Functional Event Loops and Extension APIs

By Chris Penner

April 19, 2017

Contents

Abstract

Pure functional programming provides many benefits but is often criticized as being difficult to use for highly interactive or state-driven applications. We explore benefits of functional programming in application design in the context of event loops and API design. We seek to show that though functional programming is perhaps less intuitive for traditional imperative programmers, functional paradigms are adequately equipped to handle the problems presented by interactive programs, and can often provide additional benefits over their imperative counterparts.

Motivation

Pure functional languages excel at declarative programming. Programmers describe a system through which input can flow to produce output, just like functions in traditional mathematics. This is ideal for computing solutions to purely mathematic problems, however users would also like to use these languages for programming games, graphical systems, and interactive applications which do not obviously conform to this model. Modern programmers have (perhaps unfortunately) grown accustomed to thinking of these systems as a series of stateful mutations, this intuition is stretched when utilizing a functional language, but it can still serve as a useful framework for understanding complex systems. This paper approaches two aspects of application design, namely event-loops and API design from a stance which will be familiar to imperative programmers while gaining several benefits and simplifications of typed functional programming.

Designing a Functional Event Loop

Imperative Event Systems

Event loops are a common fixture in imperative programs, especially in highly interactive applications. They are a helpful architecture for separating concerns and they provide concrete extension points for your application.

The basic idea in event-loops is that extensions to the system add 'listeners' with accompanying callback functions to be run in response to the occurence of a certain event. When the event is triggered each of the registered callbacks runs and the system proceeds. This event system has been employed to great success in browsers. Clicking a button on a web page triggers an event which runs callback handlers listening for that event.

The architecture is found in many other systems including operating systems, games, and the web. It is relatively robust with a simple mental model which

many programmers are familiar with. At first glance it seems to be a model grounded firmly in imperative mutation-based programming. Callbacks typically have a 'void' return type as they are called asynchronously at some stage in the event loop which would not know where to route a return value. By necessity the callback must enact changes on the system by mutating global state or some object which the callback has a reference to. This is commonplace in imperative languages (especially so in Javascript), however this poses a real problem for pure languages which disallow any form of side-effects. A pure function which returns 'void' must have absolutely no effect on the system, if we wish to emulate the callback infrastructure we will clearly need to make some alterations.

Reclaiming Callbacks

Think about what a callback represents, in its essence it is a description of a computation which takes the current (global) state of the system, runs some procedure on it and results in some altered state of the system. The issue we have run into is that the state of the system is not available to us implicitly inside a function call and we cannot mutate it by passing a reference via closure because pure functional languages deal with immutable data structures. We will need to make operations over the system's state explicit. Luckily a well explored solution for this exists, we call upon the State Monad. The State Monad sequences state alterations and ultimately yields a description of which alterations should be performed and in which order. Many functional programming languages have some form of syntactic-sugar which makes working within a monadic context quite convenient, this will emulate the implicit state of an imperative program while remaining explicit in intention by the type of the monadic expression. The reader should familiarize themselves with the State monad, but we show the type of the State Monad in Haskell syntax here:

```
data State s a = State (s -> (a, s))
```

Do not be alarmed if this fails to clarify anything, the idea is that a state alteration can be expressed as a function which takes the current state of type '`s`' and returns the next state alongside some result of type '`a`' if desired. If we choose State Monads as the interface for our event listener callbacks we get the following signature for simple callback:

```
callback :: Event -> State s ()
```

We fix the type of '`a`' to `()` (pronounced 'Unit') denoting that we are not returning any useful value. This forms the functional equivalent of a 'void' function, it describes a state mutation which returns no other value. If we expand State monad to its underlying type we see that this is actually a higher order function which takes some Event and returns a function over the state.

```
callback :: Event -> (s -> ((), s)
```

We may have many such callbacks which respond to the same event listener. Upon receiving a matching event we must run each callback in sequence; to achieve this we partially apply each callback with the event (via currying) and combine the resulting State monads together using bind/flatMap. To simplify this task we may use `mapM` provided by the Haskell Prelude, since we currently have no need for the return value (which is just `()`) we use the `mapM_` variant which discards any return values. When specialized to our situation `mapM_` has the following type:

```
-- An EventListener takes an Event and returns a State Monad expression
type EventListener = Event -> State s ()
mapM_ :: (EventListener -> State s ()) -> [EventListener] -> State s ()
```

This tells us that if we can give `mapM_` a function which takes an EventListener and produces a `State s ()` from it then we can squash a whole list of EventListeners down into a single state expression! We know that to go from an `EventListener` to a `State s ()` we need to apply the listener to an `Event`. We can use flipped function application to fill this slot by applying the dispatched event to each listener, getting the state monad that we needed. We will wrap this logic up into a new `dispatchEvent` function which takes a list of event listeners and an event and returns the combined state monad.

```
dispatchEvent :: [EventListener] -> Event -> State s ()
dispatchEvent listeners event = mapM_ applyEvent listeners
    where applyEvent listener = listener event
```

We have successfully sequenced our list of state transformations and can dispatch events to get back a description of the state alterations which they would like to perform. Remember that the State Monad does not actually enact any of the alterations until we choose to **run** it over our application state, this is the primary behaviour provided by the state monad, consult the appendix for clarification on running state monads.

Extension Queries

Synchronous Callbacks

Interactive applications often provide the event API as a means of extensibility. Extensions may be executing alongside the main application and the primary execution path of the application may not be aware of augmentations made by extensions. It is useful to have a mechanism for the application to query extensions for information. Unfortunately it is not possible to call functions exported by the extension directly since the base application cannot know at compile time which extensions might be enabled. Though perhaps unintuitive, we may abuse the event system to serve this purpose. When we wish to 'ask' registered extensions for some piece of information we may query them by

dispatching a particular type of event which extensions have registered listeners for. When this special event is fired the registered listeners compute their response and return it from the callback. This *almost* works with the described event system, however since we are ignoring return values extensions must store their responses somewhere in the system's state which is unsafe, unreliable, and needlessly complicated. With a small alteration we gain a simplified interface for this use-case.

So far we been using `()` as the return value from State Monad expressions. We had made the assumption that the callbacks for events would be executed at an arbitrary time, as it is in Javascript, and that any return value would be inaccessible to whoever dispatched or caused the event. Though certainly Javascript has reasons for its design, we are our own masters and may choose to change the structure of the system to run all matching event listeners immediately when an event is dispatched. This makes any return values from the listeners available to the event dispatcher. It may appear that by executing event listeners synchronously at dispatch time would hamper concurrency in some way, however it is clear that state alterations must be executed sequentially in order to avoid inconsistencies. It is similar within Javascript's event loop; multiple event listeners may not be executed concurrently since each listener may mutate the same piece of global state. Javascript's solution is to simply delay execution of callbacks until the currently executing 'task' returns. This results in *asynchronous*, but not *concurrent* execution. We choose to shift this to *synchronous* in-line execution.

Certain actions, such as HTTP calls, benefit inherently from asynchronous execution, a modification to support such asynchronous tasks is perfectly feasible in the functional event-loop and indeed one is implemented by the Eve framework upon which this paper is based, but it is out of scope for our discussion here, refer to Eve's documentation or source code for more details (Penner 2017).

So by choosing to execute listeners synchronously we lose nothing, we must only recognize the shift in semantics which occurs and be aware that dispatching an event will enact any mutations specified by listeners immediately. This approach is perhaps slightly less intuitive, but is actually more powerful as the component which dispatched the event may address changes caused by listeners if it so desires. If behaviour similar to Javascript is desired, the user need only dispatch events at the end of the currently executing block, thus avoiding any effects during the block's execution.

Returning Results

Now that we have synchronous execution of event callbacks there is no barrier preventing us from receiving and handling return-values. First we alter our implementation of `dispatchEvent` to use `mapM` which returns its result, then change the type of the return value to a more generic `result` in place of `()`:

```
type EventListener = Event -> State s result
mapM :: (EventListener -> State s result) -> [EventListener] -> State s [result]

dispatchEvent :: [EventListener] -> Event -> State s [result]
dispatchEvent listeners event = mapM applyEvent listeners
    where applyEvent listener = listener event
```

Notice that the return value from the resulting State Monad is a list of results since we receive a result from each of our (possibly many) listeners. This is quite useful in practice, we may query extensions for information and receive back a list of their responses, however there is room for a generalization here which serves to improve the power of our system and also simplify its implementation.

Employing Monoids

`mapM` is combining all of the listeners into a single State Monad while simultaneously aggregating the results into a list. Typically wherever an aggregation is taking place a Monoid may be employed, and indeed this is the case here. If we require our result type to be a Monoid we may reform our implementation like so:

```
dispatchEvent :: Monoid result =>
                     [Event -> State s result] ->
                     Event ->
                     State s result
dispatchEvent listeners event = (mconcat listeners) event

-- or in embarrassingly short point-free form:
dispatchEvent = mconcat
```

This new implementation is much cleaner, yet requires some explanation. If you are unfamiliar with Monoids it is beneficial to take some time to gain an appreciation for them as they are of great significance to the sections which follow; reference the appendix for some learning resources. The simple explanation is that a Monoid represents a Group as in abstract algebra, but without the constraint that inverses are defined. This means that for every Monoid we know that there is an associative combination operator, namely `mappend`, and there is an identity element `mempty`.

Consider the key portion of the previous code block: `mconcat listeners`. `mconcat` is a method provided by `Data.Monoid` in haskell which reduces a list of Monoids down to a single element by iteratively `mappend`ing all of its elements. Notice how we are no longer mapping over the list before we collapse it, we can `mconcat` the `Event -> State s result` types together 'as-is'. This is made possible because this type is itself a Monoid! It is not immediately clear where the Monoid instance comes from, since we only have a Monoid constraint on only `result`, we will break down the instance for clarity.

Ordinary Haskell functions have a Monoid instance defined if and only if the return value of the function is a Monoid; here is the implementation from `Data.Monoid`:

```
instance Monoid b => Monoid (a -> b) where
        mempty _ = mempty
        mappend f g x = f x `mappend` g x
```

When two functions f and g of the same type are combined with `mappend` the result is a function which accepts an input and applies it to each of f and g then uses `mappend` to combine the outputs together. This requires the output of the function to be a Monoid. In our case the function is of type `Event -> State s result`, so the return value is `State s result`, which is also a Monoid by definition that follows.

A Monoid instance for the State Monad is not defined in `Control.Monad.State`, however we can implement one without much difficulty:

```
instance (Monoid result) => Monoid (State s result) where
    mempty = return mempty
    a `mappend` b = do
      resultA <- a
      resultB <- b
      return $ resultA `mappend` resultB
```

The keen observer may note that this instance makes no mention of anything specific to State Monad itself, and indeed this instance is general enough to make ***any*** monad with a Monoidal return value into a Monoid. In practice this instance is not provided in the Haskell Prelude because not all such instances are sensible. Reference Gabriel Gonzales' conference talk (2017) on the subject for a more thorough explanation. Thus, `State s result` is a Monoid if `result` is a Monoid, we may require this as a constraint on all event callbacks. This is no great burden, if a return value isn't needed the listeners may simply take the form `State s ()` since `()` is the trivial unit Monoid consisting of only the identity element. If a list of results is desired (as we had before), the listeners may take the form `State s [result]` and the results will be concatenated into a list. To provide a simpler API one may wish to provide a helper function which converts `State s result` into `State s [result]` before registering the listener by first applying `fmap (:[])`.

We have now constructed the required machinery to dispatch events, execute the relevant event callbacks and even to query extensions for information. The following is an example demonstrating use of the system when handling the needs of a rather strange office building. In the Eve framework listeners are stored and registered within the state of the application thus avoiding the need to pass them to `dispatchEvent` manually.

```
{-# language ExistentialQuantification #-}
{-# language FlexibleInstances #-}
```

```haskell
{-# OPTIONS_GHC -fno-warn-orphans #-}
module Main where

import Control.Monad.State
import Data.Monoid

instance (Monoid result) => Monoid (State s result) where
  mempty = return mempty
  a `mappend` b = do
    resultA <- a
    resultB <- b
    return $ resultA `mappend` resultB

  -- An event signalling a form submission with a text payload
data Submission = Submission String

  -- A query event which will check whether any extensions require a feature
data QRequiresLightbulbs = QRequiresLightbulbs

-- Our simple state will just hold a list of submissions
data ApplicationState = ApplicationState [String] deriving Show

-- This simple listener appends any submitted text to the submission list
collectSubmission :: Submission -> State ApplicationState ()
collectSubmission (Submission payload) = do
  ApplicationState submissions <- get
  put (ApplicationState (payload:submissions))

-- This listener would be registered by an extension, allowing the main
-- application to ask extensions about their state.
-- In this case we don't require any lightbulbs, but since we've chosen
-- 'Any' to be the result Monoid the result will be True if ANY registered
-- listener specifies that they need lightbulbs.
requiresLightbulbs :: QRequiresLightbulbs -> State ApplicationState Any
requiresLightbulbs _ = return (Any False)

-- This dispatches any event over a matching list of listeners and
-- returns the result aggregation
dispatchEvent :: forall event result. Monoid result =>
    [event -> State ApplicationState result] ->
    event ->
    State ApplicationState result
dispatchEvent listeners event = mconcat listeners event

simpleAction :: State ApplicationState ()
simpleAction = do
```

```
  dispatchEvent [collectSubmission] (Submission "12 Green Staplers")
  dispatchEvent [collectSubmission] (Submission "6 Purple Paperclips")
  Any needLightbulbs <- dispatchEvent [requiresLightbulbs] QRequiresLightbulbs
  when needLightbulbs $
    dispatchEvent [collectSubmission] (Submission "18 Infrared Bulbs")
```

When we can then execute the `simpleAction` over an `ApplicationState` object. The events are triggered, the listeners fire, and we get the resulting state out:

```
execState simpleAction (ApplicationState [])
-- > ApplicationState ["6 Purple Paperclips","12 Green Staplers"]
```

Which is what we expect since we don't need any lightbulbs in this particular case.

Building Extension DSLs using the Free Monad

Pre-requisites

This section assumes a slightly deeper understanding of State Monads than the previous section. It will be sufficient to understand the use of `get`, `put`, and `modify`, which get, set and modify the state of the State Monad respectively.

The `mtl` monad transformer library is used shortly, the reader need only know that monad transformers provide a means to 'mix' the effects of several monads, employed in this case to allow the use of IO inside of a State Monad. See the appendix for suggested reading.

Defining the Extension API

In the previous section we devised a system wherein extensions may register callbacks to be run in response to events. To perform any useful actions in these callbacks we must expose a way for them to alter the state of the application itself (and not just their localized state). In traditional imperative extension APIs a set of operations are chosen to be made available to extension implementers which may then be called at will by extensions. In fact once the functions are exposed, control lies entirely in the hands of the caller. We shall first demonstrate how we can emulate this approach in a functional style, and then shall improve upon it by employing the Free Monad to return control to the application and add additional features and optimizations.

Let us first consider the simplest approach of exposing functionality to consumers. We observe a very simplified subset of operations one could conceive to see in a text-editor's API as an example. Since we have implemented our event system in terms of the State Monad these actions are State Monads which act on the editor's state, which in our example shall be represented naively as a simple string. We define the following actions `getText` and `setText` to be our external API.

```
type EditorText = String

getText :: State EditorText String
getText = get

setText :: String -> State EditorText ()
setText text = put text
```

This is quite reasonable and should allow clever extension writers to implement their desired functionality. Here is an action which capitalizes the editor text. For example the following extension is implemented using this API.

```haskell
capitalize :: State EditorText ()
capitalize = do
  text <- getText
  -- Uppercase the text by mapping over each character with toUpper
  setText $ fmap toUpper text
```

Haskell's do-notation provides us with a readable way to express our monadic actions concisely. At this point we have achieved parity with the imperative method, or perhaps have even surpassed it by way of providing a nice DSL-style notation to use while still maintaining the full expressive power of haskell.

Problems with the Traditional Approach

As we work further on our text editor we realize that extensions may wish to trigger a reaction when text is edited, and so we decide to dispatch an event each time `setText` is called. We cleverly alter our implementation of `setText` to utilize the event system we designed; assume for simplicity that a list of event listeners is maintained inside the editor state somewhere.

```haskell
-- A simple event type which signals that the text has been changed, and what
-- the new value of the text is.
data TextChanged = TextChanged String

setText :: String -> State EditorText ()
setText text = do
  put $ EditorText text
  dispatchEvent $ TextChanged text
```

We are pleased that extensions may now respond each time that the text is changed by registering a callback for the `TextChanged` event. Editor development continues, and later we realize that it would improve efficiency to provide the ability to append to the editor's text rather than always overwriting.

```haskell
appendText :: String -> State EditorText ()
appendText newText = modify (++ newText)
```

We have successfully implemented `appendText`, however there's a problem; we have forgotten to use the `setText` primitive we previously defined since it was inconvenient and more verbose to do so. If we expose this implementation extensions will not be notified when text is appended. We see that this approach is fragile and requires great discipline, it is easy for engineers to circumvent assumptions, accidentally avoiding routines for logging or event handling. Perhaps even accidentally circumventing carefully implemented performance optimizations. We consider one more failure case before proceeding to the proposed solution.

We next decide to add a new action to the API: the ability to save the text to a file.

```haskell
save :: String -> StateT EditorText IO ()
save filename = do
  text <- getText
  liftIO $ writeFile filename text
```

This is easily implemented, but now that we are interacting with the file-system (which is potentially impure) Haskell requires that we use the IO monad. We need both the State Monad and IO monad now so we employ monad-transformers to accomplish this task. Monad transformers allow us to compose monads into monad 'stacks' which allow intermingling of the effects of each monad. This is a well established approach and allows us to write the `save` function, however we have now changed the type of the underlying monad for all of our actions! In order to use them in combination with each other we are required to change the type of all other actions in our API to be of the type `StateT EditorText IO`. This is already an annoyance, but there are further implications. The monad stack now contains IO which means that extensions may use `liftIO` inside their own actions to perform arbitrary IO. For certain applications this may represent a significant security risk; an extension could delete important files on the user's computer or upload information to the internet. Impure languages unfortunately have no choice but to accept the possibility that an extension they have installed could delete their file-system at will, but we seek to provide stronger guarantees to our users, in large part so that they will continue to be users in the future. Functional elitism aside, we would prefer to prevent extensions from performing any IO directly and instead be restricted to only the actions provided by the API.

What we have learned from the previous examples is that we require a robust way to provide our API wherein we have the ability to control the extension's access to our system and the user's computer. As an additional benefit we would prefer to provide clear extension points onto which concerns such as logging, event-handling etc. may be affixed. I posit that Free Monads provide a clean solution.

Encountering Free Monads

What is a Free Monad? A Free Monad is a Monad which you get 'for free'; it allows us to promote to a Monad a set of primitives encoded as an algebraic datatype in such a way that do-notation and monadic combinators may be used to interact with and programmatically sequence members of the datatype without requiring commitment to any concrete implementation of the primitives' behaviour. It effectively allows us to build up an abstract syntax tree which represents the computation that an extension wishes to perform but without executing any code in the process.

The (unenlightening) definition of the Free datatype is as follows:

```haskell
data Free f result =
```

```
  Free (f (Free f result))
  | Pure result
```

Values in the free monad may take one of two forms, they may represent a pure value without any computation as a `Pure` value, or they represent a link in the chain of our sequential computation. Each link in the chain is also responsible for determining how to compute the next link in the chain or to otherwise return a result. With this we have the power to model arbitrary computation in terms of the primitives we define. The `f` in `Free f result` represents the algebraic datatype which models the primitive operations we wish to allow. The `f` in this case stands for `Functor`, which means that the datatype we wrap in Free must have a 'slot' which can contain an arbitrary type and that there is an `fmap` function defined over the datatype which maps over the values of the contained type. If we have a Functor `f` with such a slot `a` we have a Monad over `Free f a` as follows:

```
instance Functor f => Monad (Free f) where
  return = Pure
  Pure a >>= func = func a
  Free m >>= func = Free ((>>= func) <$> m)
```

This may take a few thought experiments or additional reading to understand. The intuition is that each member of the `f` container contains a reference to the next link in the chain, or a `Pure` value, we follow the chain using `fmap` until we eventually reach a `Pure` value. At that point we simply apply the function, in many cases replacing the `Pure` value with the next link in the chain, and on and on we go. The important result is that the monadic sequencing operators now simply append links to our execution chain in the spot specified by the algebraic datatype. This will hopefully become more clear in light of our concrete use-case.

We can model the operations in our editor as a Functor for use with Free like so:

```
data Action next =
  GetText (String -> next)
  | SetText String next
  | Save Filename next
```

The data type we have described takes a single type parameter which we've called `next`. There is a constructor for each of the primitive actions we wish to allow, we shall see later how `appendText` is implemented in terms of these primitives. The reference to the `next` type parameter appears in each constructor, this is the 'slot' in which we will store the next action to be performed forming a recursive tree of actions which describe the computation. Note the interesting case of the `GetText` constructor where `next` is in the position of the result of a function call. This is valid and represents the idea that we may not know which computation to perform after the `GetText` action until the particular String is known. It is exactly this notion which allows us to define a Monad on the Free structure and sequence commands.

We could choose to have Haskell derive the Functor instance for us, but it is perhaps useful to see it written out long-hand.

```
instance Functor Action where
   -- The next link is in the result of `stringToNext`
   -- so we need to use fmap over the function to transform it
  fmap f (GetText stringToNext) = GetText (fmap f stringToNext)
  fmap f (SetText text next) = SetText text (f next)
  fmap f (Save filename next) = Save filename (f next)
```

We now gain the Monad instance 'for free' simply by wrapping values of our Action type in the `Free` constructor.

Now that our types are created we must expose a Monadic interface for the system. This is accomplished by wrapping each member of our data-type within Free. Remember that the `Free` constructor has the type: `Free (f (Free f result))` which is a bit confusing, but means that the type we wrap in the `Free` constructor has to be a Functor with another value of the `Free` type inside its 'slot'. We've already seen that `Action` is a functor, so the simplest way we can satisfy the type required to have a monad is to embed each member of Action into a `Free`, and to ensure that all of the slots in the constructor are filled by some member of `Free`. For actions like `setText` and `saveFile` which do not provide any return values we can fill the slot with a `Pure ()` and be done with it. `getText` provides text as a return value when executed, so we fill the slot with a function which returns the `Pure` text.

```
getText :: Free Action String
getText = Free (GetText (\text -> Pure text))

setText :: String -> Free Action ()
setText text = Free (SetText text (Pure ()))

save :: String -> Free Action ()
save filename = Free (Save filename (Pure ()))
```

Making use of the Free Monad

We have now redefined all of the primitives in the API, we can now rely on the Monad provided by Free to sequence the primitives and build up more complex commands. Let's start by defining the missing `appendText` operator:

```
appendText :: Free Action ()
appendText newText = do
  text <- getText
  setText $ text ++ newText
```

It may seem strange that we have started using the primitives even though we have not yet defined the actual behaviour of saving a file or setting text; this is

how Free Monads allow us to separate the semantics of *what* we wish to do from *how* we do it. All actions are described by discrete combinations of the Actions we have described; this means that the State Monad combinators are no longer valid, and also that arbitrary IO is not allowed.

Here is a peek at what the resulting data describing the `appendText` action looks like when expanded from do-notation:

```
appendText newText =
  Free (GetText (\text -> Free (SetText (text ++ newText) (Pure ()))))
```

This is tricky to read, but shows how the description of a computation translates into an algebraic data structure. This has been a lot of boiler-plate to achieve quite similar results to the state-based API, so now let's look at the pay-off by examining the two problems the old system had.

Note how the new `appendText` has behaviour equivalent to the State Monad version which used `modify` directly. We have achieved the same result, but in this case were restricted to using only the actions we've defined in our Action type; we cannot 'cheat' around the interface we are provided since we don't have direct access to any underlying State Monad via `get`, `put`, or `modify`. It is true that someone could add a new member of the Action type to circumvent using `getText` and `setText`, but this is more work than implementing it in terms of existing combinators and requires the implementer to look directly at the existing commands before adding another so it is much less likely to occur. Also notice how the new `saveFile` has no mention of IO whatsoever and does not allow the extension implementer to do anything malicious.

Interpreters

Eventually we will need to actually run actions which we describe, this is accomplished by designing an **interpreter** for our data structure. The `free` package provides many combinators which assist in this process, but to be thorough we will write our interpreters in long-hand. Here is our first interpreter:

```
type EditorText = String
runAction :: Free Action a -> StateT EditorText IO a
runAction (Pure a) = return a
runAction (Free action) =
  case action of
    (GetText getNextAction) -> do
      text <- get
      runAction $ getNextAction text

    (SetText newText nextAction) -> do
      put newText
      runAction nextAction
```

```
    (Save filename nextAction) -> do
      text <- get
      liftIO $ writeFile filename text
      runAction nextAction
```

The interpreter handles each action description and translates our AST into a `StateT EditorText IO` which operates the same as the pre-free version we had written earlier. It seems like we added a lot of intermediate steps to end up back where we started, but we have gained some useful guarantees. The extension may not invoke any IO directly, they may only say that they would like to save a file and the specification of the IO for this is left to the interpreter. `runAction` is one particular interpreter, but we can implement many others! For example if we would like to log which actions are performed by an extension an alternate interpreter may build up a log file which describes what will happen, this might be useful for debugging complex actions, we can run them through our logger and get a record of which steps occurred without worrying about saving to a file.

```
logActions :: Free Action a -> StateT EditorText (Writer String) ()
logActions (Pure _) = return ()
logActions (Free action) =
  case action of
    (GetText getNextAction) -> do
      tell "GetText\n"
      text <- get
      logActions $ getNextAction text

    (SetText newText nextAction) -> do
      tell "SetText: " ++ newText ++  "\n"
      put newText
      logActions nextAction

    (Save filename nextAction) -> do
      tell "Saved File to: " ++ filename ++  "\n"
      logActions nextAction
```

The `logActions` interpreter runs the same actions as the `runAction` interpreter, but builds up a log of what happened by using the Writer monad. We chose not to actually write any files since we don't want to mutate the file-system when simply logging actions.

Finally we may imagine an interpreter designed for testing. 'Mocking' out stateful operations during testing is a tricky problem in most languages, pure functional languages are not immune from these difficulties. The difficulty comes because we cannot get 'inside' of IO operations to alter return values from external sources like database queries or file-system access. In a Free Monad however we can solve this problem by avoiding IO entirely, we know exactly when the program is querying the data-base or reading from the file-system

because it is specified as a piece of data, when we interpret such actions inside a test interpreter we may inject data at specific query points while falling back on a standard interpreter for other actions. Here is one way we may choose to mock-out calls to a file-system when testing some API, it seems trivial in this case since the only possible actions involve the file-system, but you may imagine a more complex system in which file-system access is only a small piece.

```
import Data.Map

type Filename = String
type File = String
data Action =
  ReadFile Filename (File -> next)
  WriteFile Filename File next

data TestResult =
  Success
  | Failure String

-- We can mock out the file-system as a map of filenames to file contents
type FilesMock = Map Filename File

testAction :: Free Action TestResult -> State FilesMock TestResult
testAction (Pure testResult) = return testResult
testAction (Free action) =
  case action of
    (ReadFile filename toNextAction) -> do
      filesMap <- get
      let file = lookup filename filesMap
      case file of
        -- If the file exists we pretend we read it and pass on the contents
        Just f -> testAction $ toNextAction f
        -- If the file does not exist the test fails
        Nothing -> Failure $ "Couldn't find file: " ++ filename

    (WriteFile filename file nextAction) -> do
      modify (insert filename file)
      testAction nextAction
```

Assuming we have written similar helpers to embed WriteFile and ReadFile into Free as we did with the previous example we may define a test for our filesystem like so:

```
testSavesFiles :: Free Action TestResult
testSavesFiles = do
  writeFile "testfile.txt" "hello"
  result <- readFile "testfile.txt"
```

```
  return $ if result == "hello"
            then Success
            else Failure $ "Expected 'hello' but received: " ++ result
```

Though this is a contrived example wherein the entire system is mocked it is a useful pattern in larger more complex systems which rely on external sources or even libraries.

Program Transformations

We have seen how the Free Monad allows us to design a DSL for our API, how we can restrict the actions performed by extensions and untrusted code, and how it can benefit us during testing. There is still more that we can do! Another benefit of representing our computation as data, is that data can easily be transformed. We can choose to run transformations over computational data structures in order to improve performance, add logging, or handle other cross-cutting concerns.

We return to the text-editor example we explored earlier. `saveFile` was one of our available actions; typically file-system access is a relatively expensive operation which we would prefer to avoid whenever possible. We can write a simple transformation over our data-type which removes redundant `saveFile` operations when no changes have occurred in between.

```
-- This transforms the AST without affecting its type;
-- this means we can run it as an optimization step over
-- all of our computations before interpreting them.
removeRedundantSaves :: Free Action a -> Free Action a
-- Pure values are unaffected
removeRedundantSaves (Pure a) = Pure a
removeRedundantSaves (Free f) = Free $
  -- pattern match on the current term
  -- if it is a 'Save' with another 'Save' as the next action
  -- and the filenames are the same, then we can reduce them to
  -- a single save, we make sure to recurse on the rest of the
  -- datatype to perform the optimization over the entire AST.
  case f of
    (Save fname inside@(Free (Save fname' next))) ->
      if fname == fname'
         then (Save fname (removeRedundantSaves next))
         else (Save fname (removeRedundantSaves inside))
    _ -> fmap removeRedundantSaves f
```

This is just one such simple optimization, there is substantial existing research regarding other similar forms of optimizations over Free Monads and Free Applicatives (Marlow et al. 2016, Capriotti and Kaposi (2014)); one such example includes batching sequential Database reads/writes to run in parallel.

These transformations can be performed dynamically at run-time if necessary and in many cases are lazily queued to run only once a computational path is finitely determined; this is necessary since Monads may choose their execution path based on previous values. Since the transformations return the initial type they may be chained together through composition. In many cases compilers can 'fuse' all transformations into a single pass over the structure.

When combined, the benefits gained by utilizing Free Monads provide a powerful tool-set with a strong set of guarantees and enhanced flexibility. They allow the implementer to separate cross-cutting concerns into transformations which may be applied selectively.

In Summary

We have discovered that although a pure functional language such as Haskell does not always have a one-to-one equivalence for each imperative concept, it does have some method of achieving feature parity in each case. The description of such methods have been long-winded in this paper primarily to lay the ground-work for functional concepts which may perhaps be unfamiliar to an imperative programmer, but this is a fixed cost inherent in the learning of any new discipline. Once initial barriers to understanding have been surmounted we see that functional methods provide many guarantees and benefits over the imperative approach, and in many cases result in a system of simple composable components. For an understanding of how these methods may be applied in practice you may read further in the guides and documentation of the Rasa Text-Editor (Rasa 2016–2017) and the Eve Application Framework (Eve 2017).

Appendix: Learning Resources

- Reading Haskell: https://wiki.haskell.org/How_to_read_Haskell
- Monoids Tour: https://www.schoolofhaskell.com/user/mgsloan/monoids-tour
- Functors and Monads: http://learnyouahaskell.com/functors-applicative-functors-and-monoids
- Monad Transformers: https://en.wikibooks.org/wiki/Haskell/Monad_transformers
- State Monad: http://brandon.si/code/the-state-monad-a-tutorial-for-the-confused/
- Free Monads: http://www.haskellforall.com/2012/06/you-could-have-invented-free-monads.html

Citations

Capriotti, Paolo, and Ambrus Kaposi. 2014. "Free Applicative Functors." In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, Msfp@ETAPS 2014, Grenoble, France, 12 April 2014.*, 2–30. doi:10.4204/EPTCS.153.2.

Gonzales, Gabriel. 2017. "Applied Category Theory and Abstract Algebra - Lambdaconf Winter Retreat 2017." https://youtu.be/WsA7GtUQeB8; LambdaConf.

Marlow, Simon, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. "Desugaring Haskell's Do-Notation into Applicative Operations." In *Proceedings of the 9th International Symposium on Haskell*, 92–104. Haskell 2016. New York, NY, USA: ACM. doi:10.1145/2976002.2976007.

Penner, Chris. 2017. "Eve: Extensible Application Framework." https://github.com/chrispenner/eve.

———. 2016–2017. "Rasa: Modular Text Editor." https://github.com/chrispenner/rasa.