. The construction of the pattern includes two functions: requester and response processor. The basic idea and workflow are shown in figure 3.8.

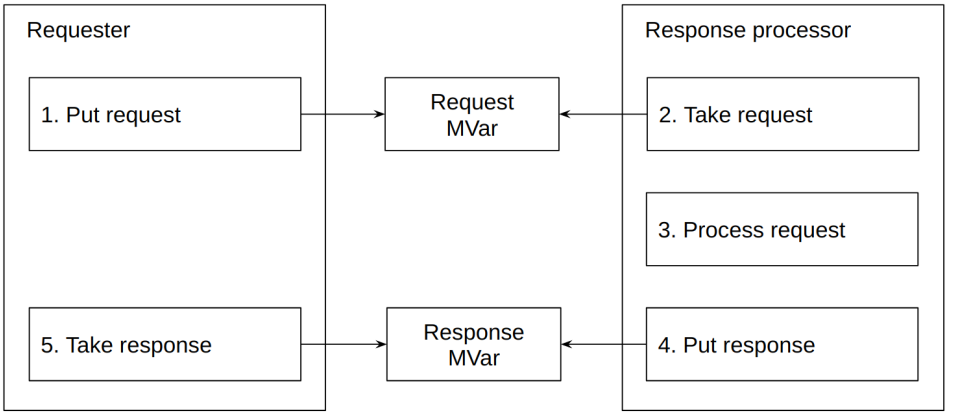


Figure 3.8 The MVar request–response pattern flow

The requester performs two steps:

A. Put a value into the request MVar.

B. Take the result from the response MVar.

If the response processor isn't yet ready to accept the request, the first step will make the requester wait. The second step will block the requester if the processor is busy processing the previous request.

The response processor does the opposite. It performs three steps:

C. Take the request value from the request MVar.

D. Process the request to get the response.

E. Put the response into the response MVar.

If a request hasn't been placed yet, the first step will block the processor until this is done. The third step can block the processor if the requester hasn’t had time to extract the previous response.

This is the description. Now let's demonstrate this pattern in an example. We’ll create a separate thread that receives a number and converts it to a FizzBuzz word. We’ll also create the number generator thread that puts requests, waits for results, and prints them. Let's start by defining the `Pipe` type, the `createPipe` function, and the generic function `sendRequest`. Both

`createPipe` and `sendRequest` functions are operating by the `MVar` type, so they should be impure. This means these functions are monadic in the `IO` monad:

```
type Request a = MVar a
type Response b = MVar b
type Pipe a b = (Request a, Response b)

createPipe :: IO (Pipe a b)
createPipe = do
  request  <- newEmptyMVar
  response <- newEmptyMVar
  return (request, response)

sendRequest :: Pipe a b -> a -> IO b
sendRequest pipe@(request, response) a = do
  putMVar request a
  takeMVar response
```

The `sendRequest` function puts the `a` value into the request MVar and waits for the `b` value from the response MVar. As the function `takeMVar` is the last in the `do` block, it returns the `b` value received. That's why the type of this function is `IO b`.

Next, we create a worker function that will listen to the channel forever and process requests when they occur. It returns nothing useful to the client code. We say that it does some impure effect in the `IO` monad, and then no result is returned, so this function will have the `IO ()` type:

```
worker :: Pipe a b -> (a -> b) -> IO ()
worker pipe@(request, response) f = do
  a <- takeMVar request
  putMVar response (f a)
  worker pipe f
```

Then we parameterize this worker using our `fizzBuzz` function:

```
fizzBuzz x | isDivided x 15 = "FizzBuzz"
           | isDivided x 5  = "Buzz"
           | isDivided x 3  = "Fizz"
           | otherwise      = show x
```

```
isDivided x n = (x `mod` n) == 0

fizzBuzzProcessor :: Pipe Int String -> IO ()
fizzBuzzProcessor pipe = worker pipe fizzBuzz
```

Next, we create the `generator` function that will feed the remote worker forever:

```
generator :: Pipe Int String -> Int -> IO ()
generator pipe i = do
  result <- sendRequest pipe i
  putStrLn ("[" ++ show i ++ "]: " ++ result)
  generator pipe (i + 1)
```

Almost done. The last step is to fork two threads and see what happens:

```
main = do
  pipe <- createPipe
  forkIO (fizzBuzzProcessor pipe)
  forkIO (generator pipe 0)
```

As a result, we'll see an infinite request–response session with the FizzBuzz words printed:

```
...
[112370]: Buzz
[112371]: Fizz
[112372]: 112372
[112373]: 112373
...
```

The preceding code illustrates the use of this pattern well. Now it's time to create a remote impure service.

3.3.4 Remote impure service

In this case, there will be just one thread for the processor that will accept a command with parameters, evaluate it, and return the result. The function `evaluateCommand` looks like the following:

```
type Command = (String, String)
type Result = String
type ServicePipe = Pipe Command Result
```

```
evaluateCommand :: Command -> String
evaluateCommand ("createDevice", args) = let
  hdl = read args :: Hdl ()
  dev = makeDevice hdl
  in (show dev)
evaluateCommand ("blankDevice", _) = show blankDevice
evaluateCommand _ = ""
```

As you can see, it receives the command serialized into the `String`. With the help of pattern matching, it calls the needed function of the Hardware subsystem and then returns the result.

> **NOTE** To enable the simplest serialization for the types `Hdl ()` and `Device`, you can derive the type classes `Show` and `Read` for them.

Next, just a few words about the worker — there's nothing special for it:

```
serviceWorker :: ServicePipe -> IO ()
serviceWorker pipe = worker pipe evaluateCommand
```

And finally, the service itself. We don't need to change the `Handle` type for impure services that we introduced earlier — it's okay to abstract the remote service too. But instead of the default handle, we compose the remote version that uses a pipe to push commands and extract results. It will also print diagnostic messages to the console:

```
createDevice' :: ServicePipe -> Hdl () -> IO Device
createDevice' pipe hdl = do
  print "Request createDevice."
  result <- sendRequest pipe ("createDevice", show hdl)
  print "Response received."
  return (read result :: Device)

blankDevice' :: ServicePipe -> IO Device
blankDevice' pipe = do
  print "Request blankDevice."
  result <- sendRequest pipe ("blankDevice", "")
  print "Response received."
  return (read result :: Device)

newRemoteHandle :: ServicePipe -> Handle
```

```
newRemoteHandle pipe
    = newHandle (createDevice' pipe) (blankDevice' pipe)
```

The usage is obvious: create the pipe, fork the service worker, create the service handle, and pass it to the client code. The function `test` from previous code listings will look like this:

```
test = do
  pipe <- createPipe
  forkIO $ serviceWorker pipe

  putStr "With real service: "
  testDevice (remoteHandle pipe)

  putStr "With mocked service: "
  testDevice mockedHandle
```

That's it. We ended up with a remote service. What else is there to say? Well done! You may use this design pattern to wrap REST services, remote procedure calls, sockets, and whatever else you want. It gives you an abstraction that’s better than just direct calls to subsystems — and that’s the goal we’re moving toward.

3.4 Summary

It's time to leave this beautiful place. We’ve gained a lot of knowledge about DSLs and interpreters, we’ve learned how to design functional interfaces, and we’ve investigated functional services. But the main aim of this chapter is not just introducing techniques and patterns, but also decreasing accidental complexity with the help of the abstractions introduced. That is, we want to achieve the following goals by dividing the application into subsystems, modules, layers, and services:

- *Low coupling*. We can easily observe the ways of communication with a subsystem if it has fewer relations with the outside environment.
- *Better maintainability*. Changing requirements leads to code modification. We prefer modular code rather than spaghetti code because it can be modified more easily. Also, introducing DSLs and adopting proper abstractions makes the code self-descriptive.
- *Wise responsibility distribution*. The SRP says that each part of the code

should have only one responsibility, only one idea to implement. Why? Because we want to know what this part does and what it will never do, but don't want to have to examine the code to check. When each part does a single thing, it's easy to reason about a whole system.

- *Abstractions over implementations*. Abstractions hide implementation details from the client code. This is good because the client code sees only a high-level interface to a subsystem, and the knowledge about the internals is limited by this interface. Ideally, an interface to a subsystem should be small and clear to simplify its usage. Moreover, an abstraction makes it possible to modify (or even completely replace) an implementation without breaking the client code. Finally, correct abstractions give additional power and expression, which leads to less code and more functionality.
- *Testability*. We designed a pure subsystem that can be tested well: we just need to test all its pure functions to be sure it's working as we expect.

Functional programming elaborates great abstractions that are supported by mathematically correct idioms like Free monads and monad transformers. That’s the main reason why functional abstractions compose safely. The laws they follow make it almost impossible to do the wrong things. Consequently, our applications will be more correct when they are written functionally.

The next chapter will tell you a lot about domain modeling. We’ll design the domain model of the Logic Control subsystem along with learning functional idioms and patterns we haven’t touched on yet.

Chapter 4
Domain model design

This chapter covers

- How to analyze a domain mnemonically and find its general properties
- How to model the domain in different eDSLs
- Combinatorial languages and domain modeling
- External language parsing and translation

When Sir Isaac Newton invented his theory of gravitation, he probably never imagined that the basis of that theory would be significantly reworked in the future, involving an entirely different set of mathematical abstractions. The theory, while being inapplicable to very massive or very fast objects, still works fine for a narrower scale of conditions. Newton's apple will fall down accurately following the law of universal gravitation if it has a low speed or falls to a relatively low mass object. What abstraction we choose for our calculations — Newton's classical mechanics or Einstein's more modern theory of relativity — depends on the problem we want to solve. It's important to consider whether we're landing our ship on the Moon or traveling near the black hole in the center of our galaxy, because the conditions are very different, and we must be sure our formulas are appropriate to give us an adequate result. After all, our lives depend on the right decision of what abstraction to choose, so we must not be wrong.

Software developers do make wrong decisions, and it happens more often than it actually should. Fortunately, software development is not typically so fatalistic. People's lives don't usually depend on the code written — usually, but not

always. That’s definitely not true when we talk about sensitive domains like SCADA and nuclear bomb calculations. Every developer must be responsible and professional to decrease the risks of critical software errors, but the probability of such will never become zero (unless we shift to formal methods to prove the correctness of the program, which is really hard). Why then do we ignore approaches that are intended to help developers write valid, less buggy code? Why do we commit to technologies that often don't prove to be good for solving our regular tasks? The history of development has many black pages in which using the wrong abstractions ruined great projects. Did you know there was a satellite called the Mars Climate Orbiter that burned up in that planet’s atmosphere because of a software bug? The problem was a measurement unit mismatch, in which the program returned a pound-seconds measurement, but it should have been newton-seconds. This bug could have been eliminated by testing or static type-level logic. It seems the developers missed something important when they were programming this behavior.

Abstractions can save us from bugs in two ways:

- By making it simple to reason about the domain; in other words, decreasing accidental complexity (which is good), so that we can easily see the bugs.
- By making it impossible to push the program into an invalid state; in other words, encoding a domain so that only correct operations are allowed.

A domain is the knowledge of how a certain object or process works in the real world. A domain model is a representation of a domain. A domain model includes a more or less formal definition of the data and transformations of that domain. We usually deal with domain models that are expressed by code, but a model can also be a set of diagrams or a specific domain language. In this chapter, we'll study how to design a domain model, including what tools we have available to make it correct and simple. While it's certainly fine to want correctness in software, it's not so obvious why unnecessary complex abstractions can lead to bugs. The main reason is that we lose the focus of the domain we’re implementing and start treating the abstraction as a universal hammer that we think can solve all our problems at once. You probably know the result when the project suddenly falls into abstraction hell and no one piece of domain becomes visible through it. How can you be sure all the requirements are handled? In this situation, you'll more likely find many bugs you could have avoided by having a fine-readable and simple but still adequate abstraction. The

abstractions shouldn't be too abstract; otherwise they tend to leak and obfuscate your code. We'll discuss domain modeling as the main approach to follow and see what abstractions we have for this. We'll also continue developing the ideas we touched on in the previous chapters.

Roll up your sleeves; the hard work on the Logic Control subsystem is ahead. So far, the Logic Control eDSL we wrote as a demonstration of domain-driven modeling seems to be naive and incomplete because it doesn't cover all the corresponding domains. The scope of work in this chapter includes the following:

- Define the domain of Logic Control.
- Describe the Logic Control subsystem's functional requirements and responsibilities.
- Implement the domain model for Logic Control in the form of eDSLs.
- Elaborate the combinatorial interface to the Logic Control subsystem.
- Create an external representation of the Logic Control DSL.
- Test the subsystem.

The first goal of this chapter is to create the code that implements most of the functional requirements of Logic Control. The second goal is to learn domain model design and new functional concepts that are applicable at this scale of software development. Keep in mind that we're descending from the architecture to the design of subsystems and layers in a top–bottom development process. We discussed the approaches to modularizing the application in chapter 2, and we even tried out some ideas of domain modeling on the Logic Control and Hardware subsystems, but not all questions were clarified. Let's clarify them, then.

4.1 Defining the domain model and requirements

What is a domain? What is a domain model? We all have an intuitive understanding of these terms just because every time we code, we're implementing a tool in some domain. By doing this, we want to ease tasks and maybe introduce automation into them. Our tool should simplify solving hard tasks and make it possible to solve tasks one cannot solve without computers at all. As we plunge into the SCADA field, a good example of this will be manufacturing automation — a SCADA domain in which the quality and an

amount of production play the leading role. But this is what our developer's activity looks like to the customer from the outside. In contrast, when we discuss a domain, we’re interested in the details of its internal structure and behavior, what notions we should take into consideration, and what things we can ignore. Then we start thinking about the domain model, namely, what data structures, values, and interactions could represent these notions and behaviors in order to reflect them properly. This is what we have to learn to do every day. In fact, we’re already familiar with domain modeling — a process we participate in to design domain models. We just need more tools to do it right.

As usual, we'll start from requirements, building on what we described for Logic Control earlier. Remember how we converted requirements into a DSL for hardware definition in chapter 3? We can't be successful in functional domain modeling if we haven't sorted out the real nature of the domain and its notions.

The Logic Control subsystem encapsulates code that covers a relatively big part of the spaceship domain. The subsystem provides a set of tools (functional services, eDSLs, and so on) for the following functionality (partially described in the mind map in figure 2.7):

1. Reading actual data from sensors.
2. Accessing archive data and parameters of the hardware under control.
3. Handling events from other subsystems and from hardware.
4. Running control scripts and programs. Scripts can be run by different conditions:
   - By Boolean condition
   - By exact time or by hitting a time interval
   - By event
   - By demand from other subsystems
   - Periodically with a time interval
5. Accessing the hardware schema of a spaceship.
6. Monitoring of spaceship state and properties.
7. Autonomously correcting spaceship parameters according to predefined behavioral programs.
8. Sending commands to control units.

9. Evaluating mathematical calculations.
10. Handling hardware errors and failures.
11. Testing scenarios before pushing them into a real environment.
12. Abstracting different hardware protocols and devices.
13. Logging and managing security logs.

Characterizing this list of requirements as a comprehensive domain description would be a mistake, but it's what we have for now. You probably noticed that we discover requirements in increments while descending from the big scale of application design to the details of particular subsystems and even modules. This gives us a scope of the functionality we know we can implement immediately. In real development, you'll probably want to focus on one concrete subsystem until you get it working properly. But it's always probable you'll end up with a malformed and inconsistent design that doesn't match the rest of the subsystems, and then you'll be forced to redesign it when problems emerge.

4.2 Simple embedded DSLs

It would be perfect to unify all this domain stuff with a limited yet powerful DSL. You may ask, why a DSL? Why not just implement it "as is" using functional style? In talking about design decisions like this one, we should take into consideration the factors discussed in chapter 1. Will it help to achieve goals? Can a DSL give the desired level of simplicity? Does the solution cover all the requirements? Do we have human resources and money to support this solution in the later stages of the software lifecycle?

The DSL approach addresses one main problem: the smooth translation of the domain's essence into the code without increasing accidental complexity. Done right, a DSL can replace tons of messy code, but, of course, there's a risk that it will restrict and annoy the user too much if it's designed badly. Unfortunately, there are no guarantees that introducing a DSL will be a good decision. Fortunately, we don't have any choice: the SCADA software should be operated by a scripting language. We just follow the requirements.

In this section, we'll take several approaches and see if they're good for modeling embedded languages. The first one is rather primitive and straightforward: a "pyramid" of plain functions. It's fine if you can fit all the domains in one page. Otherwise, you'd better try ADTs. In this case, your

algebraic structures encode every notion of a domain, from objects to processes and users. Depending on your taste, you may design it with more or less granularity. It's fine if you just keep data in these structures, but you still need a "pyramid" of plain functions to do related stuff. Otherwise, you'd better try a combinatorial approach, which I describe in the next section.

4.2.1 Domain model eDSL using functions and primitive types

In functional programming, everything that has some predefined behavior can be implemented by a composition of pure and impure functions that operate on primitive types. The following example produces a Morse-encoded FizzBuzz — a simple problem we all know very well:

```
import Data.Char (toLower)

fizzBuzz :: Int -> String
fizzBuzz x | (x `mod` 15) == 0 = "FizzBuzz"
           | (x `mod` 5)  == 0 = "Buzz"
           | (x `mod` 3)  == 0 = "Fizz"
           | otherwise = show x

morseCode :: [(Char, String)]
morseCode =
  [ ('b', "-..."), ('f', "..-."), ('i', ".."), ('u', "..-")
  , ('z', "--.."), ('0', "-----"), ('1', ".----")
  , ('2', "..---"), ('3', "...--"), ('4', "....-")
  , ('5', "....."), ('6', "-...."), ('7', "--...")
  , ('8', "---.."), ('9', "----.")
  ]

toMorse :: Char -> String
toMorse char = case lookup char morseCode of
  Just code -> code
  Nothing   -> "???"

morseBelt :: Int -> [String]
morseBelt = map (' ' :)
          . map toMorse
          . map toLower
          . fizzBuzz
```

```
morseFizzBuzzes :: String
morseFizzBuzzes = (concat . concatMap morseBelt) [1..100]

main = putStrLn morseFizzBuzzes
```

You see here a long functional conveyor of transformations over primitive data types — the `morseBelt` function that takes an integer and returns a list of strings. Four separate functions are composed together by the composition operator `(.)`: each of them does a small piece of work and passes the result to the next workshop, from right to left. The transformation process starts from a `fizzBuzz` function that converts a number to a FizzBuzz string. Then the function `map toLower` takes the baton and lowercases every letter by mapping over the string. Going further, the lowercased string becomes a list of Morse-encoded strings, and the last function `(' ' :)` pads it with spaces. We strongly view the composition of functions like this one as functional code.

> **DEFINITION** A *functional eDSL* is an embedded DSL that uses functional idioms and patterns for expressing domain notions and behavior. A functional eDSL should contain a concise set of precise combinators that do one small thing and can be composed in a functional manner.

On the basic level of “functional programming Jediism,” you don't even need to know any advanced concepts coming from mathematical theories, because these concepts serve the purpose of unifying code patterns to abstract behavior and make the code much more safe, expressive, and powerful. But the possibility of just writing functions over functions is still there. By going down this path, you'll probably end up with verbose code that will look like a functional pyramid (see figure 1.10 in chapter 1). It will work fine, though. Moreover, the code can be made less clumsy if you group functions together by the criterion that they relate to the same part of the domain, regardless of whether this domain is really the domain for what the software is or is the auxiliary domain of programming concepts (like the message-passing system, for example).

Let's compose an impure script for obtaining `n` values from a sensor once a second. It gets the current time, compares it with the old time stamp, and, if the difference is bigger than the delay desired, reads the sensor and decrements a

counter. When the counter hits zero, all `n` values are read. Here is a Haskell-like pseudocode:

```
getTime :: IO Time                                          #A
readDevice :: Controller -> Parameter -> IO Value           #B

readNEverySecond n controller parameter = do                #C
  t <- getTime
  out <- reading' n ([], t) controller  parameter
  return (reverse out)

reading' 0 (out, _) _ _ = return out                        #D
reading' n (out, t1) controller parameter = do
  t2 <- getTime
  if (t2 – t1 >= 1)
  then do
    val <- readDevice controller parameter
    reading' (n-1) (val:out, t2) controller  parameter
  else reading' n (out, t1) controller name
```

#A Function from system library
#B Function from hardware-related library
#C "eDSL" (not really)
#D Auxiliary recursive function

This whole code looks ugly. It wastes CPU time by looping indefinitely and frequently asks the time from the system. It can't be configured normally using a custom time interval because the `readNEverySecond` function is too specific. Imagine how many functions will be in our eDSL for different purposes:

```
readNEverySecond
readTwice
readEveryTwoSecondsFor10Seconds
...
```

And the biggest problem with this eDSL is that our functions aren't that handy for use in higher-level scenarios. Namely, these functions violate the SRP: there are mixed responsibilities of reading a sensor, counting the values, and asking the time. This DSL isn't combinatorial because it doesn't provide any functions

with a single behavior you might use as a constructor to solve your big task. The preceding code is an example of spaghetti-like functional code.

Let's turn the preceding code into a combinatorial language. The simplest way to get composable combinators is to convert small actions into higher-order functions, partially applied functions, and curried functions:

```
threadDelay :: DiffTime -> IO ()                          #A
replicate :: Int -> a -> [a]
sequence :: Monad m => [m a] -> m [a]
(>>=) :: Monad m => m a -> (a -> m b) -> m b

readDevice :: Controller -> Parameter -> IO Value         #B

delay :: DiffTime -> Value → IO Value                     #C
delay dt value = do
    threadDelay dt
    return value

times :: Int -> IO a -> IO [a]
times n f = sequence (replicate n f)

readValues                                                #D
  :: DiffTime -> Int -> Controller
  -> Parameter -> IO [Value]
readValues dt n controller param = times n (reader >>= delayer)
  where
    reader  = readDevice controller param
    delayer = delay dt
```

#A Functions from system libraries
#B Function from hardware-related library
#C eDSL
#D Script

Let's characterize this code:

- *Impure*. It makes unit testing hard or even impossible.
- *Instantly executable*. I stated in the previous chapter that interpretable languages give us another level of abstraction. You write a script, but it really can't be executed immediately. Rather, it should be translated into executable form first and then executed. This means your script is

declarative. The closest analogy here is the string of code the `eval` function will evaluate in such languages as PHP and JavaScript. So, you don't describe your domain in a specific language — you program your domain in the host language.[4]

- *Vulnerable*. Some decisions (like `threadDelay`, which blocks the current thread) can't be considered acceptable; there should be a better way. Indeed, we'll see many ways that are better than the eDSL implemented here.

Interestingly, every part of functional code that unites a set of functions can be called an internal DSL for that small piece of the domain. From this viewpoint, the only measure of a function to be a part of a DSL is to reflect its notion with the appropriate naming.

4.2.2 Domain model eDSL using ADTs

It's hard to imagine functional programming without ADTs. ADTs cover all your needs when you want to design a complex data structure — tree, graph, dictionary, and so forth — but they're also suitable for modeling domain notions. Scientifically speaking, an ADT can be thought of as "a sum type of product types," which simply means "a union of variants" or "a union of named tuples." Algebraic type shouldn't necessarily be an exclusive feature of a functional language, but including ADTs in any language is a nice idea due to good underlying theory. Pattern matching, which we dealt with in the previous chapter, makes the code concise and clean, and the compiler will guard you from missing variants to be matched.

We can design a domain model using ADTs in different ways. The `Procedure` data type we developed in chapter 2 represents a kind of straightforward, naive approach to sequential scenarios. The following listing shows this type.

[4] Well, the line between programming a domain and describing it using an eDSL is not that clear. We can always say that the IO action isn't really a procedure, but rather a definition of a procedure that will be evaluated later.

Listing 4.1 Naive eDSL using an ADT

```
-- These types are defined by a separate library
data Value = FloatValue Float
           | IntValue Int
           | StringValue String
           | BoolValue Bool
           | ListValue [Value]

-- Dummy types, should be designed later
data Status = Online | Offline
data Controller = Controller String
data Power = Float
type Duration = Float
type Temperature = Kelvin Float

data Procedure
  = Report Value
  | Store Value
  | InitBoosters (Controller -> Script)
  | ReadTemperature Controller (Temperature -> Script)
  | AskStatus Controller (Status -> Script)
  | HeatUpBoosters Power Duration

type Script = [Procedure]
```

The “procedures” this type declares are related to some effect: storing a value in a database, working with boosters after the controller is initialized, reporting a value, and so on. But they’re just declarations. We don't actually know what real types will be used in runtime as a device instance, as a database handler, and as a controller object. We abstract from the real impure actions: communicating with databases, calling library functions to connect to a controller, sending report messages to a remote log, and so on. This language is a declaration of logic because when we create a value of the `Script` type, nothing actually happens. Something real will happen when we bind these actions to real functions that do the real work. Notice also that the sequencing of procedures is encoded as a list, namely, the `Script` type. This is how a script might look:

```
storeSomeValues :: Script
storeSomeValues =
  [ StoreValue (FloatValue 1.0)
  , StoreValue (FloatValue 2.0)
  , StoreValue (FloatValue 3.0)
  ]
```

This script may be transferred to the subsystem that works with the database. There should be an interpreting function that will translate the script into calls to a real database, something like

```
-- imaginary bindings to database interface:
import Control.Database as DB

-- interpreter:
interpretScript :: DB.SQLiteConnection -> Script -> IO ()
interpretScript conn [] = return ()
interpretScript conn (StoreValue v : procs) = do
  DB.store conn "my_table" (DB.format v)
  interpretScript procs
interpretScript conn (unknownProc : procs) =
  interpretScript procs
```

It should be clear why three of the value constructors (`Report`, `Store`, and `HeatUpBoosters`) have arguments of a regular type. We pass some useful information the procedure should have to function properly. We don't expect the evaluation of these procedures to return something useful. However, the other three procedures should produce a particular value when they're evaluated. For example, the boosters initialization procedure should initialize the device and then return a sort of handle to its controller. Or one more example: on being asked for the temperature, that controller should give you a measured value back. To reflect this fact, we declare additional fields with function types in the "returning" value constructors: `(Controller -> Script)`, `(Status -> Script)`, and (`Temperature -> Script`). We also declare the abstracted types for values that should be returned (`Controller`, `Status`, and `Temperature`). This doesn't mean the particular subsystem will use exactly these types as runtime types. It's more likely that the real interpreter, when it meets a value of `Controller` type, will transform it to its own runtime type — let's say, `ControllerInstance`. This `ControllerInstance` value may have many useful fields, such as time

of creation, manufacturer, or GUID; we don't have to know about that stuff. The only thing we should know is that successful interpretation of `InitBoosters` should return some handle to the controller. But why do we use function type `(Controller -> Script)` to declare this behavior? Because we want to do something with the value returned. We may want to read the temperature using the controller instance. This means we want to combine several actions, make them chained, dependent. This is easily achievable if we adopt the same ideas we discussed in the previous chapter: we use recursive nesting of continuations for this purpose. The field `(Controller -> Script)` of the `InitBoosters` will hold a lambda that declares what to do with the controller handle we just obtained. Given the language's design, all we can do now is read the temperature or ask the status. The following demonstrates a complex script:

```
doNothing :: Temperature -> Script
doNothing _ = []

readTemperature :: Controller -> Script
readTemperature controller =
  [ ReadTemperature controller doNothing ]

script :: Script
script = [ InitBoosters readTemperature ]
```

Now all three scripts are combined. For simplicity, we can say that the `InitBoosters` procedure "returns" `Controller`, `ReadTemperature` "returns" `Temperature`, and `AskStatus` "returns" the controller's status

Listing 4.2 gives you one more example that contains a script for the following scenario:

1. Initialize boosters and get a working controller as a result.
2. Read current temperature using the controller. Process this value: report it and store it in the database.
3. Heat up boosters for 10 seconds with power 1.
4. Read current temperature using the controller. Process this value: report it and store it in the database.

Listing 4.2 Heating the boosters and reporting the temperature

```
initAndHeatUpScript :: Script                              #B
initAndHeatUpScript = [ InitBoosters heatingUpScript ]

heatingUpScript :: Controller -> Script                    #B
heatingUpScript controller =
  [ ReadTemperature controller processTemp
  , HeatUpBoosters 1.0 (seconds 10)
  , ReadTemperature controller processTemp
  ]

reportAndStore :: Value -> Script                          #C
reportAndStore val = [ Report val, Store val ]

processTemp :: Temperature -> Script
processTemp t = reportAndStore (temperatureToValue t)
```

#1 Step 1 of the scenario.
#B Steps 2, 3 and 4 of the scenario.
#C Scripts for steps 2 and 4.

An impure testing interpreter we wrote earlier (see listing 2.5) traces every step of the script to the console. The following listing shows the interpretation for the script `initAndHeatUpScript`:

```
"Init boosters"
"Read temperature"
"Report: FloatValue 0.0"
"Store: FloatValue 0.0"
"Heat up boosters"
"Read temperature"
"Report: FloatValue 0.0"
"Store: FloatValue 0.0"
```

Unfortunately, the approach of modeling a domain 1:1 in ADTs has many significant problems:

- *It's too object-oriented*. Types you design by copying the domain notions will tend to look like "classes" (the `Controller` type) and "methods" (the `ReadTemperature` value constructor). A desire to cover all the domain notions will lead you to notions-specific code. It looks unlikely

that you'll see the abstract properties of your data from such a model. For example, the procedures `Store` and `Report` can be generalized by just one `(SendTo Receiver Value)` procedure that is configured by the specific receiver: either a database, a reporter, or something else you need. The `Receiver` type can be a lambda that knows what to do: `(type Receiver :: Value -> IO ())`; however, your domain doesn't have this exact object, and you should invent it yourself.

- *Different scopes are mixed in just one god-type* `Procedure`. It's easy to dump everything into a single pile. In our case, we have two scopes that seem to be separate: the procedures for working with the controller and the reporting/storing procedures.
- *It's inconvenient*. Verbose lists as sequences of actions, value constructors of the `Procedure` type you have to place in your code, limits of the actions you can do with your list items — all these issues restrict you too much.
- *It encodes a domain "as is."* The wider a domain is, the fatter the DSL will be. It's like when you create classes `DeadCell` and `AliveCell`, inheriting them from the interface `IGameOfLifeCell`, and your class `Board` holds a `FieldGraph` of these objects, which are connected by `GraphEdge` objects …. And this whole complexity can be removed by just one good old, two-dimensional array of short integers. If there is a lesser set of meaningful abstractions that your domain can be described by, why avoid them?
- *It's primitive*. The language doesn't provide any useful abstractions.

The issues listed can be summarized as the main weakness of this modeling approach: we don't see the underlying properties of a domain. Despite this, there are some good points here:

- *Straightforward modeling is fast*. It may be useful for rapid prototype development or when the domain isn't so big. It also helps us understand and clarify the requirements.
- *It's simple*. There are only two patterns all the "procedures" should match: specifically, a procedure with a return type and a procedure without one. This also means the approach has low accidental complexity.
- *The eDSL is safe*. If the language says you can't combine two incompatible

procedures, then you can't, really.

- *The eDSL is interpretable*. This property allows you to mock subsystems or process an eDSL in different ways.
- *The eDSL can be converted to a Free monad eDSL*. When it's done, the domain is effectively hidden behind the monadic interface, so the client code won't be broken if you change the internal structure of your eDSL.

We'll soon see how to decompose our domain into much smaller, consistent, and high-cohesive parts than this eDSL has. We'll then investigate the properties these parts have. This should enlighten us by hinting at how to design a better, combinatorial eDSL.

4.3 Combinatorial eDSLs

Functional programming is finally entering the mainstream, its emergence stemming from three major directions: functional languages are becoming more popular, mainstream languages are extending their syntax with functional elements, and functional design patterns are being adopted by cutting-edge developers. While enthusiasts are pushing this wave, they're continuing to hear questions about what functional programming is and why people should care. The common use case of lambdas that we see in the mainstream is passing simple operations into library functions that work with collections generically. For example, the operations over an abstract container in the C++ Standard Template Library may receive lambdas for comparison operators, for accumulation algorithms, and so on. But be careful about saying this kind of lambda usage is functional programming. It's not; it's elements of functional programming but without a functional design. The essence of functional programming is composition of combinators and the functional idioms that make this composition possible. For example, the function `map :: (a -> b) -> [a] -> [b]` is a combinator that takes a function and a list, and returns a new list with every element modified by that function. The function `map` is a combinator because you can combine several of them:

```
morseBelt :: Int -> [String]
morseBelt = map (' ' :) . map toMorse . map toLower . fizzBuzz
```

And you may even improve this code according to the rewriting rule:

```
map f . map g == map (f . g)

morseBelt' :: Int -> [String]
morseBelt' = map ((' ' :) . toMorse . toLower) . fizzBuzz
```

It would be wrong to say that a procedure that simply takes a lambda function (for instance, `std::sort()` in C++) is functional programming. The procedure isn't a combinator because it's not a function, and therefore you can't combine it with something else. In fairness, the C++ Standard Template Library is an implementation of generic style programming that is close to functional programming, but many of the functions this library has imply both mutability and uncontrolled side effects. Immutability and side effects can ruin your functional design.

Functional programming abounds with embedded combinatorial languages. The accidental complexity of a language you design in combinatorial style is small due to the uniform way of reasoning about combinatorial code, regardless of the combinators' size. Have you heard of the `parsec` library, perhaps the best example of a combinatorial language? `Parsec` is a library of monadic parsing combinators. Every monadic parser it provides parses one small piece of text. Being monadic functions in the `Parser` monad, parsers can be combined monadically into a bigger monadic parser that is in no way different but works with a bigger piece of text. Monadic parsers make a code look like a Backus–Naur Form (BNF) of the structure you're trying to extract from text. Reading such code becomes simple even for nonprogrammers after they've had a little introduction to BNF and monadic parsing concepts. Consider the following example of parsing constant statements. The text we want to parse looks like so:

```
const thisIsIdentifier = 1
```

The following code shows parser combinators and the `Statement` structure in which we'll keep the result parsed:

```
import Text.Parsec as P
data Expr = ...   -- some type to hold expression tree
data Statement = ConstantStmt String Expr

constantStatement :: P.Parser Statement
constantStatement = do
```

```
    P.string "const"           -- parses string "const"
    P.spaces                   -- parses one or many spaces
    constId <- identifier      -- parses identifier
    P.spaces                   -- parses spaces again
    P.char '='                 -- parses char '='
    P.spaces                   -- parses spaces, too
    e <- expr                  -- parses expression
    return (ConstantStmt constId e)

str = "const thisIsIdentifier     =    1"
parseString = P.parse constantStatement "" str
```

Here, `identifier` and `expr` are the parser combinators with the same `Parser a` type:

```
identifier :: Parser String
expr :: Parser Expr
```

We just put useful stuff into variables and wrapped it in the `Statement` ADT. The corresponding BNF notation looks very similar:

```
constantStatement ::=
  "const" spaces identifier spaces "=" spaces exprAnd
```

If we line every token of BNF, it becomes even closer to the parser:

```
constantStatement ::=
  "const"
  spaces
  identifier
  spaces "=" spaces
  expr
```

We'll return to this theme in section 4.4, where we'll build an external DSL with its own syntax. Parsec will help us to parse external scripts into ASTs that we then translate to our combinatorial eDSL (going ahead, it will be a compound Free eDSL with many Free DSLs inside). Before that, we should create an eDSL. Let's do this.

4.3.1 Mnemonic domain analysis

The `Procedure` data type was modeled to support the scenario of heating boosters, but we haven't yet analyzed the domain deeply because the requirements were incomplete. We just projected a single scenario into ADT structure one-to-one and got what we got: an inconsistent DSL with dissimilar notions mixed together. In this section, we'll redesign this eDSL and wrap the result into several combinatorial interfaces — but we have to revisit the domain of Logic Control first.

We'll now try a method of analysis that states a scenario to be written in mnemonic form using various pseudolanguages. Determining the internal properties of domain notions can't be done effectively without some juggling of the user scenarios being written in pseudolanguages. This juggling can also lead you to surprising ideas about how to compose different parts uniformly or how to remove unnecessary details by adopting a general solution instead.

The next scenario we'll be working with collects several needs of the Logic Control subsystem (conditional evaluation, mathematical calculations, and handling of devices):

```
Scenario: monitor outside thermometer temperature
Given: outside thermometer @therm
Run: once a second

scenario:
  Read temperature from @therm,
       result: @reading(@time, @temp, @therm)

  If @temp < -10C Then
      register @reading
      log problem @reading
      raise alarm "Outside temperature lower than bound"

  Else If @temp > 50C Then
      register @reading
      log problem @reading
      raise alarm "Outside temperature higher than bound"

  Else register @reading
```

```
register (@time, @tempCelsius, @device):
  @tempCelsius + 273.15, result: @tempKelvin
  Store in database (@time, @tempKelvin, @device)
```

It should be clear that the scenario reads a thermometer and then runs one of the possible subroutines: registering the value in a database if the temperature doesn't cross the bounds; logging the problem and raising an alarm otherwise. The `register` subroutine is defined too. It converts the value from Celsius to Kelvin and stores it in the database along with additional information: the time stamp of the measurement and the device from which the value was read.

According to the Logic Control elements diagram (see figure 2.18), the instructions this scenario has can be generalized and distributed to small, separate DSLs: Calculations DSL, Data Access DSL, Fault Handling DSL, and so on. Within one DSL, any instruction should be generalized to support a class of real actions rather than one concrete action. A language constructed this way will resist domain changes better than a language that reflects the domain directly. For example, there’s no real sense in holding many different measurements by supporting a separate action for each of them, as we did earlier, because we can make one general parameterized action to code a whole class of measurements:

```
data Parameter
  = Pressure
  | Temperature

data Procedure
  = Read Controller Parameter (Measurement -> Script)
```

where the `Parameter` type will say what we want to read.

A different conclusion we can draw from the scenario is that it's completely imperative. All the parts have some instructions that are clinging to each other. This property of the scenario forces us to create a sequential embedded domain language, and the best way to do this is to wrap it in a monad. We could use the `IO` monad here, but we know how dangerous it can be if the user of our eDSL has too much freedom. So, we'll adopt a better solution — namely, the Free monad pattern — and see how it can be even better than we discussed in chapter 3.

However, being sequential isn’t a must for domain languages. In fact, we started thinking our scenario was imperative because we didn't try any other forms of mnemonic analysis. Let's continue juggling and see what happens:

```
Scenario: monitor outside thermometer temperature
Given: outside thermometer @therm

// Stream of measurements of the thermometer
stream therm_readings <once a second>:
  run script therm_temperature(), result: @reading
  return @reading

// Stream of results of the thermometer
stream therm_monitor <for @reading in therm_readings>:
  Store in database @reading

  run script validate_temperature(@reading), result: @result

  If @result == (Failure, @message) Then
    log problem @reading
    raise alarm @message
  return @result

// Script that reads value from the thermometer
script therm_temperature:
  Read temperature from @therm,
       result: @reading(@time, @tempCelsius, @therm)
  @tempCelsius + 273.15, result: @tempKelvin
  return (@time, @tempKelvin, @therm)

// Script that validates temperature
script validate_temperature (@time, @temp, @therm):
  If @temp < 263.15K Then
    return (Failure, "Outside T < than bound for " + @therm)
  Else If @temp > 323.15K Then
    return (Failure, "Outside T > than bound for " + @therm)
  Else return Success
```

Oh, wow! This scenario is wordier than the previous one, but you see a new object of interest here — a stream. Actually, you see two of them: the `therm_readings` stream that returns an infinite set of measurements and the `therm_monitor` stream that processes these values and does other

stuff. Every stream has the evaluation condition: once a second or whenever the other stream is producing a value. This makes the notion of a stream different from a script: the former works periodically and infinitely, whereas the latter should be called as a single instruction.

This form of mnemonic scenario opens a door to many functional idioms. The first, which is perhaps obvious, is functional reactive streams. These streams run constantly and produce values you can catch and react to. The “functionality” of streams means you can compose and transform them in a functional way. Reactive streams are a good abstraction for interoperability code, but here we’re talking about the design of a domain model rather than the application architecture. In our case, it's possible to wrap value reading and transforming processes into the streams and then construct a reactive domain model. The scenario gives a rough view of how it will look in code.

Functional reactive streams could probably be a beneficial solution to our task, but we'll try something more functional (and perhaps more mind-blowing): arrows and arrowized languages. The scenario doesn't reveal any evidence of this concept, but, in fact, every function is an arrow. Moreover, it's possible to implement an arrowized interface to reactive streams to make them even more composable and declarative. Consequently, using this concept, you can express everything you see in the scenario, like scripts, mathematical calculations, or streams of values, and the code will be highly mnemonic. So what is this mysterious concept of arrows? Keep reading; the truth is out there. First, though, we'll return to our simple and important approach to creating composable embedded languages — using the Free monad pattern — and see how it can be improved.

4.3.2 Monadic Free eDSL

Any monad abstracts a chain of computations and makes them composable in a monadic way. Rolling your own monad over the computations you have can be really hard because not all sequential computations can be monads in a mathematical sense. Fortunately, there’s a shortcut called the Free monad pattern. We discussed this pattern already in chapter 3; now we'll create another Free language that will be abstract, with the implementation details hidden.

First, we'll generalize working with remote devices. In reality, all things we do with sensors and devices we do by operating the intellectual controller that’s

embedded into any manageable device. So, reading measurements from a sensor is equivalent to asking a controller to read measurements from that particular sensor, because one device can have many sensors. In turn, measurements vary for different kinds of sensors. Also, the controller has an internal state with many properties that depend on the type of controller, for example, its local time, connectivity status, or errors. The scripting language should allow us to get and set these properties (in a limited way, possibly). Finally, the device may be intended for certain operations: open and close valves, turn lights on and off, start and stop something, and more. To operate the device, we send a command to the controller. Knowing this, we can redesign our `Procedure` data type as shown in the following listing.

Listing 4.3 Improved Procedure eDSL for working with remote devices

```
-- These types are defined in a separate library
data Value = FloatValue Float
           | IntValue Int
           | StringValue String
           | BoolValue Bool
           | ListValue [Value]
data Measurement = Measurement Value
data Parameter = Temperature | Pressure

-- Dummy types, should be designed later
data Property = Version | Status | SensorsList
data Controller = Controller String
data Command = Command String
type CommandResult = Either String String
type SensorIndex = String

-- Parametrized type for a Free eDSL
data Procedure a
  = Set Controller Property Value a                    #A
  | Get Controller Property (Value -> a)               #B
  | Run Controller Command (CommandResult -> a)
  | Read Controller SensorIndex Parameter (Measurement -> a)
```

#A “Nonreturning” definition
#B “Returning” definitions

> **NOTE** The `Measurement` type knows nothing about measurement units. This is a problem. What if you requested a `Temperature` parameter but accidentally got pressure units? How would your system behave then? In the Andromeda project, this type is improved by a phantom type tag (`Measurement tag`), so you really should use it with units, like so: (`Measurement Kelvin`). The `Parameter` type also has this tag: (`Parameter tag`). These two types, while used together, require units to be consistent, which means the values should have types with identical tags.

This new language is composed of instructions for working with remote devices through a controller. This type is also parameterized by a type variable because it will be a Free monad language, and we need to make it a `Functor`. To illustrate this better, we'll need to complete the rest of the "free monadizing" algorithm: namely, make this type an instance of `Functor` and provide convenient smart constructors. Listing 4.4 shows the instance.

Listing 4.4 The instance of Functor for the Procedure type

```
instance Functor Procedure where
  fmap g (Set c p v next)    = Set c p v   (g next)
  fmap g (Get c p nextF)     = Get c p     (g . nextF)
  fmap g (Read c si p nextF) = Read c si p (g . nextF)
  fmap g (Run c cmd nextF)   = Run c cmd   (g . nextF)
```

Let's figure out how this works and why there are two ways to apply the `g` function passed to `fmap`:

```
(g next)
(g . nextF)
```

From the previous chapter, we know that a type is a `Functor` if we can apply some function `g` to its contents without changing the whole structure. The `fmap` function will apply `g` for us, so to make a type a `Functor`, we should define how the `fmap` function behaves. The Free monad uses the `fmap` function to nest actions in continuation fields we provide in our domain algebra. This is the main way to combine monadic operations (actions) in the Free monad. Every value constructor of our algebra should have a continuation field.

We have four value constructors encoding four domain operations (actions) in the `Procedure` type. The `Set` value constructor is rather simple:

```
data Procedure a
  = Set Controller Property Value a
```

It has four fields of type `Controller`, `Property`, `Value`, and a continuation field with a generic type `a`. This field should be mappable in the sense of `Functor`. This means that the `fmap` function should apply a generic `g` function to this continuation field:

```
fmap g (Set c p v next) = Set c p v (g next)
```

We call this field `next` because it should be interpreted next after the `Set` procedure.

In the last field, every value constructor has denoted the continuation. In other words, this is the action that should be evaluated next. Also, the action encoded by the `Set` value returns nothing useful. However, the actions encoded by the `Get`, `Read`, and `Run` values do return something useful, namely, the `Value`, `Measurement`, and `CommandResult` values, respectively. This is why the continuation fields differ. The `next` field is now has type (`someReturnType -> a`):

```
data Procedure a
  = Get Controller Property (Value -> a)
  | Run Controller Command (CommandResult -> a)
  | Read Controller SensorIndex Parameter (Measurement -> a)
```

Such continuation fields hold actions that know what to do with the value returned. When the Free monad combines two actions, it ensures that the value the first action returns is what the second action is awaiting as the argument. For example, when the `Get` value constructor is interpreted, it will return a value of type `Value`. The nested action should be of type (`Value -> something`) to be combined.

The `fmap` function uses this fact of nesting. It receives the `g` function and applies it to the mappable contents of this concrete value constructor:

```
fmap g (Get c p nextF) = let
  newNextF = g . nextF
  in Get c p newNextF
```

As shown, the application of function `g` to a regular value `next` is just (`g next`). The application of function `g` to a function `nextF` is a composition of them: `g . nextF`. We map function `g` over the single field and leave all other fields unchanged.

The trick of nesting continuations is exactly the same one we used in the previous version of the `Procedure` type, but we're now dealing with a better abstraction — the Free monad pattern. Strictly speaking, the Free monad pattern can handle returning values by keeping a continuation in the field with a function type; a continuation is nothing more than a function in the same monad that accepts a value of the input type and processes it.

The next step of the "free monadizing" algorithm is presented in listing 4.5. We define a synonym for the `Free` type and declare smart constructors.

Listing 4.5 Type for monadic eDSL and smart constructors

```
type ControllerScript a = Free Procedure a

-- Smart constructors:
set :: Controller -> Property -> Value -> ControllerScript ()
set c p v = Free (Set c p v (Pure ()))

get :: Controller -> Property -> ControllerScript Value
get c p = Free (Get c p Pure)

read :: Controller -> SensorIndex -> Parameter
     -> ControllerScript Measurement
read c si p = Free (Read c si p Pure)

run :: Controller -> Command -> ControllerScript CommandResult
run c cmd = Free (Run c cmd Pure)
```

These smart constructors wrap procedures into the monadic `ControllerScript a` type (the same as `Free Procedure a`). To be precise, they construct a monadic value in the Free monad parametrized by the `Procedure` functor. We can't directly compose value constructors `Get`,

`Set, Read`, and `Run` in the monadic scripts. The `Procedure a` type is not a monad, just a functor. But the `set` function and others make a composable combinator in the `ControllerScript` monad instead. This may all look monstrous, but it's actually not that hard — just meditate over the code and try to transform types one into another, starting from the definition of the `Free` type (we discussed it in the previous chapter):

```
data Free f a = Pure a
              | Free (f (Free f a))
```

You'll discover that the types `Free` and `Procedure` are now mutually nested in a smart, recursive way.

Notice the `Pure` value constructor in the smart constructors. It denotes the end of the monadic chain. You can put `Pure ()` into the `Set` value constructor, but you can't put it into the `Get`, `Read`, and `Run` value constructors. Why? You can infer the type of the `Get`'s continuation field, as we did in the previous chapter. It will be (`Value -> ControllerScript a`), while `Pure ()` has type `ControllerScript a`. You just need a function instead of a regular value to place it into a continuation field of this sort. The partially applied value `Pure :: a -> Free f a` is what you need. Compare this carefully:

```
set c p v = Free (Set c p v (Pure ()))
get c p = Free (Get c p Pure)

fmap g (Set c p v next) = Set  c p v (g next)
fmap g (Get c p nextF) = Get c p (g . nextF)
```

Whenever you write `Pure`, you may write `return` instead; they do the same thing:

```
set c p v = Free (Set c p v (return ()))
get c p = Free (Get c p return)
```

In the monad definition for the `Free` type, the `return` function is defined to be a partially applied `Pure` value constructor:

```
instance Functor f => Monad (Free f) where
  return = Pure

-- Monadic composition
  bind (Pure a) f = f a
  bind (Free m) f = Free ((`bind` f) <$> m)
```

Don't worry about the definition of the monadic bind. We don't need it in this book.

The sample script is presented in listing 4.6. Notice that it's composed from the `get` action and the `process` action. The `process` function works in the same monad `ControllerScript`, so it might be composed with other functions of the same type monadically.

Listing 4.6 Sample script in the eDSL

```
controller = Controller "test"
sensor = "thermometer 00:01"
version = Version
temperature = Temperature

-- Subroutine:
process :: Value -> ControllerScript String
process (StringValue "1.0") = do
  temp <- read controller sensor temperature
  return (show temp)
process (StringValue v) = return ("Not supported: " ++ v)
process _ = error "Value type mismatch."

-- Sample script:
script :: ControllerScript String
script = do
  v <- get controller version
  process v
```

Let's now develop a sample interpreter. After that, we'll consider how to hide the details of the language from the client code. Should the value constructors of the `Procedure` type be public? It seems that this isn't a must. If they are public, the user can interpret a language by pattern matching. Listing 4.7 shows how we roll out the structure of the `Free` type recursively and interpret the procedures

nested one inside another. The deeper a procedure lies, the later it will be processed, so the invariant of sequencing of the monadic actions is preserved.

Listing 4.7 Possible impure interpreter in the IO monad

```
interpret :: ControllerScript a -> IO a
interpret (Pure a) = return a
interpret (Free (Set c p v next)) = do          #A
  print ("Set", c, v, p)
  interpret next                                #B
interpret (Free (Get c p nextF)) = do           #C
  print ("Get", c, p)
  interpret (nextF (StringValue "1.0"))         #D
interpret (Free (Read c si p nextF)) = do
  print ("Read", c, si, p)
  interpret (nextF (toKelvin 1.1))              #E
interpret (Free (Run c cmd nextF)) = do
  print ("Run", c, cmd)
  interpret (nextF (Right "OK."))               #E

#A next keeps the action to be interpreted next
#B Continue interpreting
#C nextF keeps the function that's awaiting a value as argument
#D Continue interpreting after the nextF action
   has received the value. Also, returns a dummy value
#E Dummy values are returned here
```

It's not a bad thing in this case, but does the interpreter provider want to know about the `Free` type and how to decompose it with pattern matching? Do they want to do recursive calls? Can we facilitate their life here? Yes, we can. This is the theme of the next section.

4.3.3 The abstract interpreter pattern

To conceal our language's Free monad nature, to hide explicit recursion, and to make interpreters clean and robust, we need to abstract the whole interpretation process behind an interface — but this interface shouldn't restrict you in writing interpreters. It's more likely you'll wrap the interpreters into some monad. For instance, you can store operational data in the local state (the `State` monad) or immediately print values to the console during the process (the `IO` monad).

Consequently, our interface should have the same expressiveness. The pattern we'll adopt here has the name "the abstract interpreter."

The pattern has two parts: the functional interface to the abstract interpreter you should implement and the base `interpret` function that calls the methods of the interface while the `Free` type is recursively decomposed. Let's start with the former. It will be a specific type class (see listing 4.8).

Listing 4.8 Interface to abstract interpreter

```
class Monad m => Interpreter m where
  onSet  :: Controller -> Property -> Value -> m ()
  onGet  :: Controller -> Property -> m Value
  onRead :: Controller -> SensorIndex
         -> Parameter -> m Measurement
  onRun  :: Controller -> Command -> m CommandResult
```

The constraint `Monad` for type variable `m` that you can see in the class definition says that every method of the type class should operate in some monad `m`. This obligates the instance of the interface to be monadic; we allow the client code to engage the power of monads, but we don't dictate any concrete monad. What monad to choose is up to you, depending on your current tasks. The type class doesn't have any references to our language; no value constructor of the `Procedure` type is present there. How will it work, then? Patience — we need one more part: the template interpreting function. It's very similar to the interpreting function in listing 4.7, except it calls the methods that the type class `Interpreter` yields. The following listing demonstrates this code.

Listing 4.9 Abstract interpreting function for the Free eDSL

```
module Andromeda.LogicControl.Language
  ( interpret
  , Interpreter(..)
  , ControllerScript
  , get
  , set
  , read
  , run
  ) where

{- here the content of listings 4.3, 4.4, 4.5 goes -}
```

```
interpret
  :: Monad m
  => Interpreter m
  => ControllerScript a
  -> m a
interpret (Pure a) = return a
interpret (Free (Get c p next)) = do
  v <- onGet c p
  interpret (next v)
interpret (Free (Set c p v next)) = do
  onSet c p v
  interpret next
interpret (Free (Read c si p next)) = do
  v <- onRead c si p
  interpret (next v)
interpret (Free (Run c cmd next)) = do
  v <- onRun c cmd
  interpret (next v)
```

We hide the `Procedure` type, but we export the function `interpret`, the type class `Interpreter`, the type `ControllerScript`, and the smart constructors. What does this mean? Imagine you take someone's weird library. You want to construct a script and interpret it too. The first task is easily achievable. Let's say you've written the script like in listing 4.6. Now you try to write an interpreter like in listing 4.7, but you can't because no value constructors are available outside the library. But you notice the `interpret` function, which requires the type class `Interpreter` to be instantiated. This is the only way to interpret your `Free` script into something real. You should have a parameterized type to make this type an instance of the `Interpreter` type class. The type should also be an instance of a monad. Suppose you're building an exact copy of the interpreter in listing 4.7. You should adopt the `IO` monad then. The code you'll probably write may look like this:

```
import Andromeda.LogicControl.Language

instance Interpreter IO where
  onSet c prop v = print ("Set", c, v, prop)
  onGet c prop = do
    print ("Get", c, prop)
    return (StringValue "1.0")
  onRead c si par = do
```

```
    print ("Read", c, si, par)
    return (toKelvin 1.1)
  onRun c cmd = do
    print ("Run", c, cmd)
    return (Right "OK.")
```

After this, you interpret the script in listing 4.6:

```
interpret script
```

The result:

```
("Get",Controller "test",Version)
("Read",Controller "test","thermometer 00:01",Temperature)
"Measurement (FloatValue 1.1)
```

Hiding the implementation details will force the developer to implement the type class. This is a functional interface to our subsystem. The functional interface to the language itself is now accompanied by the functional interface to the interpreter. The idea of this pattern is very close to the idea of the object-oriented Visitor pattern. But the functional pattern is better because it can restrict all impure and mutable operations by prohibiting the `IO` monad.

Other interpreters could be implemented inside other monads — for example, `State` or `State` + `IO`. A consequence of this design is that it keeps our interpreters consistent automatically because when the language gets another procedure, the `Interpreter` type class will be updated, and we'll get a compilation error until we update our instances. So, we can't forget to implement a new method, in contrast to the previous design, where the `Free` language was visible for prying eyes.

It's perfect, wouldn’t you agree?

4.3.4 Free language of Free languages

Scripts we can write with the `ControllerScript` language cover only a small part of the domain, while the requirements say we need to operate with a database, raise alarms if needed, run calculations, and do other things to control the ship. These “subdomains” should be somewhat independent from each other because we don't want a mess like we saw in the “naive” language. Later, we will probably add some capabilities for user input/output communication — this

will be another declarative language that we can embed into our focused domain languages without too much trouble. There's a chance that you may find the approach I suggest in this section too heavy, but it gives us some freedom to implement domain logic partially, prove it correct with concise tests, and move on. When the application's skeleton is designed well, we can return and complete the logic, providing the missing functionality. That is, such a design lets us stay on the top level, which is good in the early stages of development.

Let's discuss this design approach — a pointer of pointers to arrays of pointers to foo! Oh, sorry, wrong book ... I meant a Free language of Free languages!

The idea is to have several small languages to cover separate parts of the domain. Each language is intended to communicate with some subsystem, but not directly, because every language here is a Free eDSL. Remember, we make this "letter of intent" act using the interpretation process. We can even say the compilation stage is our functional analogue of "late binding" from OOP. Our eDSLs are highly declarative, easily constructible, and interpretable. That's why we have another big advantage: the ease of translating our scripts to an external language and back again (you'll see this in the corresponding section of this chapter). This would be very difficult to do otherwise.

Check out the elements diagram for Logic Control (figure 2.13) and the architecture diagram (figure 2.15): this approach was born there, but it was a little cryptic and had inaccurate naming. We'll adopt these names for the languages:

- `ControllerScript` — the Free eDSL for communicating with the Hardware subsystem
- `InfrastructureScript` — the Free eDSL for logging, authorization, filesystem access, operating system calls, raising events, and so on (or maybe we should partition these responsibilities?)
- `ComputationScript` — the eDSL for mathematical computations (not a Free language, possibly)
- `DataAccessScript` — the Free eDSL for operating with the database
- `ControlProgram` — the Free eDSL that allows us to run any of the scripts and provides reactive capabilities (we'll return to this in the next

chapters)

Figure 4.1 illustrates the whole design.

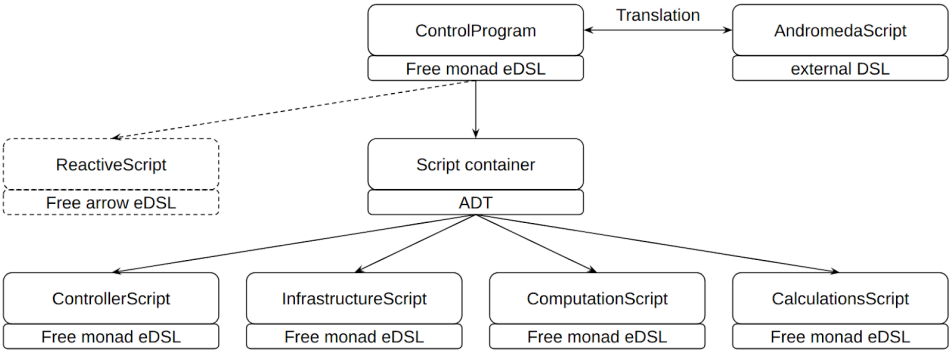


Figure 4.1 Design of a domain model for Andromeda scripting

Pictures are good; code is better. Let's get familiar with the `InfrastructureScript` DSL (see listing 4.10).

Listing 4.10 The InfrastructureScript free DSL

```
{-# LANGUAGE DeriveFunctor #-}                                    #1

module Andromeda.LogicControl.Language.Infrastructure

-- Dummy types, should be designed later.
type ValueSource = String
type Receiver = Value -> IO ()

data Action a = StoreReading Reading a
              | SendTo Receiver Value a
              | GetCurrentTime (Time -> a)
  deriving (Functor)                                             #2

type InfrastructureScript a = Free Action a

storeReading :: Reading -> InfrastructureScript ()
sendTo :: Receiver -> Value -> InfrastructureScript ()
logMsg :: String -> InfrastructureScript ()
alarm :: String -> InfrastructureScript ()
```

```
getCurrentTime :: InfrastructureScript Time

#1 Useful Haskell language extension
#2 The automatic instantiation of a Functor
   type class in Haskell
```

Notice that we went by a short path: we automatically derived a `Functor` instance for the `Action` type (#2). No more annoying `fmap` definitions! We're too lazy to do all this boilerplate by hand. This only works with the `DeriveFunctor` extension enabled (#1).

Listing 4.11 displays the `Script` ADT, which ties many languages together.

Listing 4.11 The Script container

```
data Script b
  = ControllerScriptDef (ControllerScript b)
  | ComputationScriptDef (ComputationScript b)
  | InfrastructureScriptDef (InfrastructureScript b)
  | DataAccessScriptDef (DataAccessScript b)
```

The `b` type denotes something a script returns. There aren't any restrictions to what `b` should be. We may want to return value of any type. The following two scripts return different types:

```
startBoosters :: ControllerScript CommandResult
startBoosters = run (Controller "boosters") (Command "start")

startBoostersScript :: Script CommandResult
startBoostersScript = ControllerScriptDef startBoosters

getTomorrowTime :: InfrastructureScript Time
getTomorrowTime = do
  time <- getCurrentTime
  return (time + 60*60*24)

getTomorrowTimeScript :: Script Time
getTomorrowTimeScript = InfrastructureScriptDef getTomorrowTime
```

The problem here is that the types of the two "wrapped" functions `startBoostersScript` and `getTomorrowTimeScript` don't match. `Script Time` is not equal to `Script CommandResult`. This

means that we have two different containers; what can we do with them? Suppose we want to unify all these scripts in one top-level, composable Free eDSL as intended in figure 4.1. Consider the first try:

```
-- Top-level Free language
data Control b a = EvalScript(Script b) a
```

This is the algebra that should store a script container. The container's type is parametrized, so we provide a type argument for `Script b` in the type constructor `Control b`. We know this type will be a functor. There should be a field of another type `a` that the `fmap` function will be mapped over. Therefore, we add this type parameter to the type constructor: `Control b a`. However, we forgot a value of the `b` type that a script will return. According to the Free monad pattern, the value must be passed to nested actions. The continuation field should be of function type

```
data Control b a = EvalScript(Script b) (b -> a)
```

So far, so good. Let's define a `Functor` with the `b` type variable frozen because it should be a functor of the single type variable `a`:

```
instance Functor (Control b) where
  fmap f (EvalScript scr g) = EvalScript scr (f . g)
```

Finally, a `Free` type with a smart constructor:

```
type ControlProgram b a = Free (Control b) a

evalScript :: Script b -> ControlProgram b a
evalScript scr = Free (EvalScript scr Pure)
```

It's pretty good, except it won't compile. Why? Is that fair? We did all that we usually do. We did it right. But the compiler cant’ match the two type variables `b` from the `evalScript`. It can't be sure they are equal for some weird reason. However, if you try the following definition, it will compile:

```
evalScript :: Script a -> ControlProgram a a
evalScript scr = Free (EvalScript scr Pure)
```

But it's all wrong because the type variable of the `Script` can't be specialized more than once. Consider the following script that should be of “quantum” type:

```
unifiedScript :: ControlProgram ??? (CommandResult, Time)
unifiedScript = do
  time <- evalScript getTomorrowTimeScript
  result <- evalScript startBoostersScript
  return (result, time)
```

Why quantum? Because two scripts in this monad have return types `CommandResult` and `Time`. But how do we say this in the type definition instead of using three question marks? Definitely, the `b` type variable takes two possible types in quantum superposition. I believe you can do so in some imaginary universe, but here, quantum types are prohibited. The type variable `b` must take either `CommandResult` or the `Time` type. But this completely ruins the idea of a `Free` language over `Free` languages. In dynamically typed languages, this situation is easily avoided. Dynamic languages do have quantum types! Does that mean statically typed languages are defective?

Fortunately, no, it doesn't. We just need to summon the type-level tricks and explain to the compiler what we actually want. The right decision here is to hide the `b` type variable behind the scenes. Look at the `unifiedScript` and the `ControlProgram` type again: do you want to carry `b` everywhere? I don't think so. This type variable denotes the return type from script. When you call a script, you get a value. Then you pass that value to the continuation. Consequently, the only place this type exists is localized between the script itself and the continuation. The following code describes this situation:

```
{-# LANGUAGE ExistentialQuantification #-}
data Control a = forall b. EvalScript (Script b) (b -> a)
```

As you can see, the `b` type variable isn't presented in the `Control` type constructor. No one who uses this type will ever know that there’s an internal type `b`. To declare that it's internal, we write the `forall` quantificator. Doing so, we defined the scope for the `b` type. It's bounded by the `EvalScript` value constructor (because the `forall` keyword stays right before it). We can use the `b` type variable inside the value constructor, but it's completely invisible from the outside. All it does inside is show that the `b` type from a script `(Script b)` is the same type as the argument of continuation `(b -> a)`. It doesn't matter what the `b` type actually is. Anything. All you want. It says to the compiler, just put a script and an action of the same type into the `EvalScript` value constructor and don't accept two artifacts of different

types. The action will decide by itself what to do with the value the script returns. One more note: this is all possible due to Haskell's Existential Quantification extension.

The complete design of the `ControlProgram` free language is shown in listing 4.12.

Listing 4.12. The ControlProgram Free eDSL

```
data Script b
  = ControllerScript (ControllerScript b)
  | ComputationScript (ComputationScript b)
  | InfrastructureScript (InfrastructureScript b)
  | DataAccessScript (DataAccessScript b)

-- Smart constructors:
infrastructureScript :: InfrastructureScript b -> Script b
infrastructureScript = InfrastructureScriptDef

controllerScript :: ControllerScript b -> Script b
controllerScript = ControllerScriptDef

computationScript :: ComputationScript b -> Script b
computationScript = ComputationScriptDef

dataAccessScript :: DataAccessScript b -> Script b
dataAccessScript = DataAccessScriptDef

-- Top-level eDSL. It should be a Functor
data Control a = forall b. EvalScript (Script b) (b -> a)

instance Functor Control where
  fmap f (EvalScript scr g) = EvalScript scr (f . g)

-- Top-level Free language
type ControlProgram a = Free Control a

-- Smart constructor
evalScript :: Script a -> ControlProgram a
evalScript scr = Free (EvalScript scr Pure)

-- Sample script
unifiedScript :: ControlProgram (CommandResult, Time)
```

```
unifiedScript = do
  time <- evalScript getTomorrowTimeScript
  result <- evalScript startBoostersScript
  return (result, time)
```

Notice that the smart constructors are added for the `Script` type (`infrastructureScript` and others). Smart constructors make life much easier.

As usual, in the final stage of the Free language development, you create an abstract interpreter for the `ControlProgram` language. Conceptually, the abstract interpreter has the same structure: the `Interpreter` type class and the base function `interpret`.

```
class Monad m => Interpreter m where
  onEvalScript :: Script b -> m b

interpret
  :: Monad m
  => Interpreter m
  => ControlProgram a
  -> m a
interpret (Pure a) = return a
interpret (Free (EvalScript s nextF)) = do
  v <- onEvalScript s
  interpret (nextF v)
```

When someone implements the `Interpreter` class type, he should call `interpret` functions for nested languages. The implementation may look like this:

```
module InterpreterInstance where

import qualified ControllerDSL as C
import qualified InfrastructureDSL as I
import qualified DataAccessDSL as DA
import qualified ComputationDSL as Comp

interpretScript (ControllerScriptDef scr) = C.interpret scr
interpretScript (InfrastructureScriptDef scr) = I.interpret scr
interpretScript (ComputationScriptDef scr) = DA.interpret scr
interpretScript (DataAccessScriptDef scr) = Comp.interpret scr
```

```
instance Interpreter IO where
  onEvalScript scr = interpretScript scr
```

The `Control` type now has only one field that's a declaration to evaluate one of the scripts available, but in the future, we can extend it to support, for example, the declaration of a reactive model for FRP. It's an interesting possibility, but not that simple. Stick with it, we go further!

4.3.5 Arrows for eDSLs

Are you tired of learning about complex concepts? Take a break and grab some coffee. Now let's look more closely at the concept of arrows.

The arrow is just a generalization of the monad, which is just a monoid in the category of ... oh, forget it. If you've never met functional arrows before, I'll try to give you a little background on them, but with a light touch, because our goal differs: we would do better to form an intuition of when and why the arrowized language is an appropriate domain representation than to learn how to grok arrows. For more information, consider consulting some external resources; there are many of them for Haskell and Scala.

You should be motivated to choose an arrowized language when you have

- Computations that are like electrical circuits: there are many transforming functions ("radio elements") connected by logical links ("wires") into one big graph ("circuit").
- Time-continuing and time-varying processes, transformations, and calculations that depend on the results of one another.
- Computations that should be run by time condition: periodically, once at the given time, many times during the given period, and so forth.
- Parallel and distributed computations.
- Computation flow (data flow) with some context or effect; the need for a combinatorial language in addition to or instead of a monadic language.

It's not by chance that, being a concept that abstracts a flow of computations, an arrowized script is representable as a flow diagram. Like monads, the arrowized computations can depend on context that's hidden in the background of an arrow's mechanism and thus completely invisible to the developer. It's easy to convert a monadic function `f :: a -> m b` into the arrow `arr1 ::`

`MyArrow a b`, preserving all the goodness of a monad `m` during `arr1` evaluation. This is how the concept of arrows generalizes the concept of monads. It's even easier to create an arrow `arr2 :: MyArrow b c` from just a non-monadic function `g :: b -> c`. This is how the concept of arrows generalizes the function type.

Finally, when you have two arrows, it's not a big deal to chain them together:

```
arr1 :: MyArrow a b
arr1 = makeMonadicArrow f

arr2 :: MyArrow b c
arr2 = makePureArrow g

arr3 :: MyArrow a c
arr3 = arr1 >>> arr2
```

All we should know to compose arrows is their type: the first arrow converts values of type `a` to values of type `b`, and the second arrow converts values of type `b` to values of type `c`. That is, these two arrows both convert values by the scheme `(a -> b) -> c`. Applied to a value of type `a`, the arrow `arr3` will first do `f a` with monadic effect, resulting in a value of type `m b`, and will then evaluate `g b`, resulting in a value of type `c` (or `m c` — it depends on the concrete monad you use inside the arrow). In short, if `runArrow` is the application of your arrow to an argument, then `runArrow arr3 a` may be equivalent to this:

```
-- not escaped from monad
apply :: (a -> m b) -> (b -> c) -> m c
apply f g a = do
  b <- f a
  let c = g b
  return c
```

or to this:

```
-- escaped from monad
apply :: (a -> m b) -> (b -> c) -> c
apply f g a =   let b = runMonad f a
                    c = g b
                in c
```

That's how the `(>>>)` combinator works: it applies the left arrow and then the right one. And it's aware of the arrow's internals, so it may run a monad for a monadically composed arrow. This operation is associative:

```
arr1 >>> arr2 >>> arr3
```

To apply an arrow to a value, you call a "run" function (`runArrow`) from a library:

```
toStringA :: MyArrow Int String
toStringA = arr show

evaluateScenario = do
  result <- runArrow toStringA 10
  print result
```

The `arr` function should be defined for every arrow because it's present in the `Arrow` type class (ignore the `Category` type class for now):

```
class Category a => Arrow a where
  arr :: (b -> c) -> a b c
  first :: a b c -> a (b,d) (c,d)
```

You might have noticed that when looking at arrow types, you often can't conclude whether there's a monadic effect or not. For example, what monad is hidden under the imaginary arrow `WillItHangArrow Program Bool`? Is there a kind of `IO` monad? Or maybe the `State` monad is embedded there? You'll never know unless you open its source code. Is that bad or good? Hard to say. We went to the next level of abstraction, and we can even cipher different effects in one computation flow by switching between different arrows. But the purity rule works anyway: if a particular arrow is made with IO inside, you'll be forced to run that arrow in the `IO` monad.

```
ioActionArrow :: MyIOArrow () ()
ioActionArrow =
  makeMonadicArrow (\_ -> putStrLn "Hello, World!")

-- Fine:
main :: IO ()
main = runMyIOArrow ioActionArrow ()
```

```
-- Won't compile:
pureFunction :: Int -> Int
pureFunction n = runMyIOArrow ioActionArrow ()
```

The arrows composed only with the sequential combinator `(>>>)` look quite boring in the flow diagram (see figure 4.2).

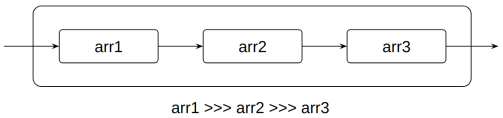


Figure 4.2 Sequential flow diagram

Certainly, we aren't limited to sequential composition only. As usual, if we realize that our domain model can fit into an arrowized language, we can take advantage of all the combinators the arrow library provides. We can choose from a wide range of arrow combinators to make our computational networks much more interesting: parallel execution of arrows, splitting the flow into several flows and merging several flows into one, looping the computation, and conditional evaluation are all supported.

We'll construct an arrowized interface over the `Free` languages in the Logic Control subsystem, so you can add the flow graph to your toolbox for domain modeling. But before we do that, consider the mnemonic arrow notation. The arrow `arrowA` that accepts the `input` value and returns the `output` value is written as

```
output <- arrowA -< input
```

Because every arrow is a generalization of a function, it should have input and output, but we can always pass the unit value `()` if the arrow doesn't actually need this:

```
-- no input, no output:
() <- setA ("PATH", "/home/user") -< ()
```

If two arrows depend on the same input, they can be run in parallel. Whether it’s a real parallelization or just a logical possibility depends on the arrow's

mechanism. You can construct an arrow type that will run these two expressions concurrently:

```
factorial <- factorialA -< n
fibonacci <- fibonacciA -< n
```

Arrows can take and return compound results. The most important structure for arrows is a pair. In the arrow machinery, a pair is split to feed two arrows with that pair's parts (see the following split (`***`) operator). You may write an arrow that will change only the first or the second item of a pair. For example, the following two arrows take either `n` or `m` to calculate a factorial but leave another value unchanged:

```
(factorialN, m) <- first  factorialA -< (n, m)
(n, factorialM) <- second factorialA -< (n, m)
```

The combinators `first` and `second` should be defined for every arrow, as well as the `(>>>)` combinator and others. The fanout `(&&&)` combinator makes an arrow from two arrows, running them in parallel with the input argument cloned. The output will be a pair of results from the first and second arrows:

```
(factorial, fibonacci) <- factorialA &&& fibonacciA -< n
```

The split `(***)` combinator behaves like the `(&&&)` combinator but takes a pair of inputs for each of two arrows it combines:

```
(factorialN, fibonacciM) <- factorialA *** fibonacciA -< (n, m)
```

Figure 4.3 illustrates these combinators as input/output boxes.

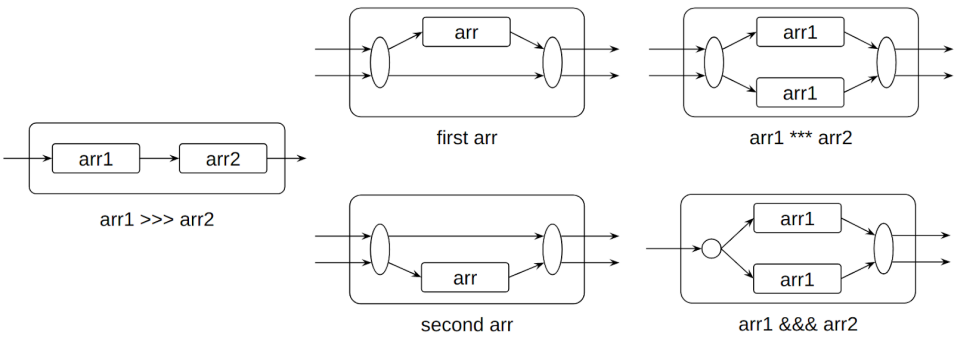


Figure 4.3 Arrow combinators

TIP Some self-descriptiveness can be achieved with a "conveyor belt diagram," where arrows associate with machines, and the belt supplies them with values to be processed. The tutorial "Haskell/Understanding arrows" uses this metaphor (see visualization) and gives a broad introduction to arrows.

LINK *Understanding arrows*
https://en.wikibooks.org/wiki/Haskell/Understanding_arrows

4.3.6 Arrowized eDSL over Free eDSLs

Let's take the last mnemonic monitoring scenario and reformulate it in the arrowized way. Meet the arrow that monitors readings from the thermometer:

```
Scenario: monitor outside thermometer temperature
Given: outside thermometer @therm
Evaluation: once a second, run arrow thermMonitorA(@therm)

arrow thermMonitorA [In: @therm, Out: (@time, @therm, @tempK)]
  @tempC <- thermTemperatureA -< @therm
  @tempK <- toKelvinA         -< @tempC
  @time  <- getTimeA          -< ()
  ()     <- processReadingA   -< (@time, @therm, @tempK)
  return (@time, @therm, @tempK)
```

It calls other arrows to make transformations and to call scripts from `Free` domain languages. The `thermTemperatureA` arrow reads the temperature:

```
arrow thermTemperatureA [In: @therm, Out: @tempC]
  @tempC <- runScriptA -< thermTemperatureS(@therm)
  return @tempC
```

Arrows that store readings, validate temperatures, and raise alarms when problems are detected are combined in the `processReadingA` arrow:

```
arrow processReadingA [In: (@time, @therm, @tempK),  Out: ()]
  ()      <- storeReadingA    -< @reading
  @result <- validateReadingA -< @reading
  ()      <- alarmOnFailA     -< @result
  return ()
```

We could define other arrows, but I think it's now obvious how they describe scenarios mnemonically. The full computation is better shown by a flow diagram (see figure 4.4).

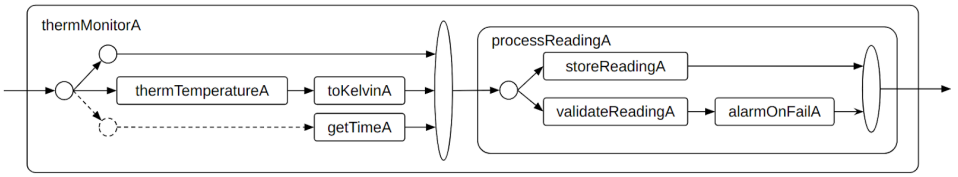


Figure 4.4 Flow diagram for thermometer monitoring arrow

If the mnemonic arrow notation and the computational graph have scared you a little, you haven't seen the combinatorial code yet! There are several ways to compose an arrow for the diagram, and one of them is to make the calculation process completely sequential. In the following code, many arrows are combined to transform results from each other sequentially:

```
thermMonitorA
    = (arr $ \b -> (b, b))
  >>> second (thermTemperatureA >>> toKelvinA)
  >>> (arr $ \x -> ((), x))
  >>> first getTimeA
  >>> (arr $ \(t, (inst, m)) -> (t, inst, m))
  >>> (arr $ \b -> (b, b))
  >>> second (storeReadingA &&& validateReadingA)
  >>> second (second alarmOnFailA)
  >>> (arr $ \(b, _) -> b)
```

What is the code here? It looks very cryptic, like Perl or a paragraph from a math paper. This is actually valid Haskell code that you may freely skip. It's here for those who want a full set of examples, but you may be impressed that Haskell has a nicer `proc` notation for arrows that's very close to the mnemonic notation I've introduced. Before I show it to you, let's prepare the arrow type. Suppose we've constructed a `FlowArr b c` arrow type somehow that can describe the diagram in figure 4.4. This arrow is specially designed to wrap our free languages and scenarios. It doesn't have any of its own logic, it only provides an arrowized interface to the languages. You may or may not use it, depending on your taste.

As the mnemonic scenario says, the `thermMonitorA` arrow takes an instance of thermometer (let it be of type `SensorInstance`) and returns a single reading of type `Reading` from it:

```
thermMonitorA :: FlowArr SensorInstance Reading
thermMonitorA = proc sensorInst -> do
  tempK <- toKelvinA <<< thermTemperatureA -< sensorInst
  time  <- getTimeA -< ()

  let reading = (time, sensorInst, tempK)

  ()      <- storeReadingA    -< reading
  result  <- validateReadingA -< reading
  ()      <- alarmOnFailA     -< result
  returnA -< reading
```

The `proc` keyword opens a special syntax for arrow definition. The variable `sensorInst` is the input argument. The arrowized `do` block, which is

extended compared to the monadic `do` block, defines the arrow's body. At the end, the `returnA` function should be called to pass the result out.

> **TIP** To enable the `proc` notation in a Haskell module, you should set the compiler pragma `Arrows` at the top of the source file: `{-# LANGUAGE Arrows #-}`. It's disabled by default due to nonstandard syntax.

Here's the definition of the `FlowArr` arrow type:

```
type FlowArr b c = ArrEffFree Control b c
```

This type denotes an arrow that receives `b` and returns `c`. The `ArrEffFree` type, which we specialize by our top-level eDSL type `Control`, came from the special library I designed for the demonstration of the Free Arrow concept. This library has a kind of stream transformer arrow wrapping the Free monad. Sounds menacing to our calm, but we won't discuss the details here. All you need from that library now is the `runFreeArr`. Remember, we were speaking about whether you should interpret a Free language if you want to run it in a real environment. This is the same for the arrowized interface over a Free language. To run an arrow, you pass exactly the same interpreter to it:

```
sensorInst = (Controller "boosters", "00:01")
test = runFreeArr interpretControlProgram
                  thermMonitorA
                  sensorInst
```

Here, `interpretControlProgram` is an interpreting function for the `ControlProgram` language, `thermMonitorA` is the arrow to run, and `sensorInst` is the value the arrow is awaiting as input. Running the arrow calls the interpreter for the top-level language, and the internal language interpreters will be called from it. We'll omit this code. What we'll see is the implementation of combinators.

"Run script X" arrows are simple — we just wrap every monadic action with the library arrow creator `mArr` for effective (monadic) functions:

```
thermTemperatureA :: FlowArr SensorInstance Measurement
thermTemperatureA = mArr f
  where
    f inst :: SensorInstance -> ControlProgram Measurement
    f inst = evalScript (readSensor Temperature inst)
```

Thus, `f` is the monadic function in the `ControlProgram` monad. It's a composition of two functions: the `evalScript` function and `readSensor`, the custom function defined like so:

```
readSensor :: Parameter -> SensorInstance -> Script Measurement
readSensor parameter (cont, idx) = controllerScript readSensor'
  where
    -- The script itself.
    -- "read" is a smart constructor.
    readSensor' :: ControllerScript Measurement
    readSensor' = read cont idx parameter
```

Figure 4.5 shows the structure of nested scripts. Top blocks are functions, bottom blocks are types.

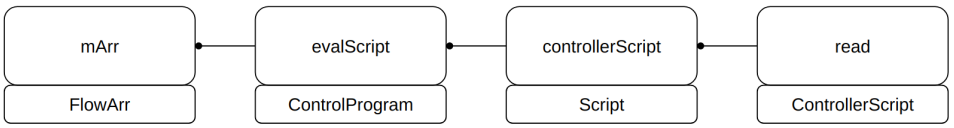


Figure 4.5 Scripts nesting

The `readSensor` function puts the script in the `ControllerScript` monad into an intermediate `Script` container, the form the `evalScript` function is awaiting as input (see listing 4.12):

```
data Control a = forall b. EvalScript (Script b) (b -> a)

evalScript :: Script a -> ControlProgram a
evalScript scr = Free (EvalScript scr Pure)
```

We do the same with the infrastructure script and others:

```
getTimeA :: FlowArr b Time
getTimeA = mArr f
  where
    f :: b -> ControlProgram Time
    f _ = evalScript (infrastructureScript getCurrentTime)

storeReadingA :: FlowArr Reading ()
storeReadingA = mArr (evalScript . infrastructureScript .
storeReading)
```

We also convert pure functions to arrows with the library wrapper `arr`:

```
validateReadingA :: FlowArr Reading ValidationResult
validateReadingA = arr validateReading

validateReading :: Reading -> ValidationResult
validateReading (_, si, Measurement (FloatValue tempK)) = ...
```

Finally, when the arrowized language is filled with different arrows, we can write comprehensive scenarios in a combinatorial way — not just with monads!

Let's weigh the pros and cons. The arrowized eDSL is good for a few reasons:

- *It’s useful for flow diagrams*. This is a natural way to express flow scenarios to control the ship.
- *It’s highly combinatorial and abstract*. As a result, you write less code. You don't even need to know what monad is running under the hood.

However, arrowized eDSLs have disadvantages too:

- *Arrows don't get the consideration they deserve and Free Arrows are not investigated properly*. This is a cutting-edge field of computer science, and there aren’t many industrial applications of arrows.
- *They’re harder to create*. We made a simple arrowized language over `Free` languages, but the library this language is built on is complex.
- *Combinators for arrows such as `(&&&)` or `(>>>)` may blow your mind.* Besides combinators, how many languages have a special arrowized syntax? Only Haskell, unfortunately.

Arrows are quite interesting. There are combinators for choice evaluation of arrows for looped and recursive flows, which makes arrows an excellent choice

for complex calculation graphs — for example, mathematical iterative formulas or electrical circuits. Some useful applications of arrows include effective arrowized parsers, XML processing tools, and FRP libraries. It’s also important to note that the arrow concept, while being a functional idiom, has laws every arrow should obey. We won't enumerate them here, so as not to get bogged down in details, but will rather stay on the design level.

4.4 External DSLs

Every SCADA application has to support external scripting. This is a main functional requirement because SCADA is an environment that should be carefully configured and programmed for a particular industrial process. Scripting functionality may include

- *Compilable scripts in programming languages such as C*. You write program code in a real language and then load it into a SCADA application directly or compile it into pluggable binaries.
- *Pluggable binaries with scripts*. After the binaries are plugged into the application, you have additional scripts in your toolbox.
- *Interpretable external DSLs*. You write scripts using an external scripting language provided by the application. You may save your scripts in text files and then load them into the application.

Perhaps this is the most difficult part of the domain. All industrial SCADA systems are supplied with their own scripting language and integrated development environment (IDE) to write control code. They can also be powered by such tools as graphical editors, project designers, and code analyzers. All these things are about programming language compilation. Compilers, interpreters, translators, various grammars, parsing, code optimization, graph and tree algorithms, type theory, paradigms, memory management, code generation, and so on and so forth. This part of computer science is really big and hard.

We certainly don't want to be roped into compilation theory and practice: this book is not long enough to discuss even a little part of it! We'll try to stay on top of the design discussion and investigate the place and structure of an external eDSL in our application.

4.4.1 External DSL structure

First, we'll fix the requirement: Andromeda software should have an external representation of the Logic Control eDSLs with additional programming language possibilities. The internal representation, namely, *ControlProgram Free eDSL*, can be used in unit and functional tests of subsystems, while the external representation is for real scripts that a spaceship will be controlled by. We call this external DSL *AndromedaScript*. The engineer should be able to load, save, and run AndromedaScript files. The mapping of eDSLs to AndromedaScript isn’t symmetric:

- Embedded Logic Control DSLs can be fully translated to AndromedaScript.
- AndromedaScript can be partially translated into the Logic Control eDSLs.

AndromedaScript should contain the possibility of common programming languages. This code can't be translated to eDSLs because we don't have such notions there: neither Free eDSL of Logic Control contains `if-then-else` blocks, variables, constants, and so on. This part of AndromedaScript will be transformed into the intermediate structures and then interpreted as desired.

Table 4.2 describes all the main characteristics of AndromedaScript.

Table 4.2 Main characteristics of AndromedaScript

Characteristic	Description
Semantics	AndromedaScript is strict and imperative. Every instruction should be defined in a separate line. All variables are immutable. Delimiters aren't provided.
Code blocks	Code blocks should be organized by syntactic indentation with four spaces.
Type system	Implicit dynamic type system. Type correctness will be partially checked in the translation phase.

Supported constructions	`if-then-else`, range `for` loop, procedures and functions, immutable variables, custom lightweight ADTs.
Base library	Predefined data types, ADTs, procedures, mapping to the Logic Control eDSLs, mapping to the Hardware eDSLs.
Versions	Irrelevant at the start of the Andromeda project; only a single version is supported. Version policy will be reworked in the future when the syntax is stabilized.

The next big deal is to create a syntax for the language and write code examples. The examples can be meaningless, but they should show all the possibilities in pieces. We'll use them to test the translator. See the sample code in listing 4.13.

Listing 4.13 Example of AndromedaScript

```
val boosters = Controller ("boosters")
val start = Command ("start")
val stop = Command ("stop")
val success = Right ("OK")

// This procedure uses the ControllerScript possibilities.
[ControllerScript] BoostersOnOffProgram:
  val result1 = Run (boosters, start)
  if (result1 == success) then
    LogInfo ("boosters start success.")
    val result2 = Run (boosters, Command ("stop"))
    if (result2 == success) then
      LogInfo ("boosters stop success.")
    else
      LogError ("boosters stop failed.")
  else
    LogError ("boosters start failed.")

// Script entry point.
// May be absent if it's just a library of scripts.
Main:
```

```
LogInfo ("script is started.")
BoostersOnOffProgram
LogInfo ("script is finished.")
```

You might notice that there is no distinction between predefined value constructors such as `Controller` or `Command` and procedure calls such as `Run`. In Haskell, every value constructor of an ADT is a function that creates a value of this type — it's no different from a regular function. Knowing this, we simplify the language syntax by making every procedure and function a kind of value constructor. Unlike the Haskell syntax, arguments are comma-separated and bracketed because it's easier to parse.

What parts should a typical compiler contain? Let's enumerate them:

- *Grammar description*. Usually a BNF for simple grammars, but it can be a syntax diagram or a set of grammar rules over the alphabet.
- *Parser*. Code that translates the text of a program into the internal representation, usually an AST. Parsing can consist of lexical and syntax analysis before the AST creation. The approaches for how to parse a certain language depend highly on the properties of its syntax and requirements of the compiler.
- *Translator, compiler, or interpreter*. Code that translates one representation of the program into another. Translators and compilers use translation rules to manipulate ASTs and intermediate structures.

The grammar of the AndromedaScript language is context-free, so it can be described by the BNF notation. We won't have it as a separate artifact because we'll be using the monadic parsing library, so the BNF will take the form of parser combinators. There’s no need to verify the correctness of the BNF description: if the parser works right, then the syntax is correct. In our implementation, the parser will read text and translate the code into the AST, skipping any intermediate representation. There should be a translator that can translate a relevant part of the AST into the `ControlProgram` eDSL and an interpreter translator that does the opposite transformation. Why? Because it's our interface to the Logic Control subsystem, and we assume we have all the machinery that connects the `ControlProgram` eDSL with real hardware and other subsystems. The rest of the AndromedaScript code will be evaluated by the AST interpreter.

The project structure is updated with a new top-level subsystem, `Andromeda.Language`:

```
Andromeda\
  Language                  #A
  Language\
    External\               #B
      AST
      Parser
      Translator
      Interpreter
```

```
#A Top-level module that reexports modules of the compiler
#B AST, operational data structures, parsers,
   and translators of the AndromedaScript language
```

In the future, the external language will be complemented by the foreign programming language C, and we'll place that stuff into the folder *Andromeda\Language\Foreign*, near *Andromeda\Language\External*.

4.4.2 Parsing to the abstract syntax tree

The AST is a form of grammar in hierarchical data structures that is convenient for transformations and analysis. We can build the AST from top to bottom by taking the entire program code and descending to separate tokens, but it's likely we'll come to a dead end when it's not clear what element should be inside. For example, the main data type should contain a list of ... what? Procedures? Statements? Declarations?

```
data Program = Program [???]
```

The better way to construct the AST is related to BNF creation or, in our case, parser creation, starting from small parsers and going up to big ones. The `parsec` library already has many important combinators. We also need combinators for parsing integer constants, string constants, identifiers, end-of-lines, and lists of comma-separated things between brackets. Here are some of them:

```
-- Integer constant parser. Returns Int.
integerConstant :: Parser Int
integerConstant = do
  res <- many1 digit  #A
```

```
  return (read res)   #B

-- Identifier parser: first character is lowercase,
-- others may be letters, digits, or underscores.
-- Returns parsed string.
identifier :: Parser String
identifier = do
  c <- lower <|> char '_'   #C
  rest <- many (alphaNum <|> char '_')
  return (c : rest)
```

#A Parse one or more digits and put to res
#B The variable res is a string; the read function converts it to an integer
#C Parse lowercase letter; if this fails, parse the underscore symbol

Having plenty of small general parsers, we build bigger ones — for example, the parser for the value constructor entry. This gives us the corresponding ADT:

```
data Constructor = Constructor String ArgDef

constructorName :: Parser String
constructorName = do
  bigChar <- upper
  smallChars <- many alphaNum
  return (bigChar : smallChars)

constructor :: Parser Constructor
constructor = do
  name <- constructorName
  spaces
  argDef <- argDef
  return (Constructor name argDef)
```

We then have to define the parser and data type `argDef`. A small test of this concrete parser will show if we're doing things right or something is wrong. Parsec has the `parseTest` function for this (or you may write your own):

```
test :: IO ()
test = do
  let constructorStr = "Controller(\"boosters\")"
  parseTest constructor constructorStr
```

The AST we'll get this way can consist of dozens of ADTs with possibly recursive definitions. The AndromedaScript AST has more than 20 data types, and this is not the limit. Figure 4.6 shows the structure of it.

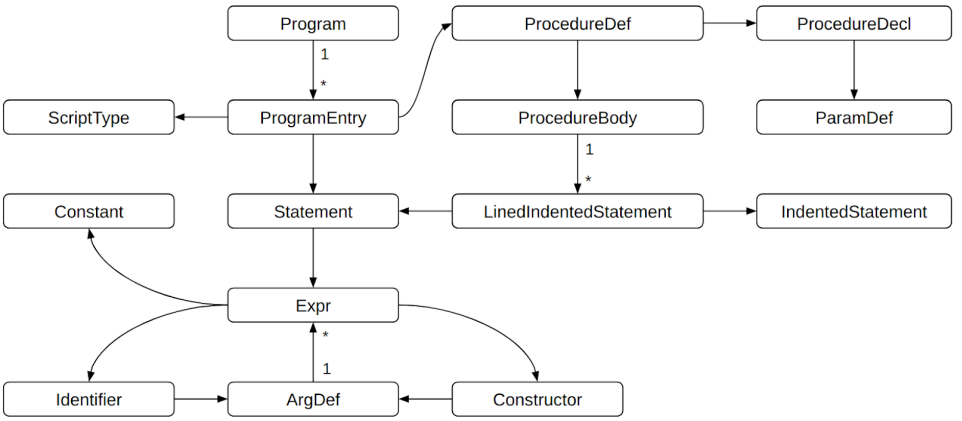


Figure 4.6 The AndromedaScript AST

The more diverse your grammar, the deeper your syntax tree. Then you have to deal with it during the translation process. That's why Lisp is considered a language without syntax: it has only a few basic constructions that will fit into a tiny AST. Indeed, the s-expressions are the syntax trees themselves, which makes the transformation much simpler.

4.4.3 The translation subsystem

Although translation theory has more than a half-century of history, the methods and tricks it suggests are still high science that can require the developer to have significant adjacent skills and knowledge. We'll talk about the translator, focusing not on how to write one but on how it should be organized. Let's treat the translator as a subsystem that’s responsible for interpreting the AST into evaluable code from one side and into embedded languages from the other.

The translator is the code that pattern matches over the AST and does the necessary transformations while building the resulting representation of the script being processed. The translator has to work with different structures: some of them are predefined and immutable; others will change during the translation. Immutable data structures include symbol tables, number transition rules, and dictionaries. Tables may contain patterns and rules of optimization, transitions for state machines, and other useful structures. Mutable (operational) data structures include graphs, number generators, symbol tables, and flags controlling the process. The state of the translator depends highly on the tasks it's intended to solve, but it’s never simple. Consequently, we shouldn't make it even more intricate by choosing the wrong abstractions.

There are several ways to make stateful computations in functional programming. From chapter 3, we know that if the subsystem has to work with state, the `State` monad and similar are good choices unless we’re worried by performance questions. It's very likely we'll want to print debug and log messages describing the translation process. The shortest (but not the best) path to do this is to make the translator impure. So, this subsystem should have the properties of two monads: `State` and `IO`, a frequent combination in Haskell applications. We’ll study other options we have to carry the state in the next chapter.

We define the translation type as the state transformer with the `IO` monad inside, parameterized by the `Translator` type:

```
type TranslatorSt a = StateT Translator IO a
```

The `Translator` type is an ADT that holds the operational state:

```
type Table = (String, Map String String)
type ScriptsTable = Map IdName (Script ())

data Tables = Tables
  { _constants :: Table
  , _values :: Table
  , _scriptDefs :: ScriptsDefsTable
  , _sysConstructors :: SysConstructorsTable
  , _scripts :: ScriptsTable
  }
```

```
data Translator = Translator
  { _tables :: Tables
  , _controlProg :: ControlProgram ()
  , _scriptTranslation :: Maybe ScriptType
  , _indentation :: Int
  , _printIndentation :: Int
  , _uniqueNumber :: Int
  }
```

The state has some management fields (`indentation`, `printIndentation`, `uniqueNumber`), tables, and other data. It's organized as nested ADTs. You might notice that the nesting of data structures unavoidably complicates making changes when they're immutable: you need to unroll the structures from the outside in, modify an element, and roll them back up. You can mitigate this problem by providing mutation functions, but it becomes annoying to support all new data structures this way. Fortunately, there's a better approach, known as lenses. You may have heard about lenses or even be using them, but if not, I'll give you a very short overview.

Lenses are a way to generalize working with deep, immutable data structures of any kind. Lenses are very similar to getters and setters in OOP, but they're combinatorial and do many things that getters and setters can't — for example, traversing a container and mutating every element inside the container or even deeper. You describe this operation with a few lens combinators and apply it to your container. The rest of the unrolling and wrapping of items will be done by the `lens` library. The following listing shows this idea.

Listing 4.14 A simple example of lenses

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens (traverse, makeLenses, set)

data BottomItem = BottomItem { _str :: String }
data MiddleItem = MiddleItem { _bottomItem :: BottomItem }
data TopItem    = TopItem    { _middleItem :: MiddleItem }

-- Making lenses with TemplateHaskell:
makeLenses 'BottomItem
makeLenses 'MiddleItem
makeLenses 'TopItem
```

```
-- Now you have lenses for all underscored fields.
-- Lenses have the same names except underscores.
-- Lenses can be combined following the hierarchy of the types:
bottomItemLens = traverse.middleItem.bottomItem.str

container = [ TopItem (MiddleItem (BottomItem "ABC"))
            , TopItem (MiddleItem (BottomItem "CDE"))]

expected = [ TopItem (MiddleItem (BottomItem "XYZ"))
           , TopItem (MiddleItem (BottomItem "XYZ"))]

container' = set bottomItemLens "XYZ" container
test = print (expected == container')
```

Here, the `set` combinator works over any lens irrespective of the structure it points to. The structure may be a single value lying deeply or a range of values inside any traversable container (lists, arrays, and trees are the example). The `test` function will print `True` because we've changed the internals of the container by applying to it the `set` combinator and the lens `bottomItemLens`, which pointed out what item to change. The definition of a lens looks like a chain of accessors to internal structures in the OOP manner, but it's a functional composition of smaller lenses that know how to address the particular part of the compound structure:

```
middleItem :: Lens TopItem MiddleItem
bottomItem :: Lens MiddleItem BottomItem
str        :: Lens BottomItem String

(middleItem . bottomItem)       :: Lens TopItem BottomItem
(middleItem . bottomItem . str) :: Lens TopItem String

traverse . middleItem . bottomItem . str
    :: Lens [TopItem] String
```

These small lenses are made by the `lens` library from named fields of ADTs prefixed by an underscore. It's the naming convention of Haskell's `lens` library that indicates what lenses we want to build for. Returning to the translator's state, we can see that it has many fields with underscore prefixes — that is, we'll get a bunch of lenses for these underscored fields.

Both Haskell and Scala have lens libraries with tons of combinators. The common operations are extracting values, mutating values, producing new structures, testing matches with predicates, transforming, traversing, and folding container-like structures. Almost any operation you can imagine can be replaced by a lens applied to the structure. Also important is that many of Haskell's lens combinators are designed to work inside the `State` monad: you don't have to store results in the variables, but you mutate your state directly (in the sense of the `State` monad mutation). For example, the translator tracks whether the syntactic indentation is correct. For this purpose, it has the `_indentation` field, and there are two functions that increase and decrease the value:

```
incIndentation :: TranslatorSt ()
incIndentation = indentation += 1

decIndentation :: TranslatorSt ()
decIndentation = do
  assertIndentation (>0)
  indentation -= 1

assertIndentation :: (Int -> Bool) -> TranslatorSt ()
assertIndentation predicate = do
  i <- use indentation
  assert (predicate i) "wrong indentation:" I
```

Here, `indentation` is the lens pointing to the `_indentation` field inside the `Translator` data type. The `use` combinator reads the value of the lens from the context of the `State` monad. Note how the operators `(+=)` and `(-=)` make this code look imperative! Building a translator can be really hard. Why not make it less hard by plugging in lenses? l'll stop here with studying the translation subsystem. If you want, you can keep going, digging into the code of the Andromeda software available on GitHub.

4.5 Summary

What are the reasons for developing DSLs? We want to accomplish goals such as the following:

- Investigate the domain and define its properties, components, and laws.
- Based on the domain properties, design a set of domain languages in a

form that’s more suitable and natural for expressing user scenarios.

- Make the code testable.
- Follow the SRP, keeping accidental complexity as low as possible.

In functional programming, it's more natural to design many DSLs to model a domain. The myth that this is complicated has come from mainstream languages. The truth here is that traditional imperative and object-oriented languages weren't intended to be tools for creating DSLs. Neither the syntax nor the philosophy of imperative languages is adapted to supporting this development approach. You can't deny the fact that when you program something, you’re creating a sort of domain language to solve your problem, but when the host language is imperative, it's more likely that your domain language will be atomized and dissolved in unnecessary rituals. As a result, you have that domain language, but you can't see it; you have to dissipate your attention on many irrelevant things.

Imagine that you wonder what a ball of water looks like, and you mold a lump of wet earth to find out. Well, it's spherical and has water inside, but you didn't get an answer to your question. You can only really see a ball of water in free fall or in zero gravity. Functional programming is like that water ball. Due to its abstractions, a domain can be finely mapped to code without any lumps of earth. When nothing obfuscates your domain language, maintaining it becomes simple and obvious, and the risk of bugs decreases.

The techniques and patterns for designing DSLs we discussed in this chapter are by no means comprehensive, and our work on the Logic Control subsystem is still not complete. The simple, “straightforward” eDSLs we developed first can be good, but it seems the monadic ones have more advantages. The Free monad pattern helps to build a scenario that we can interpret. In doing so, we separate the logic of a domain from the implementation. We also wrapped our Free languages into an arrowized interface. With it, we were able to illustrate our scenarios using flow diagrams.

You may ask why our domain model missed the requirement of starting scripts by time or event condition. We could probably model this in an event-driven manner: we run this functionality when we catch an event that the special subsystem produces in a time interval. But this design often blurs the domain model because the time conditional logic lies too far from the domain logic, and changing time conditions can be really hard. Also, we can't really interpret the

“runnable” code. The second option is to expand the `Control` eDSL with a special language, something like this:

```
data Control a
  = forall b. EvalScript (Script b) (b -> a)
  | forall b. EvalByTime Time (Script b) (b -> a)
  | forall b. EvalOnEvent Event (Script b) (b -> a)
```

But introducing such actions immediately makes our eDSL reactive (that is, the actions are reactions to events). To be honest, the domain of Logic Control has this property: it's really reactive, and we want to write reactive scenarios. But we’re trying to invent FRP. Again, we have two options: use existing FRP libraries somehow or continue developing our own with functionality limited. The first option is inappropriate because our scenarios should be interpretable. Consequently, it's necessary to create a custom interpretable FRP library. However, this will be a bit harder than we can imagine. In further chapters, we’ll see some ways to create reactivity on the basis of STM. We’ll leave the question about FRP for future books and materials, because it’s really huge.

Chapter 5

Application state

This chapter covers

- Stateful application in functional programming
- How to design operational data
- What the State monad is useful for in pure and impure environments

What puzzles me from time to time is that I meet people who argue against functional programming by saying that it can't work for real tasks because it lacks mutable variables. No mutable variables, - and therefore no state could change, so interaction with the program isn't possible. You might even be hearing that "a functional program is really a math formula without effects, and consequently it doesn't work with memory, network, standard input and output, and whatever else the impure world has. But when it does, it's not functional programming anymore." There are even more emotional opinions and questions there, for example: "Is it a cheating in Haskell when `IO` monad just masks the imperative paradigm?" An impure code that Haskell's `IO` monad abstracts over makes someone skeptical about how it can be functional while it's imperative.

Hearing this, we functional developers start asking ourselves whether we're all wandering in the unmerciful myths, trying to support immutability when it's infinitely more beneficial to use good old mutable variables. However, this isn't the case. Functional programming doesn't imply the absence of any kind of state. It's friendly to side effects but not so much as to allow them to vandalize our code. When you read an imperative program, you probably run it in your head

and see if there's any discrepancy between two mental models: the one you're building from the code (operational) and the one you got from the requirements (desired). When you hit an instruction that changes the former model in a contrary way to the latter model, you feel this instruction does a wrong thing. Stepping every instruction of the code, you change your operational model bit by bit. It's probably true that your operational model is mutable, and your mind doesn't accumulate changes to it the way lazy functional code can do. The next time you meet a code in the `State` monad, you try the same technique to evaluate it. You succeeded, because it can be read this way, but functions in the stateful monadic chain aren't imperative, and they don't mutate any state. That's why the `State` monadic code is easily composable and safe.

What about the `IO` monad? The code just feels imperative. It's fine to reason this way on some level of abstraction, but for a deep understanding of the mechanism, one should know that the chain of `IO` functions is just a declaration. By declaring an impure effect, we don't make our code less functional. We separate the declarative meaning of impurity from actual impure actions. The impurity will happen only when the `main` function is run. With the help of a static type system, the code that works in the `IO` monad is nicely composable and declarative — in a sense, you can pass your `IO` actions here and there as first-class objects. Still, running such code can be unsafe because we can make mistakes. This is a bit of a philosophical question, but it really helps to not to exclude Haskell from pure functional languages.

However, these two monads — `State` and `IO` — are remarkable because the state that you can express with them can be safe, robust, convenient, truly functional, and even thread-safe. How so? This is the theme of this chapter.

5.1 Stateful application architecture

This section prepares the ground for introducing concepts about state. Here, you'll find requirements to the spaceship's stateful simulator, and you'll develop its high-level architecture to some degree. But before this is possible, you'll need the notion of another free language, namely, a language for defining a ship-controllable network. In this section, I also give you some rationale for why it's important to build something partially implemented that already does a few real things. You'll see that the design path we chose in previous chapters works well and helps to achieve simplicity.

5.1.1 State in functional programming

State is the abstraction that deals with keeping and changing a value during some process. We usually say that a system is stateless if it doesn't hold any value between calls to it. Stateless systems often look like a function that takes a value and “immediately” (after the small amount of time needed to form a result) returns another value. We consider a function to be stateless if there’s no evidence for the client code that the function can behave differently given the same arguments. In imperative language, it's not often clear that the function doesn't store anything after it's called. Moreover, effects are allowed, so the function can, theoretically, mutate a hidden, secret state — for instance, a global variable or file. If the logic of this function also depends on that state, the function isn’t deterministic. If imperative state isn’t prohibited by the language, it's easy to fall into the “global variable anti-pattern”:

```
secretState = -1

def inc(val):
    if secretState == 2:
        raise Exception('Boom!')
    secretState += 1
    return val + secretState
```

Most functional languages don't watch you like Big Brother: you’re allowed to write such code. However, it's a very, very bad idea because it makes code behave unpredictably, breaks a function’s purity, and brings code out of the functional paradigm. The opposite idea — having stateless calculations and immutable variables everywhere — might make someone think state isn’t possible in functional language, but it's not true. State does exist in functional programming. Moreover, several different kinds of state can be used to solve different kinds of problems.

The first way to categorize state lies along the lifetime criteria:

- *State that exists during a single calculation*. This kind of state isn’t visible from outside. The state variable will be created at the beginning and destroyed at the end of the calculation (note that with garbage collection, this may be true in a conceptual sense, but isn’t how it really is). The variable can freely mutate without breaking the purity of the code until the mutation is strictly deterministic. Let's name this kind of state *auxiliary*, or

localized.

- *State with a lifetime comparable to that of the application*. This kind of state is used to drive the application's business logic and to keep important user-defined data. Let's call it *operational*.
- *State with a lifetime exceeding that of the application*. This state lives in external storage (databases) that provides long-term data processing. This state can be naturally called *external*.

The second division concerns a purity question. State can be

- *Pure*. Pure state isn't really some mutable imperative variable that's bound to a particular memory cell. Rather, it's a functional imitation of mutability. We can also say that pure state doesn't destroy the previous value when assigning a new one. Pure state is always bounded by some pure calculation.
- *Impure*. Impure state always operates by dealing with impure side effects such as writing memory, files, databases, or imperative mutable variables. While an impure state is much more dangerous than a pure one, there are techniques that help to secure impure stateful calculations. Functional code that works with impure state can still be deterministic in its behavior.

The simulation model represents a state that exists during the simulator's lifetime. This model holds user-defined data about how to simulate signals from sensors and keeps current network parameters and other important information. The simulator application's business logic rests on this data. Consequently, this state is operational across the entire application.

In contrast, the translation process from a hardware network description language (HNDL) script to the simulation model requires updating an intermediate state specifically to each network component being translated. This auxiliary state exists only to support HNDL compilation. After it's done, we get a full-fledged simulation model ready for the simulator to run. In previous chapters, we touched slightly on this kind of state here and there. Remember the external language translator that works inside the `State` monad? Every interpretation of a free language can be considered stateful in the bounds of an `interpret` function. In the rest of this chapter, we'll look more at this, while building the simulation model and an interface to it.

5.1.2 Minimum viable product

So far, we've built separate libraries and implemented distinct parts of the Logic Control and Hardware subsystems. According to the architecture diagram (see chapter 2, figure 2.15), the Andromeda Control Software should contain the following functionality: database, networking, GUI, application, and native API mapping. All we know about these parts is a list of their high-level requirements and a general plan for how to implement them. For example, a database component is needed to store data about ship properties: values from sensors, logs, hardware specifications, hardware events, calculation results, and so on. What concrete types of data should be stored? How many records are expected? What type of database is better suited for this task? How should we organize the code? The answers to these questions are still unknown and should be carefully analyzed. Imagine that we did this. Imagine that we went even further and implemented a subsystem to deal with databases. All is fine, except that we created another separate component among separate components. While these components don't interact with each other, we can't guarantee that they'll match up like Lego blocks in the future.

This is a risk that can destroy your project. I've seen it many times. Someone spends weeks developing a big, god-like framework, and when the deadline happens, realizes that the whole system can't work properly. He has to start from scratch. The end. To avoid this problem, we should prove that our subsystems can work together even if the whole application still isn't ready. Integration tests can help a lot here. They are used to verify the system on the whole when all parts are integrated and functioning in the real environment. When integration tests pass, it shows that the circle is now complete — the system is proven to be working. Beyond this, however, there is a better choice: a sample program, a prototype that has limited but still sufficient functionality to verify the idea of the product. You can find many articles about this technique using the keywords "minimal viable product," or MVP. The technique aims to create something real, something you can touch and feel right now, even if not all functionality is finished. This will require a pass-through integration of many application components; the MVP is more presentable than integration tests for this.

This chapter is the best place to start working on such a program, namely, a spaceship simulator. The simulator has to be stateful, no exceptions. In fact, devices in spaceships are micro-controllers, with their own processors, memory, network interfaces, clock, and operating system. They behave independently.

Events that they produce occur chaotically, and each device can be switched off while others stay in touch. All the devices are connected to the central computer, or Logic Control. Signals between computers and devices are transmitted through the network and may be retransmitted by special intermediate devices. Consequently, the environment we want to simulate is stateful, multi-thread, concurrent, and impure.

As you can see, we need a new concept of a network of devices. Going ahead, this will be another free language in the Hardware subsystem in addition to HDL (the hardware description language). This language will allow us to declare hardware networks and construct a simulation model. Let's first take a quick look into this eDSL. We won't follow the complete guide on how to construct it because there will be nothing new in the process, but it's worth getting familiar with the main takeaways anyway.

5.1.3 Hardware network description language

The mind maps we designed in chapter 2 may give us useful information about the structure of a spaceship. What else do we know about it?

- A spaceship is a network of distributed controllable components.
- The following components are available: Logic Control Unit (LCU), Remote Terminal Units (RTUs), terminal units, devices, and wired communications.
- Devices are built from analogue and digital sensors and one or many controllers.
- Sensors produce signals continuously, with a configurable sample rate.
- The controller is a device component with network interfaces. It knows how to operate by the device.
- Controllers support a particular communication protocol.
- The LCU evaluates general control over the ship, following instructions from users or commands from control programs.
- The network may have reserve communications and reserve network components.
- Every device behaves independently from the others.
- All the devices in the network are synchronized in time.

In chapter 3, we defined a DSL for device declaration, namely, HDL, but we still didn't introduce any mechanisms to describe how the devices are connected. Let's call this mechanism HNDL (for the *hardware network description language*), as mentioned in listing 3.1 — the Andromeda project structure. The HNDL scripts will describe the network of HDL device definitions.

We'll now revisit the Hardware subsystem. HDL is a free language that may be able to compose a device from sensors and controllers. Listing 5.1 shows the structure of HDL and the sample script in it.

Listing 5.1 The Free hardware description language

```
data Component a
  = SensorDef ComponentDef ComponentIndex Parameter a
  | ControllerDef ComponentDef ComponentIndex a

type Hdl a = Free Component a

boostersDef :: Hdl ()
boostersDef = do
   sensor aaa_t_25 "nozzle1-t" temperature
   sensor aaa_p_02 "nozzle1-p" pressure
   sensor aaa_t_25 "nozzle2-t" temperature
   sensor aaa_p_02 "nozzle2-P" pressure
   controller aaa_c_86 "controller"
```

Every instruction defines a component of the device. The HNDL script will utilize these HDL scripts. In other words, we face the same pattern of scripts over scripts introduced in chapter 4.

Let's assume the items in the network are connected by wires. The network is usually organized in a “star” topology because it's a computer network. This means the network has a tree-like structure, not a spider-web-like one. We'll adopt the following simple rules for our control network topology:

- LCUs can be linked to many terminal units.
- One terminal unit may be linked to one device controller.

Every device in the network should have its own unique physical address that other devices can use to communicate with it. The uniqueness of physical addresses makes it possible to communicate with a particular device even if

many of them are identical to each other. This isn't enough, however, because each device might have many controllers inside, so we also need to point to the particular controller. As long as a controller is a component, we can refer to it by its index. We have the `ComponentIndex` type for this. The pair of physical address and component index will point to the right controller or sensor across the network. Let it be the `ComponentInstanceIndex` type:

```
type PhysicalAddress = String
type ComponentInstanceIndex = (PhysicalAddress, ComponentIndex)
```

We're about to make HNDL. As usual, we map the domain model to the ADT that will be our eDSL. As mentioned, there are three kinds of network elements we want to support: LCUs, RTUs, and devices. We'll encode links between them as specific data types, so we can't connect irrelevant elements. You can think of links as specific network interfaces encoded in types. Listing 5.2 introduces HNDL. Notice that the automatic `Functor` deriving is used here to produce the `fmap` function for the `NetworkComponent` type. I believe you already memorized why the `NetworkComponent` type should be a functor and what role it plays in the structure of the Free monad.

Listing 5.2 The hardware network description language

```
{-# LANGUAGE DeriveFunctor #-}
module Andromeda.Hardware.HNDL where

type PhysicalAddress = String

data DeviceInterface = DeviceInterface PhysicalAddress

data TerminalUnitInterface
  = TerminalUnitInterface PhysicalAddress

data LogicControlInterface
  = LogicControlInterface PhysicalAddress

-- | Convenient language for defining devices in the network.
data NetworkComponent a
  = DeviceDef PhysicalAddress
              (Hdl ())
              (DeviceInterface -> a)
  | TerminalUnitDef PhysicalAddress
                    (TerminalUnitInterface -> a)
```

```
  | LogicControlDef PhysicalAddress
                    (LogicControlInterface -> a)
  | LinkedDeviceDef DeviceInterface TerminalUnitInterface a
  | LinkDef LogicControlInterface [TerminalUnitInterface] a
  deriving (Functor)

-- | Free monad Hardware Network Description Language.
type Hndl a = Free NetworkComponent a

-- | Smart constructors.
remoteDevice :: PhysicalAddress
             -> Hdl ()
             -> Hndl DeviceInterface

terminalUnit :: PhysicalAddress -> Hndl TerminalUnitInterface

logicControl :: PhysicalAddress -> Hndl LogicControlInterface

linkedDevice :: DeviceInterface
             -> TerminalUnitInterfac
             -> Hndl ()

link :: LogicControlInterface
     -> [TerminalUnitInterface]
     -> Hndl ()
```

Notice how directly the domain is addressed: we just talked about physical addresses, network components, and the links between them, and the types reflect the requirements we've collected. Let's consider the `DeviceDef` value constructor. From the definition, we can conclude that it encodes a device in some position in the network. The `PhysicalAddress` field identifies that position, and the (`Hdl ()`) field stores a definition of the device. The last field holds a value of type (`DeviceInterface -> a`), which we know represents a continuation in our free language. The `removeDevice` smart constructor wraps this value constructor into the Free monad. We can read its type definition as "`remoteDevice` procedure takes a physical address of the device, a definition of the device, and returns an interface of that device." In the HNDL script, it will look like so:

```
networkDef :: Hndl ()
networkDef = do
```

```
  iBoosters <- remoteDevice "01" boostersDef
  -- rest of the code
```

where `boostersDef` is the value of the `Hdl ()` type.

Also important is that all network components return their own “network interface” type. There are three of them:

```
DeviceInterface
TerminalUnitInterface
LogicControlInterface
```

The language provides two procedures for linking network elements:

```
linkedDevice :: DeviceInterface
             -> TerminalUnitInterface
             -> Hndl ()

link :: LogicControlInterface
     -> [TerminalUnitInterface]
     -> Hndl ()
```

The types restrict links between network components in a way that exactly follows the requirements. Any remote device can be linked to the intermediate terminal unit, and many terminal units can be linked to the LCU. It seems that this is enough to form a tree-like network structure that maybe doesn't reflect the complexity of real networks but is suitable for demonstrating ideas. In the future, we may decide to extend the language with new types of network components and links.

Finally, listing 5.3 shows the HNDL script for the simple network presented in figure 5.1.

Listing 5.3 Sample network definition script

```
networkDef :: Hndl ()
networkDef = do
  iBoosters <- remoteDevice "01" boostersDef
  iBoostersTU <- terminalUnit "03"
  linkedDevice iBoosters iBoostersTU
  iLogicControl <- logicControl "09"
  link iLogicControl [iBoostersTU]
```

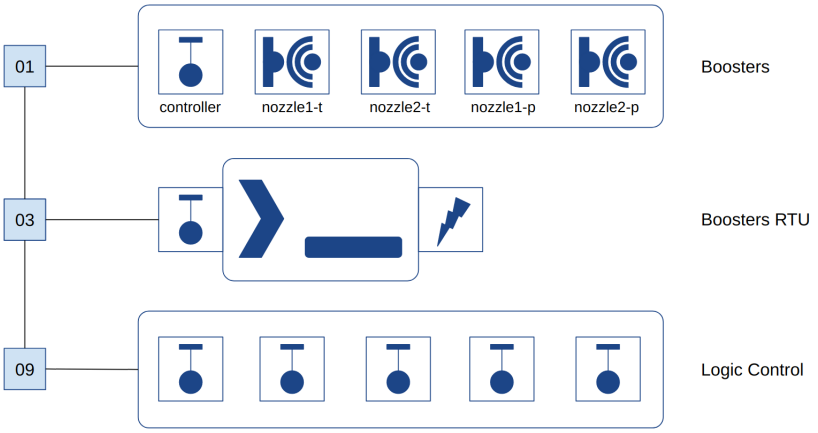


Figure 5.1 Sample network scheme

Our next station is “the simulator.” Please keep calm and fasten your seat belts.

5.1.4 Architecture of the simulator

The simulator will consist of two big parts: the simulator itself and the GUI to evaluate control over the simulated environment. Let's list the functional requirements for the simulation part:

- The simulation model should emulate the network and devices as close to reality as possible.
- Simulated sensors should signal the current state of the measured parameter. Because none of the physical parameters can really be measured, the signal source will be simulated too.
- Sensors signal measurements with a defined sampling rate.
- It should be possible to configure virtual sensors to produce different signal profiles. Such profiles should include random noise generation and generation by mathematical function with time as a parameter.
- Every network component should be simulated independently.
- There should be a way to run Logic Control scripts over the simulation as if it were a real spaceship.
- The simulation should be interactive to allow reconfiguring on the fly.

We said that the simulator is an impure stateful and multi-threaded application because it reflects a real environment of distributed independent devices. This statement needs to be expanded:

- *Multi-threaded*. Every sensor will be represented as a single thread that will produce values periodically, even if nobody reads it. Every controller will live in a separate thread as well. Other network components will be separately emulated as needed. To avoid wasting CPU time, threads should work with a delay that, in the case of sensors, is naturally interpreted as the sample rate.
- *Stateful*. It should always be possible to read current values from sensors, even between refreshes. Thus, sensors will store current values in their state. Controllers will hold current logs and options, terminal units may behave like stateful network routers, and so on. Every simulated device will have a state. Let's call the notion of threaded state a *node*.
- *Mutable*. State should be mutable because real devices rewrite their memory every time something happens.
- *Concurrent*. A node's internal thread updates its state by time, and an external thread reads that state occasionally. The environment is thereby concurrent and should be protected from data races, dead blocks, starvation, and other bad things.
- *Impure*. There are two factors here: the simulator simulates an impure world, and concurrency eventually necessitates impurity.

If you found these five properties in your domain, you should know that there’s an abstraction that covers exactly this requirement of a stateful, mutable, concurrent, and impure environment: STM, or Software Transactional Memory, introduced in chapter 1. Today, STM is the most reasonable way to combine concurrent impure stateful computations safely and program complex parallel code with much less pain and fewer bugs. In this chapter, we’ll consider STM as a design decision that significantly reduces the complexity of parallel models, but the more closer look on it you can find in the next chapter.

All information about the spaceship network is held in the HNDL network definition. Let me tell you a riddle. Once HNDL is a free language that we know does nothing real, but declares a network, how do we convert it into a simulation model? We do with this free language exactly what we did with other free

languages: we interpret it and create a simulation model during the interpretation process. We visit every network component (for example, `TerminalUnitDef`) and create an appropriate simulation object for it. If we hit a `DeviceDef` network component, we then visit its `Hdl` field and interpret the internal free HDL script as desired. Namely, we should create simulation objects for every sensor and every controller we meet in the device definition. Let's call the whole interpretation process a compilation of HNDL to a simulation model.

Once we receive the model, we should be able to run it, stop it, configure sensors and other simulation objects, and do other things that we stated in the requirements. The model will probably be somewhat complex because we said it should be concurrent, stateful, and impure. The client code definitely wants to know as little as possible about the guts of the model, so it seems wise to establish some high-level interface to it. In our case, the simulator is just a service that works with the simulation model of the spaceship's network. To communicate with the simulator, we'll adopt the MVar request–response pattern. We'll send actions that the simulator should evaluate over its internal simulation model. If needed, the simulator should return an appropriate response or at least say that the action has been received and processed. The request–response pipe between the simulator and the client code will effectively hide the implementation details of the former. If we want to do so, we can even make it remote transparently to the client code.

Figure 5.2 presents the simulator architecture.

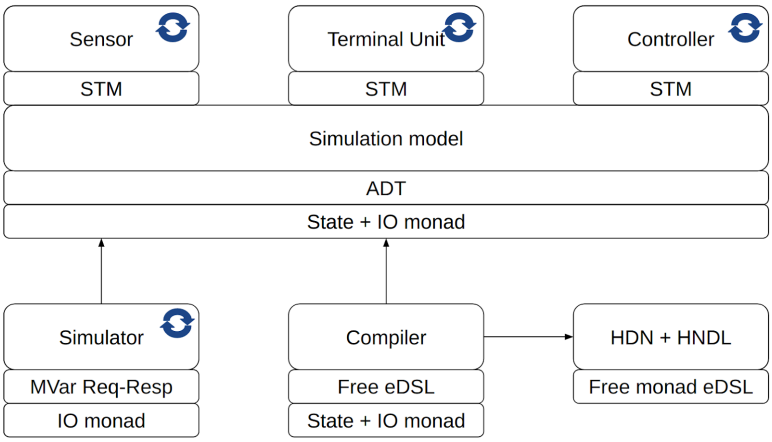


Figure 5.2 Architecture of the simulator

Now we're ready to do real things.

5.2 Pure state

By default definition, pure state is a state that can be created once, and every update of this value leads to it being copied. This is the so-called copy-on-write strategy. The previous value shouldn't be deleted from memory, although it's allowed that references no longer be pointed to it. Therefore, pure state can't be mutable in the sense of destroying an old value to put in a new one instead. A pure function always works with a pure state, but what do we know about pure functions? Just three things:

- A pure function depends only on the input arguments.
- A pure function returns the same value on the same arguments.
- A pure function can't do any side effects.

However, there is an escape from these narrow conditions. The third point usually includes interaction with operative memory because the latter is an external system that may fail: memory may end. Nevertheless, the memory may end just because you call too many pure functions in a recursion that's not tail-optimized, so the third requirement for the function to be pure isn't that convincing. What if you somehow pass to the function an empty array that can

be changed in whatever ways the function wants in order to calculate what it wants? It may freely mutate values in the array, but as long as the function does this the same way every time it's called, the regular output will be the same. The only requirement here is that no other code should have any kind of access (reading, writing) to the array. In other words, the mutable array will exist only locally, for this concrete function, and when calculations are done, the array should be destroyed. It's easy to see that the first and the second points in the previous list are satisfied.

Indeed, this notion of local mutable state exists and is known as, well, *local mutable state*. In Haskell, this notion is represented by the `ST` monad. Unfortunately, I can't provide more info about this monad here. Your best source will be a book by Vitaly Bragilevsky, "Haskell in Depth".

> **LINK** Vitaly Bragilevsky, *Haskell in Depth*
> https://www.manning.com/books/haskell-in-depth

In this section, we'll learn the following things:

- Argument-passing pure state
- The `State` monad

Think about it as an opportunity to nail down the knowledge in connection to developing the next part of the application. Also, the following text isn't about state, just because you're very familiar with the concepts it describes; rather, this section is about coherent modeling and development of functionality that hasn't been implemented yet.

5.2.1 Argument-passing state

If we consider that the simulator is our new domain, then domain modeling is the construction of the simulation model and its behavior. Let's develop the ADT `SimulationModel`, which will hold the state of all simulated objects:

```
data SimulationModel = SimulationModel
  {
    ????     -- The structure is yet undefined
  }
```

We concluded that simulated objects should live in their own threads, so we need some mechanism to communicate with them. First of all, there should be a way to identify a particular object the client code wants to deal with. As soon as the model is built from the HNDL description, it's very natural to refer to every object by the same identifications that are used in HNDL and HDL scripts. This is why the `PhysicalAddress` type corresponds to every network component, and the `ComponentIndex` type identifies a device component (see the definition of the HDL language). A pair of `PhysicalAddress` and `ComponentIndex` values is enough to identify a sensor or controller within the whole network. Let's give this pair of types an appropriate alias:

```
type ComponentInstanceIndex = (PhysicalAddress, ComponentIndex)
```

From the requirements, we know that we want to configure our virtual sensors; in particular, we want to set up a value-generation algorithm (potentially many times). For sure, not only sensors but every type of simulated object will have some specific options and state. It's wise to put options into separate data types:

```
data ControllerNode = ControllerNode
  {
    ????    -- The structure is yet undefined
  }

data SensorNode = SensorNode
  {
    ????    -- The structure is yet undefined
  }
```

Because every simulation object (node) is accessed by a key of some type, we can use a dictionary to store them — well, many dictionaries for many different types of nodes. This is the easiest design decision for keeping things simple and understandable:

```
import qualified Data.Map as M

type ComponentInstanceIndex = (PhysicalAddress, ComponentIndex)

data ControllerNode   = ControllerNode
data SensorNode       = SensorNode
data TerminalUnitNode = TerminalUnitNode
```

```
type SensorsModel = M.Map ComponentInstanceIndex SensorNode

type ControllersModel
  = M.Map ComponentInstanceIndex ControllerNode

type TerminalUnitsModel = M.Map PhysicalAddress TerminalUnitNode

data SimulationModel = SimulationModel
  { sensorsModel       :: SensorsModel
  , controllersModel   :: ControllersModel
  , terminalUnitsModel :: TerminalUnitsModel
  }
```

Let's return to the `SensorNode`. It should keep the current value and be able to produce a new value using a generation algorithm. The straightforward modeling gives us the following:

```
data ValueGenerator
  = NoGenerator
  | StepGenerator (Measurement -> Measurement)

data SensorNode = SensorNode
  { value :: Measurement
  , valueGenerator :: ValueGenerator
  , producing :: Bool
  }
```

If the `producing` flag holds, then the worker thread should take the current value, apply a generator to it, and return a new value. The value mutation function might look like so:

```
applyGenerator :: ValueGenerator -> Measurement -> Measurement
applyGenerator NoGenerator v = v
applyGenerator (StepGenerator f) v = f v

updateValue :: SensorNode -> SensorNode
updateValue node@(SensorNode val gen True) =
  let newVal = applyGenerator gen val
  in SensorNode newVal gen True
updateValue node@(SensorNode val gen False) = node
```

The `updateValue` function takes a value of the `SensorNode` type (the `node`), unpacks it by pattern matching, changes the internal `Measurement` value by calling the `applyGenerator` function, then packs a new `SensorNode` value to return it as a result. A function with type (`SensorNode -> SensorNode`) has no side effects and therefore is pure and deterministic.

In pure functional code, the state is propagated from the top pure functions down to the very depths of the domain model, walking through many transformations en route. The following function works one layer up from the `updateValue` function:

```
updateSensorsModel :: SimulationModel -> SensorsModel
updateSensorsModel simModel =
  let oldSensors = sensorsModel simModel
      newSensors = M.map updateValue oldSensors
  in newSensors
```

As you can see, the state is unrolled, updated, and returned as the result. You can go up and construct the `updateSimulationModel` function that unrolls all simulation models and updates them as necessary. The primer is shown in listing 5.4; notice how many arguments travel there between the functions.

Listing 5.4 Argument-passing state

```
-- functions that work with nodes:
updateValue :: SensorNode -> SensorNode
updateLog :: ControllerNode -> ControllerNode
updateUnit :: TerminalUnitNode -> TerminalUnitNode

updateSensorsModel :: SimulationModel -> SensorsModel
updateSensorsModel simModel =
  let oldSensors = sensorsModel simModel
      newSensors = M.map updateValue oldSensors
  in newSensors

updateControllersModel :: SimulationModel -> ControllersModel
updateControllersModel simModel =
  let oldControllers = controllersModel simModel
      newControllers = M.map updateLog oldControllers
  in newControllers
```

```
updateTerminalUnitsModel
  :: SimulationModel
  -> TerminalUnitsModel
updateTerminalUnitsModel simModel =
  let oldTerminalUnits = terminalUnitsModel simModel
      newTerminalUnits = M.map updateUnit oldTerminalUnits
  in newTerminalUnits

updateSimulationModel :: SimulationModel -> SimulationModel
updateSimulationModel simModel =
  let newSensors = updateSensorsModel simModel
      newControllers = updateControllersModel simModel
      newTerminalUnits = updateTerminalUnitsModel simModel
  in SimulationModel newSensors newControllers newTerminalUnits
```

A code with the argument-passing state you see in listing 5.4 can be annoying to write and to read because it requires too many words and ceremonies. This is a sign of high accidental complexity and bad functional programming. The situation tends to worsen for more complex data structures. Fortunately, we can solve this problem somewhat. Just use some function composition and record update syntax in Haskell or an analogue in another language:

```
updateSimulationModel :: SimulationModel -> SimulationModel
updateSimulationModel m = m
  { sensorsModel       = M.map updateValue (sensorsModel m)
  , controllersModel   = M.map updateLog (controllersModel m)
  , terminalUnitsModel = M.map updateUnit (terminalUnitsModel m)
  }
```

Despite being told that making more tiny functions is the key to clear and easily maintainable code, sometimes it's better to stay sane and keep it simple.

We just discussed the argument-passing style, which I'm convinced isn't that exciting because it just solves a small problem of pure state in functional programming. But remember, this kind of functional concept has given birth to functional composition, to lenses, and to all functional programming in the end. In chapter 3, we also noticed that the `State` monad is really a monadic form of the argument-passing style. Let's revise it and learn something new about monads as a whole.

5.2.2 The State monad

We'll compose a `SimState` monad that will hold a `SimulationModel` value in the context. The following functions from listing 5.4 will be rewritten accordingly:

```
updateSensorsModel         --->   updateSensors
updateControllersModel     --->   updateControllers
updateTerminalUnitsModel   --->   updateUnits
```

The following functions will stay the same (whatever they do):

```
updateValue
updateLog
updateUnit
```

Finally, the `updateSimulationModel` function will do the same thing as well, but now it should call a stateful computation over the `State` monad to obtain an updated value of the model. The monad is presented in listing 5.5.

Listing 5.5 The State monad

```
import Control.Monad.State
type SimState a = State SimulationModel a

updateSensors :: SimState SensorsModel
updateSensors = do
  sensors <- gets sensorsModel                    #1
  return $ M.map updateValue sensors

updateControllers :: SimState ControllersModel
updateControllers = do
  controllers <- gets controllersModel            #2
  return $ M.map updateLog controllers

updateUnits :: SimState TerminalUnitsModel
updateUnits = do
  units <- gets terminalUnitsModel                #3
  return $ M.map updateUnit units

#1 Extracting sensors model
#2 Extracting controllers model
#3 Extracting units model
```

The type `SimState a` describes the monad. It says that a value of the `SimulationModel` type is stored in the context. Every function in this monad can access that value. The `State` monad's machinery has functions to `get` the value from the context, `put` another value instead of the existing one, and do other useful things with the state. In the previous code, we used the `gets` function that has a type

```
gets :: (SimulationModel -> a) -> SimState a
```

This library function takes an accessor function with type (`SimulationModel -> a`). The `gets` function should then apply this accessor to the internals of the `SimState` structure to extract the internal value. In the `do` notation of the `State` monad, this extraction is designated by the left arrow (`<-`). In all monads, this means “do whatever you need with the monadic context and return some result of that action.”

The `gets` function is generic. It extracts the `SensorsModel` value (#1), `ControllersModel` (#2), and the `TerminalUnitsModel` (#3). After that, every model is updated with the result returned. It's important to note that working with the bounded variables (`sensors, controllers,` and `units`) doesn't affect the context, so the original `SimulationModel` stays the same. To actually modify the context, you can `put` a value into it:

```
modifyState :: SimState ()
modifyState = do
  ss <- updateSensors
  cs <- updateControllers
  us <- updateUnits
  put $ SimulationModel ss cs us

updateSimulationModel :: SimulationModel -> SimulationModel
updateSimulationModel m = execState modifyState m
```

Remember the `execState` function? It returns the context you'll get at the end of the monadic execution. In our case, the original model `m` was first put into the context to begin the computation, but then the context was completely rewritten by an updated version of the `SimulationModel`.

> **TIP** It won't be superfluous to repeat that the monadic approach is a general one — once you have a monad, you can apply many monadic combinators to your code irrespective of what the monad is. You can find monadic combinators in Haskell's `Control.Monad` module and in Scala's `scalaz` library. These combinators give you a "monadic combinatorial freedom" in structuring your code. There's usually more than one way to solve the same problem.

If you decide not to affect the context, you can just return a new value instead, using the `put` function, like so:

```
getUpdatedModel :: SimState SimulationModel
getUpdatedModel = do
  ss <- updateSensors
  cs <- updateControllers
  us <- updateUnits
  return $ SimulationModel ss cs us

updateSimulationModel :: SimulationModel -> SimulationModel
updateSimulationModel m = evalState getUpdatedModel m
```

In this case, you should use another function to run your state computation. If you've forgotten what the functions `execState` and `evalState` do, revisit chapter 3 and external references.

The following code commits to the "monadic combinatorial freedom" idea. Consider two new functions: `liftM3` and the bind operator (`>>=`):

```
update :: SimState SimulationModel
update = liftM3 SimulationModel
                updateSensors
                updateControllers
                updateUnits

modifyState :: SimState ()
modifyState = update >>= put

updateSimulationModel :: SimulationModel -> SimulationModel
updateSimulationModel m = execState modifyState m
```

There’s no way not to use the bind operator in the monadic code because it's the essence of every monad. We didn't see it before because Haskell's `do` notation hides it, but it is no doubt there. The equivalent `do` block for the `modifyState` function will be as follows:

```
modifyState :: SimState ()
modifyState = do
  m <- update
  put m
```

You may think that the bind operator exists somewhere in between the two lines of the `do` block (in fact, it exists before the left arrow). The truth is that nothing can be placed between lines, of course. The `do` notation will be desugared into the bind operator and some lambdas:

```
modifyStateDesugared :: SimState ()
modifyStateDesugared = update >>= (\m -> put m)
```

The expression (`\m -> put m`) is equivalent to just (`put`), which is an eta-converted form of the former.

I leave the joy of exploring the mystical `liftM3` function to you. “Monadic combinatorial freedom” becomes even sweeter with this and other monadic combinators: `forM`, `mapM`, `foldM`, and `filterM`. Being a proficient monadic juggler, you'll be able to write a compact, extremely functional and impressive code.

We'll continue to develop this in the section “Impure state with State and IO monads.” But what about the compiler of HNDL to `SimulationModel`? Let this (quite familiar, indeed) task be another introduction to lenses in the context of the `State` monad.

First, you declare an ADT for holding state. In Haskell, lenses can be created with the `TemplateHaskell` extension for fields that are prefixed by an underscore:

```
data CompilerState = CompilerState
  { _currentPhysicalAddress :: PhysicalAddress
  , _composingSensors       :: SensorsModel
  , _composingControllers   :: ControllersModel
  , _composingTerminalUnits :: TerminalUnitsModel
  }

makeLenses ''CompilerState

type SimCompilerState a = State CompilerState a
```

These lenses will be created:

```
currentPhysicalAddress :: Lens' CompilerState PhysicalAddress
composingSensors       :: Lens' CompilerState SensorsModel
composingControllers   :: Lens' CompilerState ControllersModel
composingTerminalUnits :: Lens' CompilerState TerminalUnitsModel
```

The `Lens'` type came from the `Control.Lens` module. It denotes a simplified type of lens. The type of some lens `Lens' a b` should be read as "lens to access a field of type `b` inside `a` type." Thus, the `composingSensors` lens provides access to the field of type `SensorsModel` inside the `CompilerState` ADT. The compiler itself is an instance of the `Interpreter` type class that exists for the HNDL free language. There's also the `interpretHndl` function. In order to save space, I don't present this stuff in this chapter, but you may see it in code samples for this book. The compiler entry point looks like so:

```
compileSimModel :: Hndl () -> SimulationModel
compileSimModel hndl = do
  let interpreter = interpretHndl hndl
  let state = CompilerState "" M.empty M.empty M.empty
  (CompilerState _ ss cs ts) <- execState interpreter state
  return $ SimulationModel ss cs ts
```

The implementation of two interpreter type classes follows: one for HNDL and one for HDL. The first interpreter visits every element of the network definition. The most interesting part here is the `onDeviceDef` method that calls the `setupAddress` function:

```
setupAddress addr = do
  CompilerState _ ss cs ts <- get
  put $ CompilerState addr ss cs ts

instance HndlInterpreter SimCompilerState where
  onDeviceDef addr hdl = do
    setupAddress addr
    interpretHdl hdl
    return $ mkDeviceInterface addr
  onTerminalUnitDef addr = ...
  onLogicControlDef addr = ...
  onLinkedDeviceDef _ _ = ...
  onLinkDef _ _ = ...
```

The `setupAddress` function uses the state to save the physical address for further calculations. This address will be used during device compilation. However, the function is too wordy. Why not use lenses here? Compare it to this:

```
setupAddress addr = currentPhysicalAddress .= addr
```

The (`.=`) combinator from the `lens` library is intended for use in the `State` monad. It sets a value to the field that the lens points to. Here, it replaces the contents of the `_currentPhysicalAddress` field with the `addr` value. The function becomes unwanted because it's more handy to set up the address in the `onDeviceDef` method:

```
instance HndlInterpreter SimCompilerState where
  onDeviceDef addr hdl = do
    currentPhysicalAddress .= addr
    interpretHdl hdl
    return $ mkDeviceInterface addr
```

Next, the instance of the `HdlInterpreter`:

```
compileSensorNode :: Parameter -> SimCompilerState SensorNodeRef
compileSensorNode par = undefined

instance HdlInterpreter SimCompilerState where
  onSensorDef compDef compIdx par = do
    node <- compileSensorNode par
    CompilerState addr oldSensors cs ts <- get
```

```
    let newSensors = Map.insert (addr, compIdx) node oldSensors
    put $ CompilerState addr newSensors cs ts
  onControllerDef compDef compIdx = ...
```

The `onSensorDef` method creates an instance of the `SensorNode` type and then adds this instance into the map from the `_composingSensors` field. To do that, we should `get` the state (the map) from the context, update the state, and `put` a new state (new map) back. These three operations can be easily replaced by one lens combinator (`%=`). You'll also need the `use` combinator. Compare:

```
instance HdlInterpreter SimCompilerState where
  onSensorDef compDef compIdx par = do
    node <- compileSensorNode par
    addr <- use currentPhysicalAddress                    #A
    let appendToMap = Map.insert (addr, compIdx) node
    composingSensors %= appendToMap
```

`#A Get value from the context`

The `use` combinator uses a lens to extract a value from the context. It's monadic, so you call it as a regular monadic function in the `State` monad. The function `Map.insert (addr, compIdx) node` is partially applied. It expects one more argument:

```
Map.insert (addr, compIdx) node :: SensorsModel -> SensorsModel
```

According to its type, you can apply it to the contents of the `_composingSensors` field. That's what the (`%=`) operator does: namely, it maps some function over the value behind the lens. The two monadic operators (`.=`) and (`%=`) and some simple combinators (`use`) from the lens library can replace a lot of boilerplate inside any `State` monad. Moreover, the `lens` library is so huge that you can dig through it like it's another language. But… Using lenses beyond basic getters and setters brings a lot of accidental complexity into the code. You have to memorise those lens operators, what they do and how to combine them. Hundreds of operators! With rather cryptic names, and even more cryptic types. So it would be more wise to not have a fancy for lenses. At least, a manually written boilerplate will be more accessible to the newcomers.

It never fails to be stateless, except the state is always there. It's never bad to be pure, unless you deal with the real world. It never fails to be immutable, but sometimes you'll be observing inefficiency. State is real. Impurity is real. Mutability has advantages. Is pure functional programming flawed in this? The answer is coming.

5.3 Impure state

We talked about pure immutable states while designing a model to simulate a hardware network. A good start, isn't it? The truth is that in real life, you'll more often need mutable imperative data structures rather than immutable functional ones. The problem becomes much sharper if you want to store your data in collections. This kind of state requires careful thinking. Sometimes you can be pleasured with persistent data structures that are efficient enough for many cases. For example, you may take a persistent vector to store values. Updates, lookups, and appends to persistent vectors have complexity O(1), but the locality of data seems to be terrible because a persistent vector is constructed over multiple tries. If you need to work with C-like arrays guaranteed to be continuous, fast, and efficient, it's better to go impure in functional code. Impure mutable data structures can do all the stuff we like in imperative languages, are less demanding to memory, can be mutated in place, and can even be marshaled to low-level code in C. On the other hand, you sacrifice purity and determinism going down to the impure layer, which of course increases the code's accidental complexity. To retain control over impure code, you have to resort to functional abstractions that solve some imperative problems:

- Haskell's `IORef` variable has exactly the same semantics as a regular variable in other languages. It can be mutated in place, making potential problems (nondeterminism, race conditions, and so on) the developer's responsibility. The `IORef a` type represents a reference type[5] over some type `a`.
- `MVar` is a concept of thread-safe mutable variables. Unlike the `IORef`, this reference type gives guarantees of atomic reading and writing. `MVar` can be used for communication between threads or managing simple use cases with data structures. Still, it's susceptible to the same problems: race conditions, deadlocks, and nondeterminism.

[5] Reference type in Wikipedia: https://en.wikipedia.org/wiki/Reference_type

- `TVar`, `TMVar`, `TQueue`, `TArray`, and other STM primitives can be thought of as further development of the `MVar` concept. STM primitives are thread-safe and imperatively mutable, but unlike `MVar`, STM introduces transactions. Every mutation is performed in a transaction. When two threads compete for access to the variable, one of two transactions will be performed, while the other can safely delay (be retried) or even roll back. STM operations are isolated from each other, which reduces the possibility of deadlock. With advanced combinatorial implementation of STM, two separate transactional operations can be combined into a bigger transactional operation that's also an STM combinator. STM is considered a suitable approach for maintaining complex state in functional programs; with that said, STM has many issues and properties one should know about to use it effectively.

We're now going to discuss how to redesign the simulation model with `IORef`s.

5.3.1 Impure state with IORef

Look at the `SimulationModel` and `updateSimulationModel` function again:

```
data SimulationModel = SimulationModel
  { sensorsModel :: SensorsModel
  , controllersModel :: ControllersModel
  , terminalUnitsModel :: TerminalUnitsModel
  }
```

The problem here is that this model doesn't fit into the idea of separate acting sensors, controllers, and terminal units. Imagine that the model was compiled from the network we defined earlier (`networkDef`):

```
test = do
  let simModel = compileSimModel networkDef
  print "Simulation model compiled."
```

where the `compileSimModel` function has come from the `SimulationCompiler` module:

```
module Andromeda.Simulator.SimulationCompiler where
compileSimModel :: Hndl () -> SimulationModel
compileSimModel = undefined
```

With a pure state, the only thing you can do is update the entire model. We wrote the `updateSimulationModel` function for that:

```
test = do
  let simModel1 = compileSimModel networkDef
  let simModel2 = updateSimulationModel simModel1

  print $ "initial: " ++ show (sensorsModel simModel1)
  print $ "updated: " ++ show (sensorsModel simModel2)
```

It seems impossible to fork a thread for each sensor as it was planned because neither sensor is seen in this test. Forking a thread for updating the whole model will also be useless. See the proof:

```
import Control.Concurrent (forkIO, ThreadId)

updatingWorker :: SimulationModel -> IO ()
updatingWorker simModel1 = do
  let simModel2 = simModel1
  updatingWorker simModel2

forkUpdatingThread :: SimulationModel -> IO ThreadId
forkUpdatingThread model = forkIO $ updatingWorker model

test = do
  threadId <- forkUpdatingThread (compileSimModel networkDef)
  -- what to do here??
```

The model will spin constantly in the thread, but it's not accessible from the outside. How to get values from sensors while the model is updating? How to set up another value generator to a specific sensor? How to query the controllers? This design of a pure simulation model is wrong. We'll try another approach.

The idea is that you can observe the impure mutation of an `IORef` value from different threads, which happens in the imperative world with any reference types and pointers. You first create a mutable variable with some value and then pass it to the threads, so they can read and write it occasionally. Listing 5.6 introduces the `IORef` type, some functions to work with, and stuff for threads.

This program has two additional threads forked. While the main thread is sleeping for 5 seconds, the first worker thread increases `refVal` by 1, and the second worker thread prints what it sees currently in the same `refVal`. Both threads then sleep for a second before they continue their business with `refVal`. When the program runs, you see some numbers from 0 to 5 being printed, with some repeating or absent, for example, 1, 2, 2, 3, 4.

Listing 5.6 IORef example

```
module IORefExample where

import Control.Monad (forever)
import Control.Concurrent (forkIO, threadDelay, killThread,
ThreadId)
import Data.IORef (IORef, readIORef, writeIORef, newIORef)

second = 1000 * 1000

increaseValue :: IORef Int -> IO ()
increaseValue refVal = do
  val <- readIORef refVal
  writeIORef refVal (val + 1)
  threadDelay second

printValue :: IORef Int -> IO ()
printValue refVal = do
  val <- readIORef refVal
  print val
  threadDelay second

main :: IO ()
main = do
  refVal <- newIORef 0
  let worker1 = forever $ increaseValue refVal
  let worker2 = forever $ printValue refVal
  threadId1 <- forkIO worker1
  threadId2 <- forkIO worker2
  threadDelay (5 * second)
  killThread threadId1
  killThread threadId2
```

Here, the purpose of the `newIORef`, `readIORef`, and `writeIORef` functions is obvious. All of them work in the `IO` monad because creating, reading, and writing a mutable variable is certainly a side effect:

```
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

The `forever` combinator repeats a monadic action forever:

```
forever :: Monad m => m a -> m b
```

In our case, there are two monadic actions called `increaseValue` and `printValue`. The `forever` combinator and an action passed represent a worker that might be forked into a thread:

```
worker1 :: IO ()
worker2 :: IO ()
forkIO :: IO () -> IO ThreadId
```

Due to Haskell's laziness, the construction

```
let worker1 = forever $ increaseValue refVal
```

doesn't block the main thread because it won't be evaluated; it's just binded to the `worker1` variable. It will be called by the `forkIO` function instead.

> **NOTE** There’s no thread synchronization in the code — the threads are reading and writing the shared state (`refVal`) at their own risk because neither the `readIORef` nor `writeIORef` function gives guarantees of atomic access. This is a classic example of code that one should avoid. To make it safer, it's worth replacing the `writeIORef` function with the “atomic” version: `atomicWriteIORef`. Still, programming with bare imperative freedom may lead to subtle bugs in parallel code. What if the second thread raises an exception immediately when it's forked? The first thread will never be stopped, so you'll get a zombie that just heats the CPU. Something can probably break the `threadDelay` and `killThread` functions, which can zombificate your threads too. With shared state and imperative threads you may find yourself plagued by a tiresome debugging of sudden race

conditions, dastardly crashes, and deadlocks. Conclusion: don't write a code like in listing 5.6.

How about the simulation model? Let's redesign the sensors-related part of it only, because the other two models can be done by analogy. Revise the sensors model that is a map of index to node:

```
type SensorsModel = M.Map ComponentInstanceIndex SensorNode
```

You can wrap the node into the reference type:

```
type SensorNodeRef = IORef SensorNode
type SensorsModel = M.Map ComponentInstanceIndex SensorNodeRef
```

The `SimulationModel` type remains the same — just a container for three dictionaries — but now every dictionary contains references to nodes. Next, you should create an `IORef` variable every time you compile a sensor node. The compiler therefore should be impure, so the type is now constructed over the `State` and `IO` monads with the `StateT` monad transformer:

```
type SimCompilerState = StateT CompilerState IO
```

So, the `HdlInterpreter` and the `HndlInterpreter` instances now become impure. In fact, replacing one monad with another doesn't change the instances that you see in the previous listings because the definition of interpreter type classes is restricted to the generic monad class but not to any concrete monad. The lenses will work too. What will change is the `compileSensorNode` function. Let's implement it here:

```
compileSensorNode :: Parameter -> SimCompilerState SensorNodeRef
compileSensorNode par = do
  let node = SensorNode (toMeasurement par) NoGenerator False
  liftIO $ newIORef node
```

According to the requirements, there should be a lever to start and stop the simulation. When the simulation is started, many threads will be forked for every node. When the simulation is stopped, threads must die. This means you need to store thread handles (the type `ThreadId` in Haskell) after the starting function is called. It would be nice to place this information about a sensor and a thread into a special type:

```
type SensorHandle   = (SensorNodeRef, ThreadId)
type SensorsHandles = M.Map ComponentInstanceIndex SensorHandle

forkSensorWorker       :: SensorNodeRef  -> IO SensorHandle
startSensorsSimulation :: SensorsModel   -> IO SensorsHandles
stopSensorsSimulation  :: SensorsHandles -> IO ()
```

The implementation of these functions is quite straightforward; it's shown in listing 5.7 it's really short and understandable, but it uses three new monadic combinators: the `when` combinator, a new version of the `mapM` monadic combinator, and the `void` combinator. You can learn more about them in the corresponding sidebar, or you may try to infer their behavior from the usage by analogy as the compiler does type inference for you.

Listing 5.7 discovers starting–stopping functions, the sensor updating function, and the worker forking function.

Listing 5.7 IORef-based simulation of sensors

```
import Data.IORef (IORef, readIORef, writeIORef, newIORef)
import Data.Traversable as T (mapM)
import Control.Monad (forever, void)
import Control.Concurrent
  (forkIO, threadDelay, killThread, ThreadId)

updateValue :: SensorNodeRef -> IO ()
updateValue nodeRef = do
  SensorNode val gen producing <- readIORef nodeRef
  when producing $ do
    let newVal = applyGenerator gen val
    let newNode = SensorNode newVal gen producing
    writeIORef nodeRef newNode
    threadDelay (1000 * 10)   -- 10 ms

type SensorHandle   = (SensorNodeRef, ThreadId)
type SensorsHandles = M.Map ComponentInstanceIndex SensorHandle

forkSensorWorker :: SensorNodeRef -> IO SensorHandle
forkSensorWorker nodeRef = do
  threadId <- forkIO $ forever $ updateValue nodeRef
  return (nodeRef, threadId)

startSensorsSimulation :: SensorsModel -> IO SensorsHandles
```

```
startSensorsSimulation sensors = T.mapM forkSensorWorker sensors

stopSensorWorker :: SensorHandle -> IO ()
stopSensorWorker (_, threadId) = killThread threadId

stopSensorsSimulation :: SensorsHandles -> IO ()
stopSensorsSimulation handles =
  void $ T.mapM stopSensorWorker handles
```

With the additional function `readSensorNodeValue` that's intended for tests only, the simulation of sensors can be examined, as in listing 5.8.

Listing 5.8 Simulation usage in tests

```
readSensorNodeValue
    :: ComponentInstanceIndex
    -> SensorsHandles
    -> IO Measurement
readSensorNodeValue idx handles =
  case Map.lookup idx handles of
    Just (nodeRef, _) -> do
      SensorNode val _ _ <- readIORef nodeRef
      return val
    Nothing -> do
      stopSensorsSimulation handles
      error $ "Index not found: " ++ show idx

test :: IO ()
test = do
  SimulationModel sensors _ _ <- compileSimModel networkDef
  handles <- startSensorsSimulation sensors
  value1 <- readSensorNodeValue ("01", "nozzle1-t") handles
  value2 <- readSensorNodeValue ("01", "nozzle2-t") handles
  print [value1, value2]
  stopSensorsSimulation handles
```

This will work now, but it will print just two zeros because we didn't set any meaningful value generator there. We could say the goal we're aiming toward is really close, but the solution has at least three significant problems:

- It's thread-unsafe.
- The worker thread falls into the busy loop anti-pattern when the producing

variable is false.

- The worker thread produces a lot of unnecessary memory traffic when the producing variable is true.

The problem with thread safety is more serious. One example of wrong behavior can occur if you duplicate the forking code unwittingly:

```
forkSensorWorker :: SensorNodeRef -> IO SensorHandle
forkSensorWorker nodeRef = do
  threadId <- forkIO $ forever $ updateValue nodeRef
  threadId <- forkIO $ forever $ updateValue nodeRef
  return (nodeRef, threadId)
```

Congratulations, zombie thread achievement is unblocked ... unlocked. The two threads will now be contending for writing access to the `SensorNode`. Mutation of the `nodeRef` is not atomic, so nobody knows how the race condition will behave in different situations. A huge source of nondeterminism that we mistakenly mold here can lead programs to unexpected crashes, corrupted data, and uncontrolled side effects.

The `updateValue` function reads and rewrites the whole `SensorNode` variable in the `IORef` container, which seems avoidable. You can — and probably should — localize mutability as much as possible, so you can try to make all of the `SensorNode`'s fields independent `IORef`s that will be updated when needed:

```
data SensorNode = SensorNode
  { value :: IORef Measurement
  , valueGenerator :: IORef ValueGenerator
  , producing :: IORef Bool
  }
type SensorsModel = M.Map ComponentInstanceIndex SensorNode
```

If you want, you may try to rework the code to support such a sensor simulation model. It's likely that you'll face many problems with synchronization here. This is a consequence of parallel programming in the imperative paradigm. Unexpected behavior, nondeterminism, race conditions — all this is the curse of every imperative-like threaded code, and we can do better. In spite of our current inability to refuse threads, there’s hopefully a cure for the imperative curse that we can use to diminish the problem. Software Transactional Memory. No, really,

try it, it’s great! I’ll leave this as an exercise. We’ll discuss STM in the next chapter, but in a slightly different context.

5.3.2 Impure state with State and IO monads

So far, we’ve been communicating with the simulation model directly (see listing 5.8; for instance, the functions `startSensorsSimulation` and `readSensorNodeValue`). Now we’re going to add another level of abstraction — the simulator service. Just to recall, let's revise what we know about it. According to the architecture in figure 5.2, the simulator will be stateful because it should spin inside its own thread and maintain the simulation model. The `State` monad that’s alloyed with the `IO` monad by means of a monad transformer will provide the impure stateful context in which it's very natural to place the simulation model. The simulator should receive requests about what to do with the simulation model, should do that, and then should send the results back. From a design viewpoint, this is a good place for the MVar request–response pattern. Every time the simulator thread gets the request, it transforms the request into the `State-IO` monadic action and applies that action to the simulation model. The simulator will provide some simple embedded language for the requests and responses. It's worthwhile to show the communication eDSL right now:

```
data In = StartNetwork
        | StopNetwork
        | SetGenerator ComponentInstanceIndex ValueGenerator
data Out = Ok | Fail String

type SimulatorPipe = Pipe In Out
```

The interaction approach is really ad hoc for now. The three actions it contains can't cover all needs, but we have to make something minimally viable to be sure that we’re going in the right direction.

A typical scenario is shown in figure 5.3.

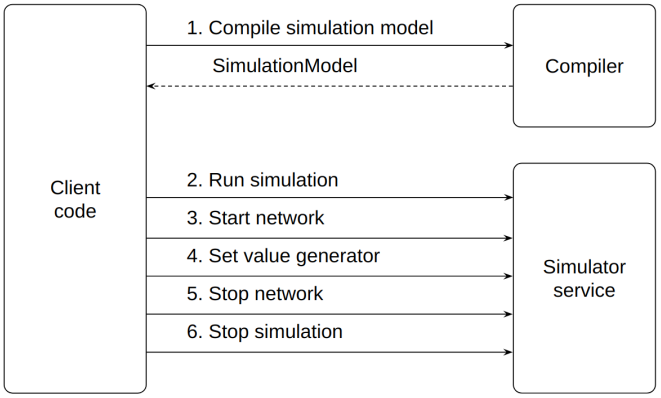


Figure 5.3 Simple interaction scenario

Let's try to write a test that shapes the minimal interface to the simulator that lets us support the scenario. It's fine that the machinery doesn't exist yet; following the TDD philosophy, we'll implement it later. Fortunately, something we already have is the compilation subsystem. This piece fits the picture well. Listing 5.9 shows the code.

Listing 5.9 Simulator test

```
module SimulatorTest where

import SampleNetwork (networkDef)
import Andromeda.Hardware
  ( Measurement(..)
  , Value(..)
  , ComponentInstanceIndex
  )
import Andromeda.Service (sendRequest)
import Andromeda.Simulator
    ( compileSimModel
    , startSimulator
    , stopSimulator
    , ValueGenerator(..)
    , In(..)
    , Out(..)
    )
```

```
increaseValue :: Float -> Measurement -> Measurement
increaseValue n (Measurement (FloatValue v))
  = Measurement (FloatValue (v + n))

incrementGenerator :: ValueGenerator
incrementGenerator = StepGenerator (increaseValue 1.0)

test = do
  let sensorIndex = ("01", "nozzle1-t")
  simulationModel <- compileSimModel networkDef
  (simulatorHandle, pipe) <- startSimulator simulationModel
  sendRequest pipe (SetGenerator sensorIndex incrementGenerator)
  stopSimulator simulatorHandle
```

The workflow is very straightforward: start, do, stop, using a simple interface and no matter what miles the simulator has to walk to make this real. That's why our interface is good. However, we have to elaborate the internals that aren't so simple. The most interesting function here is the `startSimulator` one. From the preceding code, it's clear that the function takes the simulation model and returns a pair of some handle and pipe. The handle is an instance of the special type `SimulatorHandle` that contains useful information about the service started:

```
data SimulatorHandle = SimulatorHandle
  { shSimulationModel :: SimulationModel
  , shSensorsHandles :: SensorsHandles
  , shStartTime :: UTCTime
  , shThreadId :: ThreadId
  }

startSimulator
  :: SimulationModel
  -> IO (SimulatorHandle, SimulatorPipe)
startSimulator = undefined
```

Clear enough. So, this function somehow starts a sensor model (that we know how to do), gets current time (the `UTCTime` is the standard type in Haskell), creates the pipe, and forks a thread for the simulator. This is the code:

```
forkSimulatorWorker
  :: SimulationModel
  -> SimulatorPipe
  -> IO ThreadId
forkSimulatorWorker simModel pipe = undefined

startSimulator
  :: SimulationModel
  -> IO (SimulatorHandle, SimulatorPipe)
startSimulator simModel@(SimulationModel sensorsModel _ _) = do
   pipe <- createPipe :: IO SimulatorPipe

   startTime <- getCurrentTime                              #A
   sensorsHandles <- startSensorsSimulation sensorsModel    #B
   threadId <- forkSimulatorWorker simModel pipe            #C

   let handle = SimulatorHandle
                  simModel
                  sensorsHandles
                  startTime
                  threadId
   return (handle, pipe)
```

#A System call from Data.Time.Clock
#B Known part
#C Forking a worker – not implemented yet

Notice that most of this function's parts are assembled from code that's already done. All we've written before is applied without any modifications. The main gap here is the forking of a thread. Let's give birth to the stateful impure service that's awaiting requests from the pipe. This is the type for its state:

```
import qualified Control.Monad.Trans.State as S
type SimulatorState a = S.StateT SimulationModel IO a
```

Fine, we know how that works. Now consider the following listing, which describes the core of the service.

Listing 5.10 The simulator core

```
import qualified Control.Monad.Trans.State as S

type SimulatorState = S.StateT SimulationModel IO

startNetwork :: SimulatorState ()                       #1
startNetwork = undefined

stopNetwork :: SimulatorState ()
stopNetwork = undefined

setGenerator
  :: ComponentInstanceIndex
  -> ValueGenerator
  -> SimulatorState ()
setGenerator idx gen = undefined

process :: In -> SimulatorState Out                     #2
process StartNetwork = do
  liftIO $ print "Starting network..."
  startNetwork
  return Ok
process StopNetwork = do
  liftIO $ print "Stoping network..."
  stopNetwork
  return Ok
process (SetGenerator idx gen) = do
  liftIO $ print "Setting value generator..."
  setGenerator idx gen
  return Ok

processor :: SimulatorPipe -> SimulatorState ()         #3
processor pipe = do
  req <- liftIO $ getRequest pipe
  resp <- process req
  liftIO $ sendResponse pipe resp
```

#1 Impure monadic actions that do something
with the simulator state (that is, SimulationModel)
#2 Translation of request into monadic action
#3 The processor of requests that spins inside
the SimulatorState monad and is driven by a separate thread

By the points:

- #1: Think about the `startNetwork` and `stopNetwork` functions. They should somehow affect the simulation model, keeping in the state context. By seeing their names, you can guess that they should switch every simulated device on or off — whichever is necessary for a particular node. Thus, they will evaluate some concurrent operations on the state, as well as the `setGenerator` action that probably should alter a value generator of some sensor node. If you're curious, see the code samples for this book, but for now, let's omit their implementation.
- #2: The `process` function translates the ADT language to the real monadic action. It can also do something impure, for example, write a log. The `liftIO` function allows impure calls inside the `State-IO` monad.
- #3: The `processor` function is a worker function for the thread. It's supposed to be run continuously while the simulator service is alive. When it receives a request, it calls the #2 process, and then the request is addressed to the simulation model being converted into some action.

The final step is `forkSimulatorWorker`:

```
forkSimulatorWorker
  :: SimulationModel
  -> SimulatorPipe
  -> IO ThreadId
forkSimulatorWorker simModel pipe = do
  let simulatorState = forever $ processor pipe
  forkIO $ void $ S.execStateT simulatorState simModel
```

All these things may feel familiar to you; that's right, we've learned every single combinator you see here; but there is one significant idea that might not be so easy to see. Remember the state of the simulation model compiler. You run it like so:

```
(CompilerState _ ss cs ts) <- S.execStateT compiler state
```

Or even remember how you run a stateful factorial calculation:

```
let result = execState (factorialStateful 10) 1
```

For all these occurrences of the `State` monad, you run your stateful computation to get the result right now. The state lives exactly as long as needed to execute a computation, not less, not more. When the result is ready, the monadic state context will be destroyed. But with the `SimulatorState`, this isn't the case. This state continues to live even after the `s.execStateT` function is finished! Woah, magic is here!

There is no magic, actually. The `s.execStateT` function will never finish. The thread we've forked tries very hard to complete this monadic action, but the following string makes the action proceed over and over again with the same state context inside:

```
let simulatorState = forever $ processor pipe
```

So Achilles will never overtake the tortoise. But this is normal: if you decide to finish him, you may just kill him:

```
stopSimulator :: SimulatorHandle -> IO ()
stopSimulator (SimulatorHandle _ sensorsHandles _ threadId) = do
  stopSensorsSimulation sensorsHandles
  killThread threadId
```

This is why we saved handles for sensors and the simulator's thread identifier.

I believe this core of the simulator is tiny and understandable. You don't need weird libraries to establish your own service, and you don't need any explicit synchronization. I'll tell you a secret: the `State-IO` monad here offers one more interesting design solution that isn't visible from the code presented. Did you have a question about what happened with the languages from Logic Control and why we don't proceed with them in this chapter? In reality, the `SimulatorState` monad makes it easy to incorporate script evaluation over the simulation model. This means that all the developments, the Free eDSLs we made in previous chapters, have started working! It requires only a little effort to add some new simulator API calls.

5.4 Summary

In this chapter, you learned a few (but not all) approaches to state in functional programs. You also improved your understanding of monads. The examples of the `State`, `IO`, `Free` monads you see here commit to the idea that this universal concept — monads — solves many problems in a handy way. This is why the book pays so much attention to giving you a practical view of monads instead of explaining a theory of how they really work. At this moment, it should be clear that monads are a much more useful thing for code design than the functional programming community believed before.

Indeed, designing with monads requires you to atomize pieces of code to smaller and smaller functions that have only a single responsibility. If that weren't so, then the composition would surely be impossible. If there are two functions `f` and `g` that you need to combine, you can't do this while `g` has two or more responsibilities. It's more likely that the `f` function doesn't return a value that's useful for all parts of the `g` function. The `f` function is simply unaware of the internals of `g`, and this is completely right. As a consequence, you have to follow SRP in functional programming. Again, as a consequence, you immediately gain huge reusability and correctness of code, even in a multi-threaded environment.

So, what concepts did you get from this chapter?

- The `State` monad is revisited. Although you can do stateful functional applications without any monads, the `State` monad can save lines of code along with time needed to write, understand, debug, and test a code. The `State` monad has many useful applications in every functional program.
- Pure state is good, but for some circumstances, you might want the `IORef` concept, which provides you with impure mutable references that are a full analogue of imperative variables. But of course, you should be aware of the problems the imperative nature of `IORef` drags in. At least you don't want to use it with threads.

The practices you've learned from the previous chapters are closely related to the methodology known as Domain Driven Design. My goal was to teach you this reasoning, this way of converting domain notions to a real code. We

developed dozens of eDSLs, we learned many techniques of analysis, we even touched some advanced Haskell concepts. So many topics were discussed and inscribed into a big picture of the Software Design discipline. However, there are more techniques and ideas to study. We're stopping our work on the Andromeda application here, but we're opening a new big theme - designing of real-world applications with a high quality. In the next chapters we'll be developing a free monadic framework for web backends and command line applications. As well as the applications themselves. This will be another sight onto software architecture, concurrency, persistence and testing. Our main interest will be focussed on design patterns, design approaches and design principles. Keep going!

Part III

Designing real world software

Chapter 6

Multithreading and concurrency

This chapter covers

- Multithreading and complexity
- Abstraction for threads and concurrent state
- Useful patterns for multithreaded applications

This chapter opens the third part of this book. Goding forward, we want to discuss more common questions of software design as it relates to real-world tasks that we might encounter in a regular software company. In future chapters, we'll be constructing a framework suitable for building multithreaded, concurrent web servers and standalone applications. We'll talk about Free monads as a way to organize a foundation for business logic code that will be testable, simple, and maintainable. Of course, we can't ignore such important themes as SQL and key-value (KV) database support, logging, state handling, concurrency and reactive programming, error handling, and more. We'll learn new design patterns and architecture approaches, such as Final Tagless/mtl, the ReaderT pattern, the Service Handle pattern, and others. By the end of the book you'll have learned how to create REST APIs, command-line tools, and web applications.

The reference project for this final part of the book will be the `Hydra` framework. This is actually not one but several frameworks built specifically for

this book. These frameworks provide the same functionality but are based on different approaches to more easily compare them. Hydra is a showcase project, but I've designed even more technologies for commercial companies using ideas from the book that I presented in previous chapters. We'll see what other interesting solutions can help us create a real-world software, and the Hydra framework will be a nice source of inspiration.

> **LINK** The Hydra framework
> https://github.com/graninas/Hydra

You might want to investigate some demo applications shipped with Hydra:

> **LINK** The *Labyrinth* game. This is an interactive command-line application that demonstrates Hydra's CLI subsystem, the layering of business logic, and the work with SQL and KV databases.
> https://github.com/graninas/Hydra/tree/master/app/labyrinth

> **LINK** The *astro* application. This is a web server and HTTP client powered by a famous Haskell library, *Servant*. The astro application is a tool that lets astronomers track cosmic activities such as meteors, supernova events, comets, and other things. The client application also demonstrates how to do Dependency Injection with Final Tagless, Free monads, the ReaderT pattern, bare IO, and the Service Handle pattern.
> https://github.com/graninas/Hydra/tree/master/app/astro

> **LINK** The *MeteorCounter* application. This demonstrates the work with processes, threads, and concurrency.
> https://github.com/graninas/Hydra/tree/master/app/MeteorCounter

In this chapter, we'll establish a foundation for our brand-new free monadic framework and talk about STM-based concurrency. However, the chapter can't cover all possible questions. Consider getting familiar with the excellent book by Simon Marlow, *Parallel and Concurrent Programming in Haskell* — it's entirely dedicated to explaining the many different aspects of this topic.

> **LINK** Simon Marlow, *Parallel and Concurrent Programming in Haskell*
> https://www.oreilly.com/library/view/parallel-and-concurrent/9781449335939/

As introduced earlier, the tool for astronomers seems to be a good playground for this chapter. And as usual, we should start by defining a domain of astronomical observations. We'll create a centralized catalogue of meteorites falling to Earth. It will be a server that accepts reports about sky events detected by astronomers. With the help of the client application, a scientist can report the region, the mass, and the time of a meteor. The client will be interacting with the server over some channel: TCP, UDP, HTTP, or WebSockets (this decision will be made at a future point in the process). The server will be a multithreaded, concurrent application with a database.

The `Meteor` data type will hold the info about meteors:

```
data Meteor = Meteor
  { size :: Int
  , mass :: Int
  }
  deriving (Show, Read, Eq)
```

The `getRandomMeteor` function creates a random meteor. This function may look different depending on the environment: `IO` monad, custom eDSL, mtl-style monad, and so on:

```
getRandomMeteor :: IO Meteor
getRandomMeteor = do
  rndSize <- randomRIO (1, 100)
  rndMass <- randomRIO (rndSize, rndSize * 10)
  pure $ Meteor rndSize rndMass
```

We'll get familiar with the application and other functions soon.

In the next section, we'll briefly talk about bare threads and multithreading code: why it's hard, what problems we might run into, and what problems we will run into, ten to one. In section 6.1.3, we'll start solving some of these problems and will continue searching for a better approach till the end of the chapter.

6.1 Multithreaded applications

Multithreading and concurrency remains one the most difficult themes in programming despite decades of research in this field. Yes, we've used bare threads in our programs for a long time, we know how to organize our systems as separate processes, and we've even collected broad experience in the wider theme of distributed calculations. But this was never an easy field of practice. We won't go too deep into multithreading in this chapter — we can't, essentially. Whole books are dedicated to explaining the huge number of pitfalls and approaches, and it's still not enough to cover them all. We have another goal, which is to see how multithreading fits into the application architecture. We'll try the most viable techniques and will try to keep our multithreaded code readable and maintainable. Fortunately, the functional programming world has given us such a thing as composable STM; this technology deserves much more attention because it can drastically reduce the complexity of multithreaded applications. Composable STM provides a (relatively) new reasoning that results in several useful application structuring patterns. STM so deftly addresses the main problems of multithreading that there's no real reason to avoid this technology except maybe in cases where you need to maximize performance and confidence in how the code evaluates under the hood. We'll take a look at the boundaries of STM and bare threads in this chapter.

6.1.1 Why is multithreading hard?

There are several reasons why creating multithreaded applications is hard. Most of them stem from the nature of the interaction among threads in a concurrent, mutable environment. The more operations a thread performs over a shared mutable state, the more cases we have in which another thread can change this shared state unexpectedly. Our intuition and reasoning about multithreaded code is usually very poor, and we can't easily predict all the problems. But we have to: even a single concurrent bug we missed can break all the logic and put the program into undefined behavior. The following problems can occur:

- *Race condition*. This is a frequent problem. It occurs when a shared resource can be switched to an invalid state by two or more threads accessing it in an unexpected way. When this situation happens, all subsequent computations become meaningless and dangerous. The program can still run, but the data is already corrupted and continues to corrupt other data. It's better to stop the program than let it keep running,

but detecting the bug is a matter of luck.

- *Deadlock*. A thread may get stuck in an attempt to read some lock-protected shared data when it's not ready. And it might result that the data will never be ready, so the thread will be blocked forever. If the program is configured to wait for this thread, the whole computation will be blocked. We'll get a running program that just hangs forever, waiting for nothing.
- *Livelock*. We can construct two or more threads in a way that has them playing ping-pong with the data at some point in the calculations but without ever going further. The threads will be very busy doing nothing useful. This is a kind of deadlock, but it makes the program simulate doing the work when it isn't.
- *Thread starvation*. This situation occurs when a thread works less than it could because it can't access resources fairly. Other threads take ownership of the resources, either because they're luckier, because it's easier for them, or for another reason. The result is the same: statistically, the starving thread works with less throughput than we would expect. In general, solving this problem can be tricky, and even STM doesn't provide guarantees of fairness (at least not in Haskell).
- *Unclear resource management*. This isn't a problem particular to multithreaded applications but is rather a general one. It's just amplified in a multithreaded environment. It's quite easy to complicate the code if we don't decide how we want to handle resources. We can step on bad race conditions accessing a killed resource (reference, pointer, file, handle, and so on), and our program will crash.
- *Resource leakage*. Leakage of some resource (system handles, memory, space, and so on) is another result of poor resource management.
- *Resource overflow*. It's relatively easy to run a thread that will produce more resources than the application can consume. Unfortunately, this problem can't be solved easily. Designing a fair multithreaded (or even distributed) system is a hard topic. We'll touch just the surface of it in this chapter. If you need more information, consider reading books about distributed systems and actor models.
- *Incorrect exceptions handling*. Exceptions are hard. There's no true way to organize exception safety in the application. Exceptions are another dimension that's orthogonal to other code, so we must think about a

multiplied number of cases when we write a program. Correct exceptions management in multithreaded applications is near to impossible, but we can probably mitigate the problem a little. We'll see.

Concurrent bugs are subtle, hard to search, test, and fix, and their presence can be spotted well after the code is deployed to production. This is why constructing concurrent multithreaded applications should be done with some care. Hopefully, we have immutability, `MVars`, and STM. These three concepts eliminate different problems — not only bugs but also the ridiculous complexity of multithreaded code writing.

6.1.2 Bare threads

There are many reasons why we'd want to introduce multithreading in our program:

- We know it will be evaluated on a more or less standard CPU, and we want to utilize the resources effectively.
- Our application is a service that will be processing irregular external requests, which could come at random moments in time and possibly in parallel.
- Our application is an active member of some bigger system and can produce results and send them to other members.

We could certainly try automatic or semiautomatic parallelization, but this works well only when you can define a separate task that can run independently. Once you introduce any kind of concurrency and mutable shared state, automatic parallelization becomes hard or impossible.

Let's proceed with our domain and create a code that simulates the two active members of the system:

- An astronomer who is watching for meteorites and reporting about events to the remote tracking center
- The remote tracking center that's accepting reports from astronomers

"Simulation" here means the threads will play a role of these two actors within a single application, so we'll do it without interprocess communication — just

bare threads and shared mutable objects. Let's deconstruct an application written like that.

Channel to report newly detected meteors. Essentially, a mutable reference having no thread safety facilities. It will be a shared resource for threads:

```
type ReportingChannel = IORef [Meteor]
```

Method for reporting the meteor:

```
reportMeteor :: ReportingChannel -> Meteor -> IO ()
reportMeteor ch meteor = do
  reported <- readIORef ch
  writeIORef ch $ meteor : reported
```

Simulation of an astronomer who is detecting meteorites at a random moment of time. When a meteor is found, it is immediately reported to the channel:

```
astronomer :: ReportingChannel -> IO ()
astronomer ch = do
  rndMeteor <- getRandomMeteor
  rndDelay <- randomRIO (1000, 10000)
  reportMeteor ch rndMeteor
  threadDelay rndDelay
```

Simulation of the tracking center. It polls the channel and waits for new meteorites reported. Currently, it does nothing with the meteors, but it can put these meteors into a catalogue:

```
trackingCenter :: ReportingChannel -> IO ()
trackingCenter ch = do
  reported <- readIORef ch
  -- do something with the reported meteors
  writeIORef ch []
  threadDelay 10000
```

Running the simulation. We're mandating the threads (existing one and additionally forked) to communicate using shared mutable reference.

```
app :: IO ()
app = do
  ch <- newIORef []
  forkIO $ forever $ astronomer ch
```

```
forever $ trackingCenter ch
```

While this program is deadly simple, it's also deadly wrong. It won't crash, but it will produce a lot of invalid data. It can lose meteorites, it can track meteorites multiple times, astronomers can report more meteorites than the `trackingCenter` is able to process — in other words, this program has destructive race conditions, it doesn't consider the balance of production and consumption, and it may occasionally overuse memory.

Finding such concurrency problems can be a kind of masochistic art because it's never obvious where the bug is in a more or less nontrivial program. Here, however, the problem is easy to find but not easy to fix. We have the two processes happening at the same time, as in figure 6.1.

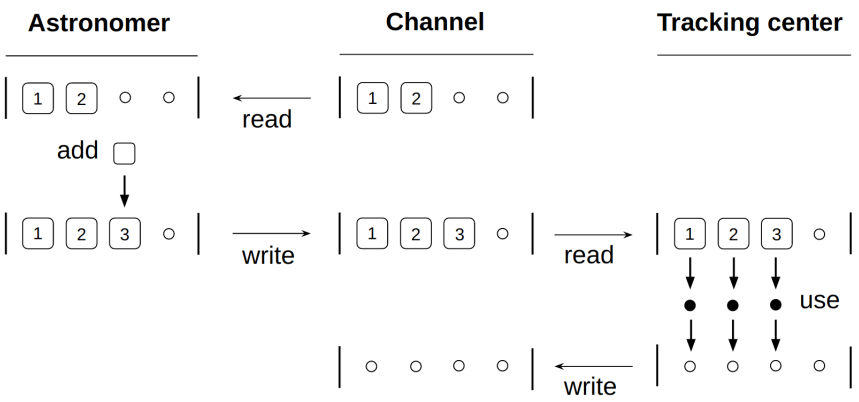


Figure 6.1 Accessing a shared state

An `Astronomer` first reads the channel, adds the third meteorite to the list, and writes the list back. The `Tracking center` then reads the updated channel, uses what it found there, and clears the channel. But this is the ideal situation. The astronomers and the center don't wait for each other, which is why operations with channels can sequence in an unexpected order, causing data loss or double usage. Consider figure 6.2, which shows a situation with double usage of meteorites 1 and 2.

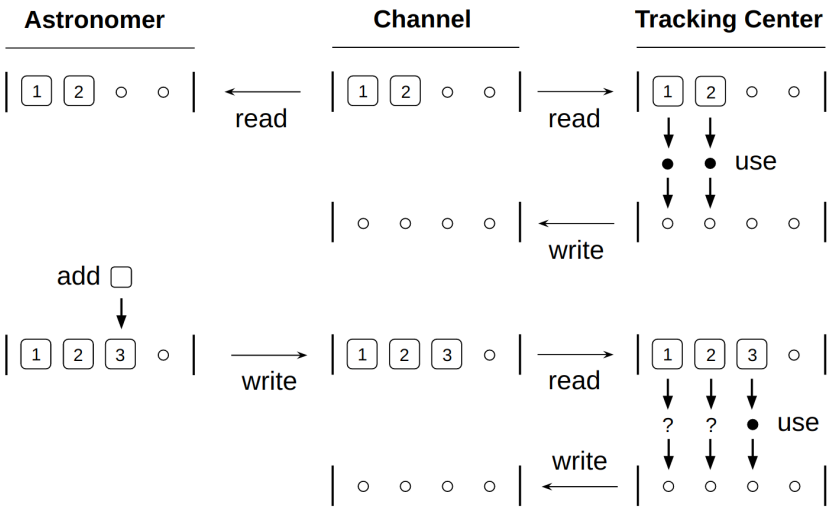


Figure 6.2 Race condition in accessing a shared resource and double usage error

This is clearly the race condition. We should organize access to the channel in a thread-safe manner somehow. It can be a mutex, which prevents a thread from operating on the channel if another thread still does its work. Or it can be a concurrent data structure, like `ConcurrentQueue` in C#. Sometimes we can say, "Ok, we're fine with data loss and double usage, it's not critical for our task, let it be." However, it would be better to consider these operations dangerous. In most languages, accessing a shared mutable data structure in the middle of the write procedure will introduce another problem: a partial change of new data might be observed along with a piece of old data. The data we read from an unprotected shared resource can occasionally vanish, leaving an undefined memory cell. And the program will crash.

So, we agree we need thread safety here. But what about the design we used in our previous program? Is it good enough? Well, if we're writing a big program with complex business logic that's triggering when the channel got a meteorite, then the design has flaws:

- First, we don't control side effects. Both the `Astronomer` and the `Tracking center` are operating inside `IO`.
- Second, we don't control resources. What if someone adds a resource initialization to the `astronomer` function? This function runs forever and many times. Hardly anyone can predict what will happen:

```
astronomer :: ReportingChannel -> IO ()
astronomer ch = do
  someRef <- newIORef [1..10]
  forever $ forkIO $ do         -- Infinite threads forking
    val <- readIORef someRef
    print val
```

A really strange thing happens here! This program will run out of memory within seconds. Who would write code like this? Well, this is just a metaphoric exaggeration, but it happens from time to time, and we get a cycle with uncontrolled resource initialization. We probably can't do much when it comes to bad luck, but we can prohibit mindlessly forking threads. Let's do this:

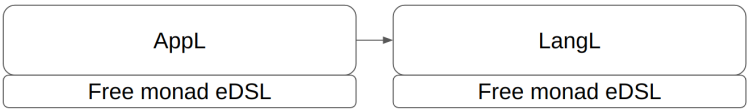
- We define parts of business logic that are allowed to fork threads.
- We create an unavoidable pool of threads, so a developer will be required to specify the maximum number of threads he wants in the system. (Let's not take into account that these `forkIO` threads are green — that's irrelevant for now).

6.1.3 Separating and abstracting the threads

The key idea is to allow only a certain set of operations in the business logic. On the top level, it will be a logic with the ability to fork threads — let's call them processes. Every process will be restricted so that it won't be allowed to fork child processes or threads. We'll only allow a process to get random numbers, operate with shared state, delay the execution, and maybe perform other useful business logic actions (logging, database access, and so on). Everything you need. Except forking child threads.

This separation of concerns will not only make some bad things irrepresentable (like infinite forking threads inside other threads) but will also structure the business logic in a layered manner. Earlier, I said that the application should be divided into layers, and business logic is one of them, but nothing can stop us

from layering the business logic itself. We'll do this using the same approach with Free monads. See figure 6.3, which describes this separation clearly.



- Configs
- Initialization (app state, DB)
- Eval LangL scripts
- Declare process (LangL script)

- Delaying of execution
- Random numbers generation
- Working with state
- Logging
- Working with DB

Figure 6.3 Separation of responsibilities

These languages aren't raw `IO` anymore, as in the previous section; we abstracted our intents and described the only things that are possible on these two layers: `AppL` and `LangL`.

The `AppL` scripts will be responsible for initializing the app state, the declarative description of processes and relations between them, config management, and so on. Let's agree that we don't want to have some specific domain logic here; rather we must prepare an environment for this domain logic in which it will be run. In turn, the `LangL` scripts should describe this logic, at least its parts. We can still call any `LangL` script and methods from the `AppL` directly. In fact, all the logic can be written in `AppL` scripts, unless we consider multithreading. Once it's introduced as shown in figure 6.3, the scripts evaluating in additional processes can only be `LangL` scripts.

Going further, we can define a more granular language hierarchy to achieve less coupling. I call this approach *Hierarchical Free Monads* (see figure 6.4).

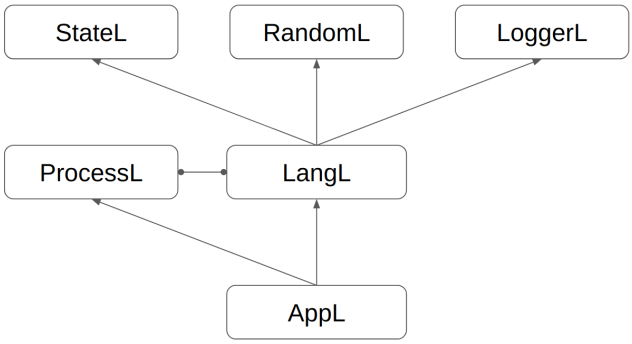


Figure 6.4 Free monadic languages organized hierarchically

> **NOTE** This is how I recommend arranging the "real" code. There's no good point to smashing everything together like other effect systems require. We'd better follow the main principle of software design and keep things decoupled. Also, it makes sense to not call these languages "effects" because functional programming has a much wider set of effects that can't be classified as subsystems (for example, the effect of a monadic list). The term "subsystems" works much better. Mainstream developers may find many similarities to how they're structuring applications in object-oriented and imperative languages.

But let's try to build things one by one, adding more bits gradually. Figure 6.3 describes a design that's very simple and straightforward. The corresponding eDSLs will look like the following listing.

Listing 6.1 Two languages for the business logic layer

```
-- Lower layer of business logic, the LangL eDSL
data LangF next where
  Delay        :: Int -> (() -> next)          -> LangF next
  GetRandomInt :: (Int, Int) -> (Int -> next)  -> LangF next
  NewVar       :: a -> (IORef a -> next)       -> LangF next
  ReadVar      :: IORef a -> (a -> next)       -> LangF next
  WriteVar     :: IORef a -> a -> (() -> next) -> LangF next

type LangL = Free LangF
```

```
-- Smart constructors
delay        :: Int          -> LangL ()
getRandomInt :: (Int, Int)   -> LangL Int
newVar       :: a            -> LangL (IORef a)
readVar      :: IORef a      -> LangL a
writeVar     :: IORef a -> a -> LangL ()

-- Upper layer of business logic, the AppL eDSL
data AppF next where
  EvalLang    :: LangL a  -> (a -> next)  -> AppF next
  ForkProcess :: LangL () -> (() -> next) -> AppF next

type AppL = Free AppF

-- Smart constructors
forkProcess :: LangL () -> AppL ()
evalLang    :: LangL a  -> AppL a
```

With these two languages, we'll immediately get the business logic divided into two layers. The next listing shows the code. Notice that the code in listing 6.2 is very similar to the "bare threads" code except that some problems are now fixed. We achieved our goal of prohibiting unwanted behavior; we created an abstraction over bare threads.

Listing 6.2 Business logic with process

```
getRandomMeteor :: LangL Meteor
getRandomMeteor = do
  rndSize <- getRandomInt (1, 100)
  rndMass <- getRandomInt (rndSize, rndSize * 10)
  pure $ Meteor rndSize rndMass

reportMeteor :: ReportingChannel -> Meteor -> LangL ()
reportMeteor ch meteor = do
  reported <- readVar ch
  writeVar ch $ meteor : reported

astronomer :: ReportingChannel -> LangL ()
astronomer ch = do
  rndMeteor <- getRandomMeteor
  rndDelay  <- getRandomInt (1000, 10000)
  reportMeteor ch rndMeteor
  delay rndDelay
```

```
trackingCenter :: ReportingChannel -> LangL ()
trackingCenter ch = do
  reported <- readVar ch
  -- do something with the reported meteors
  writeVar ch []
  delay 10000

app :: AppL ()
app = do
  ch <- evalLang $ newVar []
  forkProcess $ forever $ astronomer ch
  evalLang $ forever $ trackingCenter ch
```

Eh … you know what? The `app` script feels too imperative. While it's true it evals another script and forks a thread, we can consider it a declarative definition of the `app`. Actually, it's better to think of it as a declaration, which will allow more freedom in the design. Here, we don't fork processes, we define a process. We don't call scenarios, we define a logic to be evaluated in the main thread. So, let's just define two aliases for the `forkProcess` and `evalLang` functions:

```
process :: LangL () -> AppL ()
process = forkProcess

scenario :: LangL a -> AppL a
scenario = evalLang

app :: AppL ()
app = do
  ch <- evalLang $ newVar []
  process $ forever $ astronomer ch
  scenario $ forever $ trackingCenter ch
```

Just a little tweak that makes us think about the `AppL` script differently. In FDD, we should seek opportunities for declarative descriptions of our intents. In the future, we could add more declarative definitions — for example, starting some network server:

```
app :: AppL ()
app = do
  serving TCP $ do
    handler someHandler1
    handler someHandler1

  process logic1
  process logic2
```

What do you think — is it better now?

Okay. The picture still lacks a big part, namely, the application runtime and interpreters. Will this abstraction even work? What if our design cannot be properly implemented? This usually isn't a problem for Free monads, in which objectification of the design ideas doesn't require additional concepts, such as Haskell's advanced type-level tricks, multiparametric type classes, type families, and so on. Moreover, it's better to compose a business logic on top of unimplemented concepts to see how it feels rather than start from the implementation. Just to remind you, there is a clear sequence for designing eDSLs:

1. Design the eDSL from a usage point of view.
2. Create some logic over it and ensure it looks good.
3. Make it compile.
4. Make it work.

Our next goal is to make it work. One question: what if someone wrote the following code?

```
app :: AppL ()
app = do
  ch <- evalLang $ newVar []

  -- forever process forking
  forever $ process $ forever $ astronomer ch
```

Yes, this problem again. One can still fork threads infinitely within an `AppL` script. Let's think about how we can enhance our abstraction here and introduce a thread pool.

6.1.4 Threads bookkeeping

Different methods can be invented to limit the number of threads acting in the system. Depending on our needs, either checking for limits can be explicit, and therefore we'll have some methods in our eDSLs to do that, or it's possible to hide the control check from the business logic entirely and make it implicit. Decisions should be argued by the answer to the question, "What do you want to achieve?" We'll discuss both explicit and implicit approaches.

Having an explicit way means we can vary the business logic depending on the limits. We need methods in our eDSLs to ask for current status. We could end up with a language design as follows:

```
-- Token for accessing the process forked.
data ProcessLimits = ProcessLimits
  { currentCount :: Int
  , maximumCount :: Int
  }

data ProcessToken a = ProcessToken Int

data ProcessF next where
  GetProcessLimits                                  #A
    :: (ProcessLimits -> next)
    -> ProcessF next
  TryForkProcess                                    #B
    :: LangL a
    -> (Maybe (ProcessToken a) -> next)
    -> ProcessF next
  AwaitResult                                       #C
    :: ProcessToken a
    -> (a -> next)
    -> ProcessF next

type ProcessL = Free ProcessF

#A Method for getting the limits
#B Method for forking processes
   If success, a token will be returned
#C Method for awaiting results (blocking)
```

`ProcessL` fitted to the `AppL`:

```
data AppF next where
  EvalLang :: LangL a -> (a -> next)  -> AppF next
  EvalProcess :: ProcessL a -> (a -> next) -> AppF next

type AppL = Free AppF

-- Smart constructors. Note they are in the AppL.

getProcessLimits :: AppL ProcessLimits
tryForkProcess :: LangL a -> AppL (Maybe (ProcessToken a))
awaitResult :: ProcessToken a -> AppL a
```

It's supposed that a client will run `tryForkProcess` to get a process running. The limit might be exhausted, and the function will return `Nothing` (or another result with more info provided). The client will also be blocked, awaiting a result from the process it forked earlier. The next listing shows the usage:

```
getMeteorsInParallel :: Int -> AppL [Meteor]
getMeteorsInParallel count = do
  mbTokens <- replicateM count (tryForkProcess getRandomMeteor)
  let actualTokens = catMaybes mbTokens
  mapM awaitResult mbTokens    -- Blocked on every process
```

It might seem like this design is good enough, but there are several flaws. The language doesn't look like a thread pool. It certainly could be brought closer to the usual thread pools in imperative languages — just name the methods accordingly, and that's it. This isn't a problem; the design can be freely updated according to your requirements. But there are more serious problems here. The biggest one is that it's possible to run `awaitResult` for a process already finished and destroyed. It's not very clear what to do in this situation. The best option is to return `Either` having error or success:

```
data ProcessF next where
  AwaitResult
   :: ProcessToken a
   -> ((Either Error a) -> next) -> ProcessF next

awaitResult :: ProcessToken a -> ProcessL (Either Error a)
```

(The `awaitResult` method is blocking. It will stop the evaluation until some results from the specified process are observed. In case the process is absent, failed, or exceptioned, the error is returned.)

The language also doesn't allow us to set up new limits. They are predefined on the program start, exist on the implementation level, and can only be observed without modification. If we imagine we have a method for changing the limits, we immediately get the possibility of a race condition by design:

```
data ProcessF next where
  GetProcessLimits :: (ProcessLimits -> next) -> ProcessF next
  SetMaxCount :: Int -> (() -> next) -> ProcessF next
```

What if the first actor got the max count already, and the second one decreased it? The first actor will get invalid data without knowing this fact. Or what if we decreased the max count, but there was a thread pool occupied to its maximum? Nothing extremely bad will likely happen, except that there will be more processes evaluating than the limits allow. Other race conditions are possible with this mutability involved, so we have to be careful. Ideally, our languages shouldn't introduce problems. This is even more important in a multithreaded situation.

Just to be clear: we didn't plan to run either the `ProcessL` or `AppL` methods from different threads. In fact, we're trying to avoid this. According to eDSL design and semantics, the `ProcessL` and `AppL` scripts will be evaluated sequentially, in a single thread. The `LangL` scripts are also sequential. The difference is that the `LangL` scripts can be evaluated in parallel, which brings real multithreading into the whole problem. From the `AppL` point of view, these processes represent asynchronous actions. With no mutability involved, this approach is thread-safe. But the problems of multithreading environments occur immediately if we introduce mutable methods in the `ProcessL` or add a shared mutable state to the `LangL` layer. And we already did: we have methods to work with `IORef` in `LangL`. This becomes very dangerous. What's the right way to handle this concurrency? Let's talk about this situation in the next paragraph, since we're about to finish our current talk.

Let's examine how to implicitly control the limits. Here, we need to reveal the underground side of the Free monad languages, namely, runtime facilities (the

implementation layer). All the important activity will be there: checking limits, awaiting resources to be released, and obtaining a resource. For the client code, the interface will be the same as in listing 6.1:

```
data AppF next where
  EvalLang    :: LangL a  -> (a -> next)  -> AppF next
  ForkProcess :: LangL () -> (() -> next) -> AppF next

type AppL = Free AppF

-- Evaluate a LangL script.
evalLang :: LangL a -> AppL a

-- Fork a process. Block if the pool is full.
forkProcess :: LangL () -> AppL ()
```

If the hidden process pool is exhausted, the `forkProcess` method should block the execution and wait for it. In the implementation layer, we'll simply be counting the number of active processes. The following data type shows the structure for the runtime behind the scenes:

```
data Runtime = Runtime
  { _maxThreadsCount :: Int
  , _curThreadsCount :: MVar Int
  }
```

An interesting moment pops up here. We have to use a synchronized variable to store the number of active threads. Why? The reason will be clear from the way we're doing it. If the limit is fine, the two actions have to be done in a single shot: increasing the counter and forking a thread. We cannot increase without forking or fork without increasing. These actions should never desync. From the other side, the counter should be decreased once a thread finishes its activity. Polling the thread and waiting for when it's done would work, but will require a supervisor thread and that more complex logic be written. Instead, this lets us tie a finishing action to the thread on its creation, so this action will be automatically called in case of success, error, or any exception thrown by a thread. Here's the code:

```
interpretAppF :: Runtime -> AppF a -> IO a

interpretAppF rt (EvalLang act next) =
  next <$> runLangL rt act

interpretAppF rt (ForkProcess act next) =
  next <$> go 1
  where
    go factor = do
      let psVar = _curThreadsCount rt
      let maxPs = _maxThreadsCount rt
      ps <- takeMVar psVar                              #A

      when (ps == maxPs) $ do                           #B
        putMVar psVar ps                                #C
        threadDelay $ 10 ^ factor
        go $ factor + 1

      when (ps /= maxPs) $ do
        void $ forkFinally                              #D
          (runLangL rt act)
          (const $ decreaseProcessCount psVar)
        putMVar psVar $ ps + 1

decreaseProcessCount :: MVar Int -> IO ()               #E
decreaseProcessCount psVar = do
  ps <- takeMVar psVar
  putMVar psVar $ ps - 1
```

#A Blocking on the synchronized process counter until it's available.
#B Checking the current limit.
#C Limit is reached. Releasing the MVar. Restarting the try after a delay.
#D Limit is fine. Releasing the MVar and forking a thread with a cleanup action.
#E Decreasing the counter thread safely

NOTE Here, we use exponential backoff for the sleeping time after each failure. It might not be the best solution, and we might want to consider other strategies: constant delay time, arithmetic or geometric

progression, Fibonacci numbers, and so on. But this question of delaying effectiveness moves away from the book's theme.

Synchronizing on the `psVar MVar` is necessary. A forked thread will finish at a random time and will interfere with one of these situations: the main thread can perform another forking operation by changing the counter; or possibly another forked thread is about to finish and therefore wants to change the counter. Without synchronization, exactly the same problem will happen as we saw in figure 6.2. Figure 6.5 shows a race condition when `MVar` isn't used. The figure demonstrates a data loss and corruption resulting in an invalid counter state.

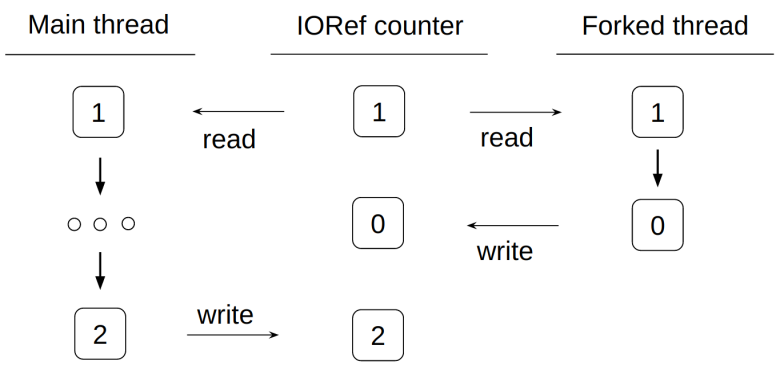


Figure 6.5 Concurrency bug caused by a race condition

Figure 6.6 explains how `MVar` can save us from this situation.

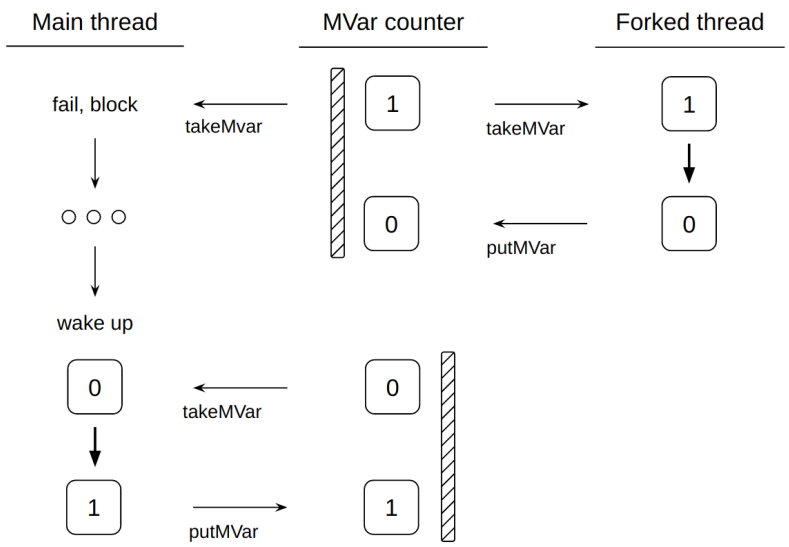


Figure 6.6 Safe concurrent interaction over the synchronized MVar

The paragraph will be incomplete without the final part of the code, the starting function. The runtime takes the configs on the maximum limit of threads:

```
app :: AppL ()
app = …

runAppL :: Runtime -> AppL a -> IO a
runAppL rt = foldFree (interpretAppF rt)

main :: IO ()
main = do
  psVar <- newMVar 0
  let rt = Runtime 4 psVar
  runAppL rt app
```

Bookkeeping can be done differently. For example, you might want to keep thread IDs for controlling the threads from outside: ask a thread about the status, kill it, pause, or resume.

There are also difficulties related to the implementation of the `LangL` runner and getting a typed result from the thread. These problems are technical and mostly Haskell-related. Nothing that scary, but we'll leave it untouched here.

6.2 Software Transactional Memory

STM is an approach to concurrent mutable data models. The key idea of STM is coded in its name: a memory (data) that can be mutated only within a single isolated transaction. STM has similarities with transactional databases: while a value is handled by one thread, another concurrent thread will wait until it's free. In contrast to databases, STM isn't an external service, it's a concurrent application state programmed to support your particular domain needs. With STM, you define a model that can be changed by different threads simultaneously and safely if there's no collision, but if there is, STM will decide how to solve the collision in a predictable way. This predictability is what we were missing in more low-level approaches involving custom-made mechanisms of concurrency in our apps. Of course, you can take `ConcurrentQueue` or `ConcurrentDictionary` in C#, and as long as you use these structures in a simple way, you're fine. But if you need a code that will be interacting with *both* `ConcurrentQueue` and `ConcurrentDictionary`, you'll immediately get a higher-level concurrency problem: how to avoid race conditions and unpredictability while keeping the complexity low. Monadic STM (like Haskell has) solves all these problems. It offers not only a predictable concurrency but also a composable concurrency, in which you can operate with an arbitrary data structure in a safe manner even though it's a simple concurrent variable, composition of them, or even a bunch of complex concurrent data structures. We can say monadic STM is an orchestra conductor. It has the whole picture of the symphony, and it looks for the correct, completely definite evaluation of the music.

6.2.1 Why STM is important

STM works over data models locally and allows separate parts to be operated independently in different transactions in case these operations aren't mutually dependent. Figure 6.7 demonstrates a forest-like data structure that's composed from STM primitives, so the two threads may access their parts without blocking, whereas the third thread will wait its turn.

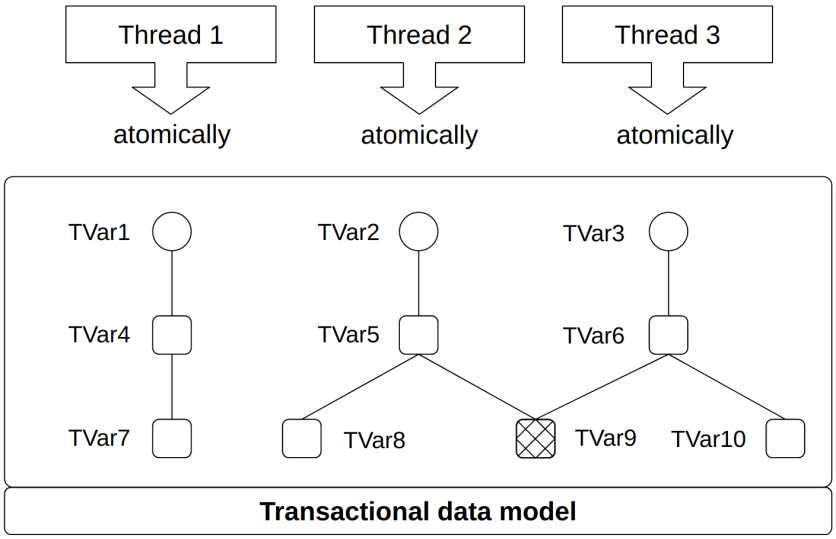
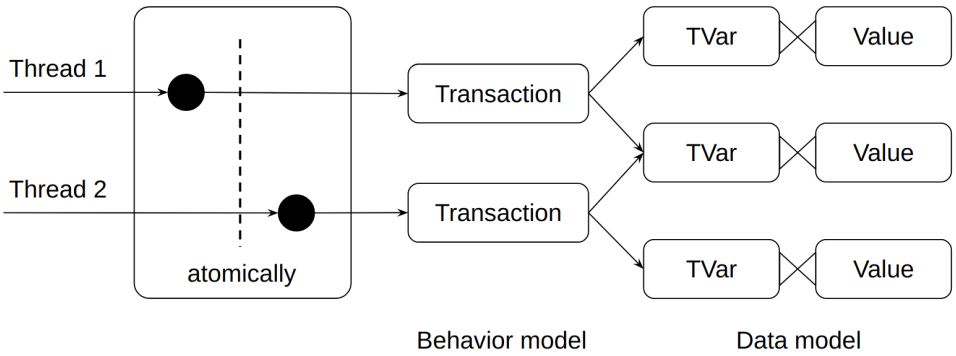


Figure 6.7 STM model

Like many functional concepts, STM follows the “divide and conquer” principle, and like many functional concepts, it separates data models from modification (behavior) models. Using STM primitives (like queues, mutable variables, or arrays), you first construct a data structure for your domain — this will be your concurrent domain model. Second, using special functions from the STM library, you write transactions to access your data structure or its parts; doing so, you'll get a behavior model that’s guaranteed to be thread-safe. Finally, you run your transactions in a threaded environment, passing them an instance of the domain model. Figure 6.8 offers some basic intuition about STM and the division of the data and behavior models.

Figure 6.8 Data model and behavior model in STM

A typical STM library has several primitives that are used to construct concurrent data structures. For example, the STM primitive `TQueue` from Haskell's `stm` library is the analogue of `ConcurrentQueue` from the .NET framework. Both are First in First Out (FIFO) queues, both are thread-safe, and both have similar methods for writing and reading values to and from a queue. However — and this is how STM differs from just concurrent collections —



access to `TQueue` can be done through an evaluation of transactions only together with guarantees of atomicity. Strictly speaking, the synchronization of access to STM primitives is taken out of the primitives themselves, in contrast to concurrent collections, where synchronization is burned inside. This becomes more important when we consider that having several distinct operations over some STM data structure lets us combine these operations into one big operation that will be evaluated atomically in the `IO` monad. Here's the combinator:

```
atomically :: STM a -> IO a
```

where `STM` is the transactional monad type. In other words, *STM concurrency is composable* in the sense of true functional programming composition. So, every monadic code in the `STM` monad is a separate transaction that may be evaluated over data or may be composed with some other transaction to obtain a bigger one. Monadic STM operations may be thought of as combinators of transactions.

Another important idea in STM is the *retrying* operation. Any transaction can be rolled back and retried again after a while. For example, some thread A has committed its own transaction in which a transactional queue (`TQueue`) variable has been modified. At the same time, thread B happens to be in its own transaction, which tries to take an item from the same queue. However, the second thread should roll back its transaction because it sees that the queue has changed. Figure 6.9 clarifies this.

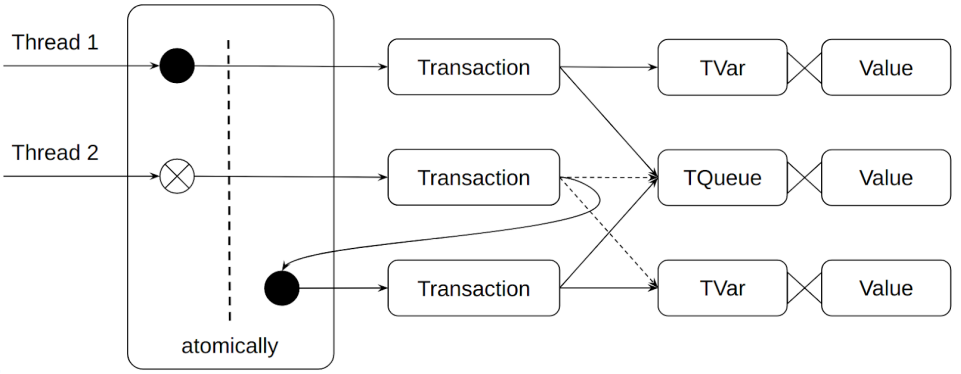


Figure 6.9 Retrying a transaction when concurrent changes occur

The retrying operation is automatic. However, it's often necessary to wait for a particular state of transactional variables to continue a transaction. The `retry` combinator can be used to block a thread until some transactional variable is changed and to retry the transaction again. For instance, you have a boolean `TVar` (transactional variable, the analogue of `IORef`), and you want it to be `True` to enter a transaction. You read your variable within the transaction, test it, and, if it holds `False`, you call a `retry` action:

```
transaction :: TVar Bool -> STM Int
transaction tvFlag = do
  flag <- readTVar tvFlag
  if flag
    then return 100
    else retry
```

The transaction will be restarted in order to return `100` when and only when `tvFlag` is changed to `True`. If there are no other threads that can change the `tvFlag` variable, this thread will be blocked forever.

> **TIP** Indefinite blocking of a thread is considered an exceptional situation. Depending on the STM implementation, you will or won't get some exception thrown.

The `retry` combinator is a very powerful STM tool, although it's very simple:

```
retry :: STM a
```

With this combinator, it becomes possible to do “wise tricks” in concurrent code. The `retry` combinator and other STM operations can look like magic because the code remains human-readable and short compared to the maintenance hell with mutexes, conditional variables, callbacks, and other imperative things that overburden and buggify parallel imperative code.

> **NOTE** Computer scientists who have researched STM note that this retrying approach has some performance impact, but this impact is acceptable when weighted against the easiness you get in constructing correct concurrent models. You can find a much deeper look into STM in such books as *Parallel and Concurrent Programming in Haskell* (by Simon Marlow) and *Real World Haskell* (by Bryan O'Sullivan, Don Stewart, and John Goerzen). There are also good materials about STM in Clojure.

> **LINK** Bryan O'Sullivan, Don Stewart, and John Goerzen, *Real World Haskell*
> http://book.realworldhaskell.org/

6.2.2 Reducing complexity with STM

Recently, our program was using `MVar` for the safe synchronization of the threads counter on the implementation layer. `MVar` is fine; it behaves similarly to mutexes and gives reliable concurrency guarantees like the following:

- All threads will be woken up when a current owner thread releases the `MVar`.
- After taking the `MVar` by a thread, there’s no way for others to observe the internals of the `MVar`.
- The program will be terminated if a thread took the `MVar` and died, leaving all other threads blocked.

Still, `MVar` requires careful code writing because it’s so easy to make a bug when either of the two operations — taking and releasing `MVar` — is left

unpaired and blocks the program. More complex concurrent data structures based on many `MVars` will amplify this problem exponentially.

Let's see how STM changes the game rules here. First, we replace `MVar` with a transactional variable `TVar` — it will be enough for the counter:

```
data Runtime = Runtime
  { _maxThreadsCount :: Int
  , _curThreadsCount :: TVar Int
  }
```

Now, the most pleasant part. Functions for increasing and decreasing the counter will be separate transactions over this `TVar`. The increasing function should track the state of the counter and decide when to increase and when to block. With the help of a magical STM combinator `retry`, the code becomes very simple:

```
-- Increasing the counter thread safely.
-- Block if the pool is on its maximum.
increaseProcessCount :: Int -> TVar Int -> IO ()
increaseProcessCount maxCount psVar = atomically $ do
  ps <- readTVar psVar
  when (ps == maxCount) retry       -- block here
  writeTVar psVar $ ps + 1

-- Decreasing the counter thread safely.
decreaseProcessCount :: TVar Int -> IO ()
decreaseProcessCount psVar =
  atomically $ modifyTVar psVar (\x -> x - 1)
```

The interpreter becomes much simpler: no need to manually poll the state of the resource after exponential delays. When the resource isn't ready, STM will block the thread on the retry operation and will resume it after observing that `psVar` has changed. The interpreter code:

```
forkProcess' :: Runtime -> LangL () -> IO ()
forkProcess' rt act = do
  let psVar = _curThreadsCount rt
  let maxPs = _maxThreadsCount rt
  increaseProcessCount maxPs psVar  -- blocking here
  void $ forkFinally
    (runLangL rt act)
```

```
    (const $ decreaseProcessCount psVar)

interpretAppF :: Runtime -> AppF a -> IO a
interpretAppF rt (EvalLang act next) = do
  r <- runLangL rt act
  pure $ next r
interpretAppF rt (ForkProcess act next) = do
  forkProcess' rt act  -- block here
  pure $ next ()
```

As a result, we get a nice sequential code acting predictably, without race conditions. We can go further and propagate this practice to all concurrency we deal with in our program. For now, we have only threads bookkeeping on the implementation layer. Later, we may want to add facilities for serving a TCP/UDP/HTTP API, we may need to have asynchronous behavior in our scripts, we may be required to call external services in parallel, and so on.

We implemented a particular blocking semantics for our `forkProcess` method. This is our design decision, and it should be specified in the method's documentation. Now, what if we have a requirement to fork a process asynchronously when the pool is freed? We don't want to block the main thread in this case. Let's add one more method to our language and see what changes will be needed:

```
data AppF next where
  EvalLang         :: LangL a  -> (a -> next)  -> AppF next
  ForkProcess      :: LangL () -> (() -> next) -> AppF next
  ForkProcessAsync :: LangL () -> (() -> next) -> AppF next

type AppL = Free AppF

-- Evaluate a LangL script.
evalLang :: LangL a -> AppL a

-- Fork a process. Block if the pool is full.
forkProcess :: LangL () -> AppL ()

-- Fork a process asynchronously.
--   Do not block if the pool is full.
forkProcessAsync :: LangL () -> AppL ()
```

The interpreter will get one more part; it's very simple:

```
interpretAppF rt (ForkProcessAsync act next) = do
  void $ forkIO $ forkProcess' rt act  -- do not block here
  pure $ next ()
```

However, we should understand that we introduced the same problem here. The thread counter doesn't have any meaning now. It's very possible to fork tenths of intermediate threads, which will be waiting to run a real working thread. Should we add one more thread pool for intermediate threads? This sounds very strange. Do you have deja vu? The good thing here is that green threads in Haskell don't cost that much, and while they're waiting on the STM locks, they don't consume CPU. So, the leakage problem is mitigated a little. Still, it's better not to call the `forkProcessAsync` function in an infinite cycle.

6.2.3 Abstracting over STM

So far, we've dealt with `IORefs` in our business logic. This is pretty much unsafe in a multithreaded environment like we have. The `LangL` eDSL can create, read, and write `IORefs`. The language is still pure because it doesn't expose any impure semantics to the client code. It's still impossible to run `IO` actions because we've abstracted them out:

```
data LangF next where
  NewVar       :: a -> (IORef a -> next)       -> LangF next
  ReadVar      :: IORef a -> (a -> next)       -> LangF next
  WriteVar     :: IORef a -> a -> (() -> next) -> LangF next

type LangL = Free LangF

newVar       :: a             -> LangL (IORef a)
readVar      :: IORef a       -> LangL a
writeVar     :: IORef a -> a -> LangL ()

somePureFunc :: IORef Int -> LangL (IORef Int)
somePureFunc inputVar = do
  val <- readVar inputVar
  newVar $ val + 1
```

All the impurity moved to the implementation layer (into the interpreters). It would be worse if we made a slightly different decision on this. Consider the following code, where the `LangL` has a method to run `IO` actions:

```
data LangF next where
  EvalIO :: IO a -> (a -> next) -> LangF next

type LangL = Free LangF

-- Smart constructor
evalIO :: IO a -> LangL a
evalIO ioAct = liftF $ EvalIO ioAct id
```

The same interface for working with `IORef` can be expressed with this functionality:

```
newVar :: a -> LangL (IORef a)
newVar val = evalIO $ newIORef a

readVar :: IORef a -> LangL a
readVar var = evalIO $ readIORef var

writeVar :: IORef a -> a -> LangL ()
writeVar var val = evalIO $ writeIORef var val
```

The `somePureFunc` didn't change, but now we made a giant black hole that's threatening to explode our application unexpectedly:

```
somePureFunc :: IORef Int -> LangL (IORef Int)
somePureFunc inputVar = do
  val <- readVar inputVar
  evalIO launchMissiles       -- A bad effect here
  newVar $ val + 1
```

We could of course leave the `evalIO` function unexported and unavailable, but still. This is kind of an attraction: everyone should decide the degree of danger they're fine with. It might not be that bad to have black holes in your project if you have good discipline.

Nevertheless, it's impossible to calculate the number of variables that have been created, or to see who owns them and what will happen with these variables in a

multithreaded environment. A quick answer: it would be very dangerous to access a raw mutable `IORef` from different threads. This is automatically a race condition originated from the language design. We shouldn't be so naive as to think no one will step on this. Murphy's Law says it's inevitable. So what can we do? We can abstract working with state. And we can make the state thread-safe and convenient. We need STM in our business logic too.

There's an obvious way to run an STM transaction from the `LangL` script: either with `evalIO` or with a custom `atomically` function:

```
data LangF next where
  EvalIO :: IO a -> (a -> next) -> LangF next
  EvalStmAtomically :: STM a -> (a -> next) -> LangF next

type LangL = Free LangF

evalIO :: IO a -> LangL a
evalStmAtomically :: STM a -> LangL a
```

Correspondingly, the business logic will become

```
type ReportingChannel = TVar [Meteor]

reportMeteor' :: ReportingChannel -> Meteor -> STM ()
reportMeteor' ch meteor = do
  reported <- readTVar ch
  writeTVar ch $ meteor : reported

reportMeteor :: ReportingChannel -> Meteor -> LangL ()
reportMeteor ch meteor =
  evalStmAtomically $ reportMeteor' ch meteor
```

Sure, why not. The interpreter will transform this call into a real `atomically`:

```
interpretLangF rt (EvalStmAtomically stmAct next) = do
  res <- atomically stmAct
  pure $ next res
```

All the STM facilities — `TVars`, `TQueues`, `TArrays` — will be available right from the scripts. This design is fast and fine in general, unless you need full control over the state in your app. You might want to

- Introspect the current application state and see what variables are actively used at this moment
- Limit the number of variables produced
- Be able to persist and restore your application state
- Have a consistent set of languages

Let's do the full abstraction over STM and see what benefits and flaws it gives us.

For the sake of modularity, we'll create the `StateL` language and extract it from `LangL`, as in figure 6.4. Single language, single responsibility. This language will operate with a custom `StateVar`, which will represent a `TVar` in the business logic:

```
type VarId = Int

-- | Concurrent variable (STM TVar).
newtype StateVar a = StateVar
  { _varId :: VarId
  }
```

Let's take a look at the logic in the `StateL` monad (which is a direct analogue of the `STM` monad):

```
type ReportingChannel = StateVar [Meteor]

reportMeteor' :: ReportingChannel -> Meteor -> StateL ()
reportMeteor' ch meteor = do
  reported <- readVar ch
  writeVar ch $ meteor : reported

reportMeteor :: ReportingChannel -> Meteor -> LangL ()
reportMeteor ch meteor = atomically $ reportMeteor' ch meteor
```

The language itself:

```
data StateF next where
  NewVar   :: a -> (StateVar a -> next) -> StateF next
  ReadVar  :: StateVar a -> (a -> next) -> StateF next
  WriteVar :: StateVar a -> a -> (() -> next) -> StateF next
  Retry    :: (a -> next) -> StateF next

type StateL = Free StateF
```

The `LangL` eDSL will have its own `atomically` function:

```
data LangF next where
  EvalStateAtomically :: StateL a -> (a -> next) -> LangF next

type LangL = Free LangF

atomically :: StateL a -> LangL a
atomically act = liftF $ EvalStateAtomically act id
```

There's nothing new in the design of these two languages. We just nest `StateL` into `LangL`. The `StateL` monad defines a transactional context, exactly as the `STM` monad does. It's not a coincidence that we just repeated the interface STM for `TVars`. The `StateVar` value keeps an integer reference to the actual `TVar`, but not the value itself. The corresponding `TVar` will be hidden in the runtime of the interpreter. When a concurrent value is requested from the logic, the `StateVar` reference will be used to obtain the corresponding `TVar`. The interpreter of the `StateL` language will track the variables for us. The text that follows describes the pattern in detail.

The interpreter for `StateL` should translate the actions to the STM environment. This is because all the monadic chains of the `StateL` monad should be a single transaction, and we shouldn't evaluate each method separately. Also, another runtime structure is needed to store `StateVars` and the corresponding `TVars`:

```
newtype VarHandle = VarHandle (TVar Any)

data StateRuntime = StateRuntime
  { _varId  :: TVar VarId
  , _state  :: TMVar (Map VarId VarHandle)
  }
```

Notice that we have to store `TVars` in an untyped form. Conversion between `GHC.Any` and a particular type will be done with `unsafeCoerce`, which is fine because the `StateVar` is always typed and keeps this type on the business logic layer. This, for the moment, is another interesting idea: a typed eDSL and business logic that runs over an untyped runtime without sacrificing type safety. I call this design pattern *Typed avatar* (see figure 6.10).

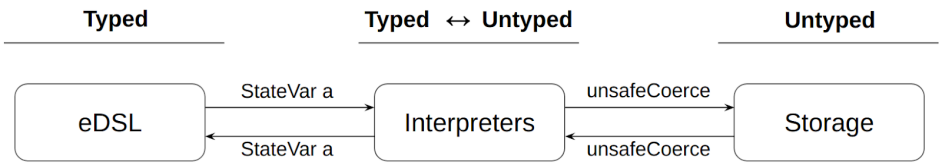


Figure 6.10 The Typed avatar pattern

The implementation details aren't that important; let's take a look at the shape of the interpreter only:

```
interpretStateF :: StateRuntime -> StateF a -> STM a

runStateL :: StateRuntime -> StateL a -> STM a
runStateL stateRt = foldFree (interpretStateF stateRt)
```

Nothing fancy, but internally the code works with the variables catalogue using STM operations. Running the interpreter against a `StateL` means composing a particular transaction that can create, read, and write `TVars`. The `retry` method is also available. In this design, you can do some intermediate operations and keep the results in the `StateRuntime`. A very obvious next step is to interpret the state:

```
data Runtime = Runtime
  { _stateRuntime :: StateRuntime
  }

interpretLangF :: Runtime -> LangF a -> IO a
interpretLangF rt (EvalStateAtomically action next) = do
  res <- atomically $ runStateL (_stateRuntime rt) action
  pure $ next res
```

In the Free monad languages hierarchy, the pieces organized in a logical structure will match perfectly. The clear separation of concerns and the ability to have different kinds of runtime make it easy to design a semantics for eDSLs and control every single aspect of its evaluation. Other interesting things can be implemented on top of these ideas. Some useful patterns are ahead in this chapter.

6.3 Useful patterns

In this section, we'll see how to embed a logging language into the `StateL` eDSL and touch on some questions about reactive programming. With a good fantasy, you can find even more use cases for Free monadic STM. For example, it makes sense to accompany the `StateVar` data type with such concurrent primitives as `TQueue`, `TChan`, and `TArray`. These can all be implemented via `StateVar`, or maybe it's better to add your own wrappers for them.

One small note about performance. `StateVar` in its current design isn't as performant as `TVar` because each time we use it, we have to request a real `TVar` from the internal `Data.Map` structure. And the map structure is protected by another concurrent variable — `TMVar`:

```
_state :: TMVar (Map VarId VarHandle)
```

If this is a problem, we can refuse to track the variables. The `StateVar` data type can be modified so that the actual `TVar` is stored in it:

```
data StateVar a
  = StateVar { _varId :: VarId }
  | UntrackedStateVar (TVar a)
```

We need a new method in the `StateL` language:

```
data StateF next where
  NewUntrackedVar :: a -> (StateVar a -> next) -> StateF next
```

The interpreter should respect this change. Its full code might look like this:

```
interpretStateF :: StateRuntime -> StateF a -> STM a

interpretStateF stateRt (NewUntrackedVar !val next)
```

```
  = next . UntrackedStateVar <$> newTVar stateRt val

interpretStateF stateRt (NewVar !val next)
  = next . StateVar <$> newVar' stateRt val

interpretStateF stateRt (ReadVar var next)
  = next <$> readVar' stateRt var

interpretStateF stateRt (WriteVar var !val next)
  = next <$> writeVar' stateRt var val

interpretStateF _ (Retry _) = retry
```

where `newVar'`, `readVar'`, and `writeVar'` are functions that can obtain a referenced `TVar` from the `StateRuntime`, for example:

```
readVar' :: StateRuntime -> StateVar a -> STM a
readVar' stateRt (UntrackedStateVar tvar) = readTVar tvar
readVar' stateRt (StateVar !varId) = do
  nodeState <- readTMVar $ _state stateRt
  case Map.lookup varId nodeState of
    Nothing -> error $ "Var not found: " +|| varId ||+ "."
    Just (VarHandle tvar) -> unsafeCoerce <$> readTVar tvar
```

So, the design has some flexibility and can be tuned according to your needs.

6.3.1 Logging and STM

Let me ask a question: if you have a `StateL` (STM) transaction, how would you log something inside it? All external effects are prohibited, and the transaction might be restarted many times. Logging here isn't usually needed because we want to keep our transactions as short as possible. We can log something post factum when the transaction has finished:

```
-- Transaction, returns a message to log.
reportMeteor' :: ReportingChannel -> Meteor -> StateL String
reportMeteor' ch meteor = do
  reported <- readTVar ch
  writeTVar ch $ meteor : reported
  pure $ "Meteors reported currently: "
       <> show (1 + length reported)
```

```
-- Evaluates the transaction and outputs the message as log.
reportMeteor :: ReportingChannel -> Meteor -> LangL ()
reportMeteor ch meteor = do
  msg <- atomically $ reportMeteor' ch meteor
  logInfo msg        -- Method of the LoggerL eDSL
```

But what if we do want to log inside `reportMeteor`? Or we have multiple log calls to make? Should we return a list of strings from the function? Although it's possible, it's not a good decision. We obligate the caller function to process our logs. A very strange convention.

```
-- Returns multiple log entries.
reportMeteor' :: ReportingChannel -> Meteor -> StateL [String]
```

There's a possible solution that involves the application state. We add a special variable for collecting log entries while the transaction evaluates. After the transaction finishes, we flush this collection of logs into a real logging subsystem. The next code listing shows a new application state type and logs collection (listing 6.3).

Listing 6.3 Collecting log entries in the application state

```
data AppState = AppState
  { _reportingChannel :: ReportingChannel
  , _logEntries :: StateVar [String]
  }

-- Transaction, which updates log entries
logSt :: AppState -> String -> StateL ()
logSt st msg = modifyVar (_logEntries st) (\msgs -> msg : msgs)

-- Transaction, which collects log entries
reportMeteor' :: AppState -> Meteor -> StateL String
reportMeteor' st meteor = do
  logSt st "Reading channel"
  ...
  logSt st "Writing channel"
  ...

-- Service function, which flushes log entries
flushLogEntries :: AppState -> LangL ()
flushLogEntries st = atomically getLogs >>= mapM_ logInfo
```

```
  where
    getLogs = do
      msgs <- readVar (_logEntries st)
      writeVar (_logEntries st) []
      pure msgs

-- Evaluates the transaction and flushes the entries
reportMeteor :: AppState -> Meteor -> LangL ()
reportMeteor st meteor = do
  atomically $ reportMeteor' st meteor
  flushLogEntries st
```

The problems with this approach seem obvious:

- Need to keep additional variables for log entries, and it has to be concurrent.
- Passing the state variable here and there.
- Explicit function for logs flushing that you need to remember.
- Not appropriate in the business logic.
- Only supports the “info” level of log messages. To support more levels, you’ll need more functions, more variables, or a kind of generalization.

A much better solution is to move transactional logging to the next layer down. We’ll do a similar thing, essentially, but we’ll be collecting messages in the interpreter’s state rather than in the business logic’s state. To improve user experience, we’ll add a special method into the `StateL` language, so it will be possible to do logging from a `StateL` script and not think about its flushing. The logs will be flushed automatically in the interpreters when a transaction is done. Let’s call this approach *delayed STM logging*.

Figure 6.11 explains this idea schematically.

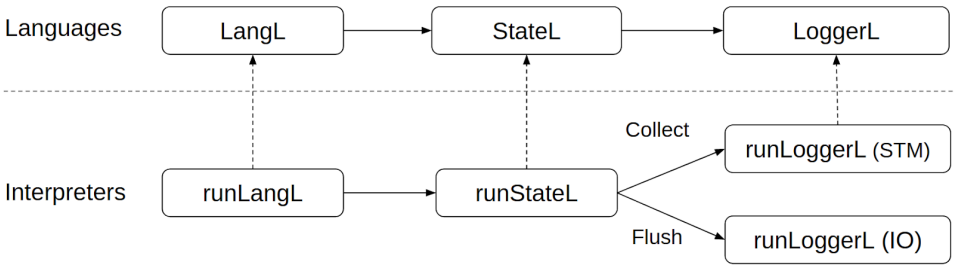


Figure 6.11 Schema of delayed STM logging

Words are cheap, show me the code! First, the `LoggerL` language:

```
data LogLevel = Debug | Info | Warning | Error
type Message = Text

data LoggerF next where
  LogMessage
    :: !LogLevel
    -> !Message
    -> (() -> next)
    -> LoggerF next

type LoggerL = Free LoggerF

logMessage :: Level -> Message -> LoggerL ()
logMessage level msg = liftF $ LogMessage level msg id

logInfo :: Message -> LoggerL ()
logInfo = logMessage Info
```

Second, updates in the `StateL`:

```
data StateF next where
  EvalStmLogger :: LoggerL () -> (() -> next) -> StateF next
```

Third, an additional variable in the runtime to keep delayed log entries (`LogEntry` is a type describing an entry, its level, and message):

```
data LogEntry = LogEntry LogLevel Message
```

```
data StateRuntime = StateRuntime
  { _varId  :: TVar VarId
  , _state  :: TMVar (Map VarId VarHandle)
  , _stmLog :: TVar [LogEntry]
  }
```

Third, add a new STM-based interpreter for `LoggerL`; notice that it writes all the entries into a concurrent variable:

```
-- | Interpret LoggerF language for a stm log.
interpretStmLoggerF :: TVar LogEntry -> LoggerF a -> STM a
interpretStmLoggerF stmLog (LogMessage level msg next) =
  next <$> modifyTVar stmLog
                      (\msgs -> LogEntry level msg : msgs)

-- | Run LoggerL language for a stm log.
runStmLoggerL :: TVar Log -> LoggerL () -> STM ()
runStmLoggerL stmLog = foldFree (interpretStmLoggerF stmLog)
```

Fourth, call this interpreter from the `StateL` interpreter. Notice how the interpreters in `STM` match nicely:

```
interpretStateF :: StateRuntime -> StateF a -> STM a
interpretStateF stateRt (EvalStmLogger act next) =
  next <$> runStmLoggerL (_stmLog stateRt) act
```

Finally, flushing the STM logs from the `LangL` interpreter after each transaction evaluation:

```
interpretLangF :: Runtime -> LangF a -> IO a
interpretLangF rt (EvalStateAtomically action next) = do
  let stateRt   = _stateRuntime rt

  -- Doing STM stuff
  res <- atomically $ runStateL stateRt action

  -- Flushing
  flushStmLogger stateRt

  pure $ next res
```

Now the business logic code will be looking pretty neat:

```
anyStateFunc :: StateL (StateVar Int)                #A
anyStateFunc = do
  logInfo    "Some info in StateL"
  logWarning "Some warning in StateL"
  newVar 10

anyLangFunc :: LangL ()
anyLangFunc = do
  logInfo "Some info in LangL"
  var <- atomically anyStateFunc                     #B
  writeVar var 20
```

#A Transaction that updates log entries
#B Logs will be flushed here automatically

Reducing complexity with this pattern became possible because we abstracted the state, the logger, and the language for business logic. This is a benefit of hierarchical organization of languages, too. It's pretty satisfying, isn't it?

> **NOTE**. Well, I silently kept quiet about the fact that we can't use the same functions `logInfo`, `logWarning` for both `StateL` and `LangL` monads — at least, not the monomorphic versions of these functions. The complete solution includes a type class `Logger`, which is instantiated for `StateL` and `LangL` (as well as for `AppL`); after this, the logging functions will be polymorphic enough to be used everywhere.

6.3.2 Reactive programming with processes

The attentive reader might have noticed that something is wrong with the `app` function in which we started our processes. Let me remind you of the relevant piece of code from listing 6.3:

```
type ReportingChannel = StateVar [Meteor]

trackingCenter :: ReportingChannel -> LangL ()
trackingCenter ch = do
  reported <- readVar ch
  -- do something with the reported meteors
  writeVar ch []
  delay 10000
```

```
app :: AppL ()
app = do
  ch <- evalLang $ newVar []
  process  $ forever $ astronomer ch
  scenario $ forever $ trackingCenter ch
```

When we start this application, it will be working forever because the latest row mandates running the `trackingCenter` function infinitely many times. In reality, this usually isn't what we would need. Our applications should be properly terminated when some condition is met. It's either an external command (from the CLI, from the network, and so on), or it's an internal condition signaling that the application has reached some final state. Irrespective of the reason, we would have our business logic to gracefully finish all the activities. With STM, we can easily make the code reactive, so it can adapt its behavior to the dynamically changing situation.

Let's first interrupt the evaluation of the `app` function at the point when the total number of meteors exceeds the max count. For this, we'll need to reimplement the `trackingCenter` function and introduce a kind of catalogue — which, in fact, should be in the app anyway because this is what the program was intended for. The only valid place for the catalogue is `AppState`:

```
type ReportingChannel = StateVar [Meteor]
type Catalogue        = StateVar (Set Meteor)

data AppState = AppState
  { _reportingChannelVar :: ReportingChannel
  , _catalogueVar        :: Catalogue
  }
```

Now, let's add collecting logic. This new function will take meteors from the channel and put them into the catalogue, returning the total number of items tracked:

```
trackMeteors :: AppState -> StateL Int
trackMeteors st = do
  reported  <- readVar $ _reportingChannelVar st
  catalogue <- readVar $ _catalogueVar st
```

```
  writeVar (_catalogueVar st)
    $ foldr Set.insert catalogue reported
  writeVar (_reportingChannelVar st) []

  pure $ length reported + Set.size catalogue
```

We now need to react to the number of tracked meteors somehow. Currently, the tracking center doesn't operate as a separate process. We could rewrite `trackingCenter` and `app` so that it finishes after a certain condition. To do this, we need a manual recursion instead of `forever`:

```
trackingCenter :: AppState -> Int -> LangL ()
trackingCenter st maxTracked = do
  totalTracked <- atomically $ trackMeteors st

  -- Manual recursion
  when (totalTracked <= maxTracked) $ do
    delay 10000
    trackingCenter st maxTracked

-- Initializing the state
initState :: LangL AppState
initState = atomically $ do
  channelVar   <- newVar []
  catalogueVar <- newVar Set.empty
  pure $ AppState channelVar catalogueVar

app :: AppL ()
app = do
  let maxTracked = 1000      -- Finish condition
  st <- initState            -- Init app state

  process  $ forever $ astronomer st
  scenario $ trackingCenter st maxTracked
```

Now it looks fine. The `app` function will finish among the `trackingCenter` recursive function. But … is this that beautiful? On the program end, we don't care about the astronomer process at all! Can the tracking center kill the astronomer and finish itself afterwards? Sounds terrible, but this is what computer threads usually do!

This new task requires more reactive programming. Let's clarify the algorithm:

1. The `astronomer` process starts working.
2. The `trackingCenter` process starts working.
3. The `trackingCenter` process checks the number of meteors tracked.
    a. It does not exceed the max count — continue.
    b. It exceeds the max count — goto 4.
4. The `trackingCenter` signals to the `astronomer` process to finish.
5. The `trackingCenter` waits for the `astronomer` process to actually finish.
6. The `trackingCenter` finishes.

As you can see, this algorithm requires the tracking center process to know about the astronomer process. The latter should signal back when it's about to finish. How can these processes interact? By using the signaling `StateVars`. There are a few schemes of this interaction involving either one or more `StateVars`. Deciding which scheme to choose may be a situational task. For our purposes, let's have two signaling variables. The code you'll see next isn't the shortest way to do it, but it's good enough for demonstration purposes.

The first signal variable — `appFinished` — will represent a condition for the whole application. Perhaps `AppState` is the best place for it:

```
type SignalVar = StateVar Bool

data AppState = AppState
  { _reportingChannelVar :: ReportingChannel
  , _catalogueVar        :: Catalogue
  , _appFinished         :: SignalVar              #A
  }
```

#A When this variable is True, all child threads should finish

The second signal variable should be owned by a certain process. Using this variable, the process will notify the main thread that it has finished. Let's see how the astronomer process should be reworked to support this "cancellation token" (see listing 6.3 as the base code):

```
astronomer :: AppState -> SignalVar -> LangL ()
astronomer st doneVar = do
  rndMeteor <- getRandomMeteor
  rndDelay  <- getRandomInt (1000, 10000)
  reportMeteor st rndMeteor

  finish <- atomically $ readVar $ _appFinished st

  if finish
     then atomically $ writeVar doneVar True
     else do
       delay rndDelay
       astronomer st
```

Once the process gets a signal to quit, it sets up its `doneVar` variable and finishes. Now, let's rework the `trackingCenter` function:

```
trackingCenter :: AppState -> Int -> LangL ()
trackingCenter st maxTracked = do
  totalTracked <- atomically $ trackMeteors st

  -- Manual recursion
  if (totalTracked <= maxTracked)
    then do
      delay 10000
      trackingCenter st maxTracked
    else atomically $ writeVar (_appFinished st) True
```

Looks very similar, either doing a manual recursion or finishing with signaling. Now, we need to rework the `app` script. Notice that we've moved `trackingCenter` to its own process and removed the `forever` combinator from everywhere:

```
app :: AppL ()
app = do
  let maxTracked = 1000                                  #A
  st <- initState
  doneVar <- atomically $ newVar False                   #B
  process $ astronomer st doneVar
  process $ trackingCenter st maxTracked

  -- Blocking while the child process is working
```

```
  atomically $ do
    done <- readVar doneVar
    unless done retry

  -- Just for safety, wait a bit before really quit:
  delay 1000
```

```
#A Finish condition
#B Signaling variable for astronomer
```

The last part of the code will block the execution of the whole `app` until the `readVar` becomes `True`. In fact, it's possible to expand the example for multiple astronomer threads, and the pattern will handle this situation as well! Check it out:

```
startAstronomer :: AppState -> AppL SignalVar
startAstronomer st = do
  doneVar <- atomically $ newVar False
  process $ astronomer st doneVar
  pure doneVar

app :: AppL ()
app = do
  let maxTracked = 1000
  st <- initState

  -- Starting 10
  doneVars <- replicate 10 $ startAstronomer st

  process $ trackingCenter st maxTracked

  -- Waiting for all child processes to finish:
  atomically $ do
    doneSignals <- mapM readVar doneVars
    unless (all doneSignals) retry

  -- Just for the safety, wait a bit before really quit:
  delay 1000
```

STM provides even more opportunities to write concise code that works very well. There are, of course, some subtle things regarding a correct usage of STM

and threads, yet these difficulties aren't as big as when programming bare threads with traditional approaches to concurrency.

6.4 Summary

In this chapter, we learned several approaches to concurrency in multithreaded applications. We've developed an architecture that allows us to divide the application into layers: DSLs organized hierarchically into business logic and implementation layer. It was shown that there's real value in separating responsibilities. Thus, having a language `LangL` that can't fork threads or directly work with impure mutable state made the application more structured and prevented unwanted things like thread leakage — at least on the level of this language. Also, we introduced a higher-level language `AppL` that can fork processes and do the necessary initialization. This language shouldn't be used for actual business logic, but it declares the environment for this logic. Hierarchical eDSLs gave us another layering within the business logic layer and allowed us to control the responsibilities more granularly.

We can use STM not only as a concurrently safe application state but also as a mechanism for reactive programming. In fact, it's possible to turn a business logic into FRP code without changing the current eDSL layer. STM provides a composable and declarative way of defining thread-safe concurrent data types, relations with threads, and operational state for the application.

We've wrapped STM and processes into specific languages and embedded them into the hierarchy of Free monad languages. This allowed us to abstract and retrospect the usage of state variables, implement a kind of thread pool, and avoid resource leakage. We were also able to implement an implicit yet transparent and obviously behaving logging for our `StateL` language.

In conclusion, the chosen architecture of the application gave us many benefits as well as opportunities for future improvements.

Chapter 7
Persistence

This chapter covers

- Designing relational databases and key-value database support
- Embedding a type-safe SQL library
- How to create database data models
- Type safety and conversion questions
- Transactions and pools

While travelling in space, you encounter a cute solar system with a nice, small planet near a bright star. When you investigate the planet, you discover life on it. The creatures are really social — they interact with each other, but in a very simple way. You swear they're just nodes in an unstructured tree, and that they're connected by randomly displaced edges. You can easily find several roots, many intermediate nodes, and millions of leaf nodes. What an interesting life! After watching them for a while, you realize that similar patterns repeat once and once again in their interactions. These interactions start from the leaves; in there, each leaf node gets a sequential index; then a calculation starts in the parent nodes. This calculation is rather simple. You find that it's just a summation of all the child indexes; and once this procedure is done, the next level of nodes starts doing the same thing, and the calculation repeats until it hits the top root node. The number of the node that is essentially a summation of all the indexes of all the leaves is `-1/12`. After getting this result, the tree with all

the nodes disappears, and a new, very much the same tree is restored from the nearest gigantic bin. And all the calculations start over again.

Now think. Do you want a magic button that saves your life into an external storage so that you can occasionally restore it? What if the only condition is that your life will start from the moment of saving, with all your recent experience, knowledge, and impressions lost? That's not that attractive, right?

Well, investigating this alien society more closely, you find a problem that prevents their tree from being stored back to the bin once they finish their calculations. Fortunately, you can fix this bug, and — magically — the society will continue to live further and develop their unique, non-circularizing history.

This means you can probably consider saving your life as well …

7.1 Database model design

It's a rare case in which a program doesn't interact with any external data storage, whether it's a relational database, key-value database, filesystem, cloud storage, or something else. While the model of interaction may vary, the idea is always the same: to get access to a significant amount of data that can't be located directly in the program's memory. Observing all the possible data storages and the nuances of their use isn't possible, even with hundreds of books dedicated to only this purpose. It's too much to consider, and isn't our interest here. Why should a key-value cache be chosen over a relational database, what are the differences between various file storage systems, how do we tame the beast of cloud services … not this time, not in this book.

Instead, we'd do better to focus on design patterns and architecture approaches. We can and probably should challenge ourselves to make interaction with these systems more correct, robust, simple, convenient, and so on. We want to discuss the paths to abstraction of the related code. We might like to introduce a typical pattern to solve typical problems. So, we're here to see how database support can be implemented with the Free monad approach.

We'll start with some basics, though. Let's introduce one more layer of our architecture — a database model — and talk about its design and best practices.

7.1.1 Domain model and database model

We learned a lot about domain models in the previous chapters. In fact, the second part of the book was dedicated to the methodology known as domain-driven design. Although it's a very good and very widespread methodology, it hasn't always been so. Before the idea of designing systems from a domain point of view became popular, developers designed applications starting from a database model. They would first outline a relational representation of the task, and could then introduce the needed data structures in the code. But this methodology had some flaws. The main disadvantage was that the database representation types had a significant bias in favor of a relational structure that wasn't as intuitive as a domain-favored structure could be. It's not really convenient to operate by the database model compared to the corresponding domain model. This distinction was rethought, and the approach "domain model first, database model second" began to prevail.

> **DEFINITION** A database model is a set of data types and structures that represent a database scheme, either directly or indirectly. Most commonly, these types are heavily infiltrated by the notions of databases such as primary and foreign keys, database-specific timestamps, and database-specific fundamental types.

It seems very beneficial and natural to separate domain and database models. Being independent, the database model can be extracted into its own project or library so that it can evolve without breaking the main code. The difference between models becomes deeper when the application grows in size and complexity. Strictly speaking, these models have their own lifetimes, and even the way we obtain the two may vary. Here, some mainstream technologies require generating a database model from the external XML representation. Sometimes, the database model can be absent or merged with a domain model; a number of important techniques help to organize such a code. For reference, consider getting familiar with the following approaches and design patterns from OOP: Repository, Active Record, and object-relational mapping (ORM). Of course, functional programming has its own substitutions for these things, and we'll see some of them here.

> **LINK** The Repository pattern in C#
> https://dotnettutorials.net/lesson/repository-design-pattern-csharp/

LINK Wikipedia: *Active record pattern*
https://en.wikipedia.org/wiki/Active_record_pattern

LINK Wikipedia: *Object-relational mapping*
https://en.wikipedia.org/wiki/Object-relational_mapping

7.1.2 Designing database model ADTs

Let's continue with our astronomical application. We forced it to generate random meteors in several threads. We got a simple domain model, though we've done nothing with it. Just counting the meteors can be done with a mere integer variable, unlike a meteor with size and mass. It's time to at least learn how to store and load those objects. Let's look closer at the `Meteor` domain data type:

```
data Meteor = Meteor
  { size :: Int
  , mass :: Int
  }
```

What if there are meteors with the same mass and size? One of them will be lost because there's no unique information that could distinguish a particular space stone. For example, there could be dimension and time coordinates that make meteors unique. There's the `Region` type, but it's too simple and fuzzy. Astronomers might find it too arbitrary. Real coordinates like azimuth, altitude, and time would be enough for a start:

```
data Meteor = Meteor
  { size      :: Int
  , mass      :: Int
  , azimuth   :: Float
  , altitude  :: Float
  , timestamp :: DateTime
  }
```

We can now process meteors and not be afraid to get them confused, because in the physical world, it's not possible for two different meteors to be in the same point of space at the same time. So, we're within our rights to design a relational database now. It will contain only a single table. The corresponding data model

will be represented by the following ADT (we'll talk about underscored fields a bit later):

```
data DBMeteor = DBMeteor
  { _size      :: INTEGER
  , _mass      :: INTEGER
  , _azimuth   :: FLOAT
  , _altitude  :: FLOAT
  , _timestamp :: TIMESTAMP
  }
```

We have to use types that a specific database can understand. Most likely, special data types will come with particular libraries, but we're assuming that we have `INTEGER, FLOAT,` and `TIMESTAMP` standard SQL types for our needs. Figure 7.1 shows the relation between the two models.

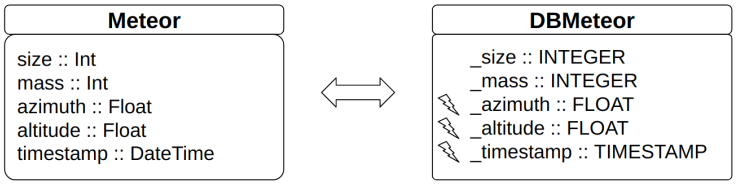


Figure 7.1 Simple domain and database models

Later, the `Meteor` type can be changed somehow, for example, by extracting the `azimuth` and the `altitude` fields into an additional `Coords` type:

```
data Coords = Coords
  { azimuth   :: Float
  , altitude  :: Float
  }
```

The separation of the two models will protect the database-related code from being affected. But to make it really robust, you'll want to concentrate the changes in a single place. You shouldn't allow the database types to leak into the business logic, and all the conversion between the two models should be done through a bunch of functions:

```
fromDBMeteor :: DBMeteor -> Meteor
toDBMeteor :: Meteor -> DBMeteor
```

This isn't only about the separation of the two models but is also about their isolation from each other. The thinner the edge is, the easier it will be to handle the problems in case of friction between the subsystems.

Moving along, we have more topics to discuss. Looking at figure 7.1, you might have noticed that we marked the three fields to be a composite natural primary key: `(FLOAT, FLOAT, TIMESTAMP)`. This solution has two big drawbacks at once: 1) float numbers should not be used as keys; 2) natural keys should be avoided except when there are strong considerations for their use. In the next section, we'll see how to solve these design flaws, and, by doing this, we'll find a good place for the so-called Higher-Kinded Data pattern (HKD).

7.1.3 Primary keys and the HKD pattern

You'd better avoid using float numbers as a part of a primary key. The main reason is that you can't be sure of the exact value due to the rounding issues of float numbers. By default, float data types in all programming languages support the equality operation, so you can compare them strictly. But you probably shouldn't do this. We all know that the following code, while logically correct, produces a counterintuitive result:

```
ghci> 0.1 + 0.1 == 0.2
True
ghci> 0.1 + 0.2 == 0.3
False
```

This also means that float numbers as primary keys in databases are lurking beasts that will attack you when you least expect it.

Another consideration is related to natural keys. We expect a huge number of meteors, and we want to fetch them from the database for processing. Natural keys consisting of many fields aren't convenient for querying. They can be very inefficient because they require extra memory in relations between tables. Natural keys are hard to maintain. What if these fields should be slightly updated? Should we update all the data in the database? And what about backward compatibility? More questions arise here. Sometimes, natural keys are fine, but more likely, we need a unique surrogate key. It can be an integer number, UUID, hash, or something else, like this:

```
data DBMeteor = DBMeteor
  { _meteorId :: INTEGER
  , _size :: INTEGER
  , ...
  }
```

Seems that we should update the domain type `Meteor` by adding an identification field to it. However, a problem occurs with this: how can we create a `Meteor` value if we don't have a primary key yet? We should either make the field `Maybe` or duplicate the `Meteor` type. All options are shown as follows:

```
data Meteor = Meteor
  { size :: Int
  , ...
  }

data MeteorPK = MeteorPK
  { meteorId :: Int
  , size     :: Int
  , ...
  }

data MeteorMbPK = MeteorMbPK
  { meteorId :: Maybe Int
  , size     :: Int
  , ...
  }
```

Multiplying the types (having both `Meteor` and `MeteorPK`) feels like unnecessary work. It's quite simple, but will double the number of overall domain-related types, as well as the database types, so in reality, you'll have a triple instantiation of every type. This is a boilerplate, and we can mitigate the problem a little by using an HKD pattern. This pattern allows us to encode a varying part of an ADT so that it's possible to choose what form of the type to use in certain situations. The following `Meteor` type will be an HKD type because it has one additional type parameter `idType`:

```
data Meteor' idType = Meteor
  { meteorId  :: idType
  , size      :: Int
  , mass      :: Int
  , azimuth   :: Float
  , altitude  :: Float
  , timestamp :: DateTime
  }
```

We can now declare several end-user types using this HKD template:

```
type MeteorID = Int
type RawMeteor = Meteor' ()
type MeteorMbPk = Meteor' (Maybe MeteorID)
type Meteor = Meteor' MeteorID
```

And we certainly need some conversion functions:

```
fromDBMeteor :: DBMeteor -> Meteor
toDBMeteor :: Meteor -> DBMeteor
fromRawMeteor :: RawMeteor -> MeteorID -> Meteor
```

This is quite simple, but the power of HKD types can be used for more impressive tricks. Let's take a little step away from the main theme and talk about some Haskell type-level magic that might be useful in your real practice.

7.1.4 Polymorphic ADTs with the HKD pattern

Earlier, I said that native SQL libraries can have their own data types, which will be directly mapped to actual SQL types. While there's a standard for SQL, many relational database management systems (RDBMS) provide a wider set of types. So do the bindings to those services. We can assume that our `INTEGER`, `FLOAT`, and `TIMESTAMP` types came from a native library `imaginary-sqlite`, but what if we want to incorporate another library `imaginary-postgres`? We'd have to create a Postgres-specific version of the same model in addition to the existing SQLite model and domain model:

```
data PgMeteor = PgMeteor
  { pgMeteorId  :: PgPrimaryKey
  , pgSize      :: PgInt
  , pgMass      :: PgInt
  , pgAzimuth   :: PgFloat
  , pgAltitude  :: PgFloat
  , pgTimestamp :: PgTimestamp
  }
```

That's not funny, the entities continue to multiply. Let's outline the current status in table 7.1.

Table 7.1 Variety of types

Type	Domain model	SQLite database model	Postgres database model
Integer	`Int`	`INTEGER`	`PgInt`
Floating point	`Float`	`FLOAT`	`PgFloat`
Timestamp	`DateTime`	`TIMESTAMP`	`PgTimestamp`
Primary key	`Int`	`INTEGER`	`PgPrimaryKey`

Having three similar things in a row (in our case, it's three types `DBMeteor`, `Meteor`, and `PgMeteor`) should make us think that there's some hidden pattern we could recognize and utilize in order to reduce the boilerplate. In Haskell, there are at least two possible ways: going Template Haskell or choosing the path of type-level magic. We want the second. With HKD and type families, we'll build a construction that will be able to produce each of the three meteor ADTs.

But first, we should articulate what we actually want to do:

- We want to have a single template ADT for meteors. This template should have all the fields, but the types of these fields should not be fixed.
- We want to somehow select a particular set of types. We should be able to specify what set of types is needed when we deal with `DBMeteor`,

`Meteor`, or `PgMeteor`.

- There should be a way to describe sets of types.

Let's assume that the types from the `imaginary-sqlite` library look like this:

```
newtype ID = ID Int
newtype INTEGER = INTEGER Int
newtype FLOAT = FLOAT Float
newtype TIMESTAMP = TIMESTAMP UTCTime
```

We don't know much about the types from the `imaginary-postgres` library because they're opaque. We only know that we can compose those values using smart constructors provided by the library:

```
toPgPrimaryKey :: Int -> PgPrimaryKey
mkPgInt :: Int32 -> PgInt
mkPgFloat :: Float -> PgFloat
mkPgTimestamp :: UTCTime -> PgTimestamp
```

These are the two sets. For future use, we need a mark for each set. This mark will be used to indicate what set we want to summon. A good way to declare such marks in Haskell is via empty ADTs:

```
data SQLite'
data Domain'
data Postgres'
```

We can't create values of these types, but we can pass the types themselves as selectors. This is how we'll get a needed ADT type with those marks:

```
type SQLiteMeteor = MeteorTemplate SQLite'
type PostgresMeteor = MeteorTemplate Domain'
type Meteor = MeteorTemplate Postgres'
```

These resulting types are quite usual: we can construct them as usual, and we can access the fields and pattern match over them. All operations will be the same except the field types. The following code demonstrates this for `Meteor` and `SQLiteMeteor`:

```
mTime :: UTCTime
mTime = UTCTime day seconds
  where
    seconds = secondsToDiffTime 0
    day = toEnum 1

testMeteor :: Meteor
testMeteor = Meteor 1 1 1 1.0 1.0 mTime

getSize :: Meteor -> Int
getSize (Meteor {size}) = size

testSQLiteMeteor :: SQLiteMeteor
testSQLiteMeteor = Meteor
  { meteorId  = ID 1
  , size      = INTEGER 1
  , mass      = INTEGER 1
  , azimuth   = FLOAT 1
  , altitude  = FLOAT 1
  , timestamp = TIMESTAMP mTime
  }

getSize' :: SQLiteMeteor -> INTEGER
getSize' (Meteor {..}) = size
```

As you can see, the fields of the `Meteor` data constructor behave like they're polymorphic ("getSize" functions return different types). It's not even a problem to have them in a single scope; they don't conflict with each other, which is nice.

Enough with the intrigue! Let's see how this works. The following HKD type is the foundation of the solution:

```
data MeteorTemplate typeSet = Meteor
  { meteorId  :: TypeSelector typeSet Id'
  , size      :: TypeSelector typeSet Int'
  , mass      :: TypeSelector typeSet Int'
  , azimuth   :: TypeSelector typeSet Float'
  , altitude  :: TypeSelector typeSet Float'
  , timestamp :: TypeSelector typeSet Timestamp'
  }
```

All the fields are indeed polymorphic because they can take any shape as long as this shape is allowed by the `TypeSelector`. This type family will check the

two parameters: a `typeSet` selector and a selector of a particular data type. The second one is needed because our type sets have more than one base type. For better clarity, I declared marks for each type:

```
data Id'
data Int'
data Float'
data Timestamp'
```

I could pass the usual types to the `TypeSelector` type family instead: `Int`, `Float`, `Timestamp`. It would work similarly because the `TypeSelector` type family doesn't bother about what we use as a mark. It just accepts two marks and turns into the target type. You can think about the phrase `TypeSelector Domain' Int'` as just `Int`, because after the type family is filled with these arguments, it implicitly becomes `Int`.

This is the `TypeSelector` type family itself:

```
type family TypeSelector typeSet t
  where
    TypeSelector SQLite' Id'        = ID
    TypeSelector SQLite' Int'       = INTEGER
    TypeSelector SQLite' Float'     = FLOAT
    TypeSelector SQLite' Timestamp' = TIMESTAMP

    TypeSelector Domain' Id'        = Int
    TypeSelector Domain' Int'       = Int
    TypeSelector Domain' Float'     = Float
    TypeSelector Domain' Timestamp' = UTCTime

    TypeSelector Postgres' Id'        = Int
    TypeSelector Postgres' Int'       = Int
    TypeSelector Postgres' Float'     = Float
    TypeSelector Postgres' Timestamp' = UTCTime
```

It looks like a table for pattern matching over the `typeSet` and `t` parameters. The table is limited by those three packs of types. To add support for more libraries, you'll need to update this type family. You can't update it from the outside. This is why such type families are called *closed type families*. They're closed for an extension other than editing the type family itself.

There are other tricks with the HKD pattern. I don't even think that we've discovered them all yet. The pattern is relatively simple, so it can be your first type-level spell to learn.

7.2 SQL and NoSQL database support

Our current task is to track meteors and other astronomical objects in a single database. There's no special requirement to it: we should be able to write and read stuff, and we're happy with a plain-old database application. Seems we don't have a broad relational model for the domain — we need a simple warehouse to store objects, that's it. Can a SQL database be used here? Yes, definitely. We'd want to query meteors with particular properties like "get meteors whose mass and size exceeds a dangerous threshold" or "get meteors that occurred in that region, at that period of time." Can we use a key-value database then? It depends. NoSQL databases provide different mechanisms for querying, and it's also possible to construct indexes and relations specifically for certain needs. For ours, it's enough to have simple database support with minimum functionality. We could maybe treat a SQL database as the main storage and a key-value database as a kind of cache.

Before we move further, let's fixate a version of a database model that will have a varying primary key field:

```
data Meteor' k = Meteor
  { _meteorId  :: k
  , _size      :: Int
  , _mass      :: Int
  , _coords    :: Coords
  , _timestamp :: DateTime
  }
  deriving (Show, Eq, Ord, Generic, ToJSON, FromJSON)

type MeteorID  = Int
type RawMeteor = Meteor' ()
type Meteor    = Meteor' MeteorID
```

Note that this `Meteor'` data type has an additional property: it can be serialized to and from JSON automatically. We enabled this quite nice possibility by deriving the `ToJSON` and `FromJSON` type classes provided by the famous `aeson` library. We can now transform the values of this type easily;

there are several functions for this in `aeson`: `encode`, `decode`, `toJSON`, `toJSONKey`, `fromJSONKey`, and others. You definitely can't miss the library, since it's ubiquitous.

The later development will show whether this domain model design is good enough or if we just invented something that doesn't comply with the KISS and YAGNI principles.

7.2.1 Designing an untyped key-value database subsystem

It seems like a good idea to demonstrate a way to substitute one database service with another without affecting the main logic. This will force us to design a high-level interface to the database subsystem in the form of a Free monadic language. This Free monadic language should be suitable for different implementations. We'll choose the following, well-known key-value database systems:

- *RocksDB*. Embedded database for key-value data.
- *Redis*. Remote, distributed, in-memory key-value database storage.

While both these key-value storage systems offer a must-have basis — namely, setting and getting a value by a key — there are unique properties that differentiate the database systems from each other. This is a problem because we can't unify these databases under a single interface: it will be too broad and implementation-specific. It's a very tempting challenge — to design such an interface so that the maximum capabilities are somehow represented in it; if we had such a goal, we could even profit from selling this solution. I'd do this! In the far, far future. When I become an expert in databases. For now, we have to limit our desires and narrow the requirements to the smallest subset of common features. At least we should implement

- *Storing/loading a value by a key*. Nice start. Not enough for most real-world scenarios, though.
- *Transactional/non-transactional operations*. We'll try to distinguish the two contexts from each other.
- *Multiple thread-safe connections*. Some applications might need to keep several connections to several databases — for example, cache, actual

storage, metrics, and so on. The access to each connection and to the set of connections should be thread-safe.

- *Raw, untyped, low-level interface (eDSL)*. It's fine to have raw strings for keys and values in the low-level interface. This is what all key-value storage must support.
- *Typed higher-level interface (eDSL) on top of an untyped one*. This one should allow us to represent a database model in a better way than just raw keys and values.

Enough words, let's code! In listing 7.1, you can see the simplest monadic key-value database language, which has only two methods for saving and loading a value.

Listing 7.1 Basic string-based key-value database interface

```
type KVDBKey   = ByteString
type KVDBValue = ByteString

data KVDBF next where
  Save :: KVDBKey -> KVDBValue
       -> (DBResult () -> next)
       -> KVDBF next
  Load :: KVDBKey
       -> (DBResult (Maybe KVDBValue) -> next)
       -> KVDBF next

type KVDBL db = Free KVDBF

save :: KVDBKey -> KVDBValue -> KVDBL db (DBResult ())
load :: KVDBKey -> KVDBL db (DBResult (Maybe dst))
```

Notice that both methods can fail for some reason, and we encode this reason as `DBResult`. Its description is presented in listing 7.2.

Listing 7.2 A type for database results

```
data DBErrorType
    = SystemError
    | InvalidType
    | DecodingFailed
    | UnknownDBError

data DBError = DBError DBErrorType Text

type DBResult a = Either DBError a
```

There's no information about a database connection here. In fact, the database connection can be considered an implementation detail, and you might want to completely hide it in the application runtime in some designs. In other situations, it's better to represent a connection explicitly in the domain and business logic. For example, you might want to query a database constantly and notify a user about changes, but there are several distinct connections existing in the logic. You would then start a separate process and pass a connection value to it — and let it work.

So many different requirements, so many decisions, and it's very unclear what decision is best. There are hundreds of "best practices" being described, and opinions here might declare opposite statements. This means that there are no best practices at all. This is why database connectivity management is a hard question. A bunch of hard questions, actually. Should a connection be kept alive all the time, or should it be open only for a particular query? Is it safe to connect to the database without disposing of connections? Has a database system some timeout for idle connections? Will it be doing a reconnection automatically if the connection is lost? How can a database system define that the connection has been lost?

Well, I'd suggest proposing more example solutions and justifications in addition to the solution described here. The KVDBL language should abstract only operations with data: reading, writing, searching, updating, deleting, and so on. It doesn't matter whether the underlying database libraries separate connecting and querying actions; we're free to model our own semantics that will be more convenient, or at least less burdensome. As an example, the

`rocksdb-haskell` library (binding for RocksDB) requires a database value that's essentially a connection handle:

```
-- Database handle
data DB = DB RocksDBPtr Options'

-- Read a value by key
get :: MonadIO m => DB -> ReadOptions
    -> ByteString -> m (Maybe ByteString)

-- Write a key/value pair
put :: MonadIO m => DB -> WriteOptions
    -> ByteString -> ByteString -> m ()
```

Those functions work in the bare `IO` (the `MonadIO` type class requires this), and it's pretty easy to incorporate them into a client code. But the `hedis` library exposes another design. It provides a monadic context in which you can declare your queries. The real work will be done after evaluating the `runRedis` function. The latter takes a monadic action with queries and a connection:

```
-- Read value
get :: (RedisCtx m f) => ByteString -> m (f (Maybe ByteString))

-- Write value
set :: (RedisCtx m f)
    => ByteString -> ByteString -> m (f Status)

-- Runner with a connection
runRedis :: Connection -> Redis a -> IO a

-- Sample scenario:
scenario = runRedis conn $ do
  set "hello" "hello"
  set "world" "world"
```

Let's revise the structure of languages from the previous chapter (see figure 7.2).

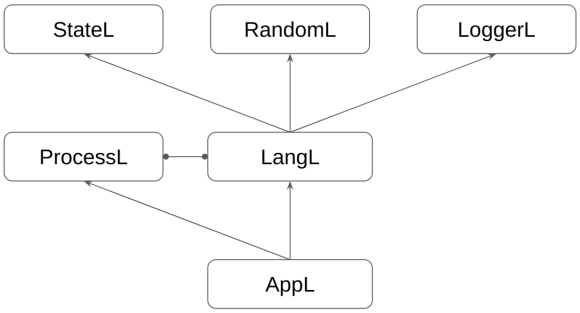


Figure 7.2 Structure of the languages in the framework

Now, where should we place the new language, `KVDBL`, and what should the connectivity management look like? For our case, there's no need to invent something complex. Why not replicate the same idea as the `hedis` library provides? Let it be a method for running a `KVDBL` scenario, and let it accept a connection. We should place it into `LangL` because working with databases is a significant part of business logic, where `LangL` is the main eDSL.

See in the following code: a method, an additional type for a database connection (we call it `DBHandle` here), and some type class for denoting a database that we'll discuss a bit later:

```
class DB db where
  getDBName :: DBName

data DBType = Redis | RocksDB
type DBName = String
data DBHandle db = DBHandle DBType DBName
```

Now the updated LangL language:

```
data LangF next where
  EvalKVDB
    :: DB db => DBHandle db
    -> KVDBL db a -> (a -> next) -> LangF next

type LangL = Free LangF
```

```
evalKVDB :: DB db => DBHandle db
         -> KVDBL db a -> LangL a
```

Here, `DBHandle` can be parametrized by any data type that's an instance of the `DB` type class. In other words, the instance of this type class is always related to a single key-value database storage (a file in the case of RocksDB and an instance in the case of Redis). For the astronomical application, we might want to specify the following database:

```
data AstroDB

instance DB AstroDB where
  getDBName = "astro.rdb"
```

Here, the database name is exactly the same as the file name. When we're working with this `AstroDB` type, the framework will take responsibility for dispatching our calls to this real RocksDB storage. See section 7.3.2 for detailed information about the usage of this `AstroDB` phantom type.

We now have a complete interface to run key-value database actions (non-transactionally), but we don't have any method by which we could call and obtain a connection to a database storage. We can imagine two possible solutions:

1. Connect to a database storage outside of the business logic, namely, in the runtime initialization stage. Pass this connection into the business logic scenarios. The connection will be opened during the whole application evaluation.
2. Add methods for connection/disconnection into one of the languages. With the language structure we have currently (figure 7.2), there are two options:
    a. Add connect/disconnect methods into `LangL`. This will allow you to administer the connections from the main scenarios. Makes sense if we want to dynamically connect to databases on some event or if we don't want the connections to hang open for a long time.
    b. Add such methods into the `AppL` language. When the `AppL` scenario starts, it connects to a database storage, obtains a long-living connection, and passes it into the logic. This option

gives more flexibility than option #1 but less flexibility than option #2-a.

We'll proceed with option #2-a. Take a look at the AppL language and a type for a key-value database configuration:

```
data KVDBConfig db
  = RedisConfig
  | RocksConfig
      { _path            :: FilePath
      , _createIfMissing :: Bool
      , _errorIfExists   :: Bool
      }

data AppF next where
  InitKVDB
    :: DB db
    => KVDBConfig db -> DBName
    -> (DBResult (DBHandle db) -> next)
    -> AppF next
  DeinitKVDB
    :: DBHandle
    -> (DBResult () -> next)
    -> AppF next

type AppL = Free AppF

initKVDB :: DB db => KVDBConfig db
         -> AppL (DBResult (DBHandle db))
deinitKVDB :: DBHandle db -> AppL (DBResult ())
```

Let's update the diagram by adding a new language to it (see figure 7.3).

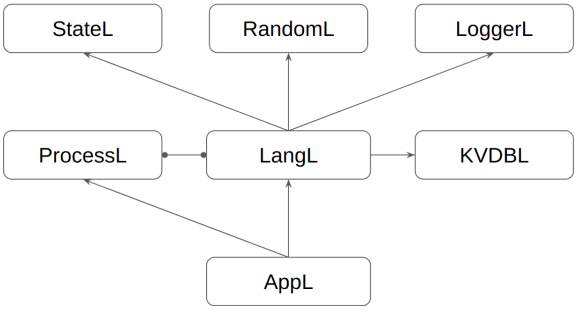


Figure 7.3 New language for the framework

We'll study the implementation in the next paragraph. Before that, have you noticed something strange with the `DBHandle` type? Something that you didn't expect to see there? Or vice versa, you don't see a thing that should be there? This type does not carry any real database connection. Having knowledge from chapters 5 and 6, you might have guessed that this type is just another "avatar" — an abstraction, a bridge, a key to accessing the real connection. We'll see more benefits of this pattern, especially for testing (see chapter 9).

7.2.2 Abstracted logic vs. bare IO logic

Abstracting native types has a particular goal: to separate implementation from interface. Abstractions — in the form of types and languages — are crucial for controllable, manageable, and testable systems. Returning to the database system, we have a single interface (`KVDBL`) that can be interpreted against any real key-value database. As usual, the types and eDSLs we've introduced shouldn't sop impurity and details from the implementation layer. Now let's transform it into real calls.

The following interpreter incorporates the `rocksdb-haskell` library into the framework:

```
import qualified Database.RocksDB as Rocks
import qualified Hydra.Core.KVDB.Impl.RocksDB as RocksDBImpl

interpretAsRocksDBF :: Rocks.DB -> KVDBF a -> IO a
interpretAsRocksDBF db (Load key next) =
  next <$> RocksDBImpl.get db key
```

```
interpretAsRocksDBF db (Save key val next) =
  next <$> RocksDBImpl.put db key val
```

Simple. Luckily, we don't need to somehow convert keys and values: both our interface and `rocksdb-haskell` use the same `ByteString` type. Unluckily, the `get` and `put` native methods don't return a failure type but rather throw an exception. We handle it by catching and wrapping it into our `DBResult` type presented earlier. Other possible solutions might propose wrapping the underlying unsafe resource (like real database calls) into the `ExceptT` monad, or maybe organizing the resource with a special `resourcet` library (by Michael Snoyman). As for the current code, there's an administrative contract that exceptions are prohibited and should be caught as soon as possible. See the sample implementation of the `get` function, which will be used by the interpreter of the `KVDBL` language:

```
get :: Rocks.DB -> KVDBKey -> IO (DBResult (Maybe KVDBValue))
get db key = do

  -- Real call and catching an exception
  eMbVal <- try $ Rocks.get db Rocks.defaultReadOptions key

  pure $ case eMbVal of                -- Result conversion
    Right mbVal -> Right mbVal
    Left (err :: SomeException) ->
      Left $ DBError SystemError $ show err
```

Implementation for `hedis` follows the same scheme but looks more complicated due to the specificity of the `Redis` monad. For the sake of comparison, take a look at the function:

```
get :: Redis.Connection -> KVDBKey
    -> IO (DBResult (Maybe KVDBValue))
get conn key = Redis.runRedis conn $ do
  fStatus <- Redis.get key
  pure $ case fStatus of
    Right mbVal -> Right mbVal
    Left (Redis.Error err) ->
      Left $ DBError SystemError $ decodeUtf8 err
    res -> Left
        $ DBError UnknownDBError
        $ "Redis unknown response: " <> show res
```

Can we now connect different databases without changing the interface? Well, yes — at least, we can do it for basic save/load actions. Have we unified the error handling? Yes, just convert exceptions or native return types into the language-level `DBResult` type. Easy. Abstractions matter, even if there's no requirement to have several implementations of the subsystem.

Let's do a bit more investigation of the design approaches and compare abstracted and non-abstracted versions of the key-value database subsystem. This is a sample scenario. Its task is to load two string values and return them concatenated:

```
data ValuesDB

instance DB ValuesDB where
  getDBName = "raw_values.rdb"

loadStrings
  :: DBHandle ValuesDB -> KVDBKey
  -> KVDBKey -> LangL (DBResult (Maybe KVDBValue))
loadStrings db keyForA keyForB = evalKVDB db $ do
  eMbA :: DBResult (Maybe KVDBValue) <- load keyForA
  eMbB :: DBResult (Maybe KVDBValue) <- load keyForB
  pure $ do       -- Either used as a monad here!
    mbA <- eMbA
    mbB <- eMbB
    pure $ do     -- Maybe used as a monad here!
      a <- mbA
      b <- mbB
      pure $ a <> b
```

A note on Either and Maybe

In the previous code listing, `Either` and `Maybe` were used as monads.

```
pure $ do       -- Either used as a monad here!
  mbA <- eMbA
  mbB <- eMbB
  pure $ do     -- Maybe used as a monad here!
    a <- mbA
    b <- mbB
    pure $ a <> b
```

This helps to unwrap such deep data structures as `Either Error (Maybe a)`, but there's an even shorter way. With help from the `Functor` and `Applicative` type classes, it can be rewritten as follows:

```
pure $ (<>) <$> eMbA <*> eMbB
```

The straightforward approach with no abstractions involved may seem much simpler at first sight. For example, in the following listing, the bare `IO` is used along with raw, non-abstract methods from the `RocksDB` library:

```
loadStringsRocksDB
    :: Rocks.DB -> KVDBKey
    -> KVDBKey -> IO (DBResult (Maybe KVDBValue))
loadStringsRocksDB db keyForA keyForB = do
  eA <- load' db keyForA    -- see below
  eB <- load' db keyForB
  pure $ (<>) <$> eA <*> eB

-- Wrapper function
load' :: Rocks.DB -> KVDBKey
      -> IO (Either String (Maybe KVDBValue))
load' db key = do

  eMbVal <- liftIO
     $ try
     $ Rocks.get db Rocks.defaultReadOptions key
```

```
pure $ case eMbVal of
  Right mbVal -> Right mbVal
  Left (err :: SomeException) -> Left $ show err
```

This similarity between abstracted and bare scenarios is actually deceptive. Let's compare them side-by-side (table 7.2).

Table 7.2 Comparison of bare IO scenario and abstracted scenario

Abstracted scenario	**Bare IO scenario**
Does not depend on a particular database implementation	Highly coupled with a particular library (`rocksdb-haskell`)
Abstracts any kind of errors, provides the unified interface	Mechanism of error handling is specific to the selected library
Makes it impossible to run any unwanted effect — more guarantees	The `IO` monad allows running any effect at any place — fewer guarantees
As simple as possible, no additional constructions and syntax	Somewhat burdened by implementation details (`liftIO`, `try`, `Rocks.DB`, and so on)
All the code has uniform syntax and semantics	Code looks like a crazy quilt
Easier to test	Harder to test

And now, two minutes of philosophy.

The very big temptation of keeping all possible levelers handy (in bare `IO` code) comes into conflict with the idea of delegating control to where it can be better served. This temptation ingrains a false belief that we can finely rule the code. And when we're caught, we get blinded and can no longer see a risk of turning the code into a conglomeration of unrelated objects. We forgive ourselves for mixing different layers, or we don't even recognize it as a sin, and we get high

accidental complexity as a result. On the other hand, in Software Design, there are no inherently bad solutions; there are solutions that satisfy or do not satisfy the requirements. The world of software development is too diverse to have a commonly accepted set of practices. Still, the fact of considering such practices and attempts to apply them leads to a better understanding of our requirements.

When we design a subsystem, our responsibility, our duty as code architects, is to define a methodology that everyone should follow. Without this policy, the project will end up as Frankenstein's monster, and no amount of galvanization would be sufficient to revive it after a sudden death.

7.2.3 Designing a SQL database model

When a developer starts thinking about the need for SQL databases in his project, he immediately steps into difficulties that imply activities that stray far from the usual development process. These difficulties start from the realization of the importance of having clear, well-elaborated requirements. The requirements will affect all the decisions and database behavior, and a wrong or incomplete understanding of needs can easily scupper the project in the near future. After some requirements are obtained, the developer has to make architectural decisions on what database storage should be used, how it should be accessed, what libraries are available, and which database management approach to follow. Irrespective of the way the developer will be designing a database model, he'll meet challenges in defining a database schema, tables, relations, normalization, and indexes. He'll be deciding how much logic should be placed into stored procedures and views and how much logic should be implemented as regular code. He'll also need to write a bunch of necessary SQL queries, and provide some views and server-side functions to support the logic. This is all considerable activity that requires a set of specific skills, such as knowing relational algebra, SQL, normal forms, indexing algorithms, and decomposition patterns. For better performance, one would also need to use the specificity of a particular database storage and its unique features and capabilities. It's not a surprise that some companies dedicate an entire development position, Database Engineer, to designing, managing, and implementing such solutions. This all feels very enterprisy; although industry has become way too complicated, with its hundreds of database solutions and practices, and has attempted to step back from relational databases, the latter are still a major discipline that's impossible to get rid of in real practice.

As software architects, we're interested in knowing these nuances from the perspective that they're directly related to complexity and risk management. Choosing the methods of database implementation influences the complexity of relational models a lot. This complexity is unavoidable in the sense that you have to implement tables and relations somehow, but it can be between the three points: database storage, intermediate layer, and database model. Figure 7.4 shows these three points.

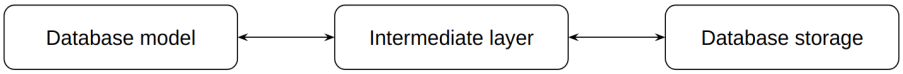


Figure 7.4 Database subsystem layers

In OO languages is the idea of ORM, which hides some complexity behind specific OO models. The ORM layer can be seen as a mediator between the program and the database. ORM systems seem to be overkill for small programs, but it's hard to build big applications without them. Supporting code made using bare, string-based queries without any database schema description is really difficult. So, mainstream developers have to balance the accidental complexity that comes with ORM systems.

And what about functional programming? In functional programming, we have the same tasks and problems but different solutions. Let's build a correspondence between the two worlds: object-oriented and functional (table 7.3).

Table 7.3 Correspondence between problems and solutions

Problem	**Object-oriented world**	**Functional world**
Mapping solution	Object Relational Mapping, Reflection	Type-level mapping, Generics, Template Haskell
Schema declaration	Internal eDSLs with classes or external eDSLs with markup language (XML)	Internal eDSLs with ADTs, Parametrized ADTs, Type Families, Type Classes

Database mapping	Repository pattern, Active Object pattern	Internal eDSLs with ADTs, Parametrized ADTs, Type Families, Type Classes
Entity mapping	Classes, Attributes (in C#, Java)	ADTs, Higher-Kinded Data Types, Advanced Types
Relations between tables	Nested collections, Inheritance	Nested ADTs, Type Families

We now have enough background to start designing the SQL database subsystem. Except we probably don't want to roll one more type-level abstraction library. It's a hard, expensive, and long activity that requires strong experience in type-level design and a good understanding of relational algebra and SQL-database-related stuff. So here, we have to choose one of the existing solutions and use it.

In Haskell, there's a bunch of libraries that abstract access to SQL databases with more or less type magic.

LINK The Selda library
https://selda.link/

LINK The Esqueleto library
http://hackage.haskell.org/package/esqueleto

LINK The Beam library
https://tathougies.github.io/beam/

There are less complicated libraries and libraries that provide some interface to particular database storages, like `postgres-simple`. But it seems the complexity of defining SQL queries and database models is unavoidable in Haskell. We want to have a type-safe database model, and we can't express a mapping mechanism in a simple manner. Haskell's type system is very powerful, yet very mind-blowing and wordy. There's a hope that inventing dependent

types for Haskell will solve a lot of problems and remove a fair amount of hardness as well.

Until then, we have to deal with what we have. And to show the problem in its essence, we'll take the `beam` library. Fortunately, it has some documentation, tutorials, and articles, but it's still not well-explained in many aspects. The bolts and pieces you can see in the source code are extremely unobvious. So, we'll limit ourselves to simple cases only.

Let's see how the domain model will look in this case. This model is not a part of the framework, but rather a part of the actual logic. `Beam` requires defining an ADT that will correspond to a particular table:

```
import Database.Beam (Columnar, Beamable)

data DBMeteorT f = DBMeteor
  { _id          :: Columnar f Int
  , _size        :: Columnar f Int
  , _mass        :: Columnar f Int
  , _azimuth     :: Columnar f Int
  , _altitude    :: Columnar f Int
  , _timestamp   :: Columnar f UTCTime
  }
  deriving (Generic, Beamable)
```

This is again an HKD type, and it will be used by `beam` to make queries and convert data from the real table into the code data. You have to think about the types you can use as columns here. Not all of them are supported by `beam`, and not all of them are supported by a specific database backend. Also, we won't talk about database schema layout in the `beam` representation. You can get familiar with the tutorial on the official `beam` site.

Now, we need to specify details about the table: its name and a primary key. For the primary key, a type class `Table` with an associated type should be instantiated:

```
instance Table DBMeteorT where
  data PrimaryKey DBMeteorT f = DBMeteorId (Columnar f Int)
    deriving (Generic, Beamable)
  primaryKey = DBMeteorId . _id
```

Here, `PrimaryKey t f` is the associated ADT. There can be other primary key data types, but this particular one works for `DBMeteorT`. It can't be confused with other tables. We should give a name to its constructor here: `DBMeteorId`. The `DBMeteorId` constructor will keep a primary key value. Also, the `primaryKey` function will extract this primary key from the ADT `DBMeteorT` and wrap it into the constructor (of type `PrimaryKey DBMeteorT f`). As you can see, there are some extra words in this definition that grab the attention and make the code a bit burdensome. We can do nothing about it. These are the details that leak from the `beam's` machinery. We can guess that deriving `Beamable` hides even more details from our eyes, or at least something.

The next necessary data type defines the database schema along with table names. We currently have only a single table `meteors`:

```
data CatalogueDB f = CatalogueDB
  { _meteors :: f (TableEntity DBMeteorT)
  }
  deriving (Generic, Database be)

-- Settings for the database.
catalogueDB :: DatabaseSettings be CatalogueDB
catalogueDB = defaultDbSettings
```

Again, some magic is happening here. The `CatalogueDB` type is also parameterized (with no visible reason), and it derives a special something called `Database`. The `be` parameter can declare that this schema is intended for only a specific SQL database storage (`beam` calls it "database backend"), for example, SQLite:

```
data CatalogueDB f = CatalogueDB
  { _meteors :: f (TableEntity DBMeteorT)
  }
  deriving (Generic, Database Sqlite)
```

But hey, we'll be considering our schema as a database-agnostic one, so let it be `be`.

The preparation of the database model finishes by defining two types for convenience:

```
type DBMeteor   = DBMeteorT Identity
type DBMeteorId = PrimaryKey DBMeteorT Identity
```

And what are these types for? The `DBMeteor` type will be appearing in the SQL queries. You can't call a query for a wrong table. Here, for example, is a query that selects all meteors having a predefined mass:

```
import qualified Database.Beam.Query as B

selectMeteorsWithMass size
  = B.select                                                  #A
  $ B.filter_ (\meteor -> _size meteor ==. B.val_ size)       #B
  $ B.all_ (_meteors catalogueDB)                             #C
```

```
#A SELECT query
#B WHERE clause condition
#C A kind of FROM clause
```

The type declaration of this function is omitted because it's too complex and hard to understand. The `filter_` function accepts a lambda that should specify what rows we're interested in. The lambda accepts a value of the `DBMeteor` type, and you can access its fields in order to compose a boolean-like predicate. It's not of the `Bool` type directly, but is a kind of `QExpr Bool`. The `beam` library provides several boolean-like comparison operators: `(==.)`, `(&&.)`, `(||.)`, `(>=.)`, `(<=.)`, `not_`, `(>.)`, `(<.)`, and others. It shouldn't be a problem to use them.

The `selectMeteorsWithMass` query doesn't query any particular database. The query itself is database-agnostic. We can assume that the query will be correctly executed on any database storage that supports a very basic SQL standard. So how would we execute this query on SQLite, for example? The `beam-sqlite` package provides a runner for SQLite databases that's compatible with `beam`. You can find the `runBeamSqlite` function there:

```
-- In the module Database.Beam.Sqlite:

runBeamSqlite :: Connection -> SqliteM a -> IO a
```

Don't pay that much attention to `SqliteM` for now. In short, it's a custom monad in which all the real calls to SQLite will happen. This type should be hidden in the internals of the framework because it's certainly related to implementation-specific details. So, the developers working with the framework can hopefully be liberated from additional knowledge.

Unfortunately, having the `runBeamSqlite` function isn't enough to launch the query. There are even more details coming from the `beam` machinery. The next function that we need allows us to specify a way we want to extract the results. It's called `runSelectReturningList`:

```
-- In the module Database.Beam.Query:

runSelectReturningList
  :: ( MonadBeam be m
     , BeamSqlBackend be
     , FromBackendRow be a)
  => SqlSelect be a -> m [a]
```

And again, it contains a fair amount of internal bolts and pieces. Look at those type constraints and some parameters like `SqlSelect be a`. It's not immediately obvious that we're dealing with special type families here, and if you decide to learn `beam` design, please be prepared for a mind-bending journey into type-level magic. We would like to avoid this complexity in our business logic. The `runSelectReturningList` should also be hidden and abstracted, especially its type. Consider the following final call stack, composed from all of those functions to just select some meteors:

```
import Database.Beam.Query as B (runSelectReturningList)
import Database.Beam.Sqlite as SQLite (runBeamSqlite)
import Database.SQLite.Simple as SQLite (Connection)

runGetMeteorsWithMass
  :: SQLite.Connection -> Int -> IO [DBMeteor]
runGetMeteorsWithMass conn size
  = SQLite.runBeamSqlite conn
  $ B.runSelectReturningList
  $ selectMeteorsWithMass size
```

If we suppose that we've obtained this native connection somehow, we can pass it into this function. It will interact with the real database. The `selectMeteorsWithMass` function represents a query, and it will be transformed into the SQL string by `beam`.

Interestingly, declaration of the queries and execution steps in `beam` is done with Church Encoded Free Monads. The library provides a broad interface for queries. Each part of this chain can be configured, and it's possible to compose complex queries with JOINs, subqueries, aggregation functions, and other standard SQL transformations. A whole world of different combinators is hidden there — the very impressive world of `beam`. But to be honest, our road runs away from this blissful island. We want to embed this eDSL into our language hierarchy without increasing accidental complexity too much. Before we assemble a mechanism for this, let's take a look ahead and foresee a client code working with such a SQL database subsystem. In the next listing (listing 7.3), an `AppL` scenario is presented in which there's a connection procedure, and a query embedded into the `AppL` and `FlowL` languages.

Listing 7.3 Database-related business logic

```
meteorsApp :: AppL ()
meteorsApp = do
  eConn <- initSQLiteDB "./meteors.db"             #A
  case eConn of
    Left err -> logError "Failed to init connection."
    Right conn -> do
      meteors <- getMeteorsWithMass conn 100
      logInfo $ "Meteors found: " <> show meteors

getMeteorsWithMass
  :: SQLite.Connection
  -> Int
  -> AppL [DBMeteor]
getMeteorsWithMass sqliteConn size = do
  eMeteors <- scenario
    $ evalSQLiteDB sqliteConn                      #B
    $ runBeamSelect                                #C
    $ selectMeteorsWithMass size                   #D
  case eMeteors of
    Left err -> do
```

```
      logError $ "Error loading meteors: " <> show err
      pure []
    Right ms -> pure ms

#A function provided by the framework
#B evalSQLiteDB is provided by the framework
#C runBeamSelect is provided by the framework
#D selectMeteorsWithMass defined earlier in the chapter
```

Now we're going to add some new functionality to the framework that makes the shown code possible.

7.2.4 Designing a SQL database subsystem

The language from listing 7.3 supports only the SQLite database backend. The corresponding scheme of embedding the languages looks like that in figure 7.5 (bold text is from the `beam` library).

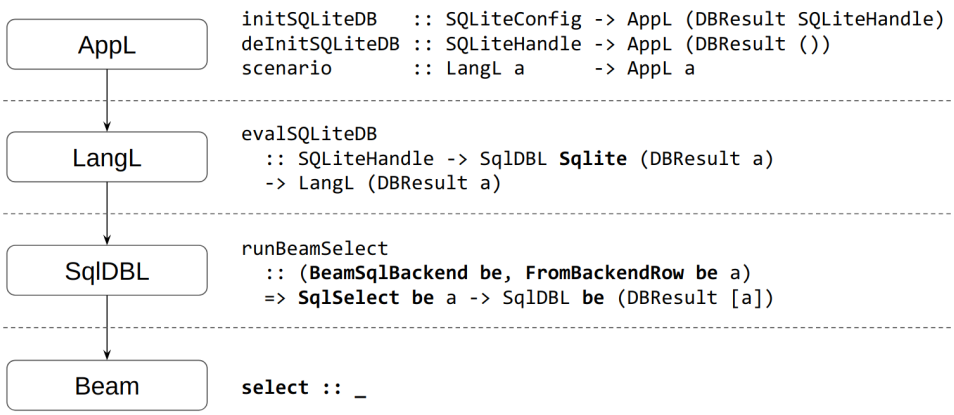


Figure 7.5 Embedding of SQL language scheme

The `InitSQLiteDB` method is quite simple. It accepts a path to the database file and possibly returns a native connection. In case of error, the `DBResult` type will keep all the needed info about it:

```
type DBPath = String

data AppF next where
  InitSQLiteDB
    :: DBPath
    -> (DBResult SQLite.Connection -> next)
    -> AppF next

initSQLiteDB :: DBPath -> AppL (DBResult SQLite)
initSQLiteDB path = liftF $ InitSQLiteDB path id
```

In the interpreter, we just call a native function from the `sqlite-simple` package. Don't forget about catching the exceptions and converting them into the wrapper `DBResult` type! Something like this:

```
import qualified Database.SQLite.Simple as SQLite

interpretAppF appRt (InitSQLiteDB path next) =
  next <$> initSQLiteDB path

initSQLiteDB :: DBPath -> IO (DBResult SQLite.Connection)
initSQLiteDB path = do
  eConn <- try $ SQLite.open dbName
  pure $ case eConn of
    Left err   -> Left $ DBError SystemError $ show err
    Right conn -> Right conn
```

Just a side note: catching the exceptions in the interpreter and not allowing them to leak outside a particular method seems like a good pattern because it lets us reason about the error domains. If all the exception sources are controlled on the implementation layer, the interfaces (our Free monadic eDSLs) and scenarios will behave predictably. If we want to process the error, we can pattern match over the `DBResult` explicitly. In other words, different layers should handle errors and exceptions with different tools. By doing this, we divide our application into so-called error domains, and thus get an additional separation of concerns. Believe me, you'll be well-rewarded for this order in your application very soon in the form of low accidental complexity, increased robustness, and more obvious code.

NOTE We'll discuss error domains in the next chapter more closely.

Integration of the `beam` machinery will consist of two parts: the `EvalSQLiteDB` method and the `SqlDBL` Free monad language. The latter will keep the calls to `beam` so that we can transfer the evaluation into the implementation layer (into the interpreters). This is important because we shouldn't allow `beam` methods to be directly called from the scenarios. They're essentially impure and also have a clumsy interface with all that type-level magic involved. We can provide a slightly more convenient interface.

Okay, check out the `EvalSQLite` method:

```
import Database.Beam.Sqlite (Sqlite)

data LangF next where
  EvalSQLiteDB
   :: SQLite.Connection
   -> SqlDBL Sqlite (DBResult a)
   -> (DBResult a -> next)
   -> LangF next

evalSQLiteDB
  :: SQLite.Connection
  -> SqlDBL Sqlite (DBResult a)
  -> LangL (DBResult a)
evalSQLiteDB conn script = liftF $ EvalSQLiteDB conn script id
```

It accepts a raw SQLite connection and a scenario in the `SqlDBL` language. Notice that the language is parametrized by a phantom type `Sqlite` (from the corresponding `beam-sqlite` library). Unfortunately, this design won't be as generic as we would like it to be. The `Sqlite` phantom type leaks into our business logic, which isn't good. We'll improve this approach in the next paragraphs and will also make sure that we can run `SqlDBL` not only with SQLite but with any database backend preliminarily chosen. For now, `Sqlite` is an implementation detail that the business logic depends on, as is the native SQLite `Connection` type.

Now for the `SqlDBL` Free monad language. It's tricky because it tracks info about a particular database backend in types. In our case, it's SQLite, but it can be replaced by Postgres, MySQL, or anything else. Let's examine the code:

```
import Database.Beam (FromBackendRow, SqlSelect)
import Database.Beam.Backend.SQL (BeamSqlBackend)

data SqlDBF be next where
  RunBeamSelect :: (BeamSqlBackend be, FromBackendRow be a)
    => SqlSelect be a
    -> (DBResult [a] -> next) -> SqlDBF be next
```

`SqlDBF` is a wrapper for the underlying `beam` methods. This type will hold a `SqlSelect` action — a type from `beam` denoting the `SQL SELECT` query. We can add more methods for `SqlUpdate`, `SqlInsert`, and `SqlDelete` actions later on. Notice that we also specify some type constraints `BeamSqlBackend` and `FromBackendRow` here. It's quite a long story; I won't describe the `beam` library here. Just take for granted that we deal with these instances in such a way as is shown to make it compile and work. If we decided to choose another library for the low-level SQL database engine, we could face other difficulties specific to that library.

The monadic type and smart constructor are straightforward:

```
type SqlDBL be = Free (SqlDBF be)

runBeamSelect
  :: forall be a
   . BeamSqlBackend be
  => FromBackendRow be a
  => SqlSelect be a
  -> SqlDBL be (DBResult [a])
runBeamSelect selectQ = liftF $ RunBeamSelect selectQ id
```

We don't specify a particular SQLite backend here, but just accept the `be` type parameter. But the `EvalSQLiteDB` method from the previous listings fixates this phantom type as `Sqlite`. Thus we cannot interpret the `SqlDBL` action against a wrong backend.

And what about the interpreter? There are different interesting points here related to the `beam` internals. Compared to what you can find in Hydra, the following implementation is simplified for better demonstrability. Look how we run a function `runBeamSqliteDebug`, which we can easily recognize as an implementation-specific function. It directly works with an impure subsystem, it does some debug logging, and it works in the `IO` monad. The

client code shouldn't bother about this function, so we place it into the interpreter.

Here's the interpreter that works with a native connection and impure subsystems directly:

```
import qualified Database.Beam as B

interpretSQLiteDBF
  :: (String -> IO ())
  -> B.Connection
  -> SqlDBF Sqlite a
  -> IO a
interpretSQLiteDBF logger conn (RunBeamSelect selectQ next) =
  next . Right <$> B.runBeamSqliteDebug logger conn
       $ runSelectReturningList
       $ selectQ

runSQLiteDBL
  :: (String -> IO ())
  -> SQLite.Connection
  -> SqlDBL Sqlite a
  -> IO a
runSQLiteDBL logger conn act =
  foldFree (interpretSQLiteDBF logger conn) act
```

Several notes here:

- The interpreter is closely tied to the SQLite database backend.
- The `runBeamSqliteDebug` function accepts a logger for printing debug messages, which could be very useful. We can (and should) pass our logger from the logging subsystem. It's easy to do, just try to figure it out yourself. Also, there's another function available in the library: `runBeamSqlite`. It doesn't print anything, in case you want the system to be silent.
- We explicitly require the query to return a list of things by using `runSelectReturningList`.
- We don't catch any exceptions here, which is strange. It's a bug, more likely, because there's no guarantee that the connection is still alive. However, we don't really know if the `runBeamSqliteDebug` function

throws exceptions. Hopefully, this information is written in the documentation, but if not … well, in Haskell we have to deal with incomplete documentation constantly, which is sad, of course.

- We've completely ignored a question about the transactional evaluation of queries. It's possible to support transactions; we just focused on the idea of embedding an external complex eDSL into our languages hierarchy.

So this is it. Happy querying SQLite databases in your business logic!

Or read the next sections. We have some limitations in our SQL database mechanism. Let's design a better support for this subsystem.

7.3 Advanced database design

In this section, we'll continue discussing the structure and implementation of database programs. Some questions are still uncovered:

- Support of many different backends for SQL database subsystems
- Transactions in SQL DB
- A higher-level model for key-value database and type machinery for better type safety of the mapping between a domain model and key-value database model
- Transactions in a key-value database
- Pools and better connections

The following sections will provide more approaches and mechanisms that we can build on top of Free monads, and you'll probably find more new ideas here. Still, we're operating with the same tools, with the same general idea of functional interfaces, eDSLs, and separation of concerns. We carefully judge what's an implementation detail and what's not, so we're gaining great power against the complexity of the code. This is our primary goal: do not bring more complexity than is really needed.

7.3.1 Advanced SQL database subsystem

The `beam` library supports Postgres, SQLite, and MySQL. Setting aside the quality of these libraries, we can rely on the fact that they have a single interface so that it's now possible to incorporate a generic eDSL to interact with any SQL

database backend we need. Our goal is to enable such business logic as is shown in listing 7.4. We should create a config value using the function `mkSQLiteConfig`, then call an unsafe helper function `connectOrFail`, which does what its name tells us. After that, we query the database to get meteors with a specified mass.

Listing 7.4 Usage of the advanced SQL subsystem

```
sqliteCfg :: DBConfig SqliteM
sqliteCfg = mkSQLiteConfig "test.db"

connectOrFail :: AppL (SqlConn SqliteM)
connectOrFail cfg = do
  eConn <- initSqlDB sqliteCfg
    Left e     -> error $ show e    -- Bad practice!
    Right conn -> pure conn

dbApp :: AppL ()
dbApp = do
  conn <- connectOrFail

  eDBMeteor :: Either DBError DBMeteor <- scenario
        $ evalSqlDB conn
        $ findRow
        $ selectMeteorsWithMass 100

  case eDBMeteor of
    Left err       -> logError $ show err
    Right dbMeteor -> logInfo  $ "Found: " <> show dbMeteor

  where

    selectMeteorsWithMass size
      = Beam.select
      $ Beam.filter_ (\m -> _size m ==. Beam.val_ size)
      $ Beam.all_ (_meteors catalogueDB)
```

In the previous simplified approach, we created the `initSQLiteDB` function, which is rigid and specific. No, it's not bad to be precise in your intentions! Not at all. It's just that we want to move database backend selection to another place.

Now we'll be specifying it by filling a config structure. Consider these new types:

```
type ConnTag = String

data DBConfig beM
  = SQLiteConf DBName
  | PostgresConf ConnTag PostgresConfig
  | MySQLConf ConnTag MySQLConfig

data SqlConn beM
  = SQLiteConn ConnTag SQLite.Connection
  | PostgresConn ConnTag Postgres.Connection
  | MySQLConn ConnTag MySQL.Connection
```

The `ConnTag` will help us distinguish different connections. `DBConfig` will hold a config for any of the three supported databases, and `SqlConn` will represent a connection in our business logic. It makes sense to turn `DBConfig` and `SqlConn` into abstract types by hiding their constructors. What about the type variable `beM`? We summoned it to be our phantom denoting a specific monad for any `beam` backend.

At the moment, we have several monads for Postgres, SQLite, and MySQL, and these monads are certainly very much a detail of implementation. Here:

```
import Database.Beam.Sqlite (SqliteM)
import Database.Beam.Postgres (Pg)
import Database.Beam.MySQL (MySQLM)

mkSQLiteConfig :: DBName -> DBConfig SqliteM

mkPostgresConfig :: ConnTag -> PostgresConfig -> DBConfig Pg

mkMySQLConfig :: ConnTag -> MySQLConfig -> DBConfig MySQLM
```

Hopefully, business logic developers won't be writing the code in any of these monads (try to remember all the details of those monads!). It's only needed for this advanced design, to keep and pass this specific monad into the internals and avoid a fight with the compiler. Certainly, other designs can be invented, and some of them are probably much better, but who knows.

Now, considering that you have a config type, and you need a connection type, guess how a single-sign-on `initSqlDB` function from the `AppL` language should look. Like this:

```
initSqlDB :: DBConfig beM -> AppL (DBResult (SqlConn beM))
```

Now the idea becomes clearer, right? The `beM` phantom type we store in our config and connection ADTs will carry the information of what backend we need. You might say that we already have this information as a value constructor (like `SQLiteConn`, `PostgresConn`, or `MySQLConn`) — why should we duplicate it in such a manner? Okay, this is fair. Suggestions are welcome! It might be possible to deduce a monad type for the backend by just pattern matching over the `SqlConn` values. Also, it feels like dependent types can do the job here, so maybe in the future, this code will be simplified. Further design research can reveal more interesting options, so try it yourself. So far so good, we're fine with the current approach.

The `AppL` language should be updated:

```
data AppF next where
  InitSqlDB
    :: DBConfig beM
    -> (DBResult (SqlConn beM) -> next)
    -> AppF next
```

During the interpretation process, you'll have to decide on the following:

- Should a duplicated connection be allowed, or should the attempt to call `InitSqlDB` twice fail?
- If it's allowed, then should a new connection be created or the old one have to be returned?
- If the old one is returned, then what about a multithreaded environment? How to avoid data races?

Ohhh … pain. Seventy years of programming discipline, 60 years of a functional programming paradigm, 30 years of massive industrial development, 10 years of Haskell hype, and we're still solving the same problems, again and again, in different forms, in different environments. We're extremely good at inventing existing techniques and repeating work already done. Nevermind, this is a book

about design approaches. It doesn't provide you all the information about everything we have in our field.

Let's just skip these questions and jump directly into the incorporation of `beam` into the framework. We should consider a new requirement this time: the query should be database-backend-agnostic, and a real database backend be defined by the `SqlConn beM` type. Going ahead, there are several difficulties caused by the design of `beam`, and we'll invent some new creative ways to solve the task.

Let's investigate a bare `IO` call stack with `beam` closely for SQLite and Postgres:

```
querySQLiteDB :: SQLite.Connection -> IO [DBMeteor]
querySQLiteDB sqliteConn
  = SQLite.runBeamSqlite sqliteConn                  #A
  $ B.runSelectReturningList                         #B
  $ B.select                                         #C
  $ B.all_ (_meteors catalogueDB)                    #D

queryPostgresDB :: Postgres.Connection -> IO [DBMeteor]
queryPostgresDB pgConn
  = Postgres.runBeamPostgres pgConn                  #A
  $ B.runSelectReturningList                         #B
  $ B.select                                         #C
  $ B.all_ (_meteors catalogueDB)                    #D

#A Real runner and real connection
#B Expected structure
#C Query
#D Query conditions
```

Notice how we construct a call: real runner -> real connection -> expected structure -> query. In this chain, we need to hide a real runner and select it only on the interpreter side. This effectively means that we need to transfer the knowledge about the backend from the language layer to the implementation layer. In other words, pass `SqlConn beM` there. Okay, so we draft the following method for `LangL`:

```
data LangF next where
  EvalSqlDB :: SqlConn beM
            -> ??? query here ???                #A
            -> (DBResult a -> next)
            -> LangF next
```

#A TODO: Introduce a Free monad query language here

Should we construct the call stack like this? Pseudocode:

```
queryDB :: SqlConn beM -> LangL [DBMeteor]
queryDB conn
  = evalSqlDB conn
  $ B.runSelectReturningList
  $ B.select
  $ B.all_ (_meteors catalogueDB)
```

The `runSelectReturningList` has the following type:

```
runSelectReturningList
  :: ( MonadBeam be m
     , BeamSqlBackend be
     , FromBackendRow be a
     )
  => SqlSelect be a
  -> m [a]
```

`Beam` exposes a lot of type classes along with its own monad `MonadBeam`! Embedding it into the method `EvalSqlDB` requires you to keep all those type-level bits. Pseudocode:

```
data LangF next where
  EvalSqlDB
    :: ( MonadBeam be m
       , BeamSqlBackend be
       , FromBackendRow be a
       )
    => SqlConn beM
    -> (SqlSelect be a -> m [a])
    -> (DBResult a -> next)
    -> LangF next
```

That's … d-i-f-f-i-c-u-l-t. And by the way, an observation: we just forgot about all other query types like `Insert`, `Update`, and `Delete`. Embedding the `beam` facilities this way is technically possible, but it's for people strong in spirit. You'll have to solve a lot of type-level mismatches if you go this way. I tried many different designs and came to the understanding that I don't have enough skills on the type level to make it work. So, I ended up with a design that hides the `runSelectReturningList` from the user, as well as real runners like `runBeamPostgres`. Listing 7.4 demonstrates the result I achieved; explaining this approach will be quite a challenge, but I'll try. Please keep in mind three goals:

- Hide the beam details as much as possible.
- Enable more or less simple incorporation of different beam queries.
- Keep the design open so that other SQL backends can be added later.

The new approach has two subparts: Free monad languages for a generic SQL and real runner selectors. In general, all the important wrappings are made on the language layer, not on the implementation layer. This differs from the old designs. Reminder: previously, the interpreters were the only modules aware of the real functions and libraries. Here, we put all the beam queries, types, and real runners into eDSLs directly, making the language modules depend on these details.

Moreover, we'll be pre-evaluating the `beam` functions in order to reduce the need for passing type classes. For example, if we have the following function from `beam`:

```
runInsert :: (BeamSqlBackend be, MonadBeam be m)
          => SqlInsert be table
          -> m ()
```

We'll hide it behind our own type class so that it won't be that problematic to add new queries:

```
import Database.Beam as B

class (B.BeamSqlBackend be, B.MonadBeam be beM)
      => BeamRuntime be beM
      | be -> beM, beM -> be where                        #A
  rtInsert :: B.SqlInsert be table -> beM ()
  -- TODO: rtDelete, rtUpdate...

#A Functional dependency: when we know be,
#A   we'll be knowing beM, and vice versa
```

Next, we'll put its pre-evaluation version into the following type:

```
data SqlDBAction beM a where
  SqlDBAction :: beM a -> SqlDBAction beM a
```

This helper function will dispatch the `insert` query from our `SqlDBAction` to `beam` through the `BeamRuntime` type class:

```
insert''
  :: BeamRuntime be beM
  => B.SqlInsert be table
  -> SqlDBAction beM ()
insert'' a = SqlDBAction (rtInsert a)
```

Notice that we partially evaluate the function `rtIsert`, and the `SqlDBAction` method gets the only thing: an action in the real backend monad, like `SqliteM`. Now, with the help of the `BeamRuntime` type class and the `SqlDBAction` type, it's possible to incorporate more `beam` queries. The SQL standard offers a lot of ways to construct queries, and `beam` tries to handle that; in particular, its functions can return different types of result values: `Maybe a`, `[a]`, `()`, and others. This is how it looks for SQLite and several queries:

```
class (B.BeamSqlBackend be, B.MonadBeam be beM)
  => BeamRuntime be beM
  | be -> beM, beM -> be where
  rtInsert :: B.SqlInsert be table -> beM ()
  rtUpdate :: B.SqlUpdate be table -> beM ()
  rtDelete :: B.SqlDelete be table -> beM ()
```

```
instance BeamRuntime B.Sqlite B.SqliteM where
  rtInsert = B.runInsert
  rtUpdate = B.runUpdate
  rtDelete = B.runDelete
```

The further reduction of the type `beam` classes will require one more type class. This one will help to dispatch the evaluation of the queries to one of several supported SQL backends. The `BeamRunner` type class:

```
class BeamRunner beM where
  getBeamDebugRunner
    :: SqlConn beM -> beM a -> ((String -> IO ()) -> IO a)
```

We need this type class because we want to add support for more SQL libraries, and because the beam library offers several distinct runners: some of them do a debugging print, some of them just evaluate a query.

Finally, we'll utilize the power of Free monads to introduce a language for the SQL database subsystem. The client code should call the methods of this language only, to avoid being too strictly coupled with `beam.` Interestingly, the monad of this language will denote a scope for transactions — the aspect we forgot about undeservedly:

```
data SqlDBMethodF beM next where
  SqlDBMethod
    :: (SqlConn beM -> (String -> IO ()) -> IO a)       #A
    -> (a -> next)
    -> SqlDBMethodF beM next

type SqlDBL beM = Free (SqlDBMethodF beM)
```

The field `#A` will hold a function that will encapsulate the details. See the following function that will enclose the details of `beam` when placed into the `#A` field:

```
getBeamRunner :: (BeamRunner beM, BeamRuntime be beM)
              => SqlConn beM
              -> SqlDBAction beM a
              -> ((String -> IO ()) -> IO a)
getBeamRunner' conn (SqlDBAction beM)
  = getBeamDebugRunner conn beM      -- calling to BeamRunner
```

Look, there's nothing about the `beam` in this function, and the `IO` actions are what we'll be evaluating on the interpreter level. The interpreter becomes very simple:

```
interpretSqlDBMethod
  :: SqlConn beM
  -> (String -> IO ())
  -> SqlDBMethodF beM a
  -> IO a
interpretSqlDBMethod conn logger (SqlDBMethod runner next) =
  next <$> runner conn logger

runSqlDBL
  :: SqlConn beM
  -> (String -> IO ())
  -> SqlDBL beM a
  -> IO a
runSqlDBL conn logger =
  foldF (interpretSqlDBMethod conn logger)
```

However, it's not easy to avoid the `beam` machinery for the queries themselves (types `SqlSelect`, `SqlInsert`, `SqlUpdate`, and `SqlDelete`), so we're trying to at least simplify this interface by introducing one more helper:

```
sqlDBMethod
  :: (BeamRunner beM, BeamRuntime be beM)
  => SqlDBAction beM a -> SqlDBL beM a
sqlDBMethod act = do
  let runner = \conn -> getBeamRunner' conn act           #A
  liftFC $ SqlDBMethod runner id
```

#A Storing only the IO action in the language

And, finally, convenient smart constructors of the `SqlDBL` language:

```
insertRows
  :: (BeamRunner beM, BeamRuntime be beM)
  => B.SqlInsert be table -> SqlDBL beM ()
insertRows q = sqlDBMethod $ SqlDBAction (rtInsert q)

updateRows :: (BeamRunner beM, BeamRuntime be beM)
  => B.SqlUpdate be table -> SqlDBL beM ()
```

```
updateRows q = sqlDBMethod $ SqlDBAction (rtUpdate q)

deleteRows :: (BeamRunner beM, BeamRuntime be beM)
  => B.SqlDelete be table -> SqlDBL beM ()
deleteRows q = sqlDBMethod $ SqlDBAction (rtDelete q)
```

Figure 7.6 represents a schema of this design (the bold is beam-related stuff).

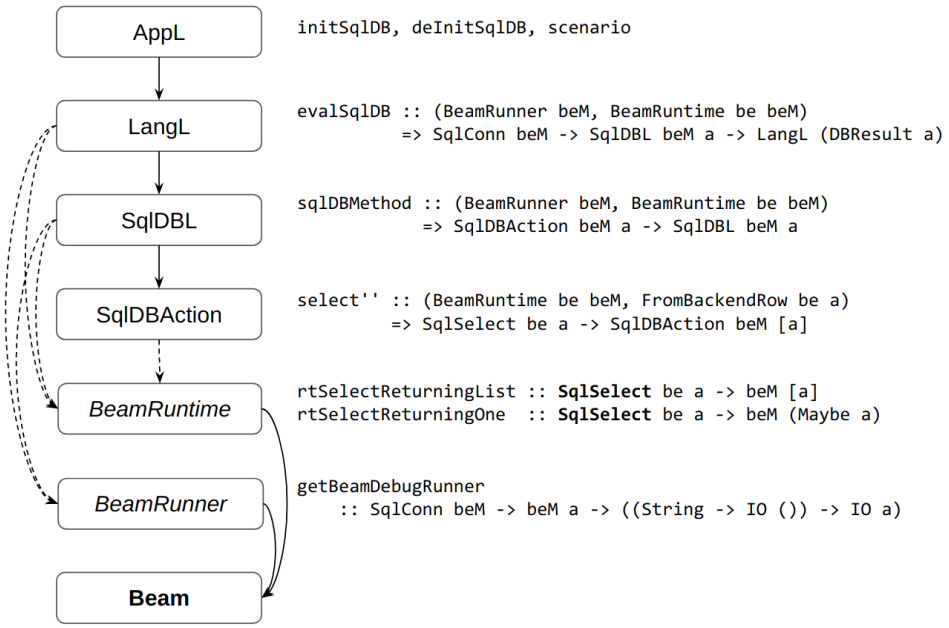


Figure 7.6 Advanced languages structure

Seems you just finished the most difficult part of the book. No, really. Unfortunately, Haskell by its nature offers too many temptations for over-complicating things. We can conclude that when some library imposes a predefined design or involves a lot of type-level magic, it becomes tricky to incorporate it into our code. And let me ask a question: is it possible to simplify the interface even more? Why should we care about such a strange embedding scheme? Well, probably yes. Don't judge too quickly, nevertheless. Sometimes we have to make dirty and imperfect decisions just to be able to move further. We'll pay for this complication later for sure, but software design can't be imagined without the need to balance goals, requirements, and simplicity. And

it's always better to have something at least reasonable and working rather than perfect but not yet ready.

We'll pause with the SQL database subsystem now and return to the key-value database subsystem. The section is still the same — "Advanced database design" — but I think we need to focus on constructing our own type-level mechanisms rather than trying to tame someone else's.

7.3.2 Typed key-value database model

Earlier, we forged an untyped key-value database subsystem in which all the data (keys, values) was represented by just `ByteStrings`. The key-value database subsystem doesn't support any strictly typed data layout, so there's no guarantee that the data from a key-value database will be treated correctly. Fine — we designed a working mechanism and can just start writing some real code interacting with Redis or RocksDB. Neither a perfect nor a smart system that forces us to convert data manually. Still, this raw key-value database interface with bare `ByteStrings` covers 90% of our needs, and is very fast and fairly simple. And now, when we have more time, we can create a typed key-value database layer on top of this untyped interface without even changing the underlying framework.

Before this journey starts, we have to highlight the current design basics. According to the `KVDBL` language from section 7.2.2, all we have is two methods for loading and storing values:

```
save :: KVDBKey -> KVDBValue -> KVDBL db (DBResult ())
load :: KVDBKey -> KVDBL db (DBResult dst)
```

Remember the `AstroDB` phantom data type? Now this type will become a door to the astro catalogue, with different objects being tracked. Objects? Well, items. Data. Just values, if we're talking about a domain type. Entities, when we're talking about the database model. We want to distinguish these key-value database entities from each other. Okay, the underlying framework methods will be untyped, but our custom key-value database model will be typed. Let's say we want to load some meteor from the database by its key, and we know it's stored as a stringified JSON object. This is an example of such rows in the database, in the form of `(DBKey, DBValue)`:

```
("0", "{\"size\": 100,
        \"mass\": 100,
        \"azmt\": 0,
        \"alt\": 0,
        \"time\":\"2019-12-04T00:30:00\"}")
("1", "{\"size\": 200,
        \"mass\": 200,
        \"azmt\": 2,
        \"alt\": 0,
        \"time\": \"2019-12-04T00:31:00\"}")
```

We could do the job without inventing an additional layer on top of the `ByteString`-like interface. Listing 7.5 demonstrates a solution that might be good enough in certain situations (probably not — it's unsafe). Its idea is to serialize and deserialize database entities with `aeson`.

Listing 7.5 Manual serialization of database entities

```
data KVDBMeteor = KVDBMeteor
  { size  :: Int
  , mass  :: Int
  , azmt  :: Int
  , alt   :: Int
  , time  :: DateTime
  }
  deriving (Show, Generic, ToJSON, FromJSON)

encodeMeteor :: KVDBMeteor -> ByteString
encodeMeteor = encode

decodeMeteorUnsafe :: ByteString -> KVDBMeteor
decodeMeteorUnsafe meteorStr =
  case decode meteorStr of
    Nothing -> error "Meteor: no parse"     -- Bad practice
    Just m -> m
```

To establish some ground for future improvements, we'll also write a client code with manual parsing in it:

```
saveMeteor astroDB meteorId meteor
  = scenario
  $ evalKVDB astroDB
  $ save (show meteorId) (encodeMeteor meteor)

loadMeteor astroDB meteorId
  = scenario
  $ evalKVDB astroDB $ do
    meteorStr <- loadOrFail $ show meteorId           #A
    pure $ decodeMeteorUnsafe meteorStr
```

#A Unsafe function from the KVDBL language

You might also want to convert this `KVDBMeteor` into the `Meteor` domain data type from section 7.2.1. The converter function is simple, especially with the `RecordWildCards` extension:

```
fromKVDBMeteor :: MeteorID -> KVDBMeteor -> Meteor
fromKVDBMeteor meteorId KVDBMeteor {..} = Meteor
  { _id        = meteorId
  , _size      = size
  , _mass      = mass
  , _coords    = Coords azmt alt
  , _timestamp = time
  }
```

And now let's ask ourselves: how can we hide all this conversion stuff from the business logic code? How can we hide the very knowledge about the internal serialization format? If we have dozens of such data structures, then writing the same conversion functions will be too tedious. But the main reason we want to abstract the conversions and data loading/saving is to have a generic way to work with any database structures, even if we don't know about them yet. Some readers might remember the Expression Problem here — the problem of how to extend the code without changing its mechanisms, how to make the code future-proof. This is a well-known problem from mainstream development, but the name "Expression Problem" isn't widespread there. You might have heard that OOP developers are talking about extensibility, extensible points, and customization. In OOP languages, it's pretty simple to add a new type and make the existing logic able to work with it. Class inheritance or duck typing solves the extensibility problem for most cases.

On the other hand, developers of strongly typed functional languages without subtype polymorphism (such as Haskell) experience difficulties implementing such mechanisms. We not only need to think about what types can occur in the future, but we also have to use advanced type-level stuff to express the idea that a particular code accepts any input with predefined properties. Haskell's type classes do the job, but their expressiveness is very limited. More type-level features were added to provide comprehensive tooling and to talk with the type checker: functional dependencies, extensions for type classes, type families, existential types, GADTs, and other things. We can do very impressive things, but it still seems that we're not satisfied. There's a common idea that we need even more powerful tools: Dependent Types. Okay, maybe it's so, but until this feature is implemented, we should learn how to solve our tasks with the features we have. Therefore, I encourage you to consult with other books that can provide such knowledge for you.

The task we'll be solving is about generalization of the saving/loading mechanisms. We don't want to put all this conversion functionality into scenarios, but we want guarantees that conversion works safely and reliably. Here, if we save a value using the new typed mechanism, we're guaranteed that loading and parsing will be successful. Let's establish the goal in the form of the code we'd like to write. The next function asks to load a typed entity from the key-value database:

```
loadMeteor
  :: DBHandle AstroDB
  -> MeteorID
  -> AppL (DBResult Meteor)
loadMeteor astroDB meteorID
  = scenario
  $ evalKVDB astroDB
  $ loadEntity
  $ mkMeteorKey meteorID
```

You can guess that the new mechanism is hidden behind the functions `loadEntity` and `mkMeteorKey`. Let's get familiar with them.

The `loadEntity` knows how to load raw `ByteString` data from a key-value database and convert it to the target type. Notice that it refers to

several type-level artifacts: `DBEntity`, `AsValueEntity`, and `KeyEntity`. We'll figure out what they are in further detail:

```
loadEntity
  :: forall entity dst db
   . DBEntity db entity
  => AsValueEntity entity dst
  => KeyEntity entity
  -> KVDBL db (DBResult dst)
```

A small hint of what `KeyEntity` is can be obtained from the following smart constructor:

```
mkMeteorKey :: MeteorID -> KeyEntity MeteorEntity
```

You can think about `KeyEntity` as something like an ADT `KeyEntity` parametrized by a tag type. Previously, we did such things with HKDs, but now it's about type classes and associated types. Such a small piece of code, and so many concepts emerged from it. For those of you who are familiar with type-level programming, this picture won't be a surprise. Sadly, you can't be sure of what concept you deal with just by looking at the function definition. For example, the `KeyEntity` entity can be a regular type with a type parameter, an associated type, a type family, or a HKD. They all look similar but behave differently. This is why the type-level code is far from obvious.

Let's try to get a sense of the `loadEntity` function. We'll start from defining a type class and two associated types. Every item in our key-value database should have its entity representation. If there's a meteor in the domain model, there will also be its `DBEntity` instance. The presence of an entity means that the type-level mechanism knows how to serialize a domain type and parse it back. This will be an additional layer between the domain model and the untyped `ByteString` interface of the key-value database:

```
class DBEntity db entity | entity -> db where
  data KeyEntity   entity :: *
  data ValueEntity entity :: *
```

With this type class, we model:

- The relation between a database and an entity, hence the functional dependency `entity -> db`. It says that this entity can only be

related to a single database. Once you have a type for an entity, you'll immediately know the database.

- Two associated types `KeyEntity` and `ValueEntity`. Details of these types will be known whenever you have a specific entity type.

The scheme in figure 7.7 helps us to understand the type classes and associated types that we'll be talking about.

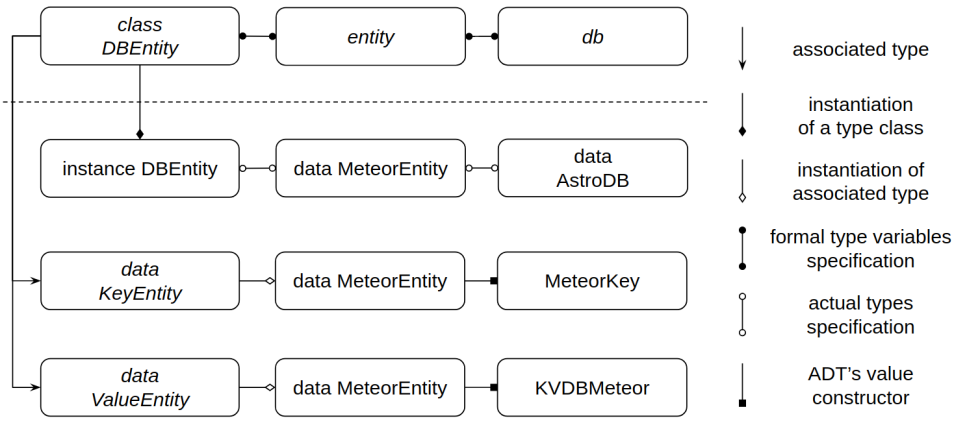


Figure 7.7 Key-value database type-level machinery

You can see a specific `DBEntity` type class that's an entry point into the type-level models for a key-value database. Instantiating this type class for two specific types as database and entity allows us to provide information on how to convert the entity to and from the target domain type, how to store its key, and how to serialize its value. For example, we can define a `MeteorEntity` and show how conversion for this entity works (see listing 7.6).

Listing 7.6 Meteor as DBEntity

```
data AstroDB
data MeteorEntity

instance DBEntity AstroDB MeteorEntity where

  data KeyEntity MeteorEntity = MeteorKey MeteorID
    deriving (Show, Eq, Ord)
```

```
  data ValueEntity MeteorEntity = KVDBMeteor
    { size  :: Int
    , mass  :: Int
    , azmt  :: Int
    , alt   :: Int
    , time  :: D.DateTime
    }
    deriving (Show, Eq, Ord, Generic, ToJSON, FromJSON)
```

With this instance, we can create a key value specific to this entity. Meteor objects should have their own IDs, and we put this integer `MeteorID` into the ADT. We can think of this data type as a separate ADT:

```
data KeyEntity MeteorEntity = MeteorKey MeteorID
```

The same is true for value type `ValueEntity MeteorEntity`. It can be seen as a separate ADT, and once you've imported this module, you can operate with this ADT in functions. The `mkMeteroKey` helper function presented earlier is defined simply; we just put the `MeteorID` into the `MeteorKey` constructor:

```
mkMeteorKey :: MeteorID -> KeyEntity MeteorEntity
mkMeteorKey = MeteorKey
```

So, we just defined a typed key-value database model. Imagine that this model is water (or magma, if you like Dwarf Fortress) — then several tubes and pumps are still lacking. More functions and type classes are needed to perform conversions from domain types to `KeyEntity MeteorEntity`, `ValueEntity MeteorEntity`, and back. The `DBEntity` type class also carries additional methods that should be specified for each database entity:

```
class DBEntity db entity | entity -> db where
  toDBKey     :: KeyEntity entity -> KVDBKey
  toDBValue   :: ValueEntity entity -> KVDBValue
  fromDBValue :: KVDBValue -> Maybe (ValueEntity entity)

instance DBEntity AstroDB MeteorEntity where
  toDBKey (MeteorKey idx) = show idx
  toDBValue valEntity = encode valEntity
  fromDBValue strEntity = decode strEntity
```

Now we know how the `KVDBMeteor` value constructor (of type `KeyEntity MeteorEntity`) becomes a `ByteString` that goes directly to the `KVDBL` methods. We can convert this value back to the `KVDBMeteor` form. But what about conversion to and from domain types like `Meteor`? And here is where the last piece of the puzzle goes. Two more helper type classes will show the relation between the key-value database model and domain model: `AsKeyEntity, AsValueEntity.` Not by chance, we want to track keys and values separately, as they can have different encoding schemes. The type classes are simple:

```
class AsKeyEntity entity src where
  toKeyEntity :: src -> KeyEntity entity

class AsValueEntity entity src where
  toValueEntity
    :: src -> ValueEntity entity
  fromValueEntity
    :: KeyEntity entity -> ValueEntity entity -> src
```

As you can see, they refer to the associated types from the `DBEntity` type class. This means we can operate with our specific data types once we know what the entity type is. Several instances are defined for `MeteorEntity`:

```
instance AsKeyEntity MeteorEntity MeteorID where
  toKeyEntity = MeteorKey

instance AsKeyEntity MeteorEntity Meteor where
  toKeyEntity = MeteorKey . meteorId

instance AsValueEntity MeteorEntity Meteor where
  toValueEntity (Meteor _ size mass (Coords azmt alt) time)
    = KVDBMeteor {..}
  fromValueEntity (MeteorKey idx) KVDBMeteor {..}
    = Meteor ...   -- some code here
```

Finally, we can talk about the typed `loadEntity` function. Remember, this function demands the type-level machinery for a particular database and entity. It requests a `ByteString` value from the key-value database subsystem, and, with the help of all those conversion functions, it converts the `ByteString`

value into the typed key-value database model first, and from the key-value database model to the domain model second:

```
loadEntity
  :: forall entity dst db
   . DBEntity db entity
  => AsValueEntity entity dst
  => KeyEntity entity
  -> KVDBL db (DBResult dst)
loadEntity key = do

  eRawVal <- load $ toDBKey key                        #A

  pure $ case eRawVal of
    Left err  -> Left err
    Right val -> maybe decodingFailedErr
        (Right . fromValueEntity key)                  #B
        (fromDBValue val)
```

#A DBEntity instance is used here
#B AsValueEntity instance is used here

So, the initial goal for this type-level machinery was the ability to specify a key-value database model for entities, but we got several interesting consequences:

- We can add more entities without affecting the mechanism itself: `MeteorEntity`, `StarEntity`, `AsteroidEntity`, and so on.
- A tag type for key-value databases like `AstroDB` now combines all those entities into a single key-value database model.
- It's possible to put the same entities into another key-value database by defining a new key-value database tag type, for example, `CosmosDB`, and instantiating the `DBEntity` type class for it. Thus, we can share entities between models if we need to.
- The loading and saving functions now use the conversions without knowing what entities we want to process. The `loadEntity` function, for example, just states several requirements and uses the typed interface to access specific associated data types.
- We've proven that a simple, raw `ByteString-typed` interface can

be improved and extended even without changing the framework itself. You can create more type-level machinery of different designs on top of it.

Still, the type-level machinery is more complicated and scary. Exposing all those multicomponent types like `ValueKey` or type classes like `DBEntity` to the business logic would be a bad idea because it raises the bar on what your colleagues should know and learn. The client code shouldn't be aware of what's happening there. Writing scenarios should remain simple and boilerplate-free. So, limiting the existence of these gears only on the hidden layer can be acceptable for your situation.

7.3.3 Pools and transactions

With monads, functional programming gets not only a way of sequencing effectful functions, but also a new tool for design. We've seen several design tricks with monads, and the next one we'll briefly cover here we also talked about in previous chapters — namely, a transactional context within some monad. Previously, it was the `STM` and `StateL` monads, and now we should complement our SQL database subsystem by the same mechanism.

Initial conditions for this solution include

- `SqlDBL` monadic language
- Existing machinery for incorporating the `beam` library (type classes `BeamRunner` and `BeamRuntime`, data type `SqlDbAction`)
- Runner and interpreter methods (`runSqlDBL`, `interpretSqlDBMethod`)
- SQL backend config data type (`DBConfig beM`)
- Connection data type (`SqlConn beM`)

Going ahead, our current implementation is suitable for incorporating transactions, except that some changes are required.

The `SqlConn beM` type, which was keeping a native connection, will get a major update. It will hold not a connection itself, but rather a pool, which means we can't simply obtain a native connection from it; rather, we should request it from the pool (we use the `resource-pool` library for this):

```
import Data.Pool as Pool

data SqlConn beM
  = SQLitePool ConnTag (Pool.Pool SQLite.Connection)
```

We can think about this as a possibility for obtaining a connection once it's available in the pool. We can specify how many connections are allowed to be created and how long they should be kept alive. The pool should be configured, hence the following new type and smart constructors:

```
data PoolConfig = PoolConfig
  { stripes :: Int
  , keepAlive :: NominalDiffTime
  , resourcesPerStripe :: Int
  }

mkSQLitePoolConfig
  :: DBName
  -> PoolConfig
  -> DBConfig SqliteM
mkSQLitePoolConfig dbname poolCfg =
  SQLitePoolConf dbname dbname poolCfg
```

We can't just pass a pool here and there. We should concentrate the work with it in a single place and not allow different subsystems to freely access the internals of the `SqlConn` type. To simplify this, we'll introduce the `NativeSqlConn` type for our internal usage. The contract will be, when you successfully query the pool, you get a `NativeSqlConn` value. We'll see how this happens a bit later:

```
data NativeSqlConn = NativeSQLiteConn SQLite.Connection
```

Notably, the interface — the `SqlDBL` language — won't change; it's good enough and can be treated as a transactional context. When you see an `SqlDBL` scenario, you should remember that all the queries will be evaluated by the strategy all-or-nothing. An example of this is presented in listing 7.7.

TIP We discussed the `evalIO` method in chapter 6. This method gives you the ability to run any `IO` action safely:

```
evalIO :: IO a -> LangL a
```

Having this method, you can extend your framework without reworking its core.

Listing 7.7 Multiple queries within a single transaction

```
app :: AppL ()
app = do
  conn <- connectOrFail sqliteCfg
  eRes <- scenario $ evalSqlDB conn $ do
    insertMeteorQuery 1 1 1 1 1 time1
    insertMeteorQuery 2 2 2 2 2 time2
    insertMeteorQuery 3 3 3 3 3 time3
  case eRes of
    Right _  -> evalIO $ print "Successfully inserted."
    Left err -> evalIO $ do
      print "Error got:"
      print err

insertMeteorQuery
  :: Int -> Int -> Int -> Int -> Int -> UTCTime
  -> SqlDBL SqliteM ()
insertMeteorQuery meteorId size mass azmth alt time = ...
```

Three meteors will be inserted if and only if the whole `evalSqlDB` method evaluates successfully, and the transaction is committed.

While this approach looks like an explicit transaction scope, the details about how to deal with transactions depend on a particular database backend. There are relational databases without transactional support, but most of them have it, so we can expect to see methods in native libraries like `openTransaction`, `commitTransaction`, `rollbackTransaction`, or something like that. However, this functionality can be missing in some libraries, although the database supports transactions. In this case, we can try to evaluate raw SQL queries with the corresponding commands and hope that the RDBMS understands it.

The following code shows a bracket-like function for the transactional evaluation of an `IO` action. It's assumed that all the queries to a native library should be done here. Notice that we transform the `SqlConn beM` to a special type that stores native connections:

```
withTransaction
  :: SqlConn beM
  -> (NativeSqlConn -> IO a)
  -> IO (DBResult a)
withTransaction p actF =
  withNativeConnection p $ \nativeConn -> do

    let action = const $ do
      res <- actF nativeConn
      commit nativeConn
      return res

    processResult <$> try $ bracketOnError
                    (begin nativeConn)
                    (const (rollback nativeConn))
                    action
```

Important parts here are the functions `withNativeConnection`, `begin`, `commit`, and `rollback`. They're presented as follows. Notice that we finally found a place where `NativeSqlConn` and the pool started working:

```
begin, commit, rollback :: NativeSqlConn -> IO ()
begin    (NativeSQLiteConn conn) = openSqliteTrans conn
commit   (NativeSQLiteConn conn) = commitSqliteTrans conn
rollback (NativeSQLiteConn conn) = rollbackSqliteTrans conn

withNativeConnection
  :: SqlConn beM
  -> (NativeSqlConn -> IO a)
  -> IO a
withNativeConnection (SQLitePool _ pool) f =
  Pool.withResource pool $ \conn -> f (NativeSQLiteConn conn)
```

A little change is needed in the `BeamRunner` type class. It should now work with `NativeSqlConn`, not with `SqlConn beM`, and maybe some other fixes in the `beam`-related code. But this doesn't offer us any new ideas, so we

can just stop here. If you want to inspect a complete solution, grab the Hydra framework and check it out.

7.4 Summary

This chapter gave us a broad idea of how to organize access to SQL or key-value database subsystems. Not that many usage examples, this is true, but at least we considered

- A simple interface to a particular SQL database (Postgres)
- A more generic interface that supports any SQL database supported by `beam`
- A raw `ByteString`-based interface for two different key-value databases (RocksDB and Redis)
- A typed key-value database interface to a key-value database on top of the raw interface

And by the way, there's a nice principle that we implicitly followed:

> *A very good way to improve some interface is to implement it several times and assess it in several different use cases.*

We also talked about the differences between the database model and domain model. Let me remind you:

- The domain model represents a set of types (and probably functions) directly related to the essence of the domain.
- The database model reflects the idea that the internal representation of data in databases can separate from the domain model and often has other structuring principles.

"Structuring principles" here define how we lay out the data over the internal database structures. Relational model is one principle, stringified JSON is another principle, but usually we don't work with relational structures in the business logic. We would rather operate by lists, trees, maps, and complex ADTs with all the goodies like pattern matching, so we want to convert between the database model and the domain model.

We can either convert things in place by calling some functions right in the business logic, or we can implement some machinery for this and place it in the middle between the business logic layer and the framework layer. Here, we're free to choose how much type safety we need. For example, the `beam` library that we touched on a little assumes that the database model is well-typed, and we just can't avoid defining our table representations in the terms this library exposes. But for the key-value database model, we can either use a simple `ByteString`-based interface with no guarantees in conversions, or we can rely on the typed machinery for the key-value database model, which offers a certain level of guarantees. Still, the type-level magic that we use in Haskell isn't really simple, and we have to balance between the complexity of the code and the guarantees we want to have.

In the next chapter, we'll discuss a boring theme: how to approach the business logic layer. It's boring, but there are still several patterns we can use to make the logic more structured and maintainable. And by the way, business logic is the most important part of the application because it reflects the main value of a business. So stay tuned!

Chapter 8

Business logic design

This chapter covers

- How to layer and structure business logic
- How to decouple parts from services
- Many different approaches to Dependency Injection
- How to create web services with Servant

Programming is simple. I recommend it to everyone: writing code, creating small applications for your own pleasure. Programming is simple. But development — development is extremely hard. So many technological options. So many subtle details. So many controversial requirements. So many things to account for. Writing code is only a small part of development, and not the most difficult part. If you choose to be a software developer, or even a software architect, you'll have to talk with people, understand their needs, clarify requirements, make decisions, and balance between bad, very bad, and unacceptable solutions. Certainly you'll have to learn the domain you're working in. In their professional lives, developers can change their second occupation many times. Today, you're a telecom developer, and you have to be fluent in networking, billing, and accounting. Tomorrow, you learn cryptography and start working for a security company. Next time, you'll move to writing search engines, so you'll buy several new books about natural language processing, algorithms, and data mining. Learning never stops.

In this regard, every developer should study Software Engineering properly because this knowledge is universal. Software Engineering principles can be transferred from one domain to another, from one paradigm to another, from one language to another. Software Engineering allows a developer to not lose the control when changing projects. All software has two parts: an engineering part, with all those technological essentials that drive the application, and a domain part, with business knowledge incorporated into the application somehow. It's more likely that you won't be proficient in the domain part, but you can obtain sufficient experience to freely deal with the engineering part, at least. And if you're a software architect, you must be able to distinguish these two parts to simplify everyone's work. When there's a clear line between the business domain and abstract engineering mechanisms, this reduces risks, simplifies development, and makes the code better.

In this chapter, we'll talk about the "business" part of our applications.

8.1 Business logic layering

No need to explain what business logic is. We discussed it earlier in the form of scenarios, we built domain models, and we described domain behavior many times. Business logic is something we really value; it's the code that does the main thing for the business. With Free monads, it becomes essential to treat the scenario layer as a business logic layer, and we've shown how finely it's separated from the implementation. For example, the Hydra project is organized as figure 8.1 describes.

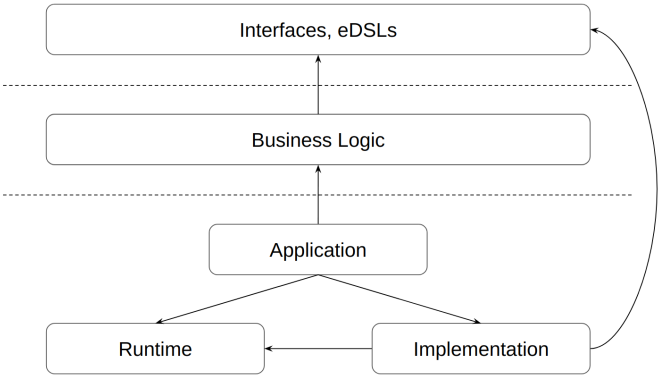


Figure 8.1 Three main application layers

With this architecture, it's impossible to bypass the underlying framework. All the methods you specify in the business logic will go through a single point of processing — the `LangL` language. This is a simplification of the application architecture: you have a predefined, opinionated way to write the code, and your business logic will look more or less identical in all its parts. There's only one way, only one legal viewpoint, only one allowed approach. Limitations evoke ingenuity. Suddenly, you discover many good ideas that otherwise might have remained hidden. For example, one good consequence of having a Free monadic framework is the nice testability of the code. The other one we'll be discussing here: a purified business logic that in turn can be treated as a separate subsystem that's a subject of Software Design. In particular, we'll see why and how to organize additional layers within the business logic layer. And importantly, these patterns can be applied in a wider context as well.

8.1.1 Explicit and implicit business logic

Sometimes, it's hard to recognize what's a part of business logic and what's not. I can imagine some scientific code in Agda, Idris, or Coq that's written not for serving business but for verifying some scientific ideas, and therefore it's strange to call it "business logic." However, in research-only companies, such code will be a business logic in some sense — if we can call the production of papers a "business."

In a badly designed application, we might have a hard time finding what lines do the actual job. All those flaws we call “unclean, dirty, fragile, unmaintainable, unreadable code” distract our attention from the really important things. It’s not a surprise that the books *Code Complete* (by Steve McConnell), *Clean Code* (by Robert C. Martin), and *Refactoring* (by Martin Fowler) are strictly recommended for every developer to read. The advice given in these books teaches you to always remember that you’re a team player and shouldn’t intricate your code, your teammates, and, finally, yourself.

> **LINK** Steve McConnel, *Code Complete*
> https://www.amazon.com/Code-Complete-Practical-Handbook-Construction/dp/0735619670

> **LINK** Robert C. Martin, *Clean Code*
> https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882

> **LINK** Martin Fowler, *Refactoring*
> https://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672

Probably the best way to be transparent lies through the transparency of what the code does. Explicit business logic that’s well-understood and well-written can only be compared to good literature telling a complete story — a story about what really matters for your business. Not particular subsystems, not particular services, but rather a sensitive data you won't leak somehow. Exposing the details of your framework seems safe, but your business logic is your commercial advantage and should be kept secret.

The main conclusion here will be, do your best in separating the layers, try to write your domain logic more explicitly, and don’t bring too much complexity into it. This will drastically improve the quality of the code.

8.2 Exceptions and error handling

Error handling? One of the most difficult themes. I’ve worked with many languages: C++, C#, Python, Haskell — I’ve gained considerable wisdom about coding, but I’ve never seen a way to handle errors that I’d call perfect. I doubt it’s even possible — to have a single, perfect solution. Every project has its own

guidelines on how to do this. Sometimes, exceptions are prohibited; in some projects, people don't distinguish exceptions and errors. But more often, errors are just ignored in the false belief that nothing bad will happen. Even validation, which we'll talk about later, isn't a must-follow practice. After those years of experience, it's still not clear to me if we really have a well-shaped error-handling discipline in software engineering. In this section, I'll present some ideas, so at least you'll be informed, and maybe we'll find a better way.

The theme of error handling can be split into three subtopics:

- Language-specific features to work with errors and exceptions
- General data structures, approaches, and idioms of error handling
- Guidelines on how to apply these tools in projects

First, let's clarify the difference between errors and exceptions.

> **DEFINITION** An *exception* is a language-specific mechanism for terminating the control flow. Exceptions can be caught, but, if they aren't, this usually forces the application to finish urgently to prevent further data corruption and security breaches. Exceptions are mostly implicit and domain-agnostic.

> **DEFINITION** An *exceptional situation* is when the execution of the program has led to an unexpected condition. An exceptional situation is more likely to be an indication of a bug.

> **DEFINITION** An *error* is a probable state in the program that's expected to happen when some specific conditions are met. Errors are mostly explicit, and they're usually related to some domain.

Some languages have mixed practices in which errors and exceptions are indistinguishable, or they substitute for each other. Exceptions as a mechanism may be supported but completely prohibited; see, for example, the following a well-known guide for C++ developers.

> **LINK** *Google's C++ Style Guide*
> https://google.github.io/styleguide/cppguide.html#Exceptions

Haskell's story of exceptions can compete with C++, and this isn't a compliment. Haskell supports synchronous and asynchronous exceptions as language features; also, the community has invented a lot of practices for handling both exceptions and errors. There are

- Synchronous exceptions
- Asynchronous exceptions
- The `Exception` data type and related types
- `Either`, `EitherT`, `ExceptT`, and `ErrorT` monads
- `Maybe` and `MaybeT` monads
- `MonadMask`, `MonadThrow`, `MonadCatch`, `MonadBracket`, and other type classes (generally usable within the mtl/Final Tagless setting)
- Something else?

In my opinion, the situation with errors and exceptions in Haskell can't be called "good, simple, easy to grasp." I would even say it's a total disaster. Each of these approaches tries to be useful, but it turns out that you can't be sure you've considered all the possible cases and done it right. The intersection of the approaches in a particular code base can confuse everything completely. There are different articles describing why you should prefer one approach over another, and maybe you'll find them appropriate to your requirements.

> **LINK** Michael Snoyman, *Exceptions Best Practices in Haskell*
> https://www.fpcomplete.com/blog/2016/11/exceptions-best-practices-haskell

> **LINK** Michael Snoyman, *Asynchronous Exception Handling in Haskell*
> https://www.fpcomplete.com/blog/2018/04/async-exception-handling-haskell/

> **LINK** Mark Karpov, *Exceptions Tutorial*
> https://markkarpov.com/tutorial/exceptions.html

This book wouldn't be complete without its own approach. Fourteen competing standards, what's one more or one less, huh? Yes, but there's one key difference: I'm offering a consistent philosophy of exceptions and errors within the Hierarchical Free Monads setting. It won't be one-hundred-percent correct but it will work for 95% of cases. Let's see ...

8.2.1 Error domains

We've seen how Free monads help to separate concerns: interfaces, implementation, and business logic. It would be natural to bound each layer with a specific way of error processing. One layer, one method. Thus, we divide our application into error domains.

> **DEFINITION** An *error domain* is an isolated part of an application that has a predefined error-handling mechanism and provides a guarantee that there will be no other behavior. Don't confuse *error domain* and *domain error*.

Error domains don't fit layers strictly, because it's allowed that several layers be unified by a single error domain. Also, you can have many error domains within a layer. There's no strict rule except one: as an error mechanism defines a domain, all the artifacts that are crossing the boundary between domains should be converted from one form into another. Figure 8.2 introduces error domains for a Free monad application.

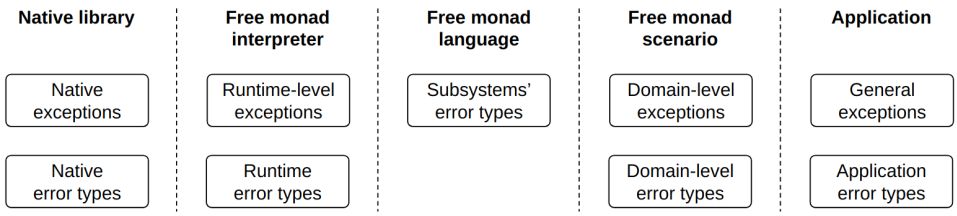


Figure 8.2 Error domains

From the figure, you can conclude that native errors and native exceptions should not leak into Free monad languages. Ideally, all the native error types should be converted into a more abstracted form by the interpreter. This is very similar to native types that we saw in the previous chapter. Reminder: we converted a native connection type into our own, and also created some framework-level types for database errors. More precisely, we decouple a pure world of Free monad languages and scenarios from the impure world of native libraries. The interpreter here represents a nice mediator between the two. Besides this, interpreters can have their own internal exceptions and errors. Here, design choices depend on your requirements (for example, your

interpreter's stack can be wrapped into the `ExceptT` monad transformer), but we can discuss several schemes emerging from common needs.

8.2.2 Native exceptions handling schemes

The first case occurs when an external library throws a synchronous exception. The interpreter should capture it and convert if necessary. After this, a value can be passed back into the Free monad scenario. Alternatively, the value can be ignored — so the scenario should not expect it. Also, you can convert the native value in the interpreter, or on the upper level, which is the methods of the Free language. A diagram with these approaches is presented in figure 8.3.

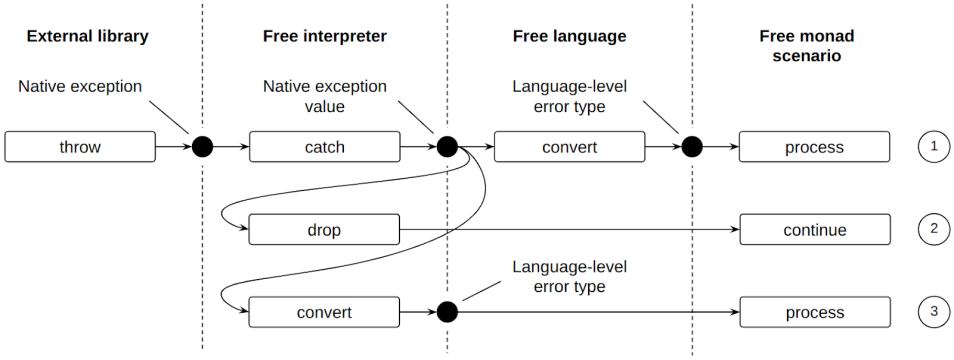


Figure 8.3 Working with native exceptions from external libraries

Let's construct a sample to accompany the diagram. For instance, we want to add a Free monad language `FileSystem`. We know that `readFile` and `writeFile` functions may throw, and we want to wrap the native `IOException` in a custom error type:

```
import qualified GHC.IO.Exception as IOE

type NativeResult a = Either IOE.IOException a

data FSError
  = FileNotFound (Maybe FilePath)
  | OtherError String

type FSResult a = Either FSError a
```

Listing 8.1 contains a code for option #1, in which we convert `IOException` not in the interpreter but rather in the smart constructors on the language level. Sometimes, we're required to exhibit native types from the Free monad language, so in this case, there can be two constructors for convenience.

Listing 8.1 Converting exceptions into errors on the language level

```
data FileSystemF next where                                    #A
  WriteFile :: FilePath -> String
            -> (NativeResult () -> next) -> FileSystemF next

type FileSystem = Free FileSystemF

writeFileIO                                                    #B
  :: FilePath -> String -> FileSystem (NativeResult ())
writeFileIO filePath content =
  liftF $ WriteFile filePath content id

writeFile' :: FilePath -> String                               #C
           -> FileSystem (FSResult ())
writeFile' filePath content = do
  eRes <- writeFileIO filePath content
  pure $ fromNativeResult eRes

fromNativeResult :: NativeResult a -> FSResult a               #D
fromNativeResult (Right a) = Right a
fromNativeResult (Left ioException) = let
  fileName = IOE.ioe_filename ioException
  errType  = IOE.ioe_type ioException
  in case errType of
       IOE.NoSuchThing -> Left $ FileNotFound fileName
       _               -> Left $ OtherError $ show errType

interpretFileSystemF :: FileSystemF a -> IO a                  #E
interpretFileSystemF (WriteFile p c next) =
  next <$> (try $ writeFile p c)
```

#A This language is aware of the native error type
#B Smart constructor, returns a native error type
#C Smart constructor, returns a custom error type
#D Native error to custom error converter
#E Interpreter that calls a native function

Interestingly, you can do a lot in smart constructors. They're just normal functions of your language, and you can even call other neighbor methods. We do this in the preceding code. But beware of recursive calls, which can be dangerous.

Option #3 from the diagram looks very similar, but the transformation happens in the interpreter. This is probably a more preferable design because we don't want to allow native types to leak and appear in the language layer. The relevant part of the code looks like this:

```
data FileSystemF next where
  WriteFile :: FilePath -> String
            -> (FSResult () -> next)
            -> FileSystemF next

writeFile' :: FilePath -> String
           -> FileSystem (FSResult ())
writeFile' filePath content =
  liftF $ WriteFile filePath content id

interpretFileSystemF :: FileSystemF a -> IO a
interpretFileSystemF (WriteFile p c next) = do
  eRes <- try $ P.writeFile p c
  pure $ next $ fromNativeResult eRes
```

What about option #2? Well, the actual result of `writeFile'` is unit `().` We could just ignore the error, but this is a different semantics of the method, so we should reflect it somehow — for example, in the documentation:

```
-- This method doesn't provide a guarantee
--   of a successful write operation.
-- In case of exceptions, nothing will happen.
writeFile' :: FilePath -> String -> FileSystem ()
writeFile' filePath content =
  void $ liftF $ WriteFile filePath content id
```

Voiding the result can also be done in the interpreter, if you will.

However, neither of these techniques will work properly without following two golden rules:

- *Interpreters should catch all the exceptions from the other world.* In other words, there should be no eDSL method allowing the external exceptions to slip into the business logic.
- *Asynchronous exceptions should be prohibited.* Neither the framework nor the underlying libraries should emit asynchronous exceptions because it's really hard to reason about the logic. All the things become really complicated. Even more, the need for asynchronous exceptions indicates design flaws. Asynchronous exceptions violate the Liskov substitution principle because they're able to crash an innocent thread very suddenly.

By following these two rules, we can guarantee that nothing unexpected can happen. All the worlds are effectively separated from each other; all the code is under control. This gives you enough clarity on how the logic behaves and how to fix possible problems. Look at figure 8.4 and judge for yourself.

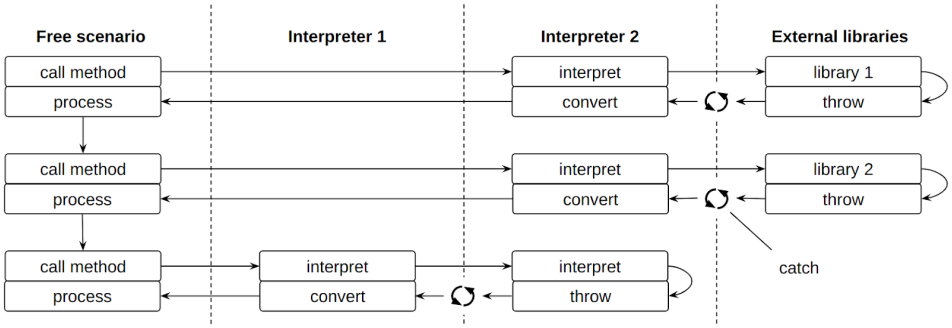


Figure 8.4 Total exceptions gatekeeping

But this is only half of the methodology. Let's complete it by considering one more mechanism: domain-level exceptions and the way to work with them in your Free monadic languages.

8.2.3 Free monad languages and exceptions

There's some specificity on how exceptions move through the business logic. It looks like the control flow zigzags between an interpreter and a Free monadic scenario, like in figure 8.4. For a long time, it was even believed that Free monads don't allow any exception handling. The argument was that the exception will terminate the control flow, and the evaluation point will never

return to the scenario, so there's no sense in having a "catch" method in the language.

Let's consider an example to clarify this. In the previous chapter, we wrote a function `meteorsApp`. It could initialize SQL DB, check for success, and proceed with a created connection. But in case of a failure, it just closed after logging a message:

```
meteorsApp :: AppL ()
meteorsApp = do
  eConn <- initSQLiteDB "./meteors.db"
  case eConn of
    Left err -> logError "Failed to init connection to SQLite."
    Right conn -> do
      meteors <- getMeteorsWithMass conn 100
      logInfo $ "Meteors found: " <> show meteors
```

Now, for some reason, we want to throw a domain-level exception instead of logging the error. Also, we want to regroup the code such that there will be an unsafe method for connecting to the database. The script will change:

```
-- Domain-level exceptions
data AppException
  = InvalidOperation String
  deriving (Show, Read, Eq, Ord, Generic,
            ToJSON, FromJSON, Exception)

-- Unsafe operation
unsafeInitSQLiteDB :: DBName -> AppL (SQLConn SqliteM)
unsafeInitSQLiteDB dbName = do
  eConn <- initSQLiteDB dbName
  when (isLeft eConn) $ logError "Init connection failed."
  case eConn of
    Left err   -> throwException $ InvalidOperation $ show err
    Right conn -> pure conn

meteorsApp :: AppL ()
meteorsApp = do
  conn    <- unsafeInitSQLiteDB "./meteors.db"
  meteors <- getMeteorsWithMass conn 100
  logInfo $ "Meteors found: " <> show meteors
```

So, we have a `throwException` method that comes from the framework. Obviously, the interpreter of this method should throw a real exception, and once this happens, the script stops evaluating on the `unsafeInitSQLiteDB` method, and the whole `AppL` program terminates. The exception will then appear at the place where we started our Free monadic evaluation:

```
eResult <- try $ runAppL rt meteorsApp
case eResult of
  Left (err :: SomeException) -> ...
  Right _ -> ...
```

The corresponding scheme is presented in figure 8.5.

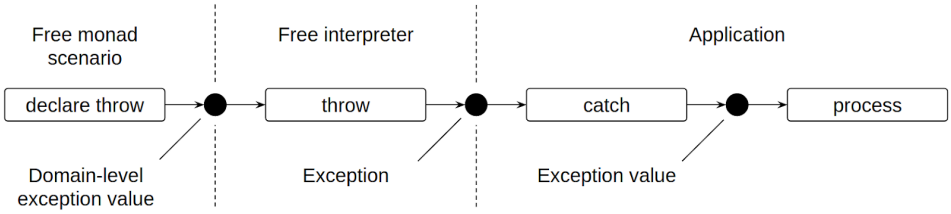


Figure 8.5 Handling exceptions on the application level

If we want to capture it while staying in the logic, we should write something like this:

```
meteorsApp :: AppL ()
meteorsApp = do
  eConn <- try $ unsafeInitSQLiteDB "./meteors.db"
  case eConn of
    Left (err :: SomeException) -> ...
    Right conn -> ...
```

But we can't use `try` because it's specified for the `IO` type only:

```
try :: Exception e => IO a -> IO (Either e a)
```

And I believe this was the main concern against the very possibility of handling exceptions in Free monads. But who said that we can't define our own combinator for this? Ladies and gentlemen, the `runSafely` method in propria

persona! And the `throwException` method for your pleasure. See listing 8.2.

Listing 8.2 New methods: throwException and runSafely

```
data LangF next where

  ThrowException
    :: forall a e next
     . Exception e
    => e -> (a -> next) -> LangF next

  RunSafely
    :: forall a e next
     . Exception e
    => LangL a -> (Either e a -> next) -> LangF next

throwException :: forall a e. Exception e => e -> LangL a
throwException ex = liftF $ ThrowException ex id

runSafely :: Exception e => LangL a -> LangL (Either e a)
runSafely act = liftF $ RunSafely act id
```

Notice that we nest a `LangL` action recursively into the `LangL` eDSL. Because of this, the interpreter can run itself recursively and try to catch the specified exception:

```
import Control.Exception (throwIO)

interpretLangF :: LangF a -> IO a
interpretLangF (ThrowException exc _) = throwIO exc
interpretLangF (RunSafely act next) = do
  eResult <- try $ runLangL coreRt act
  pure $ next $ case eResult of
    Left err -> Left err
    Right r  -> Right r

runLangL :: LangL a -> IO a
runLangL = foldFree interpretLangF
```

And yeah, this is it. No magic, actually. The `runSafely` and `try` combinators have the same semantics. An exception will be handled if its type

matches the `e` type from the definition, or if the type is a subtype of a more general exception, for example, `SomeException`, which is the most general in Haskell:

```
meteorsApp :: AppL ()
meteorsApp = do
  eConn <- runSafely @SomeException
    $ unsafeInitSQLiteDB "./meteors.db"

  case eConn of
    Left (err :: SomeException) -> ...
    Right conn -> ...
```

To be honest, I love these diagrams, so I prepared one more that shows this schematically. See figure 8.6.

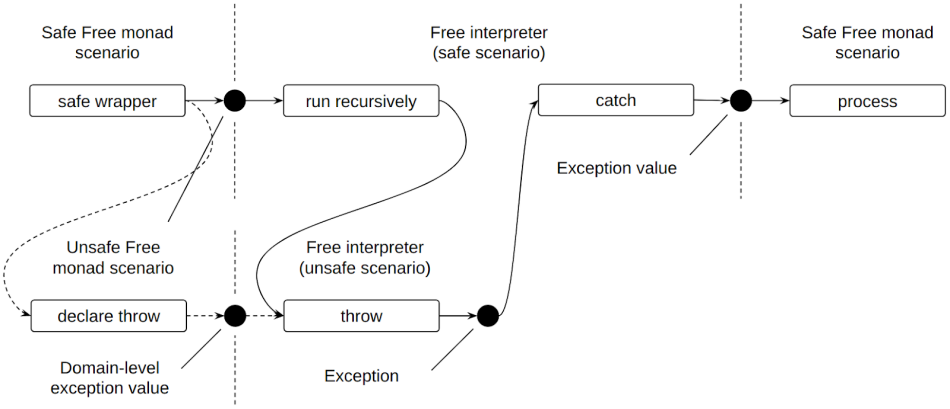


Figure 8.6 Exception flow in the Free monadic scenario

Let’s make some conclusions:

- We added two new methods for handling exceptions. The hierarchical nature of our framework helped with this because we can nest one action in another and run the interpreters recursively.
- It’s now possible to implement many functions from the `Control.Exception` module. `bracket`? Yes. `finally`? Of course. `onException`? No problem.
- We divided the whole application into domain errors and thus simplified

the reasoning about each of them. The divide and conquer principle, again.

- We now better understand the distinction between exceptions and errors. We can convert one into another if needed.
- We established a philosophy of error handling. And I believe it's much simpler than other approaches.

8.3 A CLI tool for reporting astronomical objects

In the next part of this chapter, we'll be discussing the more practical questions: command-line applications, web servers, and validation. We'll consider even more layering. But this time, we'll learn how to slice the business logic layer itself, and how to do Dependency Injection with different tools. We'll talk about REST APIs and application-wide configs. Generally speaking, we'll be battling with the most common tasks in the software world.

Now, to be more specific, we need to provide a common ground for the upcoming talk. What business logic will we be dissecting? In the previous chapter, we were working on the code for the astronomical domain. We created a database for storing meteors, and we wrote several queries. Supposedly, this logic should be a part of a server for tracking sky events. We don't have that server yet — neither a REST server with an HTTP interface, nor a remote server with some other interface like TCP or web sockets.

What would you do if you were tasked with implementing a client for this API, but the requirements weren't available at the moment? Does it really block you? Would you start implementing at least a skeleton of the client that you're sure will be unable to interact with the server until the situation clears up? A good strategy here is to not lose time and money. The client application should have several components implemented irrespective of the channel interface to the server. This is perfectly approachable. And to avoid redesigning the application, we'll hide the interaction with the server behind a generic interface: the interface for reporting an asteroid.

8.3.1 API types and command-line interaction

Consider that the API type for the meteor is the following type (should be convertible to / from JSON for transferring via HTTP or TCP):

```
data MeteorTemplate = MeteorTemplate
  { size        :: Int
  , mass        :: Int
  , azimuth     :: Int
  , altitude    :: Int
  }
  deriving (Generic, ToJSON, FromJSON)
```

The interface for a sending function should operate within the `AppL` language (or possibly within the `LangL` language):

```
reportMeteor
  :: MeteorTemplate
  -> AppL (Either ByteString MeteorId)
reportMeteor m = ...
```

In the future, we'll add more features to the client and server. For example, why not add reporting asteroids, pulsars, exoplanets, and other interesting stuff?

```
data AsteroidTemplate = AsteroidTemplate
  { name        :: Maybe Text
  , orbital     :: Orbital       -- Orbital and physical
  , physical    :: Physical      -- characteristics
  }
  deriving (Generic, ToJSON, FromJSON)

reportAsteroid
  :: AsteroidTemplate
  -> AppL (Either ByteString AsteroidId)
reportAsteroid a = ...
```

The user should be able to run the client application and input the data he wants to send. The client application should be configured to parse a string typed into the CLI, convert it into an astronomical object, and report to the server. Normally, conversion to and from JSON is done by `aeson`. We certainly want the `HTTP` interface because it's a de facto for the Web. We'd like to use the same data types for both the client and server side. Happily, this is possible with the `servant` library, another notable tool in modern Haskell. It offers both server and client sides, and the `servant` library is compatible with `aeson`. The HTTP API for the astro server is presented in listing 8.3.

Listing 8.3 HTTP API for server and client side

```
type AstroAPI
  = (  "meteor"                                    #A
    :> ReqBody '[JSON] MeteorTemplate              #B
    :> Post '[JSON] MeteorId                       #C
    )
  :<|>
    (  "asteroid"
    :> ReqBody '[JSON] AsteroidTemplate
    :> Post '[JSON] AsteroidId
    )
```

#A Method POST, route "/meteor"
#B JSON body should be MeteorTemplate
#C Method returns MeteorId

The server will take care of parsing once `MeteorTemplate` and `AsteroidTemplate` have the instances for `ToJSON/FromJSON`. The client API will utilize the same `AstroAPI` description, which is very convenient. We'll see how to do this in the next section.

While the HTTP part is nicely handled by Servant, what can we do with the CLI part? Again, we want to reuse the same types `AsteroidTemplate` and `MeteorTemplate`. How should we approach the parsing? There's a little pitfall here. Let's write a piece of code:

```
consoleApp :: AppL ()
consoleApp = do
  -- Reading user's input
  line <- evalIO getLine

  -- Trying to decode
  let mbMeteor   = decode line
  let mbAsteroid = decode line

  -- Checking the result and sending
  case (mbMeteor, mbAsteroid) of
     (Just meteor, _)   -> reportMeteor meteor
     (_, Just asteroid) -> reportAsteroid
     (Nothing, Nothing) -> logWarning "Unknown command"
```

You see? Not beautiful. Two lines for each object! A tuple component for each object! Adding more objects will multiply the number of lines. What a waste of letters. Can we make it a little bit better? Yes, we can, but we'll encounter the problem of how to specify a type of expected object because it can vary. Let me explain.

The first function we'll need is `tryParseCmd`:

```
tryParseCmd :: FromJSON obj
    => ByteString -> Either ByteString obj
tryParseCmd line = case decode line of
  Nothing  -> Left "Decoding failed."
  Just obj -> Right obj
```

It's polymorphic by return type. Thus, to use it with different types, we have no other choice than to pass the type somehow, like this (pseudocode, won't compile):

```
let supportedObjects =
     [ tryParseCmd @MeteorTemplate   line
     , tryParseCmd @AsteroidTemplate line
     ]
```

The idea is to fold this structure and get either a single object parsed or an error value if none of the types match the contents of the string. Unfortunately, this won't compile because the `supportedObjects` list isn't homogenous (the two items return different types). Happily for you, there's a nice trick. We should parse an object and immediately utilize its result with a reporting function. The result of such a function will be the same each time, and it won't be a problem to keep as many parsers in the list as you want. The following code illustrates this, step by step.

Declarative specification of the supported objects and commands:

```
let runners =
     [ reportWith reportMeteor   $ tryParseCmd line
     , reportWith reportAsteroid $ tryParseCmd line
     ]
```

`reportWith` function that utilizes the result of parsing irrespective of the object type:

```
reportWith
  :: (FromJSON obj, Show obj)
  => (obj -> AppL (Either ByteString res))          #A
  -> (Either ByteString obj)                        #B
  -> AppL (Either ByteString String)

reportWith _ (Left err) = pure $ Left err           #C
reportWith reporter (Right obj) = do
  reporter obj                                      #D
  pure $ Right $ "Object reported: " <> show obj
```

#A Specific reporter for the object
#B Either a parsed object or a failure
#C Return error if the string is not recognized
#D Calling a specific reporter

Our reporting functions `reportMeteor` and `reportAsteroid` provide the needed type info for the `reportWith` function. Now, we just run all these monadic actions in a single shot with the monadic sequence function. The action that got a value successfully parsed will be evaluated; other actions will be ignored. Listing 8.4 puts all the pieces together.

Listing 8.4 Complete code of business logic

```
consoleApp :: AppL ()
consoleApp = forever $ do
  line <- evalIO getLine

  let mbParsed = tryParseCmd line
  let runners =
       [ reportWith reportMeteor   mbParsed
       , reportWith reportAsteroid mbParsed
       ]

  eResults <- sequence runners
  printResults eResults

  where
```

```
printResults :: [Either ByteString String] -> AppL ()
printResults [] = pure ()
printResults (Right msg : rs) =
    putStrLn msg >> printResults rs
printResults (_ : rs) = printResults rs
```

One might think, “Pulsars, comets, stars, black holes, exoplanets — too much stuff to add manually, so why not invent a generic mechanism here? This code could accept an arbitrary object type without even touching the business logic! Right?”

Well, maybe. Sure you have to edit this code once you want to support a new object, but we won’t go further in making this part more generic. That path is very slippery — you may occasionally find yourself solving the Expression Problem with advanced type-level stuff with no real benefit on the horizon. It’s pretty simple to update the runners list manually, and it’s fine to have some extra work considering that the new report functions should be written anyway. So, we’ll stop here with the objects extensibility problem.

But don’t worry, there’s a theme about extensibility we should discuss: extensibility of the reporting methods. Currently, the functions `reportMeteor` and `reportAsteroid` do something. We don’t yet know how, exactly, they send objects to the server. Let’s figure this out.

8.3.2 HTTP and TCP client functionality

Now, before we move further, let’s briefly learn how to implement the `HTTP` client functionality with the help of `servant-client`, a library additional to Servant. We already have the API description (see listing 8.3). The rest of the work is straightforward: provide a bunch of helper functions and incorporate the client functionality into the business logic.

First, we’ll add some helpers for the client API. We take the `AstroAPI` and wrap it like this:

```
import Servant.Client (ClientM, client)

-- Helper methods description
meteor   :: MeteorTemplate   -> ClientM MeteorId
asteroid :: AsteroidTemplate -> ClientM AsteroidId

-- Helper methods generation
(meteor :<|> asteroid) = client (Proxy :: AstroAPI)
```

Second, we'll add a new method for calling these helpers into the `LangL` language. Essentially, this is how the `servant-client` mechanism can be incorporated into the framework:

```
import Servant.Client (ClientM, ClientError, BaseUrl)

data LangF next where
  CallServantAPI
    :: BaseUrl -> ClientM a
    -> (Either ClientError a -> next)
    -> LangF next

callAPI :: BaseUrl -> ClientM a
        -> LangL (Either ClientError a)
callAPI url clientAct =
  liftF $ CallServantAPI url clientAct id
```

As you can see, we store the `ClientM` action here to later run it in the interpreter. Going ahead, I notice that there should be a pre-created *`HTTP client manager`* to run the `ClientM` action. Things are slightly complicated by the fact that the manager should be exclusive across the whole application. This isn't a big problem for us, though: the manager is an implementation detail, and why not just put it into the runtime structures? The interpreter will take it out, use it, and return it. Check out the complete code of this design in the Hydra project; it's really simple. For this story, however, we'd like to stay on the level of the eDSLs and business logic because we're not interested in the hidden details of the HTTP client library.

Finishing the client business logic, we add the following function for reporting meteors:

```
reportMeteorHttp
  :: BaseUrl
  -> MeteorTemplate
  -> AppL (Either ByteString MeteorId)
reportMeteorHttp url m = do
  eMeteorId <- scenario
    $ callAPI url                                #A
    $ meteor m                                   #B
  pure $ case eMeteorId of
    Left err       -> Left $ show err
    Right meteorId -> Right meteorId
```

#A Method of the LangL language
#B Client method for the HTTP API

The same works for asteroids and other stuff you want to query. No need to show the `reportAsteroidHttp` function, right?

Now, two words on what we'll be doing next. Let's consider all these reporting functions to be implementation details, even if they're operating on the business logic layer:

```
reportMeteorHttp
  :: BaseUrl -> MeteorTemplate
  -> AppL (Either ByteString MeteorId)
reportAsteroidHttp
  :: BaseUrl -> AsteroidTemplate
  -> AppL (Either ByteString AsteroidId)
reportMeteorTcp
  :: TcpConn -> MeteorTemplate
  -> AppL (Either ByteString MeteorId)
reportAsteroidTcp
  :: TcpConn -> AsteroidTemplate
  -> AppL (Either ByteString AsteroidId)
```

You can spot the common interface, here, which we saw earlier:

```
reportMeteor
  :: MeteorTemplate -> AppL (Either ByteString MeteorId)

reportAsteroid
  :: AsteroidTemplate -> AppL (Either ByteString AsteroidId)
```

In the next sections, we'll be discussing various ways to hide the implementation functions behind a common interface. Our `clientApp` should be completely unaware what channel is used: TCP, HTTP, or something else. We're big boys and girls, we're now allowed to inject these "services" as dependencies. In this particular task, I want to teach you how to do Dependency Injection in Haskell, along with introducing several approaches to functional interfaces. We had such a talk already in chapter 3, "Subsystems and services," but we didn't have a chance to carefully compare the approaches. Especially as there are even more of them described in this chapter:

- Service Handle
- ReaderT
- Free monad
- GADT
- Final Tagless (mtl)

What's interesting here, is that it seems that there are two meta kinds of functional interfaces in Haskell:

- *Scenario-like interfaces*. These interfaces are intended for writing scenarios, scripts, and sequential logic in general. The most sensible example is all our framework eDSLs: `AppL`, `LangL`, `LoggerL`, and other monadic languages. We have a lot of knowledge about this stuff!
- *API-like interfaces*. The methods of these interfaces are intended only for solitary usage, and the design of these methods doesn't imply a chaining. These interfaces are needed only to represent some API. Samples are `REST`, `ClientM` from Servant, and our reporting methods.

This meta kind of interface is somewhat new to us. Finally, we can start investigating it. Let's go!

8.4 Functional interfaces and Dependency Injection

Dependency Injection in OOP serves the purpose of providing an implementation hidden behind an interface so that the client code need not bother about the implementation details and is completely decoupled from them.

This technique becomes very important in big systems with a significant amount of IO operations. We usually say such systems are IO bound, not CPU bound, meaning the main activity is always related to external services or other effects. Controlling these effects in the business logic becomes very important, and Dependency Injection helps with this, making the logic less complex and more maintainable. In functional programming, we aren't freed from IO. Even in functional programming, it's nice to have a kind of Dependency Injection because this principle is paradigm-agnostic. We talked about this already in chapter 3 and discussed several approaches there. While implementing a Free monad layer for core effects is still a viable idea, we might also want to use interfaces and Dependency Injection within business logic to make it even more simple and approachable.

8.4.1 Service Handle Pattern

The Service Handle Pattern (also known as Service Pattern or Handle Pattern) is the simplest way to describe an interface for a subsystem. For the client program, we may want to create the following interface:

```
data AstroServiceHandle = AstroServiceHandle
  { meteorReporter
        :: MeteorTemplate
        -> AppL (Either ByteString MeteorId)
  , asteroidReporter
        :: AsteroidTemplate
        -> AppL (Either ByteString AsteroidId)
  }
```

Here, we store the reporting functions in a data structure for greater convenience. It's possible to just pass those functions (`meteorReporter` and `asteroidReporter`) as arguments, which will be fine once you have a small number of them. But for services with dozens of methods, grouping into handle structures is better.

Nothing more to add here except the usage. You pass a handle into your business logic like this:

```
consoleApp :: AstroServiceHandle -> AppL ()
consoleApp handle = do
  line <- getUserInput

  let mReporter = meteorReporter handle
  let aReporter = asteroidReporter handle

  let mbParsed = tryParseCmd line
  let runners =
        [ reportWith mReporter mbParsed                 #A
        , reportWith aReporter mbParsed
        ]

  eResults <- sequence runners
  printResults eResults
```

#A Using methods of handler

Providing a constructor for the handle is a good idea:

```
data ReportChannel = TcpChannel | HttpChannel

makeServiceHandle :: ReportChannel -> AstroServiceHandle
```

Handle with TCP reporters. For now, TCP channel config is hardcoded:

```
tcpConfig = TcpConfig "localhost" 33335

makeServiceHandle TcpChannel =
  AstroServiceHandle
    (reportMeteorTcp   tcpConfig)
    (reportAsteroidTcp tcpConfig)
```

Handle with HTTP reporters, with HTTP config hardcoded:

```
baseUrl = BaseUrl Http "localhost" 8081 ""

makeServiceHandle HttpChannel =
  AstroServiceHandle
    (reportMeteorHttp   baseUrl)
    (reportAsteroidHttp baseUrl)
```

You don't like the hardcode? Okay, just pass the configs into the `makeServiceHandle` helper:

```
makeServiceHandle
  :: ReportChannel -> BaseUrl
  -> TcpConn -> AstroServiceHandle
```

The last thing is running the `consoleApp` by injecting a service handle into it:

```
main = do
  ch :: ReportChannel <- getReportChannel              #A
  runtime <- createRuntime                             #B
  result <- runApp runtime                             #C
    $ consoleApp (makeServiceHandle ch)                #D
```

#A Getting ReportChannel somehow
#B Creating the runtime
#C Running a normal AppL scenario
#D Configured by a specific service implementation

By reading a `ReportChannel` value from the command line on the application start, you can specify how your program should behave. Alternatively, you can pass it via environment variable or using a config file. This is a pretty common practice in Java and C#, when there's a text file (usually XML) that contains different settings and configs for particular subsystems. Many IoC containers allow you to choose the implementation and thus have additional flexibility for the system. All the patterns we learn here may be a basis for such a Dependency Injection framework, and I'm sure we'll see several of them in Haskell soon.

> **NOTE** This actually happened. Eric Torrebore created a DI framework: `registry`.
> https://github.com/etorreborre/registry

> **NOTE** Seems that the recent release of a framework *Integrated Haskell Platform (IHP)* also confirms this tesis.
> https://ihp.digitallyinduced.com/

8.4.2 ReaderT Pattern

The `ReaderT` pattern has the same idea and a very similar implementation. We're not passing a handle structure now, we're hiding it in the `ReaderT` environment. Alternatively, the `StateT` transformer can be used because `ReaderT` is just half of it. Let's check out the code.

There should be a handle-like structure; let's now call it `AppEnv` (environment for the business logic):

```
data AppEnv = AppEnv
  { meteorReporter
     :: MeteorTemplate
     -> AppL (Either ByteString MeteorId)
  , asteroidReporter
     :: AsteroidTemplate
     -> AppL (Either ByteString AsteroidId)
  }
```

If you're trying to find the differences with `AstroServiceHandle`, there are none. Just another name for the same control structure to keep the naming more consistent. The difference is how we pass this structure into logic. We actually don't. We store this handle in the `Reader` context in which we wrap our `AppL` monad:

```
type AppRT a = ReaderT AppEnv AppL a
```

As all our business logic should be wrapped into the `ReaderT` monad transformer, our code starts to look slightly overloaded by occasional lifts. Check this out:

```
consoleApp :: AppRT ()
consoleApp = do
  AppEnv {meteorReporter, asteroidReporter} <- ask         #A

  line <- getUserInput

  let mbParsed = tryParseCmd line
  let runners =
        [ lift $ reportWith meteorReporter mbParsed
        , lift $ reportWith asteroidReporter mbParsed ]
```

```
  eResults <- sequence runners
  lift $ printResults eResults
```

#A Getting the control structure from the ReaderT environment

Some adjustments are needed for the runner. Just because the actual Free monadic scenario is wrapped into the `ReaderT` transformer, it should be "unwrapped" first by providing a `Reader` environment variable:

```
main = do
  ch :: ReportChannel <- getReportChannel              #A
  let appEnv = makeAppEnv ch                           #B
  result <- runApp runtime                             #C
    $ runReaderT consoleApp appEnv                     #D
```

#A Getting ReportChannel somehow
#B Creating a handle
#C Running a normal AppL scenario
#D Wrapped into the ReaderT environment

Notice how many lifts happened in the business logic. The `reportWith` function and the `printResults` function both have the type `AppL`, so we can't just call these functions within the `AppRT` monad. Lifting between the two monad layers here is unavoidable. Or not? In Haskell, there are ways to reduce the amount of boilerplate. Just to name some: additional type classes for the `AppRT` type; a newtype wrapper for the `ReaderT` monad powered by some automatic derivings; and others. I'll leave this question for you to contemplate.

It's unfortunate that we had to do so many steps. This is probably too high a cost for just removing the handle from `consoleApp` arguments. Maybe it's not worth it. Just use the Service Handle Pattern. Keep it simple.

8.4.3 Additional Free monad language

To satisfy our curiosity, let's see how much code will be required for the Free monad interface. This is the language and smart constructors:

```
data AstroServiceF a where
  ReportMeteor
    :: MeteorTemplate
    -> (Either ByteString MeteorId -> next)
    -> AstroServiceF next
  ReportAsteroid
    :: AsteroidTemplate
    -> (Either ByteString AsteroidId -> next)
    -> AstroServiceF next

type AstroService a = Free AstroServiceF a

reportMeteor
   :: MeteorTemplate
   -> AstroService (Either ByteString MeteorId)
reportMeteor m = liftF $ ReportMeteor m id

reportAsteroid
   :: AsteroidTemplate
   -> AstroService (Either ByteString AsteroidId)
reportAsteroid a = liftF $ ReportAsteroid a id
```

The `AstroServiceF` should be a `Functor` instance. No need to show it again. More interesting is how the interpreters should look. You might have guessed that there should be two interpreters for two communication channels. The interpreters transform the `AstroService` scenario into the `AppL` scenario, which differs from our usual transformation of Free languages into the IO stack. Here's the interpreter for the HTTP channel:

```
asHttpAstroService :: AstroServiceF a -> AppL a
asHttpAstroService (ReportMeteor m next) =
  next <$> reportMeteorHttp tcpConn m
asHttpAstroService (ReportAsteroid a next) =
  next <$> reportAsteroidHttp tcpConn a
```

The HTTP version is pretty much the same: one more additional function for traversing the algebra with the same type definition:

```
asTcpAstroService :: AstroServiceF a -> AppL a
```

The Free monad runner will be common for the two interpreters. It's parameterizable:

```
runAstroService
  :: (forall x. AstroServiceF x -> AppL x)
  -> AstroService a -> AppL a
runAstroService runner act = foldFree runner act
```

Both functions `asHttpAstroService` and `asTcpAstroService` can be passed into it as the first argument runner. To avoid revealing these functions, we add the following constructor and use it in the `AppL` runner:

```
getAstroServiceRunner
  :: ReportChannel -> (AstroServiceF a -> AppL a)
getAstroServiceRunner TcpChannel  = asTcpAstroService
getAstroServiceRunner HttpChannel = asHttpAstroService

main = do
  ch :: ReportChannel <- getReportChannel                #A
  let astroServiceRunner = getAstroServiceRunner ch      #B
  result <- runApp runtime                               #C
    $ consoleApp astroServiceRunner                      #D
```

#A Getting ReportChannel somehow
#B Getting a service implementation (runner)
#C Evaluating a normal AppL scenario
#D Configured by the service runner

And what about the logic? Is it good? The logic is fine. It doesn't know anything about the internals of the `AstroService` runner. It just uses the additional Free monad language with it:

```
consoleApp
  :: (forall x. AstroServiceF x -> AppL x) -> AppL ()
consoleApp astroServiceRunner = do

  line <- getUserInput

  let mbParsed = tryParseCmd line
  let runners =
        [ reportWith astroServiceRunner reportMeteor mbParsed
        , reportWith astroServiceRunner reportAsteroid mbParsed
        ]

  eResults <- sequence runners
  lift $ printResults eResults
```

The `reportWith` function should call the Free monad interpreter (`runAstroService`):

```
reportWith
  :: FromJSON obj
  => (forall x. AstroServiceF x -> AppL x)
  -> (obj -> AstroService a)
  -> (Either ByteString obj)
  -> AppL (Either ByteString ())
reportWith runner _        (Left err)  = pure $ Left err
reportWith runner reporter (Right obj) = do
  void $ runAstroService runner $ reporter obj        #A
  pure $ Right ()
```

#A Calling the interpreter

In comparison to the `ReaderT` approach, the Free monad approach grows in another direction. There's no additional lifting here, but the Free monad machinery requires a bit more effort to implement. However, lifting in the `ReaderT` approach is pure evil for the business logic: so many things completely unrelated with the actual domain! And if we want to hide, the additional machinery will be as boilerplatey as with the additional Free monad language.

This is clearly a question of the appropriate usage of tools. Having a Free monadic framework as the application basis is one story, and going this way on the business logic is another story. But maybe it's the task we're solving here that doesn't give us much freedom. Anyway, just use the Service Handle. Keep it simple.

8.4.4 GADT

Although you might find the GADT solution very similar to Free monads, this is only because the interface we're implementing is an API-like interface. For this simple service, the GADT solution will be better than Free monads because we don't need a sequential evaluation. Let's elaborate.

Conciseness of the API language makes us happy:

```
data AstroService a where
  ReportMeteor
     :: MeteorTemplate
     -> AstroService (Either ByteString MeteorId)
  ReportAsteroid
     :: AsteroidTemplate
     -> AstroService (Either ByteString AsteroidId)
```

Very similar to Free monads but without additional fields (“next”) for the carrying of continuations. The GADT interface is clear, right? The value constructors expect an argument of some type (like `MeteorTemplate`) and have some return type encoded as an `AstroService` parametrized instance (`AstroService (Either ByteString MeteorId)`). The interpreting code is obvious or even “boring.” Just pattern match over the value constructors to produce the service implementation you need. If it was an OOP language, we could say it’s a kind of Fabric pattern:

```
-- Service creation function, analogue
--   of the Fabric pattern from OOP
getAstroServiceRunner
  :: ReportChannel -> (AstroService a -> AppL a)
getAstroServiceRunner TcpChannel  = asTcpAstroService
getAstroServiceRunner HttpChannel = asHttpAstroService

-- Specific implementations
asTcpAstroService :: AstroService a -> AppL a
asTcpAstroService (ReportMeteor m)   =
    reportMeteorTcp tcpConn m
asTcpAstroService (ReportAsteroid a) =
    reportAsteroidTcp tcpConn a

asHttpAstroService :: AstroService a -> AppL a
asHttpAstroService (ReportMeteor m) =
  reportMeteorHttp localhostAstro m
asHttpAstroService (ReportAsteroid a) =
  reportAsteroidHttp localhostAstro a
```

The main function that utilizes this GADT-based service will look exactly the same as with Free monads:

```
main = do
  ch :: ReportChannel <- getReportChannel                 #A
  let astroServiceRunner = getAstroServiceRunner ch       #B
  result <- runApp runtime                                #C
      $ consoleApp astroServiceRunner                     #D
```

```
#A Getting ReportChannel somehow
#B Getting a service implementation (runner)
#C Evaluating a normal AppL scenario
#D Configured by the service runner
```

Even the `consoleApp` remains the same:

```
consoleApp :: (forall x. AstroService x -> AppL x) -> AppL ()
consoleApp runner = do

  line <- getUserInput

  let mbParsed = tryParseCmd line
  let runners =
        [ reportWith runner ReportMeteor   mbParsed
        , reportWith runner ReportAsteroid mbParsed
        ]

  eResults <- sequence runners
  printResults eResults
```

Notice that the runner argument has complete info about the type to be parsed from the line. You may, but you don't have to, specify it explicitly via type application:

```
  let runners =
        [ reportWith runner ReportMeteor
            $ tryParseCmd @(MeteorTemplate) line
        , reportWith runner ReportAsteroid
            $ tryParseCmd @(AsteroidTemplate) line
        ]
```

This is because we already specified the value constructor as a hint (`ReportMeteor` and `ReportAsteroid`). The `reportWith` just tries to parse the line to the type associated with the GADT's current value

constructor. The report function is almost the same as in the Free monad approach, except the runner argument can be called directly:

```
-- In the Free monadic reportWith:
reportWith runner reporter (Right obj) = do
  void $ runAstroService runner $ reporter obj
  pure $ Right ()

-- In the GADT's reportWith:
reportWith runner reporter (Right obj) = do
  void $ runner $ reporter obj
  pure (Right ())
```

A very simple approach, right? Still, you should remember that in contrast to Free monads, GADT-based languages can't form sequential patterns, at least without additional mechanisms. You can probably make a tree-like structure and evaluate it, but this structure won't share the same properties as Free monadic languages do by just parametrizing a `Free` type with a domain algebra. This means GADTs are really suitable for API-like interfaces and not that convenient for scenario-like interfaces.

8.4.5 Final Tagless/mtl

Final Tagless is a bit problematic from the start. First, it's a dominant approach in Haskell (and in Scala, recently), which means it outshines other approaches in Haskell's technical folklore. Second, there's no single predefined pattern of Final Tagless. In contrast, the practices Haskellers call Final Tagless may involve more or less additional type-level magic. They say that mtl style in Haskell is a special case of Final Tagless, and they also say that Final Tagless is much more interesting than mtl. Well, I willingly believe it. But we'll consider a relatively simple mtl style here, and it will probably be enough to get a general idea of the approach.

It all starts from the Big Bang. In our case, it's a language definition. For mtl, it will be a type class with the same interface we saw previously:

```
class AstroService api where
  reportMeteor
     :: MeteorTemplate
     -> AppL (Either ByteString MeteorId)
  reportAsteroid
     :: AsteroidTemplate
     -> AppL (Either ByteString AsteroidId)
```

Except there's a tiny difference, namely, the `api` phantom type. This type will help us select the needed instance, currently one of the two. This time, we may only pass either `HttpAstroService` or `TcpAstroService`, special type selectors provided for this purpose only. We can omit constructors for these types:

```
data HttpAstroService
data TcpAstroService

instance AstroService HttpAstroService where
  reportMeteor   m = reportMeteorHttp localhostAstro m
  reportAsteroid a = reportAsteroidHttp localhostAstro a

instance AstroService TcpAstroService AppL where
  reportMeteor   m = reportMeteorTcp tcpConn m
  reportAsteroid a = reportAsteroidTcp tcpConn a
```

Notice that `AstroService` instances are quite concise, which is certainly good.

Now, the business logic code should receive a type-selector to apply a proper instance (implementation) of the service. Let's reflect this fact in the type definition of the `consoleApp` function:

```
consoleApp
  :: forall api
   . AstroService api   -- Explicitly define the api type
  => AppL ()
consoleApp = do

  line <- getUserInput

  let mbParsed = tryParseCmd line
```

```
  let runners =
        [ reportWith (reportMeteor @api)   mbParsed
        , reportWith (reportAsteroid @api) mbParsed
        ]

  eResults <- sequence runners
  printResults eResults

  consoleApp @api
```

Look on those handy type applications `@api`. Pretty cool, isn't it? Finally, the caller should just concretize what implementation it wants:

```
main = do
  ch :: ReportChannel <- getReportChannel            #A
  result <- runApp runtime $                         #B
    case ch of
      TcpChannel  -> consoleApp @(TcpAstroService)   #C
      HttpChannel -> consoleApp @(HttpAstroService)  #D
```

#A Getting ReportChannel somehow
#B Evaluating a normal AppL scenario
#C Where the scenario is configured
#D By the service implementation

Despite some implicitness here (no physical values that could represent the implementations), this seems like a nice pattern. Or not?

Okay, I cheated a little bit. It's not a classical shape of Final Tagless/mtl. Several design decisions I made with the preceding code make it not really an mtl styled code. Two key points made the approach quite usable: phantom type-selector (`api`) and a fixed monad type (`AppL`). Great news, the `AstroService` subsystem works in the same monad `AppL` and is displaced in the same layer of business logic, so it integrates with the code very well.

But in the common practice of mtl usage, this is often the case when the core effects (`Logger`, `State`, `Database`) are mixed together with the domain-related effects (`AstroService`), which is really bad. Additionally, the mtl pattern has a little bit of a different structure. Classically, there are no phantom type-selectors. The implementations are selected according to the

monad type only. The following code demonstrates what I mean. Two effects defined as type classes:

```
class MonadLogger m where
  log :: LogLevel -> Message -> m ()

class MonadAstroService m where
  reportMeteor
      :: MeteorTemplate
      -> m (Either ByteString MeteorId)
  reportAsteroid
      :: AsteroidTemplate
      -> m (Either ByteString AsteroidId)
```

A “business logic” which doesn’t know anything about the `m` type except the two effects are allowed here, in mtl style:

```
sendAsteroid
  :: (MonadLogger m, MonadAstroService m)
  => AsteroidTemplate
  -> m ()
sendAsteroid asteroid = do
  eResult <- reportAsteroid asteroid
  case eResult of
    Left _  -> log Error $ "Failed to send asteroid"
    Right _ -> log Info "Asteroid sent"
```

What are the consequences you anticipate regarding this design? At least a separate monad should be defined for each API type. Something like this (pseudocode):

```
newtype HttpAppM m a = HttpAppM { unHttpAppL :: m a }
    deriving (Functor, Applicative, Monad)
```

But not only that. There will be no separation of the business logic layer and implementation layer. The domain-related functions will know about all the internal stuff because the resulting monad is the same. In particular, it’s a widespread practice to wrap the business logic and core effects in the same `ReaderT` environment on top of the `IO` monad, as in the following:

```
newtype AppM a =
    AppM { unAppM :: ReaderT AppRuntime IO a }
    deriving (Functor, Applicative, Monad, MonadIO)
```

The `AppM` monad will be our work horse for the `sendAsteriod` function. In pseudocode:

```
sendAsteroid :: AsteroidTemplate -> AppM ()
```

Still, this seems to be an incomplete definition of the so-called Application Monad pattern. I'm not sure I presented all the details correctly here. If you're interested in learning more stuff around mtl/Final Tagless, consider the materials from this list:

> **LINK** Software Design in Haskell — Final Tagless
> https://github.com/graninas/software-design-in-haskell#Final-Tagless--mtl

But remember, the rabbit hole is very deep.

8.5 Designing web services

This book tries to be diversified in its useful information around Software Engineering and Software Design. Talking about abstract or toy examples can be dissatisfying (still, unavoidable), so can we try to go more realistic while introducing more design approaches? Web services here look like a good theme: first, the tools described earlier work nicely in this context; and second, web services are a very common task nowadays. Besides this, we've already designed a client for our imaginary service; why not fake it till we make it?

8.5.1 REST API and API types

Let's play Haskell and Servant! It's a lot like life, because Servant is a very widespread solution in Haskell for building RESTful services. Can't say it's simple — it's not, because its special type-level eDSL requires a bit of learning. But it's simple enough due to this eDSL's aim of being used by those of us who aren't proficient in advanced type-level magic. So, once you have a sample of a Servant-based service, you may rework and extend it for most scenarios. Hopefully, you don't have any uncommon requirements (like streaming, probably), because otherwise it will be very hard to grasp with Servant.

Thumb through the first sections of this chapter till listing 8.3 with a definition of the REST API for the `Astro` service. It exposes two methods: `/meteor` and `/asteroid`. These methods should be used to report the corresponding objects, and, in order to report more objects, will be required to extend this type. We could probably design a generic method for an arbitrary astronomical object, something the client and server know how to treat. For example, the following API type and REST API definition would be sufficient for many cosmic objects:

```
type AstronomicalUnit = Double

-- Orbital characteristics
data Orbital = Orbital
  { apoapsis              :: AstronomicalUnit
  , periapsis             :: AstronomicalUnit
  , epoch                 :: UTCTime
  , semiMajorAxis         :: AstronomicalUnit
  , eccentricity          :: Double
  , inclination           :: Double
  , longitude             :: Double
  , argumentOfPeriapsis   :: Double
  , orbitalPeriod         :: Double
  , avgOrbitalSpeed       :: Double
  }

-- Physical characteristics
data Physical = Physical
  { meanDiameter    :: Double
  , rotationPeriod  :: Double
  , albedo          :: Double
  }

data AstroObject = AstroObject
  { astroObjectId :: Int
  , name          :: Maybe Text
  , objectCass    :: Text
  , code          :: Text
  , orbital       :: Orbital
  , physical      :: Physical
  }
```

Okay, these types provide evidence that tracking astronomical objects is tricky because of so many parameters and variations. If we want to attach some

additional info, like the parameters of the telescope or raw data from computers, we'll have to extend the API types somehow. We should also remember the limitations, which are described in different RFCs for HTTP. It might be that passing a whole bunch of data is problematic due to its size, and, in this case, we could expose an API allowing partial data to be transferred. First, we would send this template:

```
data AstroObjectTemplate = AstroObjectTemplate
  { name        :: Maybe Text
  , objectCass  :: Text
  , code        :: Text
  }
```

We expect the server to return an identifier for this record, and we can use this identifier to provide more info about the object. Let's design a Servant API type for this situation.

Listing 8.5 Extended Server API

```
type AstroObjectId = Text
type AstroAPI =
    (  "object_template"                          #A
    :> ReqBody '[JSON] AstroObjectTemplate
    :> Post '[JSON] AstroObjectId
    )
  :<|>
    (  "object"                                   #B
    :> Capture "object_id" AstroObjectId
    :> Get '[JSON] (Maybe AstroObject)
    )
  :<|>
    ( "orbital"                                   #C
    :> Capture "object_id" AstroObjectId
    :> ReqBody '[JSON] Orbital
    :> Post '[JSON] AstroObjectId
    )
  :<|>
    ( "physical"                                  #D
    :> Capture "object_id" AstroObjectId
    :> ReqBody '[JSON] Physical
    :> Post '[JSON] AstroObjectId
    )
```

```
#A Route POST "/object_template"
#B Route GET "/object"
#C Route POST "/orbital"
#D Route POST "/physical"
```

In this code, four methods are described: three post methods for passing data about a specific object, and one to get the object by its `ID`. Notice the `Capture` clause, which mandates that the client specify the `object_id` field. Servant has a set of different modifiers for URL, for body content, for headers, for queries, and so on. Check out the idea: the type you're building with those type-level combinators will be used to generate the handlers and routes when it's interpreted, but not only that. With this declarative type-level API, you can generate a nice-looking API reference documentation, `Swagger` definitions, and even client functions, as we did previously. So, we must state that Servant is one of the most well-designed and convenient type-level eDSLs existing in Haskell. It's very clear what this magic gives us. And as it's always for good eDSLs, we don't need to get into the details of how Servant works.

However, there are still complexities in how we define the actual code of the server. Before we go deeper, we should prepare the handlers, which are the functions for processing the requests. We need the following environmental type in which the handlers will be working:

```
type Handler a = ExceptT ServerError IO a
```

The `ExceptT` monad transformer will be guarding exceptions of its type in order to convert them into errors suitable for `Servant` (`ServerError`). Also, you can see that the underlying type is just a bare `IO` because our handlers should evaluate real effects, the sequence of which we call "business logic." This is probably the typical design choice with Servant across projects. For example, a handler for the "object" request can be written in this `Handler` monad directly. From the definition of the method, it should capture the `AstroObjectId` value, so we encode it as a parameter. The return type `(Maybe AstroObject)` should match the definition as well:

```
-- Method for GET "/object":
getObject :: AstroObjectId -> Handler (Maybe AstroObject)
getObject objectId = do
  dbConn <- liftIO $ DB.connect dbConfig
  liftIO
    $ DB.query dbConn
    $ "SELECT * FROM astro_objects WHERE id == "
         ++ show objectId
```

Now the `getObject` method can handle the requests. It's not yet embedded into the server infrastructure, and even more methods should be implemented:

```
-- Method for POST "/object_template":
submitObjectTemplate
   :: AstroObjectTemplate
   -> Handler AstroObjectId

-- Method for POST "/orbital":
submitObjectOrbital
   :: AstroObjectId
   -> Orbital
   -> Handler AstroObjectId

-- Method for POST "/physical":
submitObjectPhysical
   :: AstroObjectId
   -> Physical
   -> Handler AstroObjectId
```

You might notice that the API is very clumsy. With real APIs, it's better to group the methods into namespaces. Servant allows this via a special (overloaded) type operator `(:>)`. For example, moving the updating methods into a separate namespace `/object/physical`: would require the following changes in the API definition.

Listing 8.6 Server API with nested routes

```
type AstroAPI =
    (  "object_template"                              #A
    :> ReqBody '[JSON] AstroObjectTemplate
    :> Post '[JSON] AstroObjectId
    )
  :<|>
```

```
"object" :>
(
   ( Capture "object_id" AstroObjectId          #B
   :> Get '[JSON] (Maybe AstroObject)
   )
 :<|>
   ( "orbital"                                  #C
   :> Capture "object_id" AstroObjectId
   :> ReqBody '[JSON] Orbital
   :> Post '[JSON] AstroObjectId
   )
 :<|>
   ( "physical"                                 #D
   :> Capture "object_id" AstroObjectId
   :> ReqBody '[JSON] Physical
   :> Post '[JSON] AstroObjectId
   )
)
```

#A Route POST "/object_template"
#B Route GET "/object"
#C Route POST "/object/orbital"
#D Route POST "/object/physical

This will affect the way the actual method handlers are sequenced to form a complete server definition. Check out section 8.4.3 to know more. For now, let's discuss what's wrong with placing the business logic into the `Handler` monad directly and how we can incorporate the framework into this transformation pipeline (figure 8.7).

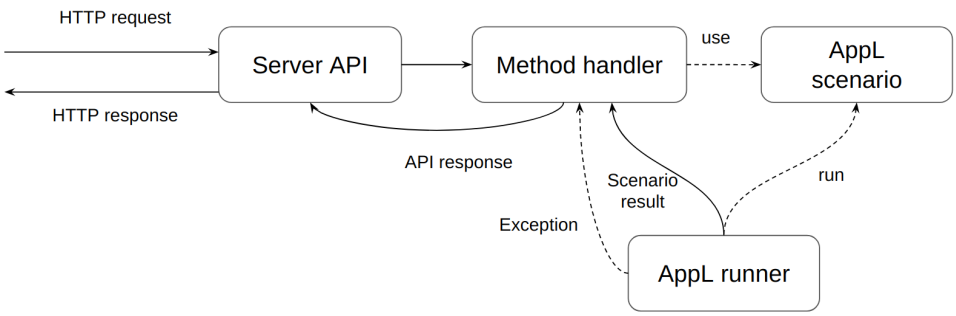


Figure 8.7 Call stack of server processing pipeline

8.5.2 Using the framework for business logic

Putting the business logic into the `Handler` monad is a straightforward approach; it has the flaws we've discussed many times: too easy to break, too hard to test, and it's imperative and impure. We created a whole framework specifically to overcome the problems of bare `IO`, and why not separate the business logic from the server-related code? Why not layer the application properly? Check out the diagram of how this should be (figure 8.8).

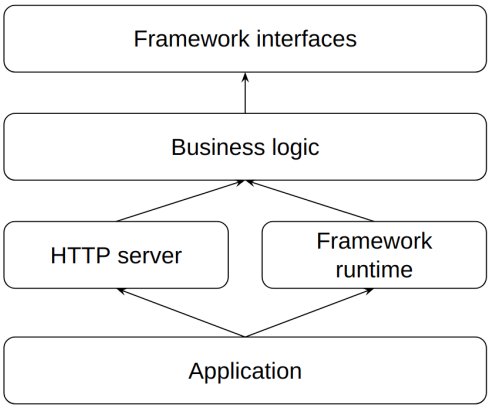


Figure 8.8 Layered web service application

We can't just call an `AppL` scenario; the `Runtime` structure should already be created. There's no point in recreating this runtime structure every time an API method is called: this will cause a high performance drop needlessly. The `Runtime` for the Free monadic `AppL` framework should be created earlier, even before we start the web server. Imagine that we've done this, and now the handler should be able to get this runtime from the outside and run the `AppL` scenario. It's better to place the `Runtime` structure into the `ReaderT` environment, so all the server methods can access it easily. The type will change:

```
type AppHandler = ReaderT AppRuntime (ExceptT ServerError IO)
```

The method should be moved to the `AppL` rails from the `Handler` ones, which should only be a bridge between the server facilities and the business

logic. Now the `getObject` method and the corresponding handler should be like this:

```
getObject' :: AstroObjectId -> AppL (Maybe AstroObject)
getObject' objectId = do
  dbConn <- getSqlDBConnection dbConf
  scenario
     $ runDB dbConn
     $ findRow
     $ ...       -- some beam query here
```

Notice the connection to the database and the framework method `getSqlDBConnection`. Let's assume for now that the database connections are established and operable, and we only need to get them via the SQL DB config. We'll talk about how to create such permanent connections later. Having shared connections helps to save extra ticks for each method, the pool makes it safe across many methods, and all seems fine. Except maybe the situation in which the connection dies. The internal pool will recreate it automatically if configured to do so, but if this whole behavior with shared SQL connections isn't appropriate for your cases, consider initializing it every time you do the query.

This is how the handler has changed. The new one:

```
getObject :: AstroObjectId -> AppHandler (Maybe AstroObject)
getObject objectId = runApp (getObject' objectId)
```

Helper function to run any `AppL` method:

```
runApp :: AppL a -> AppHandler a
runApp flow = do
  runtime <- ask                                          #A
  eRes <- lift $ lift $ try $ runAppL runtime flow        #B
  case eRes of                                            #C
    Left (err :: SomeException) -> do
      liftIO $ putStrLn $ "Exception handled: " <> show err
      throwError err500
    Right res -> pure res
```

#A Getting the runtime
#B Safely running the scenario
#C Converting the exception into the response

And if you've allowed for domain-level exceptions in your business logic, you can catch them in handlers and process them somehow. Maybe you'd want to convert those error types into the HTTP error values. Or run an additional business logic scenario. Or just ignore the erroneous result and do something else. Does this sound familiar to you? Correct: we have another example of error domains here. The division between the two domains lies through the Servant handlers. And if this is so, we shouldn't mix the errors from the business logic and the errors of the Servant library.

The latest piece of the puzzle goes further: having these handlers, how do we define the server itself? Clearly, there should be a mapping between the API type (`AstroAPI`) and the handlers (the `AppHandler` monadic functions). In order to keep the story complete, we'll also discuss how to create permanent connections and, in general, prepare the runtime for the framework within this architecture.

8.5.3 Web server with Servant

Now we have to declare one more monad transformer on top of the `Handler` type:

```
type AppServer a = ServerT AstroAPI AppHandler a
```

This monad stack is finally our full environmental type for the whole server. Yes, it's a monad, and it's aware of `AstroAPI`. It also has the `ReaderT` environment, the `ExceptT` environment, and `IO` on the bottom. Figure 8.9 reflects the monadic stack.

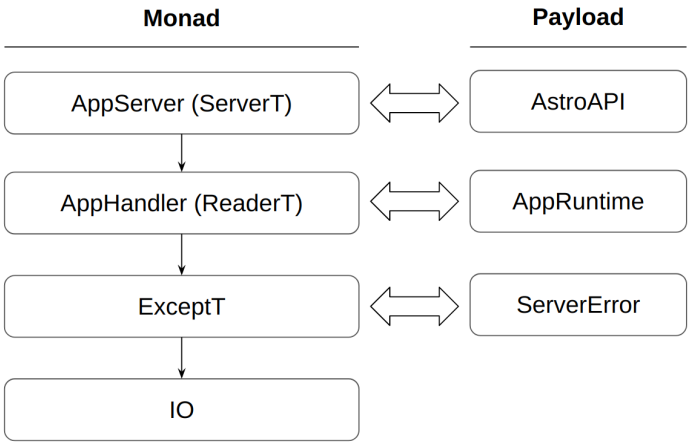


Figure 8.9 Monadic stack of the server

Now we have handlers, we have the API definition, and we have the `AppServer` type. Connecting it all together is simple: we need to repeat the shape of the `AstroAPI` type while interleaving the handlers with the `(:<|>)` operator (the shape of the API from listing 8.6 is used here):

```
astroServer' :: AppServer
astroServer' =
  submitObjectTemplate
  :<|>
    (    getObject
    :<|> submitObjectOrbital
    :<|> submitObjectPhysical
    )
```

The Servant framework mandates that we should convert our `AppServer` to the native `Server` type, which is

```
type Server layout = ServerT layout (ExceptT ServantErr IO)
```

This is because the top function for serving has the following (rather cryptic) type:

```
serve :: HasServer layout '[]
      => Proxy layout -> Server layout -> Application
```

where

- Proxy layout is a proxy of the AstroAPI:

```
astroAPI :: Proxy AstroAPI
astroAPI = Proxy
```

- Server layout is the server handlers definition similar to `astroServer'` but within the `Server` monad stack rather than in our own `AppServer`.
- `Application` is a type from the separate library to work with the network - `wai`.

> **NOTE** Beware type-level witchcraft! This is a type-level list: `'[]`. It should be filled by some types, but for now it's empty. That type constraint, `HasServer layout '[]`, is very much like a wardrobe. Open it, and you'll get into a magical land of miracles.

Now let's convert the `astroServer'` function from the `AppServer` type to the `Server` type. A special `hoistServer` function from Servant will help in this:

```
astroServer :: AppRuntime -> Server AstroAPI
astroServer appRt = hoistServer astroAPI f astroServer'
  where
    f :: ReaderT AppRuntime (ExceptT ServerError IO) a
      -> Handler a
    f r = do
      eResult <- liftIO $ runExceptT $ runReaderT r appRt
      case eResult of
        Left err  -> throwError err
        Right res -> pure res
```

So many preparations! We transform But this isn't the end. The `Server` type should be transformed to the `Application` type suitable for the `wai` library to be run. The `serve` function is, by the way,

```
astroBackendApp :: AppRuntime -> Application
astroBackendApp appRt = serve astroAPI $ astroServer appRt
```

Finally, the `run` function from the `wai` library should be used to make things real, like this:

```
main = do
  appRt <- createRuntime
  run 8080 $ astroBackendApp appRt
```

That was a long path just to define and start the server. You can use this knowledge for your backends now, but some questions remain unanswered. Namely, how would we prepare the `AppRuntime` and the permanent connections to the databases, to the external services, and so on. Where's a good place for it? Right here, before we call the `run` function. Let's investigate the following functions step-by-step:

`withAppRuntime` safely creates the `AppRuntime` structure to be used in the framework. It also handles the exceptions and gracefully destroys the environment.

```
runAstroServer :: IO ()
runAstroServer =
  withAppRuntime loggerCfg $ \appRt -> do
    runAppL appRt prepareDB                          #A
    appSt <- runAppL appRt prepareAppState           #B
    run 8080 $ astroBackendApp $ Env appRt appSt     #C

#A Preparing the permanent connections,
     making migrations, initializing DBs
#B Possibly, initializing state for the application
#C Starting the servant server
```

Notice several changes here. The environment type is now not just `AppRuntime` but rather something called `Env`, which carries the `AppRuntime` and the `AppState`. The latter is similar to the state we discussed in chapter 6. Check this out:

```
data AppState = AppState
  { _astroObjectCount :: StateVar Int
  }

data Env = Env AppRuntime AppState
```

The `prepareDB` function is more interesting. It can do many things you need: open permanent connections (which will be kept inside the `AppRuntime` as usual), make migrations, and evaluate premature queries to the databases or external services. It's your choice whether to write this separate business logic with the framework or make it `IO` only. In the previous case, both the functions `prepareDB` and `prepareAppState` are of the `AppL` type:

```
prepareDB :: AppL ()

prepareAppState :: AppL AppState
```

So, with this design, you can not only process the particular API queries with the Free monadic framework, but you can also use it more wisely for infrastructure tasks as well. Why? Because it makes the code much more modular and simple due to the pure interfaces of the framework's eDSLs.

8.5.4 Validation

The World Wide Web is a truly communist foundation in which everyone can obtain the resources he needs, and all the communities around the globe are united in helping each other to build a bright future together. What can go wrong in this world of true brightness of mankind!

Well, unfortunately, these are only dreams.

In reality, we developers of web services can't trust the data coming from the network. It's not a rare case in which some intruder is trying to scam the service by sending invalid requests in order to find security bugs. A golden rule of all services declares that we should protect from these attacks by validating incoming requests. This not only provides protection from the occasional breakage of the service but also a sanity check about whether all cases are considered, and whether the API is consistent and robust. So, all the incoming data should be checked for validity, and the logic should operate with the correct

values only. Otherwise, how should the logic behave? Garbage in values leads to garbage in calculations.

From a design point of view, this means we can distinguish the two models: the API model, whose values we're expecting from the outside, and the domain model, which is the properly justified data types transformed from the API model. In other words, the process shown in figure 8.10 will be the best for API services from an architectural perspective.

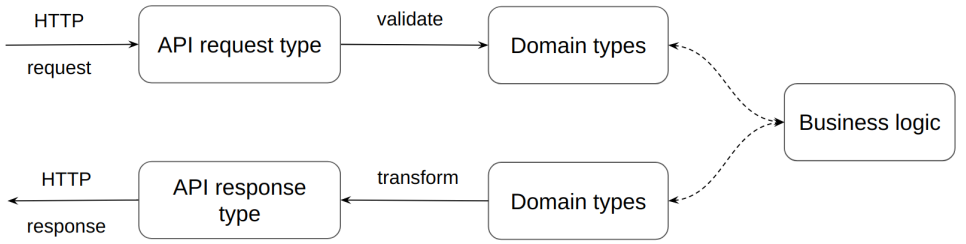


Figure 8.10 API and domain types

In Haskell, there are several interesting approaches to validation. It's known that validation can be applicative, meaning the process of validation can be represented as an expressional calculation over user-defined types. The applicative validation approach also helps to simplify the wording with the handy combinators the `Applicative` instance provides. But first things first.

Before we jump into applicative validation, we'd better see how to validate data in the simplest way possible. Let's investigate the `submitObjectTemplate` method, which receives the following API type as a request body:

```
data AstroObjectTemplate = AstroObjectTemplate
  { name        :: Maybe Text
  , objectClass :: Text
  , code        :: Text
  }
```

A poor man's validation is quite simple. Just do a straightforward check of values and react to the wrong ones somehow. In our case, we can do this in the

handler, and once we get some invalid value, we immediately return the `400` error ("Bad Request"). A possible list of errors:

```
err1 = err400
       { errBody = "Name should not be empty if specified."}
err2 = err400
       { errBody = "Object class should not be empty."}
err3 = err400
       {errBody = "Object code should not be empty."}
```

Suppose, we'd like the two fields to be non-empty (`objectClass` and code) and the name to be non-empty if and only if it's not `Nothing`. With a little help from the `MultiwayIf` and `RecordWildCards` extensions, we can write a handler like this:

```
submitObjectTemplate
  :: AstroObjectTemplate -> AppHandler AstroObjectId
submitObjectTemplate template@(AstroObjectTemplate {..}) = do

  if | name == Just ""   -> throwError err1
     | objectClass == "" -> throwError err2
     | code == ""        -> throwError err3
     | otherwise         -> pure ()

  -- Structure is valid, do something with it.
  -- For example, call the AppL scenario:
  runApp $ createObjectTemplate template

createObjectTemplate
    :: AstroObjectTemplate -> AppL AstroObjectId
createObjectTemplate template = error "Not implemented yet"
```

Here, we use `throwError` from the `ExceptT` monad because the `AppHandler` type contains the `ExceptT` transformer, and Servant will make a proper response from it.

Alternatively, if you need to report something when a wrong value is found, it's a good idea to perform validation within the `AppL` scenario and call the methods from the framework. In this case, validation becomes a part of the business logic, kind of. We can use the methods of the framework for logging errors and throwing exceptions, like this:

```
  let failWith err = do
        logError $ show err
        scenario $ throwException err

  if | name == Just ""   -> failWith err1
     | objectClass == "" -> failWith err2
     | code == ""        -> failWith err3
     | otherwise         -> pure ()
```

There's one obvious issue with such an approach. The very first erroneous field will breach the evaluation, which leaves the rest of the fields unchecked. For some APIs, this is probably acceptable, and the client will have to pop out the errors one by one, making several calls to the server. But often it's better to decrease the number of API calls, no matter whether it's for performance reasons or for the convenience of clients. This means we'd better try some other approach to validation. Maybe invent a mechanism that allows us to validate all the fields in turn and collect all the errors from the structure. Then we can send this list back to the client somehow.

This is actually a big theme. The task of validation suits functional programming very well, this is why many different combinatorial approaches have emerged in the Haskell community. The most interesting of them focuses on Applicative-style validation. You may find several libraries that do this; but my own library, `pointed-validation` has an interesting ability to point to the specific place in the validated, possibly hierarchical structure where the error has been found. In other words, you may get not only a message that "some internal field is invalid" but also bread crumbs to it.

> **LINK** The `pointed-validation` library:
> https://github.com/graninas/pointed-validation

For example, consider the two following data structures organized hierarchically:

```
data Inner = Inner
  { _intField   :: Int
  }
data Outer = Outer
  { _stringField :: String
  , _innerField   :: Inner
  }
```

There are also requirements for these fields:

- `_intField` should be > `0` and less `100`.
- `_stringField` should not be empty.

If the `Inner` value is malformed, and the `_stringField` is malformed, then we can expect the following result of validation:

```
[ ValidationError { path = ["innerField",intField"],
   errorMessage = "Inner intField: should be > 0"}
, ValidationError { path = ["stringField"],
   errorMessage = "Outer stringField: should not be empty"}
]
```

Here `path` is a list of bread crumbs to the specific place deeply nested in the whole structure.

Short story long, the `pointed-validation` library has several Applicative-style combinators to define validators in a convenient way. Check out this validator for the case just described:

```
innerValidator :: Validator Inner
innerValidator = validator $ \inner -> Inner
  <$> (inner ^. intField'                              #A
         &  condition (> 0)    err1
         &. condition (< 100)) err2
  where
    err1 = "Inner intField: should be > 0"
    err2 = "Inner intField: should be < 100"

outerValidator :: Validator Outer
outerValidator = validator $ \outer -> Outer
  <$> (outer ^. stringField'                           #B
         &  condition (not . null) err)
  <*> (nested outer innerField' innerValidator)
  where
    err = "Outer stringField: should not be empty"

#A intField' is a pointed getter
#B stringField' is a pointed getter
```

These validators represent the most idiomatic functional code: they are pure declarations, and they are combinable in the truest sense. Notice how we nest the `innerValidator` into the `outerValidator` using the `nested` combinator from the library. This means we can compose more validators from the existing ones. We can stack several validators for a single field. We use Applicative style to compose different validators. With the help of the `fmap (<$>)` and `ap (<*>)` operators, we "visit" all the fields and can't skip anything.

And when we described how a data type should be validated, we can apply these validators using the `applyValidator` function:

```
main = do
  let value = Outer "" (Inner 0)
  case applyValidator outerValidator value of
     SuccessResult _      -> putStrLn "Valid."
     ErrorResult _ errors -> print errors
```

There's something called a "pointed getter." Pointed getters are a key to how the library forms the bread crumbs. Essentially, pointed getters are normal lens getters, except they return not only a value that they point to but also a path to this value. For instance, the `intField'` pointed getter looks like so:

```
intField' :: HasIntField a Int => Getter a (Path, Int)
intField' = mkPointedGetter "intField" ["intField"] intField
```

You can create such a getter manually, although the library provides several Template Haskell functions to auto-generate them.

We won't dive too deep into the theme of structural data validation. You can find many different articles on this topic. We'll also skip the question of an effectful validation, which is when you have to make some side effect before the field is considered valid. A common use case here is when you need to query a database to check the uniqueness of the data passed. All these things are very naturally implementable with functional programming, so different interesting approaches exist out there. A lot of stuff for curious people! But we have to move on.

8.5.5 Configuration management

This section highlights the most important topics for web services construction. I should say a few words about configuration management, though.

Every real-world application requires some customization. The following is a list of options that comes to mind immediately:

- Logger config, such as this:
  - log file path
  - logging severity
  - credentials of external logging services
  - log entry format
- Database credentials
- Your application's HTTP server config
- External services configuration, such as
  - Internet address
  - credentials
  - runner paths

Many other configs can be required, no need to enumerate them all. The channel of configs obtained can vary:

- Application's command-line options
- Environment variables
- Config file — you know, there are tons of file formats:
  - JSON
  - YAML
  - INI
  - Others
- External config management systems

This is not an advanced theme, although it can be overcomplicated by choosing some heavy libraries for config management. I won't speak about this more. If you're interested in ready patterns of configs parsing, consider the astro project, which demonstrates the `optparse-applicative` library usage.

LINK Command-line options in the astro application
https://github.com/graninas/Hydra/.../astro/src/Astro/ConsoleOptions.hs

8.6 Summary

It might be that many new design patterns are still hidden in the deep recesses of functional programming. With time, these patterns will be found and developed. Our task is to prepare the ground for such knowledge because when separated, facts aren't that helpful compared to a complete methodology with all the answers to main questions.

This chapter taught us many tricks and tips of functional design, which have a great power of complexity reduction. We learned several different ways to do Dependency Injection and services:

- Free monads
- Final Tagless
- GADTs (for API-like interfaces)
- Service Handle Pattern
- ReaderT pattern

These patterns should be used when you want to make the code less coupled, more controlled, and high level. We've seen examples of business logic being based on different approaches, and it's now possible to compare them by several criteria:

- Simplicity
- Boilerplate
- Convenience
- Involvement of advanced language features

The following opinionated table tries to summarize the observations (table 8.1).

Table 8.1 Comparison of approaches

	Simplicity	**Boilerplate**	**Convenience**	**Advanced language features**
Free monads	Simple	Some	Not for this task	Nothing special
Final Tagless	Simple	Some	Probably not	A bit of type-level
GADTs	Very simple	Almost none	Very convenient	Nothing special
Service Handle	Very simple	Almost none	Very convenient	Nothing special
ReaderT	Very simple	Much (due to lifting)	Probably not	Nothing special

But be warned, this table only aims to be a starting point for further reasoning and cannot be the final truth. There are different cases and different situations, and our responsibility as software engineers is to make the appropriate decisions and not lose common sense.

Chapter 9
Testing

This chapter covers

- How to do functional, unit, and integration testing in Haskell
- What property testing is and how it's applicable to common tasks
- How to do mocking for whitebox testing
- How to completely automate whitebox test creation with Free monads

Testing is the last theme we need to discuss to complete the methodology developed in this book — last, but not the least. The software engineering discipline has long described various testing techniques, but while this knowledge is considered mandatory, we can disappointedly state that many code bases aren't well-tested. Earlier, we named testability as one of the main requirements for a good architecture. Given that we're software architects, we cannot refuse to learn testing theory and practice.

Testing is a part of a development process. Every professional developer writes tests, irrespective of how many things are expressed by the type-level magic. Tests serve different purposes: fixating the contracts, correctness verification, prevention of regressions, and documentation. Any software architect has to think ahead about how to reduce project risks, and testing is one of the most important ways to do this. Given that we're software architects, we don't have a right to be ignorant about testing theory and practice.

And the first thing we should know is that, in contrast to a widespread belief, tests are not intended to verify the correctness of the code. Tests are intended to validate that the code satisfies the requirements (functional and non-functional ones). Because the correctness is just one of them, and it can be more or less important depending on the project. So when we test our code, we usually say that we validate the contracts and expectations rather than search for bugs. Sometimes we hunt bugs with tests, though.

Testing is a big topic — it deserves its own book. Moreover, numerous such books exist. In this chapter, we'll touch on the theme slightly, and we'll see why a Free monadic architecture is uniquely testable compared to other approaches.

> **TIP** The following Wikipedia page on software testing has a very good and broad summary of the topic that could be useful for you to peruse before reading this chapter:
>
> **LINK** Wikipedia: *Software testing*
> https://en.wikipedia.org/wiki/Software_testing

We'll also investigate a small subset of testing approaches and talk about what makes an architecture testable.

9.1 Testing in functional programming

We've been told that functional programming shines in several areas:

- Multithreading and concurrency
- DSLs
- Refactoring (we're still talking about functional languages with static type systems, right?)
- Expressiveness (we're still talking about functional languages with rich and powerful syntax, correct?)

But what about testing? Can we say that functional programs have better testability? Does functional programming offer something besides traditional approaches to testing? And how can mainstream practices be applicable here?

Yes.

Yes ... what?

Functional programs are more testable than programs in any other paradigm. We know this because we have two key points in functional programming that make our code predictable and deterministic: immutability and purity. We've talked a lot about determinism, and it seems all testing approaches have a single idea in common: they pretend that once a code is deterministic, evaluating it in a test environment will show a similar behavior as evaluating it in a real environment. This, of course, isn't completely true, and tests cannot save the code from 100 percent of unwanted problems. But avoiding test practices won't save it either.

So, what's a general strategy for writing tests? It's always the same, irrespective of what paradigm we use. The strategy is to reduce risks for the project, and different types of testing reduce specific risks. Let's enumerate a few different testing approaches:

- *Black box testing*. The code is treated as a black box where we don't know much about its contents. Black box testing uses public interfaces to interact with this box and see what results it produces.
- *Grey or white box testing*. The testing code can be partially (grey box) or completely (white box) aware of how the program's code is done. The tests can use this knowledge to cover some subtle and specific cases that wouldn't be immediately obvious from public interfaces.
- *Functional testing*. We can understand this in two ways: 1) testing of functional requirements in general and 2) testing of a separate functionality in the form of a subsystem, module, function, and so on. Don't confuse its name with "functional programming" — the latter is about programming with functions, while the former is about testing functionality. In functional testing, we won't be calling real or external services. We want to make these tests more or less independent from the environment and thus more or less reliable. Functional testing can be black box or grey box testing.
- *Integration testing*. This testing also covers functional requirements, but with this approach, you test integrated application parts rather than separate pieces of logic. The goal of this kind of testing is to reveal inappropriate behavior in a big part of the application, especially its edges between subsystems. Integration tests can be (and often are) unreliable because they involve real subsystems and services. Integration testing is

black box.

- *Property-based testing* can be viewed as a part of functional testing with a particular technique applied — feeding the code with some pregenerated, possibly random data in order to establish and check its invariants. It's usually black box testing.
- *Unit testing*. In OOP, this is quite a vague approach because it's not clear what to consider a unit — a class, a method, a module, or something else. But the general consensus is that this testing should focus on the smallest possible pieces of the code. However, it's not often possible to separate these pieces easily, which is where the principles of IoC and Dependency Injection start playing a major role. By injecting mocked, test-only dependencies, we can isolate the unit code from the outside world and investigate only its properties. This makes unit testing a kind of white box testing. In functional programming, unit testing becomes more like functional testing if we agree that the smallest piece of the code is a pure function. Due to its purity, it can be made of other pure functions, so it's very deterministic and thus suitable for unit testing. It's not necessary to know the exact contents and internal structure of a pure function in this case. But once we're talking about impure functions, we have to deal with the same problems: dependency from the outer world, side effects, and unpredictability. So, testing an impure functional programming code in this way also requires IoC, Dependency Injection, and mocking.
- *Acceptance testing* happens when one runs and assesses a code as a finished product. I mention this one in the chapter only slightly because it doesn't really have anything to do with software design.

Many other testing techniques can be helpful here and there. Knowing about their existence and purpose seems a good idea for software architects, even if these approaches aren't related to the code directly. Testing may look like a simple area, but it's really not. Arranging a proper quality assurance process requires a lot of resources and an understanding of project and business goals, and it can't be made blindly. I just want you to keep this in mind.

9.1.1 Test-driven development in functional programming

How far can you go with TDD in functional programming? Well, it really depends on the architecture and design of your applications. Although TDD is

kind of controversial (there are strong arguments why a pure TDD leads to a bad design), writing tests to accompany development is still reasonable. I can propose the following workflow:

- Develop initial Free monadic framework or eDSL.
- Develop a small, demo scenario to prove that your design is okay.
- Choose to create a real interpreter or skip this step.
- Proceed with a testing infrastructure and add a simple testing interpreter.
- Write some simple functional tests.
- Ensure again that your eDSLs are okay because tests magically help find flaws in the design.
- Repeat.

Delaying the creation of a testing infrastructure may considerably harm the project in the mid- and long-term. In my personal experience, a little slowdown caused by rolling into tests early is worth every penny, even if the tests you've developed are for a toy scenario. It seems acceptable to have this for at least every Free monad method. At least. Functional testing of a Free monad framework isn't hard. It might even be less difficult than property-based testing, despite the frequent need to mock effects and calls. But mocking is really simple, as we'll see in the next sections; in contrast, establishing property-based testing can be a bit difficult due to the need to understand what properties your code should have. If we're talking about a code base for a web service that we expect to be big, it won't have that much algorithmic code — especially at the start. More likely, you'll have a set of database calls, or maybe some external REST API calls, that aren't really suitable for property-based testing. So, going TDD with property-based testing is at least questionable compared to functional or even integration testing.

I would like to know what people think about TDD for functional languages in terms of its impact on application design. This is another theme to research. Seems like someone should take a classical book on TDD written by Kent Beck and try to port its ideas to Haskell. Why? Just because we have whole books on TDD applied to different languages — Python, C, C#, Ruby, Java — but not Haskell. If we assume our language is suitable for real-world development, we

should cover more topics. It's essential to describe these topics — when they're missing, we can't say that the language is good enough.

> **LINK** Kent Beck, *Test Driven Development*
> https://www.amazon.com/Test-Driven-Development-Kent-Beck

9.1.2 Testing basics

We can mean two different things when talking about testing frameworks:

- A library that helps to organize tests, test cases, and test suites. You might find many such libraries for different languages: `hspec`, `hunit`, `tasty` (Haskell), `GTest/GMock` (C++), `NUnit` (C#), and others. More often, these frameworks are only about the testing of functionality, but there are tools for performance testing and benchmarks (see criterion in Haskell).
- A special runtime and implementation shipped with a development framework to help you better test your applications. The options may vary: mocking support, concurrent code step-by-step testing, mixing integration and functional testing, configs, and so on. We'll build one for Hydra, just to see how it can be.

As a base tool, we'll choose `hspec`. This general-purpose testing framework doesn't differ from others of its kind. Hspec supports all the features you'd expect to see:

- Test suites and test case distinction
- Ability to prepare an environment for each test case
- Assertion functions
- Integration with other testing frameworks (`QuickCheck`, `SmallCheck`, `HUnit`, and so on)
- Parallel test evaluation
- Automatic test suite discovery

Let's write some tests on a pure function. It will be a "spec" definition with a test case inside:

```
square :: Integer -> Integer
square x = x * x

spec :: Spec
spec = do
  describe "square algorithm tests" $ do
    it "squaring positive integer test" $ do
      square 2 `shouldBe` 4
      square 5 `shouldBe` 25
    it "squaring negative integer test" $ do
      square (-2) `shouldBe` 4
      square (-5) `shouldBe` 25
    it "squaring zero test" $
      square 0 `shouldBe` 0
```

As simple as that. We checked a reasonable number of cases, including one edge case with `0`. And we all know that pure functions won't produce unexpected results (if written correctly, without hacks like `unsafePerformIO`). Determinism, remember?

You might have heard about "triple A." AAA stands for Arrange, Act, Assert.

- *Arrange*. You prepare your test data.
- *Act*. You perform the action you want to test with this data.
- *Assert*. You check the result.

This practice teaches us not to bring more into tests than is necessary. A test should have only one responsibility, only one act, and should be independent from other tests. If independence is broken, the test results may be wrong. The following test case contains two tests with an unwanted dependency. Due to this, parallel runs of tests will fail:

```
impureAction :: IORef Int -> (Int -> Int) -> IO Int
impureAction ref f = do
   val <- readIORef ref
   pure (f val)

spec :: Spec
```

```
spec = do
  describe "IORef tests" $ do
    ref <- newIORef 0

    it "test 1" $ do
      writeIORef ref 10                       -- Arrange
      result <- impureAction ref (*10)        -- Act
      result `shouldBe` 100                   -- Assert

    it "test 2" $ do
      writeIORef ref 5
      result <- impureAction ref (+10)
      result `shouldBe` 15
```

Being run in parallel, these cases will fall into a race condition. The result depends on what writing operation is evaluated first before any reading has happened. In general, code with effects should be tested in isolation, without interleaving with anything outside. This effectively means that the code should be written with a good separation of concerns, using all sorts of tricks to control the effects. Furthermore, we'll see that Free monadic languages can be very convenient and testable in this sense.

9.1.3 Property-based testing

Briefly speaking, this kind of functional testing enjoys its popularity in functional languages. It works best when you have a small chunk of some purely algorithmic code. Pure algorithms can always be seen as a transformation of data. Each pure algorithm has some inherent properties, or better yet, *invariants*. All you need is to define what invariants should be held, and then you can encode them as properties in tests. These properties will be challenged by random values and verified for whether they actually hold. If not, there's a value that ruined the property. A testing framework should do its best to shrink this value to the bare minimum because you'll want to get a minimal viable counterexample for your logic.

Property-based testing rocks when you have small, pure algorithms and input parameters that aren't that wide. You rarely want to check all the cases by hand because this may result in dozens if not hundreds of distinct inputs. But with property-based testing, these cases can be generated automatically.

For example, we want to test a squaring function. Suppose it was written awfully for some reason:

```
malformedSquare :: Integer -> Integer
malformedSquare 2 = 4
malformedSquare 5 = 25
malformedSquare 10 = 100
malformedSquare x = 0
```

Clearly, the property of being squared does not hold for many integers. The algorithm of squaring is extremely simple. We can check its validity quickly. Hence QuickCheck — a library for property-based testing. It can be used together with hspec. See this code snippet:

```
spec :: Spec
spec = prop "square"
    -- xs will be generated:
   $ \x -> (malformedSquare x == x * x)

-- Test output:
> Falsified (after 2 tests): 1
```

This test ensures that the function called with any pregenerated `x` produces a result satisfying the boolean condition. The test works not only for several specific cases but for a whole class of `Integer` values … well, not really. QuickCheck will generate a number of distinct cases around zero. The idea relies on the hypothesis that most problems can be spotted just by trying the initial set of values that are reasonably “small” (close to `0`, or lying on the edges of a type range). However, the test can’t guarantee the function to be free of subtle logical bugs. The following `malformedSquare` function will be completely correct according to a property-based test because QuickCheck won’t generate values greater than `100`:

```
malformedSquare :: Integer -> Integer
malformedSquare x | x < 1000000 = x * x
malformedSquare _ = 0

-- Test output:
> +++ OK, passed 100 tests.
```

You can increase the generation threshold. In QuickCheck, the `withMaxSuccess` combinator controls the number of successful tests needed to admit that all is okay, but it's certainly impossible to traverse the whole line of infinite integer values.

In reality, simple algorithms don't occur that often, and the probability of implementing them incorrectly is very low — much lower than the effort we'd spend to test this code. Focusing on such small functions doesn't seem that valuable. Hopefully, we can sometimes use property-based testing for big chunks of pure logic, even if this logic is written using a Free monadic framework. Let's talk about this in more detail, as this is not a trivial question.

9.1.4 Property-based testing of a Free monadic scenario

The Hydra project has the "Labyrinth" showcase application. This is a game in which you explore a labyrinth. Your goal is to find a treasure and leave the labyrinth through an exit. Initially, the labyrinth is completely hidden, and you have to rebuild its structure by walking around. It's a lightweight, roguelike game with the main idea preserved: the labyrinth should be randomly generated. Let's see how we can test this function for correctness.

There are two generating functions: one that doesn't require any labyrinth parameters and another one that takes several arguments — bounds, and the number of exits and wormholes (portals). The functions are short, so let me show them both:

```
type Bounds = (Int, Int)

-- | Generating a labyrinth of random size,
--   with a random number of exits and wormholes.
generateRndLabyrinth :: LangL Labyrinth
generateRndLabyrinth = do
  xSize     <- getRandomInt (4, 10)
  ySize     <- getRandomInt (4, 10)
  exits     <- getRandomInt (1, 4)
  wormholes <- getRandomInt (2, 5)
  generateLabyrinth (xSize, ySize) exits wormholes

-- | Generating a labyrinth with predefined parameters.
```

```
generateLabyrinth :: Bounds -> Int -> Int -> LangL Labyrinth
generateLabyrinth bounds exits wormholes =
  generateGrid bounds
    >>= generatePaths bounds
    >>= generateExits bounds exits
    >>= generateWormholes bounds wormholes
    >>= generateTreasure bounds
```

As you can see, the generation sequence is pretty simple and quite readable. The internals of these two functions use only three effects, three methods: `getRandomInt`, `evalIO` (for a local mutable state with `IORefs`), and `throwException`. Although there are many other effects "embedded" in the `LangL` monad, it's unlikely that they'll be used in the generation. No database interaction, no logging, no threads; only algorithms with a bit of randomness. In other words, it's a good candidate for effective property-based testing — which is supported by the QuickCheck library as well.

As I said, it will be a little bit tricky because we need to run a `LangL` scenario with real interpreters within a QuickCheck setting. In turn, real interpreters use the `AppRuntime` structure, which should be pre-created before the test. We'll need the following bits:

Runtime creation function:

```
withCoreRuntime :: (CoreRuntime -> IO a) -> IO a
```

Interpreting process runner:

```
runLangL :: CoreRuntime -> LangL a -> IO a
```

Hspec method which allows evaluating a setup action before each test case. Will be used for the CoreRuntime pre-creation:

```
around :: (ActionWith a -> IO ()) -> SpecWith a -> Spec
```

QuickCheck method to run a property described as an IO action:

```
monadicIO :: Testable a => PropertyM IO a -> Property
```

QuickCheck method to configure how many successful test runs are expected:

```
withMaxSuccess :: Testable prop => Int -> prop -> Property
```

QuickCheck method to run an IO method:

```
run :: Monad m => m a -> PropertyM m a
```

Consult with this cheatsheet when you're examining a complete solution (see listing 9.1).

Listing 9.1 Property-based testing of a Free monadic scenario

```
spec :: Spec
spec = around withCoreRuntime
  $ describe "generation tests"
  $ it "generated labyrinth has correct bounds"
  $ \runtime -> property
    $ withMaxSuccess 5
      $ monadicIO $ do

        eLab <- run $ try
                    $ runLangL runtime generateRndLabyrinth

        case eLab of
          -- Treating negative cases as failure
          Left (err :: SomeException) -> assert False

          -- Checking bounds
          Right lab -> do
            let (x, y) = getBounds lab
            assert $ x * y >= 16 && x * y <= 100
```

Here, we don't actually use QuickCheck as it's intended. The test doesn't generate any arbitrary data to check the properties; it just calls the `generateRndLabyrinth` five times and verifies the conditions that we assume should be true. Only one property is checked there: whether the size of the labyrinth fits to the specified range. But can we properly check this property? QuickCheck provides a `pick` combinator to ask for an arbitrary value, and the `pre` combinator to cut off those arbitraries that shouldn't go into the tested logic. We could write the following test case for the customizable `generateLabyrinth` function:

```
...
$ monadicIO $ do
   x :: Int <- pick arbitrary
   y :: Int <- pick arbitrary

   -- condition to cut off some values
   pre $ x >= 4 && x <= 10 && y >= 4 && y <= 10

   let action = generateLabyrinth (x, y) 3 3

   eLab <- run $ try $ runLangL coreRuntime action

   case eLab of
     Left (err :: SomeException) -> assert False
     Right lab -> do
       let (x, y) = getBounds lab
       assert $ x * y >= 16 && x * y <= 100
```

However, this test will more than likely fail. Why? Formally, the test is correct, but QuickCheck will generate too few appropriate arbitraries, so after 50 tries, it will give up on satisfying the `pre` condition. Think about this: the `pre` condition will be a cause of arbitrary values rejection, not the method itself! For example, the following values will be generated:

```
(0,0)  (-1,1)  (2,-2)  (-2,0)   (-1,3)  (-3,3)
(1,1)  (1,-1)  (0,0)   (-1,-2)  (3,-3)  (3,0)
(0,1)  (1,-1)  (1,1)   (0,-1)   (-2,-1) (2,2)
(0,1)  (-1,0)  (1,-2)  (-2,2)   (1,-1)  (-2,-2)
(1,0)  (0,-1)  (-2,2)  (0,0)    (3,0)   (-1,0)
```

The first useful pair `(6,4)` will occur only on the 84th turn! Given this, the actual logic won't be called as much as needed, and the whole test case will fail due to an insufficiency of successful outcomes. Of course, this is all tweakable, and we could make it work better. To be precise, we could

- Generate only natural numbers by specifying the `Word8` type instead of the `Int` type.
- Remove condition, pass all the generated values into the `generateLabyrinth` function.
- Not treat the negative cases as failures but consider them a part of the

property to be checked.

- Customize the depth of arbitrary values generation.
- Customize the number of test runs.
- Customize the values generated (by providing a better `Arbitrary` instance).
- Ensure that enough "good" meaningful pairs are challenging the `generateLabyrinth` function to check its actual behavior.

After these changes, we would get a more or less useful property test. However, I doubt it's worth it. Too much effort will have been traded for too little profit. We enjoyed the fact that the labyrinth generation functions are algorithmic, and they're kind of sealed, meaning they don't interact with databases or other external services. But this isn't usually true for a web application's regular business logic. Let's agree that property-based testing is best suited for testing small, pure functions but not for something more effects-involving.

9.1.5 Integration testing

Since we discussed the limitations of property-based testing, we should talk about integration testing. The latter seems to be the most important for checking the "sanity" of the code because its environment resembles the production environment as much as possible. While this is an advantage, it's also a disadvantage because real environments tend to be unreliable, slow, and hard to set up. But once you get your integration tests up and running, your confidence in the code will rise to the sky — assuming that you wrote enough good tests, of course. So, integration testing is needed to assess the code's functional requirements, its interaction with other services, and its behavior on the edges between different subsystems.

Let's start from testing our previous function, `generateLabyrinth`. We don't need to implement anything special on the testing side. We can proceed with the real interpreters for this simple scenario. In fact, the test will be almost the same as we saw in the previous section. Check it out:

```
spec :: Spec
spec = around (withCoreRuntime Nothing)
  describe "Labyrinth generation tests"
    $ it "generateLabyrinth" $ \runtime -> do

      let action = generateLabyrinth (4, 4) 3 5
      lab <- runLangL runtime action
      let (bounds, wormholes, exits) = analyzeLabyrinth lab

      bounds `shouldBe` (4, 4)
      (Map.size wormholes)
         `shouldSatisfy` (\x -> x >= 2 && x <= 5)
      (Set.size exits)
         `shouldSatisfy` (\x -> x >= 1 && x <= 3)
```

Yes, I know, there's nothing new or interesting here. Integration tests of a Free monadic scenario use a real interpreter, so we can't test all the code. For example, the following logic for loading a labyrinth from a RocksDB catalogue explicitly points to a "production" database (listing 9.2).

Listing 9.2 Scenario for loading a labyrinth from KV DB

```
-- LabVal and LabKey are a part of the KV DB model.

loadLabyrinth :: Int -> LangL Labyrinth
loadLabyrinth labIdx =
  withKVDB' kvDBConfig $ do
    labVal :: LabVal <- loadEntity (LabKey labIdx)
    pure $ fromLabVal labVal

-- "Production" database of type LabKVDB
kvDBConfig :: KVDBConfig LabKVDB
kvDBConfig = RocksDBConfig "./prod/labyrinths"
```

> **NOTE** For the sake of clarity, the KV DB related code here differs slightly from the code in Hydra. In particular, the `loadEntity` function returns not just a raw value type but rather a value type wrapped in the `DBResult` type. Before the value can be used, the result should be checked for an error. See chapter 7 for details.

Testing this function against a real environment immediately leads to many possible problems, such as data corruption and security breaches. We can't be

sure that the `loadKVDBLabyrinth` is written correctly, so the test may be occasionally harmful before it starts bringing any safety.

A common practice here can be separating the environments and making the code suitable for a development environment. This usually means that all the parameters and configs should be passed from outside so that the integration tests can be configured to use a specially formed database, not a production one. The loading function with an extracted config is as follows:

```
loadLabyrinth :: DBConfig LabKVDB -> AppL Labyrinth
loadLabyrinth kvDBConfig =
  withKVDB' kvDBConfig $ do
    ...
```

The test can now be configured properly; just define a special database for it and ship both together. Let's write it:

```
testDbCfg :: DBConfig LabKVDB
testDbCfg = RocksDBConfig "./labyrinths.rdb"


...
  it "Load labyrinth from RocksDB test" $ \coreRuntime -> do
      lab <- runLangL coreRuntime $ loadLabyrinth cfg
      getLabyrinthBounds lab `shouldBe` (4, 4)
```

Thankfully, a RocksDB database can be committed into the project as a bunch of files. You can find the following test data in Hydra:

```
app/labyrinth/test/Labyrinth/TestData/labyrinths.rdb/CURRENT
app/labyrinth/test/Labyrinth/TestData/labyrinths.rdb/IDENTITY
app/labyrinth/test/Labyrinth/TestData/labyrinths.rdb/LOCK
app/labyrinth/test/Labyrinth/TestData/labyrinths.rdb/LOG
app/labyrinth/test/Labyrinth/TestData/labyrinths.rdb/MANIFEST
```

The Redis database, however, can't be easily bundled with your application because Redis is a standalone service, not an embeddable one (as with RocksDB or SQLite). Normally, you'd need to improve your testing infrastructure so that you could set up and tear down your external services before each test case. Usually, the slowness of integration tests prevents them from being run on a regular basis. And their instability can be annoying for a developer who's finished a feature but has tests that are failing for some unclear reason. The difficulty of organizing a proper environment also makes integration testing a bit

costly. While every project should have integration tests, they require a fair amount of preparation before they can become useful.

But this is a separate story. We've learned how to do integration testing with Free monads (nothing special, really). Further reading will demonstrate even more testing techniques.

9.1.6 Acceptance testing

Climbing a bit higher than integration testing, we get into the area of acceptance testing. Now our goal is not to check the validity of component integration but to see how the software solves a business's problems — whether the application works as expected, for example, or does what was prescribed.

Acceptance testing can come in different forms. Usually, it's something like manual testing, where you click the application's UI or send some test requests to its public interfaces. It can sometimes be automated, but this can be tricky. Even though your application isn't a code, you'll have to find a way to send commands to it, emulate user activities, understand the results it returns, and so on. Certainly, you need a ready environment to start your application in.

Let's try to enumerate our possible options in case of different application types:

- The application is a RESTful backend. Simple. Acceptance testing is just sending some requests to the server and getting results back. Even a bare bash script can serve as an automated acceptance test. Special tools for dealing with HTTP APIs (like `Postman`) will help here more though.
- The application is a web service. Most likely, testing would be the same as in the previous point.
- The application is a command-line tool. It can receive commands via a standard input stream and can produce results by outputting them into a standard output stream. Testing will be a script that interacts with the application using pipes. However, parsing and understanding the results can be tricky.
- The application is a desktop GUI program. This is hard to automate. You can find some solutions for different GUI systems (like for Qt or GTK), but automating this kind of testing isn't a common practice. It's a big problem. So, manual testing by your friend is your friend.

- The application is …

Okay, here I'll stop talking about acceptance testing. Why? Because it's not really about design and architecture. I just wanted to show you that there are different kinds of testing, more or less automatic, more or less useful. As a software architect, you'd better have a whole picture of software engineering — in case you need it.

As for the application we have … feel free to run the Labyrinth application and do a manual acceptance testing of it. Maybe you'll find it funny (see figure 9.1).

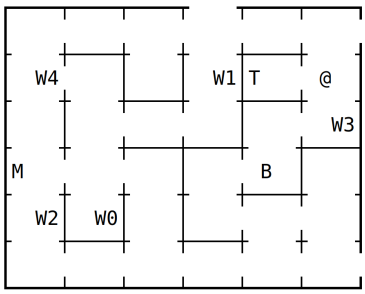


Figure 9.1 Sample of a labyrinth from the Labyrinth game

9.2 Advanced testing techniques

This might sound too brave — "advanced techniques" — because mocking and unit testing are so common in mainstream development that it's hard to call this secret knowledge. Still, it might not be that clear how and why we should deal with mocking in functional programming. Isn't our functional programming world blessedly free from these OOP practices like mocking? Aren't we satisfied by the correctness Haskell gives us by default? Should we even borrow these ideas from the mainstream? In Haskell, all the code is pure — even that which works in the `IO` monad. Does this mean we can just use black box testing and be happy?

Actually, no. The problems we face when running our code in tests are very much the same as in mainstream development. A badly designed application can't be easily tested because it often depends on dozens of external services and does a lot of side effects. Controlling these side effects will not only make the

code and the architecture better but will also enable many nice possibilities for testing.

Let's sort this out.

9.2.1 Testable architecture

As we know, functional programming isn't different from OOP in the design space. So, when do we have a right to call the application architecture testable? Several requirements should be met for this:

- Every subsystem (external service) is hidden behind an interface.
- Business logic is written against interfaces.
- There are no hacks to bypass interfaces in the business logic.
- There are no bare effects in the business logic.
- It's possible to substitute the implementation of every interface with a mock implementation (sometimes called a "stub") without rewriting the logic.

You'll see that functional interfaces are quite suitable for this. You can do mocking and white box unit testing. Free monads here seem to have the best testability out of all other approaches because they provide the best separation of concerns. Also, it's very hard to bypass a Free monadic framework in order to evaluate some hidden, unpredictable effect. Compare the two labyrinth generation functions (see listing 9.3). They do the same thing, but the second can be easily hacked by an intruding side effect, which ruins the code's testability:

Listing 9.3 Two functions: an innocent one and a criminal one

```
-- Completely safe logic in the LangL monad:
generateRndLabyrinth :: LangL Labyrinth
generateRndLabyrinth = do
  xSize <- getRandomInt (4, 10)
  ySize <- getRandomInt (4, 10)
  generateLabyrinth (xSize, ySize)

-- Logic with a dangerous effect in the IO monad:
generateRndLabyrinthIO :: IO Labyrinth
generateRndLabyrinthIO = do
  xSize     <- randomRIO (4, 10)
```

```
  ySize     <- randomRIO (4, 10)
  System.Process.runCommand "rm -fr /"            #A
  generateLabyrinthIO (xSize, ySize)
```

#A Dangerous side effect

Now that we agree on what makes an architecture testable, a new question arises: why it should be. Is inventing a testable architecture just extra time and effort that could be spent more effectively on writing and delivering features?

The reason is usual: you can secure a lot of risks with a good architecture and save a lot of time and money in the long term. And the architecture can't be good if it's untestable.

> **TIP** Martin Fowler said a lot about good architectures. Check out his resources, they are worth it!
>
> **LINK** Martin Fowler, *Architecture*
> https://martinfowler.com/architecture/.

Introducing interfaces should immediately imply that we can provide several implementations for them. This in turn should imply that we can provide a mock implementation, with dummy data returned when some method of the interface is called. Running business logic in tests when many interfaces are mocked will lead us to white box testing. Figure 9.2 explains the idea of interfaces, white box testing, and mocks.

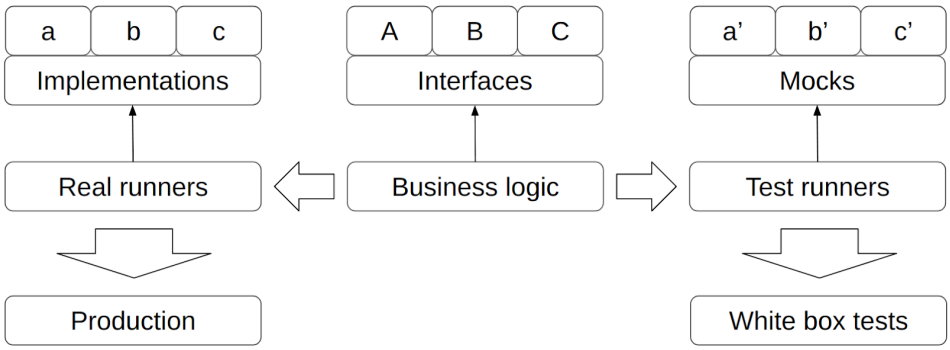


Figure 9.2 Real and test environments, white box tests, and mocks

This is a universal scheme equally applicable to OOP or functional programming no matter what kind of interfaces you have: OOP interfaces, Free monads, the Service Handle pattern, or just first-class functions in the `IO` monad. We can for sure treat functional programs as pure expressions that won't necessarily be white box testable. Even the `IO` monadic code can be seen as a pure one. But it's a technical detail rather than a reasoning we should use when we talk about software design and architecture. A bare `IO` code without any notion of interface can't be recognized as nicely architected because it lacks a separation of concerns. Sometimes this is okay, but the bigger a code base, the more you need testing with mocks. Bugs, inappropriate behavior, regressions, meeting the requirements, important invariants — all these things can be secured with white box tests. Integration testing within greenhouse conditions is important and can help, but it may be insufficient to tell us everything we're interested in.

For example, with integration tests, it's not easy to challenge the code using imaginary subsystem failures like an RDBMS outage, because how would you precisely control the moment the outage should happen during the test run? It's tricky unless you know about IoC, Dependency Injection, and mocking. With these techniques, you can configure your test environment so that a call to the database will be redirected to a custom mocked service that's programmed to answer with a failure in the middle of the logic. Other subsystems can be mocked to produce a different behavior needed for this specific test. A testable architecture allows you to slip into every possible dark corner of your business

logic in order to see how it handles the unpredictable behavior of any external system.

Now let's talk about specific techniques more closely.

9.2.2 Mocking with Free monads

We've talked about Free monads and their ability to divide the code into layers clearly separated from each other. In fact, there can be interpreters not only for real subsystems but also for mocked ones. Figure 9.3 specifies figure 9.2 in this sense.

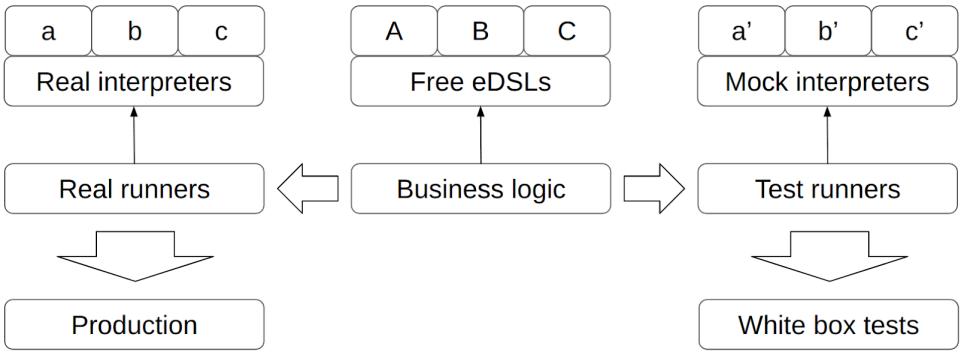


Figure 9.3 Free monads, mocks, and white box tests

There are two possible ways to implement this scheme:

- Update real interpreters with some additional mocking mechanism adjacent to the existing code.
- Introduce new interpreters specifically for a testing framework.

Although the first option seems to be a violation of the SRP, the section about automatic white box testing offers some promising possibilities. The second option interests us here.

Suppose we want to mock a random generation subsystem that is represented by the `RandomL` free language. Let me show you the language itself, which has a single method, and the live interpreter, a very simple one:

```
-- The language:
data RandomF next where
  GetRandomInt :: (Int, Int) -> (Int -> next) -> RandomF next

type RandomL = Free RandomF

-- The live interpreter:
interpretRandomF :: RandomF a -> IO a
interpretRandomF (GetRandomInt range next) = do
  r <- randomRIO range     -- native function call
  pure (next r)

runRandomL :: RandomL a -> IO a
runRandomL rndAct = foldFree interpretRandomF rndAct
```

Now, a special mocking interpreter should be able to produce a predefined value (mock) without touching any native functions:

```
-- The mocking interpreter:
interpretRndMocking :: RandomF a -> IO a
interpretRndMocking (GetRandomInt _ next) =
  pure (next 4)      -- chosen by fair dice roll.
                     -- guaranteed to be random.
```

We can also configure the interpreter using an additional parameter, like this:

```
interpretRndMocking :: Int -> RandomF a -> IO a
interpretRndMocking mockVal (GetRandomInt _ next) =
  pure $ next mockVal

runRndMocking :: Int -> RandomL a -> IO a
runRndMocking mockVal rndAct =
  foldFree (interpretRndMocking mockVal) rndAct
```

This code ignores the range variable, so it can produce an inappropriate value like `“7”` when the range is `[0,6]`. Depending on your needs, you may want to add a sanity check. An exception can be thrown here, which will fail a test:

```
interpretRndMocking mockVal (GetRandomInt (from, to) next) = do
  when (mockVal < from || mockVal > to)
       (error "Invalid mock value: out of range")
  pure (next mockVal)
```

But more often, the mocks should vary on each method call. First, it returns `4`, then `3`, then `100`, then something else — this should be controlled in tests and completely defined by a test case. In OOP languages, this is usually achieved by having a stateful mock object that holds a list of mock values and returns them one by one. Let's do this for our mock interpreter by analogy. A stateful interpreter with a mutable list of mocks is presented in the next listing.

Listing 9.4 Stateful mocking interpreter

```
interpretRndMocking :: IORef [Int] -> RandomF a -> IO a
interpretRndMocking mocksRef (GetRandomInt _ next) = do
  mockVal <- popNextRndMock mocksRef
  pure (next mockVal)

popNextRndMock :: IORef [Int] -> IO Int
popNextRndMock mocksRef = do
  mocks <- readIORef mocksRef
  case mocksRef of
  []     -> error "No mocks available"
  (v:vs) -> do
    writeIORef mocksRef vs
    pure v
```

Having such a configurable test interpreter gives us a lot of flexibility in tests. You can provide a certain value for very specific cases and enter into any branch of your business logic. Suppose you have the following scenario for getting a random activity to the nearest weekend:

```
games :: Int -> String
games 0 = "Minecraft"
games 1 = "Factorio"
games 2 = "Dwarf Fortress"
games _ = "Your decision"

getWeekendActivity :: RandomL String
getWeekendActivity = do
  baseActivity <- getRandomInt (0, 4)
  case baseActivity of
    0 -> pure "Sleep"
    1 -> pure "Walk"
    2 -> pure "Read"
```

```
    3 -> do  -- play computer game
      gameIdx <- getRandomInt (0, 2)
      pure $ games gameIdx
```

Now, it's easy to route an evaluation to the point when the activity "play computer games" with an option "Dwarf Fortress" will be chosen. There should be two mock values provided: `3` and `2`. The corresponding test is presented as follows:

```
spec :: Spec
spec = describe "Random scenarios test" $
  it "getWeekendActivity returns the best game ever" -> do

    mocksRef <- newIORef [ 3, 2 ]    -- create mocks

    activity <- runRndMocking mocksRef getWeekendActivity
    activity `shouldBe` "Dwarf Fortress"
```

So, mocking with Free monads is simple. However, we missed something important. Think about three questions:

- The return type of the `GetRandomInt` method is known — it's `Int`. But what if the method returns an arbitrary value of an unknown type, which is determined by a concrete script only?
- How would we do mocking in case of Hierarchical Free Monads? How would the testing framework be constructed?
- How do we write more generic white box tests with an option to call a real effect by occasion? In other words, how to mix real and mocked interpretation of calls?

The next sections answer these questions.

9.2.3 White box unit testing

The detailed explanation given earlier on how to do mocking with Free monads now opens a nice path to a complete unit/functional, white/gray box testing. However, the first question — why should we do this? — has some importance, so let's talk about motivation.

With white box unit testing we can

- Fixate the exact behavior of the code.
- Provide protection against regressions when some part of the business logic is considered sensitive and should not be changed by occasion.
- Ensure that IO-bound scenarios work as expected in all manifestations (effects, algorithms, interaction with external services, and so on).
- Simulate a subsystems outage and check the reaction of the code.
- Refer to tests as a sort of documentation.
- Implement a deep fuzzy testing to investigate the stability of the code.

On the other hand, we should be careful. Although white box testing can shed light on the most subtle details of the code, this technique can be expensive and mind-, time-, and patience-consuming. Full, 100% testing coverage tends to be heavy, hardly maintainable, and often senseless. The more code there is, the more unit tests there should be, and the more labor is expected. All-encompassing white box tests can slow down a project drastically, and any extra lag can turn into a money loss.

> **NOTE** I personally have a hypothesis that unit testing of business logic requires `O(n^3)` effort, and I'm proposing the following formula to calculate it:
>
> ```
> Efforts = N(scenarios) * AvgN(calls) * AvgN(cases)
>
>         = O(n^3)
> ```
>
> where `N` stands for "number," and `AvgN` is for "average number." It's obvious what "scenarios" mean here. The `calls` variable represents how many calls a regular scenario has — that is, how long the scenario is. The `cases` variable counts the variability of the data and control flow branches (if conditions, case switches, and so on). This variable is needed because full coverage by unit tests requires checking all possible data values that have a meaning, that lie on bounds, and that can be processed by the scenario.
>
> Don't take that formula too seriously — I'm not a scientist and can be wrong.

While the usefulness of white box tests remains debatable, there's one strong argument why we should learn it: it's another software developer's tool, and we don't have a right to be unaware of it.

Do you remember listing 9.2, a scenario for loading a labyrinth from a KV database? The next listing does a bit more. It uses several effects (KV DB, state handling) and reveals some critical points of possible failure.

Listing 9.5 A broader scenario for loading a game session

```
-- A type to keep all the operational data about the game.
data GameState = GameState
  { labyrinth :: StateVar Labyrinth
  , playerPos :: StateVar (Int, Int)
  }

-- Loading a game from KV DB.

type GameIdx = Int

loadGame :: DBConfig LabKVDB -> GameIdx -> AppL GameState
loadGame kvDBConfig gameIdx = do

  labyrinth <- loadLabyrinth' kvDBConfig gameIdx          #A
  playerPos <- loadPlayerPosition' kvDBConfig gameIdx     #A

  labyrinthVar <- newVarIO labyrinth
  playerPosVar <- newVarIO playerPos

  pure $ GameState labyrinthVar playerPosVar
```

#A These operations can throw an exception

The two functions used here, `loadLabyrinth'` and `loadPlayerPosition'`, are known to be unsafe. They can throw exceptions if the KV database doesn't respond or responds unexpectedly. A possible implementation can be as follows. Notice two database functions and the error call:

```
loadPlayerPosition' :: DBConfig LabKVDB -> Int -> AppL Pos
loadPlayerPosition' kvDBConfig idx = do
  conn       <- getKVDBConn kvDBConfig
```

```
  ePlayerPos <- withKVDB conn (loadEntity idx)
    Left _      -> error "Failed to load the player pos"
    Right plPos -> pure plPos
```

For a better understanding, consult figure 9.4, in which you can see all the methods. Bars with numbers are marking methods that will be mocked.

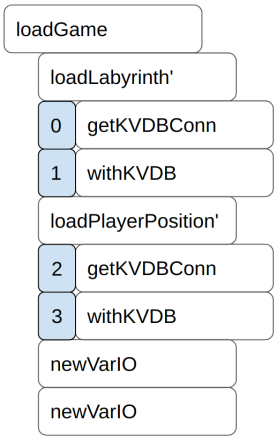


Figure 9.4 Loading scenario

Let's simulate a failure. Suppose we want to test that the `loadEntity` function has returned `Left`, and a dangerous `error` function is triggered. The following test (listing 9.6) does this. It's written on top of an advanced testing framework that we'll investigate a bit later. The test declares a series of mocks and configs to pass through the logic. The mocks make `loadLabyrinth'` and `loadPlayerPosition'` behave differently. Try to guess why we don't need mocks for the two `newVarIO` functions.

Listing 9.6 White box test of the game loading function

```
spec :: Spec
spec = describe "Labyrinth KV DB tests" $
  it "Load player position should throw exception" $ do

    mocksVar <- newIORef                                    #A
      [ (0, mkMock $ MockedKVDBConn "LabKVDB")              #B
      , (1, mkMock $ Right testKVDBLabyrinth)
```

```
      , (2, mkMock $ MockedKVDBConn "LabKVDB")            #C
      , (3, mkMock $ Left $ DBError "KVDB Failure")
      ]

    eGameState <- try $ Test.runAppL mocksVar             #D
                      $ loadGame testKVDBConfig 1

    case eGameState of
      Left (e :: SomeException) ->
         show e `shouldBe` "Failed to load the player pos"
      Right _ -> fail "Test failed"
```

#A Mocks preparation
#B 2 mocks for the DB stuff in the loadLabyrinth' function (connection and successful query result)
#C 2 mocks for the DB stuff in the loadPlayerPosition' function (connection and failed query result)
#D Running the scenario using a special testing interpreter

In general, we should avoid the `error` function in favor of more controllable ways of processing errors. The Hydra framework is designed with the idea that writing a safe business logic should be simple: no need to think about asynchronous exceptions, no need to care about the particular native libraries and their anarchical attitude to this difficult theme. Therefore, we might expect that someone will decide to refactor the preceding code. As you remember, the `LangL` and `AppL` monads in Hydra have a nice method for throwing domain-level exceptions: `throwException`. With it, the following change was made in `loadPlayerPosition'`:

```
-    Left _ -> error "Failed to load the player pos"
+    Left _ -> throwException $ LoadingFailed "Player pos"
```

where `LoadingFailed` is a custom exception:

```
data AppException = LoadingFailed Text
  deriving (Show, Generic, Exception)
```

In turn, the refactoring continues, and the next change aims to protect the whole scenario from occasional exceptions. This can be done with the `runSafely` method, which of course is able to catch the `LoadingFailed` exception. See listing 9.7, which is the result of refactoring.

Listing 9.7 A safe loading scenario after refactoring

```
loadGame
  :: DBConfig LabKVDB
  -> GameIdx
  -> AppL (Either Text GameState)
loadGame kvDBConfig gameIdx = do

  let load1 = loadLabyrinth' kvDBConfig gameIdx            #A
  let load2 = loadPlayerPosition' kvDBConfig gameIdx

  eLabyrinth <- runSafely @AppException load1              #B
  ePlayerPos <- runSafely @AppException load2

  case (eLabyrinth, ePlayerPos) of                         #C
    (Left exc, _) -> pure $ Left $ show exc
    (_, Left exc) -> pure $ Left $ show exc

    (Right labyrinth, Right playerPos)  -> do
      labyrinthVar <- newVarIO labyrinth
      playerPosVar <- newVarIO playerPos
      pure $ Right $ GameState labyrinthVar playerPosVar
```

#A Two unsafe actions
#B Catching the AppException exception
#C Processing the result somehow

> **TIP** `runSafely` will not catch exceptions other than `AppException`, and thus the `loadGame` function isn't completely safe. You might want to catch `SomeException` instead.

Apart from the more verbose and more explicit business logic, this looks better, right? We can now trust this function because it follows the rule "explicit is better than implicit." Refactoring is a very important practice. We should all do it at every opportunity. However, the test we wrote earlier doesn't work anymore. It fails with an exception "Failed to convert mock" or something similar. Why? Because the scenario has changed significantly. By the nature of the `runSafely` method, the scenario no longer contains the two monadic chains of the two loading helper functions. These chains are now embedded into

the `runSafely` method, and the test doesn't know how to get inside. The mocks now mismatch the calls (see figure 9.5).

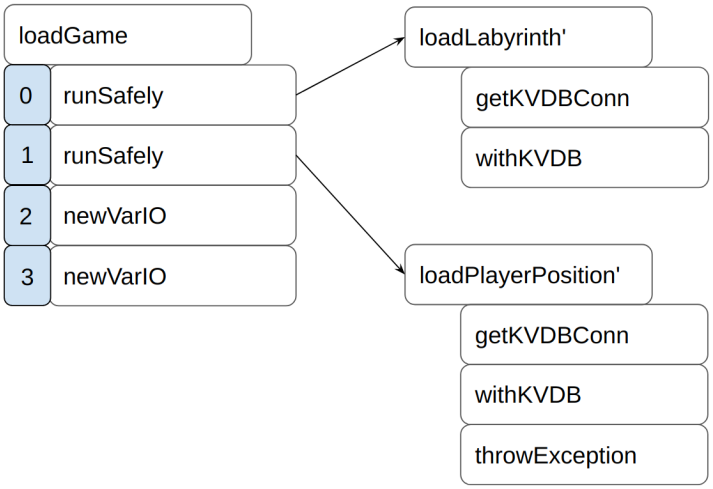


Figure 9.5 Scenario after refactoring with runSafely

> **NOTE** The `throwException` method in the `loadLabyrinth` won't be called, so there's no such bar in figure 9.5.

Previously, mocks were matched to KV DB functions, and now they're associated with `runSafley` and `newVarIO` methods. The test should be updated after the logic itself. But now we're entering a quite subtle area: the essence of mocks for hierarchical languages.

Let me remind you about the type of the `runSafely` method:

```
runSafely :: AppL a -> AppL (Either Text a)
```

The method works like so: the internal action is evaluated by the interpreter recursively. It's also surrounded by the `try` combinator, here is the interpreter:

```
interpretAppF appRt (RunSafely act next) = do
  eResult <- try $ runAppL appRt act
  pure $ next $ case eResult of
    Left (err :: SomeException) -> Left $ show err
    Right r -> Right r
```

Now we have a choice. We can mock the result of `runSafely` while ignoring the action. Or we can configure the mocking interpreter to descend deeper and interpret the action. Interpreting the internal action, of course, is no different than interpreting the external code because it's the same `AppL` language. However, after descending, we should be able to mock the methods of the internal action. In our case, mocks should be lifted down to the scenarios `loadLabyrinth'` and `loadPlayerPosition'` (see figure 9.6).

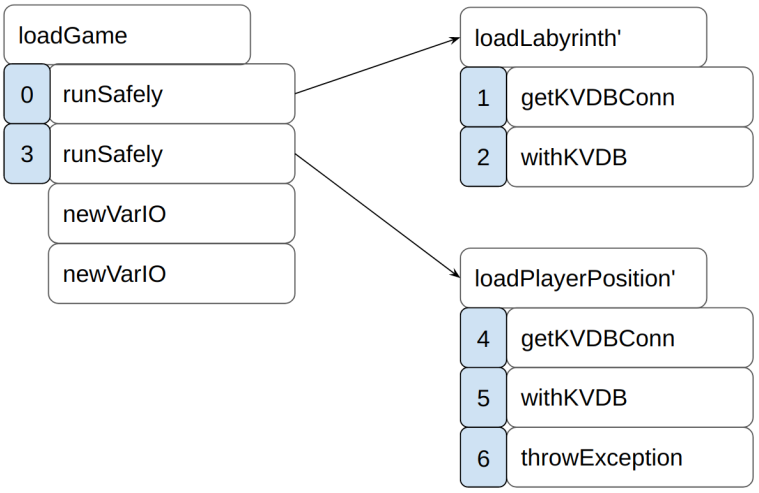


Figure 9.6 Lifting mocks down by a hierarchy of calls

Actually, the aforementioned is true for every hierarchical aspect of Free monadic languages. The `AppL` language has actions in `LangL`, and the `LangL` language has actions in other internal languages. In our white box tests, we might want to either stay on the top levels of the business logic or test it more granularly. Moreover, sometimes we'd like to evaluate a live interpreter to perform a real effect and get a result. The reasons may vary, the intentions may vary, but the mechanism to achieve this is always the same: configurable mocks.

The updated white box test is presented in listing 9.8. This time, we create `AppRuntime` in complement to `TestRuntime` because the test interpreter might meet a mock item (`RunRealInterpreter`), which issues commands to evaluate a real effect.

Listing 9.8 Updated white box test that uses a real runtime as well

```
spec :: Spec
spec
  = around withAppRuntime
  $ describe "Labyrinth KV DB tests"
  $ it "Load player position should throw exception"
  $ \appRt -> do

    mocksVar <- newIORef
      [ (0, RunRealInterpreter)                          #A
      , (1, Mock $ MockedKVDBConn "LabKVDB")             #B
      , (2, Mock $ Right testKVDBLabyrinth)
      , (3, RunRealInterpreter)                          #C
      , (4, Mock $ MockedKVDBConn "LabKVDB")             #D
      , (5, Mock $ Left $ DBError "KVDB Failure")
      , (6, RunRealInterpreter)                          #E
      ]

    let testRt = TestRuntime (Just appRt) mocksVar
    let action = loadGame testKVDBConfig 1

    eGameState <- try $ Test.runAppL testRt action
    case eGameState of
      Left (e :: SomeException) ->
         show e `shouldBe` "Failed to load the player pos"
      Right _ -> fail "Test failed"
```

```
#A Mock for the 1st runSafely real method
#B 2 mocks for the DB stuff in loadLabyrinth'
   (connection and successful query result)
#C Mock for the 2nd runSafely real method
#D 2 mocks for the DB stuff in loadPlayerPosition'
   (connection, failed query result)
#E Step for throwException real method
```

It’s not hard to guess that the testing runner is a bit more complex because we can choose the methods that will go to real interpreters and return to the testing one. We can mix white box and black box integration testing. This becomes more interesting, don’t you agree? The next section briefly describes the testing framework.

9.2.4 Testing framework

The core idea for mixing real and mocked effects lies in the `Step` data type:

```
data Step
  = Mock GHC.Any
  | RunTestInterpreter
  | RunRealInterpreter
```

We use a similar typed/untyped trick with `GHC.Any` for storing mocks — that one which is introduced in chapter 6 (see figure 6.10). However, this time the trick isn't type-safe. It's possible to provide a mocked value of a completely wrong type, not of a type a particular step requires. This will result in a runtime error on test replay. Not great, not terrible — we can live with this vulnerability. There's no strong requirement for tests to be strictly verified and typed. If a mock is "mistyped," the test will fail. Simply go and fix it.

We've already seen how the option `RunRealInterpreter` works. The option `RunTestInterpreter` means the same thing, but it's not always clear what effect should be evaluated by the test interpreter. One possible action is simulating a subsystem failure. Or you might want to collect a call stack of a scenario, with the idea to verify it in the test. Or you want to know the number of certain operations made by a scenario. And so on.

> **TIP** By the way, with Free monads and step-by-step evaluation, it's possible to specify the exact number of calls that will be evaluated. This is a great way to control blockchain smart contracts, for example. The "gas" concept doesn't require complicated mechanisms anymore! And other use cases for counting the calls are coming to mind. Thus, a Free monadic service can have a subscription model in which a small client buys a package for 1,000 steps in a scenario, a medium client can run scenarios up to 5,000 steps, and enterprise clients are offered with 10,000 steps in total.

Consider the following implementation of the testing interpreter. The `TestRuntime` data type which holds the call stack and steps for mocks:

```
type MethodName = String
type MethodData = String

data TestRuntime = TestRuntime
  { traceSteps :: Bool
  , callStack  :: IORef [(MethodName, MethodData)]
  , appRuntime :: Maybe AppRuntime
  , steps      :: IORef [Step]
  }
```

The interpreter that dispatches the current step and does the action the step requires:

```
interpretAppF :: TestRuntime -> AppF a -> IO a
interpretAppF testRt fAct@(EvalLang action next) = do

  when (traceSteps testRt)                                #A
      $ modifyIORef (("EvalLang", ""):)
      $ callStack testRt

  mbStep <- popNextStep testRt                            #B
  let mbRealAppRt = appRuntime testRt

  case (mbStep, mbRealAppRt) of                           #C
    (Just (Mock ghcAny), _) ->                            #D
       pure $ next $ unsafeCoerce ghcAny
    (Just RunTestInterpreter, _) ->                       #E
       next <$> runLangL testRt action
    (Just RunRealInterpreter, Just appRt) ->              #F
       Real.interpretAppF appRt fAct
    (Just RunRealInterpreter, Nothing) ->                 #G
       error "Real runtime is not ready."
    (Nothing, _) -> error "Mock not found."
```

#A Collecting a call stack
#B Obtaining the current step and real runtime
#C Dispatching the step
#D Returning a mock
#E Running a nested test interpreter for the LangL action
#F Running a real effect
#G Errors on case when the test is configured wrongly

Once the `RunRealInterpreter` value is met, the real interpreter will be called, utilizing the `AppRuntime` that's stored in the `TestRuntime`. Interestingly, we use a real interpreter only for a single call, but once it's evaluated, we return to the test interpreter. The `callStack` variable keeps the call stack, respectively. In this particular method, only `EvalLang` will be written, but sometimes we can output more useful info. For example, the `throwException` method carries an exception. In the KV DB methods, it's the key and value that can be shown. For SQL methods, it's logical to store the resulting SQL query. In other words, the data that's being transferred through the scenario might be in tests for additional verification. Sample test:

```
void $ Test.runAppL testRt $ loadGame testKVDBConfig 1

stack <- readIORef $ callStack testRt

stack `shouldBe`
  [ ("AppF.EvalLang", "")
  , ("AppF.InitKVDB", "./test/labyrinth.db")
  , ("AppF.EvalLang", "")
  , ("LangF.RunKVDB", "\"loadEntity\" \"LabKey 1\"
     \"No value found\"")
  , ("LangF.ThrowException"
  , "InvalidOperation \"Labyrinth with key \"1\" not found.\"")
  ]
```

The framework can be improved in many different ways: measuring and verifying timings of methods; discovering space leaks; simulating different responses from external services; evaluating concurrent code in a deterministic way when you explicitly order your threads on which operation to start and when to stop. You can create several different testing frameworks and use them for unit, integration, and functional tests. Consider rolling out your own testing environment for your particular requirements and cases. I hope you now agree that Free monadic applications are very easy to test.

But now I see that you're scared by the amount of work you have to do with white box tests. You're right that writing and maintaining white box tests can quickly become a disaster. So tedious. So boring. But don't give up too early; Free monads are here to rescue you. Free monads and the very possibility of not having to write a line of a white box test but still have the test anyway. You will

not write a single test, but all your scenarios will be covered by white box or grey box tests. I hope this sounds intriguing enough. Read the next section, where I'll briefly tell you about ...

9.2.5 Automatic white box testing

If you search for the phrase "unit testing" in Google, you'll get a lot of very similar pyramidal diagrams explaining the hierarchy of testing. Some of the diagrams say that the bottom approach (unit tests) is cheap but the higher you go the more expensive testing becomes. I personally don't agree with such "common knowledge," which is very widespread according to the number of pyramids found on Google, but let me present it anyway. Figure 9.7 resembles one such diagrams.

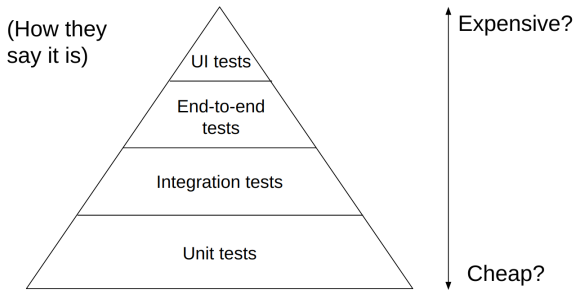


Figure 9.7 A pyramid of testing approaches (highly debatable)

Why can unit tests be expensive? This depends on what we call a "unit test." White box unit tests can take too much time to maintain because they tend to multiply with an increase in scenarios, cases within those scenarios, and the length of the logic. Seems to be an unavoidable evil, right? In this section, I present an approach to automatic test case generation. This approach frees you from writing tests and, in general, can be very useful for the methodology that we can generally call "golden tests".

> **DEFINITION** Golden tests represent a fixed set of detailed tests, white box ones mostly, that should fixate the exact behavior of some logic. It's important that this logic should be always correct and working, - hence golden tests.

> **NOTE** This section is not an instruction on how to implement automatic white box testing, but rather a short description of the idea. The approach is real and tested in production. I wrote a detailed article about the approach, including references to actual commercial code bases. You'll also find a working showcase project there. Run it and see how it works:
>
> **LINK** Alexander Granin, *Automatic white-box testing*
> https://github.com/graninas/automatic-whitebox-testing-showcase
>
> **NOTE** The Hydra framework in its current form doesn't have the approach implemented yet.

So, what's the idea? With a special mechanism baked into a Free monadic framework, every scenario becomes "serializable," kind of. When you run a scenario in the recording mode, each step can be tracked — written into a JSON file as a flow of steps, including inputs, outputs, and other useful data. After you get a recording file, you can replay it against a changed logic. The recording will act like a mock. Replaying can easily reveal the difference between the old run and the new one, reporting the issue and pointing exactly to the step where it happened.

Check the following scenario, `loadPlayer'`. This scenario is similar to what you saw in listing 9.2, but the `withKVDB` method returns `DBResult` (`Maybe Pos`). In order to bring more evils to our world, we do not accept this decent behavior and forcibly turn an `Either` value into the exception. Also, we use the unsafe connection method:

```
loadPlayer' :: LangL Pos
loadPlayer' = do
  conn <- connectOrFail kvDBConfig
  ePos :: DBResult (Maybe Pos) <- withKVDB conn
     $ loadEntity $ PosKey 99999
  case ePos of
    Left err -> throwException $ LoadingDataFailed $ show err
    Right (Just kvdbPos) -> pure kvdbPos
    Right Nothing -> do
      logInfo "No records found."
      throwException $ LoadingDataFailed "No records found."
```

Additionally, the case (`Right Nothing`) prints some info into the logger. How evil we are! Why is the logging operation here, and why is it on the "info" level when the next line throws an exception? I'd rather expect the "error" level (`logError`). I wouldn't throw the exception. I'd return a `Maybe` value instead.

Nevertheless, turning this code into a JSON recording will produce something like this (don't mind the formatting, — it's arbitrary for the demonstration purposes):

```
// recording.json:
{"tag":"Recordings","contents":
  [ [0, "GetKVDBConnEntry",
        {"result":{"Right":[]},
        "kvdbConfig":"./prod/labyrinths"}
    ]
  , [1, "RunKVDBEntry",{"result":{"Right":null}}]
  , [2, "LogMessageEntry",
        {"level":"Info", "message":"No records found."}]
  , [3, "ThrowExceptionEntry",
        {"exception":"No records found."}]
  ]
}
```

You see an imaginary recording because the Hydra framework doesn't support this feature yet, but this is how it might look. You see that every step is written in the form of some entry. Each entry is accompanied by some info describing the step. The system's replaying mechanism will traverse the entries along with scenario steps and will match the data. For example, we patch the scenario slightly in this row:

```
- logInfo "No records found."
+ logError "No records found."
```

Now the player mechanism will get an unexpected `Error` level for step 2. The player, that is either a subsystem, or a mode of your application, will stop and produce a message:

```
$ myapp player "recording.json"

Failed to replay the recording. Step 2 doesn't match.
  Expected: LogMessageEntry Info "No records found."
  Got: LogMessageEntry Error "No records found."
```

The message clearly points to the call site where the problem occured. The developer should now examine the change and decide whether it was expected or not. Fortunately, the recording file can be edited and fixed without recompiling the project. In this particular case, you just need to fix the content of step 2.

In reality, though, the changes are rarely that simple. Sometimes the code swells up from new functionality, and the old recordings become completely obsolete. No problem, new recordings can be taken just by running the application using the "recording" mode — again, without recompiling it. If the code supports the automatic white box system, all the scenarios will be automatically recordable and replayable, no matter what.

Figure 9.8 presents a common scheme for the player.

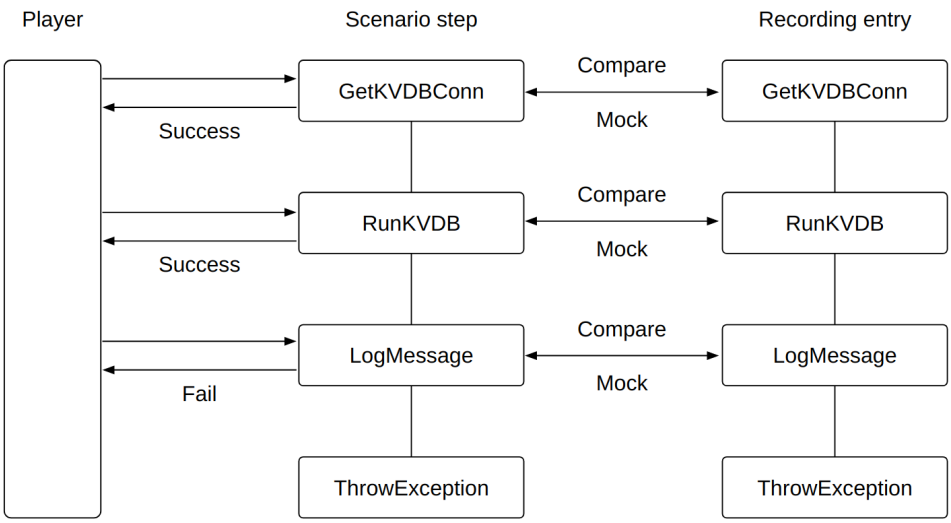


Figure 9.8 Principled scheme of the automatic white box testing system

A JSON recording can also be a source of knowledge about what's going on in your logic. It's better than just logs. Do you know any project that has sufficient logging with clear logs that were put in the right places with the right level of severity? The theme of how to do a proper logging deserves its own book, if not a yearly conference. In turn, the recordings represent a complete fingerprint of the logic (not only the effective steps, if you're wondering), and the flow of a recording always corresponds to a particular scenario flow. This means you can read the recording and understand what happened there, step-by-step, with details such as SQL queries or input parameters. Finally, your business logic won't be polluted by occasional logging entries. In fact, the business logic will be completely unaware of this internal magic.

> **NOTE** The sentence about complete unawareness is not absolutely true. The methods of a Free monadic framework should be recordable, and all the output results should be serializable. This limits the ways in which the framework can be used, but, even so, not by much. Also, a bunch of subtle things should be addressed when implementing this approach for KV DB and SQL DB subsystems, as well as for state handling and threading ones. Nothing impossible though.

Now you understand why this approach makes the white box testing much cheaper, almost free. However, was it the only benefit? It wasn't that interesting. Look more closely at the recordings. While each step is a mock in a normal usage, it can be accompanied by some config that tweaks the player's behavior. Therefore, making a real effect happen instead of a mock requires a simple change in the recording:

```
[2, "LogMessageEntry",
    {"mode":"NoMock", "level":"Info",
     "message":"No records found."}]
```

With the `NoMock` option, you'll see a line printed by the real logger. Nice? Even more options are supported by the player:

```
data EntryReplayingMode
  = Normal
  | NoVerify
  | NoMock
```

```
data GlobalReplayingMode
  = GlobalNormal
  | GlobalNoVerify
  | GlobalNoMocking
  | GlobalSkip
```

`NoMock` for a database-related step will result in the interaction with the database. `NoVerify` liberates the checks so that the discrepancies of this particular step will be ignored. Global configs will change the behavior of the player across the whole recording... There are many other player modes that might be useful for your testing workflows. I even state that this subsystem is suitable for making a performance control over the needed steps. For example, you can require that a step with a database query should not take more time than a threshold. Something like this:

```
[1, "RunKVDBEntry",
    {"result":{"Right":null}, "time_threshold":100.0}
]

$ myapp player "recording.json"

Failed to replay the recording. Step 1 exceeded the time
threshold.
  Expected: less than 100.0ms
  Got: 112.3ms
```

This approach extends the idea of white/grey box testing and empowers our code with even more possibilities and use cases. I really think it is very powerful. We just need to find the best way of utilizing it.

9.3 Testability of different approaches

A bit of philosophy here. What can we say about the testability of different design approaches, such as the ReaderT pattern, Service Handle Pattern, Final Tagless, and others? Testability depends on how easily we can isolate the implementation from the actual logic. All the approaches from the previous chapter let us do Dependency Inversion in the code. This is the major technique that leads to better testability. The more easily you can substitute your effects by mocks, the higher the test quality will be. In this sense, the testability of value-based approaches should exceed the testability of more rigid ones. Here, Final Tagless/mtl must be the worst because its mechanism for Dependency

Injection is based on type-level substitution. However, this isn't the circumstance for a bare IO code. Imperative code with naked effects cannot be tested in general.

Another consideration relates to the ability to bypass the underlying system's interfaces. When you have a bare IO code, you don't have that underlying system at all, so you aren't limited in performing any effects you want. But most approaches can't provide a strict isolated environment either. Only Free monads are able to do that, which is why we constructed a Free monadic framework. So, having this framework, you can be calm. The code that people write will be testable anyway (except when they do dirty hacks).

The next table (table 9.1) compares the testability of different approaches.

Table 9.1 Comparison of the testability of different approaches

	Dependency Injection option	**Effect restrictions**	**Bypassing restrictions**	**Testability in general**
Bare IO	No DI	No	Easy	Very low
ReaderT over IO	Value-based	No	Easy	Low
Service Handle within IO	Value-based	No	Easy	Low
Final Tagless/mtl	Type-based	No	Possible	Low
Free monads	Value-based	Yes	Impossible	Best

9.4 Summary

Testing can't save you from bugs completely. Neither property-based testing nor integration nor functional testing can. However, the point of testing is not to eliminate bugs but rather to reduce risks. It's hard to say what the risk/test relation will look like exactly, but the more tests you have, the lower the risks are.

You don't write tests to find bugs. This is a common misconception. You write tests to check the requirements and contracts. Different testing covers different sets of requirements. Black box and white box tests verify the functional requirements (what the code should and should not do). Load, performance, and stress tests examine the nonfunctional requirements. Regression tests protect from unexpected breakages. And sometimes we write tests to find bugs.

Mocking is important. Achieving good (meaning useful) test coverage without mocks doesn't seem possible in the presence of side effects. A logic with a single-dependency external service becomes untestable. Calling the external service in tests makes them unreliable and slow. This isn't acceptable if we care about the security and safety of our data. Occasional calls made from tests may do something unwanted, and we certainly don't want to leave the "live" credentials to be kept in tests. We can go with integration testing and a special testing environment, but these tests won't be that stable. This is why we should use mocking.

In turn, mocking is possible only when you have a good application architecture with the subsystems decoupled from each other and covered by interfaces. Again, we talk about the fundamental IoC principle along with the Dependency Injection idea. Not all architectures can offer you a certain level of decomposition. It seems — and I hope I've proven this — that Free monads are the most testable and reliable of all the design approaches.

Appendix A

Word statistics example with monad transformers

We considered several examples and discussed many aspects of the monads and monad transformers. Let's now try it one more time. The next example shows you a set of monad mechanisms. The code it demonstrates is much closer to real code that you might see in Haskell projects. It's highly combinatorial and functional, as it uses the point-free style, higher-order functions, and general monadic combinators. It requires a really strong will to go through, so you might want to learn more functional programming in Haskell while studying this example. Just tell yourself, "To be, or not to be: that is the question" when you're struggling with some difficult concept. By the way, Shakespeare's famous statement will be our data. The task is to get the following word statistics:

```
be: 2
is: 1
not: 1
or: 1
question: 1
that: 1
the: 1
to: 2
```

We'll take a text, normalize it by throwing out any characters except letters, then break the text into words; then we'll collect word statistics. In this data transformation, we'll use three monads: the `IO` monad to print results, the

`Reader` monad to keep the text-normalizing configuration, and the `State` monad to collect statistics. We'll solve this task twice. In the first solution, we just nest all monadic functions without mixing effects. In the second solution, we'll compose a monad stack. Listing A.1 shows the first solution.

Listing A.1 Word statistics with nested monads

```
import qualified Data.Map as Map                                    #A
import Data.Char (toLower, isAlpha)
import Control.Monad.State
import Control.Monad.Reader

data Config = Config
  { caseIgnoring :: Bool
  , normalization :: Bool
  }

type WordStatistics = Map.Map String Int

countWord :: String -> WordStatistics -> WordStatistics
countWord w stat = Map.insertWith (+) w 1 stat                      #B

collectStats :: [String] -> State WordStatistics ()
collectStats ws = mapM_ (modify . countWord) ws                     #1

tokenize :: String -> Reader Config [String]
tokenize txt = do                                                   #2
  Config ignore norm <- ask

  let normalize ch = if isAlpha ch then ch else ' '
  let transform1 = if ignore then map toLower else id
  let transform2 = if norm then map normalize else id

  return . words . transform2 . transform1 $ txt                    #3

calculateStats :: String -> Reader Config WordStatistics
calculateStats txt = do                                             #4
  wordTokens <- tokenize txt
  return $ execState (collectStats wordTokens) Map.empty

main = do
  let text = "To be, or not to be: that is the question."
  let config = Config True True
```

```
  let stats = runReader (calculateStats text) config
  let printStat (w, cnt) = print (w ++ ": " ++ show cnt)
  mapM_ printStat (Map.toAscList stats)                    #5
```

```
#A Haskell dictionary type
#B Add 1 to word count or insert a new value
#1 Computation in the State monad
#2 Another computation in the Reader monad
#3 Point-free style
#4 Monadic computation in the Reader monad
#5 Map a monadic function over a list
```

Let's discuss this code. When you call `main`, it calculates and prints statistics as expected. Note that we prepare data for calculations in the let-constructions. The `config` we want to put into the `Reader` context says that word case should be ignored and all nonletter characters should be erased. We then run the `Reader` monad with these parameters and put the result into the `stats` variable.[6] The `printStat` function will print pairs `(String, Int)` to the console when it's called in the last string of the listing, which is marked as #1. What does the function `mapM_` do? It has the following definition:

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
```

It has two arguments: a monadic function and a list of values that this monadic function should be applied to. We passed the `printStat` function that's monadic in the `IO` monad:

```
printStat :: (String, Int) -> IO ()
```

This means the `a` type variable is specialized by the type `(String, Int)`, `m` is `IO`, and `b` is `()`. This gives us the following narrowed type:

```
mapM_ :: ((String, Int) -> IO ()) -> [(String, Int)] -> IO ()
```

This monadic map takes the list of resulting word-count pairs and runs the monadic `printStat` function for every pair in the list. The underscore points out that the `mapM_` function ignores the output of its first argument. All results

[6] This isn't really true because Haskell is a lazy language. The let-construction just defines a computation, but it does nothing to evaluate it immediately. We bind this computation with the name "stats."

from the `printStat` calls will be dropped. Indeed, why should we care about the unit values `()`?

Let's move down by call stack. We run the `Reader` monad by the corresponding function (`runReader`) and pass two arguments to it: the `config` and the monadic computation (`calculateStats text`). As we see at #4, `calculateStats` calls the `tokenize` function that's a monadic computation in the same `Reader` monad. The `tokenize` function takes text and breaks it into words being initially processed by the chain of transformations #3. What these transformations will do depends on the `config` we extract from the `Reader`'s monad context by the `ask` combinator. The chain of transformations #3 is composed from three combinators:

```
return . words . transform2 . transform1 $ txt
```

Here, the calculation flow goes from right to left starting when the `txt` variable is fed to the `transformation1` function. The result (which is still `String`) is passed into the `transform2` combinator; then the new result (also `String`) is passed into the `words` combinator. The latter breaks the string into words by spaces. To demystify the point-free style used here, we can rewrite this expression as follows:

```
return (words (transform2 (transform1 txt)))
```

All these functions just evaluate their results and pass them further. Consequently, we can combine them by the composition operator `(.)`. It's simple:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

This operator makes one big combinator from several smaller consequent combinators. It's right-associative:

```
words . transform2 . transform1 :: String -> [String]
```

The ($) operator is the function application operator:

```
($) :: (a -> b) -> a -> b
```

It takes the function `(a -> b)` and applies it to the argument `a`. In this code, `a` is `txt :: String`. This operator helps to avoid unnecessary brackets. The following expressions are equal (this list is not exhaustive):

```
return . words . transform2 . transform1 $ txt
return . words . transform2 $ transform1 txt
return . words $ transform2 $ transform1 txt
return $ words $ transform2 $ transform1 txt
return $ words $ transform2 $ transform1 $ txt
return (words (transform2 (transform1 (txt))))
return (words (transform2 (transform1 txt)))
return (words (transform2 (transform1 $ txt)))
return (words (transform2 $ transform1 txt))
return (words $ transform2 $ transform1 txt)
return (words . transform2 . transform1 $ txt)
```

The `return` function is no different than other functions, which is why it's used as a regular combinator. It just wraps the argument into a monadic value:

```
return :: [String] -> Reader Config [String]
```

Try to infer the function type (`return . words . transform2 . transform1`) yourself.

Finally, consider the computation in the `State` monad #1. It takes a list of words and maps the monadic function (`modify . countWord`) over it. The definition of the `countWord` function says it's not monadic:

```
countWord :: String -> WordStatistics -> WordStatistics
```

Consequently, the library function `modify` is monadic. Indeed, it's the modifying state function that takes another function to apply to the state:

```
modify :: (s -> s) -> State s ()
modify :: (WordStatistics -> WordStatistics)
       -> State WordStatistics ()
```

It does the same as the following computation:

```
modify' f = get >>= \s -> put (f s)
```

The `mapM_` combinator takes this concrete state modifying function `(modify . countWord)` and applies it to the list of words. The results of applying it to each word in the list are identical; they are what the function `(modify . countWord)` returns being applied to the argument, namely, the unit value `()`. As we don't need these values, we drop them by using `mapM_`. Otherwise, we should use the `mapM` function:

```
collectStats :: [String] -> State WordStatistics [()]
collectStats ws = mapM (modify . countWord) ws
```

Note how the type of `collectStats` changes. It now returns the list of units. Completely uninteresting result.

It should now be clear what happens in listing A.1. The second solution for the same problem doesn't differ much. Three separate monads (subsystems) are mixed together into a single monad stack in this order: the `Reader` monad, the `State` monad, and the `IO` monad. We call functions from the `Reader` monad without any lift operators because this monad is on the top of the monad stack. For the `State` monad, we should lift its functions once because this monad is just behind the `Reader` monad; it lies one level down. Finally, the `IO` monad is on the bottom. We should call the `lift` combinator twice for all impure functions. Listing A.2 demonstrates this solution.

Listing A.2 Word statistics with the monad stack

```
type WordStatStack a                                              #1
  = ReaderT Config (StateT WordStatistics IO) a

countWord :: String -> WordStatistics -> WordStatistics
countWord w stat = Map.insertWith (+) w 1 stat

collectStats :: [String] -> WordStatStack WordStatistics
collectStats ws = lift (mapM_ (modify.countWord) ws >> get)  #2

tokenize :: String -> WordStatStack [String]
tokenize txt = do
  Config ignore norm <- ask

  let normalize ch = if isAlpha ch then ch else ' '
  let transform1 = if ignore then map toLower else id
```

```
  let transform2 = if norm then map normalize else id

  lift . lift $ print ("Ignoring case: " ++ show ignore)     #3
  lift . lift $ print ("Normalize: " ++ show norm)

  return . words . transform2 . transform1 $ txt

calculateStats :: String -> WordStatStack WordStatistics
calculateStats txt = do
  wordTokens <- tokenize txt
  collectStats wordTokens

main :: IO ()
main = do
  let text = "To be, or not to be: that is the question."
  let cfg = Config True True
  let runTopReaderMonad =
        runReaderT (calculateStats text) cfg
  let runMiddleStateMonad =
        execStateT runTopReaderMonad Map.empty

  let printStat (w, cnt) = print (w ++ ": " ++ show cnt)
  stats <- runMiddleStateMonad                              #4
  mapM_ printStat (Map.toAscList stats)
```

#1 The monad stack type
#2 Lifting state computation
#3 Lifting impure functions
#4 Running monad stack

When you run this solution, it will print a slightly different result:

```
"Ignoring case: True"
"Normalize: True"
"be: 2"
"is: 1"
"not: 1"
"or: 1"
"question: 1"
"that: 1"
"the: 1"
"to: 2"
```

Consider the type `WordStatStack a:` it represents our monad stack #4. The functions `collectStats`, `tokenize`, and `calculateStats` now belong to this custom monad. All of them share the context with the configuration, all of them are able to modify the state, and all of them can do impure calls. This is now one big subsystem with three effects. We run it from the bottom to the top of the monad stack, as mark #1 shows. We start from the `StateT` monad transformer because we don't need to run the `IO` monad transformer — we're inside the `IO` monad already:

```
runMiddleStateMonad = execStateT runTopReaderMonad Map.empty
```

The `execStateT` function calls the top monad transformer, namely, the `runReaderT`. The latter can run any function that has the type `WordStatStack a`. The function `calculateStats` now calls the monadic function `collectStats`.

The `collectStats` function #3 has something cryptic:

```
collectStats ws = lift (mapM_ (modify.countWord) ws >> get)
```

The (`>>`) monadic operator evaluates the first monadic function (`mapM_ (modify.countWord) ws`), omits its result, and runs the second monadic function (`get`). It does the same thing as the `do` block when we don't need the result of monadic action:

```
collectStats ws = lift $ do
  mapM_ (modify.countWord) ws   -- result is dropped
  get
```

The computation (`mapM_ (modify.countWord) ws >> get`) operates in the `State` monad that's represented by the `StateT` monad transformer over the `IO` monad:

```
(mapM_ (modify.countWord) ws >> get)
    :: StateT WordStatistics IO WordStatistics
```

To be run in the `WordStatStack` monad, it should be lifted once.

The `tokenize` function is very talkative now. It prints the configuration to the console #2, but first we must lift the function `print` into the `WordStatStack` monad. We call `lift` twice because the `IO` monad is two layers down from the `Reader` monad. You can also write it as follows:

```
lift (lift $ print ("Ignoring case: " ++ show ignore))
```

If you don't like this wordy lifting, you can hide it:

```
runIO :: IO a -> WordStatStack a
runIO = lift . lift
```

And then use it:

```
runIO $ print ("Ignoring case: " ++ show ignore)
```

This is a good way to provide the `runIO` combinator for your code users and hide the details of the monad stack. The client code would like to know that your monad stack has the `IO` effect, but it doesn't care how many lifts there should be.

The last thing we'll discuss is the separation of subsystems in the monad stack. In listing A.2, you really don't see this separation because all those monadic functions work in the single `WordStatStack` monad. Being composed together, they share the same effects. However, it's possible to rewrite the functions `collectStats` and `calculateStats` in order to make them work in different monads (subsystems). Our goal is to free the `collectStats` function from the `Reader` monad context because it doesn't need the configuration stored in the context. Still, the `collectStats` function should operate inside the `State` monad. Let's say it should also print words to the console. This is a new `collectStats` function that works inside its own monad stack (`State` and `IO`) and knows nothing about the `Reader` context:

```
type WordStatStateStack = StateT WordStatistics IO
type WordStatStack a = ReaderT Config WordStatStateStack a

collectStats :: [String] -> WordStatStateStack WordStatistics
collectStats ws = do
  lift $ print $ "Words: " ++ show ws
  mapM_ (modify.countWord) ws
  get

calculateStats :: String -> WordStatStack WordStatistics
calculateStats txt = do
  wordTokens <- tokenize txt
  lift $ collectStats wordTokens
```

Note that the `StateT` type has one more type argument that isn't visible from the type definition:

```
StateT s m a
```

That's why the `WordStatsStateStack` has this type argument too:

```
type WordStatStateStack a = StateT WordStatistics IO a
```

This is a valid record, but to use the `WordStatStateStack` in the `WordStatStack`, we should omit this type argument `a`:

```
type WordStatStateStack = StateT WordStatistics IO
```

For technical reasons, partially applied type synonyms aren’t allowed in Haskell. Having the type `WordStatStateStack a` and passing it to the `WordStatStack` without the type variable `a` makes the former partially applied. This isn’t possible because it makes type inference undecidable in some cases. So you might want to consult with other resources to get a more complete understanding of what’s happening in types.