Teaching Software Architecture Using Haskell

Alejandro Serrano
Department of Information and Computing Sciences
Universiteit Utrecht
A.SerranoMena@uu.nl

Software Architecture is an important part of the Computer Science curriculum. In this paper we propose using Haskell as a tool for exercising architectural patterns, and argue that this enhances understanding of the material by the students and clarity of explanation. We also provide many examples of common architectural patterns and their Haskell counterparts to back up our suggestions.

1 Introduction

Software architecture has made its own place in the Computer Science curriculum. Once students have learnt the basic of computer programming and low-level operations and protocols, they learnt the basics of creating larger software artifacts using Software Engineering techniques. However, Software Engineering usually speaks about middle-level constructs: classes, interfaces... Software Architecture is the next natural step, taking a birds eye view over a whole software system, and investigating its major components and relations between them. Since architecture has major implications on performance, maintainability or security, this branch of Software Engineering is becoming much more important.

In this paper we would like to look at some parts of a typical Software Architecture course and how Haskell could be used as a tool for teaching some important concepts. In the current situation, it's very difficult for a student to fully exercise its recently gained knowledge on Software Architecture in a practical basis: it's completely impossible to do a project in each possible architecture discussed throughout a course. Thus, most of the assignments in such a course usually entail writing diagrams and documentation, tasks would do not engage students as much as coding.

This trend is deepened by the fact that Software Architecture is taught usually in a descriptive way: how somebody would architect some system, which are the names of the components, or which relations should be taken into account. The bottomline is that there's no formal treatment of the differences between architectural patterns and styles, which make it difficult to reason about them; and no tools which could enforce a good style of architecture while learning.

The solution we propose, already hinted by the title, is to use Haskell as a tool for Software Architecture education. We shall first look at the different benefits Haskell brings to the classroom. Then, we shall see how many common architectural patterns could be encoded whithin common Haskell libraries, as examples of our teaching methodology.

2 Benefits of Using Haskell

Haskell is a pure lazy functional language, with support for many different ways of abstraction: parametric polymorphism, ad-hoc overloading via type classes or data type-generic programming. Its benefits for conciseness, reusability and maintainability of code had been argued many times; and its use in the classroom advocated. But up to this point, only low- and middle-level software construction has been considered, let's see the benefits for high-level Software Architecture teaching in Haskell.

Submitted to:
TFPIE 2014

2.1 Exercising Software Architecture

Nowadays, Haskell developers have an enormous amount of libraries ready to be used in Hackage, its package repository. Some of them support architectural pattern in a quite straightforward way.

Take for example the `stm` (Software Transactional Memory) library [9]. In a nutshell, STM enables the use of special variables whose access is mediated via transactions. That is, concurrent accesses to the variable are scheduled to not interfere, and any possible constraint over the values is checked before the final result is finally written. This description would already make it perfect to explain the very few primitive that make up a transactional system, but it's also possible to use it to explain the *shared database* architectural pattern! The same will be argued with other pairs of library and architectural pattern later in the text.

This brings to the table one of the abilities we were looking for: students can fiddle with an actual system, which is fully executable but at the same time can be made small enough to be understandable. As we see it, now instead of asking students to write a diagram of the components of a software system using a transactional shared database, they can build it.

The reason why this is important is to give a better feasibility view on the architectures that are designed as part of an exercise. Our experience with teaching Software Engineering is that in many cases students propose designs which look nice in theory, but will never be implementable. The reason is usually that they left some intermediate module or class which is needed to get the data in the right way. We think that this problems also translate to Software Architecture designs. Alas, the implications of a wrong design are every worse: a whole architecture can loose many of their performance or scalability skills if new unexpected components need to be added to make it fully functional.

In many cases, the Haskell libraries we refer too encode invariants in their types or do not allow to use types outside its intented meaning by using different artifacts. For example, the way in which monad transformers are encoded forbid any use of a function in a monad stack from lower layers. This helps students to keep honest about themselves, not introducing any trick inside some architecture. At the same time, teachers can be sure that components are not used in unexpected ways. Otherwise, code won't compile.

We shall remark that we are not proposing to change Software Architecture courses into a set of programming exercises which build into some specific libraries. Careful analysis, considerations of the pros and cons of each architectural style and discussion of the best scenarios for each kind of requirement are still needed and should be part of such a course. But for the more concrete part, that which describes which architectural patterns are commonly use, we feel that concretizing them into implementation will aid into understanding the material.

2.2 Reasoning About Patterns

A second benefit for Software Architecture patterns teaching derives from the usual way in which Haskell libraries are structured. Very often, Haskell library writers structure them from a small set of primitives, which are later combined to get higher-level functionality. A perfect example of this mentality is the separation of the whole bunch of state, input/output, references, etcetera, that imperative language provides into separate monads which can be combined at will.

At the same time, these primitives are given strong types which guarantee that only correct composition of components is allowed, and which guide the developer in the correct usage. Those types can help the teacher pinpoint the important properties and invariants of a given architectural pattern, using a more formal language instead of an ad-hoc description. We expect the types also help the students getting a better understanding of the material.

But the larger benefit we see from Haskell's strong typing system belong to the realm of reasoning:

- Haskell programmers are used (or should become used) to using equational reasoning inside their programs. In such a way, properties can be proved using a simpler algebraic style, instead of convoluted reasoning about the inner details of functions.

  An example of such property can be found in [9] regarding the `stm` primitives `orElse` and `retry`:

  ```
  M1 ‘orElse‘ (M2 ‘orElse‘ M3) = (M1 ‘orElse‘ M2) ‘orElse‘ M3
              retry ‘orElse‘ M = M
              M ‘orElse‘ retry = M
  ```

- In many cases, it's interesting to see how some architectural pattern can be implemented or simulated in term of others. Once again, the `stm` library provides an excellent example: it provides unbounded FIFO channels `TChan` implemented using only STM primitives. When looking at the implementation, students can learn how they can use channel functionality in a system where only an atomic, shared database, is available.

  Of course, this fact is in no way Haskell-specific. But the previous remarks concerning strong typing and equational reasoning helps into building confidence for students that those constructions indeed work. We can check that properties derive from the axioms of another pattern, or pinpoint extra assumptions that we need to make for everything to work.

As stated above, libraries which will be presented later in the paper define their interface in terms of a small set of primitives. This benefits looking at extra functionality that is needed in a real system in a compositional way. For example, resource management is usually a topic that is transversal to the architectural pattern in our system. The `resourcet` package, for example, pinpoints the specific primitives that you need in a system to provide safe allocation and deallocation of resources.

2.3 Side-effect: Learning More Haskell

The main aim in a Software Architecture course should be giving the students knowledge about the process of getting the high-level structure of a software system right, this is something obvious. But the idea of using Haskell as a vehicle for architectural patterns has also a nice side-effect: students can improve their Haskell skills at the same time.

Nowadays, Haskell is usually taught at program level: which are the basic functional constructs, how to use the most common abstractions and type classes, which are the most useful libraries, etcetera, at the source code level. But when the topic is about creating larger pieces of software or an entire system, the main tool is object-oriented programming. We strongly believe that Haskell is the right choice for large code bases, and that the user base of functional thinking should be enlarged. Embedding Haskell implementation into a Software Architecture course, as we propose, would help into this aim, too.

Advantages of using Haskell to explore architectural patterns
⋄ Patterns can be put in practice in code
⋄ Basic primitives point in libraries to basic components of the pattern
⋄ Strong typing encodes invariants and keeps code honest
⋄ Students keep learning Haskell

3 Main Patterns

At this point, the reader should already have the feeling that using Haskell libraries can indeed help students into absorbing architectural patterns in an easier and more structured way. But in this paper we do not only want to discuss the issue in abstract, thus we shall change the focus into which actual patterns benefit the most from this way of thinking and which are, in our opinion, the libraries that suit best each paradigm. In our search for libraries, we looked specially at two properties:

- The chosen libraries should be relatively well-known and used in the wild. In that way, students can reuse the Haskell knowledge they gain into other projects.
- We do not want to impose extra requirements on the programming side, apart from a basic knowledge of the Haskell language. Thus, we prefer libraries which do not resort into much Template Haskell or very fancy meta-programming la Template Haskell.

When chosing the patterns, we have looked at the Software Architecture course taught at Utrecht University [1]. Those patterns come indeed from the widely-used book by Len Bass et. al. [3], which gives us some confidence that what we propose can be used also in other formation centres.

3.1 Relaxed Layers – Monad Transformers

Monad transformers [14] are one of the best-known elements in the Haskell toolbox. In a system build from monad transformers, we see a set of functionality layers (implemented as monads), in which each layer is allowed to call the lower ones. Thus, it simulates quite well the *relaxed layers* architectural pattern.

The main trick is to be found in the `MonadTrans` type class from the `transformers` package:

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
  -- lift . return = return
  -- lift (m >>= f) = lift m >>= (lift . f)
```

By using `lift`, you can take a computation in some layer `m` and access its functionality in an upper layer `t m`. If you think `lift` as modelling the communication effort that must be done between two components, the need to use several `lifts` when stepping through multiple layers give the students the feeling that, although good design-wise, using a layered structure impose some extra costs in the system.

The strict separation between each transformer ensures that each layer can only access the lower ones through their public interface, which is usually encoded as a “monad class” inside the `mtl` package. This is an instance of the way in which we can use Haskell to ensure that the abstraction properties of this architectural pattern are not violated when using them in code. At the very end, each of the layers is instantiated with an actual implementation by using the corresponding `runMonadT` functions that are provided for each monad transformer. This works as sort of dependency injection, and once again ensures that we cannot access inner details of each layer when working with it from upper ones. Indeed, each layer in the system could be in a separate logical or physical machine and that access may turn out to be impossible in a real system.

We mentioned that monad transformers encode the relaxed layered pattern, in contrast to the *strict* one. This is because by using several calls to `lift`, a layer in a monad stack is allowed to access not only the very next layer, but also any lower ones. In strict layered systems, layers do not have this functionality

[1]The schedule can be found at `http://www.cs.uu.nl/docs/vakken/mswa/`.

available: any information from a further layer must be obtain from calls to the very next one. We do not know of any library available to Haskell developers which may suit this paradigm better, although one could argue that hiding the `MonadTrans` instance for a specific layer would disallow that by-pass. Discussion about the pros and cons of allowing this direct connection between components without using intermediate layers can be quite useful at the classroom to understand the trade-offs between performance and maintainability in the system.

Very often, layered systems include more than one software component per layer. However, monad transformers only allow "vertical composition", where one layer wraps the entirety of lower ones. If "horizontal composition" is desired, monad coproducts [15] can help bridging the gap.

3.1.1 Non-Commutativity and Control Flow

Monad transformers show an interesting aspect of layered systems: even if components are independent, the order in which you put them together may influence the overall functionality. In more formal terms, this is usually called the *non-commutativity* of monad composition. As an example, if you have state and error monads, you can either wrap errors with a layer of state, in which case you obtain semantics similar to exceptions in imperative languages; or you can do it in the reverse other, effectively making exceptions return to the previous state, and thus simulating transactions.

We believe that a property like this one is quite interesting to show to students, which may naively assume that independent components in a system would never interfere with each other. Furthermore, this problem is quite explicit if you look at the types involved in the composition of monads, and highlight the power of string typing while learning new concepts as stated above.

In a traditional layered system, each of the components may only affect the *data flow*. That is, you can call functions or procedures in other layers, and get a result back. Monad transformers generalize this idea, by allowing layers affect also the *control flow* of the program. Let's take as an example the `List` monad: its effect is to spawn several computations, modelling non-determinism in a quite neat way. Or also an exception monad, which enables the program to use that error construct.

Many of the practical problems related to monad composition come from this control operations, rather than data ones. We suggest delving into some of this problems in a Software Architecture course, because it points out the important effect of layers which affect the control flow of the program.

The essential solution in this case is to add to each layer a way to save its "internal state" (whatever that means depends on the specific component itself) prior to going down in the stack. This is captured by the `MonadTrans` type class from package `monad-control`:

```
class MonadTrans t => MonadTransControl t where
  data StT t :: * -> *
  liftWith :: Monad m => (Run t -> m a) -> t m a
  restoreT :: Monad m => m (StT t a) -> t m a
  -- liftWith . const . return = return
  -- liftWith (const (m >>= f)) = liftWith (const m) >>= liftWith . const . f
  -- liftWith (\run -> run t) >>= restoreT . return = t

type Run t = forall n b. Monad n => t n b -> n (StT t b)
```

A more in-depth discussion on this topic and how other libraries have approached a solution can be found within the layers package[2], which tries to provide a more correct implementation of control lifting, taking into consideration possible resource management issues.

Relaxed Layered Architecture = Monad Transformers
◇ Each monad in a stack models a layer in the system, and lift represents communication
◇ Composition order affects the functionality
◇ Control operations (such as exceptions) need special care

3.2 Broker – Effects

Instead of organising the system architecture in layers, one can make each component self-standing and have a mediator between them, which routes the requests to the correct component and takes care of relaying the answer too. This mediator is usually called the *broker* of the system.

The extensible-effects package [12] provides the same functionality of transformers, but built using a broker architecture. Each of the monad transformers in the previous section is now seen as a *effect* with a corresponding *handler*. Usually, when you want some work to be performed, you send the request to the broker, which takes care of sending the reply. But in this framework, you instead provide a *continuation*, which will be called as needed inside the corresponding handler.

Note that this is only at the level of implementation, in the outside everything looks quite similar to old plain monads. Take one of the examples in [12]:

```
example :: Member (Reader Int) r => Eff r Int
example = do v <- ask
             return (v + 1)
```

The only difference is the use of Eff and the constraint Member (Reader Int) r. This documents that for this code to work, you will eventually need to inject a component that handles requests of type Reader Int. You can do so via runReader:

```
runReader :: Eff (Reader e :> r) w -> e -> Eff r w
runReader m e = loop (admin m) where
  loop (Val x) = return x
  loop (E u)   = handleRelay u loop (\(Reader k) -> loop (k e))

newtype Reader e v = Reader (e -> v)
```

The type signature tells us that if you needed a Reader e, when using runReader this effect is taken as handled and you get back code with one requirement less. But, of course, the important part is in its implementation. Two functions are involved:

- admin asks the broker if there's a request available;
- handleRelay takes care of checking whether the request can be handled by this component (by inspecting the type of its possible arguments, in this case Reader) and execute the corresponding piece of code, or it should be given back to the broker to check for a new possible handler.

  In the example, if the request is of the correct type, we get a continuation k which is given the initial value e for the environment. Other effect will handle k in a different way.

[2]At http://hackage.haskell.org/package/layers/docs/Documentation-Layers-Overview.html.

It's worth pointing out that the need of each handler to call `handleRelay` stems from an architectural decision in the library: effects should be extensible. Thus, the broker cannot know a priori which kind of messages will be handled by each component, and needs to ask.

Broker Architecture = Extensible Effects
◇ Each component in the system is a *handler* of some *effects*
◇ The architecture of `extensible-effects` allows extending the set of components

3.3 Pipe-and-Filter – Streams and Pipes

In a pipe-and-filter system, a series of components is put in sequence: the output of one of them is the input of the next one in the queue. Each component may filter, transform, or produce new data it from the stream it receives. This model of computation is the one used in the terminal when using the `|` pipe.

In the Haskell world, a similar kind of composition is available under various names such as iteratees, pipes and conduits, each of them with a similarly named library in Hackage. Those libraries were not only developed to support a pipe-and-filter paradigm, but also to deal with problems related to lazy input/output in Haskell. History-wise, `iteratee` was the first attempt at building such a library, and more recently `pipes` was built with an emphasis on correctness and the use of category theory to drive the design, whereas `conduit` was developed with a more pragmatic view. Since one of our aims is supporting reasoning, we shall use `pipes` as our base, and add some comments about `conduit`.

Each component is represented by a value of the type `Pipe a b r`. The two basic operations that components of these systems support are `yield`, for generating one piece of data of type `b`, and `await`, for consuming one element of information of type `a`. At the end of the execution, a `Pipe` can return a final value of type `r`. To ease reasoning, better names are given to those `Pipes` which only support one kind of operation:

```
                                   -- Void is the empty type
await :: Consumer a a              type Consumer a r = Pipe a Void r
yield :: a -> Producer a ()        type Producer b r = Pipe Void b r
```

The naming for the `conduit` library is very similar: just rename `Pipe`, `Consumer` and `Producer` into `Conduit`, `Sink` and `Source`, respectively. The public interface of a component in a pipe-and-filter system is the type of data it produces and consumes, and this is encoded in the types.

Once you have each component in place, it's time to compose them. The main operator for doing so is `(>->)`, which can be thought as having the type:

```
(>->) :: Pipe a b r -> Pipe b c r -> Pipe a c r
```

Applying the type synonyms and some type-trickery, we can see that:

```
(>->) :: Producer a r -> Consumer a r -> r
```

That is, the library ensures that connecting some input generator to something that consumes it only leaves its final value as result. In the `conduit` world, the `(>->)` operator is called `(=$=)` instead, with a `($$)` variant to retrieve the end result.

Things become more interesting once we define `cat`, which just echoes everything it receives:

```
cat = forever $ await >> yield
```

Using this function, we can give a series of monoid-like laws for composing `Pipes`:

```
cat >-> f = f      f >-> cat = f      (f >-> g) >-> h = f >-> (g >-> h)
```

The above text discusses the basics of using `pipes` to model the pipe-and-filter architectural pattern. However, libraries support two extra features, which may be useful under some circumstances:

- Every `Pipe` can be allowed to do operations of a specific monad `m` apart from the conventional `yield` and `await`. For example, a pipe could write something to the disc, thus wrapping the `IO` monad. On the presence of those side-effects, the rules for `(>->)` become much more important.
- We discussed "horizontal composition" of pipes, where the output stream of one is connected to the next input stream. But in addition to those streams, `Pipes` can also produce final values. The `Monad` instance of `Pipe` allows "vertical composing" those elements.

The main distinction between `pipes` and `conduit` deserves some comments from the architectural point of view. The latter library support out-of-the-box the concept of *leftover*: the possibility of a certain component in the sequence to return back certain information to the input stream. For an example of when such functionality is important consider parsers: in some cases the entire system needs to backtrack to consider another branch in order to perform a successful parse tree.

Allowing leftovers means that either communication can flow also in the inverse direction to put back elements in the previous stream, or that each component comes with a local memory to handle those leftovers. We believe that discussing this issue in the classroom could be indeed very interesting.

As said before, `conduit` has native support for a `leftover` basic operation. In the `pipes` world, support for leftovers is built over the basic elements in the `pipes-parse` library. The way in which it is implemented is via the addition of a `StateT` layer inside the monadic effects of the pipe:

```
type Parser a m r = forall x . StateT (Producer a m x) m r
```

Thus, in this case the architectural decision was to save the leftover as an extra piece of information in some local memory. Note that the `pipes-parse` library also includes support for parsing via lenses, but we see those features very far from a real architectural pattern.

Pipe-and-Filter Architecture = Streams and Pipes
◇ A sequence of components filtering and modifying a stream can be simulated via `Pipes`
◇ The basic operations are `yield`, `await` and the composition `(>->)`
◇ Leftover support affects the architectural design of the system

3.3.1 The Power of `pipes`

Even though it may derail a bit from the purpose of showing architectural patterns, the `pipes` library has an interesting design which generalizes many of the patterns in this paper. If you look at the `Pipes.Core` module, you will notice that the real type underneath everything is not `Pipe`, but `Proxy`:

```
data Proxy a' a b' b m r
```

The difference between a `Proxy` and a regular `Pipe` is that the former is bidirectional: in one direction it takes `a` and produces `a'`, and in the other it takes `b'` and produces `b`. Apart from that, we have the final result type `r` and the inner effects monad `m`. In addition, if we have a function `e -> Proxy a' a b' b m r`, the value `e` can be seen as an entra input to the proxy.

The great thing is that we can get different patterns by "connecting the inner elements" in different ways[3]. When using pipes, only the part with `a` and `b` is used: `yield` correspond to connecting the `e` wire to `b`, and `await` to do so from `a` to `r`. Furthermore, `(>->)` will join `b` to the next `a`. Using other combinations you can simulate client-server relathionships, and both pull- and push-based Unix pipes.

[3]This is quite obvious from the pictures in the `Pipes.Core` module documentation.

3.4 Dataflow and MapReduce – `IVars`

In many cases, we can look at a whole system as a graph whose edges represent flow of information. Each component would be waiting certain inputs from other components, and work to produce a new value. Such a description is indeed very general: in this section we focus on the case where each component awaits until it receives the data from its input before resuming execution.

The Haskell library which suits better this paradigm is `monad-par`. As the name suggests, the original reason why it was developed was to support a certain kind of parallel execution [16], but we can see it also as a general dataflow modelling framework in the sense we defined earlier.

The `Par` monad, which sits at the core of the library, supports two modes of operation. The basic one involves the idea of *future*: some computation which happens in parallel to the rest of the program and which returns a token of type `future` that can be used later on to ask for the result of the computation.

```
class Monad m => ParFuture future m | m -> future where
  spawn :: m a -> m (future a)
  get :: future a -> m a
```

A more fine-grained design can be achieved using `IVars`: these are write-once boxes which can be used for communication between parallel processes. When a process tries to read an `IVar`, it's blocked until some value becomes available.

```
class ParFuture ivar m => ParIVar ivar m | m -> ivar where
  fork :: m () -> m ()
  new :: m (ivar a)
  put :: ivar a -> a -> m ()
```

Note that usually `future` = `ivar`. Thus, `get` is the function used to get the value of an `IVar`.

Looking at them from an architectural point of view, creating the processes corresponds to initializing each of the components which make up the system. Those components communicate through write-once links, which are represented by `IVars`.

Another possibility is to explore architectures where the developer is able to spawn computations in other nodes, like in a grid. In that case, Haskell has several tools available which allow students to measure the problems that spawning too few or too many computations may have in the system.

3.5 Simulating MapReduce

In present days, the MapReduce pattern [6] is coming more important to process large quantities of data. The main idea is to separate the computation in two phases: one which performs filtering or modification of the values, and another one which computes a final value by aggregating those values. In Haskell terms, we have a *map* phase and afterwards a *fold* phase.

The `monad-par` package can be used to simulate a MapReduce architecture too. You just need to spawn several processes which use a certain part of the input and calculate the desired output. The main process waits then until all the data is available to it and aggregates the values. Additionally, each node which handles part of the input can itself partition its computation on new processes. Knowing when that partition should stop is one of the key problems when using the MapReduce architectural pattern.

We think that MapReduce is one of the architectural patterns where our approach using Haskell excels. First of all, many components of the architecture can be understood in Haskell terms. From that, many properties can be derived: for example, the need of the aggregation function to be associative. In addition, common MapReduce frameworks are usually big and difficult to configure. Using Haskell, we can provide a playground for students to think how they would architect a toy system.

3.5.1 Dataflow with Lattices – `LVars`

Usually, we think of values communicated between components as something that must be completely known before the next computation can proceed. In some cases, such a dataflow graph we need a single value, in other such as pipes, we need the next element in the stream. But many gains can be achieved if components of an architecture can work with not-completely-known or approximate information. We feel that this topic is usually left out of Software Architecture courses, but once again Haskell provides a simple way to explain and understand such paradigm.

The `lvish` package provides such a model [13]. Within it, `IVars` are replaced with `LVars`, which are containers of data which allow multiple writes, given that they are monotonically increasing, i.e., they are higher in a lattice of values. When another process want to read an `LVar`, it provides a minimum threshold for the value, and blocks if it hasn't been reached yet. If this increase in the value is thought of as better or more precise information, we can readily apply it in many contexts.

Dataflow Graphs and MapReduce = `IVars` and `LVars`
◇ Dataflow graphs can be modelled using `IVars`
◇ MapReduce pattern can be simulated by spawning several computations waiting for `IVars`
◇ A system can generate monotonically increasing values by using `LVars`

3.6 Shared Database – Software Transactional Memory

Another useful architectural pattern entails a common database where components get and modify information. Integration happens thus at the data layer of the system, without any extra control orchestration between components. However, access to the shared database must still be done in a correct way, ensuring that data is persistent and consistent, as mandated by database standards.

Software Transactional Memory [9] provides a way to get those properties when accessing simple program variables. The basic idea is that we can protect a variable from going into an inconsistent state if we can rollback the changes done to it if the piece of code that must access it ends with a failure. If operations are allowed to fail by default, the system can also use optimistic locking for preventing interference between concurrent accesses.

With those constraints, Haskell becomes a perfect choice: since pure functions cannot have any side-effects we can easily retry them if something goes wrong. The `stm` package provides the actual implementation: every transaction must be implemented inside the `STM` monad, and may access one or more transactional variables, also known as `TVars`.

```
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
modifyTVar :: TVar a -> (a -> a) -> STM ()
```

Once you've composed a whole transaction using those basic pieces, you get back a value of type `STM b`. But their effects are not yet available, if you want to execute the transaction, you need to `atomically`[4]:

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

It may be the case that the transaction would lead to inconsistent state, in which case you should call `retry`. In that moment, the transaction will be blocked until new values are held in the `TVars`. Alternatively, you can call `orElse` to try another transaction in the first one fails:

[4]The output type is wrapped in `IO` because the outcome may depend on other transactions executing concurrently.

```
retry :: STM a
orElse :: STM a -> STM a -> STM a
```

As can be seen, the interface in this case is quite simple, and serves well to the task of maaking available to students an architectural pattern in a simple way.

Shared Database Architecture = Software Transactional Memory
◇ STM allows access to variables satisfying the ACID properties
◇ A shared database between components can be simulated using a set of `TVars`

3.7 Peer-to-Peer – Actors in Cloud Haskell

In the previous architectural patterns, we either have a central control or data bank, or several components which have the same "category". The peer-to-peer pattern allows decoupling even more the various agents participating in the system, which communicate to each other using just messages. Those components are distributed and collaborate to provide the final functionality in the system.

Cloud Haskell [8] allows creating a peer-to-peer system in our favorite language, using *actors*[5] as the main abstraction. In the package `distributed-process`, which implements those ideas, each actor is represented as a value of type `Process a`, with an unique `ProcessId` representing it uniquely in the network. When actors want to communicate they use `send` or `expect`, the latter used to receive a message by indicating its type:

```
send :: Serializable a => ProcessId -> a -> Process ()
expect :: forall a. Serializable a => Process a
```

If more than one type of message is supported by a certain actor, Cloud Haskell supplies the function `receiveWait`, which takes as argument a list of possible *matchers* for messages, which say whether they should be handled and how:

```
receiveWait :: [Match b] -> Process b
-- Build matchers in several ways
match :: forall a b. Serializable a => (a -> Process b) -> Match b
matchIf :: forall a b. Serializable a => (a -> Bool) -> (a -> Process b) -> Match b
matchUnknown :: Process b -> Match b
```

Being easy configurable, we think that Cloud Haskell could be great tool for students to experiment with peer-to-peer architectures, which are quite different for the most common client-server architectures that are used widely. Having to implement such a system, students are faced with questions such as how to ensure consistency of information between actors, how to manage resources or how to deal with unexpected network outages or failures.

3.7.1 Channels

One of the problems in which one may run in a completely free actor system is that one is allowed to send or receive any message in the system. This may not be what you want, and in this case the type system helps us being honest about our intentions, once more. The idea in Cloud Haskell is to use *channels*, which are message traders which only accept messages of one specific type:

[5]A concept already implemented in other systems such as Erlang.

```
newChan :: Serializable a => Process (SendPort a, ReceivePort a)
sendChan :: Serializable a => SendPort a -> a -> Process ()
receiveChan :: Serializable a => ReceivePort a -> Process a
```

When looking at these channels from an architectural point of view, it may be interesting to discuss in the classroom questions about whether these unidirectional links are enough for a typical system, what are the implications of being able or not to create new channels at will, or the simplifications achieved if channels are identified by their purpose in the system.

At this point, we can get back to one of our proposals for using Haskell in a Software Architecture class: it allows us bridging concepts of different architectures, looking how one would recreate parts of one if only resources of other are available. In this case, the focus would be in the channel implementations in the `stm` library. Take as an example the `TQueue` data type. Thanks to the open source philosophy of this library, we can see how the constructors are defined:

```
data TQueue a = TQueue {-# UNPACK #-} !(TVar [a])
                       {-# UNPACK #-} !(TVar [a])
```

The rest of the module provides the actual implementation of a channel in terms of STM primitives, and may be interesting to look at from the perspective of students.

Peer-to-Peer Architecture = Cloud Haskell
◇ Cloud Haskell follows the actor abstraction to provide a peer-to-peer architecture
◇ Actors may communicate by untyped messages or by typed channels
◇ `stm` provides an interesting implementation of channels in a shared database context

3.8 Publish-Subscribe – Reactive

Usually components in a system are though of *pulling* data from other components or a database. But in many cases it's useful to think it the other way around, and architect our components to *react* to changes in the data or certain events. Those components *publishing* the events need not know anything about the *subscribers* nor subscribers need to be aware of other components also listening at the same changes.

We can model those systems in Haskell using Functional Reactive Programming [7]. Many variations of this idea have been developed during the years, a good survey material is [1]. Without any kind of implication further than a subjective sense of easier usage, the examples will use the `sodium` library. The most basic functions in this case are the one which generates a new event source, namely `newEvent`, and that which subscribe to a certain publisher, named `listen`:

```
newEvent :: Reactive (Event a, a -> Reactive ())
listen :: Event a -> (a -> IO ()) -> Reactive (IO ())
```

If wondering how to publish new elements, the solution is to use the function returned as second element of the tuple by `newEvent`. `sodium` focuses on events which will have side-effectful counterparts, other libraries allow modelling FRP without side-effects.

An interesting feature is that `Events` are reified to be first-class citizens and can be combined at will. FRP libraries tend to provide a bunch of those functions, here's a small subset of them:

```
merge :: Event a -> Event a -> Event a          -- Any of them
once :: Event a -> Event a                      -- Only the first
split :: Event [a] -> Event a                   -- One event per element
filterE :: (a -> Bool) -> Event a -> Event a    -- Filter out
```

A nice generalization of this pattern, and which points to another interesting architectural design, involves replacing discrete events by values which may change continuously. Think of a textbox: instead of having an `Event` per keystroke, you have a `Behavior` which encapsulates the text value at any moment in time. In code, creation is similar to events:

```
newBehaviorSource :: a -> Reactive (Behavior a, a -> Reactive ())
```

However, a computer is a discrete machine and thus cannot react continuously. The way in which one manage behaviors is by converting it to one event per change:

```
updates :: Behavior a -> Event a
```

Behaviors are a really powerful abstraction, which can be used to model and simulate many systems. For example, `switch` and `switchE` allows creating behaviors or event sources which change over time, that is, that are behaviors themselves:

```
switch :: Behavior (Behavior a) -> Reactive (Behavior a)
switchE :: Behavior (Event a) -> Event a
```

We feel that introducing events and behaviors as patterns driving the architecture of a system introduce a novel viewpoint for students on the flow of information in the system, which are used to request-response patterns as found in function calls and client-server architectures.

Publish-Subscribe Architecture = Functional Reactive Programming
◇ `Events` encapsulate a channel where a publisher can inform subscribers of changes
◇ `Behaviors` model values which vary continuously over time

3.9 Resource Management – `pipes-safe` and `resourcet`

Although tangential as a topic to core Software Architecture, resource management may impose important restrictions on how a system can be designed. A very nice architectural design may fall apart once we take into consideration things such as how is access to files or connections mediated, or when are resources acquired and released. Since Haskell forces us to be honest about when we are performing these side-effects, we see it as a great tool for considering how the architecture has to change to handle those new requirements.

As an interesting case, we shall focus on the pipes-and-filter architectural pattern. This case is interesting, because if data is asked on demand, when other components need it, the previous steps may need to keep some resources open until they ensure that the next has consumed everything they need.

In the Haskell world two libraries, `pipes-safe` and `resourcet` (from the `conduit` developers), take a similar approach to solve the problem. In both cases, an extra monadic layer is added to the mix: `SafeT` and `ResourceT`, respectively. Below are the basic operations of the first one:

```
register :: m () -> SafeT m ReleaseKey
release  :: ReleaseKey -> SafeT m ()
```

The first operation *registers* some cleanup action, like closing a file handle or a socket, which is ensured to be run at later when the flow escapes from the `SafeT` monadic environment. For the case where cleanup should be done before, the `register` function provides a `ReleaseKey` which can be used to run that cleanup via `release`. This flow corresponds closely to the three stages of resource management in Software Architecture as mentioned in [11]: acquisition, lifecycle and release.

We shall remark that other patterns of resource management are also available in Hackage. For example, `resource-pool` provides a simple interface for a resource pool. We omit further discussion for space reasons.

Resource Management
◇ Resource management is a tangential topic to Software Architecture
◇ Haskell shows how to add a layer of resource management to several architectural patterns

4 Conclusions

We have seen many examples of how the discussion of a Haskell libraries enhances the explanation of architectural patterns in the classroom by:

- providing a ground for implementation and experimentation;
- focusing on the very core set of primitives, and looking at the properties they satisfy;
- making natural to ask questions about how to enlarge the architecture to support more features.

We haven't yet applied these ideas in any actual classroom, but are confident that functional programming in general, and Haskell in particular, can be used successfully in this way.

References

[1] Edward Amsden (2011): *A Survey of Functional Reactive Programming*. *Independent Study in Functional Reactive Programming, Spring 2010-2011*. Available at `http://www.cs.rit.edu/~eca7215/frp-independent-study/Survey.pdf`.

[2] USENIX Association, editor (2004): *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. USENIX Association.

[3] Len Bass, Paul Clements & Rick Kazman (2012): *Software Architecture in Practice*, 3rd edition. Addison-Wesley Professional.

[4] Koen Claessen, editor (2011): *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. ACM.

[5] Ron K. Cytron & Peter Lee, editors (1995): *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. ACM Press. Available at `http://dl.acm.org/citation.cfm?id=199448`.

[6] Jeffrey Dean & Sanjay Ghemawat (2004): *MapReduce: Simplified Data Processing on Large Clusters*. In Association [2], pp. 137–150. Available at `http://www.usenix.org/events/osdi04/tech/dean.html`.

[7] Conal Elliott & Paul Hudak (1997): *Functional Reactive Animation*. In Jones et al. [10], pp. 263–273. Available at `http://doi.acm.org/10.1145/258948.258973`.

[8] Jeff Epstein, Andrew P. Black & Simon L. Peyton Jones (2011): *Towards Haskell in the cloud*. In Claessen [4], pp. 118–129. Available at `http://doi.acm.org/10.1145/2034675.2034690`.

[9] Tim Harris, Simon Marlow, Simon L. Peyton Jones & Maurice Herlihy (2008): *Composable memory transactions*. *Commun. ACM* 51(8), pp. 91–100. Available at `http://doi.acm.org/10.1145/1378704.1378725`.

[10] Simon L. Peyton Jones, Mads Tofte & A. Michael Berman, editors (1997): *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*. ACM.

[11] Michael Kircher & Prashant Jain (2004): *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley, Chichester, UK.

[12] Oleg Kiselyov, Amr Sabry & Cameron Swords (2013): *Extensible effects: an alternative to monad transformers*. In chieh Shan [17], pp. 59–70. Available at `http://doi.acm.org/10.1145/2503778.2503791`.

[13] Lindsey Kuper & Ryan R. Newton (2013): *LVars: Lattice-based Data Structures for Deterministic Parallelism*. In: Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC '13, ACM, New York, NY, USA, pp. 71–84, doi:10.1145/2502323.2502326. Available at `http://doi.acm.org/10.1145/2502323.2502326`.

[14] Sheng Liang, Paul Hudak & Mark P. Jones (1995): *Monad Transformers and Modular Interpreters*. In Cytron & Lee [5], pp. 333–343. Available at `http://doi.acm.org/10.1145/199448.199528`.

[15] Christoph Lüth & Neil Ghani (2002): *Composing monads using coproducts*. In Wand & Jones [18], pp. 133–144. Available at `http://doi.acm.org/10.1145/581478.581492`.

[16] Simon Marlow, Ryan Newton & Simon L. Peyton Jones (2011): *A monad for deterministic parallelism*. In Claessen [4], pp. 71–82. Available at `http://doi.acm.org/10.1145/2034675.2034685`.

[17] Chung chieh Shan, editor (2013): *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*. ACM. Available at `http://dl.acm.org/citation.cfm?id=2503778`.

[18] Mitchell Wand & Simon L. Peyton Jones, editors (2002): *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*. ACM.