The Monad.Reader Issue 15

by Heinrich Apfelmus ⟨apfelmus@quantentunnel.de⟩
and Andrew Coppin ⟨MathematicalOrchid@hotmail.com⟩
and Gergely Patai ⟨patai@iit.bme.hu⟩
and Edward Z. Yang ⟨ezyang@mit.edu⟩

25 January, 2010

Brent Yorgey, editor.

Contents

Editorial

by Brent Yorgey ⟨byorgey@cis.upenn.edu⟩

Editor's block. That dreaded feeling of staring at a blank LaTeX document which is supposed to contain some sort of publication of which you are the editor... I'm sure we've all been there at some point. Emacs doesn't write articles on its own (`M-x write-article` isn't even defined), and you can't actually write anything yourself because, you know, you are the **editor**...

Of course, there's only one way out: to beg, plead, wheedle, cajole, arm-wrestle, bribe, blackmail, and otherwise coerce people to write things to fill your pathetic, empty document. For my first issue as editor of The Monad.Reader, I feared the lengths to which I might have to go. I hoped against hope that promising my second-born child would be sufficient.

I was therefore pleasantly surprised – nay, astonished – nay, **flabbergasted** to receive not one, not two, but four excellent, entirely unsolicited submissions for this issue. Gergely Patai describes his Google Summer of Code project building tools for Haskell space profiling; Edward Yang gives a readable presentation of three lesser-known monads and their uses; Heinrich Apfelmus writes an insightful tutorial on designing and implementing monads with an operational approach; and Andrew Coppin presents a pure Haskell implementation of software transactional memory.

I can only suppose that next time, the other shoe will drop and I will actually receive a negative number of submissions. So enjoy this issue while you can.

The hp2any project

by Gergely Patai ⟨patai@iit.bme.hu⟩

The name hp2any *is probably the most compact way to summarise the goal of my project: improving Haskell space profiling experience in as many ways as one can imagine. This is obviously an open-ended adventure, and the summer was really just about taking the first steps. In this article, I'll tell you about what I achieved so far as well as my future plans for the project.*

Introduction

Everyone who has ever tried space profiling Haskell programs knows about hp2ps [1], which has been the prevailing tool to work with heap profiles for the past few years. A few months ago, I was struggling with subtle laziness issues that caused memory leaks in an OpenGL game I was working on. While generating nice PostScript graphs was tremendously helpful in solving the problem, the whole process of trying and retrying and jumping around between the editor, my program, and the PostScript viewer was slightly frustrating after a while. In the end, I had a fresh itch to scratch just at the time when this year's Summer of Code was announced.

My original plan was to create a user-friendly application that makes the profiling experience much smoother by removing some of the extra jumps. This application would take care of passing profiling parameters, displaying the heap profile on the fly, loading and displaying graphs from previous runs and even exporting these graphs to other formats, so hp2ps could be finally retired.

By now, many of the features I had in mind are implemented, while some of my ideas ended up in the trash. You can see the current state of the project and follow future developments at its HaskellWiki entry [2] and its source repository [3]. This article is an attempt to complement these resources by explaining the context of the project and recalling some summer memories.

Space profiling basics

Simply put, a heap profile is a time series of memory usage statistics. In the case of GHC, this data can be quite fine-grained, as one can monitor the amount of memory retained by any sub-expression. It is also possible to group data by other criteria, *e.g.* by type.

In order to get a heap profile, one needs to instrument the executable by marking so-called **cost centres** of interest using certain compiler options and – in the case of expression-level control – pragmas. Afterwards, one needs to pass some RTS options to the program to specify the characteristics of the heap profile. The profiling section of the manual gives an exhaustive list of the relevant parameters [4].

If all went well, we'll see a file called ⟨executable-name⟩.hp. Opening it in a text editor we can see that it's just a dump of timestamped snapshots of memory consumption data. The hp2ps utility can parse this file and turn it into a nice PostScript rendering.

That's basically all there is to it. Even though the manual describes a trick to look at the graph while the program is still running, a profiling session is essentially an off-line activity. We run the program, look at the graph, then adjust the program or some profiling parameter and repeat the process until we're happy with the result.

Components

It was clear from the beginning that the task could be decomposed in a meaningful way, and the final result would be a whole set of packages instead of a single application. I identified the following components:

- data structures for live streaming and offline loading along with the functions that create them (*i.e.* core functionality);
- live grapher application;
- plugins for exporting to other formats (graphics or input for other profiling tools);
- a GUI application providing a simple interface for all of the above.

I separated the live grapher from the manager application due to an external constraint: there is no standard windowing toolkit readily available to every Haskell user, while OpenGL support is part of the Haskell Platform, and generally much easier to set up on supported platforms. Therefore, I decided to create a standalone grapher that depends only on OpenGL.

The third item of the list got the lowest priority, since I figured that if people could live with hp2ps for so many years, they can just keep using that until hp2any

has something better to offer. As it turns out, I didn't even get to start working on this item by the final deadline...

Core library

The hp2any-core package is the most important part of the toolset: it provides the basic functionality to get heap profiles from running programs as well as old runs, and query the data for statistics.

The core functionality is in the Read module. It defines the interface to load heap profiles from former runs and to read the heap profile of a running process while it is being generated. Both operations have two alternatives: loading can be synchronous or asynchronous, and in the latter case cancellable too, while in the case of live profiling we can decide whether we want the library to manage the data or we'd like to do that ourselves – through a callback function that's repeatedly invoked with the raw heap snapshots whenever they are produced.

The Types module is also vital, as it defines all the common data structures used by the library. In particular, it contains a type class that's essentially an interface to make queries on heap profiles. This makes it possible to have several representations, each tuned for a certain usage pattern. The Types module defines one of its instances, which is used during loading, as it is easy to update. However, this basic data structure provides slow queries. In contrast, the Stats module gives us a data structure optimised for querying.

In addition to the above, there are two auxiliary modules in the package. In Process, we can find a helper function to define the process to be fired up by the live profiling functions in Read. Finally, the Network module defines the protocol used during remote profiling. Curiously, it includes a simple applicative style parser to read the serialised structures. While I could have used one of the existing libraries for this purpose, I didn't want to add an extra dependency for such a simple task. The parser took at most an hour to implement, which is probably less than it would take to even pick the most suitable one, let alone familiarise myself with it, so I believe this was the right decision.

As an aside, it's worth noting that creating the parser from scratch made me truly understand the original motivation behind introducing arrows into Haskell [5]. In retrospect, this is not very surprising, since the paper builds upon the same application, but it was still a notable moment of enlightenment for me.

Live grapher

The hp2any-graph package is responsible for visualising the heap profiles. First and foremost, it contains a grapher application (also called hp2any-graph) that can invoke the application we're interested in, start reading its heap profile as it is

generated and display it in a window using OpenGL. It is also possible to do this remotely: the hp2any-relay program that's also part of the package opens a server socket instead of a window, and any number of hp2any-graph's running on other hosts can connect to it later. These latecomers can't see the output generated before joining in.

Originally, I intended the grapher to be a generic application that could be remote controlled programmatically through a socket, so any other program could use it. However, it quickly dawned on me that this solution was way too complicated, and I ended up exposing the graphing functionality through a library interface instead. Less indirection, less complication. At the moment, this interface is used by the live grapher as well as the history manager.

History manager

The unifying front-end to all the other components resides in the hp2any-manager package, which contains an executable of the same name. At the time of writing this article, this is nothing more than an application for viewing .hp files. However, my original plan was to create a common UI for all the profiling related activity from specifying profiling parameters and observing live graphs to converting heap profiles into various image formats. I haven't given up on any of these features, but I decided to put them on hold and concentrate on general clean-up by the end of the official timeline.

Since I had no prior GUI programming experience in Haskell, it was also a nice opportunity to learn. After some pondering I chose Gtk2Hs, partly because it seemed the most mature GUI toolkit binding, and partly because it is also used by other developer tools like ThreadScope [6] and vacuum-cairo [7]. I don't think this was a bad decision, as the API is quite straightforward and the examples are helpful. However, I still ran into strange bugs that hindered my progression sometimes, so I had to cut back on the feature set I wanted to implement before the firm pencils down date. All in all, development went smoothly despite these occasional setbacks, and it was a pleasant experience overall.

Looking forward

Summer of Code was a great opportunity to start a new project from scratch, but there is still a lot to do in order to get a useful tool. The most pressing issue at the moment is Windows support. Unfortunately, live profiling doesn't work due to the dreaded sharing violation problem when we try to read the .hp file before the process writing it terminates. After resolving this, the next step should be improving the interface of the manager application to support live profiling and

make the compile-invoke-adjust cycle shorter. Naturally, I haven't given up on creating converters to substitute and extend hp2ps, and integrate them into the manager too. Also, the graphing code could be improved a lot, *e.g.* by using vertex buffers instead of display lists for more flexibility, which would make it easier to add more ways to display graphs – again something that was in my original plans but didn't fit in the summer.

On the whole, I'm mostly satisfied with the outcome of the project. On one hand I implemented fewer features than I thought I would, but I think I managed not to lose too much time when running into roadblocks by successfully adapting to the situation and rescheduling the subtasks. Besides, I could become a contributing member of the community, which is much more important in the long run.

Finally...

Everyone who was involved in this work in any way deserves thanks, especially my mentor, Johan Tibell, for all his advice and free reality check services.

About the author

Gergely Patai is a researcher at Budapest University of Technology and Economics (BME) currently focusing on functional reactive programming, and continuously trying to contaminate unsuspecting young souls with a love for lambdas. He has a general passion for languages, and walking is his preferred way of getting around when feasible.

References

[1] `http://www.haskell.org/ghc/docs/latest/html/users_guide/hp2ps.html`.

[2] `http://www.haskell.org/haskellwiki/Hp2any`.

[3] `http://code.google.com/p/hp2any/`.

[4] `http://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html`.

[5] John Hughes. Generalising monads to arrows. **Science of Computer Programming**, 37:pages 67–111 (1998).

[6] `http://raintown.org/?page_id=132`.

[7] `http://hackage.haskell.org/package/vacuum-cairo`.

Adventures in Three Monads

by Edward Z. Yang ⟨ezyang@mit.edu⟩

The standard monad libraries define a number of "bread and butter" monads, including the `State`*,* `Reader`*,* `Writer`*,* `[]`*, and* `Maybe` *monads. However, they are not the only monads available to an enterprising Haskeller's toolbox. In this text, we look at three other monads—the* `Logic` *monad, the* `Prompt` *monad, and the* `Failure` *monad—each of which tackle a common problem found in the engineering of programs.*

The Logic monad

The `Logic` monad [1] implements "backtracking computations". It is provided by the `logict` package. Backtracking is a general approach that can be applied to many search problems: given a partial solution, we determine if we can generate any further partial solutions; if we cannot, we discard this partial solution and backtrack to an earlier point in the search tree. Backtracking tends to be faster than brute-force enumeration of the solution space, because if a partial solution violates a constraint, then further extensions of that partial solution can be eliminated, a process called pruning.

Strictly speaking, we don't need the `Logic` monad to implement backtracking: the list monad gives us nondeterministic computation, and over finite search spaces both the `Logic` monad and the list monad can give us the same answers. However, the `Logic` monad is much more efficient due to an underlying continuation-based implementation. Additionally, the `MonadLogic` type class exposes a few more operators that allow us to control when to perform a computation; this is a common practice in monad libraries, since it lets the interface be divorced from the actual implementation, whether it is the `Logic` monad, the `LogicT` transformer, or even the list monad. In this section, we will explicitly use `Logic` for brevity.

List monad equivalence

The `Logic` monad implements a strict superset of the list monad; as such, anything in the list monad can be directly translated into the `Logic` monad. Function signatures that have a type `[a]` now have a type `Logic a`; data in the list monad is transformed accordingly:

```
                    [1]  ⇒  return 1
                     []  ⇒  mzero
++ :: [a] -> [a] -> [a]  ⇒  mplus :: m a -> m a -> m a
 concat :: [[a]] -> [a]  ⇒  msum :: [m a] -> m a
               [1,2,3]   ⇒  (msum . map return) [1,2,3]
```

Notice that many of these transformations are simply generalizations of lists into the `MonadPlus` context. If we explicitly change `m a` to `[a]` in the type signatures of `mplus` and `msum`, we get back the original list operations.

To get data back out of the `Logic` monad (and back into lists), you can use the following transformations:

```
          id :: [a] -> [a]  ⇒  observeAll :: Logic a -> [a]
take :: Int -> [a] -> [a]   ⇒  observeMany :: Int -> Logic a -> [a]
          head :: [a] -> a  ⇒  observe :: Logic a -> a
```

You can see a brief example this equivalence in Listing 1.

Nondeterminism and backtracking

In `do` notation, the `<-` operator extracts a pure value from the monad. In the `Logic` monad, as in the list monad, the successive code takes on each value from the list in turn, and the results get concatenated together, similar to a `fork` operation in Unix. However, viewed as a search, the `<-` represents a branching operation: the list represents possible extensions of the current candidate solution, and we now select a single extension to further conduct search on.

The power in any monad is to hide away "incidental" details; in the case of the `Logic` monad the incidental detail is that we're doing a search over a nondeterministic computation, and write code as if it were deterministic. It's a little difficult to see this in a toy example, so instead we will develop code for a nondeterministic Turing machine.

A Turing machine consists of a tape, which we represent as two infinite lists of symbols, a head, which can write to a single symbol on the Turing machine, and an action table, which we represent as an array and a state register. An implementation of these data structures is shown in Listing 2. We also have two basic functions for manipulating the tape – writing and moving – in Listing 3. Our particular implementation is a two-state, five-symbol Turing machine.

```
choices :: MonadPlus m => [a] -> m a
choices = msum . map return

evensList :: [Int]
evensList = do
    n <- [1..]
    if n `mod` 2 == 0
        then [n]
        else []

evensLogic :: Logic Int
evensLogic = do
    n <- choices [1..]
    if n `mod` 2 == 0
        then return n
        else mzero
evensList' = observeAll evensLogic
```

Listing 1: Code in the List and `Logic` monads

A Turing machine operates by using its current state and symbol underneath the head to index into the action table. A deterministic Turing machine would have a single transition which encodes what the next state is, what symbol should be written to the tape, and what direction the head should move after writing the symbol. A nondeterministic Turing machine would have many possible transitions for each index, and the operator would be expected to keep track of the branching—thus `DTuringTransition` encodes a deterministic transition, whereas `TuringTransition` is nondeterministic.

We omit the deterministic implementation of a single step of running the Turing machine, precisely because the nondeterministic implementation in Listing 4 still communicates the essence of Turing machine execution clearly and can easily simulate the deterministic version.

If we'd like to use our nondeterministic Turing machine to search the space of deterministic Turing machines, we need to slightly modify the behavior of `stepMachine`: specifically, any time we make a choice with `<-`, we should stick with that choice for the rest of the machine's execution (notice, in the original implementation, that `machine` is a return value but is unchanged!) Amazingly, Listing 5 requires only a single extra line of bookkeeping (marked by `-- *`).

From there, performing a search for a machine involves having an initial "every

```
data RunningTuringMachine = RTM
    { rtmMachine :: TuringMachine
    , rtmTape    :: Tape
    , rtmState   :: TuringState
    } deriving (Show)

type TuringMachine      = Array TuringIndex TuringTransition
type TuringIndex        = (State, Symbol)
type TuringAction       = (TuringIndex, TuringTransition)
type TuringTransition   = Logic DTuringTransition
type DTuringTransition  = (TuringState, Symbol, TuringMove)

data Tape               = Tape [Symbol] Symbol [Symbol]
data TuringState        = Halt | State State
    deriving (Eq, Ord, Show)
data State              = StateA | StateB
    deriving (Eq, Ord, Show, Enum, Bounded, Ix)
data TuringMove         = MoveRight | MoveLeft | Stay
    deriving (Eq, Ord, Show, Enum, Bounded, Ix)
data Symbol             = SB | S0 | S1 | SL | SR
    deriving (Eq, Ord, Show, Enum, Bounded, Ix)
```

Listing 2: Data types for a nondeterministic Turing machine

```
moveTape :: TuringMove -> Tape -> Tape
moveTape Stay x = x
moveTape MoveRight (Tape left cur right) =
                    Tape (tail left)  (head left) (cur :right)
moveTape MoveLeft  (Tape left cur right) =
                    Tape (cur :left) (head right) (tail right)

writeTape :: Symbol -> Tape -> Tape
writeTape s (Tape left _ right) = Tape left s right
```

Listing 3: Tape manipulation functions

```
stepMachine :: RunningTuringMachine -> Logic RunningTuringMachine
stepMachine rtm@(RTM _ _ Halt) = return rtm
stepMachine (RTM machine tape@(Tape _ cur _) (State state)) = do
    (state', cur', move) <- machine ! (state, cur)
    let tape' = moveTape move $ writeTape cur' tape
    return $ RTM machine tape' state'
```

Listing 4: Nondeterministic step

```
stepMachine' :: RunningTuringMachine -> Logic RunningTuringMachine
stepMachine' rtm@(RTM _ _ Halt) = return rtm
stepMachine' (RTM machine tape@(Tape _ cur _) (State state)) = do
    trans@(state', cur', move) <- machine ! (state, cur)
    let tape' = moveTape move $ writeTape cur' tape
        machine' = machine // [((state, cur), return trans)] -- *
    return $ RTM machine' tape' state'
```

Listing 5: Step that collapses nondeterministic

machine" Turing machine, implemented in Listing 6 by filling every entry in the action table with "every transition", and then stepping through it and pruning results that don't halt or that give the wrong answer.

Fair disjunctions

A disjunction occurs whenever you combine the results of the `Logic` monad with `mplus`. The new computation `a ‘mplus‘ b` will return the results of `a` first and the results of `b` second. Shown in Listing 7 is a simple use of `mplus` to express all integers.

Unfortunately, if we try actually observing results from `integers` we find that it never returns any negative numbers: `choices [1..]` succeeds an infinite number of times.

The `Logic` monad exposes an alternative `mplus` called `interleave`, which interleaves the results of the monads it is combining, so that Listing 8 returns the following sequence:

$$0, 1, -1, 2, -2, 3, -3, 4, -4, \cdots$$

which guarantees that any integer n will be seen in finite time.

```
-- Generates all values of a bounded, indexable data type.
generate :: (Ix a, Bounded a) => [a]
generate = range (minBound, maxBound)

everyTransition :: TuringTransition
everyTransition = msum . map return $ generate

everyMachine :: TuringMachine
everyMachine = array (minBound, maxBound) $
                       zip (range (minBound, maxBound))
                           (repeat everyTransition)
```

Listing 6: Every transition, every machine

```
naiveIntegers :: Logic Integer
naiveIntegers = return 0 `mplus`
    choices [1..] `mplus` choices [-1,-2..]
```

Listing 7: Naive representation of $\mathbb{Z}$

```
fairIntegers :: Logic Integer
fairIntegers = return 0 `mplus`
    (choices [1..] `interleave` choices [-1,-2..])
```

Listing 8: Representation of $\mathbb{Z}$ with fair disjunctions

If there are more than two computations to interleave, care must be taken: the `interleave` operator is no longer associative. For example,

```
a `interleave` (b `interleave` c)
```

favors results of `a`, whereas

```
(a `interleave` b) `interleave` c
```

favors results of `c`. If we can make a balanced full binary tree, we can be completely fair. For example,

```
(a `interleave` c) `interleave` (b `interleave` d)
```

returns

$$a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2, \ldots$$

Fair conjunctions

We will unite the white rose and the red:—
Smile heaven upon this fair conjunction,
That long have frown'd upon their enmity!
—William Shakespeare, Richard III

A conjunction occurs whenever you extract a variable from the `Logic` monad with `(>>=)`; it is associated with choice. Shown in Listing 9 is a simple search for factorizations of an integer over $2^{\mathbb{N}} \times \mathbb{N}$.

```
nat :: Logic Int
nat = choices [0..]

naiveFactorize :: Int -> Logic (Int, Int)
naiveFactorize n =
    nat >>= \x ->
    nat >>= \y ->
    guard (2^x * y == n) >>
    return (x, y)
```

Listing 9: Naive factorization

Unfortunately, this code will only ever find a single pair of factors for any given number, namely, $(2^0, n)$. If we remove the `guard` line, we discover why: because the naturals are infinite, the monad has gotten "stuck" on $x = 0$, and we never try $x = \{1, 2, 3, \cdots\}$.

The distributivity law for `MonadPlus`,

```
(mplus a b) >>= k = mplus (a >>= k) (b >>= k)
```

suggests that we can use `interleave` to implement fair conjunctions. `Logic` does so, and exposes an alternate bind, `(>>-)`. Instead of pursuing a computation of a single choice to completion, it pursues the computation until a single result is found, and then begins computation of a different choice. This has some subtle implications, as can be seen in the restructuring of `factorize` in Listing 10.

```
fairGenerate :: Logic (Int, Int)
fairGenerate =
    nat >>- \x ->
    nat >>- \y ->
    return (x, y)

fairFactorize :: Int -> Logic (Int, Int)
fairFactorize n =
    fairGenerate >>= \(x, y) ->
    guard (2^x * y == n) >>
    return (x, y)
```

Listing 10: Factorization with fair conjunctions

We have split factorization into a generation step (which utilizes fair conjunctions) and a filtering step. Without this separation, the function diverges after returning two results. The `Logic` monad schedules computations for each of its choices. The fair conjunction is not actually completely "fair": if it were, it would never let any choice return more than one result in the case of infinitely many choices, and execution would look like:

$$0, 1, 2, 3, 4, \cdots$$

Instead, the `Logic` monad has a binary fair conjunction $\wedge$ and applies it recursively to the (unbalanced) tree of choices $c_1 \wedge (c_2 \wedge (c_3 \wedge \ldots))$, asymptotically allocating $1/2^k$ of the processing for the kth choice. In this case, assuming that each choice requires infinite computation, execution looks like this:

$$0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, \cdots$$

Recall that the `Logic` monad executes the first choice $x = 0$ and only switches to the next choice once a result is observed. The non-termination is then the writing

on the wall: factorize kicks out results for $x = 0$ and $x = 1$, and then returns to $x = 0$, on which in spins forever as there are no more results.

This means that computations with infinite failure can be extremely fragile: Oleg Kiselyov's example usage of `(>>-)` [1] does not terminate if an extra association is added after the `(>>-)` operator. These situations can generally be avoided by separating out interleaved generation and filtering, which removes the possibility of infinite failure.

Logical conditional

During the process of filtering candidate solutions, we can use standard `MonadPlus` functions such as `guard` to implement deterministic checks against the solution. However, there is not a good way to conditionalize on finite failure; that is, given a candidate solution, how might we spin off another nondeterministic computation to see if we want to carry on, or try something different? We could use `observeAll` to pull the data out of the `Logic` monad, and then check if the list is empty, but this defeats composability and isn't very efficient.

The `Logic` monad defines the `ifte` operator for precisely this case. The expression `ifte expr th el` is equivalent to `expr >>= th` if `expr` succeeds with at least one result, and equivalent to `el` if it does not. The intuition is similar to Haskell's `if..then..else` construct, except that the results of `expr` are made available to `th` if it succeeds. Oleg [1] suggests that `ifte` is especially useful for "explaining failure" and applying heuristics; an operator with identical semantics exists in Prolog with the name "soft cut".

Pruning

In many computations, we only care about the first result we find: we may be looking for a single counterexample. While laziness generally ensures that results become accessible "as they are needed", thus cutting down on wasteful computation, the ability to tell a computation that only the first result is needed means that we can free any memory that was being used to keep track of backtracking. We can indicate this using the `once` operator.

Further reading

For example uses of `ifte` and `once`, I highly recommend checking out the paper which defines this monad [1]. If you are not interested in how the `Logic` monad is implemented, skip the sections about `msplit`.

The `stream-monad` package [2, 3] implements some later research by Oleg on nondeterministic computations. It's a bit simpler than the `Logic` monad, and

provides more fair interleaving. It is not to be confused with the `stream-fusion` package, which also provides `Control.Monad.Stream`.

Luke Palmer has implemented the `Omega` monad [4], which is the list monad but does breadth-first search instead of depth-first search, and Sebastian Palmer has implemented the `Level` monad [5], which does breadth-first search as well as iterative deepening.

The Prompt Monad

The `Prompt` monad [6], and the associated type class `MonadPrompt`, give us the ability to restrict side effects with the type system, while giving us the flexibility to change flow control based on values that are normally stuck in the `IO` monad. It is provided by the `MonadPrompt` package.

The problem the `Prompt` monad solves requires a little motivation. Purity is a "big idea" in Haskell: it means that using just a type we can reason about the side effects a computation may have. Haskellers are encouraged to use as restrictive a type as possible to get the job done: pure code is best, and the smaller the monad stack the better.

However, the `IO` monad remains the elephant in the room: we "need" it to make side effects in the outside world, but the moment we put code in the `IO` monad we are letting it do anything it wants (such as fire ze missiles.) A simple workaround is to define a data type that encodes effectful actions we want to permit which our pure code gives to a small function in the `IO` monad responsible for actually executing these effects. The type system then guarantees that only those actions can be executed, and all is well in paradise... that is, until we want our code to respond to the external environment as well. The `Prompt` monad, as its name suggests, solves precisely this problem.

The `Prompt` monad is also an example of how monads can introduce an abstraction layer. We can swap out `IO` with some testing jig that generates and receives information as if it were the environment, without changing any of the code in the `Prompt` monad (this is quite difficult to do in traditional impure languages, which usually resort to grody metaprogramming tricks).

The Prompt API

For the `Logic` monad, we were able to appeal to our experience manipulating lists to figure out how to make the corresponding operations for the `Logic` monad. Unfortunately, the `Prompt` monad has no such equivalence; fortunately, the most commonly used portion of the monad is very short, as seen in Listing 11.

```
class Monad m => MonadPrompt p m where
    prompt :: p a -> m a

data Prompt p r
instance MonadPrompt p (Prompt p)
instance MonadPrompt p (PromptT p m)

runPrompt :: (forall a. p a -> a) -> Prompt p r -> r
runPromptM :: Monad m => (forall a. p a -> m a) -> Prompt p r -> m r
```

Listing 11: `Prompt` monad API

As with any monad API, there are three sections: the first is the most general monad type class which defines any special operations that are intrinsic to the monad; the second are instances that you'd actually use in your program; and the last are functions that let you actually run the monad.

The type class `MonadPrompt` defines a single function `prompt`, which takes as an argument `p a` representing the "request" being made, and returns the "response" inside the `Prompt` monad. We'll define values to pass as `p a` using generalized abstract data types, in order have more fine-tuned control over what `a`, the return type, is; when utilizing `Prompt` or `PromptT`, we pass simply `p` to indicate what types of requests the `Prompt` monad services. (For those of you paying attention to kinds, this might seem slightly unusual, since the kind of `p` is `* -> *`, so `Prompt` actually has kind `(* -> *) -> * -> *`, a type usually seen in monad transformers.)

The functions `runPrompt` and `runPromptM` take a function that converts our "requests" either into pure values or values inside a monad of the user's choosing, and run the prompt monad with that function. The `forall` in their type signatures indicate a rank-2 type, which is used in order to let the functions `p a -> a` and `p a -> m a` range over multiple values of `a` without getting "stuck" to a particular `a` once we've passed it to `runPrompt`. If this description seems confusing, don't worry; we'll be looking at the form of `p a` and the function `p a -> m a` closely in the following sections.

Generalized abstract data types

Idiomatic use of the `Prompt` monad involves generalized abstract data types (hereafter referred to as GADTs), so you'll need the `GADTs` GHC language extension. GADTs are an extension to normal abstract data types (the types you define using the `data` keyword) that allow richer return types for the data constructors—applications range from generic pretty-printing to strongly-typed evaluators [7].

In the case of the `Prompt` monad, we will define a GADT `Request` that we'll use to request values from outside the `Prompt` monad, and define a function of type `Request a -> IO a` to serve these requests. Without GADTs, we have no way of restricting `a` into a more specific type[1]; as Ryan Ingram says, "the GADT is serving as a witness of the type of response wanted by the [program]" [8].

To give you a feel for how GADTs are a superset of normal abstract data types, consider the following equivalent pieces of code in Listing 12. In the GADT ex-

```
data NormalExample a = Zero' | One' a | Two' a a

data GADTExample a where
    Zero :: GADTExample a
    One :: a -> GADTExample a
    Two :: a -> a -> GADTExample a
```

Listing 12: Syntax comparison for GADTs

ample, we make explicit the type signatures of the data constructors. Note that they still are data constructors, so we can still use pattern matching, and we can't return any old value (it has to be of type `GADTExample ?`, where `?` is some type, `a` in this case). In this case we don't need to make "definitions" for `Zero`, `One` and `Two` using the GADT syntax, but it does make clear their functional nature; for example, `Two` is curried and the type of `Two True` is `Bool -> GADTExample Bool`.

As mentioned before, GADTs allow richer return types; *e.g.* the return type need not be `GADTExample a`. Listing 13 contains an actual GADT that will serve as the basis for our `Prompt` example.

```
data Request a where
    Echo :: String -> Request ()
    GetLine :: Request (Maybe String)
    GetTime :: Request UTCTime
```

Listing 13: GADT for the `Prompt` monad

These data constructors look suspiciously like functions that "echo" and "get a line", returning a value in some sort of `Request` wrapper. And indeed, we've used the GADT in order to indicate both the input types and the output types of an

[1] Although, a less beautiful alternate implementation could be made with existential types.

effectful procedure. However, this type doesn't tell us how to go from input to output.

Running the monad

We need to define a function that converts `Request`s, which are plain old data types, into actual side-effects and values behind the scenes. The definition in Listing 14 is fairly straightforward, although we use some exception handling capabilities to represent lines retrieved from standard input as either `Just String` or `Nothing`, which indicates an end-of-file. Notice that `a` takes on multiple values depending on what `Request` is pattern-matched; for `Echo s` and `GetLine` it is `String`, but for `GetTime` it is `UTCTime`—this is a feature of GADTs.

```
handleIO :: Request a -> IO a
handleIO (Echo s) = putStrLn s
handleIO GetLine = catchJust
    (guard . isEOFError)
    (Just <$> getLine)
    (const (return Nothing))
handleIO GetTime = getCurrentTime
```

Listing 14: Handler function

With handler function and GADT in hand, we can now write the monadic prompt code in Listing 15 and execute it. `prompt` has the type `p a -> Prompt p a`. In our example, `p` is `Request`, and `a` is the return value of `Request`. We pass `prompt` our `Request`s, and we get the result of the request back, with `handleIO` pulling the strings behind the scenes.

```
runCat :: IO ()
runCat = runPromptM handleIO cat

cat :: Prompt Request ()
cat = do
    line <- prompt GetLine
    maybe (return ()) (\x -> prompt (Echo x) >> cat) line
```

Listing 15: Implementation of `cat`

Even better, since the monadic code makes no mention of the `IO` monad, we can easily swap out `handleIO` for some other function, for example, one that replays a transcript, as is seen in Listing 16. Instead of a handler function that shifts from the GADT to the `IO` monad, will shift to the `RWS` (Reader, Writer, State) monad to help us thread the transcript through the operation and collect the results of this computation.[2]

```
type Input  = [String]
type Output = [String]

handleRWS :: Request a -> RWS r Output Input a
handleRWS (Echo s) = tell (return s)
handleRWS GetLine = do
    lines <- get
    if null lines
        then return Nothing
        else do
            put (tail lines)
            return (Just (head lines))

rwsCat :: RWS r Output Input ()
rwsCat = runPromptM handleRWS cat

simulateCat :: Input -> Output
simulateCat input = snd $ evalRWS rwsCat undefined input
```

Listing 16: Pure simulation of `cat`

Further reading

The `Prompt` monad doesn't have to be used only for a command line interface; for example, Felipe Lessa explores several possibilities for hooking up the `Prompt` monad to GTK. [9]

While the `Prompt` monad works well with small programs, on the order of Unix utilities, it's unclear how well this monad scales to larger user applications that may be graphical, may have many more than a dozen ways for the user to interact with the application, or may require asynchronous interaction. This is an area of

[2] Disregard any naysayers claiming this is merely a very convoluted way of implementing `id` for `[String] -> [String]`.

active research: possible places to look include functional reactive programming (*c.f.* spreadsheets) and arrows.

The Failure Monad

The `Failure` monad [10] is not a monad *per se*, but a class `MonadFailure` for monads that can fail, possibly with error information. It is provided by the `control-monad-failure` and `control-monad-failure-mtl` packages (see page 31 for an explanation). There are also `Applicative` and `Functor` versions, although we will not discuss them here. The package grew out of a frustration with the variety of error handling mechanisms that abound between libraries on Hackage; given any function that may fail, you may get back a value wrapped in any of `Maybe`, `Either`, `ErrorT`, a custom error type, or perhaps get an exception, which cannot be caught until the `IO` monad. The dream is to automatically compose multiple calls to errorful functions.

The `Failure` monad doesn't quite fulfill the dream, but it's an important step in the right direction. The fact that it is a type class means that code can be written for some generic monad that may fail, and then the user of the code can instantiate whichever monad they wish to handle the error. If you are a library writer, you should strongly consider publishing an interface that is merely a generic `MonadFailure`: with some extra restrictions on the type, this interface can be made exactly backwards-compatible. And anyone, application writer or library writer, can use `MonadFailure` to delay any decision about which specific error wrapper to use until the error needs to be handled. This is good style and improves composability.

The Failure API

The `Failure` API is extremely simple, as shown in Listing 17, because it doesn't need to define any functions to run the monad; any monad that has a `MonadFailure` instance will have its own functions for running the monad. The single function `failure` takes an argument of the error type `e`, and can be used as any type within the monad.

```
class Failure e m where
    failure :: e -> m v
class (Monad m, Applicative m, Failure e m) => MonadFailure e m
```

Listing 17: `Failure` monad API

Conversions

The `MonadFailure` type class has two type parameters: `e`, which is the type of the data actually holding information about your error (`String`, `Error`, etc), and `m`, which is the actual monad that can fail. There are lots of ways to express failure in Haskell [11], so we'll demonstrate how to convert them to use the `MonadFailure` type class.

Consider the simple implementation of `safeHead` in Listing 18. The fact that this

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

Listing 18: `head` with `Maybe`

function emits no error information means we have some latitude when genericizing it. Listing 19 is one possible translation. Notice that `safeHeadFailure` can be

```
safeHeadFailure :: MonadFailure String m => [a] -> m a
safeHeadFailure [] = failure "empty list"
safeHeadFailure (x:xs) = return x

safeHead' :: [a] -> Maybe a
safeHead' = safeHeadFailure
```

Listing 19: `head` with `MonadFailure`

instantiated into the original, as is shown by `safeHead'`. You can instantiate any value of class `MonadFailure` into the `Maybe` monad without regard to the type of `e`, which is somewhat arbitrarily chosen to be `String` in this case.

Next, we'll consider converting from a monad that does retain error information, `Either`, in Listing 20. This code is written in a different style than the `Maybe` example; namely, it eschews explicit constructors of `Either` in favor of the monad functions `fail` and `return`. In fact, the code could work under any monadic type, although care should be taken since many monads don't have a meaningful implementation of `fail` and thus default to bottom, and additionally `Either` is not a monad by default; you must import `Control.Monad.Error` to make `Error e => Either e` a monad (and `String` just happens to be an instance of `Error`—if this seems convoluted, it's because it is). The conversion to

```
safeHeadPair :: [a] -> [b] -> Either String (a, b)
safeHeadPair [] [] = fail "both lists empty"
safeHeadPair [] _  = fail "first list empty"
safeHeadPair _ []  = fail "second list empty"
safeHeadPair (x:xs) (y:ys) = return (x, y)
```

Listing 20: Pair of heads of list with `Either String`

`MonadFailure` is a straightforward replacement of `fail` with `failure`, as seen in Listing 21.[3]

```
safeHeadPair' :: MonadFailure String m => [a] -> [b] -> m (a, b)
safeHeadPair' [] [] = failure "both lists empty"
safeHeadPair' [] _  = failure "first list empty"
safeHeadPair' _ []  = failure "second list empty"
safeHeadPair' (x:xs) (y:ys) = return (x, y)
```

Listing 21: Pair of heads of list with `Either String`

The final example in Listing 22 is a bit longer, and serves to illuminate the style of monadic programming that `MonadFailure` encourages, as well as demonstrate that using `Failure` can be useful even if you are not a library writer. We develop a simple system for checking out items from a library: these items are created with a checkout date and a due date. The `renew` function lets someone push their due date later, but only if the book hadn't already expired (in which case it errors).

There are a few features of note: we use `type` to explicitly create a monad stack called `TimeMonad`, which has the `Reader` monad capability to determine the current time, as well as the `Error` monad transformer, which permits us to error out. `MyError` is a userland data type that encodes errors that this application may emit; in practice the type would be much longer (a small 500-line application I wrote contained eighteen error constructors), and permits writing code in an "exception throwing" style without actually using asynchronous or imprecise exceptions: a code that throws an error bubbles up until some level of execution deals with it.

The `Error` monad is quite a heavy hammer, and I have initially written code in the `Maybe` monad, only to have to go on a search, replace and typecheck hunt when I realize `Nothing` isn't actually sufficient information when there are several

[3]Since `MonadFailure` requires the `Monad` instance on `m`, `fail` would probably work in the case of `e` being `String`.

```
type TimeMonad  = ErrorT MyError (Reader UTCTime)

data MyError = ExpiredError
             | MyParseError ParseError -- see Marshalling
             | MiscError String
    deriving (Show)
instance Error MyError where
    noMsg = MiscError "Unknown error"
    strMsg s = MiscError s

data Checkout = Checkout
    { checkoutName :: String
    , checkoutTime :: UTCTime
    , checkoutDue  :: UTCTime
    }

checkoutLength :: Checkout -> NominalDiffTime
checkoutLength c = diffUTCTime (checkoutDue c) (checkoutTime c)

shiftCheckoutTime :: Checkout -> UTCTime -> Checkout
shiftCheckoutTime c newTime = c
    { checkoutTime = newTime
    , checkoutDue  = addUTCTime (checkoutLength c) newTime
    }

renew :: Checkout -> TimeMonad Checkout
renew c = do
    curTime <- ask
    when (checkoutDue c < curTime) $ throwError ExpiredError
    return (shiftCheckoutTime c curTime)
```

Listing 22: A simple checkout renewal system

layers of code in the Maybe monad, all of which could have resulted in this error. With the `Failure` monad I can build in this capability from the start, but use it with the simpler `Maybe` monad interface unless I need detailed information about the error.

```
renew' :: (MonadReader UTCTime m, MonadFailure MyError m) =>
          Checkout -> m Checkout
renew' c = do
    curTime <- ask
    when (checkoutDue c < curTime) $ failure ExpiredError
    return (shiftCheckoutTime c curTime)
```

Listing 23: Implementation using type classes

Listing 23 contains two changes: the first is familiar; we've changed `throwError` to `failure`. The other is the changed function signature. The original implementation was tied to the `TimeMonad`; the new code is more generic because all the type requires is that the monad `m` have the `MonadReader UTCTime` and the `MonadFailure MyError` "capabilities"; the actual `m` we pass in could be arbitrarily more powerful but the type signature enforces that the resulting code will only use those capabilities. Additionally, the new type signature expresses the fact that the `Reader` monad and the `Failure` monad commute.

Marshalling

The `failure` package is still fairly nascent, so you are unlikely to see third-party libraries exporting functions with it... yet. In the meantime, `Control.Failure` exports the `try` function (part of the `Try` type class) which permits us to easily marshal values in other Monads into another `MonadFailure` form.[4] Its interface is described in Listing 24.

```
class Try m where
    type Error m
    try :: ApplicativeFailure (Error m) m' => m a -> m' a
```

Listing 24: Try API

[4]It works for applicatives too, as seen in the type signature.

If the input type `m` and output type `m'` are the same, try acts as an identity, as shown in Listing 25.[5]

```
maybeVal :: Maybe Int
maybeVal = try $ Just 3

eitherVal :: Either String ()
eitherVal = try $ Left "error"
```

Listing 25: Try as identity

However, in many cases, what we'd really like to do is take an arbitrary error type from some third-party library and convert it into our own, application specific error type. One simple way to do this is to have a wrapper constructor inside your error data type, and defer handling the error to your global error handling code.

Curiously enough, `MyError` from the `TimeMonad` example has a constructor defined just for Parsec! In Listing 26, we take a `ParseError` from Parsec and place it into `MyParseError`, which we defined previously in Listing 22. The instance "lifts"

```
instance Failure MyError m => Failure ParseError m where
    failure e = failure (MyParseError e)

theirParse :: Parser a -> String -> Either ParseError a
theirParse parser s = parse parser "" s

myParse :: (MonadFailure MyError m, Failure ParseError m) =>
           Parser a -> String -> m a
myParse parser s = try $ theirParse parser s
```

Listing 26: Implementation using type classes

the failure from `Either ParseError` into `Failure MyError m => m`. There is one oddity in this code, which is the specification of `Failure ParseError m` in the signature: without it, `m` is overly general and results in overlapping instances. If you instantiate `myParse` anywhere else in the module, Haskell will be able to infer the correct type, but otherwise you need that extra restriction.

[5] As of writing, the example for `Either` requires an import of `Control.Applicative` and, for at least `mtl`, an orphan instance of `Applicative` for `Either`.

We should note that there is a namespace collision between `failure` and `parsec` on `try`, so I suggest keeping your `parsec` code in one module and your `failure` code in another.

There is another way to pass around errors from arbitrary third parties: instead of defining an error type, define an error type class and write instances of it for every third-party error type you want to support. You don't even need `try`; any errorable type will cleanly "cast" into the more generalized type class. The downside of this approach is that this type class will have to support any type of operation you may want to do: it is the only interface you get for accessing error information.

Addendum

The `Failure` monad publishes two versions of its module: `Control.Monad.Failure` and `Control.Monad.Failure.MTL`. This stems from the fact that there are three widely recognized monad libraries inside Haskell: `mtl`, which comes by default with GHC; `transformers`, which defines monads in terms of transformers on top of the `Identity` monad; and `monadLib`, Galois' brainchild and similar to `transformers`. Each of these defines important monadic types and instances.

If you try mixing two libraries together, even indirectly (from an external library that imports a different monad library, you'll notice two things: first, you'll have numerous ambiguous occurrences of constructors from the monads, since both library will attempt to export its own, and second, you'll have overlapping instances as each library attempts to define its own. Furthermore, functions from the one library will refuse to take data from the other: defined in separate modules, they are different types.

Practically speaking, you should pick one of these libraries and stick with it. This article is written with `mtl`, and should be translatable to another monad library with a little coaxing.

Acknowledgments

I'd like to thank the SIPB members who originally introduced me to this wonderful language, Brent Yorgey, who with his wonderful Typeclassopedia convinced me to write something for The Monad.Reader and who was immensely helpful in the initial review process, as well as the denizens of `#haskell` who've put up with my ruminations and questions.

In particular, I'd like to thank Dan Doel (dolio), Walt Rorie-Baety (BMeph), Berengal, David House (dmhouse), Albert Lai (monochrom), lpsmith, elcerdo-querie, Ricardo Herrmann, T_S_, jolanda_ypsilon, Sebastian Fischer, Daniel Peebles (copumpkin), kmc, Andy Vogt (aavogt), and Wei Hu.

About the author

Edward Z. Yang [12] is a sophomore undergraduate student at MIT pursuing a Bachelor's in Computer Science. He enjoys crafting software, playing oboe and sailing.

References

[1] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In **ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming**, pages 192–203. ACM, New York, NY, USA (2005). `http://okmij.org/ftp/papers/LogicT.pdf`.

[2] Oleg Kiselyov. Simple fair and terminating backtracking monad transformer. `http://okmij.org/ftp/Computation/monads.html#fair-bt-stream`.

[3] Oleg Kiselyov and Sebastian Fischer. stream-monad: Simple, fair and terminating backtracking monad. `http://hackage.haskell.org/package/stream-monad`.

[4] Luke Palmer. control-monad-omega: A breadth-first list monad. `http://hackage.haskell.org/package/control-monad-omega`.

[5] Sebastian Fischer. level-monad: Non-Determinism Monad for Level-Wise Search. `http://hackage.haskell.org/package/level-monad`.

[6] Ryan Ingram and Bertram Felgenhauer. MonadPrompt: MonadPrompt, implementation and examples. `http://hackage.haskell.org/package/MonadPrompt`.

[7] Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In **OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**, pages 21–40. ACM, New York, NY, USA (2005). `http://research.microsoft.com/en-us/um/people/akenn/generics/gadtoop.pdf`.

[8] Ryan Ingram. An interesting monad: "Prompt". `http://www.mail-archive.com/haskell-cafe@haskell.org/msg33040.html`.

[9] Felipe Lessa. MonadPrompt + Gtk2Hs = ? `http://www.mail-archive.com/haskell-cafe@haskell.org/msg36256.html`.

[10] Pepe Iborra, Michael Snoyman, and Nicolas Pouillard. control-monad-failure: A class for monads which can fail with an error. `http://hackage.haskell.org/package/control-monad-failure`.

[11] Eric Kidd. 8 ways to report errors in Haskell. http://www.randomhacks.net/articles/2007/03/10/haskell-8-ways-to-report-errors.

[12] Edward Z. Yang. http://ezyang.com.

Appendix

```
instance Enum TuringState where
    succ Halt = State minBound
    succ (State x) = State (succ x)
    pred (State x) | x == minBound = Halt
                   | otherwise     = State (pred x)
    toEnum 0 = Halt
    toEnum n = State $ toEnum (n-1)
    fromEnum Halt = 0
    fromEnum (State x) = 1 + fromEnum x
instance Bounded TuringState where
    minBound = Halt
    maxBound = State maxBound
instance Ix TuringState where
    range (n, m) = [n..m]
    index (Halt, m) Halt = 0
    index (Halt, State m) (State x) = 1 + index (minBound, m) x
    index (State n, State m) (State x) = index (n, m) x
    inRange (Halt, _) Halt = True
    inRange (Halt, State m) (State x) = inRange (minBound, m) x
    inRange (State n, State m) (State x) = inRange (n, m) x
    inRange _ _ = False
```

Listing 27: Enum, Bounded and Ix instances for TuringState

```
instance Show Tape where
    show (Tape left cur right) = intercalate " " symbols
        where symbols = "..." : symbols' ++ ["..."]
              symbols' =
                    reverse (lshow 3 left) ++
                    ['*' : show cur ++ "*"] ++
                    lshow 17 right
              lshow n xs = map show $ take n xs
instance Show TuringTransition where
    show logic | null $ observeAll logic = "*undefined*"
               | otherwise = intercalate ", " strings
                    where strings = [show a, show b, show c]
                          (a, b, c) = observe logic
```

Listing 28: Show instances for `Tape` and `TuringTransition`

The Operational Monad Tutorial

by Heinrich Apfelmus

Another monad tutorial? Oh my god, why!? Fear not, this article is aimed at Haskellers who are already familiar with monads, though I have of course tried to keep the material as accessible as possible; the first two sections may serve as an initial introduction to monads and the monad laws for the brave.

In this tutorial, I would like to present monads from the viewpoint of ***operational semantics*** *[1] and how it makes designing and implementing new monads a piece of cake. Put differently,* `s -> (a,s)` *is not the only way to implement the state monad and this tutorial aims to present a much more systematic way. I think it is still regrettably underused, hence this text.*

Overview

The main idea is to view monads as a sequence of instructions to be executed by a machine, so that the task of implementing monads is equivalent to writing an interpreter. The introductory example will be a stack automaton, followed by a remark on a monad for random numbers. Then, to showcase the simplicity of this approach, we will implement backtracking parser combinators, culminating in a straightforward breadth-first implementation equivalent to Claessen's parallel parsing processes [2].

For those in the know, I'm basically going to present the principles of Chuan-kai Lin's Unimo paper [3]. The approach is neither new nor unfamiliar; for example, John Hughes [4] already used it to derive the state monad. But until I read Lin's paper, I did not understand how valuable it is when done systematically and in Haskell. Ryan Ingram's `MonadPrompt` package [5] is another recent formulation.

To encourage reuse, I have also released a package `operational` [6] on Hackage [7] which collects the generic bits of these ideas in a small library. For convenient study, the source code [8] from each section of this article is also available.

Stack Machine – List of Instructions?

Our introductory example will be a stack machine, *i.e.* an imperative mini-language featuring two instructions `push` and `pop` for pushing and popping values onto and from a stack.

In other words, I have imperative programs like the following in mind:

```
push 5; push 42; pop;
```

Instructions are separated by semicolons. As shown in figure 1, this program first puts the number `5` on the stack, then puts the number `42` on top of the stack and proceeds to remove it again.

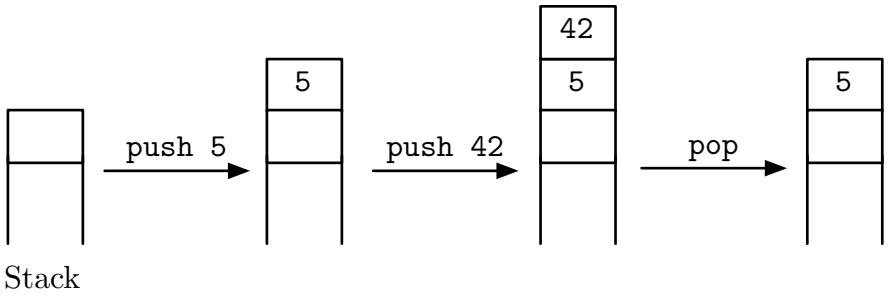

Figure 1: Example stack program.

How can we embed such programs into Haskell?

Representation

First we need some way of representing the program text, for instance as a list of instructions:

```
type Program instr    = [instr]
type StackProgram     = Program StackInstruction
data StackInstruction = Push Int | Pop
```

Our example is represented as

```
example = Push 5 : Push 42 : Pop : []
```

In a sense, the colon `(:)` for building lists takes the role of the semicolon for sequencing instructions.

Concatenation and thoughts on the interface

Note that this representation gives us a very convenient tool for assembling bigger programs from smaller subprograms: list concatenation `(++)`. For instance,

```
exampleTwice = example ++ example
     = Push 5 : Push 42 : Pop : Push 5 : Push 42 : Pop : []
```

is a program that executes `example` twice. Together with the empty program

```
empty = []
```

concatenation obeys the following three well-known laws:

```
 empty ++ is      =  is                  -- left unit
    is ++ empty   =  is                  -- right unit
(is ++ js) ++ ks  =  is ++ (js ++ ks)    -- associativity
```

which seem almost too evident to be worth mentioning. For example, it is customary to leave out the parenthesis in the last line altogether.

Once accustomed to the notion of programs and `(++)` to combine them, the special case of single instructions and `(:)` for sequencing them is unnecessary. The user of our language does not care that we deem `push` and `pop` to be primitive operations but not, for example, the program

```
replace a = Pop : Push a : []
```

which replaces the topmost stack element with `a`; he is entirely content to be given two programs

```
push :: Int -> StackProgram
pop  :: StackProgram
```

and two general combinators for building new ones

```
empty :: StackProgram
(++)  :: StackProgram -> StackProgram -> StackProgram
```

without any mention of the distinction between single instruction and compound program. Their difference is but an implementation detail.

Interpreter

Well, to be entirely content, the user also needs a way to run programs. In particular, we need to implement a function `interpret` that maps the program text to its intended meaning, here a function that transforms a stack of integers.

```
type Stack a = [a]

interpret :: StackProgram -> (Stack Int -> Stack Int)
```

The implementation follows the style of operational semantics: inspect the first instruction, change the stack accordingly, and recursively proceed with the remaining list of instructions `is`:

```
interpret (Push a : is) stack = interpret is (a : stack)
interpret (Pop    : is) stack = interpret is (tail stack)
interpret []            stack = stack
```

Oops

"All well and good, but why all the fuss with 'monads' then, when lists of instructions will do?" you may ask. Alas, the problem is of course that lists won't do! We forgot something very important: our programs are completely unable to inspect values from the stack.

For instance, how to write a program that pops the two topmost values and pushes their sum onto the stack? Clearly, we want something like

```
a <- pop;
b <- pop;
push (a+b);
```

where each `pop` returns the element just removed and the arrow `<-` binds it to a variable. But binding variables is simply impossible to express with our current representation of programs as lists of instructions.

Stack Machine – Monad

Well, if ordinary lists of instructions are not enough to represent programs that involve binding variables like

```
a <- pop; b <- pop; push (a+b);
```

then let's invent some fancy kind of list of instructions that will! The following presentation will be in close analogy to the structure of the previous section.

Representation

Return types

First, if we want to interpret `pop` as a function that returns something, we had better label it with the type of the value returned! Hence, instead of a plain type

```
Pop :: StackInstruction
```

we need an additional type argument

```
Pop :: StackInstruction Int
```

which indicates that the `Pop` instruction somehow returns a value of type `Int`.

For simplicity, we attribute a return type to `push` as well, even though it doesn't really return anything. This can modeled just fine with the unit type `()`.

```
Push 42 :: StackInstruction ()
Push    :: Int -> StackInstruction ()
```

Putting both together, our type of instructions will become

```
data StackInstruction a where
    Pop  :: StackInstruction Int
    Push :: Int -> StackInstruction ()
```

If this syntax is alien to you: this is a Generalized Algebraic Data Type [9] (GADT) which allows us to define a data type by declaring the types of its constructors directly. As of Haskell 2010, GADTs are not yet part of the language standard, but they are supported by GHC [10].

Like instructions, we also have to annotate programs with their return type, so that the definition for `StackProgram` becomes

```
data Program instr a where ...

type StackProgram a = Program StackInstruction a
```

As before, `instr` is the type of instructions, whereas `a` is the newly annotated return type.

Binding variables

How to represent the binding of variables? Lambda abstractions will do the trick; imagine the following:

Take a binding	`a <- pop; rest`
Turn the arrow to the right	`pop -> a; rest`
And use a lambda expression to move the arrow past the variable and semicolon	`pop; \a -> rest`

Voilà, the last step can be represented in Haskell, with a constructor named `Then` taking the role of the semicolon:

```
Pop `Then` \a -> rest
```

The idea is that `Then` plugs the value returned by `pop` into the variable `a`. By the way, this is akin to how `let` expressions can be expressed as lambda abstractions in Haskell:

```
let a = foo in bar   <=>   (\a -> bar) foo
```

Anyway, our motivating example can now be represented as

```
example2 = Pop `Then` (\a -> Pop `Then`
                         (\b -> Push (a+b) `Then` Return))
```

where `Return` represents the empty program which we will discuss in a moment. Remember that parentheses around the lambda expressions are optional, so we can also write

```
example2 = Pop `Then` \a ->
           Pop `Then` \b ->
           Push (a+b) `Then`
           Return
```

It is instructive to think about the type of `Then`. It has to be

```
Then :: instr a -> (a -> Program instr b) -> Program instr b
```

Except for the return type `a` in `instr a` and the lambda abstraction, this is entirely analogous to the "cons" operation `(:)` for lists.

Empty program

The empty program, corresponding to the empty list `[]`, is best represented by a constructor

```
Return :: a -> Program instr a
```

that is not "entirely empty" but rather denotes a trivial instruction that just returns the given value `a` (hence the name). This is very useful, since we can now choose return values freely. For instance,

```
example3 = Pop `Then` \a -> Pop `Then` \b -> Return (a*b)
```

is a program that pops two values from the stack but whose return value is their product.

The fancy list

Taking everything together, we obtain a fancy list of instructions, once again a GADT:

```
data Program instr a where
    Then   :: instr a -> (a -> Program instr b) -> Program instr b
    Return :: a -> Program instr a
```

And specialized to our stack machine language, we get

```
type StackProgram a = Program StackInstruction a
```

Interpreter

Before thinking thinking further about our new representation, let's first write the interpreter to see the stack machine in action. This time, however, we are not interested in the final stack, only in the value returned.

```
interpret :: StackProgram a -> (Stack Int -> a)
interpret (Push a `Then` is) stack     = interpret (is ()) (a:stack)
interpret (Pop    `Then` is) (b:stack) = interpret (is b ) stack
interpret (Return c)         stack     = c
```

The implementation is like the previous one, except that now, we also have to pass return values like `()` and `b` to the remaining instructions `is`.

Our example program executes as expected:

```
GHCi> interpret example3 [7,11]
77
```

Concatenation and interface

Just as with lists, we can build large programs by concatenating smaller subprograms. And as before, we don't want the user to bother with the distinction between single instruction and compound program.

We begin with the latter: the function

```
singleton :: instr a -> Program instr a
singleton i = i `Then` Return
```

takes the role of `\x -> [x]` and helps us blur the line between program and instructions:

```
pop  :: StackProgram Int
push :: Int -> StackProgram ()

pop  = singleton Pop
push = singleton . Push
```

Now, we define the concatenation operator (often dubbed "bind") that glues two programs together:

```
(>>=) :: Program i a -> (a -> Program i b) -> Program i b
(Return a)    >>= js  = js a
(i `Then` is) >>= js  = i `Then` (\a -> is a >>= js)
```

Apart from the new symbol `(>>=)` and the new type signature, the purpose and implementation is entirely analogous to `(++)`. And as before, together with the empty program,

```
return = Return
```

it obeys three evident laws

```
return a >>= is     = is a                          -- left unit
is >>= return       = is                            -- right unit
(is >>= js) >>= ks  = is >>= (\a -> js a >>= ks)  -- associativity
```

also called the **monad laws**. Since we need to pass return values, the laws are slightly different from the concatenation laws for ordinary lists, but their essence is the same.

The reason that these equations are called the "monad laws" is that any data type supporting two such operations and obeying the three laws is called a **monad**.

In Haskell, monads are assembled in the type class `Monad`, so we'd have to make an instance

```
instance Monad (Program instr) where
    (>>=)  = ...
    return = ...
```

This is similar to lists which are said to constitute a **monoid** [11].

We conclude the first part of this tutorial by remarking that the `(>>=)` operator is the basis for many other functions that build big programs from small ones; these can be found in the `Control.Monad` module and are described elsewhere [12].

What we've done so far

Those familiar with the state monad will recognize that the whole stack machine was just

```
State (Stack Int)
```

in disguise. But surprisingly, we haven't used the pattern `s -> (a,s)` for threading state anywhere! Instead, we were able to implement the equivalent of

```
evalState :: State s -> (s -> a)
```

directly, even though the type `s -> a` by itself is too "weak" to serve as an implementation of the state monad.

This is a very general phenomenon and it is of course the main benefit of the operational viewpoint and the new `Program instr a` type. No matter what we choose as interpreter function or instruction set, the monad laws for `(>>=)` and `return` will always hold, for they are entirely independent of these choices. This makes it much easier to define and implement new monads and the remainder of this article aims to give a taste of its power.

Multiple Interpreters

A first advantage of the operational approach is that it allows us to equip one and the same monad with multiple interpreters. We'll demonstrate this flexibility with an example monad `Random` that expresses randomness and probability distributions.

The ability to write multiple interpreters is also very useful for implementing games, specifically to account for both human and computer opponents as well as replaying a game from a script. This is what prompted Ryan Ingram to write his `MonadPrompt` package [5].

Random Numbers

At the heart of random computations is a type `Random a` which denotes **random variables** taking values in `a`. Traditionally, the type `a` would be a numeric type like `Int`, so that `Random Int` denotes "random numbers". But for the Haskell programmer, it is only natural to generalize it to any type `a`. This generalization is also very useful, because it reveals hidden structure: it turns out that `Random` is actually a monad.

There are two ways to implement this monad: one way is to interpret random variables as a recipe for creating pseudo-random values from a seed, which is commonly written

```
type Random a = StdGen -> (a,StdGen)
```

The other is to view them as a probability distribution, as for example expressed in probabilistic functional programming [13] as

```
type Probability = Double
type Random a    = [(a,Probability)]
```

Traditionally, we'd have to choose between one way or the other depending on the application. But with the operational approach, we can have our cake and eat it, too! The two ways of implementing random variables can be delegated to two different interpreter functions for one and the same monad `Random`.

For demonstration purposes, we represent `Random` as a language with just one instruction `uniform` that randomly selects an element from a list with uniform probability.

```
type Random a = Program RandomInstruction a

data RandomInstruction a where
    Uniform :: [a] -> RandomInstruction a

uniform :: [a] -> Random a
uniform = singleton . Uniform
```

For example, a roll of a die is modeled as

```
die :: Random Int
die = uniform [1..6]
```

and the sum of two dice rolls is

```
sum2Dies = die >>= \a -> die >>= \b -> return (a+b)
```

Now, the two different interpretations are: sampling a random variable by generating pseudo-random values

```
sample :: Random a -> StdGen -> (a,StdGen)
sample (Return a)             gen = (a,gen)
sample (Uniform xs `Then` is) gen = sample (is $ xs !! k) gen'
    where (k,gen') = System.Random.randomR (0,length xs-1) gen
```

and calculating its probability distribution

```
distribution :: Random a -> [(a,Probability)]
distribution (Return a)             = [(a,1)]
distribution (Uniform xs `Then` is) =
    [(a,p/n) | x <- xs, (a,p) <- distribution (is x)]
    where n = fromIntegral (length xs)
```

Truth to be told, the `distribution` interpreter has a flaw, namely that it never tallies the probabilities of equal outcomes. That's because this would require an additional `Eq a` constraint on the types of `return` and `(>>=)`, which is unfortunately not possible with the current `Monad` type class. A workaround for this known limitation can be found in the `norm` function from the paper [13] on probabilistic functional programming.

Monadic Parser Combinators

Now, it is time to demonstrate that the operational viewpoint also makes the implementation of otherwise advanced monads a piece of cake. Our example will be monadic parser combinators [14] and for the remainder of this article, I will assume that you are somewhat familiar with them already. The goal will be to derive an implementation of Koen Claessen's ideas [2] from scratch.

Primitives

At their core, monadic parser combinators are a monad `Parser` with just three primitives:

```
symbol :: Parser Char
mzero  :: Parser a
mplus  :: Parser a -> Parser a -> Parser a
```

which represent

- a parser that reads the next symbol from the input stream
- a parser that never succeeds
- a combinator that runs two parsers in parallel

respectively. (The last two operations define the `MonadPlus` type class.) Furthermore, we need an interpreter, *i.e.* a function

```
interpret :: Parser a -> (String -> [a])
```

that runs the parser on the string and returns all successful parses.

The three primitives are enough to express virtually any parsing problem; here is an example of a parser `number` that recognizes integers:

```
satisfies p = symbol >>= \c -> if p c then return c else mzero
many  p     = return [] `mplus` many1 p
many1 p     = liftM2 (:) p (many p)
digit       = satisfies isDigit >>= \c -> return (ord c - ord '0')
number      = many1 digit >>= return . foldl (\x d -> 10*x + d) 0
```

A first implementation

The instruction set for our parser language will of course consist of these three primitive operations:

```
data ParserInstruction a where
    Symbol :: ParserInstruction Char
    MZero  :: ParserInstruction a
    MPlus  :: Parser a -> Parser a -> ParserInstruction a

type Parser a = Program ParserInstruction a
```

A straightforward implementation of `interpret` looks like this:

```
interpret :: Parser a -> String -> [a]
interpret (Return a)           s = if null s then [a] else []
interpret (Symbol    `Then` is) s = case s of
    c:cs -> interpret (is c) cs
    []   -> []
interpret (MZero     `Then` is) s = []
interpret (MPlus p q `Then` is) s =
    interpret (p >>= is) s ++ interpret (q >>= is) s
```

For each instruction, we specify the intended effects, often calling `interpret` recursively on the remaining program `is`. In prose, the four cases are

- `Return` at the end of a program will return a result if the input was parsed completely.
- `Symbol` reads a single character from the input stream if available and fails otherwise.
- `MZero` returns an empty result immediately.
- `MPlus` runs two parsers in parallel and collects their results.

A note on technique

The cases for `MZero` and `MPlus` are a bit roundabout; the equations

```
interpret mzero       = \s -> []
interpret (mplus p q) = \s -> interpret p s ++ interpret q s
```

express our intention more plainly. Of course, these two equations do not constitute valid Haskell code for we may not pattern match on `mzero` or `mplus` directly. The only thing we may pattern match on is a constructor, for example like this

```
interpret (Mplus p q `Then` is) = ...
```

But even though our final Haskell code will have this form, this does not mean that jotting down the left hand side and thinking hard about the `...` is the best way to write Haskell code. No, we should rather use the full power of purely functional programming and use a more **calculational** approach, deriving the pattern matches from more evident equations like the ones above.

In this case, we can combine the two equations with the `MonadPlus` laws

```
    mzero >>= m = mzero
mplus p q >>= m = mplus (p >>= m) (q >>= m)
```

which specify how `mzero` and `mplus` interact with `(>>=)`, to derive the desired pattern match

```
  interpret (Mplus p q `Then` is)
=   { definition of concatenation and mplus }
  interpret (mplus p q >>= is)
=   { MonadPlus law }
  interpret (mplus (p >>= is) (q >>= is))
=   { intended meaning }
  \s -> interpret (p >>= is) s ++ interpret (q >>= is) s
```

Now, in light of the first step of this derivation, I even suggest to forget about constructors entirely and instead regard

```
interpret (mplus p q >>= is) = ...
```

as "valid" Haskell code; after all, it is straightforwardly converted to a valid pattern match. In other words, it is once again beneficial to not distinguish between single instructions and compound programs, at least in notation.

Depth-first

Unfortunately, our first implementation has a potential space leak, namely in the case

```
interpret (MPlus p q `Then` is) s =
    interpret (p >>= is) s ++ interpret (q >>= is) s
```

The string `s` is shared by the recursive calls and has to be held in memory for a long time.

In particular, the implementation will try to parse `s` with the parser `p >>= is` first, and then backtrack to the beginning of `s` to parse it again with the second alternative `q >>= is`. That's why this is called a **depth-first** or **backtracking** implementation. The string `s` has to be held in memory as long the second parser has not started yet.

Breadth-first

To ameliorate the space leak, we would like to create a **breadth-first** implementation, one which does not try alternative parsers in sequence, but rather keeps a collection of all possible alternatives and advances them at once.

How to make this precise? The key idea is the following equation:

```
  (symbol >>= is) `mplus` (symbol >>= js)
=  symbol >>= (\c -> is c `mplus` js c)
```

When the parsers on both sides of `mplus` are waiting for the next input symbol, we can group them together and make sure that the next symbol will be fetched only once from the input stream.

Clearly, this equation readily extends to more than two parsers, like for example

```
  (symbol >>= is) `mplus` (symbol >>= js) `mplus` (symbol >>= ks)
=  symbol >>= (\c -> is c `mplus` js c `mplus` ks c)
```

and so on.

We want to use this equation as a function definition, mapping the left hand side to the right hand side. Of course, we can't do so directly because the left hand side is not one of the four patterns we can match upon. But thanks to the `MonadPlus` laws, what we can do is to rewrite any parser into this form, namely with a function

```
expand :: Parser a -> [Parser a]
expand (MPlus p q `Then` is) = expand (p >>= is) ++
                               expand (q >>= is)
expand (MZero     `Then` is) = []
expand x                     = [x]
```

The idea is that `expand` fulfills

```
foldr mplus mzero . expand = id
```

and thus turns a parser into a list of summands which we now can pattern match upon. In other words, this function expands parsers matching `mzero >>= is` and `mplus p q >>= is` until only summands of the form `symbol >>= is` and `return a` remain.

With the parser expressed as a big "sum", we can now apply our key idea and group all summands of the form `symbol >>= is`; and we also have to take care of the other summands of the form `return a`. The following definition will do the right thing:

```
interpret :: Parser a -> String -> [a]
interpret p = interpret' (expand p)
    where
    interpret' :: [Parser a] -> String -> [a]
    interpret' ps []     = [a | Return a <- ps]
    interpret' ps (c:cs) = interpret'
        [p | (Symbol `Then` is) <- ps, p <- expand (is c)] cs
```

Namely, how to handle each of the summands depends on the input stream:

- If there are still input symbols to be consumed, then only the summands of the form `symbol >>= is` will proceed, the other parsers have ended prematurely.
- If the input stream is empty, then only the parsers of the form `return x` have parsed the input correctly, and their results are to be returned.

That's it, this is our breadth-first interpreter, obtained by using laws and equations to rewrite instruction lists. It is equivalent to Koen Claessen's implementation [2].

As an amusing last remark, I would like to mention that our calculations can be visualized as high school algebra if we ignore that `(>>=)` has to pass around variables, as shown in the following table:

Term	Mathematical operation
`return`	1
`(>>=)`	$\times$ multiplication
`mzero`	0
`mplus`	$+$ addition
`symbol`	x indeterminate

For example, our key idea corresponds to the distributive law

$$x \times a + x \times b = x \times (a + b)$$

and the monad and `MonadPlus` laws have well-known counterparts in algebra as well.

Conclusion

Further Examples

I hope I have managed to convincingly demonstrate the virtues of the operational viewpoint with my choice of examples.

There are many other advanced monads whose implementations also become clearer when approached this way, such as the list monad transformer [15] (where the naive `m [a]` is known not to work), Oleg Kiselyov's `LogicT` [16], Koen Claessen's poor man's concurrency monad [17], as well coroutines like Peter Thiemann's ingenious WASH [18] which includes a monad for tracking session state in a web server.

The `operational` package [6] includes a few of these examples.

Connection with the Continuation Monad

Traditionally, the **continuation monad transformer**

```
data Cont m a = Cont { runCont :: forall b. (a -> m b) -> m b }
```

has been used to implement these advanced monads. This is no accident; both approaches are capable of implementing any monad. In fact they are almost the same thing: the continuation monad is the refunctionalization [19] of instructions as functions

```
\k -> interpret (Instruction `Then` k)
```

But alas, I think that this unfortunate melange of instruction, interpreter and continuation does not explain or clarify what is going on; it is the algebraic data type `Program` that offers a clear notion of what a monad is and what it means to implement one. Hence, in my opinion, the algebraic data type should be the preferred way of presenting new monads and also of implementing them, at least before program optimizations.

Actually, `Program` is not a plain algebraic data type, it is a **generalized** algebraic data type. It seems to me that this is also the reason why the continuation monad has found more use, despite being conceptually more difficult: GADTs simply weren't available in Haskell. I believe that the `Program` type is a strong argument to include GADTs into a future Haskell standard.

Drawbacks

Compared to specialized implementations, like for example `s -> (a,s)` for the state monad, the operational approach is not entirely without drawbacks.

First, the given implementation of `(>>=)` has the same quadratic running time problem as `(++)` when used in a left-associative fashion. Fortunately, this can be ameliorated with a different (fancy) list data type; the `operational` library [6] implements one.

Second, and this cannot be ameliorated, we lose laziness. The state monad represented as `s -> (a,s)` can cope with some infinite programs like

```
evalState (sequence . repeat . State $ \s -> (s,s+1)) 0
```

whereas the list of instructions approach has no hope of ever handling that, since only the very last `Return` instruction can return values.

I think that this loss of laziness also makes value recursion [20] à la `MonadFix` very difficult.

About the author

After some initial programming experience in Pascal, Heinrich Apfelmus picked up Haskell and purely functional programming just at the dawn of the new millenium.

He has never looked back ever since, for he not only admires Haskell's mathematical elegance, but also its practicality in personal life. For instance, he was always too lazy to tie knots, but that has changed and he now accepts shoe laces instead of velcro.

References

[1] Sanjiva Prasad and S Arun-Kumar. An introduction to operational semantics (Apr 2003). http://www.cse.iitd.ernet.in/~sanjiva/opsem.ps.

[2] Koen Claessen. Parallel parsing processes. **Journal of Functional Programming**, 14(6):pages 741—757 (Nov 2004). http://www.cs.chalmers.se/~koen/pubs/entry-jfp04-parser.html.

[3] Chuan kai Lin. Programming monads operationally with Unimo. **ACM SIGPLAN Notices** (Jan 2006). http://web.cecs.pdx.edu/~cklin/papers/unimo-143.pdf.

[4] John Hughes. The design of a pretty-printing library. **Lecture Notes in Computer Science**, 925:pages 53–96 (1995). http://www.cs.chalmers.se/~rjmh/Papers/pretty.html.

[5] Ryan Ingram. Package 'MonadPrompt'. http://hackage.haskell.org/package/MonadPrompt.

[6] Heinrich Apfelmus. Package 'operational'. http://hackage.haskell.org/package/operational.

[7] Hackage, the repository for Haskell packages. http://hackage.haskell.org.

[8] Accompanying source code for 'The Operational Monad Tutorial'. http://code.haskell.org/~byorgey/TMR/Issue15/operational-code.zip.

[9] Generalized Algebraic Data Types. http://www.haskell.org/haskellwiki/GADT.

[10] The Glasgow Haskell Compiler. http://www.haskell.org/ghc/.

[11] Further information about Monoids. http://haskell.org/haskellwiki/Monoid.

[12] Henk-Jan van Tuyl. A tour of the haskell monad functions. http://members.chello.nl/hjgtuyl/tourdemonad.html.

[13] Martin Erwig and Steve Kollmansberger. Probabilistic functional programming in haskell. **Journal of Functional ...** (Jan 2005). http://web.engr.oregonstate.edu/~erwig/pfp/.

[14] Graham Hutton and Erik Meijer. Monadic parser combinators. **Journal of Functional Programming** (Jan 1996). http://www.cs.nott.ac.uk/~gmh/bib.html%23monparsing.

[15] http://www.haskell.org/haskellwiki/ListT_done_right.

[16] Oleg Kiselyov, Chung chieh Shan, Daniel Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). **ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming** (Sep 2005). http://okmij.org/ftp/papers/LogicT.pdf.

[17] Koen Claessen. A poor man's concurrency monad. **Journal of Functional Programming**, 9(3):pages 313–323 (Oct 1999). http://www.cs.chalmers.se/~koen/pubs/entry-jfp99-monad.html.

[18] Peter Thiemann. An embedded domain-specific language for type-safe server-side web scripting. **ACM Transactions on Internet Technology (TOIT)** (Jan 2005). http://www.informatik.uni-freiburg.de/~thiemann/WASH/draft.pdf.

[19] Oliver Danvy and K Millikin. Refunctionalization at work. **Science of Computer Programming** (Jan 2009). http://www.brics.dk/RS/07/7/BRICS-RS-07-7.pdf.

[20] Levent Erkök. Value recursion in monadic computations (Oct 2002). http://citeseer.ist.psu.edu/590305.

Implementing STM in pure Haskell

by Andrew Coppin ⟨MathematicalOrchid@hotmail.com⟩

GHC features Software Transactional Memory (STM). STM is a thread coordination/synchronisation technique based on the use of database-style transactions to access shared program variables. The implementation is currently hard-wired into the GHC runtime system (RTS), but in this article, I describe an implementation of STM in pure Haskell. This shows that STM can be implemented fairly easily in any programming language which provides threads, locks, and the ability to kill threads.

The STM Interface

Before we can think of implementing STM, we need a clear understanding of exactly what it's supposed to do.

GHC offers several abstractions for multi-core programming – sparks, threads, parallel arrays, and so forth. STM is a method for explicit concurrency (*i.e.* explicit threads created using `forkIO`). Essentially it works by putting operations which mutate shared variables into database-style atomic **transactions**. While databases have used transactions for decades, most programming languages handle concurrent programming using **locks**.

An excellent survey of why locks are bad and why STM is good can be found in the paper "Beautiful Concurrency", and a simpler user-level explanation of how it is used can be found in "Composable Memory Transactions". Both of these are linked from the following web page:

```
http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/
```

Listing 29 shows the interface provided by the STM package. (I've done some trivial renaming here.) We have a `TVar` type, which represents shared mutable variables, and a `Transaction` monad for reading and writing these variables. We

```
data Transaction x
instance Monad Transaction

runT :: Transaction x -> IO x
retry :: Transaction x
orElse :: Transaction x -> Transaction x -> Transaction x

data TVar x
newTVar :: x -> Transaction (TVar x)
readTVar :: TVar x -> Transaction x
writeTVar :: TVar x -> x -> Transaction ()
```

Listing 29: The STM programming interface

also have `runT` ("run transaction") for running a transaction. (This, unsurprisingly, is an I/O operation living in the `IO` monad.)

STM isn't exactly like database transactions. In particular, it provides the ability to block, waiting for one of several conditions to change (via the `retry` function). It also provides for trying multiple alternative execution paths (via the `orElse` function). These somewhat unusual features turn out to be the hardest part to implement.

The GHC implementation

Managing consistency

The `runT` ("run transaction") function takes an operation in the `Transaction` monad, runs it and returns its result. It is guaranteed that the transaction sees a consistent image of the system's state, and this implies that any write operations that the transaction performs must be atomically applied to the system (so that other transactions see only the before-state or the after-state, but never a mix of the two).

There are several possible ways to implement this behaviour. (This is one of the nice features of STM, in fact.) One way would be to have a Giant Lock, and have each transaction obtain this lock before finalising its writes to the system. This is very simple to implement, and it is horrifyingly inefficient. Even non-conflicting transactions would be blocked from executing concurrently.

The approach taken by GHC is **optimistic locking**, a technique borrowed straight out of the database literature. It works in the following way:

- Each transaction has a **transaction log**, which records `TVar` reads/writes.
- Read operations physically read `TVar`s (without taking any locks), and log the value read.
- Write operations record events in the log, but perform no actual writing.
- At transaction completion, the transaction is **validated** and then **committed**.
- Validation checks that the data in the read log still matches the data in the `TVar`s.
- If validation passes, the contents of the write log are (atomically) committed to the `TVar`s.
- If validation fails, the transaction is rolled back (which just involves deleting the log), and restarted from the beginning.

So the transaction may in fact be run and rerun an unbounded number of times until validation succeeds.

Suppose transaction A reads `TVar` X, and then transaction B (atomically) updates `TVar`s X and Y, and then transaction A reads `TVar` Y. Now transaction A has seen an inconsistent state of the system. But it matters not; validation will fail for transaction A, because the data it read from `TVar` X no longer matches the current contents of X. So transaction A will be rolled back, restarted, and (assuming it gets a consistent image this time) committed.

Blocking & choice

The `retry` and `orElse` functions complicate this simple picture somewhat.

The `retry` function is supposed to abort and restart the transaction. While you **could** implement this so the transaction restarts immediately, this is probably a waste of time. What you really want to do is restart the transaction once something has **changed**. (Otherwise the transaction will perform exactly the same actions – in other words, it will just `retry` again.)

You could implement `retry` to wait for any change to any `TVar`, but a better option is available: let `retry` wait for a change to one of the `TVar`s that the transaction has actually inspected so far. This information is conveniently recorded in the transaction log. So `retry` waits for one of these `TVar`s to be committed to, and then deletes the transaction log and restarts the transaction from the beginning.

Adding `orElse` complicates things still further. Now instead of `retry` performing a rollback and restart of the whole transaction, it merely rolls back the current **sub**-transaction and passes control back to `orElse`. (Unless `orElse` hasn't been called at all, in which case the behaviour is as before.)

The current GHC implementation of STM is hard-wired into the RTS. All of the actions I have described above are achieved by such techniques as placing markers on the Haskell lightweight thread stack, unwinding the stack to the next marker, and so forth. It gets complicated; consult the GHC developer wiki for the messy details.

Edge cases

The above description may seem complete, but STM turns out to be much more tricky than you'd imagine. For example, what happens if a transaction throws an exception?

The basic property of STM is this:

Theorem 1 (STM property)**.** *Only valid transactions commit and return.*

The GHC implementation of STM achieves this in the following way:

Theorem 2 (GHC STM invariant)**.** *The validity of a transaction is only guaranteed to be checked at the very end of the transaction.*

An important consequence of this is:

Corollary 3. *A running transaction may already be invalid.*

That being the case, a transaction might throw an exception **because** it saw an inconsistent state of the system. For this reason, when an exception is thrown, the transaction must be validated. If valid, the exception is propagated to user code. If invalid, the exception is silently ignored, and the transaction rolled back and restarted (since it is invalid). Note carefully that propagating an exception from an invalid transaction could potentially carry inconsistent data out of the transaction.

But wait, there is worse! A transaction could conceivably go into an **infinite loop** because of inconsistent data. This might eventually cause a stack overflow exception or similar to be thrown, but if the loop performs no allocation, the thread would just end up stuck in a loop forever.

We cannot simply wait until the end of the transaction to check its validity. The GHC developers neatly solve this problem by having the thread scheduler perform a validation check each time a new thread (running a transaction) is about to be scheduled to run. (One presumes the developers have considered the case of a single running thread and hence no thread rescheduling...)

My implementation

Development notes

Developing my STM library proved to be quite a bit harder than I anticipated. Although the final result is fairly simple, the process of obtaining it was not. I stumbled down dead-end routes and investigated techniques which turned out not to fit. Most of all, when I started coding, I lacked a clear idea of how the system I was building was going to work. By a process of building different designs and watching them fail, I eventually arrived at a design which works.

I spent a lot of time coding. I spent a lot of time throwing together hackish solutions. But Haskell is a language which discourages hacks and workarounds, and strongly encourages you to **stop**, **think** and **design** before proceeding. Eventually, after discarding many ugly and broken implementations, I arrived at a solution which was both beautiful and (as far as I can tell) correct.

This is why I love Haskell. Once you figure out your high-level design, the rest is fairly easy.

Transaction consistency

OK, so we need a `Transaction` monad, and we need `TVar`s. Initially a `TVar` is going to be a simple `IORef`, and clearly `Transaction` is going to be some kind of `newtype` over the `IO` monad. Now how the heck are we going to guarantee that only valid transactions commit/return?

Building an STM system based on a Giant Lock is fairly simple; I leave this as an exercise for the over-interested reader. For the rest of this article, I will assume that we're aiming for an optimistic locking technique similar to GHC. (The other technique used by databases such as Oracle is multi-versioning, but that's probably not appropriate for thread coordination.)

As I have explained, transactions which see inconsistent data can end up in an infinite loop, so optimistic locking implies a requirement to be able to terminate running transactions. GHC's STM implementation is currently hard-wired into the Haskell runtime system, and so has the power to manipulate running Haskell threads in arbitrarily crazy ways. I, on the other hand, have only `Control.Concurrent` to go on.

An early design choice was that each transaction runs in a separate thread, so that `killThread` can be used to terminate it. That is, `runT` does no actual work itself, it merely spawns a separate thread which actually runs the transaction. This child thread returns its data to `runT` through an initially-empty `MVar`. In this manner, the running thread can be killed and a new one spawned (provided you have the transaction code for it to run and the `MVar` for it to return through).

Originally the commit functionality was in `runT`, but I later realised that the correct place for this is in the child thread. `runT` calls a function named `tmain`, which allocates a new transaction log, runs the transaction body, and performs the commit operations before finally sending the result back to `runT` through the provided `MVar`. (See Figure 1.)

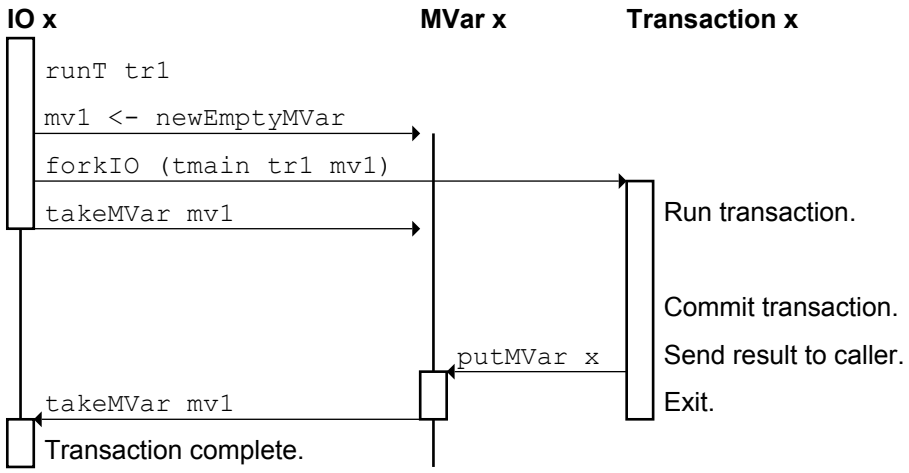

Figure 1: The interaction of `runT` and `tmain`.

I solve the infinite-loop problem by neatly side-stepping it. Recall GHC's STM invariant. Mine is different:

Theorem 4 (AC STM invariant)**.** *Every running transaction is valid.*

The way to achieve this is both simple and easy:

Corollary 5. *Every invalid transaction is terminated immediately.*

Put simply, when a write is committed to a `TVar`, any and all transactions that have read that `TVar` are terminated using `killThread`.

By this simple mechanism, I eliminate any need for a "validate" process. If a thread is still running by the time it reaches the commit phase, it is by definition still valid. In the case of only a single transaction running, no checking of any kind needs to happen. On the other hand, event handlers must be registered and then unregistered. Also, threads must sometimes be killed, and a new thread is spawned for every (retry of a) transaction, which possibly has more overhead than what the RTS STM code is doing.

A first pass

I started out with

```
newtype Transaction x = Tr (TLog -> IO x)
```

where `TLog` is the transaction log type. Essentially, this pumps the transaction log (which is mutable) to every action in the transaction. Thus, for example, `writeTVar` adds an entry to the log by mutating it.

An invariant is that only one thread can mutate the log at any one time. Usually this will be the transaction thread; other threads are only permitted to perform mutations if the transaction thread has been terminated.

A `TVar` is essentially an `IORef`, plus a (mutable) structure for holding "event handlers". When data is committed to the `TVar`, the thread performing the commit "fires" all the events in the event list. This terminates and restarts the transactions invalidated by this commit. (One of the nice things about Haskell is that I can store an `IO ()` which just needs to be run, and this closure "knows" which thread to kill and what code to run in the new thread it spawns. First-class actions are great!)

The complete system then looks something like this:

- `runT`: Create `MVar x` and pass it and the transaction body to `tmain`. Wait on the `MVar`, then return its contents to the caller.
- `tmain`: Create `TLog`, execute transaction body, commit writes, put result into `MVar`.
- `writeTVar`: Record write in the log.

The tricky part is `readTVar`. It does the following:

- Check the write log. If we've already written to this `TVar`, return that value.
- Check the read log. If we haven't read from this `TVar` yet, log the read and install the invalidate event handler.
- Read from the underlying `IORef` inside the `TVar` and return the result.

Note that the read log doesn't actually need to record the value read. It just records which `TVar`s we've read from. Note also that the invalidate handler must be installed **before** attempting to perform the physical read, so that transaction termination is guaranteed if invalidation occurs.

Tricky commit problems

Committing writes to `TVar`s is more tricky than you might imagine. The writes need to appear atomic, and there are a number of problems we need to sort out to get this to work.

First of all, how do we stop two transactions committing writes to the same bunch of `TVar`s at the same time? If we end up with one image or the other that's

fine, but if we have half of each our atomicity guarantee is broken. My solution is to attach a write-lock (which is just an `MVar ()`) to each `TVar`. The transaction takes write-locks on **all** `TVar`s to be committed to before any data is actually committed (and releases them afterwards).

Now that we're locking things, the next problem is how to prevent deadlock. I use the standard solution of imposing an arbitrary total order on all the locks in the system and taking locks in ascending order. Thus, to each `TVar` I add a unique ID number and an `Ord` instance for this purpose.

The next problem is that a transaction thread could be killed while in the middle of trying to commit. (This happens if another thread happens to invalidate the transaction at just the wrong moment.) If we have not yet taken all locks, we want to rollback as normal – but to do this, the locks must be released. I use a `finally` block to do this. (GHC kills threads by throwing a `ThreadKilled` exception, which `finally` can catch and handle.)

A nasty potential bug here is releasing the locks using `putMVar l ()`. This works fine in the normal case that the `finally` block completes normally and the locks are released at the end, but if the thread **is** terminated before all locks have been taken, `putMVar` will block on the locks not yet acquired, causing an instant deadlock. (And since this code path is rarely taken, this bug would be very intermittent and extremely hard to track down.) The correct procedure is to use `tryPutMVar l ()` instead.

Once all locks are taken, the thread reaches the point of no return. The event handlers are unregistered and the writes are committed. But one of the event handlers could fire (terminating the thread) before we can unregister it. I use the `block` function to temporarily disable asynchronous exceptions (such as `ThreadKilled`) while the event unregister and write commit functions run. See Figure 2 for an overview.

(Note that `killThread` apparently blocks until the `KillThread` exception is delivered. Since `tmain` calls exits `block` once all critical functions have been completed, the calling thread should only be blocked momentarily.)

Now two transactions trying to commit to the same `TVar`s will be forced to serialise, two transactions committing to unrelated `TVar`s may proceed concurrently, and once a commit operation has all the necessary write locks, it is guaranteed to complete (barring catastrophic program termination).

Blocking

At this point, we have a library which can atomically run transactions. But we haven't implemented blocking yet.

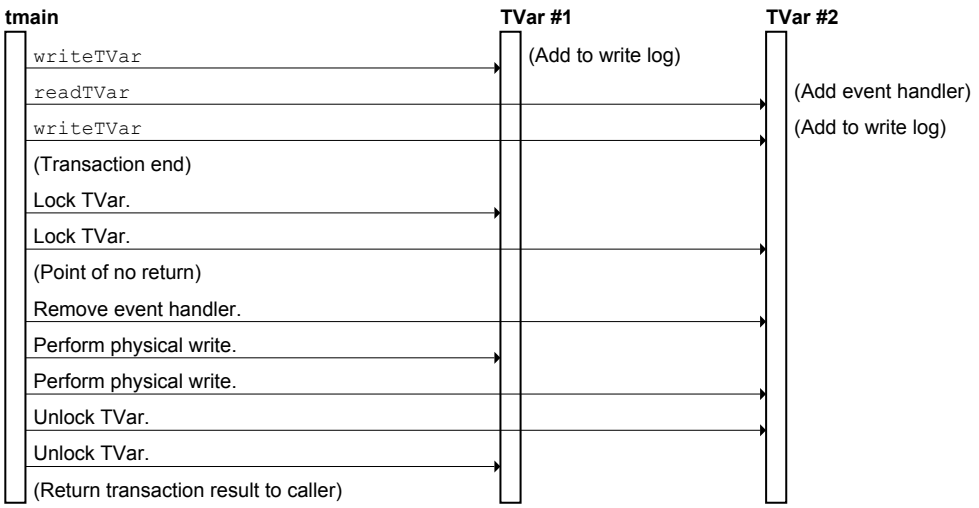

Figure 2: The commit procedure: Lock everything in the write log, disable thread termination, release all event handlers in the read log, physically perform all writes in the write log, release all the locks, and return to the caller.

Satisfyingly, the implementation of `retry` turns out to be utterly trivial:

```
t <- myThreadId
killThread t
undefined
```

The `retry` function simply causes the current thread to commit suicide! (The final line exists to abuse the type checker. It is actually unreachable.)

Recall that `retry` is supposed to wait for something to change, and then restart the transaction. Recall also that `readTVar` registers an event handler which will kill the transaction thread, unregister all event handlers, and then restart the transaction in a new thread.

In other words, if we just terminate the current thread, then when something relevant changes, the existing event handlers will kill the thread (which is already dead anyway), unregister its event handlers, and spawn a new thread, starting the transaction from the beginning again – exactly what we want!

Choice

Implementing the `orElse` function turned out to be the single hardest task I came up against. This function changes the behaviour of `retry`. I investigated all kinds of weird and wild techniques for implementing it. I started out fumbling around

with a mutable variable for storing a callback function, I looked at using custom exceptions, I attempted to comprehend call/CC and the `Cont` monad... but in the end, the answer is far simpler.

What we want is to run action X, and if that fails run action Y instead. Parsec manages to do this with parsers. And in fact `Data.Maybe` exports a function named `<|>` which does this exact thing with `Maybe` values. And that turns out to be the key: We represent a transaction not as an I/O action that returns a value, but as an I/O action which may fail:

```
newtype Transaction x = Tr (TLog -> IO (Maybe x))
```

The `>>=` function now runs the chained action, checks whether it returns `Nothing`, and if so short-circuits all further computation. The `retry` function becomes simply `return Nothing`. And the top-level `tmain` function checks the result; if it's `Nothing`, it performs the action that `retry` used to perform – namely, terminate the thread and wait for an event handler to spawn a new one.

With this change in place, `orElse` becomes delightfully trivial to implement. No call/CC, no weird exception trickery, just run each sub-transaction and check its result. The only tricky detail is the transaction log; if the left sub-transaction fails, the **write log** must be reset, but the **read log** and any event handlers registered must remain unchanged. (If somebody invalidates something looked at in the left branch, we want to restart; maybe that branch won't `retry` now.)

Further work

My completed library consists of less than 20 KB of Haskell source code, and depends only on `IORef`s, `TVar`s and a few functions such as `forkIO`, `myThreadId` and `killThread`. I still have two main tasks to complete:

- Confirm that the library works correctly.
- Determine how performant it is.

Neither of these is especially easy when dealing with a library whose fundamental purpose is to manage non-deterministic behaviour.

Those of you who have been paying attention will have noticed a small gap: exception handling. Recall that exceptions in invalid transactions **should** be silently ignored, and exceptions in valid transactions are supposed to propagate to the caller of `runT`. Currently this is not the case. It appears that exceptions get printed to `stderr`. Exceptions in an invalid transaction are almost impossible (an invalid transaction thread gets killed **almost** instantly), but are harmless; the transaction will be restarted promptly. Exceptions in valid transactions simply vanish into the luminiferous aether – which is incorrect.

There is also a small issue surrounding `killThread`. I vaguely recall reading somewhere that asynchronous exceptions (such as `ThreadKilled`) can only be raised at certain points in the execution of a thread. This suggests the possibility that a thread in a tight loop may be unkillable.

About the author

Andrew Coppin is a woefully under-employed computer science graduate living in Milton Keynes, UK. His interests include the inevitable eclectic cross-section of mathematics, science and computer-gaming, as well as more unusual pursuits such as caving, rock climbing, skiing, dancing and playing pipe organs.