Functional Design and Architecture

v.0.8

Alexander Granin
graninas@gmail.com

Table of Contents

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.

This book will teach you a complete methodology of writing real programs in Haskell using Free Monads and other concepts of modern Functional Programming.

The book is based on 2 projects:

- Andromeda, a simulator of a SCADA system for a spaceship;
- and Hydra, a comprehensive framework for creating web and console applications.

The book was initially started by the contract with Manning Publications, in 2016. I was working on the book the whole 2016 and a part of 2017 year. I managed to finish five chapters out of 10 when Manning Publications decided to terminate the project.

I stopped writing the book for two years.

In 2019, I returned to it due to my Patreon program. To that moment, I developed my ideas and concepts much more, and tested them in real production in several companies. It turned out that my approach Hierarchical Free Monads works great and can help businesses to achieve their goals.

Unfortunately, the 2-year lag cannot go without consequences. You might find that starting from the 6th chapter I changed the style and the very project (from Andromeda to Hydra). This is because I was no longer tied by the rules of my publisher and could reevaluate my writing.

I continue working on it; 8 chapters out of 10 are ready as drafts. My plan is to finish the book in 2020, edit it and make it more solid.

Although the book is draft and not yet completely consistent, it is already the most developed methodology from all we have in Haskell. I hope you'll enjoy it and learn something new.

Best wishes,

Alexander Granin

Acknowledges

Acknowledges

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

- My wife.
- Reviewers and readers who made a lot of suggestions on how to improve the book.
- Manning Publications for a big work made for this book.
- All my Patrons and supporters.

My top supporters:

- Special thanks to Vimal Kumar, my first top Patreon supporter who believed in this project and who believed in me. He made a lot for me and this book.
- Special thanks to Victor Savkov, my second top Patreon supporter who made a big contribution to it.
- Special thanks to Grammarly, my third top Patreon supporter. It's so cool to get this great service as a sponsor!

Also, special thanks to these Patrons who made a significant contribution to the project:

- Jon Fast
- Florian Pieper

<this page left blank intentionally>

1

What is software design?

Links

- My email
- Book page
- Patreon campaign
- Paypal donations

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 by physical math, at least to some degree. The deeper we look into space, time and matter, the more complex mathematical formulas become, which we need to use in order to explain the facts we observe. Finding better abstractions for natural phenomenons gives us the ability to predict the behavior the systems involved. We can build wiser and more intelligent things which can change everything around us from life comfort, culture and technology to the way we think. Homo Sapiens went a long way to the present time climbing the ladder of progress.

If you think about it, you will see that the fact we can describe the Universe using the mathematical language is not so obvious. There is no intrinsic reason why our world should obey the laws developed by physicists and verified by many natural experiments. However, it is true: given the same conditions, you can expect the same results. 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 we are talking about the Universe in a programming book. Besides the fact that it is an intriguing start for any text, it is also a good metaphor for the central theme of this book: functional programming in large. The “Functional Design and Architecture” book presents you the most interesting ideas we have discovered in functional programming about software design and architecture. You may be asking why one should break the status quo going far from plain old techniques the imperative world has 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 there are problems much easier to approach within the functional paradigm. Moreover I could argue that functional paradigm is no doubt as deserving as others. But these are just words, not that convincing, being lacking of strong facts.

There will be facts in further text. But there is a simple and great motivation to read this book. Perhaps, the main reason why I argue for functional programming is that it brings a lot of fun to all aspects of development, including the hardest of them. You have probably heard that parallel and concurrent code is where functional approaches shine. That's true, but not only. In fact, the real power of functional programming is in the ability to make complicated things much simpler and more enjoyable because functional tools are highly consistent and elaborated thanks to theirs mathematical nature. Counting this, many problems you might be facing in imperative programming are smoothed or even eliminated in functional programming. Certainly, FP has its own problems and dark places, but learning new ideas is always profitable because it gives you more chances to find a better techniques or ways of reasoning.

So you are probably looking for the new sight to what you know about software design. While reading this book, you'll be constructing a single project for controlling and simulating spaceships of a simple kind. This field is known as “supervisory control and data acquisition software” (SCADA), and certainly it's rather big and rich. You don't have to be proficient in it, because I'll be giving you all the necessary information to solve this task. Writing such a

software requires many concepts to involve, so we'll use this sample to show different sides of functional programming.

In this chapter you will find a definition of software design, a description of software complexity, and an overview of known practices. Terms we introduce in this chapter will show there are common approaches that you may be using already, and if not, you will 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 (FP) is better than object-oriented programming (OOP) and when it's not. We will discuss the pros and cons of traditional design methodologies, and then we will see how to build our own.

1. Software design

When constructing programs, we want to obey some requirements in order to make 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 over complicated code. Sooner or later we will get stuck and fail to meet the requirements. It seems there is a general law of code complexity that symmetrically reflects the phenomenon of entropy:

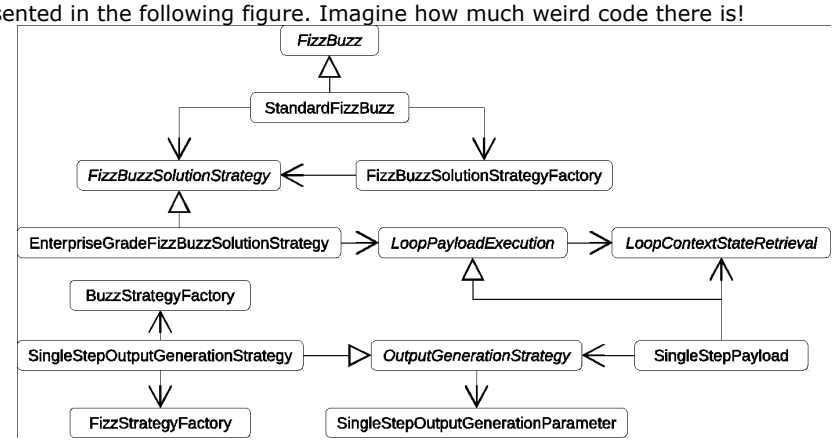
> Any big, complex system tends to become bigger and more complex.

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 make it even bigger and more complex.

Software complexity is the main problem developers deal with. Fortunately, developers have found many techniques that help decrease the acuteness of the problem. To keep a big program maintainable, correct, and clear, we have to structure it in a particular way. First, the behavior of the system 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 may 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: there is a project called "FizzBuzzEnterpriseEdition" with an enormous number of Java classes to solve the classic problem FizzBuzz. A small portion of the classes, interfaces, and dependencies is presented in the following figure. Imagine how much weird code there is!



So is going functional means you're guaranteed to write simple and maintainable code? No, as many tools, functional programming can be dangerous when used incorrectly. Consider an evidence: in the following paper, the same problem FizzBuzz is solved in a functional but yet mind-blowing manner: "FizzBuzz in Haskell by Embedding a Domain-Specific Language". 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 speak about it now.

1.1. Requirements, goals, 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 they need some spaceship management software. What a wonderful career opportunity for you! You decide to contribute to this project. Finally, when some details have been discussed, 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 1.1.

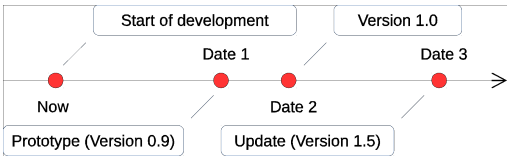


Figure 1.1 Roadmap of the development process.

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 requirements to functionality of the application. In other words, functional requirements describe a full set of things the application should do to allow its users complete their tasks.

> **DEFINITION** *Nonfunctional requirements* are the requirements to the general properties of the application: 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 require 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 the expectations of your customer: 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-upon 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 full satisfaction of each factor is your primary meta-goal, often it is an unattainable ideal in our imperfect world. This may sound fatalistic, but actually it 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 some sort of radar chart (see figure 1.2).

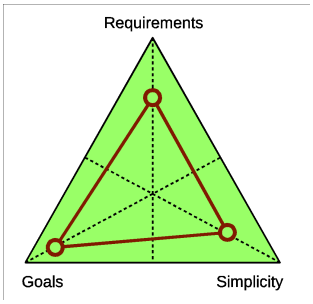


Figure 1.2 Compromising between simplicity, goals, and requirements. The ideal situation is described by the outer triangle. The inner triangle reflects the current situation.

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 typical risks 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 have finished a system that is able to 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 will 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 have 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 we 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 great mental shift.

> **NOTE** To better understand the difference between easiness and simplicity, I recommend watching the talk "Simple Made Easy" (another name is "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 also you 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.

1.2. Defining software design

You are meditating over the documentation. After a while you are ending up with the set of diagrams. Those 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 1.3.

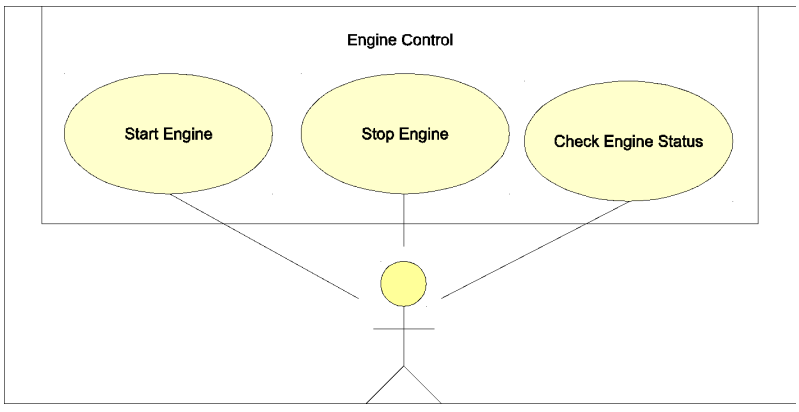


Figure 1.3 Use case diagram for the engine control subsystem.

> **NOTE** Use case diagrams are a part of the Unified Modeling Language, which is primarily used for object-oriented design. But looking to 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 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. Protocols of communication 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 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 of components' manufacturers makes the API too messy. That 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 domain-specific language.

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

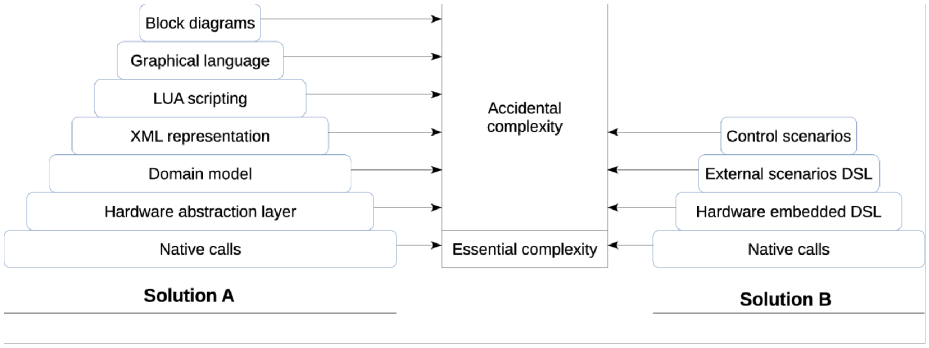


Figure 1.4: Accidental and essential complexity of two solutions.

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 best practices and patterns of software design: our predecessors have already encountered such problems and invented handy solutions that we may 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 1.5.

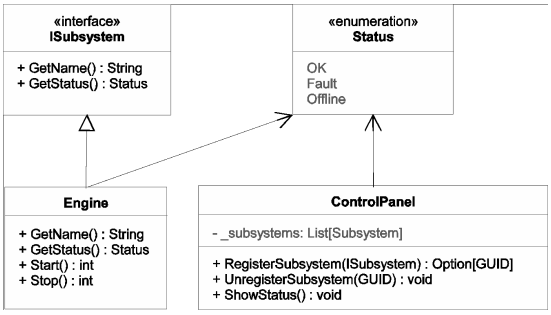


Figure 1.5 OOD class diagram for the engine control subsystem.

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 was widely used in OOD recently. Class diagrams help OO developers to 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 FP — does that really mean we can't use UML diagrams? We will 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 OO 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 only take a brief tour about 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.

1.3. 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    // defective device!
      server.connection.send("temperature", "T-201A", corrected)
```

```
    }
}
```

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 of 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 and therefore it has low cohesion. Finally it's bad because the logic we embedded into this class became untestable without accessing these subsystems.

Solving these problems requires introducing new levels of abstraction. You need interfaces to hide native functions, side effects, and the transformation algorithm, to have these responsibilities separated from each other. 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 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 1.6.

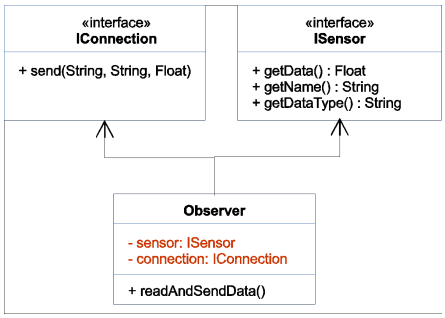


Figure 1.6 Class diagram of listing 1.2.

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              #A

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

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

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

#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

We call the functions `read` and `send` *impure*. These are the functions to work with the native device and remote server. The problem here is finding a straightforward approach to deal with side effects. There are good solutions in the object-oriented world that help to keep code loosely coupled. The functional paradigm also 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 work with the DSL only and its dependency on native calls will be eliminated. Then we have two options: first, we can use a native translator for the DSL that converts high-level commands to native functions; and second, we can test our scenario separately by inventing of some testing interpreter. The sample of how it can be done is presented in listing 1.4. The DSL `ActionDsl` shown there 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)          #A
  | Transform (a -> Float) a (TransformF a)           #A
  | Correct (Float -> Float) Float (TransformF a)     #A
  | Send DataType DeviceName Float

transform (T.Kelvin  v) = v - 273.15    #B
transform (T.Celsius v) = v             #B
correction v = v - 12.5                 #B

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

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

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

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

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. We've said that no one concept gives you a silver bullet, but the functional paradigm seems to be a really, really good try. We will discuss it more in the following chapters.

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. Separation of parts from each other making 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 OO languages like Scala, C#, or Java are a form of this principle too. An OO 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.".(Wikipedia)

Passing the implementation behind the interface to the client code is known as *Inversion of Control (IoC)*. With Inversion of Control 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 1.7.

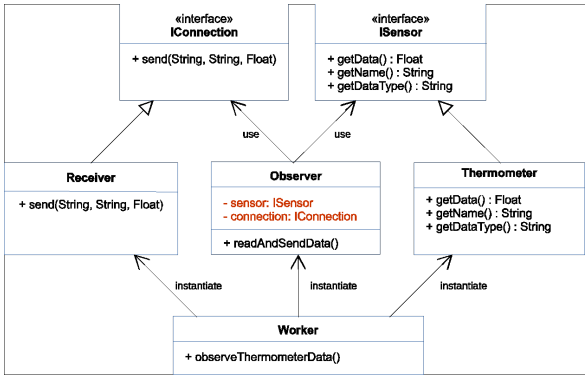


Figure 1.7 Full class diagram of listings 1.2 and 1.5.

Now, we are going to do one more simple step. Usually you have a bunch of OO interfaces related to a few aspects of the domain. To keep your code well organized and maintainable, you may want to group them 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, Inversion of Control, 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: is the functional paradigm good for software design too? What are the principles of functional design, and how they are related to OO principles? For example, can we have interfaces in functional code? Yes, we can. Does that mean that we have Inversion of Control 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, FP 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 concept is our domain model is like, 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, its nature. That allows us to generalize the properties to functional idioms (for example, functor, monad, 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 just 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 FP 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 the chapter we will discuss software design from three points of view: imperative, object-oriented, and finally functional. We want to understand the relations between these paradigms better in order to be able to operate by the terms consciously.

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; and second, the widely spread object oriented paradigm is imperative under the hood. The term "*imperative programming*" denotes a program control flow where 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 such imperative techniques 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 with such methods applied like unsafe type casting, variables destructive mutation, using side effects to get a 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 OO patterns? It seems we haven't. Despite the fact that object-oriented design is much younger than bare imperative design, it has been described many times better. But if you ask system-level developers about the design of imperative code, they will probably name such techniques as 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: unstable private implementation and 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 is changed 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 kernel/drivers/staging/unisys/include/procobjecttree.h) 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 dangerous and risky. Developers have less motivation to express their ideas in design because they always have a short path to theirs goal. It's possible to hack something in code: reinterpret the type of a value, cast a pointer even though there are 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 that, the benefits you gain like low risks and good quality of code can be much more important than inconveniences emerged. Let's see how object-oriented design deals with it.

3. Object-oriented design

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

3.1. Object-oriented design patterns

What is object-oriented design? In short, OOD 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. All the object-oriented design 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 these case diagrams and class diagrams already, so let's see one more usage of the latter. Figure 1.8 shows the Adapter design pattern structure as it is presented in the classic "Gang of Four" book.

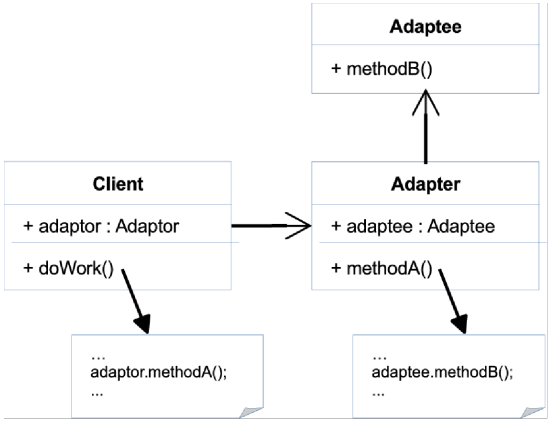


Figure 1.8 The Adapter design pattern.

> **NOTE** You can find hundreds of books describing patterns in application to almost any object-oriented language we have. The largest and most influential work is the book *Design Patterns* by E. Gamma, R. Helm, R. Johnson, and J. 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 of common design problems in object-oriented languages.

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

> **NOTE** As a 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 with using of 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 trying to comprehend object-oriented patterns from functional perspective. This is a really hot topic today.

So, we can define OO 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 of software design, as you might be thinking. Note that all the patterns

use interfaces and Inversion of Control. Those are the key things: IoC, Modularity, and Interfaces. And, certainly, design principles.

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 API of the lamps, 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 the 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.

Listring 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 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 can happen not only 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 to think that something is badly implemented. The contracts have to be followed by implementation, otherwise it's becomes really hard to predict the behavior of our program. This is why so called *contract programming* was introduced. It brings some special tools into software design which allow to express the contracts explicitly and to check whether the implementation code violates these contracts or it's fine.

Let's show how the contract violation occurs in a class diagram (figure 1.9).

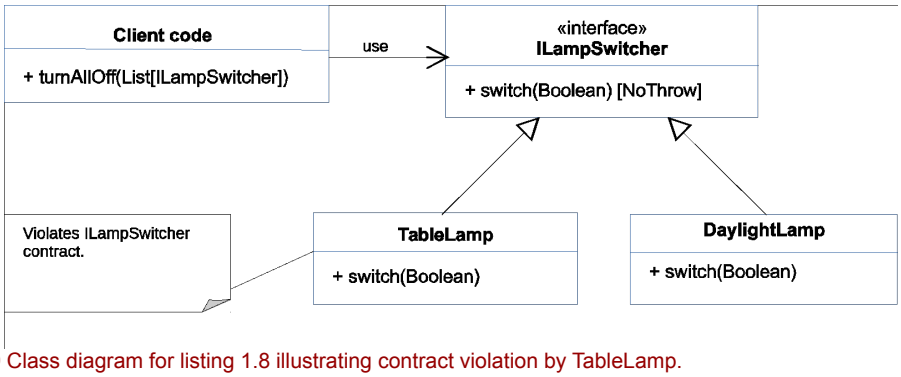


Figure 1.9 Class diagram for listing 1.8 illustrating contract violation by TableLamp.

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 OO 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 is 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. The Liskov Substitution Principle states that if you use `ILampSwitcher`, then substitution of `ILampSwitcher` by the concrete object of `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 correctness of the program. `TableLamp` obviously violates this principle as it throws an unexpected exception and breaks the client code.

Besides the LSP, SOLID contains four more principles of object-oriented programming. 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, we only note that DIP, ISP, and SRP correspond to the ideas we mentioned in section 1.1.4: Modularity, Interfaces, and Inversion of Control, respectively. That's why SOLID principles are applicable to imperative and functional design too, and that's 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 are not 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)".

3.3. Shifting to functional programming

We finished our discussion of imperative design by talking about freedom. Although object-oriented programming introduces new ways to express ideas in design (classes, object-oriented interfaces, encapsulation), 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 it's hard sometime to understand why the program crashes and where is the bug. In imperative and object-oriented programming it's common to debug a program step-by-step with a debugger. Sometimes this can be the only debugging technique able to give you an answer what's happening in the program.

But lowering the need in a debugger (and thus limiting ourselves by other debug techniques) has a good consequences for the design of the program. 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 to be lazy in his intentions. He tends to not to 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 ignore the idea of software design most likely. Unfortunately, it's rather hard to maintain the code without a design. It often looks like a mind dump of a developer and has a significantly increased accidental complexity.

So how could the step-by-step debugging be replaced?

A greater help to a developer in this will be a compiler. We can teach it 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 object correctly, in opposite 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, OOP is stuck in its evolution, and that's why functional programming is waiting for us. In functional programming we have nice 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.

4. Functional declarative design

The first functional language was born in 1958, when John McCarthy invented Lisp. For 50 years FP lived in academia, with functional languages primarily used in scientific research and small niches of business. With Haskell, FP was significantly rethought. Be defined in 1990, Haskell was intended to research the idea of laziness and issues of strong type systems in programming languages. But functional idioms and highly mathematical and abstract concepts it also introduced in the '90s and early 2000s, became a visit card of the whole functional paradigm (we mean, of course, monads). No one imagined it 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 info 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 of the mainstream languages have adopted such FP techniques 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 having it adopted incorrectly may get a project into chaos. But still, FRP shows a good potential and attracts more interest nowadays.
- *LINQ* in C#, *streams* in Java and even *ranges* in C++ are an example of a functional approach to data processing.
- *Monads*. This concept deserves its own mentioning because it is able to reduce the complexity of some code, for instance, eliminate callback hell or make parsers quite handy.
- *Lenses*. This is an idea of data structures being handled in a combinatorial way without knowing about its internals that much.

- *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 do not produce an extra accidental complexity. In opposite, raw threads and manual synchronization with mutexes, semaphores etc., usually turn the code into unmanageable mind blowing puzzle.

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

This book provides the answer. We are here to create an absolutely 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 we say "declarative" in our definition of FDD: we will put imperative and OO paradigms away and will strive to achieve declarative thinking. One can 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. Learning FDD, we will dive into genuine idiomatic functional code.

Let's sow the seeds of FDD.

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. There is one fact that shared mutable state is the main cause of bugs in parallel or concurrent code. In FP, we restrict our freedom by prohibiting data mutations and shared state, and 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 `IO Ref` 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) : 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)
}
```

And also it is convenient to support a pyramidal functional code: it always has a clear evaluation flow like the diagram in figure 1.10 shows.

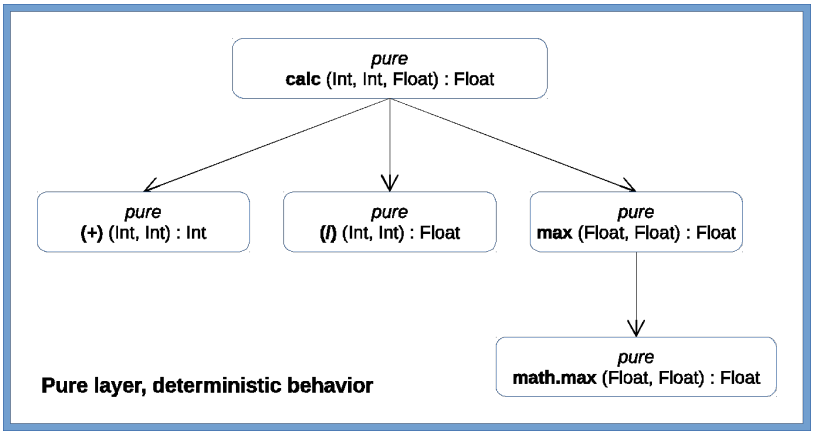


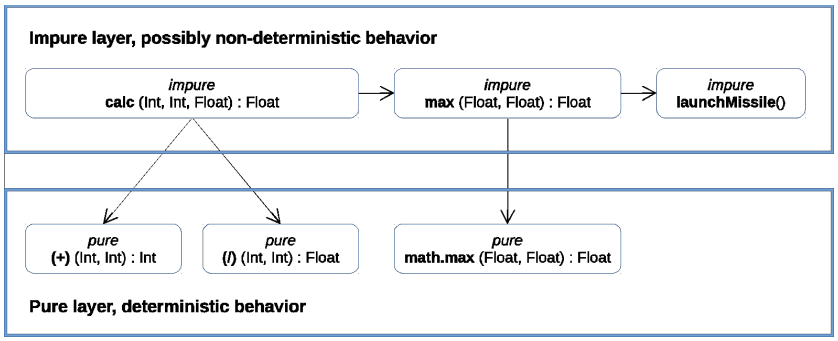
Figure 1.10 Pure pyramidal functional code.

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. The most of languages are lacking this feature and allow a developer to do 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. Like this:

```
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 1.11.

Figure 1.11 Impure pyramidal functional code.

Note there are no ways 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 of the 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 it by collecting the data in the impure layer and then call pure functions. But what the difference you might ask. Simple: on the impure layer, you are allowed to use destructive assignment and mutable variables. So you might want to collect the data into a mutable array. After that you'd better pass this data into the pure layer. There are very few cases when staying on the impure layer is preferrable, but most of the cases are fine to this scheme.

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 domain-specific language 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 = ...              #A

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

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 these two layers aren't interleaving 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 safe and unsafe world. 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 talk, 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) will be 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 isolation of 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.

4.2. Strong static type systems in FDD

We have come to the end of our path of self-restraint. We said that expressing determinism can help in reasoning about code. But what exactly does that mean? In FDD it means that we define deterministic behavior through the type of functions. 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 and 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 effect? 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 that information just from the types. Actually, we just

---

[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.

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 we will discuss a bit of type-level design too.

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 the ready-to-use solutions for our tasks and adopting of some design principles to lower risks. Functional programming is not something 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! Well, 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 of the certain type of problems. A pattern is an auxiliary compound mechanism that helps to solve a problem in abstract, generic way. Design patterns describe how the system *should* work. In particular, OO design patterns address objects and mutable interaction between them. An OO design pattern is constructed by using classes, interfaces, inheritance, and encapsulation.

> **DEFINITION** A *functional idiom* is the internal solution of the certain types of problems. It addresses the natural properties of the domain and immutable transformations of that 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 our introduction to FDD is a bit abstract and lacking in details, you're completely right. We are 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 hold onto the intrigue and 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 it can be written 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 mathematical meaning of the results. We do not 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 tho alternatives are `Nothing` and `Just`) which is a monadic thing. In the do-block, we have used the generic properties 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 this is what can be called “functional thinking”.

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

5. Summary

We learned a lot in this chapter. We talked about software design, but only in short, because it is a huge field of knowledge. In addition to object-oriented design, we introduced functional declarative design 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 done, and 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 OO patterns or reveal functional idioms. Any well-described solution helps us to represent behavior in a better, shorter, clearer way, and so to keep the code maintainable.

Functional declarative design 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 give many insights to object-oriented developers in their work.

We learned also about general design principles:

- Modularity
- Inversion of control
- Interfaces

The implementation of these principles may vary in OOP and FP, 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 pursuit the main goal of software design, which is to keep accidental software complexity at a low level.

Now we are ready to design something using FDD.

2.

Architecture of the application

Links

- My email
- Book page
- Patreon campaign
- Paypal donations

This chapter covers:

- Approaches to collecting of requirements
- Top-down design of architecture
- Model-driven development of domain-specific languages

We believe that the freedom to make choices really exists. This means we control our decisions, and we have to take responsibility for the results of our choices. In the opposite scenario — in an imaginary universe where everything was predestined, including all of our choices — we would not be responsible because we would not be able to change any of these decisions. It would be boring if all the events were defined 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 predefined 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 be repeated again.

We also believe we aren't robots here, in our cozy world. We only see the advantages of the fully deterministic model when we design software. Furthermore, in our programs we create systems with the behavior 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; then the timeline containing events reflects the very natural notion of event-driven design; then there are cycles, scripts, constructing and destructing 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 can hit us in the future. Consider the diagram in figure 2.1.

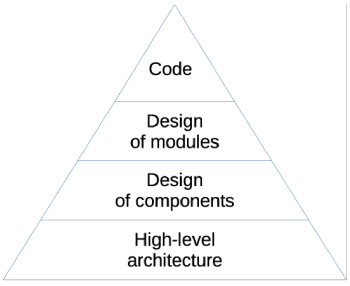


Figure 2.1 Levels of software design.

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. Sometimes it is unclear whether the architectural solution is good enough or not. To understand that, 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 will 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. Let it be called "Andromeda". This project aims to develop a spaceship control and simulation software that engineers of Space Z corporation may will use in theirs work. It has many parts and components all having own requirements and desired properties. Here you'll be collecting requirements and designing a high-level architecture of the project. To 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 how to reason about application from functional developer's point of view, and you'll have tools in your toolbox - design diagrams that are very useful to carve 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 will 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 elements of the interface. 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 composing 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. You may suppose that the application has a monolithic structure with bare WinAPI calls for drawing and controlling GUI, where the logic is encoded as transform functions over two-dimensional array of predefined values ("Bomb", "Blank" and numbers from 1 to 8). Figure 2.2 shows this structure:

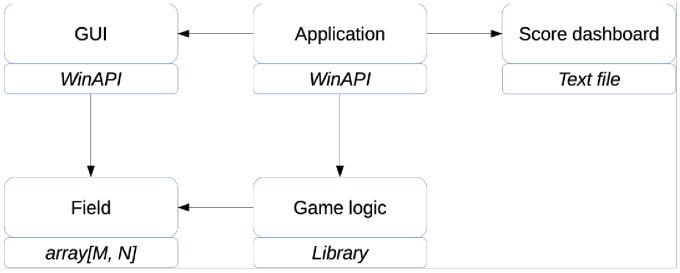


Figure 2.2 Architecture of the Minesweeper game.

Another example is Minecraft. It will be a little more challenging to say what the game code looks like as it is more complex, but actually we can make some assumptions, because we 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 developers of Minecraft, 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. It's easy to see we don't get into the implementation details; in contrast, we describe the top level of code structure only. Let's visualize it in a diagram (figure 2.3).

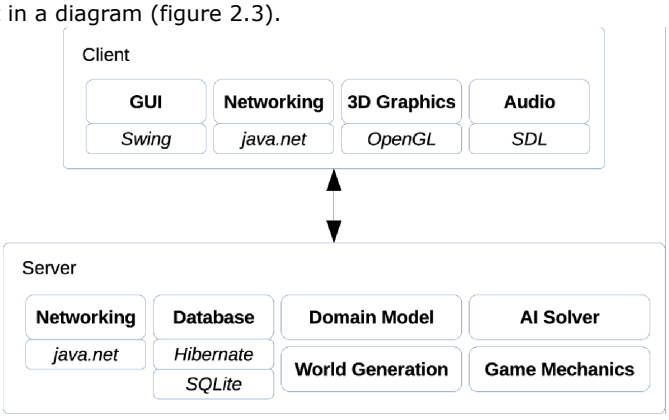


Figure 2.3 Possible architecture of Minecraft.

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

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

When newcomers join our team, we give them a diagram of the architecture. They see the big picture and can understand our project in a short time. That's how architecture diagrams solve the problem of knowledge transferring. 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 architecture of the application before we start designing low-level code and structure. Good architecture allows the independent parts to be reworked 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 to obey the requirements to the software (see "Software architecture" article in Wikipedia). By following these practices it's possible to make a raw sketch of architecture, but certainly every single project has its own unique characteristics and so every architecture will do.

In this chapter we will 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 an intuition needed to take architectural decisions.

2.1.1. Software architecture in FDD

To some degree, you may consider that functions (either first-class or higher-order) is the only abstraction all functional languages have in common. Combining functions is not a big deal: you just apply one to another while building a pyramid of functions. Not so much, you will say? But this is a quite sharp and robust tool that has expressiveness as powerful as 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 features in order to achieve a higher abstraction that solves the problem. However the taller the house, the more likely the abstractions the developer ends with will be leaking in many unexpected scenarios.

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

> **DEFINITION** A leaky abstraction - is the abstraction that tends to reveal the implementation details instead of hiding them. The leaky abstraction is solving a part of the problem and resisting to changes that are supposed to solve the whole problem. Also, a leaky abstraction reduces less complexity than it brings to code.

What is worse: to build a pyramid of functions or a card house of language features? Can 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 language specification. In opposite, all you need to know to combine functions is their type definitions. Functions in functional language are based on the strict mathematical foundation - lambda calculus, and this makes functional abstractions less "leakable" just because math is well-constructed field.

Usual concepts in functional languages include: lambdas, regular functions, higher-order functions, immutability, purity, currying, composition, separation of model and behavior, 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 the 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 a few functional patterns which are intended to solve a particular architectural problem in a functional manner. In practice, they do can leak, but usually this wouldn't be the case if they are 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:

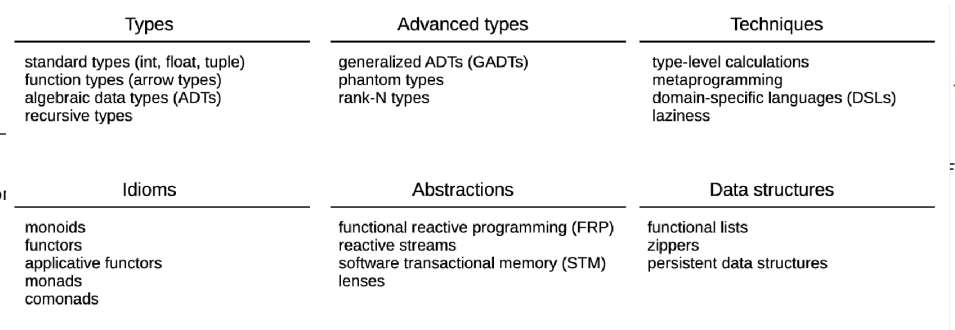


Figure 2.4 Classification of functional ideas.azc

[2] For example, the laws of mo

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

- *Standard types: 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 FP, permeates many functional idioms also. Interesting, without recursion in the level of types it would be hard to represent, for example, list data type in static type system.*
- *Meta-programming.* Meta-programming in functional languages is often an abstract syntax tree (AST) modification of the code for reducing of boilerplate. It may be a powerful tool for design too.
- *Domain-specific languages (DSLs)*. A domain-specific language 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 maintaining to comply with the current requirements.
- *Algebraic data types (ADTs)*. An algebraic data type 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*). Algebraic data types are widely used in functional programming for designing domain models, DSLs, or any kind of data structures.
- *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 also are able to 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 of inventing monads irrespective of his intentions to do or not to do that[3].
- *Nonstandard complex types: existential types, phantom types, 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 make it possible to 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 with 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… 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. Laziness helps to design code in a manner when you write a code that looks like it transforms a big structure, but it’s actually not. After that, you can compose such a “heavy” transformations, but in fact, when they are finally going to evaluation, only a small part of chained t transformations will be evaluated: no more, no less than you need. will be evaluated.

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

- *Functional reactive programming (FRP)*. FRP is a style of reactive programming merged with functional ideas. In FRP, the notion of time-varying values is introduced. 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. Then any logic can be bound to the signal as a reaction. By having a reactive model composed of signals (which often is called a *reactive network*) and actions, it is 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 so the events that have occurred) can be

---

3

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.

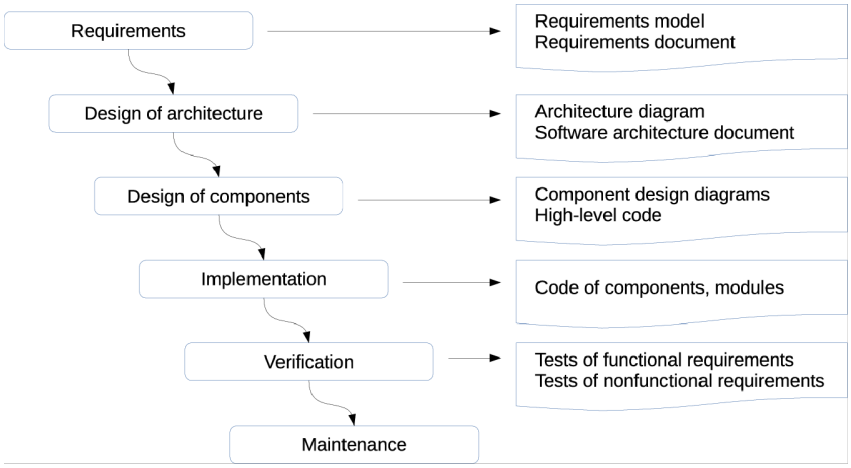
- *Software transactional memory (STM)*. STM represents a model of concurrent state with safe transactions for its modification. Unlike concurrent data structures, STM represents a combinatorial approach, so combining models gives another STM model. One of the biggest advantages of STM is that it can 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 all code turns into after rolling and unrolling of a complex structure can dramatically impact the readability. Lenses help to 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. We have reviewed the most important of them, but we will leave detailed descriptions and use cases for other chapters. Unfortunately, this book is too small 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 that follow one another until we leading up to a ready application. We started with requirements analysis. This is the process where 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 phase of software design. 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 will be an idealization of the development process. It has a well-known name: *waterfall model*. In real life, we will 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. By this exactly reason we wouldn't follow the waterfall model in this book. We'll rather adopt an iterative and incremental models with cycles than this rigid single-directional one, however you may consider the waterfall model as a reference what activities exist in the software development process.

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

The better models allow you 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 you attack the problem in code, you need to collect requirements and design the architecture and separate subsystems. Make yourself comfortable, and prepare a pencil and paper: we'll be drawing some diagrams to better understand a domain and design options we have.

> **NOTE** Before we continue, we should agree on that irrespective the methodology this book suggests, it does not matter what diagrams you'll be creating. Drawing pictures which illustrate the design of a code seems essential to software developers who wants to describe their ideas to others well. And there is much more informal ways of structuring the diagrams, than formal ones (like UML or like this book suggests). However in this book we'll be not only design an application in diagrams but also we'll be collecting and analysing 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 TDD methodology can be used to write an FP code pretty easy, because testability is one of the features where FP shines. But designing the architecture and parts of the application with TDD requires a bit more knowledge of different approaches. Test-driven design is impossible without the ability to separate an application into independent parts and thus we'll need something for our three key techniques: Interfaces, Inversion of Control and Modularity. The next chapters are aiming to cover the most suitable implementations of them in FP.

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

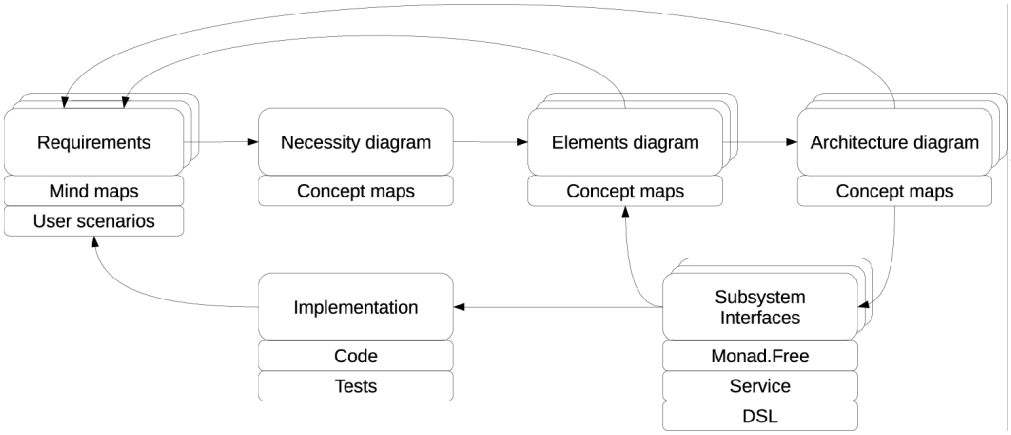


Figure 2.6 Iterative architecture design process in FDD.

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

- *Requirements*. Every development starts from collecting requirements. In FDD, it's recommended to keep requirements model in mind maps and user scenarios.
- *Necessity diagram*. This is a concept map having main parts of the application. It shows the front of the future work and basic dependencies between blocks. It can be also just a list of the big parts of the application.
- *Elements diagram*. This is a set of concept maps that fixate results of brainstorming about application architecture. Elements diagram may be unstructured and rough but it helps you to estimate every design approach you may think about.
- *Architecture diagram*. This kind of diagrams represents the architecture of an application in a structured way. It keeps your design decisions and shows different subsystems with theirs 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 the lifetime of it, 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 revealing and requirements are changing.

2.1.3. Collecting requirements

One part of the Andromeda control software is the simulator. With it, engineer 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 requirements of SCADA, 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 (FPGA, 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** 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 the project participants.

While creating the requirements model, you may feel like Sherlock Holmes. You are 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 are getting are often very unclear; some of them will be contrary to each other and even to common sense. This is normal, because we are humans and we make mistakes. Don't hesitate to ask questions when you need clarification. What is 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 formal, a set of well-formed requirements available to both developers and the customer. The most important ways to gather requirements are listed below.

- *Questions and answers*. This is the best method of investigating what the customer wants to be done. Questions can be various: 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 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 we 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 so much, because they don't allow us to reveal functional properties of the domain. In order to try our hands, though, we are going to 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 are 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 will 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 looks 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 standardization of our thoughts; we just extract associations regarding a concrete notion and write them down as part of the diagram. Then we can reorganize it if we want more order. You will see that mind maps are very useful in software analysis, and they help to 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 will be able to create a complex of nice-looking maps connected together 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, there are online tools, if you are a advocate of web applications. But there is nothing better than a good old pencil and piece of paper yet!

You may use any of these four tools for domain analyzing. “Answers and questions”, “User stories” and “Use case diagrams” are well-known, so we'll skip explanations here to have a deeper look to mind maps. As I said, associative diagrams seem to be the most suitable tool for brainstorming, but not only. You'll see how information can be organized in the mind map model where each diagram reflects a part of the domain with the 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 in the end of this chapter is build with a huge help of brainstorming. Moreover, you'll learn some design diagrams that force you to think associatively, and, I believe, lead you to the functional solutions of the architecture problems. But first you need to prepare to this journey, because you should understand your domain before you'll be able to 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 the 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 is short but not that short to miss important points able to affect the design.

So what is mind map and how do you create a mind map? That's simple. 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 mind map, take a piece of blank paper, take 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 by 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:

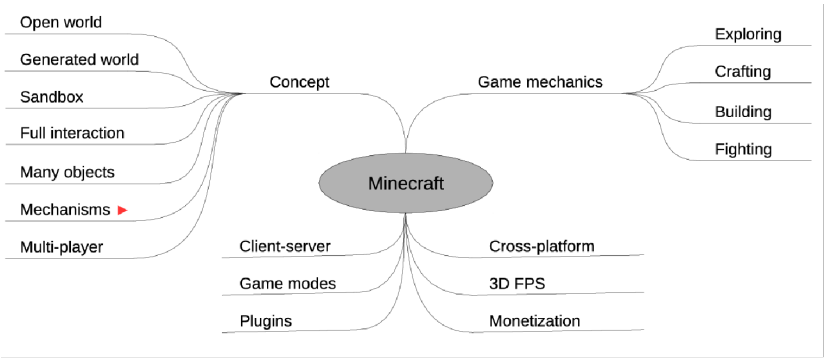


Figure 2.7 Level 1 mind map.

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 have 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 was simple yet powerful enough to find a hidden pitfalls. Mind maps are trees, but cross-relations aren't prohibited — connect elements by secondary links if you see it's necessary and helpful. Unlike with UML diagrams, there are no rigorous rules for creating good and bad mind maps, and you are not 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 that's the key to the convenience of this knowledge system. All these properties of the mind map make this tool very popular for domain analysis. And it's also very nice that in creating a mind map we can connect functional ideas and idioms with the elements and notions it contains.

The diagram in figure 2.7 doesn’t contain that much information, only large themes. We need to go deeper. How deep, - you decide by yourself. If you see it's enough to only list key functionalities, you stop on the level one, the

level of components and general requirements. Or you may 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 by user scenarios or use cases. See figures 2.8 and 2.9:

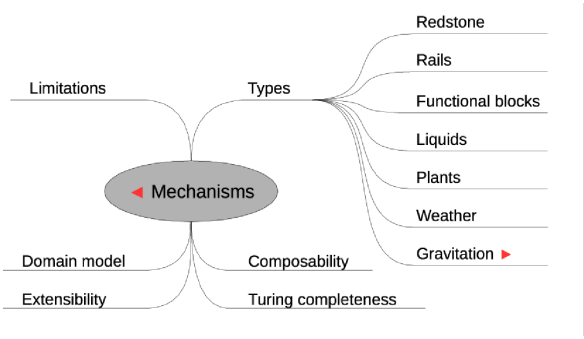


Figure 2.8 Level 2 mind map.

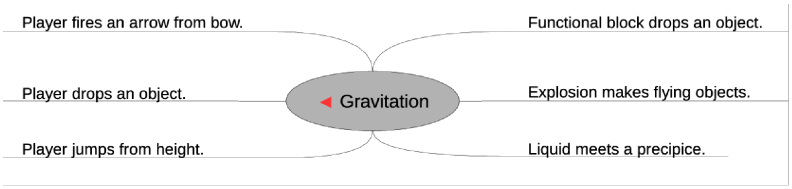


Figure 2.9 Level 3 mind map.

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, it's for astronauts and engineers. We have also collected some functional and nonfunctional requirements already. Let's organize this knowledge from the perspective of development, not of the theory of SCADA. Figure 2.10 illustrates this information in the level one mind map.

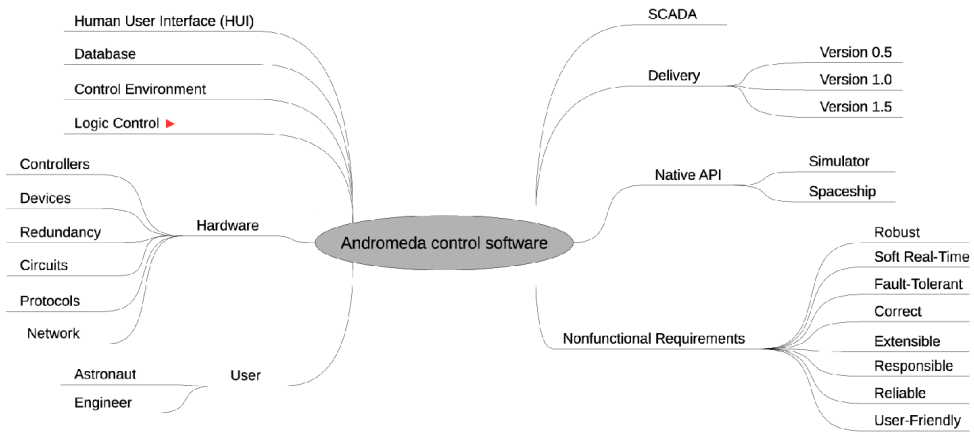


Figure 2.10 Andromeda control software mind map.

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

What is logic control? In SCADA, it's a subsystem that evaluates control of hardware: it sends commands, reads measurements, and calculates parameters. A programmable autonomous logic then utilizes the operational data to continuously correct of 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 domain-specific language. 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 are

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.

The "Logic Control" diagram of level two is shown in figure 2.11.

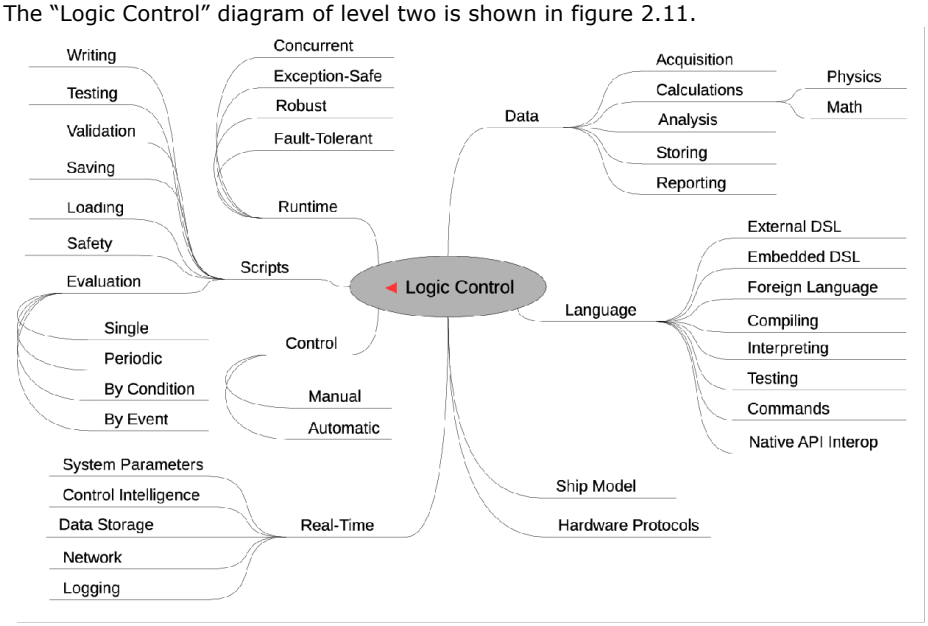


Figure 2.11 Logic Control mind map.

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 formal language of that domain.

2.2. Architecture layering and modularization

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

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 a paradigm-agnostic, and so it's suitable for FDD too.

Another term is *module*. A module is a unit of decomposition of functionality that serves one concrete problem. Every module has an interface that 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 FP, 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 complexity of the code.

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 is 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 the services of this layer should have functional interfaces and 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 to build GUIs and other presentations of the application; for example, a console or web UI. It also handles input, from the user (mouse, keyboard), 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 is a completely pure layer. The cases where it should be impure are considered harmful.
- *Interoperability layer* — This layer was born from the idea of providing one common bus 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, reactive streams) or pipelined impure code (streams, pipelines, compositions of impure low-level functions).
- *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 the subsystems provide, and it may be pure. Scenarios of business logic may be translated to impure code that should work over the interoperability layer.

Figure 2.12 brings everything into one schematic view.

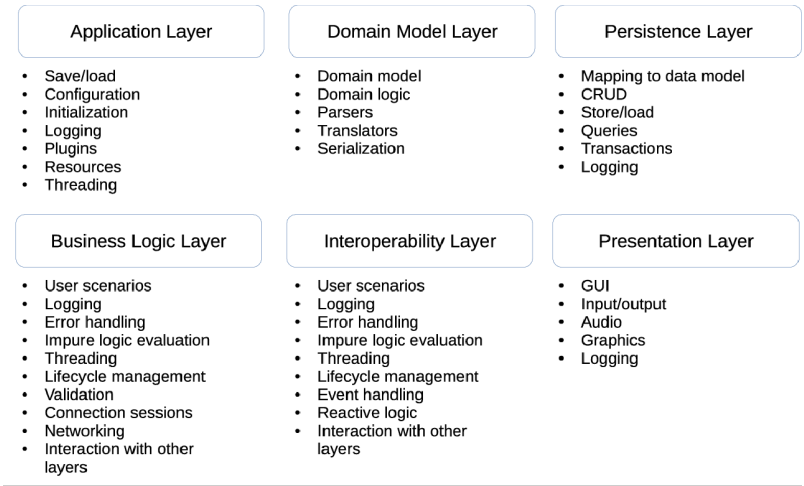


Figure 2.12 Layers: descriptions and responsibilities.

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 it 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 having the return type `IO ()` is an impure action that can be stored for further evaluation, when it's actually needed. So it becomes impure in the moment of evaluation but not on the moment of composing it 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 a simple illustration:

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 which 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 it mocks the impure subsystem the `Language` data type represents. The `run` function is the entry point to the subsystem. Being 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 Inversion of Control in functional languages.

With introducing the `Language` DSL we gained another level of abstraction over the logic. We can interpret it by an impure interpreter but also we can translate it into another DSL that serves some other goals, for example, adds a logging approach to every action. Then we 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 are 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:

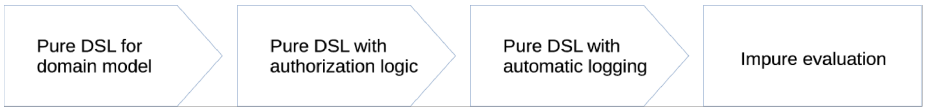


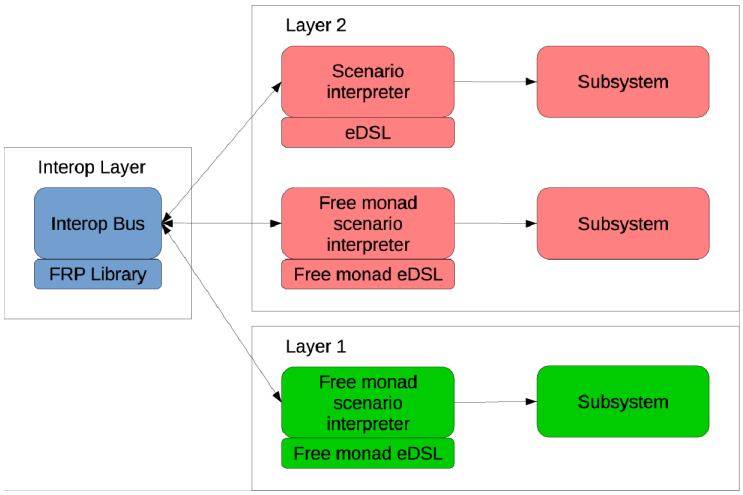
Figure 2.13 Translation from one language to another.

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 is useful for the design of DSLs. We have discussed some of the aspects of functional architecture layering, but as you can see, we have said nothing about implementation of the particular layers. We descend to this level of design in the next chapters.

2.2.2. Modularization of applications

First of all, 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, this can be achieved by introducing functional interfaces that are primarily embedded DSLs. 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 interoperability 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 by using monadic libraries and syntactic sugar like `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 you use this technique as a primary method of modularization in a functional way. In this book, we will 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 the chapter offers you a lot of practice. It also gives some more theoretical concepts we have omitted here.

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 is no graphical documentation on it but this can't deny the fact the code works. No one knows why. They don't like our children's games with colorful diagrams, but they prefer a pure code instead. The lack of colors makes them sad 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, writing the accompanying text. Unfortunately, we can forget about code itself while playing with our diagrams. When the pure code programmer has a working prototype, we only have abstract figures, arrows and pictures — nothing we can run and try. Despite designing with diagrams gives us much 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, additionally to the 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 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 the underlying ideas of the architecture? Will they have trouble using the DSLs? Can they break something? If they know your home address, do you expect they will visit you sometimes in the dead of night? How you can 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 of one separate part shouldn't lead to crashes of any of other.

- *Extensible*. The architecture should allow you to add new behaviors easily if this is required by the domain.

In this section we will 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 we are 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, and we need to implement it. Traditional examples of components include the GUI, network, database, ORM, and logger. We are 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 is the list of components extracted from figures 2.10 and 2.11.

- Logic control
- Hardware protocols
- Simulator or real spaceship
- Database
- Scripts
- Graphic user interface
- Hardware interface
- Network

Other components may reveal themselves during development, but that 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.

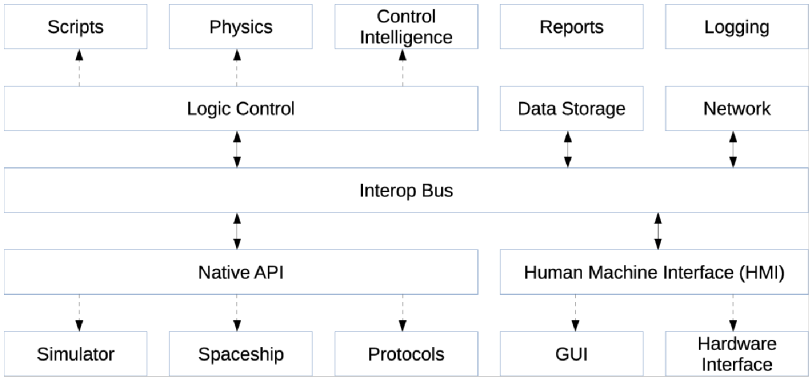


Figure 2.15 Necessity diagram of Andromeda control software.

In the middle of the diagram you see the *Interop Bus*. This was not mentioned before; where has it come from? Well, remember what we talked about in chapter 1. Simplicity and reducing of 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.

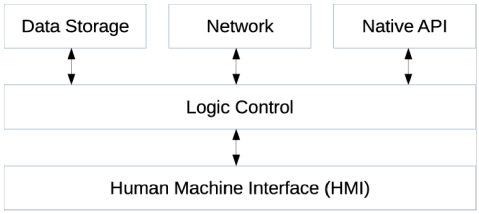


Figure 2.16 Necessity diagram without Interop Bus.

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 is called here, is an abstraction of interaction logic that is 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, message brokers — 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 is time to figure out how to do it. Again, we will 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 is 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 is 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 will be able to create a beautiful architecture diagram that will please your inner perfectionist.

As the example, I've took the block “Logic control” and expanded it like the following:

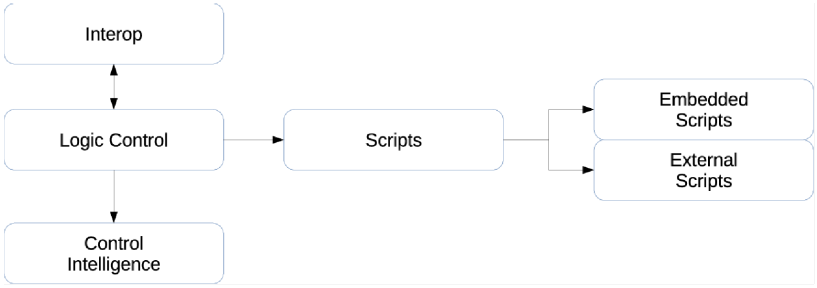


Figure 2.17 Simple elements diagram.

What elements can be in the same diagram? There are no special limitations here. If you are not 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.

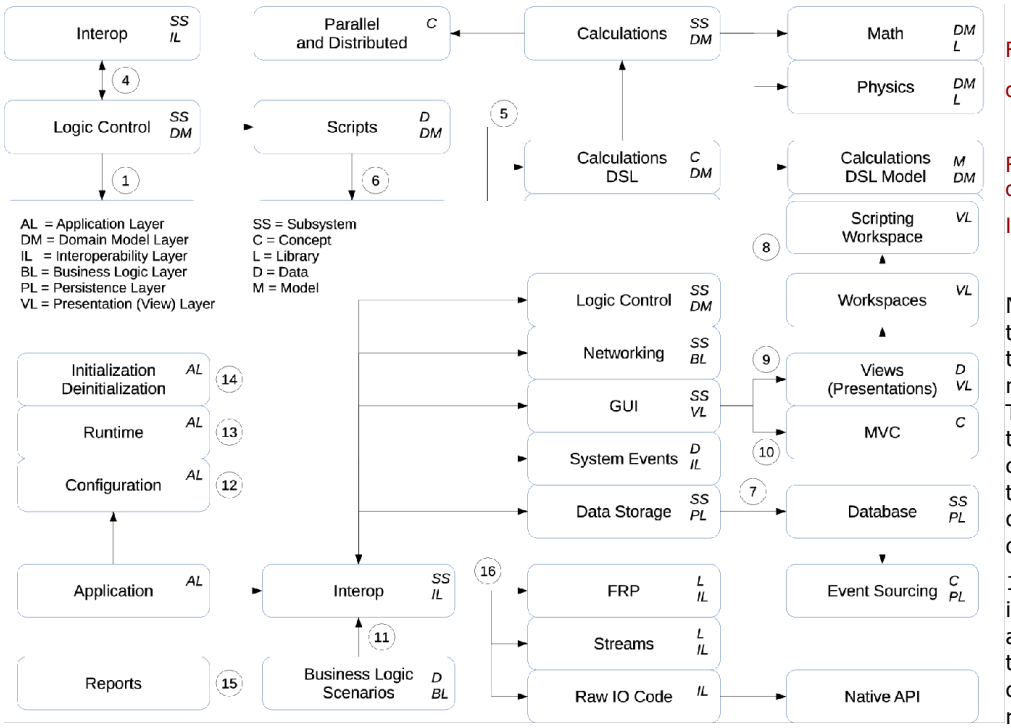


Figure 2.18 Elements diagram—Logic Control.

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 one only thing that matters: adding elements to the diagram should give you new requirements and ideas for how to organize the whole architecture of the application.

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 the iterative process is much better, and we can return to 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 between 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 also impure, because it's implementation has `Monad.IO` block, and consists of two parts: accessors to database and GUI.

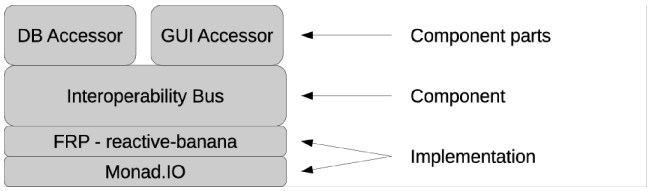


Figure 2.20 Component description block.

All components 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:

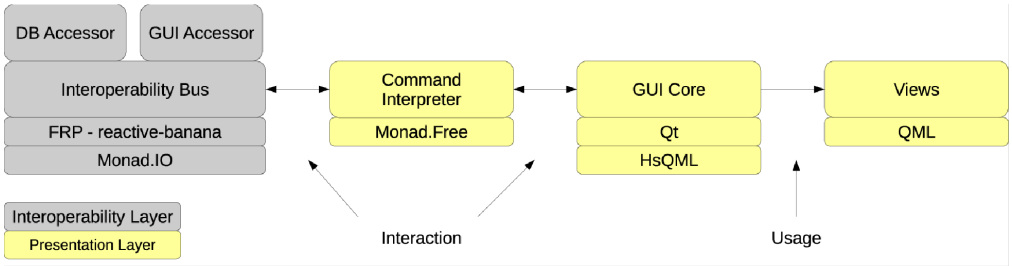


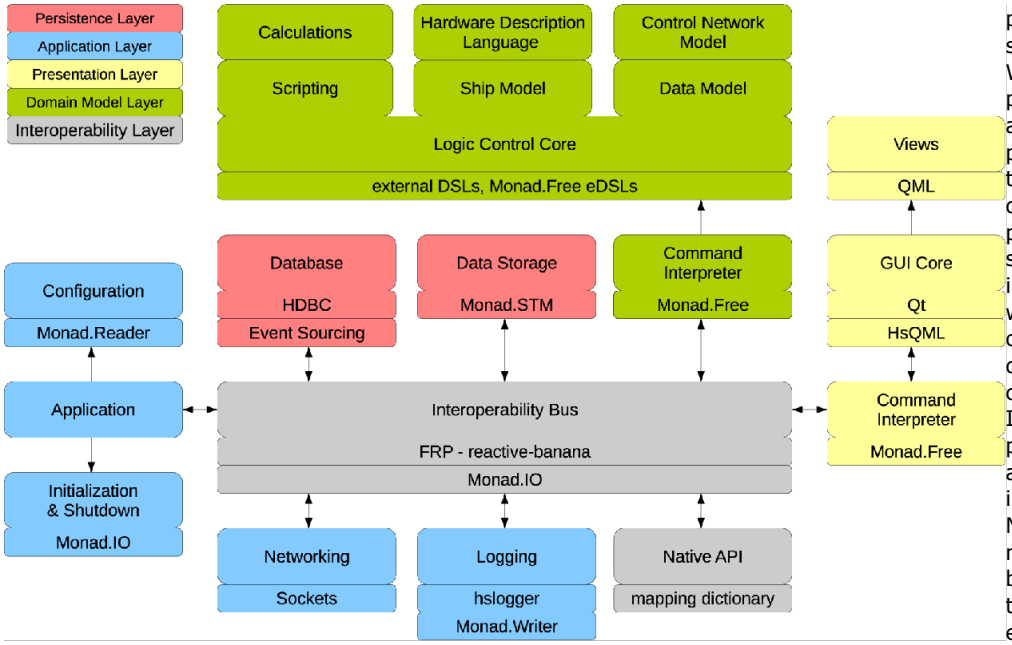
Figure 2.21 Relations between blocks.

The whole architecture diagram may tell you many important facts about the architectural decisions, but certainly it has very limited tools to describe all ideas of the application structure. So consider this diagram as a graphical roadmap for the path you are going to go. 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. Many pictures have been shown and many words spoken. We have designed the overall structure of the Andromeda control software. But don't think it's the final version of 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 "types diagrams" to types in code?

I bet your programmer's addiction requires injection of code. Mine does. In the next chapters, we will 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 have discussed are similar in OOD and FDD, but the implementation differs. First of all, architecture layers. As we know now, 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 we described purity of domain, and the advantages and disadvantages of being pure and impure. That directly maps to what we studied here. Just to revise: purity gives determinism, determinism simplifies reasoning about the code, and all together this reduces software complexity.

Another big thing we have learned about is requirements collecting. Why should we care about this? Because without requirements, we can't be sure we are doing the right things. Furthermore, the requirements model can be converted into architecture and high-level code. We have learned four models for defining requirements: questions and answers, use case diagrams, user scenarios, and, finally, mind maps. If you browse the Internet, you will find mind maps a popular tool for requirements analysis by brainstorming. We also saw how model-driven development can be used to create a domain-specific language 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 algebraic data type, 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 the architecture of our application. The process of development where 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 components of the application; 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.

3.

Subsystems and services

Links

- My email
- Book page
- Patreon campaign
- Paypal donations

This chapter covers:

- Design of modules 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 in our galaxy, the Milky Way, there was 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 the 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 and 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.

The story just told 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 is 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 by just reading the message: you need special tools to decrypt it into a human-readable form. But when we develop software we invent many small domain-specific languages 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 were discussing earlier, this helps to keep the accidental complexity low.

In this chapter we will learn some design practices for functional subsystems, communication languages, and separation of responsibility that will help us to follow such principles as Single Responsibility, Interface Segregation, and Liskov Substitution.

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 will be captured by a hurricane of meanings, a whirlwind of code, and a storm of weird concepts, but I will 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, next we should descend to the design of the components we have specified in the architecture diagrams. Now we are interested in how to delineate the responsibilities of related subsystems. We will 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 will continue to work with our sample, a SCADA-like system called Andromeda Control Software. Now 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 the state of an object in a way unexpected by the interface of a class. But a badly designed or inconsistent interface can be a source of bugs leading to invalidation of internal state. We can roughly say that encapsulation of the internal state is another type of code contract, and once it has been violated, we encounter bugs and instability. A widespread example of contract violation is unsafe mutable code that is 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 able to 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 without internal state that we may accidentally corrupt when complex logic is involved. Functional code tends to be divided into tiny pure functions, each addressing one small problem well, and 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 the code's 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 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 to add a new one, you put their code out of action. Obviously, there should be a mechanism to encapsulate internal changes and hide the actual structure of data from external users. You do it in two steps: first you introduce *smart constructors* for your algebraic data type and second you make your type to be *abstract* by placing it into *module* and providing both smart constructors and useful functions. It will be better if your language is able to export algebraic data type without its value constructors, as this forces hiding the type on language level. Many languages that have modules may either do it directly or simulate it with other language constructs. Let's deal.

> **DEFINITION** A *smart constructor* is a function that constructs a value of any 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 usage: objects in OOP and modules in FP. Modules in Haskell have syntax to control export of certain things: types, functions, type classes, algebraic data types with or without value constructors.

The next example introduces a widespread pattern of encapsulation — *smart constructors* for the algebraic data type `Value`, that it still visible to 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, 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 will see 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 is the difference? Just read the names of the modules, and you'll see `Data.Map.Strict`, `Data.Map.Lazy`, `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 of 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 the Interface Segregation Principle (ISP). It's easy to see that the Single Responsibility Principle (SRP) is also applicable to modules: a module should have a single theme of content (well, responsibility).

From a design point of view, modules are the main tool for logical organization of the code. Modules can be arranged hierarchically, and there are no reasons not to take advantage of them for reducing 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 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, 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 reexport 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 `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. In functional programming we have very nice techniques for it. But we've just gotten started: in the next section we will discuss a truly brilliant aspect of functional programming.

3.1.2. Monads as subsystems

Finally, monads! A famous invention of category theorists that is 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 the 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 discuss 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.

As we are talking about functional interfaces, there are not so many of them. Functors, Applicatives, Monads, Arrows are the mechanisms we usually use to construct such a combinatorial language. It is 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 monadic eDSL has glorified the *Parsec* library of combinatorial parsers; other examples include the *Software Transactional Memory* (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 we'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. I would recommend Haskell over Scala just because Haskell is the birthplace of monads, and also it has a clean and enjoyable syntax. Sorry, Scala; nothing personal.

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 to bind impure actions together. The following code demonstrates the `do` notation that is really a syntactic sugar for monadic chains. You may read it as imperative program at first time, and after practice you'll see this understanding isn't really accurate: monads aren't imperative but 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:"`, `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 well as the whole monadic computation composed of them. The last instruction in do block defines the output of computation. 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: we 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 ()
putStrLn "You printed:" :: 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 to understand the monadic sequencing.

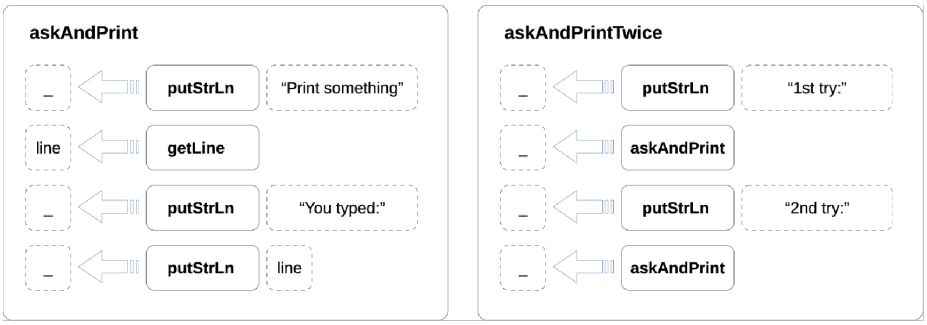


Figure 3.1 Monadic sequencing of actions in the IO monad.

We also can 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 ++ "'"     #A

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

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

#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'
```

`do` *notation* in Haskell opens a monadic chain of computations. In Scala, the analogue of `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. A relevant translation of the code in listing 3.2 is shown in listing 3.3.

Listing 3.3 Monadic code in the IO monad in Scala

```
TODO
```

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.

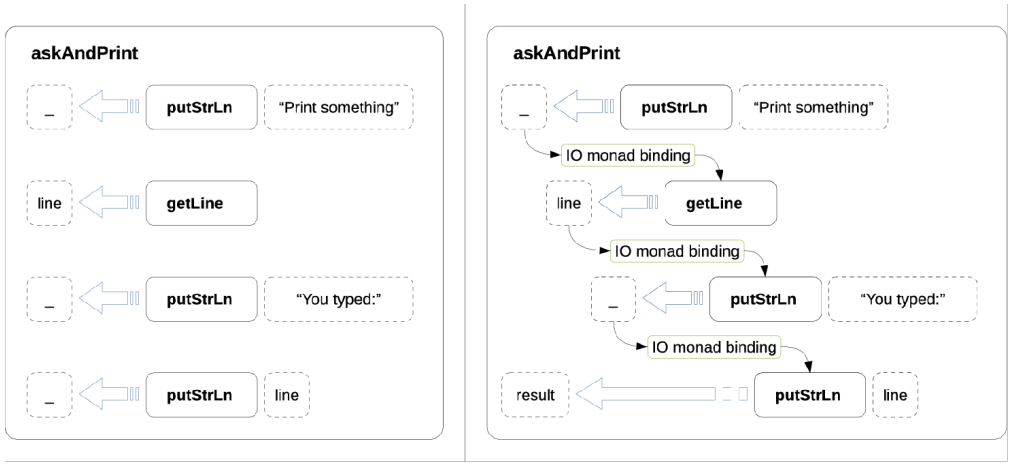


Figure 3.2 Two sides of a monad.

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. The experiments with the stolen `bind` can end sadly for you. By stealing and cutting, it's more likely you'll break the monadic laws and your "monad" will behave unpredictable. Being 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 also should learn Applicatives and Functors because every monad is Applicative Functor and every Applicative Functor is 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, and this really makes sense when you deal with very complex data structure 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 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 domain-specific languages. 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.
`Eval`	
`ST`	
`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 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 is imperative programming 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 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 scene, the `State` monad frees you from managing the state manually. It gives you two primitives to work 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. The type of this monad 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 will 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:

```
import Control.Monad.State

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", all three return either the state (`3628800`), or the value from monadic function (`()`), or both paired (`((), 3628800)`).

The last thing 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 exactly mean? In general, two monadic functions being combined together 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 context. You may 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, nesting of monadic functions you see in the examples above 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 is important to understand that effects can't be shared between independent computations. In the code below you see two different computations in the `State` monad, every 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)

-- 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 put the result back to the state. It doesn't know whether a value in the state will be a partial product of 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 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 3.4 you can see definitions of monadic functions. Note the types of the functions.

Listing 3.4 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,
    useJoins :: Bool,
    prettyPrint :: 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

------------ Monad IO -----------------
type IOMonad a = IO a

askAndPrint :: IOMonad ()
```

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 3.5 shows how we run the monadic functions defined in listing 3.4.

Listing 3.5 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        #A
    (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 of 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`, `valuesToStringCalc`), starting values for the `State` and `Reader` monads (`initialPi` and `sqlGenSettings`, respectively), and the `runX` library functions (`runState`, `runReader`, `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 all monads are pure too.

> **NOTE** You may have heard that program state is inevitably impure because of its mutability. It's not 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 is not a real imperative state everybody is aware of. The `State` monad just imitates modifying of variables (passing state as argument between monadic actions in 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 the evaluation of the monadic computation will return you a result. The monadic calculation itself can be infinitely complex and heavy; it can operate with megabytes of data, but at 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 effects should be in every part of your big system and associate these effects with a certain monad. Then you divide the subsystem into smaller ones according to the principle Single Responsibility Principle.:

- 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 do them interpretable and monadic, - this is the responsibility of the Free monad)
- Hardware description language and, probably, the two monads `Reader` and `Free` (the Reader monad may be used to hold operational information about the device we compose to not to pass it as many broad arguments).
- 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 since it is no different from the others. Moreover, the `IO` monad has the same property of being a subsystem, but in this case we are talking about a program-wide subsystem. In Haskell, the `main` function has type `IO ()`, and it is the entry point to the impure subsystem of your application. It even has some 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 having an interface to it, but you are able to compose 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 is implemented... not speaking about the problems the inheritance can cause.

So how does mixing of 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 getting the current value from state context, multiplying it to the `counter` and putting the result back. The `printFactorial` function runs this stateful computation and prints the result of factorial of 10 which is 3628800.

```
import Control.Monad.State

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
```

Now, what if we want to print all the partial products too? We may 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 may stop here if you are pleased. I believe this code isn't that 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 that because of the type `State Integer ()` which doesn't deal with the `IO` monad.

> **NOTE** We said 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 is 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*. Having a monad transformer for a monad, you can combine it with any other monad. The result will have all 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 the function to be run inside the mixed monad. Now let's rewrite our example:

```
import Control.Monad.Trans.State
import Control.Monad.Trans (lift)

type StateIO state returnval = StateT state IO returnval

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

The logical consequence we may do 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 with 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 will 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 the mathematicians were extremely smart.

At this moment you probably got a rough idea 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 go from the other side. Remember we associated monads with subsystems? We 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'll think about monads as subsystems, you may conclude it's sometimes beneficial to have a subsystem that may 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 all may be done without monads, but monads give you a better 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 an entropy of a code.

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 you may 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 are struggling with some difficult concept. By the way, this 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 character except letters, then we'll break the text into words and 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 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. Both solutions will have this set of functions while concrete types may differ:

```
 -- The entry point to the program
main :: IO a
-- Statistics calculation logic
calculateStats :: String -> statistics
-- Tokenizer, will normalize text and break it into words
tokenize :: String -> [String]
-- Takes words and counts them using dictionary as statistics data structure
collectStats :: [String] -> statistics
-- Counts single word and updates dictionary
countWord :: String -> statistics -> statistics
```

Listing 3.6 shows the first solution.

Listing 3.6: Word statistics with monads nested

```
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   #5

tokenize :: String -> Reader Config [String]
tokenize txt = do               #4
    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             #2
    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)  #1
```

#A Haskell dictionary type
#B Add 1 to word count or insert a new value
#1 Map a monadic function over a list
#2 Monadic computation in the Reader monad.
#3 Point-free style
#4 Another computation in the Reader monad.
#5 Computation in the State monad

Now, 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 the case of words should be ignored and all non-letter characters should be erased. Then we run the `Reader` monad with these parameters and put the result into the `stats` variable[4]. The `printStat` function will print pairs `(String, Int)` to the console when it's called in the last string of listing that 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 this monadic function should be applied to. We passed the `printStat` function that is 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 ()
```

So 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 output of its first argument. All results 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 #2, the `calculateStats` calls the `tokenize` function that is 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.

---

[4] This is not really true because Haskell is 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".

What these transformations will do depends from 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 feeded to the `transformation1` function. Then the result (that is still `String`) is passed into the `transform2` combinator, then the new result (`String` too) 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 following:

```
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 way different from others that's 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 type of the function (`return . words . transform2 . transform1`) yourself.

Finally, consider the computation in the `State` monad #5. 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 = do
    s <- get
    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 to each word in list are identical, they are what the function `(modify .`

`countWord)` returns being applied to 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 [()]   -- list of units
collectStats ws = mapM (modify . countWord) ws   #5     -- mapM
```

Note, how the type of the `collectStats` is changed. It now returns the list of units. Completely uninteresting result.

Now it should be clear what happens in listing 3.6. The second solution of the same problem differs not so much. Three separate monads (subsystems) are mixed together into 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's lies one level down from the `Reader` monad. Finally, the `IO` monad is on the bottom. We should call the `lift` combinator twice for all impure functions. Listing 3.7 demonstrates this solution.

Listing 3.7: Word statistics with the monad stack

```
import qualified Data.Map as Map
import Data.Char (toLower, isAlpha)
import Control.Monad.Trans.State
import Control.Monad.Trans.Reader
import Control.Monad.Trans (lift)

data Config = Config { caseIgnoring :: Bool, normalization :: Bool }
type WordStatistics = Map.Map String Int

type WordStatStack a                              #4
   = 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)  #3

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) #2
    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 config = Config True True
    let runTopReaderMonad = runReaderT (calculateStats text) config
    let runMiddleStateMonad = execStateT runTopReaderMonad Map.empty
    let printStat (w, cnt) = print (w ++ ": " ++ show cnt)
    stats <- runMiddleStateMonad            #1
    mapM_ printStat (Map.toAscList stats)
```

#1 Running monad stack
#2 Lifting impure functions
#3 Lifting state computation
#4 The monad stack type
#5 State calculation is lifted to the WordStatStack monad

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` are now belong to this custom monad. All of them share the context with configuration, all of them are able to modify the state and all of them can do impure calls. This is now a one big subsystem with three effects. We run it from bottom to top of monad stack as mark #1 shows. We start from the `StateT` monad transformer because we don't need to run the `IO` monad transformer because we are 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)  #3
```

The (`>>`) monadic operator evaluates the first monadic function (`mapM_ (modify.countWord) ws`), omits its result runs the second monadic function (`get`). It does the same thing as 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 is 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 configuration to the console #2, but first it's needed to lift the IO 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 so:

```
lift (lift $ print ("Ignoring case: " ++ show ignore))
```

If you don't like this wordy lifting, you may hide it:

```
runIO :: IO a -> WordStatStack a
runIO = lift . lift
```

And then use it:

```
runIO $ print ("Ignoring case: " ++ show ignore)
```

This is a good idea 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 he should combine. He just uses your `runIO` function for that and analogues for other stacked monads in your monad stack.

The last thing we'll discuss is the separation of subsystems in the monad stack. In listing 3.7 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 working 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 also should 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    -- it should be lifted here
```

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
```

By the technical reasons, partially applied type synonyms are not 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 is not possible as it makes type inference undecidable in some cases.

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 also every monad stack is 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 you'll get a composable and finely controllable effects.

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 of those that have already been invented. Hardware description languages 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, it should be able to describe a wide set of measuring devices and terminal units, it should aim to be simple rather than universal, and it 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 should not be delayed indefinitely.

The model-driven design of the Hardware subsystem would be the best choice to start from, but you decide to clarify the requirements first. According to the principles of Single Responsibility and Interface Segregation, this subsystem should be divided into small parts:

- *Hardware description language embedded DSL*. The base hardware description language. These data structures and helper functions are used to define the control hardware of the spaceship: engines, fuel tanks, solar panels, remote terminal units, and so on. These devices consist of components, such as sensors and terminal units to read measurements and evaluate commands. One HDL script should represent one

single device with many components inside. Many HDL scripts form a set of available devices to make control network of a spaceship. The language will be used as a tool for declaring controllable objects in the network. The definitions are not devices but stateless declarations (templates if you wish) of devices which should be instantiated and compiled into internal representation[5]. There can be many of internal representations of devices, for example, one simplified representation for tests and one for actual code. HDL should allow to extend the set of supportable devices.

- *Hardware description language external DSL*. A special format of HDL 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 the hardware runtime services.

Figure 3.3 shows the Hardware subsystem design.

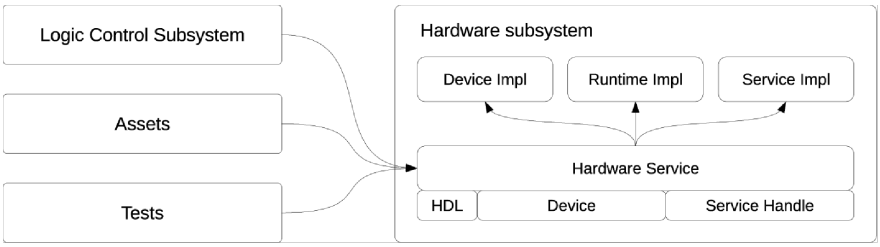


Figure 3.3 Design of the Hardware subsystem.

3.2.1. Requirements

According to the documentation, every device has a passport. It is a thick pack of papers containing 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~~	~~Description~~
Temperature sensor	AAA-T-25	None	A temperature sensor widely used in space devices. Has good measurement characteristics.
Pressure sensor	AAA-P-02	None	A pressure sensor widely used in space devices. Has good measurement characteristics.
Controller	AAA-C-86	None	A simple microcontroller.
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	The main engine of a spaceship, used for movement in space. This modification has built-in sensors of pressure and temperature in each of two nozzles.

5 This is very like as classes and objects in OOP but the definitions are still functional because they will be stateless and declarative.

Rotary engine	AAA-RE-68	Sensor AAA-P-02, “nozzle-p” Sensor AAA-T-25, “nozzle-t” Controller AAA-C-86	A small engine controlling rotations of a spaceship. A rotary engine has one nozzle and two sensors: temperature and pressure.
Fuel tank	AAA-FT-17	4 propellant-depletion sensors for fuel 4 propellant-depletion sensors for oxidizer Controller AAA-C-86	A fuel-supplying device that has one tank for fuel and one tank for oxidizer.

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

3.2.2. Algebraic data type interface to HDL

The hardware description language 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 will 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 exist outside of the evaluation flow.

Let the first design of our HDL be simple. Listing 3.8 shows the type `Hdl`, which is just a list of `Component` data type.

Listing 3.8 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    #A
    , componentGuid :: Guid               #A
    , componentManufacturer :: String     #A
    , componentName :: String }           #A

data Component
    = Sensor ComponentDef ComponentIndex Parameter
    | Controller ComponentDef ComponentIndex

type Hdl = [Component]

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

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

In listing 3.9 you can see sample definitions of hardware using lists of components.

Listing 3.9 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 algebraic data types and said we would return to advanced DSL design later. Whereas the design using primitive and algebraic data types works, it's not so enjoyable for the end user because the list syntax for composed data structures is too awkward, and 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 algebraic data type, the code based on the value constructors will become invalid, as well as 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 are the options of eDSL design we have to solve the problem of high coupling and revealing the implementation details? Great news, everyone: our simple hardware description language 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 algebraic one, due to hiding HDL details but preserving the ability to process the internal structure. But we'll delay this talk a bit more 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

The 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 may use for dealing with engines, fuel tanks, lights, and other spaceship devices. But we do not want do a decomposition of 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 to 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 having 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 the function is actually an interpreter from the `Hdl` type to the `Device` type; we will 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 is not 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 possibility to define the measurement parameter we provided for a sensor in HDL would help us to make 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 that 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...

Now 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 may name your structures differently — for example, `SensorInstance` and `ControllerInstance` — but there is 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 is nothing bad about having the same names, and it's true. But it is a bad thing 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 is 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 second define more public functions forming the interface to the Hardware subsystem. The complete code will look like listing 3.10.

Listing 3.10 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. We 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 as it depends on the value constructors. We still risk this code will be 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 algebraic data types are not 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 may adopt in practical development. The Free monad pattern helps you to separate subsystem interface from subsystem implementation and provide an embedded domain-specific language your clients may 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 and connects eDSL declarations 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.

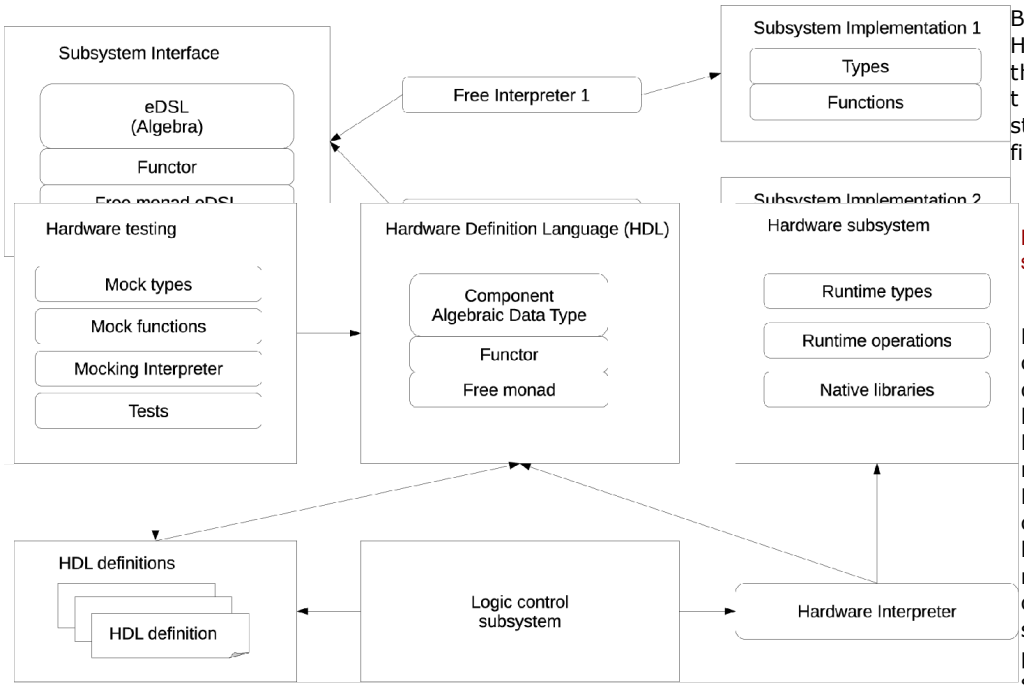


Figure 3.5: Hardware subsystem design

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

Note that Logic control subsystem doesn't need to know about Hardware subsystem runtime types. The Hardware Interpreter connects HDL language actions and runtime operations of the Hardware subsystem. It is also possible to test HDL scripts separately from native functions by mocking them in Mocking Interpreter. This design makes Hardware and Logic control subsystems low coupled and independent.Now it's time to reinforce the `Hdl` type. Listing 3.11 demonstrates the free monadic `Hdl` type and a typical script in it:

Listing 3.11: 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 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 mathematic sense. If you have such a type that is an algebra of your domain, all you should do to make it monadic is declare `Functor` for it and then declare a `Free` monad over that functor. In listing 3.11, the parametrized 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 from `Functor`.

> **TIP** This section introduces the `Free` monad, with some details of implementation described in order 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 in the Internet. Consider to start from the excellent tutorial "Why Free monads matter" by Gabriel Gonzalez. You may also find good references listed in the bibliography to this book.

Type is a `Functor` when you can map some function `f` over its internals without changing the structure of data. Plain list is a functor because you can map many functions over its internals (items), while the structure of 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 `[]` (Haskell's syntax for list of some type), `a` type variable is `Int` and `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 whole structure of the data isn't changed but internal type may be different. Both `oldList` and `newList` are lists but they carry items of different types: `Int` and `String`.

Now we are able to 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 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 may 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 one.

The `f` type variable is the type for which the `Functor` instance exist. In our case it's the `Component` type. The `a` type variable you see in the 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 `a` type 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))` 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 continuation field of `val2` and then we put `val2` into continuation field of `val3`. You may read this like "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, all of these values are monadic because they are instances of the `Free` type that is a monad. We actually don't have to nest them by hands as 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 binded into one big monadic computation. As you remember, a monadic type should be represented by some monadic type constructor `m` and a return value `a`:

```
monadicValue :: m a
```

Here, `m` equals to `Free Component` and `a` may vary depending on what we'll put into the continuation field (in the examples above `a` is `String`, `Float` and `()`). It's not visible from the code, but monadic binding uses `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'll 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 actions of your algebra, turning them into monadic functions that are composable due to their monadic nature. Actually, there is 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 the property of self-repeating. Remember when we utilized a functional list earlier to stack 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
```

But 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. Due to that, 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 don't discussed here. In particular, how to handle return values your domain actions may have. All actions of the `Component` type are declared as non-returning, 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 the learned stuff into the "free monadizing" algorithm:

1. Declare a domain-specific algebra, for example, an algebraic data type with value constructors representing domain actions.
2. Make the algebra to be a `Functor`, as the monadic nature of the `Free` monad requires. To do that, create an instance of the `Functor` type class by defining the `fmap` function. The algebra should be a type that is parametrized by a type variable. And also all the value constructors should have a field of this parametrized 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 may 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 the template we may translate to many boosters devices and operate them 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` we'll translate the script to may store state of components, uptime, error log and other important stuff. We should design a such 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, definition of actions a eDSL has, not the acting logic itself. Having a `Free` monad eDSL, we define data it transforms and declare how it should work after it is 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 either into a raw string or JSON, or XML, or even a custom format.

When the HDL language was just a list of algebraic data type, we eventually translated it to 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 to be a Free monad, so the previous `makeDevice` function can't be used anymore. We are going to translate our monadic language by a new version of the `makeDevice` function, and the `Device` type will be the target runtime data structure:

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

type Hdl a = Free Component a

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

Note, it 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 `a` 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. Now we should create an interpreter for these actions. Where should this code be placed? In other words, what module is responsible for the creation of 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 Single Responsibility Principle. 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 3.12 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 to 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`.
- As you remember, all value constructors (actions) of the type `Component` carry the continuation actions in theirs latest fields. When we pattern match over the `SensorDef` and the `ComponentDef` values, we put these continuations into the `next` value. And again, this `next` value has the `Hdl ()` type (that 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 is happening in these examples, consider reading additional sources discussing the `Free` monad, functors, algebraic data types, and domain-specific languages in functional programming.

Having the code in listing 3.12 as an example you may write your own interpreter for free language. I suggest you to write an interpreter that serializes `Hdl` definitions into JSON. In other words, the following function should be implemented:

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

This will be a good practice for you.

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 that, 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 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. What are the base bricks of this monadic language is defined by your free language. In the next chapter we'll see another good example of safe composability when we'll be talking 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 algebraic data type, you make

your language a functor and wrap it into 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 is not a big deal. You also need to keep the interpreters consistent, which may break some external code if the monadic interface is not 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 must be a functor. Although it's an easy transformation for your language, you need to understand what is it for.
- Some implementations of the Free monad can be slow as they approximate O(n^2) in the internals. But the Free monad from PureScript (`Control.Monad.Free`) has a fairly fast implementation, so this is seems not a big problem.
- The theory behind the `Free` monad is complex. For example, the `Free` type has a recursive value constructor that is kind of the fix point notion (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 is made of advanced functional concepts: GADTs, phantom types, type-level logic, and so on. This probably means you need another approach to the problem.

At this point you should be comfortable with simple eDSLs. You know how to convert a simple list-based language into a monadic language (list based language is described in listings 3.8 and 3.9, monadic language is described in listing 3.11), for which you should implement a functor 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. Following the algorithm presented in this chapter, we can build a simple eDSL with actions unable to return arbitrary values — but this is not actually a common use case. 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 are evaluated! Can the `Free` monad handle this? Completely, and you'll see it's brilliant. So, we want to revisit the Logic Control eDSL, and we will 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... 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. So, we'll adopt a simple definition of a functional service, so as not to take the bread out of the gurus' mouths.

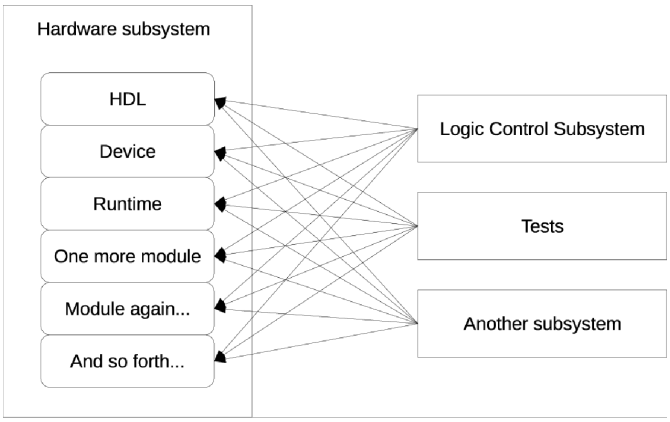
> **DEFINITION** A *functional service* is a single-task behavior hidden by a public functional interface that allows us to substitute the code at runtime transparently to the client code.

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 only requires that the functional service should have a functional interface: one of the ones we discussed previously (algebraic data type interface, interface that is made of functions, 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 will 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 supposed to be used as shown in figure 3.5.

Figure 3.5 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 3.13.

Listing 3.13 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,     #A
    getBlankDevice :: Device              #A
  }                                       #A

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 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 to initialize a new `Handle` with mock functions. This is shown in listing 3.14.

Listing 3.14 Mocking the subsystem

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

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

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 have introduced a pure service that can be considered as a boundary between subsystems. It's pure because we store pure methods in the `Handle`, and also it is stateless because we don't keep any state behind. You may ask what the benefits of the design are. The service approach reduces complexity by weakening the coupling of the application. We also can say that the service and its core type, `Handle`, represent a new kind of functional interface to the Hardware subsystem. The diagram in figure 3.6 illustrates the whole approach.

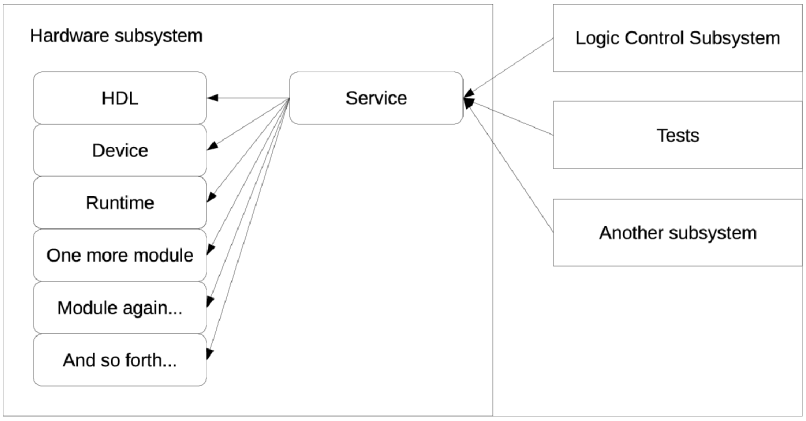


Figure 3.6 Simple relations between subsystems.

Despite the fact that we prefer pure subsystems as much as possible, reality often pushes us into situations where 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 that, 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 3.15).

Listing 3.15 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 a 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 been changed, produces the following output:

```
With real service: (getBlankDevice defaultHandle) (createDevice defaultHandle) passed
With mocked service: (getBlankDevice mockedHandle) (createDevice mockedHandle) FAILED
```

We are 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 the type `a` or hold nothing. `MVar a` is a variable of some type `a` that can be changed. The mutation of MVar is really a mutation, in the sense of imperative programming, but what is important is that it's thread-safe. Since the notion of the MVar involves mutation, it works on the `IO` layer. The interface to MVars 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. And 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 is able to 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 3.7.

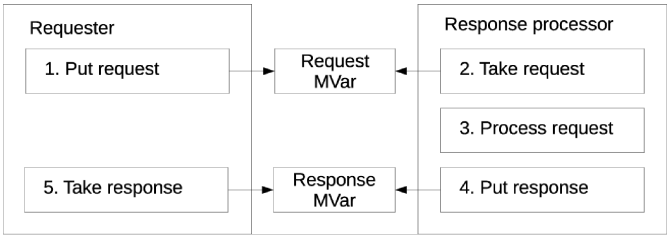


Figure 3.7 The MVar request–response pattern flow.

The requester performs two steps:

1. Put a value into the request MVar.
2. 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:

1. Take the request value from the request MVar.
2. Process the request to get the response.
3. 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 on an example. We will create a separate thread that receives a number and converts it to a FizzBuzz word. We will 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. That 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` 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, it does some impure effect in the `IO` monad and then no result is returned back, 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 be feeding 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's happened:

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

As the 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 is 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 OK 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 is better than just direct calls to subsystems — and that is the goal we are moving toward.

3.4. Summary

It's time to leave this beautiful place. We have gained a lot of knowledge about domain-specific languages and interpreters, we have learned how to design functional interfaces, and also we have investigated functional services. But the main aim of this chapter is not just introducing techniques and patterns, but 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 of requirements leads to code modifying. We prefer modular code rather than spaghetti code because it can be modified more easily. Also, introducing domain-specific languages and adopting proper abstractions makes the code self-descriptive.
- *Wise responsibility distribution*. The Single Responsibility Principle 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 is 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 will design the domain model of the Logic Control subsystem along with learning functional idioms and patterns we haven’t touched on yet.

4.

Domain model design

Links

- My email
- Book page
- Patreon campaign
- Paypal donations

This chapter covers:

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

When Sir Isaac Newton invented his theory of gravitation, he probably never imagined 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. The Newton's apple will fall down accurately following the law of universal gravitation if it has a low speed or falls to the relatively low mass object. What abstraction we should 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 are 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 if we talk about sensitive domains like SCADA (Supervisory Control and Data Acquisition) 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). So why then do we ignore the 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 where 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, where the program returned a pound-seconds measurement but it should have been newton-seconds. This bug could have been eliminated by testing or by static type-level logic. It seems, the developers had 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 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.

We call a domain the knowledge how a certain object or process works in real world. Domain model is a representation of a domain. Domain model includes more or less formal definition of data and transformations of that domain. We usually deal with domain models expressed by code, but also it can be a set of diagram or a specific domain language. In this chapter we'll study how to design a domain model, what are the tools do we have to make it correct and simple. While it's certainly fine to want correctness in software but it's not so obvious why

unnecessary complex abstractions may lead you to bugs. The main reason here is that we loseloose the focus of the domain we are implementing and start treat the abstraction as an universal hammer that we think may 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 of the requirements are handled? In this situation it's more likely you'll find many bugs you could avoid by having a fine-readable and simple but still adequate abstractions. 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 as it doesn't cover all of the corresponding domains. The scope of work in this chapter includes:

1. Define the domain of Logic Control.
2. Describe the Logic Control subsystem's functional requirements and responsibilities.
3. Implement the domain model for Logic Control in the form of embedded DSLs.
4. Elaborate the combinatorial interface to the Logic Control subsystem.
5. Create an external representation of the Logic Control DSL.
6. Test the subsystem.

The first goal of this chapter is to create the code that implements most of the functional requirements to 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 are 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 it, 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 implement a tool for solving problems in some domain. By doing this, we want to ease tasks, and maybe introduce automation into them. Our tool should simplify solving of hard tasks and make it possible to solve tasks one cannot solve without computers at all. As we plunged into SCADA field, a good example of this will be manufacture automation – a SCADA domain in which a quality and an amount of production play the leading role. But this is what our developer's activity looks like from the outside, to the customer. In contrast, when we discuss a domain, we are 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 already got familiar with domain modeling – a process we participate to design domain model. We just need more tools to do that right.

As usual, we'll start from requirements, building on what we described for Logic Control earlier. This step is very important to better understand what we should do, and how. Remember how we converted requirements into a domain-specific language 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) 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 that 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 malformed and inconsistent design the rest of the subsystems don't match with, and then you'll be forced to redesign it when the problems come out.

4.2. Simple embedded DSLs

It would be perfect to unify all this domain stuff with a limited yet powerful domain-specific language. 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 enough 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 is 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 are good to model embedded languages. The first one is rather primitive and straightforward: "a pyramid" of plain functions. It's fine, if you may fit all about domain in one page. Otherwise, you'll better try algebraic data types. In this case, your algebraic structures encode every notion of a domain, from objects to processes and users. Depending of your taste, you may design it more or less granular. 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'll better try a combinatorial approach that is described 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 the primitive types. Following is the example produces a Morse encoded FizzBuzzes – 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 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 `fizzBuzz` function that converts a number to 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 by spaces. We strongly feel composition of functions like this one as functional code.

> **DEFINITION** A *functional eDS*L is an embedded domain-specific language 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 are able to 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 to 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 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 current time, compares itto with the old time stamp, and if the difference is bigger than the delay desired, it reads sensor and decrements a counter. When the counter hits zero, all n values are read. Here is a Haskell-like pseudocode:

```
-- Function from system library:
getTime :: IO Time
-- Function from hardware-related library:
readDevice :: Controller -> Parameter -> IO Value

-- "eDSL" (not really):
readNEverySecond n controller parameter = do
    t <- getTime
    out <- reading' n ([], t) controller  parameter
    return (reverse out)

-- Auxiliary recursive function:
reading' 0 (out, _) _ _ = return out
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
```

This whole code looks ugly. It waste CPU time by looping indefinitely and asks time from the system very often. It can't be normally configured by 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 Single Responsibility Principle: there are mixed responsibilities of reading of a sensor, counting the values and of asking the time. This DSL isn't combinatorial, because it doesn't provide any functions with single behavior you might use as a constructor to solve your big task. The code above an example of spagetti-like functional code.

Let's turn the preceding code into a combinatorial language. The simplest way to get composable combinators is to convert the small actions into the higher-order functions, partially applied functions, and curried functions:

```
-- Functions from system libraries:
threadDelay :: DiffTime -> IO ()
replicate :: Int -> a -> [a]
sequence :: Monad m => [m a] -> m [a]
```

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b

-- Function from hardware-related library:
readDevice :: Controller -> Parameter -> IO Value

-- eDSL:
delay :: DiffTime -> Value → IO Value
delay dt value = do
    threadDelay dt
    return value

times :: Int -> IO a -> IO [a]
times n f = sequence (replicate n f)

-- Script:
readValues :: DiffTime -> Int -> Controller -> Parameter -> IO [Value]
readValues dt n controller param = times n (reader >>= delayer)
    where
        reader = readDevice controller param
        delayer = delay dt
```

Let's characterize this code:

- *Impure*. It makes unit testing hard or even impossible.
- *Instantly executable*. We stated earlier in the previous chapter that interpretable languages give us another level of abstraction. You write script but it really can't be executed immediately but rather it should be translated into executable form first and then executed. This means your script is declarative definition. The closest analogy here is string of code the `eval` function will evaluate in a such languages as PHP and JS. So you don't describe your domain in a specific language, you program your domain in the host language.[6]
- *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 better than the eDSL implemented as shown 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 point of view, the only measure or a functions 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 the functional programming without algebraic data types. ADTs cover all your needs when you want to design a complex data structure — tree, graph, dictionary, and so forth — but they are 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 into any language is a nice idea due to good underlying theory. Pattern matching, which we already dealt with in the previous chapter, makes the code concise and clean, and the compiler will guard you from missing variants to be matched.

Designing the domain model using algebraic data types can be done 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.

Listing 4.1 Naive eDSL using algebraic data type

```
-- 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
```

---

[6] Well, the line between programming of a domain and describing it using an eDSL is not that clear. We can always say that the I/O action isn't really a procedure, but a definition of a procedure that will be evaluated later ("later" is not in sense of program execution time but in sense of program composition. Remember, Haskell is lazy language so a code you write is not exactly what will be executed).

```
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. But they are just declarations. We actually don't know what real types will be used in runtime as a device instance, as a database handler and controller object. We abstract from the real impure actions: communicating with database, calling library function to connect to a controller, sending report message to remote log, and so on. This language is a declaration of the logic because when we create a value of the `Script` type, nothing actually happens. Something real will happen when we'll bind these actions to real functions that do the real work. Notice also the sequencing of procedures is encoded as a list, namely the `Script` type. This is how a script may 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 database. There should be an interpreting function that will translate the script into calls to real database, something like that:

```
-- 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's should be clear why three of value constructors (`Report`, `Store`, `HeatUpBoosters`) have arguments of a regular types. We pass some useful information the procedure should have to function properly. We don’t expect the evaluation of these procedures will return something useful. However other three of procedures should produce a particular values when they are evaluated. For example, the procedure of boosters initialization should initialize the device and then return a sort of handle to its controller. Or one more example: being asked for 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`, `Temperature`). This doesn't mean the particular subsystem will use exactly these types as runtime types. It's more likely the real interpreter when it meets a value of `Controller` type will transform it to own runtime type, let's say, `ControllerInstance`. This `ControllerInstance` value may have many useful fields such as time of creation, manufacturer, GUID, and we don't have to know about that stuff. The only thing we should know that successful interpretation of `InitBoosters` should return some handle to the controller. But why we use function type `(Controller -> Script)` to declare this behavior? The reason is we want to do something with the value returned. We may want to read 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. By the design of the language, all we can do now is to read the temperature or to 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 together. For simplicity we may 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 working controller as result.
2. Read current temperature using the controller. Process this value: report it and store in 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 in database.

Listing 4.2 Script for heating the boosters and reporting the temperature

```
-- Step 1 of the scenario.
initAndHeatUpScript :: Script
initAndHeatUpScript = [ InitBoosters heatingUpScript ]

-- Steps 2, 3 and 4 of the scenario.
heatingUpScript :: Controller -> Script
heatingUpScript controller =
    [ ReadTemperature controller processTemp
    , HeatUpBoosters 1.0 (seconds 10)
    , ReadTemperature controller processTemp ]

-- Scripts for steps 2 and 4.
reportAndStore :: Value -> Script
reportAndStore val = [ Report val, Store val ]

processTemp :: Temperature -> Script
processTemp t = reportAndStore (temperatureToValue t)
```

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 algebraic data types 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 of the domain notions will lead you to the notions-specific code because it's less likely you'll see abstract properties of your data by exploiting of which you could probably join several notions into one generic. For example, the procedures `Store` and `Report` may be generalized by just one `(SendTo Receiver Value)` procedure that is configured by the specific receiver: either a database or 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 exactly 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 sequence of actions, value constructors of the `Procedure` type you have to place in your code, limits of the actions you may 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` object... 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 your domain can be described by, why to 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 is not so big. It also helps to 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 will see soon how to decompose our domain into a 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 are continuing to hear questions about what functional programming is and why people should care. The common use case of lambdas 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 0the 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 just elements of functional programming but not a functional design. The essence of functional programming is composition of combinators and the functional idioms which make this composition possible. For example, the function `map :: (a -> b) -> [a] -> [b]` is a combinator that takes a function, 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 even may 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 the implementation of generic style programming that is close to functional programming, but many of functions this library has imply both mutability and uncontrolled side effects. Immutability and side effects may 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 size of the combinators. 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 give a look to a code like it's a Backus–Naur Form (BNF) of the structure you are trying to extract from text. Reading of such code becomes simple even for nonprogrammers after they have had a little introduction to BNF and monadic parsing concepts. Consider the following example of parsing constant statement. The text we want to parse looks 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 having the same `Parser a` type:

```
identifier :: Parser String
expr :: Parser Expr
```

We just put useful stuff into variables and wrapped it in the `Statement` algebraic data type. 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 these this theme in section 4.4. There we will build an external DSL with its own syntax. Parsec will help us to parse external scripts into abstract syntax tree 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 a 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. The 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 timestamp of the measurement and the device from which the value was read.

According to the Logic Control elements diagram (see figure 2.13), the instructions this scenario has can be generalized and distributed to small, separate domain-specific languages: 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 is 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.

Looking ahead...

In the Andromeda project the type `Procedure` looks different. The `Read` value constructor has one more field with type `ComponentIndex` that is just `ByteString`. It holds an index of a sensor inside a device to point the one of them are plugged inside. The types `Parameter` and `Measurement` are different too. They have one extra type variable `tag`: (`Parameter tag`) and (`Measurement tag`) that is really a *phantom type* to keep the parameter and the measurement consistent. This phantom type ensures these two fields work with the same measurement units. For example, if we read temperature than we should write (`Parameter Kelvin`) and we'll then get the value of (`Measurement Kelvin`) by no exception. This is the theme of the chapter about type logic calculations. If you are interested, the `Procedure` type is presented below. Don't be scared by `forall` keyword, you may freely ignore it now.

```
data Procedure a
    = Get Controller Property (Value -> a)
    | Set Controller Property Value a
    | forall tag. Read
        Controller ComponentIndex (Parameter tag) (Measurement tag -> a)
    | Run Controller Command (CommandResult -> a)
```

A different conclusion we may 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 is not 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 temperature lower than bound for " + @therm)
    Else If @temp > 323.15K Then
        return (Failure, "Outside temperature higher than bound for " + @therm)
    Else return Success
```

Oh, wow! This scenario is more wordy than the previous one, but you see a new object of interest her — 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 another stream is producing a value. This makes the notion of a stream different from n 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. "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 architecture of the application. In our case, it's possible to wrap value reading and transforming processes into the streams and then construct a reactive model of the domain. 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 may 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 the simple and important way of 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 is a shortcut that is called "the `Free` monad pattern." We discussed this pattern already in Chapter 3, and now we'll create another `Free` language that will be abstract, with the implementation details hidden. Let's revise the "free monadizing" algorithm. See table 4.1

Table 4.1: The algorithm of making a Free language

~~Algorithm step~~	~~Example~~

Create parametrized ADT for domain logic where type variable `a` is needed for make the type a Functor.	`data FizzBuz a`
	`= GetFizz Int Int (String -> a)`
	`| GetBuzz Int Int (String -> a)`
	`| GetNum Int Int (String -> a)`

Make it an instance of `Functor` by implementing the `fmap` function. Every Functor type should be parametrized to carry any other type inside.	`instance Functor FizzBuzz where`
	`  fmap f (GetFizz n m next)`
	`  = GetFizz n m (fmap f next)`
	`  fmap f (GetBuzz n m next)`
	`  = GetBuzz n m (fmap f next)`
	`  fmap f (GetNum n m next)`
	`  = GetNum  n m (fmap f next)`

Create Free monad type based on your Functor.	`type FizzBuzzer a = Free FizzBuzz a`

Create smart constructors that wrap (lift) your ADT into your Free monad type. You may either use the `liftF` function or define it by hand:	`-- Automatically wrapped (lifted):`
	`getFizz, getBuzz, getNum`
	`  :: Int -> Int -> FizzBuzzer String`
	`getFizz n m = liftF (GetFizz n m id)`
	`getBuzz n m = liftF (GetBuzz n m id)`
	`getNum z n  = liftF (GetNum  z n id)`
	
	`-- Manually wrapped:`
	`getFizz', getBuzz', getNum'`
	`  :: Int -> Int -> FizzBuzzer String`
	`getFizz' n m = Free (GetFizz n m Pure)`
	`getBuzz' n m = Free (GetBuzz n m Pure)`
	`getNum' z n  = Free (GetNum  z n Pure)`

Your language is ready. Create interpreters and scripts on your taste.	`getFizzBuzz :: Int -> FizzBuzzer String`
	`getFizzBuzz n = do`
	`  fizz <- getFizz n 5`
	`  buzz <- getBuzz n 3`
	`  let fb = fizz ++ buzz`
	`  s <- getNum (length fb) n`
	`  return $ s ++ fizz ++ buzz`

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 is embedded into any manageable device. So reading measurements from a sensor is equivalent to asking a controller to read measurements for from that particular sensor, because one device can have many of 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 the controller, for example, its local time, connectivity status, errors The scripting language should allow us to get and set these properties (in a limited way, possibly). Finally, the device may be intended to do some 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 that, we are able to 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)    #B
```

```
    | Read Controller SensorIndex Parameter (Measurement -> a)
#A "Non-returning" definition
#B "Returning" definitions
```

NOTE The `Measurement` type knows nothing about measurement units. This is a problem. What if you requested `Temperature` parameter but accidentally got pressure units? How would your system behave then? In 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 at once require units to be consistent, that means the values should have types with identical tags. For more information you may look into the Andromeda project or read the chapter about type-level logic.

This new language is composed of instructions to work with remote devices through a controller. This type is also parameterized by the `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 sorts of application of 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 it's contents without changing the whole structure. The `fmap` function will do the application of `g` for us, so to make a type to be 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. So 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 sense of `Functor`. This means, 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 to the `Set` procedure.

The last fields every value constructor has denote the continuation. In other words, this is the action that should be evaluated next. Also, the action encoded by the `Set` value constructor, returns nothing useful. However the actions encoded by the `Get`, `Read`, and `Run`, value constructors, do return something useful, namely the `Value`, `Measurement`, and `CommandResult` values, respectively. That is why the continuation fields differ. It's now not just of type `a` but of function type (`someReturnType -> a`):

```
data Procedure a
    = Get Controller Property (Value -> a)
    | Run Controller Command (CommandResult -> a)
    | Read Controller SensorIndex Parameter (Measurement -> a)
```

A such continuation fields hold actions that know what to do with the value returned. When the `Free` monad combines two actions it ensures 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 counts that. It receives the `g` function and applies to the mappable contents of this concrete value constructor:

```
fmap g (Get c p nextF) = Get c p (g . nextF)
```

The application of function `g` to a regular value `next` is just (`g next`) as it's shown above. The application of function `g` to a function `nextF` is composition of them: (`g . nextF`). We map function `g` over the single field and leave all other fields unchanged.

The trick of nesting of continuations is exactly the same one we used in the previous version of the `Procedure` type but now we are dealing with a better abstraction - the `Free` monad pattern. Strictly speaking, the `Free` monad pattern is able to handle of returning values by keeping a continuation in the field with a function type, and 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 composable combinator in the `ControllerScript` monad instead. This all may be looking monstrously, but it's actually not that hard, just meditate over the code and try to transform types one into other 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 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 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 may infer the type of the `Get`'s continuation field as we did it in the previous chapter. It will be (`Value -> ControllerScript a`) while `Pure ()` has type `ControllerScript a`. You just need a function instead of 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 care 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 it's composed from the `get` action and the `process` action. The `process` function works in the same monad `ControllerScript`, so it may 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 an sample interpreter. After that, we'll consider how to hide the details of the language from the client code. Whether the value constructors of the `Procedure` type should be public? It seems, this isn't a must While they are public, the user can interpret a language by pattern matching. The 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 preserving.

Listing 4.7 Possible interpreter in the IO monad

```
{- Impure interpreter in the IO monad that prints every instruction
   with parameters. It also returns some dummy values
   for Get, Read, and Run instructions. -}

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))
interpret (Free (Run c cmd nextF)) = do
    print ("Run", c, cmd)
    interpret (nextF (Right "OK."))
```

#A next keeps the action to be interpreted next.
#B Continue interpreting
#C nextF keeps the function that is awaiting a value as argument
#D Continue interpreting after the nextF action is received the value

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 theirs life here? Yes, we can. This is the theme of the next section.

4.3.3. The abstract interpreter pattern

To conceal the `Free` monad nature of our language, 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 will 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 a 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` 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 any value constructor of the `Procedure` type is present there. How it will 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 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 -}

-- The base interpret function
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 this means? Imagine you take this someone's weird library. You want to construct a script and interpret it too. The first your task is easily achievable. Let's say you have wrote the script like in listing 4.6. Now you try to write interpreter like in listing 4.7, but you can't because no value constructors are available outside the library. But you notice there is the `interpret` function that 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 be an instance of a monad, also. Suppose, you are 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 so:

```
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.[7]

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 is 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 skeleton of the application is designed well, we can return and complete the logic, providing the missing functionality. That is, such design allows us to stay on the top level, which is good in the early stages of development.

Let's now 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" acting by 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 translation of 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 to communicate with the Hardware subsystem

---

[7] 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.

- `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 also provides reactive capabilities (we'll return to this in the next chapters)

Figure 4.1 illustrates the whole design.

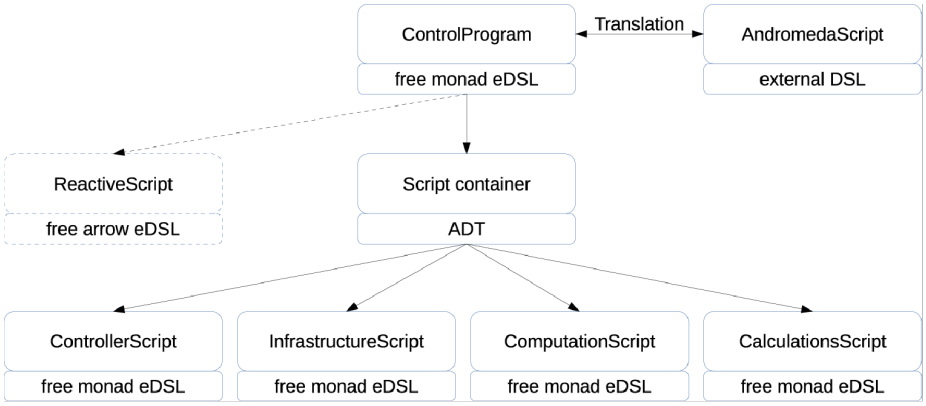


Figure 4.1 Design of domain model.

Pictures are good; code is better. Let's get familiar with the `InfrastructureScript` DSL shortly, see listing 4.10.

Listing 4.10: The InfrastructureScritp 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 we went by a short path exist in Haskell: we automatically derived a `Functor` instance for the `Action` type (#2). No more annoying `fmap` definitions! We are too lazy to do all this boilerplate by our hands. This only works with the `DeriveFunctor` extension enabled (#1).

Listing 4.11 displays the `Script` algebraic data type that ties many languages together.

Listing 4.11 The Script container

```
-- Script container, wraps all Free languages.
data Script b = ControllerScriptDef (ControllerScript b)
              | ComputationScriptDef (ComputationScript b)
```

```
                | InfrastructureScriptDef (InfrastructureScript b)
                | DataAccessScriptDef (DataAccessScript b)
```

The `b` type denotes something a script return. There is no 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 type of the two "wrapped" functions `startBoostersScript` and `getTomorrowTimeScript` don't match. `Script Time` is not equal to `Script CommandResult`. This means, we have two different containers; what we can do with them? Suppose we want to unify all these scripts in one top-level composable free eDSL as it was 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` the `fmap` function will be mapping over. Therefore we add this type parameter to the type constructor: `Control b a`. However we forgot a value of the `b` type 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)
```

And 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 what we do usually. We did it right. But the two type variables `b` the `evalScript` has the compiler can't match. It can't be sure they are equal by some weird reasons. However if you'll 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 to say it in the type definition instead of three questions marks? Definitely, the `b` type variable takes two possible types in quantum superposition. I believe you may do so in some imaginary Universe, but in here, quantum types are prohibited. The type variable `b` must take either `CommandResult` or `Time` type. But this completely ruins the idea of a `Free` language over `Free` languages. In dynamically typed languages this situation is gently 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 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 exist 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 there is 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 may use the `b` type variable inside the value constructor, but it's completely invisible from the outside. All it does inside is showing the `b` type from a script is the same type the continuation is awaiting. 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. What to do with the value the script returns, the action will decide by itself. One more note: this all is possible due to Existential Quantification extension of the Haskell language.

The complete design of the `ControlProgram` free language is shown in listing 4.12:

Listing 4.12: The ControlProgram free eDSL.

```
-- Script container, wraps all Free languages
data Script b = ControllerScriptDef (ControllerScript b)
              | ComputationScriptDef (ComputationScript b)
              | InfrastructureScriptDef (InfrastructureScript b)
              | DataAccessScriptDef (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 do our 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 so:

```
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 is a declaration to evaluate one of the scripts available, but in the future we can extend it to support, for example, declaration of a reactive model for FRP. Stay in touch, 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 have never met functional arrows before, I'll try to give you a little background on them, but this will be 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 is 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. And 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 then will evaluate `g b` resulting in value of type `c` (or `m c`. - it depends on the concrete monad you use inside the arrow). In short, if `runArrow` is application of your arrow to an argument, then `runArrow arr3 a` may be equivalent to this:

```
apply :: (a -> m b) -> (b -> c) -> m c    -- not escaped from monad
apply f g a = do
    b <- f a
    let c = g b
    return c
```

or to this:

```
apply :: (a -> m b) -> (b -> c) -> c    -- escaped from monad
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 arrow's internals so it may run monad for monadically composed arrow. This operation is associative:

```
arr1 >>> arr2 >>> arr3
```

To apply an arrow to a value you call a "run" function 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 present in the `Arrow` type class (ignore `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 is a monadic effect or not. For example: what monad is hided 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'll 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 work 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 our domain model can fit into an arrowized language, we can take advantage of all the combinators the arrow library provides. There is a wide range of arrow combinators we may use 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. Will it be a real parallelization or just logical possibility depends on the arrow's mechanism. You may 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. It is used in the arrow machinery to split a pair and feed two arrows by own parts of a pair (see the split `(***)` operator below). 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 of them, 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)
```

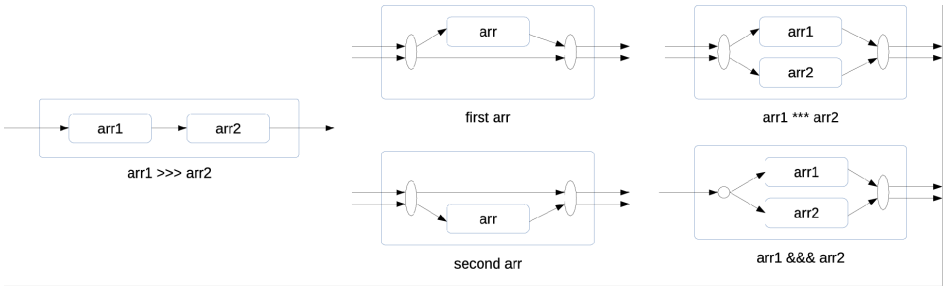


Figure 4.3 illustrates these combinators as input/output boxes.

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"[8] uses this metaphor (see visualization) and gives a broad introduction into 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 the `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).

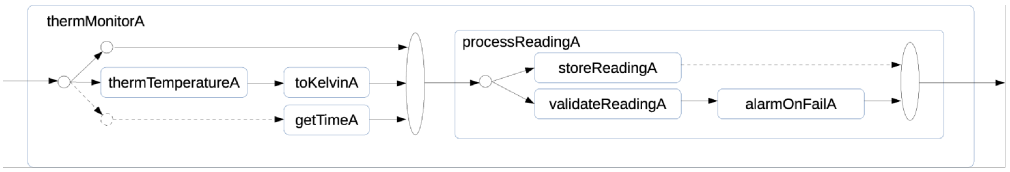


Figure 4.4 Flow diagram for thermometer monitoring arrow.

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 arrow for the diagram, and one of them - to make the calculation process completely sequential. In the following code, many arrows are combined together to transform results from each other sequenitally:

```
thermMonitorA = (arr $ \b -> (b, b))
                >>> second (thermTemperatureA >>> toKelvinA)
                >>> (arr $ \x -> ((), x))
```

---

8 https://en.wikibooks.org/wiki/Haskell/Understanding_arrows

```
>>> 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 is very close to the mnemonic notation we have introduced. Before it is introduced, let's prepare the arrow type. Suppose we have constructed the `FlowArr b c` arrow type somehow that is able to describe the diagram in figure 4.4. This arrow is specially designed to wrap our free languages and scenarios. It doesn't have any own logic, it only provides an arrowized interface to the languages. You may or you 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. If you are interested, the little intro in Appendix A is for you. All you need from that library now is the `runFreeArr`. Remember we were speaking you should interpret a Free language if you want to run it in real environment? This is the same for the arrowized interface over a Free language. To run arrow, you pass exactly the same interpreter to it:

```
sensorInst = (Controller "boosters", "00:01")
test = runFreeArr interpret 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 the 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 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.

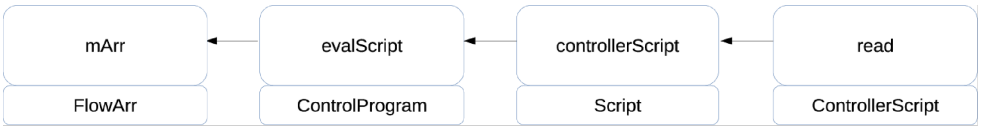


Figure 4.5: Scripts nesting.

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)
```

And also we 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 are able to 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 are not so 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 upon 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 functional reactive programming 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 fall into details and to stay on the design level.

4.4. External DSLs

Every SCADA application has to support external scripting. This is one of the main functional requirements because SCADA is an environment that should be carefully configured and programmed for the 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 of 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 programing language possibilities. The internal representation, namely, ControlProgram Free eDSL, can be simply used in unit and functional tests of subsystems, and the external representation is for real scripts 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 is not symmetric:

- Embedded Logic Control DSLs can be fully translated to AndromedaScript.
- AndromedaScript can be partially translated into the Logic Control eDSLs.

AndromedaScript should contain possibilities 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 the a separate line. All variables are immutable. Delimiters aren't provided.
Code blocks	Code blocks should be organized by syntactic indentation with 4 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 algebraic data types.
Base library	Predefined data types, algebraic data types, 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 may notice 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 algebraic data type 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 Backus–Naur Form (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 abstract syntax tree (AST). Parsing can consist of lexical and syntax analysis before the AST creation. The approaches for how to parse a certain language highly depend 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 by abstract syntax trees 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 and thus the BNF will take the form of parser combinators. There is 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 is able to translate a relevant part of the AST into the `ControlProgram` eDSL and an interpreting 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             #B
            Parser          #B
            Translator      #B
            Interpreter     #B
```

#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 abstract syntax tree 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 where 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 the BNF creation, or in our case, the parser creation, starting from small parsers and going up to the 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

---

A note about monadic parsing combinators and the Parsec library

Parsec is great. Suppose you have a log file and you want to parse it:

```
10 [err] "Syntax error: unexpected '(' (line 4, column 23)"
```

It contains an integer number, severity, and message string, each delimited by one space (exactly). The corresponding parser will be:

```
logEntry :: LogParser (Int, Severity, String)
logEntry = do
    n <- integerConstant
    space
    s <- se erityisenseverity
    space
    msg <- stringConstant
    return (n, s, msg)
```

This parser calls smaller parsers for constants and for severity. It also consumes and throws out the spaces in between. The result is a triple that describes the log entry. The `severity` parser will look like so:

```
data Severity = Err | Inf

severity' s = between (char '[') (char ']') (string s)
errSeverity = severity' "err" >> return Err
infSeverity = severity' "inf" >> return Inf

severity = errSeverity <|> infSeverity
```

Here, the (`p1 <|> p2`) expression means if `p1` fails to parse, the `p2` will try. This reads as "or". Let's run the parser against some input string:

```
str = "10 [err] \"Syntax error: unexpected '(' (line 4, column 23)\""
test = case Parsec.parse severity str of
```

```
    Left e -> print "FAILED"
    Right (i, s, msg) -> print ("Parsed", i, s, msg)
```

The functions `between`, `char`, `string`, `(<|>)`, `space`, `alphaNum`, `digit`, `many`, `many1`, `lower`, and `upper` are standard parsing combinators with the obvious meanings. A shapely set of bricks for your mason's imagination!

Having plenty of small general parsers, we build bigger ones—for example, the parser for the value constructor entry. This gives us the corresponding algebraic data type:

```
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)
```

And then we have to define the parser and data type `argDef`. A small test of this concrete parser will show if we are 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 algebraic data types 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.

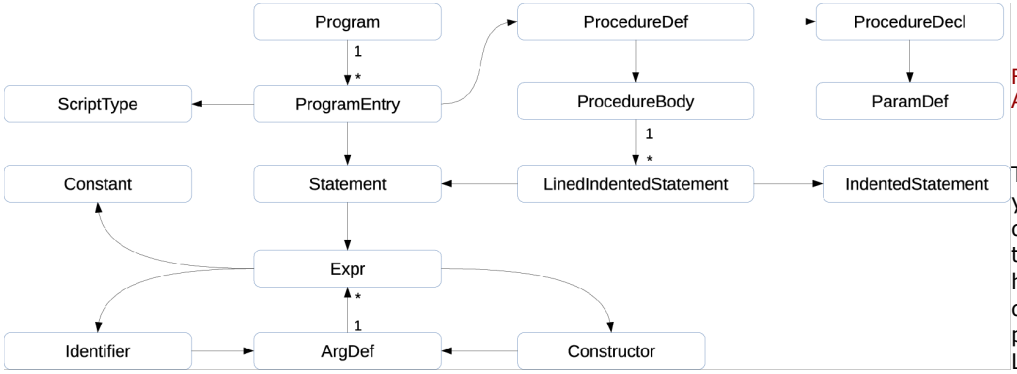


Figure 4.6 The AndromedaScript AST.

The more diverse your grammar, the deeper your syntax tree. And 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 history, the methods and tricks it suggests are still high science that can require significant adjacent skills and knowledge from the developer. 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 is responsible for interpreting the AST into evaluable code from one side and into embedded languages from the other side.

The translator is the code that pattern matches over the AST and does the necessary transformations while building a result representation of the script being processed. The translator has to work with different structures: some of them are predefined and immutable; others will be changed 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 highly depends on the tasks it's intended to solve, but it never is 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 the a good choices, unless we are 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 that is to make the translator impure. So this subsystem should have 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 algebraic data type 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 is organized as nested ADTs. You may notice that nesting of data structures unavoidably complicates making changes when they are 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 is 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 are 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 the idea.

Listing 4.14 Lenses simple example

```
{-# 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 the structure it points to. The structure may be a single value lied deeply or a range of values inside any traversable container (lists, arrays, trees are the example). The `test` function will print `True` because we have 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 algebraic data types 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 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. What else is very 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 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? I stop here 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 to develop domain-specific languages? 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 is more suitable and natural for expressing user scenarios.
- Make the code testable.
- Follow the Single Responsibility Principle, keeping accidental complexity as low as possible.

In functional programming, it's more natural to design many domain-specific languages 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 such a development approach. You can't deny the fact that when you program something, you are 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 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, the maintenance of it becomes simple and obvious, and the risk of bugs decreases.

The techniques and patterns for designing domain-specific languages we have 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 out the requirement to run 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 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 of time conditions can be really hard. Also, we can't really interpret the "runnable" code. The second option to do that 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 are trying to invent functional reactive programming. Again, we have two options: use existing FRP libraries somehow, or continue developing our own with the functionality limited. The first option is inappropriate because our scenarios should be interpretable. Consequently, it's necessary to create a custom interpretable FRP library. Fortunately, there is the concept of `Free` arrows — the analogue of the `Free` monad approach. `Free` arrows wrap an arrowized language into an interpretable form as well as a `Free` monad that wraps a monadic language. We'll return to this subject in the chapter about FRP.

5.

Application state

Links

- My email
- Book page
- Patreon campaign
- Paypal donations

This chapter covers

- What is stateful application in functional programming
- How to design operational data
- What the State monad is useful for in pure and impure environment

What puzzles me time to time is that I meet people who argue against functional programming by saying it can't work for real tasks because it lacks mutable variables, which probably means that no state could change and therefore interaction with the program is not possible. You even might 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 Haskell cheating with terms masking the imperative paradigm by its IO monad?" An impure code the Haskell's IO monad abstracts over makes someone skeptical how it can be functional while it's imperative.

Hearing that, we, functional developers, start asking ourselves whether we all wander in the unmerciful myths trying to support immutability when it's infinitely beneficial to use good old mutable variables. However it's not the case. Functional programming doesn't imply the absence of any kind of state. It's friendly to side effects but not so much to allow them vandalize our code. When you read an imperative program, you probably run it in your head and see if there is any discrepancy between two mental models: one you are building from the code (operational), and one you've got from 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 as lazy functional code can do. Next time you meet a code in State monad and you try the same technique to evaluate it. You'll be succeeded, because it can be read this way, however functions in the stateful monadic chain aren't imperative and they don't mutate any state. And that's why it easily composable and safe.

What about the IO monad, the code just feels imperative, and it's fine to reason this way on some level of abstraction, but for deep understanding of the mechanism one should know the chain of IO functions is just a declaration. By declaring an impure effect we don't make our code less functional. We separate declarative meaning of impurity from actual impure actions. The impurity will happen only when the `main` function will be run. With help of static type system, the code that works in the IO monad is nicely composable and declarative, - in sense you can pass your IO actions here and there as first-class objects. Still the running of a such code can be unsafe, because we can mistake. This is a bit philosophical question, but it really helps to not to exclude Haskell from pure functional languages.

However these two monads - State and IO - are very remarkable because the state itself you able to express with them can be safe, robust, convenient, a truly functional and even thread-safe. How so - this is the theme of this chapter.

5.1. Architecture of the stateful application

This section prepares a ground for introduction of concepts about state. You'll find here requirements to the stateful simulator of the spaceship, and also you'll develop its high-level architecture to some degree. But before that became possible, the notion of another free language is needed, namely the language for defining a ship controllable network. In this section I also give you some rationale why it's even important to build something partially implemented that already does not all but a few real things. You'll see that the design path we have chosen in previous chapters works well and helps to achieve simplicity.

5.1.1. State in functional programming

State is the abstraction about 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 a small time that is needed to form a result) returns another value. We consider function to be stateless if there is no any evidence for the client code that function can behave differently be given by the same arguments. In imperative language, it's not often clear that the function doesn't store anything after it's called. In imperative language, 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 is not deterministic. If imperative state is not 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 are allowed to write a such code. However it's a very, very bad idea, because it makes code behave unpredictably, breaks the purity of function and brings code out of functional paradigm. The opposite idea to have stateless calculations and immutable variables everywhere may firstly make someone think the state is not possible in functional language, but it's not true. State do exist in functional programming. Moreover, a several different kinds of state may be used to solve a different kinds of problems.

The first division of state kinds lies along the lifetime criteria:

- State that exist during a single calculation. This kind of state is not visible from outside. The state variable will be created in the begin and destroyed in the end of the calculation (remark: with garbage collecting, this may be true in a conceptual sense but not how it really is). The variable can be freely mutated without breaking the purity until the mutation is strictly deterministic. Let's name this kind of state auxiliary, localized.
- State with the lifetime comparable to the lifetime of the application. This kind of state is used to drive business logic of the application and to keep important user-defined data. Let's call it operational.
- State with the lifetime exceeding the lifetime of the application. This state lives in external storages (databases) which provides a long-term data processing. This state can be naturally called external.

The second division concerns a purity question. State can be:

- Pure. Pure state is not really some mutable imperative variable that is bound to a particular memory cell. Pure state is functional imitation of mutability. We also can say, pure state doesn't destruct previous value when assigning a new one. Pure state is always bounded by some pure calculation.
- Impure. Impure state is always operated by dealing with impure side effects such as writing memory, files, databases, imperative mutable variables. While impure state is much dangerous than pure one, there are techniques that help to secure impure stateful calculations. Functional code that works with impure state can be still deterministic by the behavior.

The simulation model represents a state that exist during the simulator lifetime. This model holds user-defined data about how to simulate signals from sensors, keeps current parameters of the network and other important information. Business logic of the simulator application rests on this data. Consequently, this state is operational, application-wide.

In opposite, translation process from a HNDL script to the simulation model requires updating an intermediate state specifically to each network component being translated. This auxiliary state exist only to support compilation of HNDL. After it's done, we get a full-fledged simulation model ready to be run by the simulator. In the previous chapters, we have slightly touched this kind of state here and there. The external language translator that works inside the State monad is the example, and also every interpretation of a free language can be considered stateful

in bounds of an `interpret` function. In the rest of this chapter we'll study more on this while building the simulation model and an interface to it.

5.1.2. Minimum viable product

So far we have built separate libraries and implemented distinct parts of Logic control and Hardware subsystems. According to the architecture diagram (see chapter 2, figure 2.15), Andromeda control software should contain a such functionality: database, networking, GUI, application and native API mapping. All we know about these parts is just a list of high-level requirements and a rough general plan 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 expected? What type of database is better suitable for this task? How we should organize the code? It's still unknown and should be carefully analyzed. Imagine, we did it. Imagine, we went even further and had implemented a subsystem that is responsible for dealing with database. All is fine except we created another separate component among separate components have been constructed before. While these components don't interact with each other, we can't guarantee they will match like Lego's blocks in the future.

This is the risk that is able to destroy your project. I saw this many times. Someone spends weeks developing a big god-like framework, and when deadline is happened, he realizes that the whole system can't work properly. He has to start from scratch. The end. To not to fall in this problem, we should prove that our subsystems can work together even the whole application is still not ready. Integration tests can help here a lot. They are used to verify system in 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. Besides that, there is a much better choice: a sample program, a prototype that has limited but still enough functionality to verify the idea of product. You may find many articles about this technique by keywords “minimal viable product” or MVP. The technique aims you to create something real, something you may touch and feel right now, even not all functionality is finished. This will require a pass-through integration of many application components and also the MVP is more presentable than integration tests.

This chapter is the best place to start working on a such program, namely a simulator of a spaceship. The simulator has to be stateful, no exceptions. In fact, devices in spaceship are micro-controllers with own processors, memory, network interfaces, clock and operating system. They behave independently. Events they produce occur chaotically, and also every device can be switched off while others stay in touch. All the devices are connected to the central computer that is called Logic Control. Signals between computer and devices are transmitted through the network and may be possibly retransmitted by special intermediate devices. Consequently, the environment we want to simulate is stateful, multi-thread, concurrent and impure. We'll learn many new concepts of advanced functional programming being working on the simulator in this and further chapters.

As you can see, we need a new concept of network of devices. Going ahead, this will be another free language in the Hardware subsystem in addition to the HDL language. This language will allow us to declare hardware network and construct a simulation model against it. Let's first take a quick look into it. We won't follow the complete guide how to construct it because there will be nothing new in the process, however it's a worthy idea to get familiar with main takeaways of this language.

5.1.3. Hardware network definition language

Mind maps we designed in chapter 2 may give a useful information about structure of a spaceship. What else do we know about it?

- Spaceship is network of a distributed controllable components.
- The following components are available: logic control unit (LCU), remote terminal units (RTUs, terminal units, TUs), 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.
- Controller is a device component that has network interfaces. It knows how to operate by the device.
- Controller supports a particular communication protocols.
- Logic control unit evaluates general control over the ship following the instructions from user or commands from control programs.
- The network may have reserve communications and reserve network components.
- Every device behaves independently from others.
- All the devices in the network are synchronized in time.

In chapter 3 we have already defined a DSL for device declaration, namely HDL (Hardware definition language), but we still didn't introduced any mechanisms to describe how the devices are connected together. Let's call this mechanism as Hardware network definition language (HNDL), as we 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 with small possibilities 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 definition 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 have faced the same pattern of scripts over scripts we introduced in chapter 4.

Let's assume the items in network are connected by wires. The network is usually organized in the "star" topology because it's a computer network. This means, the network has a tree-like structure, not a spiderweb-like one. We'll adopt the following simple rules for our control network topology:

- Logic control unit 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 may use to communicate with it. The uniqueness of physical addresses makes it possible to communicate with a particular device while there can be many of them identical to each other. However it's not enough because every device may have many controllers inside, so we need to point the needed controller too. As long a controller is a component, we can refer it by it's 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)
```

Now we are about to make HNDL. As usual, we map the domain model to the algebraic data type that will be our embedded DSL. As we said, there are three kinds of network elements we want to support: LCU, RTU and devices. We'll encode links between them as specific data types so we couldn't connect irrelevant elements. You may think about links as a specific network interfaces encoded in types. Listing 5.2 introduces the HNDL language. 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 the role it plays in the structure of the Free monad.

Listing 5.2: The Hardware network definition 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 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 Definition Language.
type Hndl a = Free NetworkComponent a

-- | Smart constructors.
remoteDevice :: PhysicalAddress -> Hdl () -> Hndl DeviceInterface
terminalUnit :: PhysicalAddress -> Hndl TerminalUnitInterface
logicControl :: PhysicalAddress -> Hndl LogicControlInterface
linkedDevice :: DeviceInterface -> TerminalUnitInterface -> Hndl ()
link :: LogicControlInterface -> [TerminalUnitInterface] -> Hndl ()
```

Notice how directly the domain is addressed: we just talked about physical addresses, network components and links between them, and the types reflect the requirements we have collected. Let's consider the `DeviceDef` value constructor. From the definition, we may conclude 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`) that we know represents the continuation in the 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 be looking so:

```
networkDef :: Hndl ()
networkDef = do
    iBoosters <- remoteDevice "01" boostersDef
    -- rest of the code
```

where `boostersDef` is value of the `Hdl ()` type.

What's else important, all network components return their own "network interface" type. There are three of them:

```
DeviceInterface
TerminalUnitInterface
LogicControlInterface
```

The language provides two procedures for linking of network elements:

```
linkedDevice :: DeviceInterface -> TerminalUnitInterface -> Hndl ()
link :: LogicControlInterface -> [TerminalUnitInterface] -> Hndl ()
```

The types restrict links between network components by exactly the way that follows the requirements. Any remote device can be linked to the intermediate terminal unit and many terminal units can be linked to the logic control. It seems this is enough to form a tree-like network structure that maybe doesn't reflect the complexity of real networks but is suitable for demonstration of the ideas. In the future we may decide to extend the language by new types of network components and links.

Finally, listing 5.3 shows the HNDL script for 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]
```

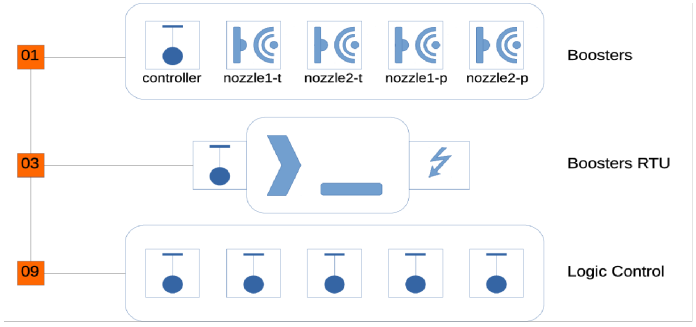


Figure 5.1: Sample network scheme
Our next station is “The simulator”. Please keep calm and fasten your seat belts.

5.1.4. Architecture of the simulator

The simulator will consist of two big parts: the simulator itself and the graphical user interface to evaluate control over the simulated environment. Let's list functional requirements to the simulation part.

- Simulation model should emulate the network and devices as close to reality as possible.
- Simulated sensors should signal the current state of measured parameter. Because none of physical parameters can be really measured, signal source will be simulated too.
- Sensors signal measurements with defined sapling rate.
- It should be possible to configure virtual sensors to produce different signal profiles. There should be such profiles: random noise generation, generation by mathematical function with time as parameter.
- Every network component should be simulated independently.
- There should be a way to run logic control scripts over the simulation as it would be a real spaceship.
- Simulation should be interactive to allow reconfiguring on the fly.

We said that the simulator is impure stateful and multi-thread 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 the separate thread as well. Other network components will be separately emulated as needed. To not to waste CPU time, threads should work with delay that in case of sensors is naturally interpreted as sample rate.
- *Stateful*. It's should be always possible to read current values from sensors, even between refreshing moments. Thus, sensors will store current values in their state. Controllers will hold current logs and options, terminal units may behave like a stateful network routers, and so on. Every simulated device will have a state. Let's call the notion of threaded state as *node*.
- *Mutable*. State should be mutable because real devices rewrite their memory every time when 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: simulator simulates an impure world; the need of threads and mutable concurrent state eventually requires impurity.

If you found these five properties in your domain, you should be knowing that there is an abstraction that covers exactly the requirement of stateful, mutable, concurrent and impure environment. It is known as *Software Transactional Memory (STM)*. Today it is the most reasonable way to combine concurrent impure stateful computations safely and program complex parallel code with much less pain and bugs. In this chapter, we will consider STM as a design decision that significantly reduces the complexity of the parallel models.

All information about the spaceship network is holding in the HNDL network definition. And now let me tell you a riddle. As soon as HNDL is a free language that we know does nothing real but declares a network, how to convert it into a simulation model? We do with this free language exactly what we did with other free language: we interpret it and create a simulation model during 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 as compilation of HNDL to 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 be probably somewhat complex because we said it should be concurrent, stateful and impure. The client code definitely wants to know as less as possible about the guts of model, so it seems a wise idea to establish some high-level interface to it. In out 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 will 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 the action is 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'll want to do so, we can even make it remote transparently to the client code.

Figure 5.2 presents the architecture of the simulator.

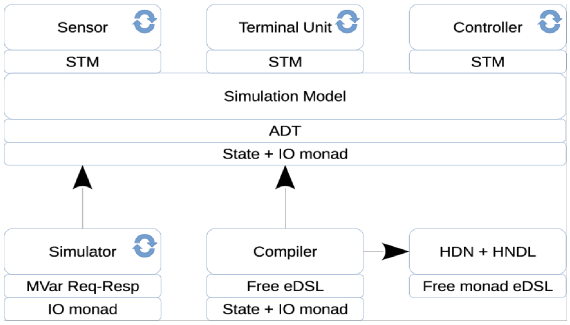


Figure 5.2: Architecture of the simulator

Now we are ready to do real things.

5.2. Pure state

By default definition, pure state is state that can be created once and every update of this value leads to copying of it. This is so called copy-on-write strategy. The previous value should not be deleted from memory, although it's allowed that no any references is longer point to it. Therefore, pure state can't be mutable in sense of destruction of an old value to place a new one instead. A pure function always works with a pure state, but what we know about pure functions? Just three things:

1. Pure function depends only on the input arguments;
2. Pure function returns the same value on the same arguments;
3. Pure function can't do any side effects.

However there is some kind of escaping from this narrow conditions. The third point includes usually 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 recursion that is not tail-optimized, so the third requirement for function to be pure is not that convincing. What if you somehow pass to the function an empty array that can be changed whatever the ways the function wants to calculate what it wants? It may freely mutate values in the array, but as longer the function does it the same way every time it's called, the regular output will be also the same. The only requirement here that no any 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 of the list above are satisfied.

Indeed, this notion of local mutable state exist and it's known as, well, local mutable state. To just name it, in Haskell this notion is represented by the ST monad. We'll probably see some applications of this in the book, but in this section we'll learn the following things:

- Argument-passing pure state.
- The State monad.

The ST monad, the RWS monad wouldn't be presented in this chapter, but you can always try them yourself. There is so much similarity between them and the concepts we have already learned, so it shouldn't be that difficult.

Also, we will do some revising in this section, but think about it as possibility to nail down the knowledge in connection to the development of the next part of the application. Also, the following text is not about state just because you are very familiar with the concepts it describes, but this section is about coherent modeling and development of functionality that haven'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 operations with it. Let's develop the algebraic data type `SimulationModel` that 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, and therefore we need some mechanism to communicate with them. First of all, there should be a way how to identify a particular object the client code wants to deal with. As soon the model is built from the HNDL description, it's very naturally to refer every object by the same identifications that 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 a controller within the whole network. Let's give this pair of types an appropriate alias:

```
type ComponentInstanceIndex = (PhysicalAddress, ComponentIndex)
```

From the requirements it's known that we want to configure our virtual sensors, in particular, we want to setup a value generation algorithm (potentially, many times). For sure, every type of simulated object will have some specific options and state, not sensors only. 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 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 that keeps things simple and understandable.

```
import qualified Data.Map as M  -- Dictionary type

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 place a new value back. The value mutation function may 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, then changes the internal `Measurement` value by calling `applyGenerator` function, then packs a new `SensorNode` value to return it as a result. Function with type (`SensorNode -> SensorNode`) has no side effects and therefore it's pure and deterministic.

A a fine line between stateful and stateless code

You may see functions with type (`a -> a`) very often in functional code because it's the most common pattern to pass state through a computation. So the `f1` function works with argument-passing state, the `f2` function does the same but takes another useful value of type `b` and the f3 function is the same as `f2` but with arguments swapped:

```
f1 :: a -> a
f2 :: b -> a -> a
f3 :: a -> b -> a
```

We can transform the `f3` function by flipping arguments:

```
f3' :: b -> a -> a
f3' b a = f3 a b
```

It can be argued that any pure unary function with type (`a -> b`) merely transforms a state of type `a` into another state of type `b`. In the other hand, every pure function with many arguments may be transformed into a function with one argument (we say it can be curried):

```
manyArgsFunction :: a -> b -> c -> d
oneArgFunction :: (a, b, c) -> d
oneArgFunction (a, b, c) = manyArgsFunction a b c
```

Consequently, any pure function that takes any number of arguments is a state-passing function.

In fact, every stateful computation can be "demoted" into stateless computation by extracting the state out and passing it as argument. Just remember C# extension methods: they could be defined in a class they work with but they separated into an external scope to not to garbage the interface of the certain class. But then these methods have to get a state (an object of that class) as a parameter.

In pure functional code, the state is propagated from the top pure functions down to the very deeps of the domain model walking through many transformations en route. The following function works one layer up over 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 updated them as necessary. The primer is shown in the listing 5.4, notice how many arguments are traveling 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 argument-passing state you see in listing 5.4 can be annoying to write and to read because it needs 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, this problem can be somewhat solved. Just use some function composition and record updating syntax in Haskell or an analogue in other 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 we are always being told that making more small, tiny functions is the key to clear and easy-maintainable code, sometimes it's better to stay sane and keep it simple.

We just discussed the argument-passing style that I'm convinced is not so exciting because it solves a small problem of pure state in functional programming. But remember, this kind of functional concept has given a birth to functional composition, to lenses, to all functional programming in the end. In chapter 3 we also noticed that the State monad is really a monadic form of argument-passing style. Let's revise it and learn something new about monads in whole.

5.2.2. The State monad

We'll compose the `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, a value of the `SimulationModel` type is stored in the context. Every function in this monad may access that value. The State monad's machinery has functions to `get` the value from the context, `put` another value instead of existing one, and do other useful things with the state. In the code above 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 updating with the result returned. It's important to note that working with the binded variables (`sensors`, `controllers`, `units`) doesn't affect the context, so the original `SimulationModel` stays the same. To actually modify the context you may `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 firstly put into the context to begin computation, but then the context was completely rewritten by an updated version of the `SimulationModel`.

> **TIP** It will not be superfluous to repeat that monadic approach is general because once you have a monad, you can apply many monadic combinators to your code irrespective what the monad is. You may find monadic combinators in the Haskell's `Control.Monad` module and in the Scala's `scalaz` library. These combinators give you a "monadic combinatorial freedom" of structuring your code. There is more than one way to solve the same problem usually.

If you decide to not to affect the context, you can just return a new value instead using the `put` function. Like this:

```
getUpdatedModel :: SimState SimulationModel
getUpdatedModel = do
    ss <- updateSensors
    cs <- updateControllers
    us <- updateUnits
    return $ SimulationModel ss cs us

updateSimulationModel :: SimulationModel -> SimulationModel
updateSimulationModel m = evalState getUpdatedModel m
```

But then you should use another function to run your state computation. If you have forgot what the functions `execState` and `evalState` do, revise 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 is no way to not to use the bind operator in the monadic code, because it's the essence of every monad. We didn't saw it before because the Haskell's do notation hides it, but it no doubt there. The equivalent do-block for the `modifyState` function will be so:

```
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 exist before the left arrow). Well, 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`) that is eta-converted form of the former.

I leave the joy of exploration of the mystical `liftM3` function to you. The "monadic combinatorial freedom" becomes even more sweet having this and other monadic combinators: `forM`, `mapM`, `foldM`, `filterM`. Being a proficient monadic juggler, you'll be able to write a compact, extremely functional and impressive code.

We'll continue 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 will be another introduction to lenses in 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 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 `Control.Lens` module. It denotes a simplified type of lenses. The type of some lens `Lens' a b` should be read as "lens to access a field of type `b` inside the `a` type". Thus, the `composingSensors` lens provides access to the field of the type `SensorsModel` inside the `CompilerState` ADT. The compiler itself is an instance of the `Interpreter` type class that exist for the HNDL free language. There is also the `interpretHndl` function. This stuff didn't presented in chapter to save the place, 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
```

Then the implementation of two interpreter type classes follows: one for the HNDL language and one for the HDL language. 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 compilation of the device. However the function too wordy. Why not use lenses here? Compare to this:

```
setupAddress addr = currentPhysicalAddress .= addr
```

The (`.=`) combinator from the `lens` library is intended for usage in the State monad. It sets a value to the field the lens points to. Here, it replaces the contents of the `_currentPhysicalAddress` field by the `addr` value. The function becomes unwanted because it's more handy to setup 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. This requires to `get` the state from the context, update the map and `put` a new state with new map back. These three operations can be easily replaced by one lens combinator (`%=`). You'll be also needing in the `use` combinator. Compare:

```
instance HdlInterpreter SimCompilerState where
    onSensorDef compDef compIdx par = do
        node <- compileSensorNode par
        addr <- use currentPhysicalAddress  -- get value from the context
        let appendToMap = Map.insert (addr, compIdx) node
        composingSensors %= appendToMap
```

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 are able to replace much boilerplate inside any kind of the State monad. Moreover, the `lens` library is so huge that you may dig it like another language. It has hundreds of combinators for all occasions.

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 faults 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 state while designing a model to simulate hardware network. A good start, isn't it? The truth is that in real life, it's more often you need a 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 a careful thinking. Sometimes you can be pleasured by persistent data structures that are efficient enough for many cases. For example, you may take a persistent vector to store values. Updates, lookups, appends to persistent vector have complexity O(1), but the locality of data seems to be terrible because persistent vector is constructed over 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, they less demanding to memory, they can be mutated in-place, they can be even marshaled to low-level code in C. In the other hand, you sacrifice purity and determinism going down to the impure layer which of course increases accidental complexity of code. In order 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 a regular variable in other languages. It can be mutated in-place leaving potential problems (non-determinism, race conditions) to the developer's responsibility. The `IORef a` type represents a reference type[9] over some type `a`.
- `MVar` is a concept of thread-safe mutable variable. 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, non-determinism.
- `TVar`, `TMVar`, `TQueue`, `TArray` and other primitives of Software Transactional Memory (STM) can be thought 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 transaction. In case of competing of two threads for the access to the variable, one of two transactions will be performed while other can safely delayed (retried) or even rollbacked. STM operations are isolated from each other which reduces the possibility of deadlocks. With advanced combinatorial implementation of STM, two separate transactional operations can be combined into a bigger transactional operation that is an STM combinator too. STM has been considered a suitable approach to maintain complex state in functional programs; with that, STM has many issues and properties one should know to use it effectively.

And now we are going to discuss how to redesign the simulation model with IORefs.

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, the model was compiled from the network we had defined earlier (`networkDef`):

```
test = do
```

---

9 Reference type in Wikipedia: https://en.wikipedia.org/wiki/Reference_type

```
    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 may do is to update it whole. We have 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 is was planned because neither sensor is seen from this test. Forking a thread for updating the whole model will be useless too. 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 be spinning constantly in the thread, but it's not accessible from the outside. How to get values from sensors while model is updating? How to setup another value generator to a specific sensor? How to query the controllers? This design of pure simulation model is wrong. We'll try another approach.

The idea is that you can observe impure mutation of an `IORef` value from different threads, as it happens in imperative world with any reference types and pointers. You firstly create a mutable variable with some value and then pass it to the threads so they can read and write it occasionally. See listing 5.6 that 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 a 5 seconds, the first worker thread increases `refVal` by 1 and the second worker thread prints what he sees currently in the same `refVal`. Both threads then sleep for a second before they continue theirs businesses with `refVal`. When the program runs, you see some numbers from 0 to 5 being printed with some of them 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 `newIORef`, `readIORef` and `writeIORef` functions is obvious. All them work in the IO monad because creating, reading and writing of 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 may 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 is no any thread synchronization in the code, the threads are reading and writing the shared state (`refVal`) at theirs own risk, because neither `readIORef` nor `writeIORef` function gives guarantees of atomic access. This is a classic example of code that one should avoid. To make it more safe, it's worth to replace the `writeIORef` function by the "atomic" version: `atomicWriteIORef`. Still, programming with bare imperative freedom may lead to subtle bugs in parallel code. What if the second thread will raise 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 the `killThread` functions, this can zombificate your threads too. With shared state and imperative threads you may find yourself hardly drawn 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 a sensors-related part of it only because 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 may 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 by another doesn't change the instances that you see in the previous listings because the definition of interpreter type classes restricts 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 that 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 is shown in listing 5.7 (see below); it's really short and understandable but it uses three new monadic combinators: the `when` combinator, a new version of the `mapM` monadic combinator the `void` combinator. You may learn more about them in the corresponding sidebar or you may try to infer theirs behavior from the usage, by the analogy as the compiler does type inference for you.

Generic mapM, void and when combinators

The `void` combinator is really simple. It drops whatever your monadic function should return, that's all:

```
void :: IO a -> IO ()
```

The `when` combinator will evaluate a monadic action when and only the condition holds:

```
when :: Monad m => Bool -> m () -> m ()
```

What can be special about `mapM` combinator that we learned already? A new version of it comes from the `Data.Traversable` module. It has a different type definition than the `mapM` combinator from the `Control.Monad` and Haskell's `Prelude` modules:

```
-- Control.Monad, Prelude:
mapM :: Monad m => (a -> m b) -> [a] -> m [b]

-- Data.Traversable:
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
```

Types are speaking for themselves. The former maps over a concrete data structure - a list of something: `[a]`, - whereas the latter maps over anything that can be traversed somehow: `t a`. The `Traversable` type class restriction ensures that the data structure you want to map over has this property - a possibility of every item to be visited. Most of data structures have this property. You can, for example, visit every item in a list starting from the head. All the trees are traversable. The `Map` data type is traversable too because it exports the corresponding type class instance. So the traversable `mapM` combinator is a more general version of the `mapM` combinator from the `Control.Monad` module.

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)  -- special mapM combinator
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 is intended for tests only, the simulation of sensors may be examined like 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 aim to is really close, but the solution has at least three significant problems:

1. It's thread-unsafe.
2. The worker thread falls into the busy loop anti-pattern when the producing variable is false.
3. 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 of the examples of wrong behavior may 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 the 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 non-determinism we mistakenly mold here may lead program 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 to be avoidable. You may, - and probably should - localize mutability as much as possible, so you can try to make all of the SensorNode's fields to be independent IORefs that will be updated when it's 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 a such sensor simulation model. It's very likely that you'll face many problems with synchronization here. This is a consequence of parallel programming in imperative paradigm. Unexpected behavior, non-determinism, race conditions, - all this is a curse of every imperative-like threaded code, and we can do better. In spite of our current inability to refuse of threads, there is hopefully a cure of imperative curse we may use to decline the problem. Welcome to the world of Software Transactional Memory.

5.3.2. Impure state with State and IO monads

So far we were communicating with the simulation model directly (see listing 5.8; for instance there are functions `startSensorsSimulation` and `readSensorNodeValue`), but now we are 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 own thread and maintain the simulation model. The State monad that is alloyed with the IO monad by means of monad transformer will provide the impure stateful context where it's very natural to place the simulation model. The simulator should receive requests what to do with the simulation model, should do that and then it should send the results back. From design point of view, 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 worth it to show the communication eDSL right now:

```
data In = StartNetwork
        | StopNetwork
        | SetGenerator ComponentInstanceIndex ValueGenerator
data Out = Ok | Fail String

type SimulatorPipe = Pipe In Out    -- Pipe from request-response pattern
```

It's really ad-hoc for now. These tree actions it contains can't cover all the needs, but we have to make something minimally viable to be sure that this design approach is good enough. Later we'll evolve this code in relation to the theme of FRP and GUI.

A typical scenario is shown in figure 5.7:

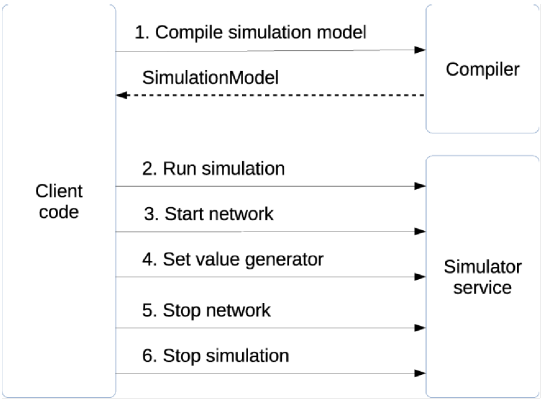


Figure 5.7: Simple interaction scenario

Let's try to write a test that shapes the minimal interface to the simulator that allows to support the scenario. It's fine that machinery isn't exist yet; following to the Test Driven Development (TDD) philosophy, we'll implement it later. Fortunately, something we already have: namely, the compilation subsystem. This piece fits to the picture well. Listing 5.12 shows the code:

Listing 5.12: 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") :: ComponentInstanceIndex
    simulationModel <- compileSimModel networkDef
    (simulatorHandle, pipe) <- startSimulator simulationModel
    sendRequest pipe (SetGenerator sensorIndex incrementGenerator)
    stopSimulator simulatorHandle
```

The workflow is very straightforward: start, do, stop using 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 are not so simple. The most interesting function here is the `startSimulator` one. From code above it's clear that the function takes the simulation model and returns 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 sensors 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 the most parts of this function are assembled from code that is already done. All we have written before is applied without any modifications. The main gap here is the the forking of a thread. Let's give a birth to the stateful impure service that is awaiting for 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 5.13: The simulator core

```
import qualified Control.Monad.Trans.State as S

type SimulatorState = S.StateT SimulationModel IO

-- Actions:
startNetwork :: SimulatorState ()         #1
startNetwork = undefined

stopNetwork :: SimulatorState ()
stopNetwork = undefined

setGenerator
    :: ComponentInstanceIndex -> ValueGenerator -> SimulatorState ()
setGenerator idx gen = undefined

-- Core:
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 "Seting 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 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. Be seeing theirs names you may guess they should switch every simulated device on or of - wherever it means for a particular node. Thus they will evaluate some STM transactions, as well as the `setGenerator` action that probably should alter a value generator of some sensor node. If you are wondering, see code samples for this book, but for now let's omit theirs implementation.

#2: The `process` function translates the ADT language to the real monadic action. It also may do something impure, for example, writing log. The `liftIO` function allows to do impure calls inside the State-IO monad.

#3: The `processor` function. It's 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
```

You may feel that all these things are similar to you; that's right, we have learned every single combinator you see here; but there is one significant idea that may be not 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 stateful factorial calculation:

```
let result = execState (factorialStateful 10) 1
```

For all these occurrences of the State monad, you run your stateful computation to get result right now. The state lives exactly the time needed to execute a computation, not less, not more. When the result is ready, the monadic state context will be destroyed. But the case with the `SimulatorState` is not so. This state continues to live even after the `s.execStateT` function is finished! Woa, magic is here!

There is no magic, actually. The `s.execStateT` function will never finish. The thread we have 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 it's 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, you don't need any explicit synchronization. Still, STM prevents the simulation model from entering into invalid states. I tell you a secret: the State-IO monad here serves one more interesting design solution that is not 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. It means, all the developments, the free eDSLs we have made in previous chapters, start working! It requires only a little effort to add some new simulator API calls. I hope it sounds intriguing to hook your motivation to go further with the book.

5.4. Summary

In this chapter, you have 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 and STM monads you see here commit to the idea that this universal concept - monads - solves many problems in handy way. This is why the book pays so much attention giving you a practical view of monads instead of explaining a theory how they really work. At this moment it should be clear that monads are much more useful thing for design of code than the community of functional programmers was thinking before.

Indeed, designing with monads requires from 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 surely was 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 the `f` function doesn't return a value that is useful for all parts of the `g` function. The `f` function is simply unaware about the internals of the `g`, and this is completely right. As a consequence, you have to follow SRP principle in FP. Again, as a consequence, you immediately gain a huge reusability and correctness of code. Even in multi-thread environment.

So what concepts you got from this chapter?

- The State monad is revisited. Although you can do stateful functional applications without any monads, the State monad is able to 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 that provides you an 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. At least you don't want to use it with threads.
- The STM monad comes to scene when you do need an impure state in multi-thread environment. STM has many transactional data structures such as `TVar`, `TMVar`, `TQueue` and others that can be adopted in modeling of concurrent data structures. You also might be interested in learning STM deeper, because this only concept is so many-sided and powerful that deserves a separate chapter or even a book. But now you should have an idea when to go this way in your code.

The practices you have learned from the first five chapters are enough for building real-world applications of good quality. However, there are more techniques and ideas to study. Keep going!

6

Multithreading and Concurrency

Links

- My email
- Book page
- Patreon campaign
- Paypal donations

This chapter covers

- Multithreading and complexity
- Abstraction for threads and concurrent state
- Useful patterns for multithreaded applications

Note 1

I'm continuing to write the book after 2 years gap. I decided to change the project: supporting the Andromeda project became somewhat meaningless because it requires refactoring first and I don't want to spend much time reworking it. So starting from this chapter we'll be developing another application that has a bit different architecture and design. But yet, it's based on the same ideas of hierarchical Free monad languages I've introduced in the book earlier.

Note 2

The chapter cannot cover all the questions about concurrency. Consider to get familiar with the excellent book by Simon Marlow, "Parallel and Concurrent Programming in Haskell" - it's all dedicated to explain many different aspects of this topic. This chapter aims to define a place for the concurrency and threading in the whole application with relation to other architecture decisions.

Application

It will be a kind of tool for astronomers. We'll create a centralized catalogue of meteorites fell to Earth. It will be a server accepting reports about sky events detected by astronomers. With the help of client application, a scientist can report the region, mass and time of a meteor. The client will be interacting with server over some channel: TCP, UDP, HTTP or WebSockets (this is a future decision to make). Server will be a multithreaded concurrent application with database. Our task is to show how the subsystems can interact with concurrent application state, how to manage resources and deal with DB. In this chapter, we'll see the basics of the languages and business logic design.

The Meteor data type will hold the info about meteors:

```
-- The description of meteors.
data Meteor = Meteor
  { size :: Int
  , mass :: Int
  }
  deriving (Show, Read, Eq)
```

The getRandomMeteor function creates a random meteor. This function may look differently depending on the environment: IO monad, custom eDSL, or mtl-style monad etc.

```
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 Section 6.1 we'll briefly talk about bare threads and multithreading code: why it's hard, what problems we can meet, what problems we will meet, ten to one. In section 6.1.3, we'll start solving some of the problems, and continue searching for a better approaches till the end of the chapter.

1. Multithreaded Applications

Multithreading and concurrency still remains one the most difficult theme in programming, yet over the decades of research in this field. Well yes, we use bare threads in our programs for a long time, we know how to organize our systems as separate processes, and we even collected a broad experience in a wider theme of distributed calculations. But this was never felt an easy field of practice. We won't go too deep in the multithreading in this chapter, and we can't essentially. Whole books are dedicated to explain the huge number of pitfalls and approaches, and still it's not enough to cover them all. We have another goal which is to see how multithreading fits into the architecture of the application. We'll try the most viable techniques and will try to keep our multithreaded code readable and maintainable. Hopefully, functional programming world gave us such thing as composable STM, and this technology deserves much more attention because it's able to reduce the complexity of multithreaded applications drastically. Composable STM provides a (relatively) new reasoning that results into several useful patterns of application structuring. STM is so finely beats the main problems of multithreading that there is no real reason to avoid this technology except maybe the cases when you need a maximum of 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.

1.1. Why is multithreading hard?

There are several reasons why creating multithreaded applications is hard. Most of them grow from the nature of interaction between threads in a concurrent, mutable environment. The more operations a thread performs over a shared mutable state, the more cases we have when other thread can change this shared state unexpectedly. Our intuition and reasoning about the multithreaded code is usually very poor, and we can't predict all the problems easily. 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 an often problem. It occurs when a shared resource can be turned to an invalid state by two or more threads accessing it in the way we didn't expect. When this situation happens, all the following computations become meaningless and dangerous. Program can still run, but the data is already corrupted continuing to corrupt other data. It's better to stop the program than to let it run further, but detecting that the bug happened is a mean of luck.
- **Deadlock**. A thread may stuck in attempt to read some lock-protected shared data when it's not ready. And it may become so that the data will never be ready, so the thread will be blocked forever. If the program is configured to wait this thread, the whole computation will be blocked. We'll get a running program that just hangs in forever waiting for nothing.
- **Livelock**. We can construct two or more threads in a way that they will be playing ping-pong with data at some point of calculations but will never go further. The threads will be very busy of doing nothing useful. This is a kind of deadlock, but it makes the program simulating it does the work when it doesn't.
- **Thread starvation**. This situation occurs when a thread works less than it could to because it can't access the resources fairly. Other threads take the ownership on the resources either because they are luckier, or because it's easier for them, or by another reason. The result is the same: statistically, the starving thread works with less throughput than we expected. In general, solving of this problem can be tricky, and even STM does not provide guarantees of fairness (at least, in Haskell).
- **Unclear resource management**. This is not a particular problem of multithreaded applications but it's rather a general problem. It just amplifies in the multithreaded environment. It's quite easy to complicate the code when we didn't decide on how we want to handle resources. We can step on bad race conditions accessing a killed resource (reference, pointer, file, handle etc) and our program will crash.

- **Resources leakage**. Leakage of some resource (system handles, memory, space etc) is another result of poor resource management.
- **Resources overflow**. It's relatively easy to run a thread that will be producing more resources than we're able to 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 a little surface of it in the chapter, and if you need more information consider reading books about distributed systems and actor models.
- **Incorrect exceptions handling**. Exceptions are hard. There is no true way to organize exception safety in the application. Exceptions are another dimension that is orthogonal to other code so we get a multiplied number of cases to think about when we write a program. Correct exceptions management in multithreaded applications is near to impossible, but we probably can mitigate the problem a little. We'll see.

Concurrent bugs are subtle, hard to search, test and fix, and their presence can be spotted a time after the code is deployed to production. This is why constructing concurrent multithreaded applications should be done with a fair carefulness. Hopefully we have immutability, MVars and STM. These three concepts eliminate different problems. Not only bugs but also a ridiculous complexity of multithreaded code writing.

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 moment of 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 the automatic or semiautomatic parallelization, but this works fine when you can define a separate task able to run independently. Once you introduced 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 a system:

- An astronomer who is watching for meteorites and reporting about events to the remote tracking center;
- The remote tracking center able to accept reports from astronomers.

Simulation here means the threads will be representing these two actors within a single application, so we'll do it without interprocess communication. Just bare threads and shared mutable objects.

Listing 6.1: Application with bare threads

```
-- 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 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
```

What could you say about this program? While it's deadly simple it's also deadly wrong. It won't crash but it will be producing a lot of invalid data. It can lose meteorites, it can track meteorites many times, astronomer can report more meteorites than trackingCenter is able to process. In other words this program has destructive race conditions, it doesn't consider to balance producing and consuming, and it may occasionally overuse the memory.

Finding a such concurrency problems can be a kind of masochistic art because it's never obvious where the bug in a more or less non trivial program. Here, however, the problem is easy to find but not easy to fix. We have the two processes happening at the same time, see the next figure:

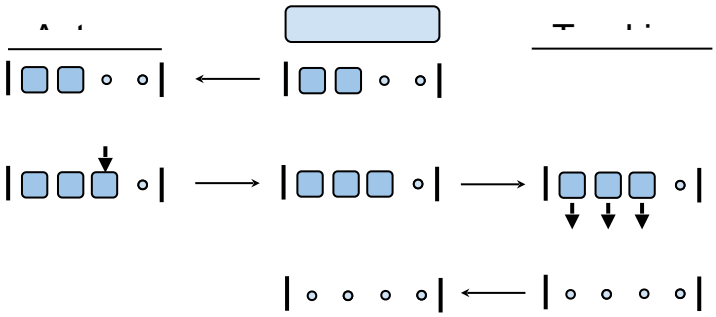


Figure 6.1: Accessing a shared state.

Astronomer is firstly reads the channel, adds the third meteorite to the list and writes the list back. Tracking center than reads the updated channel, uses what he is found there and clears the channel. But this is the ideal situation. Astronomer and Center don't wait for each other and this is why the operations with channel can sequence in the unexpected order causing data loss or double usage. Consider the next Figure that shows the situation with double usage of meteorites 1 and 2:

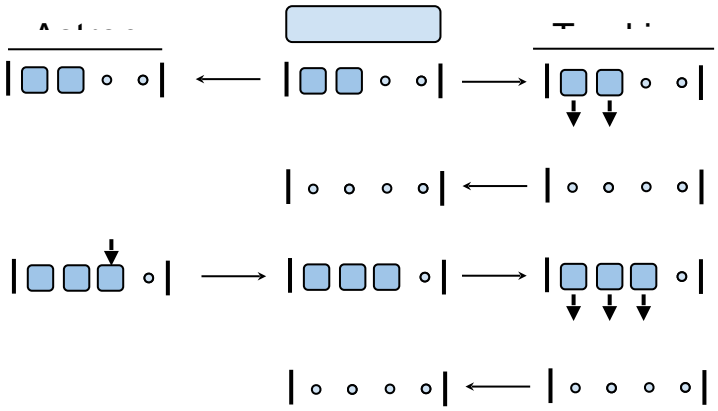


Figure 6.2: Race condition in accessing a shared resource and double usage error.

This is clearly the race condition. We should organize the access to the channel in a thread safe manner somehow. It can be mutex preventing a thread to operate on the channel if other thread still does his 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 the most languages accessing a shared mutable data structure in the middle of the write procedure will bring another problem: a partial change of new data might be observed among with a piece of old data. The

data we read from an unprotected shared resource can occasionally vanish leaving an undefined cell of memory. And the program will crash.

So we're 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 is triggering when the channel got a meteorite, than the design has flaws.

- Firstly, we don't control side effects. Both Astronomer and Tracking Center are operating inside IO.
- Secondly, we don't control resources. What if someone adds a resource initialization to the `astronomer` function? This functions runs forever and many times. Hardly one 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, however it happens from time to time and we get a cycle with uncontrolled resources initialization. We probably can't do much with a bad fortune but we can prohibit forking threads mindlessly. 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 don't take into account that these `forkIO` threads are green - that's irrelevant for now).

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 possibility 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 other useful actions in business logic (logging, DB access etc). 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 also will structure the business logic in a layered manner. Earlier we 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 it using the same approach with Free monads. See the following diagram describing this separation clearly:

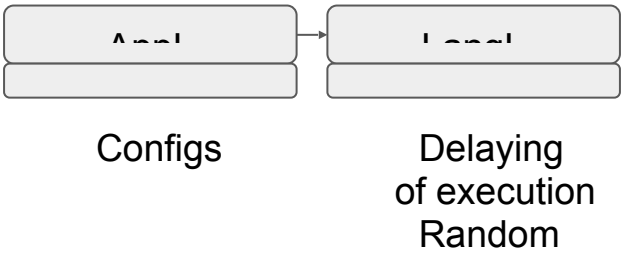


Figure 6.3: Separation of responsibilities.

These languages are not raw IO anymore as in the previous section, we just abstracted our intents and described the only things that are possible on these two layers: AppL and LangL.

So the AppL scripts will be responsible for the initialization of the app state, for the declarative description of processes and relations between them,
for configs management and so on. Let's agree that we don't want to have some specific domain logic here but rather we prepare the 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 still can call any LangL script and methods from the AppL directly. In fact, all the logic can be written in the AppL scripts, unless we consider multithreading. Once it's introduced as the Figure 6.3 says, the scripts evaluating in additional processes can be only of LangL.

Going further, we can define a more granular languages hierarchy to achieve less coupling. See Figure 6.4:

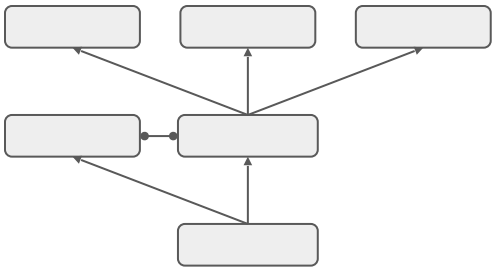


Figure 6.4: Language hierarchy.

NOTE. This is how the "real" code should be organized in this approach. There is no real point to smash all the things together, like other effect systems oblige to do. It's better to not to call these languages "effects", but rather say "subsystems". Mainstream developers may find many similarities to how they are 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 certain design, it's very simple and straightforward. The corresponding eDSLs will look like this (see Listing 6.2):

Listing 6.2: Two languages for 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
```

Having 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 6.3 is very similar to the code in Listing 6.1 except some problems are now fixed. We achieved our goal to prohibit unwanted behavior, we created an abstraction over bare threads.

Listing 6.3: 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 about it as a declaration, this will give more freedom in the design. In here, we do not fork process, we define a process. We do not call scenario, we define a logic to be evaluated in the main thread. So let's just define two aliases for 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 functional declarative design, we should seek the possibilities for declarative description 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
```

How do you think is it better now?

Okay. The picture still lacks the big part, namely the application runtime and interpreters. Will this abstraction even work? What if our design cannot be properly implemented? This is usually not a problem for Free monads, when objectification of the design ideas does not require additional concepts like Haskell's advanced type level tricks, multiparametric type classes, type families and so on. Even more, 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 that there is a clear sequence for designing eDSLs:

- Design eDSL from the usage point of view;
- Crate some logic over it and ensure it looks good;
- Make it compile;
- Make it work.

Our next goal to make it work. One question: what if someone wrote the following code?

```
app :: AppL ()
app = do
  ch <- evalLang $ newVar []
  forever $ process $ forever $ astronomer ch   -- forever process forking
```

Yes, again this problem. One can still fork threads infinitely within an `AppL` script. Let's think how we can enhance our abstraction here and introduce a thread pool.

1.4. Threads bookkeeping

Different methods can be invented to limit the count of threads acting in the system. Depending on the needs, checking for limits can be either 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 at all and made it implicit. Decision 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 such language design:

```
-- Token for accessing the process forked.
data ProcessLimits = ProcessLimits
  { currentCount :: Int
  , maximumCount :: Int
  }

data ProcessToken a = ProcessToken Int

data ProcessF next where
  -- Getting the limits
  GetProcessLimits :: (ProcessLimits -> next) -> ProcessF next
  -- Trying to fork a process. If success, a token will be returned.
  TryForkProcess :: LangL a -> (Maybe (ProcessToken a) -> next) -> ProcessF next
  -- Blocking until some results from the specified process is observed.
  AwaitResult :: ProcessToken a -> (a -> next) -> ProcessF next

type ProcessL = Free ProcessF

-- 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 also will be blocked awaiting a result from the process he 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   -- Will be blocked on every process
```

It might seem like the 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 is not 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

-- Blocking until some results from the specified process is observed.
-- On case the process is absent, failed or exceptioned the error is returned.
awaitResult :: ProcessToken a -> ProcessL (Either Error a)
```

The language also does not allow to setup 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 a 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 full thread pool used to this moment? Nothing extremely bad can probably happen except there will be more processes evaluating than the limits are allowing. More other race conditions are possible with this mutability involved, so we have to be careful. Ideally our languages should not introduce problems. This is even more important in the multithreaded situation.

Just to be clear: we didn't plan to run either ProcessL or AppL methods from different threads. In fact, we're trying to avoid this. According to the 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 a real multithreading into the whole problem. From the AppL point of view these processes represent some asynchronous actions. With no mutability involved this approach is thread safe. But the all problems of multithreading environment 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 after adding processes. How to handle this concurrency right? Let's talk about this situation in the next paragraph, and now we're about to finish our current talk.

Let's see how to implicitly control the limits. In here, we need to reveal the underground side of the Free monad languages, namely a runtime facilities (implementation layer). All the important activity will be there: checking limits, awaiting resources to be released, obtaining a resource. For the client code, the interface will be the same as in listing 6.2:

```
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 exhausted, the ForkProcess method should block the execution and wait for it. On the implementation layer, we'll be simply 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 popups 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 finished his activity. Polling the thread and waiting when it's done would work but will require a supervisor thread, and more complex logic to be written. Instead this let's 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 is the code:

```
-- Interpreting the AppF language.
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
  go 1
  pure $ next ()
  where
    go factor = do
      let psVar = _curThreadsCount rt
      let maxPs = _maxThreadsCount rt

      -- Blocking on the synchronized process counter until it's available.
      ps <- takeMVar psVar

      -- Checking the current limit.
      when (ps == maxPs) $ do
        -- Limit is reached. Releasing the MVar.
        -- Restarting the try after a delay.
        putMVar psVar ps
        threadDelay $ 10 ^ factor
        go $ factor + 1

      when (ps /= maxPs) $ do
        -- Limit is fine.
        -- Releasing the MVar and forking a thread with a cleanup action.
        putMVar psVar $ ps + 1
        void $ forkFinally
          (runLangL rt act)
          (const $ decreaseProcessCount psVar)

-- Decreasing the counter thread safely
decreaseProcessCount :: MVar Int -> IO ()
decreaseProcessCount psVar = do
  ps <- takeMVar psVar
  putMVar psVar $ ps - 1
```

NOTE. In here, we use exponential backoff for the sleeping time after each failure. It might be not the best solution, and we might want another strategies: constant delay time, arithmetic or geometric progression, Fibonacci numbers and so on. But this question of delaying effectiveness steps out from the book theme.

Synchronizing on the psVar MVar is necessary. A forked thread will finish at random time and will interfere with one of these situations: the main thread can perform another forking operation with changing the counter; or possibly another forked thread is about to finish and therefore wants to change the counter. Without the synchronization, exactly the same problem will happen as we have seen on the Figure 6.2. See the next figure showing a race condition when MVar is not used. Figure demonstrates a data loss and corruption resulting in invalid counter state:

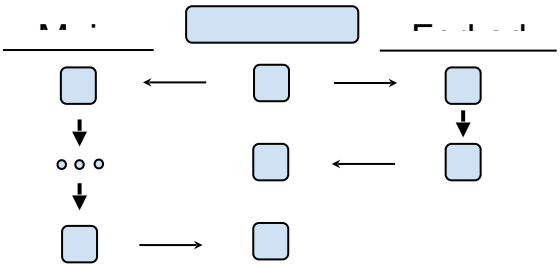


Figure 6.5: Concurrency bug caused by a race condition.

The next figure explains how MVar can save from this situation:

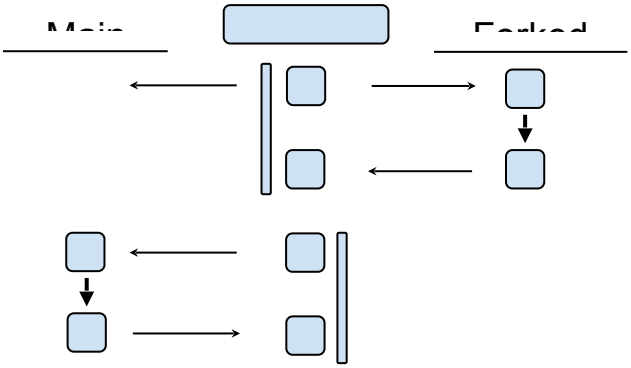


Figure 6.6: Safe concurrent interaction over the synchronized MVar.

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:

```
-- Application scenario
app :: AppL ()
app = …

-- Interpreter run function
runAppL :: Runtime -> AppL a -> IO a
runAppL rt = foldFree (interpretAppF rt)

-- Starting app
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 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.

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 only mutated within a single isolated transaction. STM has similarities with transactional databases: while a value is handled by one thread, another concurrent thread will wait when it's free. In contrast to databases, STM is not 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 in there is no collision, but if there is, STM will decide how to solve the collision in a predictable way. The predictability - is what we were missing in more low-level approaches involving a customly made mechanisms of

concurrency in our apps. Of course you can take ConcurrentQueue or ConcurrentDictionary in C#, and while you use these structures in a simple way, you're fine. But as long as you'll need a code that will be interacting *both* with ConcurrentQueue and ConcurrentDictionary, you'll immediately get a concurrency problem of a higher level: how to avoid race conditions, unpredictability while keeping the complexity low? Monadic STM (like Haskell has) solves all these problems. it gives not only a predictable concurrency, but also a composable concurrency, when you can operate with an arbitrary data structure in a safe manner no matter is it a simple concurrent variable, composition of them or even a bunch of complex concurrent data structures. We can say monadic STM is a conductor of an orchestra. It has the whole picture of the symphony, and it looks for the correct, completely definite evaluation of the music.

2.1. Why STM is important

STM works over data model locally and makes separate parts of the model 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 is composed from STM primitives so the two threads may access theirs parts without blocking whereas the third thread will wait its turn:

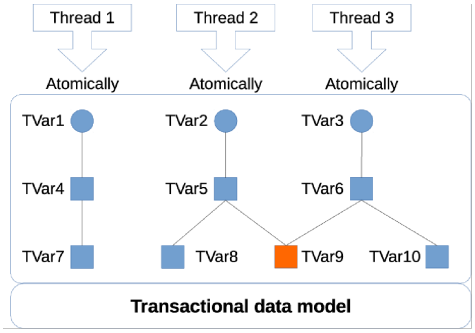


Figure 6.7: STM model

As many functional concepts, STM follows the "divide and conquer" principle, and as many functional concepts, it separates data model from modification (behavior) model. Using STM primitives (like queues, mutable variables or arrays) you firstly construct a data structure for your domain - this will be your concurrent domain model. Secondly, using special functions from STM library, you write transactions to access your data structure or its parts; doing so, you'll get some behavior model that is guaranteed to be thread-safe. Finally, you run your transactions in threaded environment passing them an instance of domain model. Figure 6.8 gives you some basic intuition of the STM and the division of data model and behavior model.

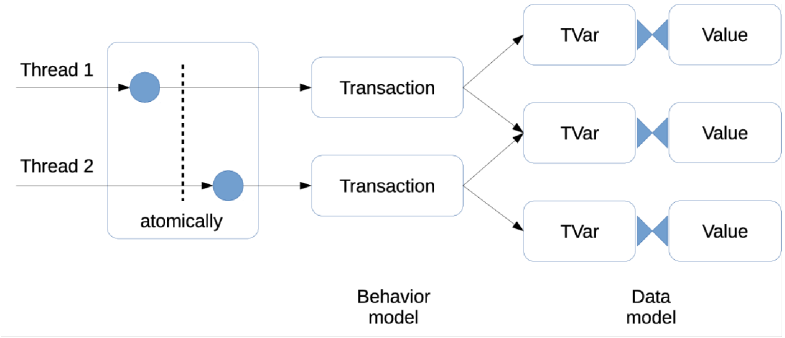


Figure 6.8: Data model and behavior model in STM

A typical STM library has several primitives that are used to construct a concurrent data structures. For example, the STM primitive `TQueue` from Haskell's `stm` library is the analogue of `ConcurrentQueue` from .NET framework. Both are 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 guaranties of atomicity. Strictly speaking, synchronization of access to STM primitives is taken out of primitives themselves, in opposite to concurrent collections where synchronization is burned inside. This becomes more important with the fact that having several distinct operations over some STM data structure, it's possible to combine these operations into one big operation that will be evaluated atomically in the IO monad. Here the combinator:

```
atomically :: STM a -> IO a
```

where `STM` is the transactional monad type. In other words, *STM concurrency is composable* in 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 another transaction to obtain a bigger one. Monadic STM operations may be thought as combinators of transactions.

Another important idea of STM is the *retrying* operation. Any transaction can be rollbacked and retried again after a while. For example, some thread A has commited his own transaction in which a transactional queue (`TQueue`) variable been modified. At the same time, thread B is happened to be in its own transaction that tries to take an item from the same queue. However the second thread should rollback his transaction because it sees the queue has changed. Figure 6.9 shows this situation:

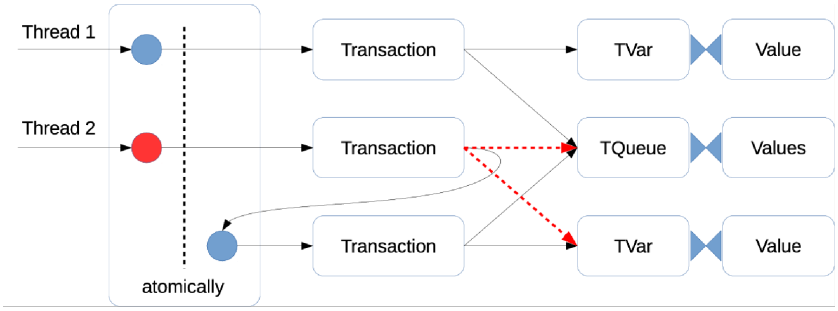


Figure 6.9: Retrying transaction when concurrent changes occur.

Retrying operation is automatic. However it's often needed to wait for a particular state of transactional variables to continue a transaction. The `retry` combinator may be used to block a thread until some transactional variable is changed, and 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 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 who may 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 be thrown.

The `retry` combinator is very powerful tool of STM although it's very simple:

```
retry :: STM a
```

With this combinator, it becomes possible to do some "wise tricks" in concurrent code. The `retry` combinator and other STM operations may look like some 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 researched STM note that this approach of retrying has some performance impact but this impact is acceptable comparing to easiness you get in constructing correct concurrent models. You may find a much deeper look into STM in such books as "Parallel and Concurrent Programming in Haskell" and "Real World Haskell". There are also good materials about STM in Clojure.

2.2. Reducing complexity with STM

Recently, our program was using MVar for safe synchronisation of the threads counter on the implementation layer. MVar is fine, it behaves similar to mutexes and gives reliable concurrency guarantees like:

* All threads will be woken up when a current owner thread releases the MVar;
* After taking the MVar by a thread, there is 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 a careful code writing because it's so easy to make a bug when any of the two operations - taking and releasing of MVar - left unpaired and blocked the program. More complex concurrent data structures based on many MVars will amplify this problem exponentially.

So let's see how STM changes the game rules here. Firstly, we replace MVar by 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 of the counter will be a 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 is not 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 got 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 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 documentation of the method. 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 does not 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 feel deja vu? Good thing here is that green threads in Haskell do not cost that much, and while they are waiting on the STM locks, they do not consume CPU. So the leakage problem is mitigated a little. Still, it's better not to call the forkProcessAsync function in an infinite cycle.

2.3. Abstracting over STM

So far we have 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 does not expose any impure semantics into the client code. It's still impossible to run IO actions because we have 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 could be worse if we took a slightly different decision on this. Consider the following code with the LangL having 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 of 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 menacing 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 he is fine to have. It might be not that bad to have black holes in your project in case of a good discipline.

Nevertheless, it's impossible to calculate the number variables have been created, to see who owns them, and what will happen with these variables in the multithreaded environment. A quick answer: it will be very dangerous to access a raw mutable IORef from different threads. This is immediately a race condition originated from the language design. We should not be so naive to think no one will step on this. Murphy's Law says it's inevitable. So what we can 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.

Can we just use STM directly? Well, yes. There is an obvious way to run an STM transaction from the LangL script: either with evalIO or with 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

-- Smart constructors
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 a 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.
- To limit the number of variables produced.
- To be able to persist and restore your application state.
- To have a consistent set of languages.

So let's do the full abstraction over STM and see what benefits and flaws it will give to us.

For the sake of modularity we'll create the StateL language and extract it from LangL as Figure 6.4 says. Single language - single responsibility. This language will operate with a custom state var which will represent a TVar in the business logic.

```
type VarId = Int

-- | Concurrent variable (STM TVar).
newtype StateVar a = StateVar
  { _varId :: VarId
  }

-- | State language. It reflects STM and its behavior.
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 own atomically function:

data LangF next where
  EvalStateAtomically :: StateL a -> (a -> next) -> LangF next

type LangL = Free LangF

-- Smart constructor
atomically :: StateL a -> LangL a
atomically act = liftF $ EvalStateAtomically act id
```

There is nothing new here, we just created another Free language wrapping a subsystem. The interface should be concise, easily understandable, simple, full and complete. Not a coincidence that we just repeated the interface STM has for TVars, - it's fine! Unfortunately, there is no other structure than StateVar (TVar) in the language. All of them can be expressed with a StateVar, it just won't be super performant. The documentation about STM's TArray says the same: it's possible to create a better structure if needed. The same works here. Consider to extend the language or design it somehow differently if your task requires so.

You might have guessed already what the business logic will look like now:

```
type ReportingChannel = StateVar [Meteor]

reportMeteor' :: ReportingChannel -> Meteor -> StateL ()
reportMeteor' ch meteor = do
  reported <- readVar ch
  writeVar ch $ meteor : reported
```

```
-- Evaluates StateL script with own `atomically` function.
reportMeteor :: ReportingChannel -> Meteor -> LangL ()
reportMeteor ch meteor = atomically $ reportMeteor' ch meteor
```

More important things are happening in the implementation layer. The interpreter for StateL should translate the actions to the STM environment rather than to IO. This is because all the monadic chain of the StateL monad should be a single transaction, and we should not evaluate each method separately. Also, another runtime structure is needed to store StateVars and the corresponding TVars.

```
newtype VarHandle = VarHandle (TVar Any)  -- Untyped data stored in here

data StateRuntime = StateRuntime
    { _varId  :: TVar VarId                    -- ^ Var id counter
    , _state  :: TMVar (Map VarId VarHandle)   -- ^ Node state
    }
```

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 a moment, another interesting idea: a typed eDSL and business logic that runs over untyped runtime without loosing in type safety. See Figure 6.10:

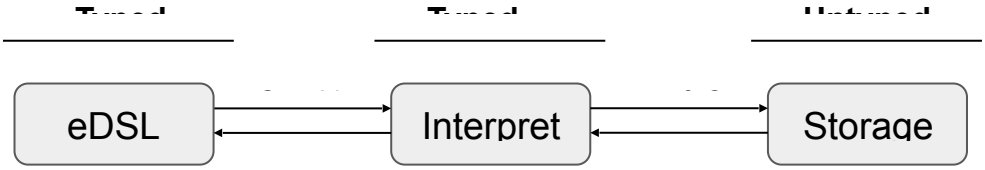


Figure 6.10: Typed and untyped layers

The implementation details are not 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 which 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
```

With 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 makes it easy to design any semantics for eDSLs and control every single aspect of its evaluation.

3. Useful patterns

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 there is not 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 be made? Should we return a list of strings from the function? Although it's possible, it's not a good decision. Log logging is the additional service, and why the domain-related functions should know about it?

```
-- Returns multiple log entries.
reportMeteor' :: ReportingChannel -> Meteor -> StateL [String]
```

There is 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 collecting (Listing 6.4):

Listing 6.4: 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 of this approach seem obvious:

- Need to keep additional variable 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.
- It's also not appropriate in the business logic.

- Finally, it only supports "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 layer down. We'll do the same essentially, but we'll be collecting messages in the interpreter's state rather than 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 any logging from a StateL script and do 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".

The next figure explains the idea schematically (Figure 6.11):

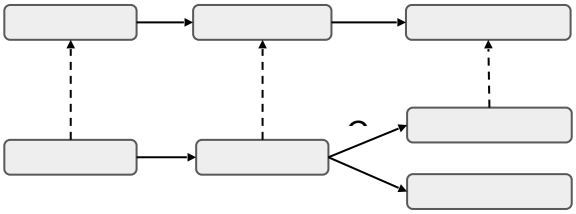


Figure 6.11: Schema of the "Delayed STM Logging".

Words are cheap, show me the code! Firstly, updates in the StateL:

```
data StateF next where
  -- Method for delayed logger:
  EvalStmLogger :: LoggerL () -> (() -> next) -> StateF next
```

Secondly, 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                    -- ^ Var id counter
    , _state  :: TMVar (Map VarId VarHandle)   -- ^ Node state
    , _stmLog :: TVar [LogEntry]               -- ^ Stm log entries
    }
```

Thirdly, add new STM-based interpreter for LoggerL, notice 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)
```

Fourthly, 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    -- Run STM interpreter
```

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
```

So now the business logic code will be looking pretty neat:

```
-- Transaction which updates log entries

anyStateFunc :: StateL (StateVar Int)
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  -- Logs will be flushed here automatically
  writeVar var 20
```

Reducing complexity with this pattern became possible because we abstracted the state, the logger and the language for business logic. This is a consequence of hierarchical organization of languages, too. It's pretty satisfying, isn't it?

NOTE. Well, I silently kept silent that there we cannot 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.

3.2. Reactive programming with processes

The attentive reader might have noticed that something is wrong with the app function in which we were starting our processes. Let me remind 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 to run the trachingCenter function infinitely many times. In reality this is usually not 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 etc.), 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 could adapt its behavior to the dynamically changing situation.

Let's first interrupt the evaluation of the app function when the total number of meteors exceeded 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
```

Now we need to react to the number of tracked meteors somehow. Currently, tracking center does not 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 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 these processes can interact? Using the signaling StateVars. There are a few schemes of this interaction involving either one or more StateVars. Deciding on which scheme to

choose may be a situational task. For our purposes, let's have two signaling variables. The code you'll see next is not the shortest way to do it but still it's good enough for demonstration.

The first signal variable - appFinished - will represent a condition for the whole application. Perhaps, AppState is the best way for it.

```
type SignalVar = StateVar Bool

data AppState = AppState
  { _reportingChannelVar    :: ReportingChannel
  , _catalogueVar           :: Catalogue

  -- When True, all child threads should finish.
  , _appFinished            :: SignalVar
  }
```

The second signal variable should be owned by a certain process. Using this variable, the process will notify the main thread it has finished. Let's see how the astronomer process should be reworked to support this "cancellation token" (see Listing 6.3 as 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
```

So once the process got a signal to quit, it setups his 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 finish with signaling. Now, need to rework the app script. Notice, we've moved trackingCenter to its own process and removed the forever combinator from everywhere:

```
app :: AppL ()
app = do
  let maxTracked = 1000                -- Finish condition
  st <- initState                      -- Init app state
  doneVar <- atomically $ newVar False  -- Signaling variable for astronomer

  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 the safety, wait a bit before really quit:
  delay 1000
```

The last block of the code will block the execution of the whole app until the readVar become 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                -- Finish condition
  st <- initState                      -- Init app state

  -- 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 possibilities to write a concise code that works very well, and signal vars is a pattern that can be used to do incredible things. There are of course some subtle things regarding a correct usage of STM and threads, yet these difficulties are not that big as programming bare threads with traditional approaches to concurrency.

4. Summary

In this chapter, we've learnt several approaches to concurrency in multithreaded applications. We've developed an architecture which allows to divide the application into layers: domain specific languages organized hierarchically, business logic and implementation layer. It was shown that there is a real value in separating of responsibilities. Thus, having a language LangL that cannot fork threads or directly work with impure mutable state made the application more structured and prevented the unwanted things like threads leakage. At least on the level of this language. Also, we've introduced a higher level language AppL that is able to fork processes and do the necessary initialization. This language should not be used for an actual business logic, but it declares the environment for this logic. Hierarchical eDSLs gave us another layering within a business logic layer and allowed to control the responsibilities more granular.

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, operational state for application.

We've wrapped STM and processes into own languages and embedded them into the hierarchy of Free Monad languages. This allowed us to abstract and retrospect the usage of state variables, to implement a kind of thread pool and avoid the leakage of resources. We were also able to implement an implicit yet transparent and obviously behaving logging for our StateL language.

Concluding, the chosen architecture of the application gave us many benefits and possibilities for future improvements.

7
Persistence

Links

- My email
- Book page
- Patreon campaign
- Paypal donations

This chapter covers

- Design of SQL DB and KV DB support
- Embedding a type safe SQL DB library
- How to create DB data models
- Type safety and conversion questions
- Transactions and pools

Being travelling in space, you encountered a cute solar system with a nice small planet near a bright star. When you investigated the planet, you discovered a life in it. Creatures were really social members, they were interacting but interacting in a very simple way. You could swear they were just a nodes in an unstructured tree, and they were connected by randomly displaced edges. You could easily find several roots, many intermediate nodes and millions of leaf nodes. What an interesting life! After you watched them for a while, you started realizing similar patterns repeating once and once again in their interactions. Are starting from the leaves; they are assigning a sequential index to all these leaf nodes; and then they start a calculation in the parent nodes. This calculation is rather simple. You found it's just a summation of all the children indexes; and once this procedure is done, the next level of nodes starts doing the same, and the calculation repeats till it hits the top root node. The number of the node which is essentially is a summation of all the indexes all the leafs was (-1/12). After they got the result, the tree with all nodes has disappeared, and a new, very much the same tree has been restored from the nearest gigantic bin. And all the calculations started over again.

Now think. Do you want a magic button that could save your life into an external storage so that you could occasionally take it back and restore your life? But the only condition will be 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 society more closely you found a problem that prevented their tree from being stored back to the bin once they finished their calculations. Fortunately you could fix this bug, and - magically, - the society continued to live further and develop their unique non-circularizing history.

And this means you probably can think of saving your life as well...

1. Persistence in FP

It's quite a rare case when a program does not interact with any external data storage whether it's a relational DB, Key-Value DB, file system, 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 cannot be located directly in the program's memory. Observing all the possible data storages and nuances of their usage is not possible even in hundred of

books dedicated for this purpose only. It's too much to consider and is not our interest. Why a KV cache should be taken over a relational database, what are the differences between various file storages, how to tame the beast of cloud services… Not this time, not in this book.

We're talking about design patterns and architecture approaches, not about external technologies. We can and we probably should challenge ourselves in making interaction with these systems more correct, robust, simple, convenient and so on. We would like to discuss the paths to abstraction of the related code. We might want to introduce a typical patterns to solve a typical problems. So we are here to see how the DB support can be implemented with the Free monad approach.

But we'll start with some basics.

2. Basics of DB Support

2.1. Domain Model and DB Model

The first lesson we need to learn is that DB Model is not the same as Domain Model, in general. The difference becomes important as fast as the application grows in size and complexity. Let's play a game in which we'll be arguing as Devil's Advocate protecting an opposite point of view: DB Model is the same as Domain Model, and there should be no distinction, as Figure 7.1 shows:

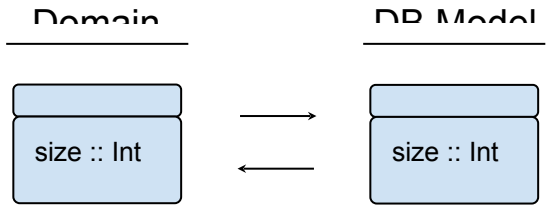


Figure 7.1: Domain Model and DB Model

Now we might be asked: what if there will be meteors having the same mass and size and happened to occur in the same region? One of them will be lost. Well, we don't have additional coordinates here that could make our meteors unique. Our "Region" type is too simple and fuzzy, it's not a dimension astronomers work with. Real coordinates like azimuth, altitude and time would be enough to distinguish one space stone from another. In fact, our domain model should be supplemented by measured coordinates as well. In the end, we are creating a soft for professionals, aren't we?

```
data Coords = Coords
  { _azimuth :: Degrees
  , _altitude :: Degrees
  }

data Meteor = Meteor
  { _size        :: Int
  , _mass        :: Int
  , _coords      :: Coords
  , _timestamp   :: DateTime
  }
```

We can now put meteors into our DB and do not be afraid to get them confused. We can query a meteor from DB if we know its coordinates, and we can always be sure there will be no other meteors with the same key.

In the physical world it's not possible for two different meteors to be in the same point of space at the same time (it's a collision of meteors - an extremely rare event), but our program is not a physical world. We could miss a bug in it that pushes the same meteor into the DB two or more times. Without uniqueness constraint on the key we'll have two identical rows about the same meteor. This can even happen if we use transactions. It's pretty easy to evaluate the same transaction twice by occasion. So we should manage key uniqueness manually or with the DB service help.

Using the Coords type as a natural composite key has some limitations. We expect a huge number of meteors that we should track. The natural key becomes less appropriate to use: it's not convenient for querying, requires extra memory in relations between tables, and not very maintainable. What if these fields should be slightly updated? Should we update all the data in the DB? And what about backward compatibility? More other 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. We can't know without considering other requirements:

- What is the supposed DB service and what are the requirements of this DB service to DB model, to primary keys, to relations?
- Is DB model essentially relational?
- Can it be described by a dictionary of something?
- What is the way to convert between DB model and domain model?
- Can the extra boilerplate be tolerated?
- Is it allowed to use more type level and compile time techniques to describe either of models?
- Should the DB model be generated from the domain model or vice versa?
- Or maybe there is an external definition of the DB model that should be parsed and converted?
- Will be a particular library used and whether it has additional requirements to the DB model?
- What are non-functional requirements such as performance, amount of data expected, what level of availability should be kept...

Our current task is to track meteors and other astronomical objects in a single database. There is no special requirement to it: we should be able to write and read stuff, and we're happy with a plain-old DB application. Seems we won't have a broad relational model for the domain, we need a simple warehouse to store objects, that's it. Can SQL DB be used here? Yes, definitely. We'd want to query meteors with particular properties like "get meteors which mass and size exceeds a dangerous threshold", or "get meteors occurred in that region, that period of time". Can we use KV DB then? Okay, it depends. NoSQL databases provide different mechanisms for querying, and also it's possible to construct indexes and relations specifically for certain needs. For ours, it's enough to have a simple DB support with minimum functionality. We could maybe treat SQL DB as a main storage and KV DB as a kind of cache.

What does this mean for our Domain and DB models? Let's just proceed with the following design. We'll define a template type for meteor and specify it for a particular usage. The key will be a varying part:

```
data Meteor' k = Meteor
  { _id          :: 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
```

The later development will show if this design is good enough, or we just invented something that does not comply with the KISS and YAGNI principles.

2.2. Designing Untyped KV DB Subsystem

It looks like a nice idea to demonstrate a way how we can substitute one DB service by another without affecting the main logic. This will force us to design a high level interface to the DB 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 KV DB systems:

- RocksDB. Embedded DB for KV data.
- Redis. Remote distributed in-memory KV DB storage.

While both these KV storage systems offer a must-have basis, - namely setting and getting a value by a key, - there are unique properties that differentiate the DB systems from each other. And this is a problem because we cannot unify these DBs 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 of capabilities is somehow represented in it, and if we had such a goal, we could even profit from selling such a solution. I'd do this! In the far, far future. When I became 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 etc. 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 KV storages must support.
- Typed higher level interface (eDSL) on top of untyped one. This one should allow to represent a DB model in a better way than just raw keys and values.

Enough words, let's code! In the following listing (Listing 7.1) you can see the simplest monadic KV DB language which has only two methods for saving and loading a value. Notice that both methods can fail for some reason, and we encode this reason as DBResult. It's description is presented in Listing 7.2:

Listing 7.1. Basic low-level string-based KV DB interface.

```
type KVDBKey = ByteString
type KVDBValue = ByteString

data KVDBF next where
  Save :: KVDBKey -> KVDBValue -> (DBResult () -> next) -> KVDBF next
  Load :: KVDBKey -> (DBResult KVDBValue -> next) -> KVDBF next

-- Free monadic language:
type KVDBL db = Free KVDBF

-- Smart constructors:
save :: KVDBKey -> KVDBValue -> KVDBL db (DBResult ())
save dbkey dbval = liftF $ Save dbkey dbval id

load :: KVDBKey -> KVDBL db (DBResult dst)
load dbkey = liftF $ Load dbkey id
```

And the type for DB result:

Listing 7.2. A type for DB result.

```
data DBErrorType
    = SystemError
    | KeyNotFound
    | InvalidType
    | DecodingFailed
    | UnknownDBError

data DBError = DBError DBErrorType Text

type DBResult a = Either DBError a
```

Notice that there is no information about DB connection here. In fact, DB 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 ping a DB constantly and notify a user as soon as the connection is broken. So many different requirements, so many decisions, and it's very unclear what decision is the best. There are hundreds of "best practices" being described, the opinions here may declare the opposite statements. This means there is no best practices at all.

This is why DB connectivity management is a hard question. I mean a bunch of hard questions. For example, should a connection be kept alive all the time, or it should be open only for a particular query? Is it safe to connect to the DB without disposing of connections? Has a DB system some timeout for idle connections? Will it be doing a reconnection automatically if the connection is lost? How can a DB system define that the connection has lost?

Well, I'd suggest you propose more solutions and justifications as an example in addition to the solution I have for you. The KVDBL language should abstract only operations with data: reading, writing, searching, updating, deleting and so on. It's not even matter if the underlying DB libraries do not separate a connection from an operation, we are free in choosing a proper semantics that will be more convenient, or, at least, less burden. As a sample, the rocksdb-haskell library (binding for RocksDB) requires a DB value that is 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 ()
```

But the hedis library has 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:
  runRedis conn $ do
     set "hello" "hello"
     set "world" "world"
```

Let's revise the structure of languages from the previous chapters. Here is a diagram of hierarchical Free monad languages:

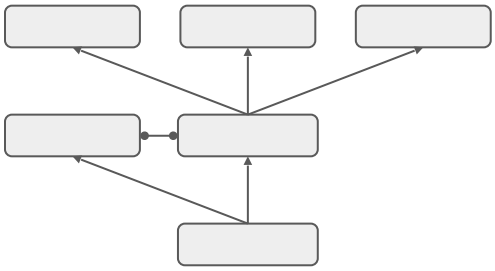


Figure 7.2. Hierarchical Free monad languages.

Now, where should we place the new language, KVDBL, and how the connectivity management should look like? For our case, 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 it's working with DBs is a significant part of business logic, where LangL is the main eDSL.

See in the following code: a method, additional type for DB connection (we call it DBHandle here), and some type class for denoting a DB that we'll discuss a bit later:

```
-- A class for denoting DB
class DB db where
  getDBName :: DBName

-- Types of DB (probably not a good idea to expose these details...)
data DBType
  = Redis
  | RocksDB

type DBName = String

-- A type representing a connection. It has a phantom type variable for more type safety.
data DBHandle db = DBHandle DBType DBName
```

```
-- Free monadic language for business logic
data LangF next where
  EvalKVDB :: DB db => DBHandle db -> KVDBL db a -> (a -> next) -> LangF next

type LangL = Free LangF

-- Smart constructor. Notice that DBHandle has a phantom type
-- denoting a particular KV DB.
evalKVDB :: DB db => DBHandle db -> KVDBL db a -> LangL a
```

Here, DBHandle can be parametrized by any data type that is an instance of the DB type class. In other words, the instance of the DB type class is always related to a single KV DB storage (a file in case of RocksDB and an instance in case of Redis). For the astronomical application, we might want to specify the following database:

```
data AstroDB

instance DB AstroDB where
  getDBName = "astro.rdb"
```

And here the DB name is exactly the file name. When we'll be working with this AstroDB type, the framework will take the responsibility to dispatch our calls to this real RocksDB storage. See Section 7.3.2 for detailed information about the usage of this AstroDB phantom type.

Now we have a complete interface to run KV DB actions (non-transactionally), but we don't have any method that we could call and obtain a connection to a DB storage. The two possible solutions can be imagined:

1. Connect to a DB storage outside of the business logic, namely, on 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 languages structure we have currently (see 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 DBs 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 DB storage, obtains a long-living connection and passes it into the logic. This option gives more flexibility than the option #1, but less flexibility than option #2-a.

In here, we'll take the option #2-a. Take a look at the AppL language and some additional types:

```
-- Config for connection to KV DB storage. Currently supports two KV databases.
data KVDBConfig db
  = RedisConfig
  | RocksConfig
      { _path            :: FilePath
      , _createIfMissing :: Bool
      , _errorIfExists   :: Bool
      }

-- Language with necessary methods
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

-- Smart constructors.
initKVDB :: DB db => KVDBConfig db -> AppL (DBResult (DBHandle db))

deinitKVDB :: DBHandle db -> AppL (DBResult ())
```

Let's update the diagram by adding a new language on it. See Figure 7.3:

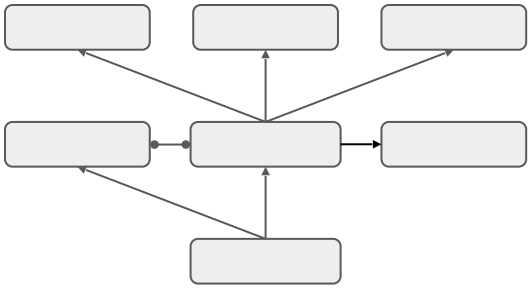


Figure 7.3. Hierarchical Free Monad Languages with KVDBL.

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 DB connection. Having knowledge from Chapter 5 and Chapter 6, you might have guessed that this type is just another "avatar" - an abstraction, a bridge, a key to access the real connection. Abstracting these types has a particular goal: to separate implementation from interface. Abstractions - in the form of types and languages - are crucial to have a controllable, manageable and testable systems. We'll talk about this in Chapter 10 along with establishing test practices and approaches.

2.3. Abstracted Logic vs Bare IO Logic

Back to the DB system, we should clearly separate interfaces to the implementation. As usual, the types and eDSLs we've introduced should not sop impurity and details from the implementation layer. It's too hard to abstract each aspect of a native library, but at least we did this for a simple save / load functionality. Now let's transform it into real calls.

Interpreting the KVDBL language against both hedis and rocksdb libraries is very similar. For example, the following interpreter just maps the methods to the implementation functions:

```
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. Except that when we'll get into the `RocksDBImpl` module, we'll see some difficulties. The get and put methods do not return success or failure, they either succeed or throw an exception. We have to convert the underlying native types to the types we use in the Free monad languages. In this sense, a Free monad language acts as the `Adapter` pattern. Luckily, the type for key and value is `ByteString` in our language and in `rocksdb`. Unluckily, the `get` and `put` native methods do not return a failure type but rather throw an exception. We handle it by catching and wrapping into our `DBResult` type presented earlier. Other possible solutions may propose to wrap the underlying unsafe resource (like real DB calls) into `ExceptT` monad, or maybe organize a kind of "exception areas" with additional Free monad languages. As for the current code, there is an administrative contract that exceptions are prohibited and should be caught as soon as possible. See a sample implementation of the get function:

```
-- In module Hydra.Core.KVDB.Impl.RocksDB:

get :: Rocks.DB -> KVDBKey -> IO (DBResult KVDBValue)
get db key = do

  -- Real call and catching an exception
  eMbVal <- try $ Rocks.get db Rocks.defaultReadOptions key

  -- Conversion of result
  pure $ case mbVal of
    Left (err :: SomeException) -> Left $ DBError SystemError $ show err
    Right Nothing -> Left $ DBError KeyNotFound $ show key
    Right (Just val) -> Right val
```

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 KVDBValue)
get conn key = Redis.runRedis conn $ do
  fStatus <- Redis.get key
  pure $ case fStatus of
    Right (Just val) -> Right val
    Right Nothing -> Left $ DBError KeyNotFound $ decodeUtf8 key
    Left (Redis.Error err) -> Left $ DBError SystemError $ decodeUtf8 err
    res -> Left $ DBError UnknownDBError $ "Unhandled response from Redis: " +|| res ||+ "."
```

You might ask, what argument I'm trying to build here. Did we abstract the types? Yes, we did. 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 our own DBResult type. Easy. Abstractions matter, even if there is no requirement of having several implementations of the subsystem. Even in cases when you need only Redis, writing a business logic using bare hedis functions and types will pollute the code by unnecessary knowledge and extra complexity.

Let's do a bit more investigation of the design approaches and compare abstracted and non-abstracted versions of the KV DB subsystem. A sample scenario written in the KVDBL free monadic language will be the following:

```
-- Saving and loading methods
save :: KVDBKey -> KVDBValue -> KVDBL db (DBResult ())
load :: KVDBKey -> KVDBL db (DBResult dst)
-- Wrapper function
evalKVDB :: DB db => DBHandle db -> KVDBL db a -> LangL a

-- Phantom type representing a particular DB storage
data ValuesDB

instance DB ValuesDB where
  getDBName = "raw_values.rdb"

-- Loads two strings values and appends them.
loadStrings :: DBHandle ValuesDB -> DBKey -> DBKey -> LangL (DBResult DBValue)
loadStrings db keyForA keyForB = evalKVDB db $ do
  eA <- load keyForA
  eB <- load keyForB
  pure $ (<>) <$> eA <*> eB
```

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:

```
-- Bare RocksDB version of the loadAndAppendStrings function:
loadStringsRocksDB :: Rocks.DB -> DBKey -> DBKey -> IO (DBResult DBValue)
loadStringsRocksDB db keyForA keyForB = do
  eA <- load' db keyForA      -- Wrapper function, defined below
  eB <- load' db keyForB
  pure $ (<>) <$> eA <*> eB

-- Wrapper function
load' :: Rocks.DB -> KVDBKey -> IO (Either String KVDBValue)
load' db key = do
  eMbVal <- liftIO $ try $ Rocks.get db Rocks.defaultReadOptions key
  pure $ case mbVal of
    Right (Just val) -> Right val
    Right Nothing -> Left "Key not found"
    Left (err :: SomeException) -> Left $ show err
    _ -> Left "Unknown error"
```

This similarity between abstracted and bare scenarios is actually deceptive. Let's compare side-to-side.

Table 7.1: Comparison of bare IO scenario and abstracted scenario

Abstracted scenario	Bare IO scenario
Does not depend on a particular DB implementation	Highly coupled with a particular library (rocksdb-haskell)
Abstracts any kinds 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 to run any effect at any place - less guarantees
As simple as possible, no additional constructions and syntax	Somewhat burden by implementation details (liftIO, try, Rocks.DB etc)
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.

A very big temptation of keeping all possible levelers handy (in bare IO code) comes into conflict with the idea of delegating the control to where it can be served better. This temptation ingrains a false belief that we can rule the code finely. And when we've caught, we got blinded and cannot see anymore 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 a high accidental complexity as the result. On the other hand, in software design there is no abstractly 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 of applying 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 aftermath of a sudden death.

2.4. Designing SQL DB Model

When a developer starts thinking about the need for SQL database in his project, he immediately steps onto the difficulties that imply activities going 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 the needs can easily scupper the project in the near future. After some requirements obtained, the developer has to take architectural decisions on what DB storage should be used, how it should be accessed, what libraries are available, and what approach to DB management to follow. Irrespective of the way the developer will be designing a database model, he will meet the challenges in defining a DB schema, tables, relations, normalization, indexes. He will be deciding how much logic should be placed into stored procedures and views and how much logic should be implemented as a regular code. He will also need to write a bunch of necessary SQL queries, provide some views, server-side functions to support the logic. This all is a considerable activity that requires a set of specific skills such as knowing of relational algebra, SQL, normal forms, indexes work, decomposition patterns. For a better performance, one also would have used the specificity of a particular DB storage, its unique features and capabilities. Not a surprise that we have a separate database developer role for designing, managing and implementing such solutions. This all feels very enterprisy; although the industry has become way too complicated with its hundreds of database solutions and practices, and attempted to step back from the relational DBs, the latter is still a major discipline that is impossible to get rid of in real practice.

As software architects, we are interested in knowing these nuances from the perspective that it's directly related to the complexity and risk management. Choosing the ways of database implementation influences the complexity of relational models a lot. This complexity is unavoidable in sense you have to implement tables and relations somehow, but it can be between the three points: DB storage, intermediate layer and DB model. The next figure shows those three points:

Figure 7.4

DB Storage <-> Intermediate Layer <-> DB Model

When the developer starts designing the DB subsystem, he may decide how much complexity he wants to place into the DB Model, how much complexity the intermediate layer should take, and how much complexity should be placed into the DB storage. The intermediate layer here can be either thin with a minimum of own logic or thick with internal mechanisms making a mapping between DB Model and DB Storage. In OO languages the idea of ORM (Object Relational Mapping), can be used for the intermediate layer. ORM systems seem to be overkill for small programs, but it's hard to build big applications without them. Supporting of the code made using bare string-based queries without any DB schema description is really difficult. So the mainstream developers have to balance the accidental complexity coming with the 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 7.2: 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

So now we have enough background to start designing the SQL DB 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 a very strong experience in type-level design and good understanding of relational algebra and SQL DB related stuff. So here we have to choose one of the existing solutions and use it.

In Haskell, there is a bunch of libraries abstracting the access to SQL databases with more or less type magic:

- Selda
- Esqueleto
- Beam
- Persistent
- ...

There are less complicated libraries, and libraries providing some interface to particular DB storages, like postgres-simple. But it seems the complexity of defining SQL queries and DB models is unavoidable in Haskell. We want to have a type-safe DB model, and we can't express a mechanism of mapping in a simple manner. Haskell's type system is very powerful, yet very mind blowing and wordy. There is 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 still it's 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 by simple cases only.

Firstly, we create a DB model for our domain. This model is not a part of the framework, but rather a part of the actual logic. Beam requires defining 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 a parameterized type that 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 algebraic data type. There can be other primary key data types, but this particular works for DBMeteorT. It can't be confused with other tables. We should give a name for 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 into the constructor (of type PrimaryKey DBMeteorT f). As you can see, there are some extra words in this definition which grab the attention and make the code burden a bit. 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, 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 a specific SQL DB storage only (beam calls it "database backend). For example, SQLite:

```
data CatalogueDB f = CatalogueDB
  { _meteors :: f (TableEntity DBMeteorT)
  }
  deriving (Generic, Database Sqlite)
```

But hey, we will be considering our schema as a DB agnostic one, so let it be **be**.

The preparation of the DB model finishes in defining two types for convenience:

```
type DBMeteor = DBMeteorT Identity
type DBMeteorId = PrimaryKey DBMeteorT Identity
```

Don't confuse this DBMeteorId type with a data constructor DBMeteorId from the associated type PrimaryKey! 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 is for example a query that selects all meteors having a predefined mass:

```
import qualified Database.Beam.Query as B

selectMeteorsWithMass size
  = B.select                                          -- Denotes a SELECT query
```

```
  $ B.filter_ (\meteor -> _size meteor ==. B.val_ size)   -- WHERE clause condition
  $ B.all_ (_meteors catalogueDB)                        -- 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 are 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 kind of (QExpr Bool). The beam library provides several boolean-like comparison operators: (==.), (&&.), (||.), (>=.), (<=.), not_, (>.), (<.) and others. Should not be a problem to use them.

The `selectMeteorsWithMass` query does not query any particular database. The query itself is database-agnostic. We can assume that the query will be correctly executed on any DB 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 which is 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 hopefully the developers working with the framework can be liberated from additional knowledge.

Unfortunately, having the runBeamSqlite function is not 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'll decide to learn beam design, please be prepared for a mind-blowing journey into the type level magic. We would like to avoid this complexity in our business logic. The runSelectReturningList should be also hidden and abstracted, especially its type. Consider the following final call stack composed from all of those functions to just select some meteors:

```
import qualified Database.Beam.Query as B (runSelectReturningList)
import qualified Database.Beam.Sqlite as SQLite (runBeamSqlite)
import qualified 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, declaring of the queries and execution steps in beam is done with Church Encoded Free Monad. 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, a 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 and do not increase accidental complexity too much. And before we assemble a mechanism for this, let's take a look ahead and foresee a client code working with such SQL DB subsystem. In the next listing (Listing 7.3), an AppL scenario is presented in which there is a connection procedure, and a query embedded into the AppL and FlowL languages:

Listing 7.3. Sample of business logic that uses SQL DB language.

```
meteorsApp :: AppL ()
meteorsApp = do
  eConn <- initSQLiteDB "./meteors.db"   -- initSQLiteDB is provided by the framework
  case eConn of
    Left err -> logError "Failed to init connection to SQLite."
    Right conn -> do
      meteors <- getMeteorsWithMass conn 100         -- see this function below
      logInfo $ "Meteors found: " <> show meteors

getMeteorsWithMass :: SQLite.Connection -> Int -> AppL [DBMeteor]
getMeteorsWithMass sqliteConn size = do
  eMeteors <- scenario
    $ evalSQLiteDB sqliteConn      -- evalSQLiteDB is provided by the framework
    $ runBeamSelect                -- runBeamSelect is provided by the framework
    $ selectMeteorsWithMass size   -- defined earlier
  case eMeteors of
    Left err -> do
      logError $ "Error occurred when extracting meteors: " <> show err
      pure []
    Right ms -> pure ms
```

Now we're going to add some new functionality to the framework that makes the shown code possible.

2.5. Designing SQL DB Subsystem

The language from the Listing 7.3 supports only the SQLite DB backend. The corresponding scheme of embedding the languages looks the following, see Figure 7.5 (bold is from the beam library):

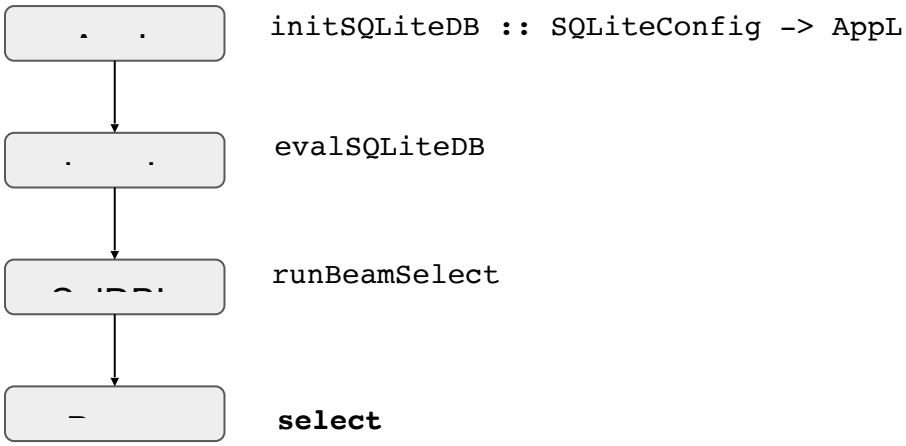


Figure 7.5: Embedding of SQL language scheme

The InitSQLiteDB method is quite simple. It accepts a path to the DB file and possibly returns a native connection. In case of error, 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
  case eConn of
    Left (err :: SomeException) -> pure $ Left $ DBError SystemError $ show err
    Right conn -> pure $ Right conn
```

Just a side note: catching the exceptions in the interpreter and not allowing them to leak outside a particular method seems a good pattern because we can reason about the error domains due to this. If all the exception sources are controlled on the implementation layer, the interfaces (our Free monadic eDSLs) and scenarios become predictably behaving. 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. Doing so we divide our application into so called error domains and thus we get an additional separation of concerns. Believe me, you'll be well praised for such an order in your application very soon in the form of low accidental complexity, increased robustness and more obvious code.

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 could transfer the evaluation into the implementation layer (into the interpreters). This is important because we should not allow the methods of beam to be directly called from the scenarios. They are essentially impure, and also have a clumsy interface with all those type level magic involved. We can provide a little bit more convenient interface.

Ok, checkout 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 the language is parametrized by a phantom type Sqlite (from the corresponding beam-sqlite library). Unfortunately, this design won't be that generic as we would like to. The Sqlite phantom type leaks into our business logic which is not good. We'll improve this approach in the next paragraphs and also will make so that we could run SqlDBL not only with SQLite but with any DB backend preliminarily chosen. For now Sqlite is an implementation detail the business logic depends on. As well as the native SQLite Connection type.

Now, the SqlDBL Free monad language. It's tricky because it tracks the info about a particular DB backend in types. In our case it's SQLite but 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 would not describe the beam library here. Just take for granted that we deal with these instances in such a way as shown to make it compile and work. If we decided to choose another library for the low-level SQL DB engine we could face another 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, 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. The following implementation is simplified compared to what you can find in Hydra for better demonstrability. Look how we run a function runBeamSqliteDebug which we can easily recognize as an implementation-specific function. It directly works with impure subsystem, it does some debug logging and works in the IO monad. The client code should not bother about this function, so we place it into the interpreter.

```
import qualified Database.Beam.Sqlite as SQLite
import           Database.Beam.Query (runSelectReturningList)
import           Database.Beam.Sqlite.Connection (runBeamSqliteDebug)
import           Database.Beam.Sqlite (Sqlite)

-- Interpreter that works with a native connection and impure subsystems directly.
interpretSQLiteDBF :: (String -> IO ()) -> SQLite.Connection -> SqlDBF Sqlite a -> IO a
interpretSQLiteDBF logger conn (RunBeamSelect selectQ next) = do
  rs <- SQLite.runBeamSqliteDebug logger conn
      $ runSelectReturningList
      $ selectQ
  pure $ next $ Right rs

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 DB 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 is another function available in the library: `runBeamSqlite`. It doesn't print anything, if 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 is 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 transactional evaluation of queries. It's possible to support transactions, we just focussed on the idea of embedding an external complex eDSL into our languages hierarchy.

So this is it. Happy querying SQLite DBs in your business logic!

Or read the next sections. We have some limitations in our SQL DB mechanism. Let's design a better support for this subsystem.

3. Advanced DB Design

In this section we'll continue discussing the structure and implementation of DB programs. Some questions are still uncovered:

- Support of many different backends for SQL DB subsystem
- Transactions in SQL DB

- A higher-level model for KV DB and type machinery for a better type safety of the mapping between a domain model and KV DB model
- Transactions in KV DB
- Pools and resources

The following sections will provide more approaches and mechanisms we can build on top of Free monads, and probably you'll 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 is an implementation detail and what is not, and thus we're gaining a great power against the complexity of the code. Which is our primary goal: do not bring more complexity than it's really needed.

3.1. Advanced SQL DB Subsystem

The beam library supports Postgres, SQLite and MySQL. Keeping apart 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 for interacting with any SQL DB backend we need. Our goal is to enable such business logic as it shown in the next listing (Listing 7.4):

Listing 7.4. Sample of business logic that uses an advanced SQL DB language.

```
-- Create a config for a particular DB backend
sqliteCfg :: DBConfig SqliteM
sqliteCfg = mkSQLiteConfig "test.db"   -- mkSQLiteConfig is provided by the framework

-- Try to connect using the connection.
-- Hardly fail with exception (do not call `error` in your real code!)
connectOrFail :: AppL (SqlConn SqliteM)
connectOrFail cfg = do
  eConn <- initSqlDB sqliteCfg    -- initSqlDB is provided by the framework
  case eConn of
    Left e     -> error $ show e  -- Bad practice!
    Right conn -> pure conn

-- A SELECT query to extract some meteors by condition (defined earlier).
selectMeteorsWithMass size
  = Beam.select                                                -- Denotes a SELECT query
  $ Beam.filter_ (\meteor -> _size meteor ==. Beam.val_ size)   -- WHERE clause condition
  $ Beam.all_ (_meteors catalogueDB)                           -- A kind of FROM clause

-- Program that creates a connection and extracts a single meteor from the DB.
dbApp :: AppL ()
dbApp = do
  conn <- connectOrFail
  eDBMeteor <- scenario
        $ evalSqlDB conn             -- More generic version for running a beam query.
        $ findRow                    -- A wrapper function, provided by the framework.
        $ selectMeteorsWithMass 100  -- Select query
  case (eDBMeteor :: Either DBError DBMeteor) of
    Left err       -> logError $ show err
    Right dbMeteor -> logInfo  $ "Meteor found: " <> show dbMeteor
```

In the previous simplified approach we created the initSQLiteDB function, which is rigid and specific. No, it's not bad to be precise in the intentions! Not at all. It's just that we want to move a DB backend selection into another place. Now we'll be specifying it by filling a config structure. Consider these new types and helper functions:

```
-- A tag for a particular connection
type ConnTag = String

-- A wrapper type for different DB backends. Contains a phantom type beM.
-- Should not be created directly (should be an abstract type).
data DBConfig beM
  = SQLiteConf DBName
  | PostgresConf ConnTag PostgresConfig     -- config type, omitted for now
  | MySQLConf ConnTag MySQLConfig           -- config type, omitted for now
```

```
-- A wrapper type for different connections
data SqlConn beM
  = SQLiteConn ConnTag SQLite.Connection
  | PostgresConn ConnTag Postgres.Connection
  | MySQLConn ConnTag MySQL.Connection
```

Where beM is a specific monad for any beam backend. At the moment, we have several monads for Postgres, SQLite and MySQL, and these monads are certainly a 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. 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 probably some of them are much better, but who knows.

Now, considering 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, MySQLConn`), why should we duplicate it in such manner. Okay, this is fair. Suggestions are welcome! It might be possible to deduce a monad type for backend by just pattern match 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:

- Should a duplicated connection be allowed, or the attempt to call InitSqlDB twice should fail;
- If it's allowed, then should a new connection be created, or the old one has to be returned;
- And if the old one is returned, then what about a multithreaded environment? How to avoid data races?

Ohhh… Pain. 70 years of programming discipline, 60 years of functional programming paradigm, 30 years of a massive industrial development, 10 years of a Haskell hype, and we're still solving the same problems, again and again, in different forms, in different environments. We are extremely good at inventing existing techniques and repeating the 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 DB backend agnostic, and a real DB backend is 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    -- Real runner and real connection
  $ B.runSelectReturningList           -- Expected structure
```

```
  $ B.select                          -- Query
  $ B.all_ (_meteors catalogueDB)     -- Query conditions

queryPostgresDB :: Postgres.Connection -> IO [DBMeteor]
queryPostgresDB pgConn
  = Postgres.runBeamPostgres pgConn    -- Real runner and real connection
  $ B.runSelectReturningList           -- Expected structure
  $ B.select                           -- Query
  $ B.all_ (_meteors catalogueDB)      -- 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             -- Connection
            -> ??? query here ???      -- Beam query embedded here
            -> (DBResult a -> next)    -- Expected result
            -> LangF next
```

Should we construct the call stack like this? Pseudocode:

```
queryDB :: SqlConn beM -> LangL [DBMeteor]
queryDB conn
  = evalSqlDB conn                     -- Method from the AppL language
  $ B.runSelectReturningList           -- Expected structure
  $ B.select                           -- Query
  $ B.all_ (_meteors catalogueDB)      -- Query conditions
```

The runSelectReturningList has the following type:

```
runSelectReturningList
  :: (MonadBeam be m, BeamSqlBackend be, FromBackendRow be a)   -- Beam constraints
  => SqlSelect be a                                             -- Beam type
  -> m [a]                                                      -- Some monad
```

Embedding it into the method EvalSqlDB requires you to keep all those type level bits (constraints). Pseudocode:

```
data LangF next where
   EvalSqlDB :: (MonadBeam be m, BeamSqlBackend be, FromBackendRow be a)   -- Unexpected
complication
            => SqlConn beM                                                 -- Connection
             -> (SqlSelect be a -> m [a])                                    -- Beam query embedded
here
            -> (DBResult a -> next)                                        -- Expected result
            -> LangF next
```

That's… d-i-f-f-i-c-u-l-t. And by the way, the observation: we just forgot about all other query types like Insert, Update, Delete. Embedding the beam facilities this way is technically possible but it's for the 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.

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. Let me remind: previously, the interpreters were the only modules aware of the real functions and libraries. Here, we put all the beam queries, types and and real runners into eDSLs directly, making the language modules to depend on these details.

Moreover, we'll be pre-evaluating the beam functions in order to reduce the need for passing of type classes. For example, if we have the following function from beam:

```
runInsert :: (BeamSqlBackend be, MonadBeam be m) => SqlInsert be table -> m ()
```

then we'll hide it behind our own type class:

```
-- in module Hydra.Core.Domain.SQLDB:

import qualified Database.Beam                    as B
import qualified Database.Beam.Backend.SQL        as B

class (B.BeamSqlBackend be, B.MonadBeam be beM) => BeamRuntime be beM where
  rtInsert :: B.SqlInsert be table -> beM ()
  -- more methods here: rtDelete, rtUpdate, rtSelectReturningList, rtSelectReturningOne
```

and then put its pre-evaluation version into the following type (a helper type for the SQL DB wrapping language):

```
data SqlDBAction beM a where
  SqlDBAction :: beM a -> SqlDBAction beM a

-- asking the BeamRuntime type class to provide a specific insert action:
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. The further reduction of the type classes happens in the Free monadic language itself. Let's figure it out:

```
data SqlDBMethodF beM next where
  SqlDBMethod :: (SqlConn beM -> (String -> IO ()) -> IO a) -> (a -> next) -> SqlDBMethodF beM
next

type SqlDBL beM = F (SqlDBMethodF beM)

-- Reducing the need in the beam machinery and type classes:

getBeamRunner' :: (BeamRunner beM, BeamRuntime be beM)
  => SqlConn beM -> SqlDBAction beM a -> ((String -> IO ()) -> IO a)
getBeamRunner' conn (SqlDBAction beM) =
    getBeamDebugRunner conn beM      -- asking the BeamRunner to provide a specific runtime
```

Now there is nothing about the beam at all, 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
```

However it's not easy to avoid the beam machinery for the queries themselves (types SqlSelect, SqlInsert, SqlUpdate, SqlDelete), so we're trying at least simplify this interface by introducing more wrappers:

```
sqlDBMethod :: (BeamRunner beM, BeamRuntime be beM) => SqlDBAction beM a -> SqlDBL beM a
sqlDBMethod act = do
  let runner = \conn -> getBeamRunner' conn act   -- Storing only the IO action in the language
  liftFC $ SqlDBMethod runner id

-- wrapper and more or less convenient function
insertRows :: (BeamRunner beM, BeamRuntime be beM) => B.SqlInsert be table -> SqlDBL beM ()
insertRows = sqlDBMethod . insert''

-- more wrappers:

findRow :: (BeamRunner beM, BeamRuntime be beM, Beam.FromBackendRow be a)
  => SqlSelect be a -> SqlDBL beM (Maybe a)

findRows :: (BeamRunner beM, BeamRuntime be beM, Beam.FromBackendRow be a)
```

```
  => SqlSelect be a -> SqlDBL beM [a]

updateRows :: (BeamRunner beM, BeamRuntime be beM)
  => B.SqlUpdate be table -> SqlDBL beM ()

deleteRows :: (BeamRunner beM, BeamRuntime be beM)
  => SqlDelete be table -> SqlDBL beM ()
```

After these wrappings, the SqlConn type specifies the needed SQL backend, the two new type classes BeamRunner and BeamRuntime help to select the corresponding evaluation functions, and wrappers are used to provide more sane interface which is DB backend-agnostic. Figure 7.6 represents a schema of the new design (bold is a beam-related stuff):

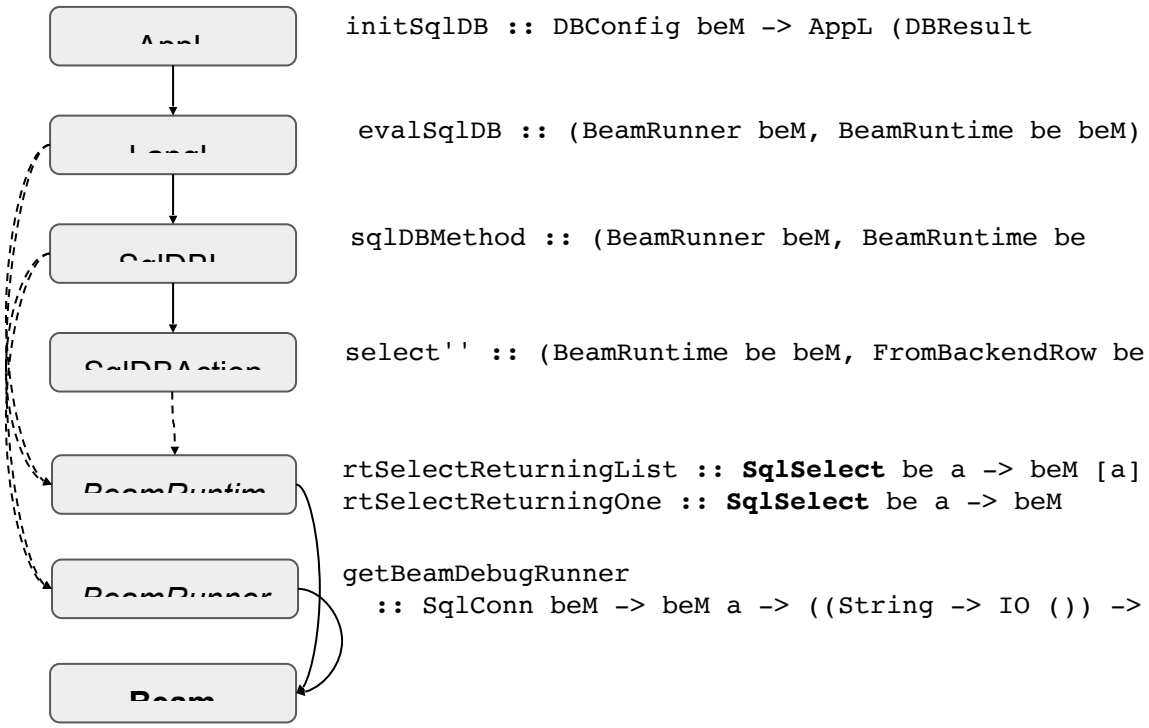


Figure 7.6: Advanced languages structure

And now the 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 that fast nevertheless. Sometimes we have to make dirty and imperfect decisions - just to be able to move further. Don't forget we have more goals other than a pursuit of perfection. It's better to have something at least reasonable and working rather than perfect but not yet ready.

Right?

Absolutely, read the next section for proof.

3.2. Typed KV DB Model

We designed an untyped KV DB subsystem in which all the data (keys, values) is represented by just ByteStrings. The KV DB subsystem does not support any strictly typed data layout, and thus there is no guarantee that the data from a KV DB will be treated correctly. Fine; we designed a working mechanism and can just start writing some real code interacting with Redis or RocksDB. Neither perfect, nor smart system that forces us to convert data manually. Still, this raw KV DB interface with bare ByteStrings covers 90% of our needs, it's very fast and fairly simple. And now, when we have more time, we may create a typed KV DB 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 KVDB language from the 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 DB model. We want to distinguish these KV DB entities from each other. Okay, the underlying framework methods will be untyped, but our custom KV DB model will be typed. Let's say we want to load some meteor from the DB by its key, and we know its stored as a stringified JSON object. This is an example of such rows in the DB in 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\"}")
("42",    "{\"size\":    300,    \"mass\":    300,    \"azmt\":    1,    \"alt\":    0,    \"time\":
\"2019-12-04T00:32:00\"}")
```

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. Its idea - to serialize and deserialize DB entities manually:

```
data KVDBMeteor = KVDBMeteor
  { size  :: Int
  , mass  :: Int
  , azmt  :: Int
  , alt   :: Int
  , time  :: DateTime
  }
  deriving (Show, Eq, Ord, Generic, ToJSON, FromJSON)

-- Tag-like type representing a particular DB storage
data AstroDB

instance DB AstroDB where
  getDBName = "astro.rdb"

saveMeteor :: DBHandle AstroDB -> MeteorID -> KVDBMeteor -> AppL (DBResult ())
saveMeteor astroDB meteorId meteor = scenario $ evalKVDB astroDB
  $ save
      (show meteorId)            -- converting integer id to ByteString
      (show meteor)              -- converting meteor to ByteString

loadMeteor :: DBHandle AstroDB -> MeteorID -> AppL (DBResult KVDBMeteor)
loadMeteor astroDB meteorId = scenario $ evalKVDB astroDB $ do
  eDBMeteorStr <- load $ show meteorId                -- converting integer id to ByteString
  case eDBMeteorStr of
    Right meteorStr -> pure $ parseMeteor meteorStr    -- parsing function, see below
    Left err -> pure $ Left err

parseMeteor :: ByteString -> DBResult KVDBMeteor
parseMeteor meteorStr = case decode meteorStr of
    Nothing -> Left $ DBError DecodingFailed "Failed to parse meteor data"
    Just m -> Right m
```

Listing 7.5. Manual serialization and deserialization of DB entities.

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 RecordWhildCards 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 why we want to abstract the conversions and data loading / saving is to have a generic way to work with any DB structures even if we don't know about them yet. Some readers may 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 the mainstream development, but the name "Expression Problem" is not spread 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. The class inheritance or duck typing solve the extensibility problem for most of the cases. On the other hand, developers on strongly typed functional languages without subtype polymorphism (such as Haskell) experience difficulties implementing such mechanisms. We not only need to think 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 a 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 now very impressive things but still it seems we're not satisfied enough. There is a common idea that we need even more powerful tool: 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. Fortunately there is a lot of materials about type-level design, and we'll return to this question in the next chapters.

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 the scenarios, but we want guarantees that conversion works fine. Here, if we saved a value using the new typed mechanism, we are 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 KV DB:

```
loadMeteor
  :: DBHandle AstroDB
  -> MeteorID
  -> AppL (DBResult Meteor)
loadMeteor astroDB meteorID = scenario
  $ evalKVDB astroDB
  $ loadEntity              -- a higher level function that understands a typed KV DB Model
  $ mkMeteorKey meteorID    -- service function that converts MeteorID to a typed key
```

You can guess that the new mechanism is hidden behind the functions `loadEntity` and `mkMeteorKey`. We cannot move further without definition of them, so:

```
-- A function that knows how to load a raw bytestring data
-- from a KV DB and convert it to the target type
loadEntity
  :: forall entity dst db
   . DBEntity db entity            -- DBEntity? What is this?
  => AsValueEntity entity dst      -- And what is this??
  => KeyEntity entity              -- And this?
  -> KVDBL db (DBResult dst)

-- A function that converts a meteor integer identifier
-- into the key type suitable for typed KV DB machinery
mkMeteorKey :: MeteorID -> KeyEntity MeteorEntity
```

Wait… What?! So many new concepts emerged from the listing. For those of you who are familiar with type level programming this picture won't be a surprise. Every time we're going there, we have to deal with type classes, functional dependencies, type families and type-level-related extensions.

The scheme on Figure 7.7 helps to understand the type classes and associated types that we'll be talking about.

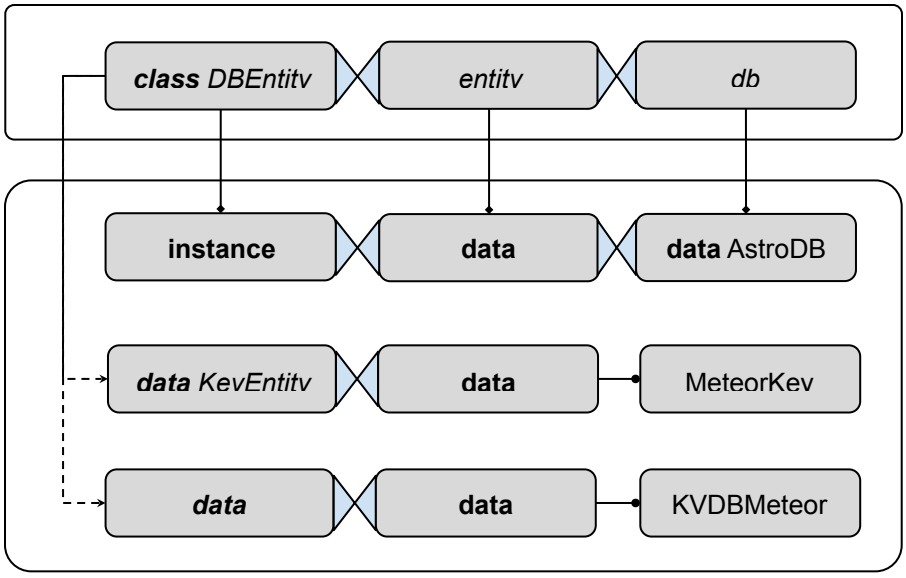


Figure 7.7: KV DB Type Level Machinery

You can see a specific DBEntity type class that is an entry point into the type level models for KV DB.

```
class DBEntity db entity
  | entity -> db where           -- Functional dependency: entity should belong to a single db
  data KeyEntity entity :: *     -- Associated algebraic data type
  data ValueEntity entity :: *   -- Associated algebraic data type
```

With this type class we model:

- Relation between a DB and an entity. Functional dependency "entity -> db" says that this entity can only be related to a single DB. Once you have a type for an entity, you'll immediately know the DB.
- Two associated types KeyEntity and ValueEntity. Details of these types will be known whenever you have a specific entity type.

Instantiating of this type class for two specific types as DB and entity allows to provide information on how to convert the entity to and from 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.

```
-- Tag-like type representing a particular DB storage
data AstroDB

-- Tag-like type representing a particular DB entity
data MeteorEntity

-- Instance
instance D.DBEntity AstroDB MeteorEntity where

  data KeyEntity MeteorEntity     -- Specific ADT, can be accessed as "KeyEntity MeteorEntity"
          = MeteorKey MeteorID    -- Value constructor of this associated ADT
    deriving (Show, Eq, Ord)      -- We can show and compare this ADT

  data ValueEntity MeteorEntity   -- Specific ADT, can be accessed as "ValueEntity MeteorEntity"
          = KVDBMeteor            -- Value constructor of this associated ADT
          { size  :: Int
          , mass  :: Int
          , azmt  :: Int
          , alt   :: Int
          , time  :: D.DateTime
          }
          deriving (Show, Eq, Ord, Generic, ToJSON, FromJSON)   -- Can be serialized with aeson
```

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 algebraic data type. We can think of this data type as separate ADT:

```
data KeyEntity MeteorEntity = MeteorKey MeteorID
```

The same is for value type ValueEntity MeteorEntity. It can be seen as a separate ADT, and once you have this module imported, you can operate by 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 KV DB model. Imagine 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 DB entity:

```
-- Type class with methods; associated types are omitted for succinctness
class DBEntity db entity | entity -> db where
  toDBKey     :: KeyEntity entity -> KVDBKey
  toDBValue   :: ValueEntity entity -> KVDBValue
  fromDBValue :: KVDBValue -> Maybe (ValueEntity entity)

-- Instance; associated ADTs are omitted for succinctness
instance DBEntity AstroDB MeteorEntity where
  toDBKey (MeteorKey idx) = show idx              -- MeteorID to ByteString
  toDBValue valEntity = encode valEntity          -- KVDBMeteor constructor to ByteString
  fromDBValue strEntity = decode strEntity        -- From ByteString to KVDBMeteor
```

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, the last piece of the puzzle goes. Two more helper type classes will show the relation between KV DB model and Domain Model. The type classes are simple:

```
-- With this type class, we can convert any arbitrary src type to the associated type KeyEntity.
class AsKeyEntity entity src where
  toKeyEntity :: src -> KeyEntity entity

-- With this type class, we can convert any arbitrary src type to and from the associated
ValueEntity type.
class AsValueEntity entity src where
  toValueEntity   :: src -> ValueEntity entity
  fromValueEntity :: KeyEntity entity -> ValueEntity entity -> src
```

As you can see they are referring 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:

```
-- Conversion from the domain type MeteorID to the associated type is simple.
instance AsKeyEntity MeteorEntity MeteorID where
  toKeyEntity = MeteorKey

-- We can construct the associated type KeyEntity from the domain type Meteor by getting its id.
instance AsKeyEntity MeteorEntity Meteor where
  toKeyEntity = MeteorKey . meteorId

-- We can construct the associated type ValueEntity from the domain type Meteor.
-- And perform the opposite operation as well.
instance AsValueEntity MeteorEntity Meteor where
  toValueEntity (Meteor _ size mass (Coords azmt alt) time) = KVDBMeteor {..}
  fromValueEntity (MeteorKey idx) KVDBMeteor {..} = Meteor . . . -- constructing a value here
```

Finally, we're able to talk about the typed loadEntity function. Remember, this function demands for the type level machinery for a particular DB and entity. It requests a ByteString value from the KV DB subsystem, and with the

help of all those conversion functions, it converts the ByteString value into the typed KV DB model first, and from the KV DB 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)                    -- DBEntity instance is used here
  pure $ case eRawVal of
    Left err  -> Left err
    Right val -> maybe
              err
              (Right . fromValueEntity key)        -- AsValueEntity instance is used here
              (mbE val)
  where
    mbE :: KVDBValue -> Maybe (ValueEntity entity)
    mbE = fromDBValue                               -- DBEntity instance is used here
    err = Left (DBError DecodingFailed "Failed to decode entity.")
```

So initially the goal for this type level machinery was the ability to specify KV DB 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 KV DB like AstroDB now combines all those entities into a single KV DB model.
- It's possible to put the same entities into another KV DB by defining a new KV DB 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.

But 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 should not 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.

3.3. Transactions

3.4. Pools

3.5. STM and in-place in-memory DB

4. Summary

This chapter gave us a broad idea on how to organize the access to SQL or KV DB subsystems. Not that many usage examples, this is true, but at least we considered:

- A simple interface to a particular SQL DB (Postgres);
- More generic interface that supports any SQL DB supported by beam;
- A raw ByteString based interface for two different KV DBs (RocksDB and Redis);
- A typed KV DB interface to KV DB on top of the raw interface.

And by the way, there is 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 DB model and Domain model. Let me remind you:

- Domain Model represents a set of types (and probably functions) directly related to the essence of the domain.
- DB Model reflects the idea that the internal representation of data in DBs 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 DB 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, complex ADTs with all the goodies like pattern matching, so we want to convert between the DB model and the Domain model.

We can convert things either 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 are free to choose how much type safety we need. For example, the beam library that we touched a little assumes that the DB model is well-typed, and we just can't avoid defining our table representations in the terms this library exposes. But for the KV DB model we can either use a simple ByteString based interface with no guarantees in conversions, or we can rely on the typed machinery for KV DB Model which gives a certain level of guarantees. Still, type level magic in Haskell that we used is not 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 still there are 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!

8
Business logic design

Links

- My email
- Book page
- Patreon campaign
- Paypal donations

This chapter covers

- How to layer and structure business logic
- How to decouple parts with services
- Many different approaches to Dependency Injection
- How to create web services with Servant

TODO

1. Business logic layering

No need to explain what business logic is. We've written it earlier in the form of scenarios, we built domain models and described domain behavior many times. Business Logic is something we really value, it's a code that does the main thing for a 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 the following figure describes (see Figure 8.1):

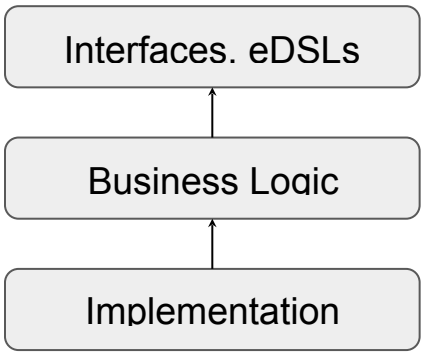


Figure 8.1: Three Main Layers of Application

With this architecture, it's impossible to bypass the underlying framework. All the methods you specify in the business logic will go through the 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 be looking more or less identically in all its parts. There is only one way, only one legal viewpoint, only one allowed approach. Limitations evoke ingenuity. Suddenly you discover many good ideas otherwise could have been left hidden. For example, one of the good consequences having a Free monadic framework is a nice testability of the code. The other one we'll be discussing here: a purified business logic that is in turn can be treated as a separate

subsystem that is a subject of Software Design. In particular, we'll see why and how to organize additional layers within the business logic layer. And what's important, these patterns can be applied in a wider context as well.

1.1. Explicit and implicit business logic

I have a simple question for you: whether all applications have business logic? Well, true that it highly depends on the definitions, still it seems inevitable to have some code that is responsible for serving the main value for a business. In other words, there should be a code implementing some functional requirements. I can imagine some scientific code in Agda, Idris or Coq that is written not for serving business but for verifying some scientific ideas, and therefore it's strange to call it "business logic". In other cases, this logic has to exist in some form.

In a badly designed application, we may have a hard time finding what lines do the actual job. All those flaws we call "non-clean, dirty, fragile, unmaintainable, unreadable code" are distracting 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 advices given in these books teaches you to always remember that you are a team player and should not intricate your code, your teammates, and finally yourself.

And probably the best way to be transparent lies through the transparency on what the code does. Explicit business logic well-understood and well-written can be only compared to a good literature telling a complete story.

TODO: what functional interfaces exist

2. A CLI tool for reporting astronomical objects

And 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, we wrote several queries. Supposedly, this logic should be a part of a server for tracking the 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're tasked to implement a client for this server, but the requirements are not available at the moment? Does it really block you? Would you start implementing at least a skeleton of the client which 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 finely approachable. And to avoid redesigning the application, we'll hide the interaction with the server behind a generic interface, the interface of reporting an asteroid.

2.1. API types and command-line interaction

Considering the API type for the meteor is:

```
-- API type for meteor.
-- Should be convertible from / to JSON for sending 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 = ... -- some implementation here
```

In the future, we'll add more features into the client and server. For example, why not reporting asteroids, pulsars, exoplanets and other interesting stuff?

```
-- API type for asteroid.
-- Should be convertible from / to JSON for sending via HTTP or TCP.
```

```
data AsteroidTemplate = AsteroidTemplate
  { name        :: Maybe Text
  , orbital     :: Orbital       -- Orbital characteristics
  , physical    :: Physical      -- Physical characteristics
  }
  deriving (Generic, ToJSON, FromJSON)

reportAsteroid :: AsteroidTemplate -> AppL (Either ByteString AsteroidId)
reportAsteroid a = ... -- some implementation here
```

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 standard de-facto in web. We'd like to use the same data types both for client and server side. To our pleasure, this is possible with the Servant library. It gives both server and client sides, and the servant library is finely compatible with aeson. The HTTP API for the astro server is presented in Listing 8.1:

Listing 8.1. HTTP API for server and client side.

```
type AstroAPI
  = (  "meteor"                          -- POST method "/meteor"
    :> ReqBody '[JSON] MeteorTemplate    -- JSON body in form of `MeteorTemplate` is requested
    :> Post '[JSON] MeteorId             -- Method returns MeteorId
    )
  :<|>
    (  "asteroid"                        -- POST method "/asteroid"
      :> ReqBody '[JSON] AsteroidTemplate      -- JSON body in form of `AsteroidTemplate` is
requested
    :> Post '[JSON] AsteroidId           -- Method returns AsteroidId
    )
```

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 is a little pitfall here. Let's write a piece of code:

```
consoleApp :: AppL ()
consoleApp = do
  line <- evalIO getLine            -- Reading user's input

  mbMeteor   = decode line          -- Trying to decode
  mbAsteroid = decode line

  case (mbMeteor, mbAsteroid) of    -- Checking the result and sending, possibly
     (Just meteor, _)   -> reportMeteor meteor
     (_, Just asteroid) -> reportAsteroid
     (Nothing, Nothing) -> logWarning "Command not recognized"
```

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 meet a problem on how to specify a type of an 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 the return type. Thus, to use it with different types, we have no other choice than to pass the type somehow. Like this:

```
-- pseudocode
let supportedObjects =
        [ tryParseCmd @MeteorTemplate   line    -- Types passed explicitly using TypeApplications
        , tryParseCmd @AsteroidTemplate line    -- Won't compile
        ]
```

The idea is to fold this structure and get either a single object parsed or an error value if none of types matched the contents of the string. Unfortunately, this won't compile because the supportedObjects list is not homogenous (the two items return different types). Happy you, there is a nice trick. We should parse an object and immediately utilize its result by 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. Checkout the code:

```
-- 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))         -- Specific reporter for the object
  -> (Either ByteString obj)                       -- Either a parsed object or a failure
  -> AppL (Either ByteString String)
reportWith _           (Left err)   = pure $ Left err   -- Return error if the string is not
recognized
reportWith reporter (Right obj) = do                -- Report the object with the reporter and
succeed
  reporter obj                                     -- Calling a specific reporter
  pure $ Right $ "Object reported: " <> show obj
```

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. That action which got a value successfully parsed will be evaluated, others actions will be ignored. Listing 8.2 puts all the pieces together:

Listing 8.2. Complete code of business logic.

```
consoleApp :: AppL ()
consoleApp = forever $ do
    line <- evalIO getLine

    let runners =
          [ reportWith reportMeteor   $ tryParseCmd line
          , reportWith reportAsteroid $ tryParseCmd line
          ]

    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 thing 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. This 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 much simple to update the runners list manually, and it's fine to have some labour work considering the new report functions should be written anyway. So we'll stop here with the objects extensibility problem.

But don't worry, there is a theme about extensibility we should discuss. Extensibility of the reporting methods. Currently, the functions reportMeteor and reportAsteroid do something. We don't know yet how exactly they send objects to the server. Let's figure this out?

2.2. HTTP and TCP client functionality

Now, before we move further, let's shortly learn how to implement the HTTP client functionality with the help of servant-client. We already have the API description, see Listing 8.1. The rest of work is straightforward: provide a bunch of helper functions and incorporate the client functionality into the business logic.

So, firstly 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

-- Helpers methods generation
(meteor :<|> asteroid) = client (Proxy :: AstroAPI)
```

Secondly, 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 see, we store the ClientM action here for the later running 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 single on the whole application. This is not 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 back. 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're adding the following function for reporting meteors:

```
reportMeteorHttp :: BaseUrl -> MeteorTemplate -> AppL (Either ByteString MeteorId)
reportMeteorHttp url m = do
  eMeteorId <- scenario
    $ callAPI url                       -- Interface method of the LangL language
    $ meteor m                          -- Client method for the HTTP API
  pure $ case eMeteorId of
    Left err       -> Left $ show err
    Right meteorId -> Right meteorId
```

The same works for asteroids and other stuff you want to query. No need to show the `reportAsteroidHttp` function, right?

So now two words on what we'll be doing next. Let's consider all these reporting functions to be implementation detail no matter if they operate 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 easily figure out the common interface here which is (we've seen it 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 any chance to carefully compare the approaches. Especially there are even more of them described in this chapter:

- Service Handle
- ReaderT
- Free Monad
- GADT
- Final Tagless (mtl)

And what's interesting here, it seems there are two meta kinds of functional interfaces in Haskell.

- Scenario-like interfaces. These interfaces are intended for writing scenarios, scripts, 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 only intended for alone usage, and the design of these methods doesn't imply a chaining. These interfaces are only needed to represent some API. Samples are: REST, ClientM from servant, our reporting methods.

This meta kind of interface is somewhat new to us. Finally, we can start investigating them. Let's go!

3. Functional interfaces and Dependency Injection

Dependency Injection in OOP serves a purpose to provide an implementation hidden behind an interface so that the client code could not bother about the implementation details and was completely decoupled with them. This technique becomes very important in big systems with a significant amount of IO operations. We usually say such systems are IO bound, and not CPU bound meaning the main activity is always related to the external services or other effects. Controlling these effects in the business logic becomes very important, and Dependency Injection helps with that making the logic less complex and more maintainable. In functional programming, we are not freed from IO. Even in Functional Programming it's nice to have a kind of Dependency Injection, because this principle is paradigm-agnostic. We've talked about this already in Chapter 3, and several approaches have been discussed 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.

3.1. 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 just store the reporting functions into a data structure for greater convenience. It's possible to just pass those functions (meteroReporter and asteroidReporter) as arguments and this will be fine once you have a small number of them. But for services with dozens 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 (AstroServiceHandle{meteorReporter, asteroidReporter}) = do

  line <- getUserInput                                    -- a nicer prompt, see below
```

```
  let runners =
        [ reportWith meteorReporter   $ tryParseCmd line   -- using methods of handler
        , reportWith asteroidReporter $ tryParseCmd line
        ]

  eResults <- sequence runners
  printResults eResults

getUserInput :: AppL ByteString
getUserInput = evalIO (putStr "> " >> getLine)
```

Providing a constructor for handle is a good idea.

```
data ReportChannel = TcpChannel | HttpChannel

makeServiceHandle :: ReportChannel -> AstroServiceHandle

-- handle with TCP reporters. For now, Tcp channel config is hardcoded:
makeServiceHandle TcpChannel  = AstroServiceHandle
  (reportMeteorTcp    $ TcpConfig "localhost" 33335)
  (reportAsteroidTcp  $ TcpConfig "localhost" 33335)

-- handle with HTTP reporters, with HTTP config hardcoded:
makeServiceHandle HttpChannel = AstroServiceHandle
  (reportMeteorHttp   $ BaseUrl Http "localhost" 8081 "")
  (reportAsteroidHttp $ BaseUrl Http "localhost" 8081 "")
```

Notice, the actual endpoints are hardcoded in these "implementation" functions. You don't like this? Okay, just pass the configs into the makeServiceHandle helper:

```
makeServiceHandle :: ReportChannel -> BaseUrl -> TcpConn -> AstroServiceHandle
```

The last thing is running the consoleApp with injecting a service handle into it.

```
main = do
  ch :: ReportChannel <- getReportChannel    -- getting ReportChannel somehow
  runtime <- createRuntime                   -- creating the runtime
  result <- runApp runtime                   -- running a normal AppL scenario
   $ consoleApp (makeServiceHandle ch)       -- 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 is a text file (usually XML) which contains different settings and configs for particular subsystems. Many IoC containers allow you to choose implementation and thus have an additional flexibility for the system. All the patterns we learn here may be a basis for such an DI framework, and I'm sure we'll see several of them in Haskell soon.

3.2. 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 a 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 loaded by occasional lifts. Check this out:

```
consoleApp :: AppRT ()
consoleApp = do

  -- Getting the control structure from the ReaderT environment
  AppEnv {meteorReporter, asteroidReporter} <- ask

  line <- getUserInput

  let runners =
        [ lift $ reportWith meteorReporter   $ tryParseCmd line
        , lift $ reportWith asteroidReporter $ tryParseCmd line
        ]

  eResults <- sequence runners
  lift $ printResults eResults
```

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 with providing a Reader environment variable:

```
main = do
  ch :: ReportChannel <- getReportChannel        -- getting ReportChannel somehow
  let appEnv = makeAppEnv ch                     -- creating a handle
  result <- runApp runtime                       -- running a normal AppL scenario
    $ runReaderT consoleApp appEnv               -- 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, and therefore 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 with the nextly going automatic derivings; and some others. Let's try this one and see.

TODO

So many additional steps! That's unfortunate and probably a too high cost for just removing the handle from consoleApp arguments. Maybe it's not worth it. Just use the Service Handle Pattern. Keep it simple.

3.3. Additional Free monad language

For the pleasure of 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, how the interpreters should look. You might have guessed 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, 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
```

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            -- getting ReportChannel somehow
  let astroServiceRunner = getAstroServiceRunner ch  -- getting a service implementation (runner)
  result <- runApp runtime                           -- evaluating a normal AppL scenario
      $ consoleApp astroServiceRunner                -- configured by the service runner
```

And what about the logic? Is it good? Logic is fine. It doesn't know anything about the internals of the AstroService runner. Just uses the additional Free monad language with it:

```
consoleApp :: (forall x. AstroServiceF x -> AppL x) -> AppL ()
consoleApp astroServiceRunner = do

  line <- getUserInput

  let runners =
        [ reportWith astroServiceRunner reportMeteor   $ tryParseCmd line
        , reportWith astroServiceRunner reportAsteroid $ tryParseCmd line
        ]

  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                -- calling the interpreter
  pure $ Right ()
```

In comparison to the ReaderT approach, the Free monad approach grows in another direction. There is no additional lifting here, but the Free monad machinery requires a bit more effort to implement. However lifting in the ReaderT approach is a 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 much boilerplaty as with 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 which doesn't give us much freedom. Anyway, just use the Service Handle. Keep it simple.

3.4. 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 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 a 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 a Fabric pattern:

```
-- Service creation function, analogue of the Fabric pattern
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 which utilizes this GADT-based service will look exactly the same as with Free monads:

```
main = do
  ch :: ReportChannel <- getReportChannel            -- getting ReportChannel somehow
  let astroServiceRunner = getAstroServiceRunner ch  -- getting a service implementation (runner)
  result <- runApp runtime                           -- evaluating a normal AppL scenario
      $ consoleApp astroServiceRunner                -- 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 runners =
        [ reportWith runner ReportMeteor   $ tryParseCmd line
        , reportWith runner ReportAsteroid $ tryParseCmd line
        ]

  eResults <- sequence runners
  printResults eResults
```

Notice that the runner argument has a 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 current value constructor of the GADT. 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 by a domain algebra. This means GADTs are really suitable for API-like interfaces and not that convenient for scenario-like interfaces.

3.5. Final Tagless / mtl

Final Tagless is a bit problematic from the start. Firstly, it's a dominating approach in Haskell (and in Scala recently), which means it outshines other approaches in Haskell's technical folklore. Secondly, there is no single predefined pattern of Final Tagless. In contrast, the practices haskellers call Final Tagless may involve more or less additional type level magic depending on how close a developer wants it to be to the original Final Tagless approach from the Oleg Kiselyov's paper. They say mtl style in Haskell is a special case of Final Tagless, and also they say Final Tagless is much more interesting than mtl. Well, I willingly believe it. But it seems people applying these techniques in practice without knowing much about the paper don't distinguish between mtl and Final Tagless. It's all Final Tagless for them. Therefore we'll consider the mtl style Final Tagless here, and it will probably be enough to get a general idea about the approach.

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 as we've seen previously:

```
class AstroService api where
  reportMeteor     :: MeteorTemplate   -> AppL (Either ByteString MeteorId)
  reportAsteroid   :: AsteroidTemplate -> AppL (Either ByteString AsteroidId)
```

Except a tiny difference, namely the api phantom type. This type will help to select the needed instance, currently one of the two. This time we may only pass either HttpAstroService or TcpAstroService, a special type selectors provided with 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 to be a selector for this type class
  => AppL ()
consoleApp = do

  line <- getUserInput
```

```
  let runners =
        [ reportWith (reportMeteor @api)   $ tryParseCmd line  -- type application of api
        , reportWith (reportAsteroid @api) $ tryParseCmd line  -- to select the instance
        ]

  eResults <- sequence runners
  printResults eResults

  consoleApp @api    -- type application of api to select the instance
```

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            -- getting ReportChannel somehow
  result <- runApp runtime $                         -- evaluating a normal AppL scenario
    case ch of
      TcpChannel  -> consoleApp @(TcpAstroService)   -- where the scenario is configured
      HttpChannel -> consoleApp @(HttpAstroService)  -- by the service implementation
```

Despite some implicitness here (no physical values which 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 took with the code above make it not really an mtl styled code. The two key points made the approach quite usable: phantom type-selector (api) and a fixed monad type (AppL). Great news, the AstroService susbystem works in the same monad AppL and is displaced in the same layer of business logic, therefore it integrates with the code very well.

But in the common practice of mtl usage, it's 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 types-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)    -- List of constraints, an attribute of the FT/mtl
style
  => AsteroidTemplate
  -> m ()
sendAsteroid asteroid = do
  eResult <- reportAsteroid asteroid
  case eResult of
    Left err -> log Error $ "Failed to send asteroid: " ++ show err
    Right _  -> log Info "Asteroid sent."
```

What are the consequences do 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 be knowing 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 like the following:

```
newtype AppM a =
    AppM { unAppM :: ReaderT AppRuntime IO a }     -- Notice, we keep our AppRuntime here now.
    deriving (Functor, Applicative, Monad, MonadIO)
```

The AppM monad will be our working 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 are interested in learning more stuff around mtl / Final Tagless, consider the materials from this list:

Software Design in Haskell - Final Tagless

But remember, the rabbit hole is very deep.

3.6. And what about effect systems?

Effect systems - those are which force you to specify a list of effects on the type level - is a bad idea. Period.

TODO

4. Designing web services

This book tries to be diversified in the useful information around Software Engineering and Software Design. Talking about abstract or toy examples can be not that satisfying (still, unavoidable), so can we try to go more realistic while introducing more design approaches? Web services here look like a good theme: firstly, the tools described earlier work nicely in this context; and secondly, web services is a very common task nowadays. Besides that, we've already designed a client for our imaginary service, why not fake it till we make it?

4.1. REST API and API types

Let's play Haskell and Servant! It's a lot like life, because Servant is a very wide-spread solution in Haskell for building RESTful services. Can't say simple, - it's not, because its special type-level eDSL requires a bit of learning. But simple enough due to the aim of this eDSL to be used by us who are not proficient in the advanced type level magic. So once you have a sample of Servant-based service you may rework and extend it for the most of scenarios. Hopefully you don't have any non-common 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 8.1 with a definition of REST API for the Astro service. It only exposes two methods: "/meteor" and "/asteroid". These methods should be used to report the corresponding objects, and for reporting 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 make the evidence that tracking of 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 also should remember about the limitations which are described in different RFCs for HTTP. It might be so that passing the 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. Like firstly we send this template:

```
data AstroObjectTemplate = AstroObjectTemplate
  { name        :: Maybe Text
  , objectCass  :: Text
  , code        :: Text
  }
```

We expect the server to return an identificator for this record, and we can use this identificator to provide more info about the object. Let's design a Servant API type for this situation:

Listing 8.3: Extended Server API

```
type AstroObjectId = Text

type AstroAPI =
    (  "object_template"                              -- route POST "/object_template"
    :> ReqBody '[JSON] AstroObjectTemplate
    :> Post '[JSON] AstroObjectId
    )
  :<|>
    (  "object"                                       -- route GET "/object"
    :> Capture "object_id" AstroObjectId
    :> Get '[JSON] (Maybe AstroObject)
    )
  :<|>
    ( "orbital"                                       -- route POST "/orbital"
    :> Capture "object_id" AstroObjectId
    :> ReqBody '[JSON] Orbital
    :> Post '[JSON] AstroObjectId
    )
  :<|>
    ( "physical"                                      -- route POST "/physical"
    :> Capture "object_id" AstroObjectId
    :> ReqBody '[JSON] Physical
    :> Post '[JSON] AstroObjectId
    )
```

In there, 4 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 the client to specify the object_id field. Servant has a set of different modifiers for URL, for body content, for headers, for queries etc. 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 nicely-looking API reference documentation, Swagger definitions and even client functions as we did previously. So we must state Servant is

one of the most well-designed and convenient type level eDSLs existing in Haskell when it's clear what this type magic gives to 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'll learn it, we should prepare the handlers, which are the functions to process 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 the errors suitable for Servant (ServerError). Also you can see 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 the 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. 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     -- impure methods require lifting due to ExceptT
wrapper
  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. In 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 8.4: Server API with nested routes

```
type AstroAPI =
    (  "object_template"                                -- route POST "/object_template"
    :> ReqBody '[JSON] AstroObjectTemplate
    :> Post '[JSON] AstroObjectId
    )
  :<|>
    "object" :>
    (
       ( Capture "object_id" AstroObjectId              -- route GET "/object"
       :> Get '[JSON] (Maybe AstroObject)
       )
     :<|>
       ( "orbital"                                      -- route POST "/object/orbital"
       :> Capture "object_id" AstroObjectId
       :> ReqBody '[JSON] Orbital
       :> Post '[JSON] AstroObjectId
       )
     :<|>
       ( "physical"                                     -- route POST "/object/physical
       :> Capture "object_id" AstroObjectId
       :> ReqBody '[JSON] Physical
```

```
      :> Post '[JSON] AstroObjectId
      )
  )
```

This will affect the way the actual method handlers are sequenced to form a complete server definition. Checkout the 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 8.2):

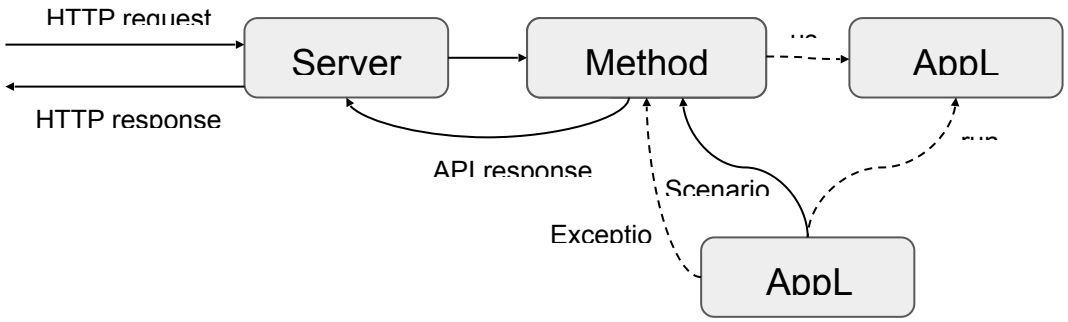


Figure 8.2: Callstack of server processing pipeline

4.2. Using framework for business logic

Putting the business logic into the Handler monad is a straightforward approach, it has the flaws we discussed many times: too easy to break, too hard to test, 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. Checkout this diagram of how this should be (Figure 8.3):

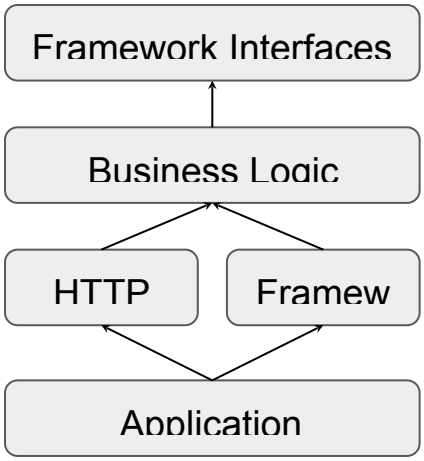


Figure 8.3: Layered web service application

We can't just call an AppL scenario, there should be the Runtime structure already created. There is no point in re-creating 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 we have 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 could 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    -- getting a ready connection
```

```
  scenario
     $ runDB dbConn
     $ findRow
     $ ...       -- some beam query here
```

Notice the connection to the DB and the framework method getSqlDBConnection. Let's consider for a while the DB 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 lately. 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 when the connection dies. The internal pool will recreate it automatically if configured so, but if this whole behaviour with shared SQL connections is not appropriate for your cases, consider to initialize it every time you do the query.

This is how the handler changed now:

```
-- The new handler. It now can access the Runtime structure from the ReaderT environment:
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                                          -- getting the runtime
  eRes <- lift $ lift $ try $ runAppL runtime flow    -- safely running the scenario
  case eRes of                                                  -- converting the exception into the
response
    Left (err :: SomeException) -> do
      liftIO $ putStrLn $ "Exception handled: " <> show err
      throwError err500
    Right res -> pure res
```

And it turns out that handlers are the best place to catch the exceptions from the business logic and convert them into the appropriate HTTP response. It's important that the framework is designed in such a way when the internal code of its parts (subsystems and services; namely - the implementation) don't emit any unexpected exceptions. For example, in Haskell it makes sense to prevent the async exceptions from even being born in the internals. Neither the framework code nor the underlying libraries should emit the async exceptions because it's really hard to predict them and catch correctly. We could even say the need in async exceptions point to the design flaws, and moreover, async exceptions violate the Liskov substitution principle. Regular synchronous exceptions also do, but at least we can have a single point of catching them without losing the control over the code.

So the latest piece of puzzle goes further: having these handlers, how to 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.

4.3. 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
```

And this monad stack is finally our full environmental type for the whole server. Yes, it's a monad, and it is aware of AstroAPI. It also has the ReaderT environment, the ExceptT environment and IO on the bottom. The following diagram (Figure 8.4) reflects the monadic stack:

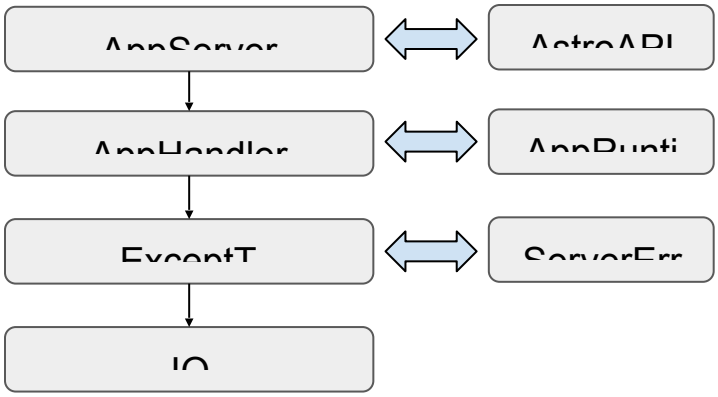


Figure 8.4: Monadic stack of the server.

Now we have handlers, we have 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 (shape of the API from Listing 8.4 is used here):

```
astroServer' :: AppServer
astroServer' =
  submitObjectTemplate
  :<|>
    (    getObject
    :<|> submitObjectOrbital
    :<|> submitObjectPhysical
    )
```

The servant framework mandates we should convert our AppServer to the native Server type which is:

```
-- from the servant library
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.

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'              -- Transforming the stacks
  where                                                              --   for the astroAPI
    f :: ReaderT AppRuntime (ExceptT ServerError IO) a -> Handler a
    f r = do
       eResult <- liftIO $ runExceptT $ runReaderT r appRt             -- Unwrapping the ReaderT
and
      case eResult of                                                --   ExceptT environments
        Left err  -> throwError err                                  -- One more place
        Right res -> pure res                                        --   to handle errors
```

So many preparations! But this is not 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 the 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, however some questions remain unanswered. Namely, how would we prepare the AppRuntime and the permanent connections to the DBs, to the external services etc. Where is the right place for it? Right here, before we call the run function. Let's investigate the following functions step-by-step:

```
runAstroServer :: IO ()
runAstroServer =

  -- withAppRuntime safely creates the AppRuntime structure to be used in the framework.
  -- It also handles the exceptions and gracely destroys the environment.

  withAppRuntime loggerCfg $ \appRt -> do

    -- Preparing the permanent connections, making migrations, initializing DBs
    runAppL appRt prepareDB

    -- Possibly, initializing state for the application (as we did in Chapter 6)
    appSt <- runAppL appRt prepareAppState

    -- starting the servant server
    run 8080 $ astroBackendApp $ Env appRt appSt
```

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 the permanent connections (which will be kept inside the AppRuntime as usual), make migrations, evaluating premature queries to the DBs or external services. It's your choice whether to write this separate business logic with the framework or make it IO only. In the case above 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 also you can use it more wisely for the 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.

4.4. Validation

World wide web is a truly communist foundation in which everyone can obtain the resources he needs, and all the communities all around the globe are unified to help each other, to build a bright future together. What can go wrong in this world of a true brightness of a 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 when some intruder is trying to scam the service by sending some invalid requests in order to find security bugs. A golden rule of all the services declares to protect from these attacks by validating the incoming requests. This not only provides protection from the occasional breakage of the service but also a sanity check whether all the cases are considered, whether the API is consistent and robust. So all the incoming data should be checked for validity, and the logic should operate by the correct values only. Otherwise, how the logic should behave? Garbage in values leads to garbage in calculations.

From the design point of view it means we can distinguish the two models: the API model which 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 following process will be the best for API services from the architectural perspective (see Figure 8.5):

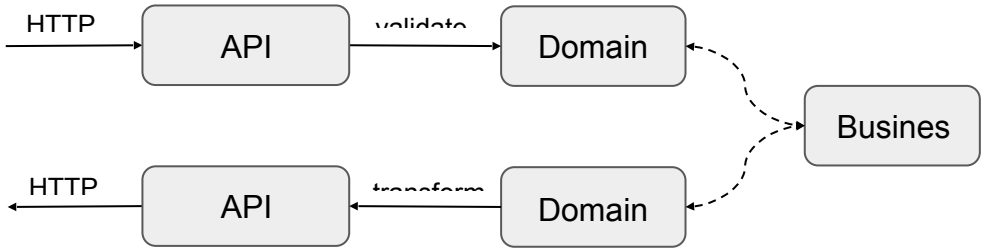


Figure 8.5: API and Domain types

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 the 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 the 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-validation is quite simple. Just do a straightforward check of values and react to 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"). For example we'd like the two fields to be non-empty (objectClass and code), and name to be non-empty if and only if it's not Nothing. With a little help of the MultiwayIf and RecordWildCards extensions, we can write the code like this:

```
-- Handler
submitObjectTemplate :: AstroObjectTemplate -> AppHandler AstroObjectId
submitObjectTemplate template@(AstroObjectTemplate {..}) = do

   if | name == Just ""    -> throwError $ err400 {errBody = "Name should not be empty if
specified."}
     | objectClass == "" -> throwError $ err400 {errBody = "Object class should not be empty."}
     | code == ""        -> throwError $ err400 {errBody = "Object code should not be empty."}
     | 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, kinda. The Hydra framework, there are methods for logging and throwing exceptions.

```
-- Qualified imports to distinguish between framework calls and other calls,
-- between API types and domain types

import qualified Hydra.Language           as L
import qualified Astro.Domain.AstroObject as D
import qualified Astro.API.AstroObject    as API

-- Handler
submitObjectTemplate :: API.AstroObjectTemplate -> AppHandler D.AstroObjectId
submitObjectTemplate template = runApp $ submitObjectTemplate' template

-- Intermediate method
submitObjectTemplate' :: API.AstroObjectTemplate -> L.AppL D.AstroObjectId
submitObjectTemplate' template@(API.AstroObjectTemplate {..}) = do

  let failWith err = do
        L.logError $ show err                -- Using framework to log errors
        L.scenario $ L.throwException err

   if | name == Just ""     -> failWith $ err400 {errBody = "Name should not be empty if
specified."}
     | objectClass == "" -> failWith $ err400 {errBody = "Object class should not be empty."}
     | code == ""        -> failWith $ err400 {errBody = "Object code should not be empty."}
     | otherwise         -> pure ()

  -- Structure is valid, do something with it.
  createObjectTemplate template

-- Save data into DB, get a domain type of it and return an ID:
createObjectTemplate :: API.AstroObjectTemplate -> L.AppL D.AstroObjectId
createObjectTemplate template = do
  astroObjectDB :: D.AstroObject <- saveAstroObject template
  pure $ astroObjectDB ^. astroObjectId
```

There is one obvious issue with such an approach. The very first erroneous field will breach the evaluation which leaves the rest of 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 convenience of clients. This means we should adopt some other approach to validation, to invent a mechanism that allows us to validate all the fields in a 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 suites for Functional Programming very well, this is why many different combinatorial approaches have emerged in the Haskell community. Most interesting of them focus on Applicative-style validation. You may find several libraries to do this: validation, validation-selective and others. 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 not only get a message "some internal field is invalid", but also get bread crumbs to it.

For example, consider the two following data structures organized hierarchically:

```
data Inner = Inner
  { _intField   :: Int       -- Should be > 0 and less 100
  }

data Outer = Outer
  { _stringField :: String   -- Should not be empty
  , _innerField   :: Inner   -- Nesting one structure into another
  }
```

If the Inner value is malformed (_intField is 0, negative or greater than 100), and the _stringField is malformed (empty), 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"}
]
```

Where 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. Checkout this validator for the case we described above:

```
innerValidator :: Validator Inner
innerValidator = validator $ \inner -> Inner
  <$> (inner ^. intField'                        -- Special pointed getter
         &  condition (> 0)   "Inner intField: should be > 0"
         &. condition (< 100) "Inner intField: should be < 100")

outerValidator :: Validator Outer
outerValidator = validator $ \outer -> Outer
  <$> (outer ^. stringField'                     -- Special pointed getter
               &  condition (not . null) "Outer stringField: should not be empty")
  <*> (nested outer innerField' innerValidator)

-- Running the validators

main = do
  let value = Outer "" (Inner 0)
  case applyValidator outerValidator value of
     SuccessResult _      -> putStrLn "Valid."
     ErrorResult _ errors -> print errors
```

Notice several facts here.

- We can stack several validators for a single field. We combined the two to check whether the _intField value lies between (0, 100].
- We use Applicative style to compose different validators. With help of the fmap (<$>) and ap (<*>) operators, we "visit" all the fields and can't skip anything.
- There is 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 they point to but also a path to this value. For instance, the intField' pointed getter looks 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 autogenerate them.

We won't dive too deep into the theme of structural data validation. You may find many different articles on this topic. We'll also skip the question about an effectful validation which is when you have to make some side effect before the field is considered valid. A common use case here when you need to query a DB 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 further.

4.5. DB Layer and project layout

4.6. Configuration management

5. Summary

It might be that many new design patterns are still hidden in the deep of Functional Programming. With time, these patterns will be found and developed. Our task is to prepare a ground for such knowledge because being separated, facts are not that helpful than 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 learnt several different ways to do Dependency Injection and services:

- Free monads
- Final Tagless
- GADTs (for API-like interfaces)
- Service Handle pattern
- ReaderT pattern
- and others.

These patterns should be used when you want to make the code less coupled, more controlled and high level. We have seen examples of business logic being based on different approaches and now it's possible to compare them by several criteria:

- Simplicity
- Boilerplate
- Convenience
- Involving of advanced language features

The following opinionated table tries to summarise 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 is only aimed to be a starting point for further reasoning and cannot be the last truth. There are different cases and different situations, and our responsibility as software engineers to make the appropriate decisions and not lose common sense.