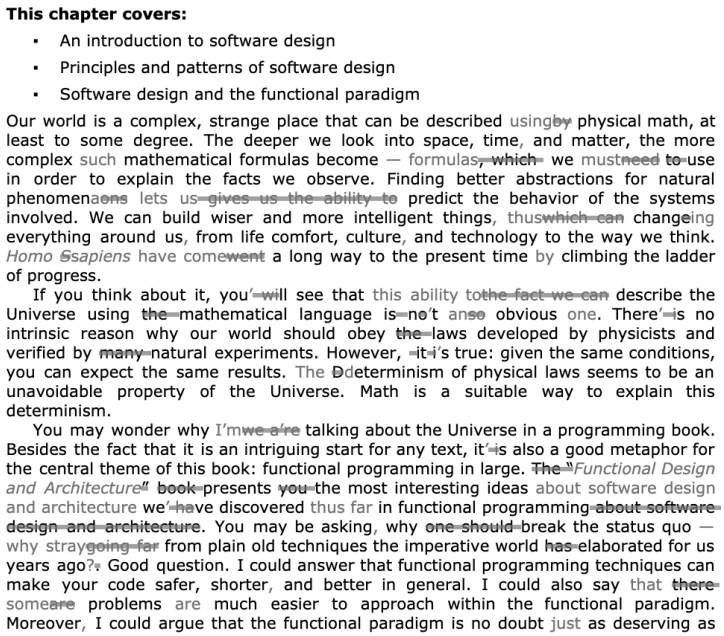
Preamble

Hello dear reader!

This is a draft of my book available for a free reading. This version of the book is incomplete, unedited, not properly styled. It won't be updated. Consider buying the book, and you'll get a complete text revised by a professional editor. You'll also get some additional materials such as educational videos.

https://leanpub.com/functional-design-and-architecture

Here is the example of how deeply the text was edited:



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 usingby physical math, at least to some degree. The deeper we look into space, time, and matter, the more complex such mathematical formulas become — formulas, which we mustneed to use in order to explain the facts we observe. Finding better abstractions for natural phenomenaons lets us gives us the ability to predict the behavior of the systems involved. We can build wiser and more intelligent things, thuswhich can changeing everything around us, from life comfort, culture, and technology to the way we think. *Homo Ssapiens* have comewent a long way to the present time by climbing the ladder of progress.

If you think about it, you'will see that this ability tothe fact we can describe the Universe using the mathematical language isno't anso obvious one. 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. The Ddeterminism of physical laws seems to be an unavoidable property of the Universe. Math is a suitable way to explain this determinism.

You may wonder why I'mwe a're 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 about software design and architecture we'have discovered thus far in functional programming about software design and architecture. You may be asking, why one should break the status quo — why straygoing 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 that there someare problems are much easier to approach within the functional paradigm. Moreover, I could argue that the functional paradigm is no doubt just as deserving as

5 Application state

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 from 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 to 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 is 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 a 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 such code can be unsafe, because we can make mistakes. This is a bit of a philosophical question, but it really helps to not to exclude Haskell from pure functional languages.

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

5.1 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 was 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 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 the 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, several different kinds of state may be used to solve 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 beginning 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 provide 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 a functional imitation of mutability. We also can say, pure state doesn't destroy 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 an impure state is much more dangerous than a pure one, there are techniques that help to secure impure stateful calculations. Functional code that works with impure state can be still deterministic by the behavior.

The simulation model represents a state that exists 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 contrast, the 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 exists 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 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 on 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 should we 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 databases. All is fine except we created another separate component among separate components. 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 the deadline happens, he realizes that the whole system can't work properly. He has to start from scratch. The end. To

not fall into this problem, we should prove that our subsystems can work together even if the whole application is still not ready. Integration tests can help here a lot. They are used to verify the 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 the product. You may find many articles about this technique by keywords "minimal viable product" or MVP. The technique aims 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 such a program, namely a simulator of a spaceship. The simulator has to be stateful, no exceptions. In fact, devices in spaceships are micro-controllers with their own processors, memory, network interfaces, clock and operating system. They behave independently. Events 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 computers 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 while working on the simulator in this and further chapters.

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

5.1.3 Hardware network definition language

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

- Spaceship is a network of 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.
- Controllers support a particular communication protocol.
- Logic control unit evaluates general control over the ship following the instructions from users 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 introduce any mechanisms to describe how the devices are connected together. Let's call this mechanism the 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 the 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 spider web-like one. We'll adopt the following simple rules for our control network topology:

- Logic control units 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 as a controller is a component, we can refer to 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 interface 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 the network.
data NetworkComponent a
    = DeviceDef PhysicalAddress (Hdl ()) (DeviceInterface -> a)
    | TerminalUnitDef PhysicalAddress (TerminalUnitInterface -> a)
    | LogicControlDef PhysicalAddress (LogicControlInterface -> a)
    | LinkedDeviceDef DeviceInterface TerminalUnitInterface a
    | LinkDef LogicControlInterface [TerminalUnitInterface] a
  deriving (Functor)

-- | Free monad Hardware Network 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 the 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]
```

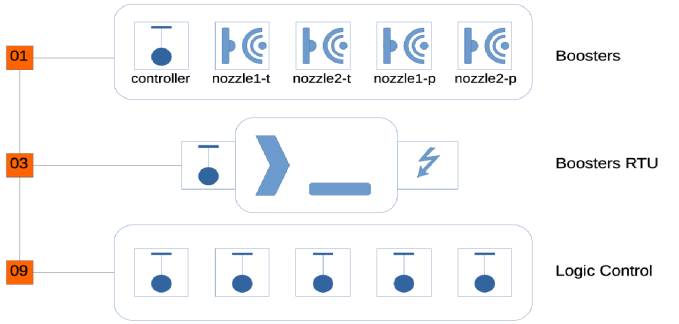


Figure 5.1: Sample network scheme

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 the measured parameter. Because none of the physical parameters can be really measured, the 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 an 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 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 stateful network routers, and so on. Every simulated device will have a state. Let's call the notion of threaded state a *node*.
- *Mutable*. State should be mutable because real devices rewrite their memory every time 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 held 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 languages: we interpret it and create a simulation model during the interpretation process. We visit every network component (for example, `TerminalUnitDef`) and create an appropriate simulation object for it. If we hit a `DeviceDef` network component, we then visit its `Hdl` field and interpret the internal free HDL script as desired. Namely, we should create simulation objects for every sensor and every controller we meet in the device definition. Let's call the whole interpretation process a compilation of HNDL to a simulation model.

Once we receive the model, we should be able to run it, stop it, configure sensors and other simulation objects and do other things that we stated in the requirements. The model will be probably somewhat complex because we said it should be concurrent, stateful and impure. The client code definitely wants to know as little as possible about the guts of the model, so it seems a wise idea to establish some high-level interface to it. In our case, the simulator is just a service that works with the simulation model of the spaceship's network. To communicate with the simulator, we'll adopt the MVar request-response pattern. We 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.

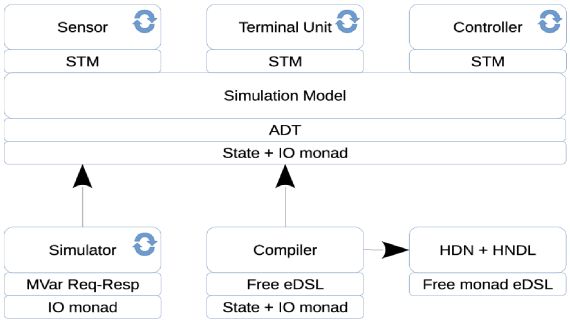


Figure 5.2: Architecture of the simulator

Now we are ready to do real things.

5.2 Pure state

By default definition, pure state is a state that can be created once and every update of this value leads to copying of it. This is the so-called copy-on-write strategy. The previous value should not be deleted from memory, although it's allowed that no references are longer pointed to it. Therefore, pure state can't be mutable in the 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 escape from these 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 the 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 long as the function does it the same way every time it's called, the regular output will be also the same. The only requirement

here is that no other code should have any kind of access (reading, writing) to the array. In other words, the mutable array will exist only locally, for this concrete function, and when calculations are done, the array should be destroyed. It's easy to see that the first and the second points of the list above are satisfied.

Indeed, this notion of local mutable state exists 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 a 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 to identify a particular object the client code wants to deal with. As soon as the model is built from the HNDL description, it's very natural to refer to every object by the same identifications that are used in HNDL and HDL scripts. This is why the `PhysicalAddress` type corresponds to every network component and the `ComponentIndex` type identifies a device component (see the definition of the HDL language). A pair of `PhysicalAddress` and `ComponentIndex` values is enough to identify a sensor or 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 depths 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 updates 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 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 updated with the result returned. It's important to note that working with the bounded 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 of what the monad is. You may find monadic combinators in Haskell's `Control.Monad` module and in 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 see it before because Haskell's do notation hides it, but it is no doubt there. The equivalent do-block for the `modifyState` function will be 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 exists 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 an 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 to develop this in the section "Impure state with State and IO monads". But what about the compiler of HNDL to `SimulationModel`? Let this (quite familiar, indeed) task will be another introduction to lenses in the context of the State monad.

First, you declare an ADT for holding state. In Haskell, lenses can be created with the `TemplateHaskell` extension for fields that are prefixed by underscore:

```
data CompilerState = CompilerState
    { _currentPhysicalAddress :: PhysicalAddress
    , _composingSensors :: SensorsModel
    , _composingControllers :: ControllersModel
    , _composingTerminalUnits :: TerminalUnitsModel
    }

makeLenses ''CompilerState
```

```
type SimCompilerState a = State CompilerState a
```

These lenses will be created:

```
currentPhysicalAddress :: Lens' CompilerState PhysicalAddress
composingSensors :: Lens' CompilerState SensorsModel
composingControllers :: Lens' CompilerState ControllersModel
composingTerminalUnits :: Lens' CompilerState TerminalUnitsModel
```

The `Lens'` type came from the Control.Lens module. It denotes a simplified type of lens. The type of some lens `Lens' a b` should be read as "lens to access a field of type `b` inside `a` type". Thus, the `composingSensors` lens provides access to the field of the type `SensorsModel` inside the `CompilerState` ADT. The compiler itself is an instance of the `Interpreter` type class that exists for the HNDL free language. There is also the `interpretHndl` function. This stuff wasn'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 is 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 the new map back. These three operations can be easily replaced by one lens combinator (`%=`). You'll be also needing 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 fails to be immutable, but sometimes you'll be observing inefficiency. State is real. Impurity is real. Mutability has advantages. Is pure functional programming flawed in this? The answer is coming.

5.3 Impure state

We talked about pure immutable states while designing a model to simulate a hardware network. A good start, isn't it? The truth is that in real life, it's more often that you need mutable imperative data structures rather than immutable functional ones. The problem becomes much sharper if you want to store your data in collections. This kind of state requires careful thinking. Sometimes you can be pleasured 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 vectors 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 as 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[1] over some type `a`.
- `MVar` is a concept of thread-safe mutable variables. Unlike the `IORef`, this reference type gives guarantees of atomic reading and writing. `MVar` can be used for communication between threads or managing simple use cases with data structures. Still, it's susceptible to the same problems: race conditions, deadlocks, non-determinism.
- `TVar`, `TMVar`, `TQueue`, `TArray` and other primitives of Software Transactional Memory (STM) can be thought of as further development of the `MVar` concept. STM primitives are thread-safe and imperatively mutable, but unlike `MVar`, STM introduces transactions. Every mutation is performed in transaction. In case of competing of two threads for the access to the variable, one of two transactions will be performed while the other can safely delay (retried) or even rollback. 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 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
    }
```

[1] Reference type in Wikipedia: https://en.wikipedia.org/wiki/Reference_type

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
    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 it 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 the model is updating? How to set up another value generator to a specific sensor? How to query the controllers? This design of a pure simulation model is wrong. We'll try another approach.

The idea is that you can observe impure mutation of an `IORef` value from different threads, as it happens in the 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 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 their 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 thread synchronization in the code, the threads are reading and writing the shared state (`refVal`) at their 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 replacing 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 an exception immediately when it's forked? The first thread will never be stopped, so you'll get a zombie that just heats the CPU. Something can probably break the `threadDelay` and 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 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 the 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 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 programs to unexpected crashes, corrupted data and uncontrolled side effects.

The `updateValue` function reads and rewrites the whole `SensorNode` variable in the `IORef` container which seems 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 the 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 its 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 about what to do with the simulation model, should do that and then it should send the results back. From a 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 three 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:

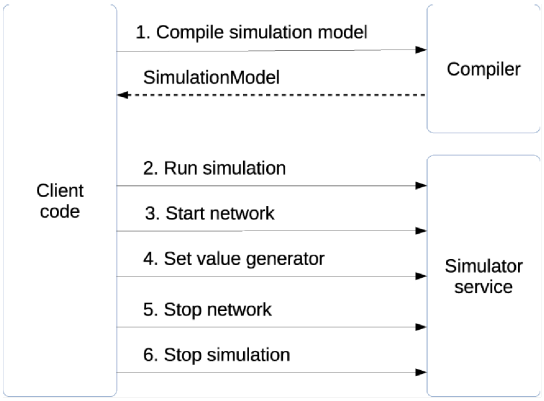


Figure 5.7: Simple interaction scenario

Let's try to write a test that shapes the minimal interface to the simulator that allows us to support the scenario. It's fine that machinery doesn't exist yet; following the Test Driven Development (TDD) philosophy, we'll implement it later. Fortunately, something we already

have: namely, the compilation subsystem. This piece fits 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 a simple interface and no matter what miles the simulator has to walk to make this real. That's why our interface is good. However we have to elaborate the internals that 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 a pair of some handle and pipe. The handle is an instance of the special type `SimulatorHandle` that contains useful information about the service started:

```
data SimulatorHandle = SimulatorHandle
    { shSimulationModel :: SimulationModel
    , shSensorsHandles :: SensorsHandles
    , shStartTime :: UTCTime
    , shThreadId :: ThreadId
    }

startSimulator :: SimulationModel -> IO (SimulatorHandle, SimulatorPipe)
startSimulator = undefined
```

Clear enough. So this function somehow starts a sensor model (that we know how to do), gets current time (the `UTCTime` is the standard type in Haskell), creates the pipe and forks a thread for the simulator. This is the code:

```
forkSimulatorWorker :: SimulationModel -> SimulatorPipe -> IO ThreadId
forkSimulatorWorker simModel pipe = undefined

startSimulator :: SimulationModel -> IO (SimulatorHandle, SimulatorPipe)
startSimulator simModel@(SimulationModel sensorsModel _ _) = do
    pipe <- createPipe :: IO SimulatorPipe

    startTime <- getCurrentTime      #A
    sensorsHandles <- startSensorsSimulation sensorsModel #B
    threadId <- forkSimulatorWorker simModel pipe  #C

    let handle = SimulatorHandle simModel sensorsHandles startTime
threadId
    return (handle, pipe)
```

#A System call from Data.Time.Clock
#B Known part
#C Forking a worker - not implemented yet

Notice that 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 forking of a thread. Let's give 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. By seeing their 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 their implementation.

#2: The `process` function translates the ADT language to the real monadic action. It also may do something impure, for example, writing a log. The `liftIO` function allows 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 the 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 a 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 a much more useful thing for design of code than the community of functional programmers was thinking before.

Indeed, designing with monads requires you to atomize pieces of code to smaller and smaller functions that have only a single responsibility. If that weren't so, then the composition 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 the SRP principle in FP. Again, as a consequence, you immediately gain a huge reusability and correctness of code. Even in a multithreaded environment.

So what concepts did you get from this chapter?

- The State monad is revisited. Although you can do stateful functional applications without any monads, the State monad 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 with impure mutable references that are a full analogue of imperative variables. But of course you should be aware of the problems the imperative nature of IORef drags. At least you don't want to use it with threads.
- The STM monad comes to the scene when you do need an impure state in a multithreaded 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 it 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!