MFlow, a continuation-based web framework without continuations

by Alberto Gomez Corona ⟨agocorona@gmail.com⟩

April 23, 2014

Most of the problems which complicate the coding and debugging of web applications come from the same properties which make web applications scalable and flexible: HTTP is stateless and HTML is untyped. Statelessness gives freedom to the user, but it implies an inversion of control, where the HTTP requests invoke different request handlers that manage a global state and generate individual web pages.

Dijkstra's famous admonitions about goto statements apply to request handling mechanisms that manage these state transitions. Handlers do not share variable scopes, so, like in the case of the goto, code relies on global variables or on global session variables. There is no top-level structure in the code to make the main sequence evident. For the maintainer, it is very difficult to know what any piece of code is trying to achieve. Even worse, there is no inherent fixed sequence and the execution may be broken by out-of-order requests that do not match with the state of the application, forcing developers to pollute code with checking conditions and error messages.

Continuation-based frameworks solve these problems since they maintain execution state snapshots that match with each possible request. But continuations are memory hungry, and they cannot be easily serialized and shared. This precludes horizontal scalability and failover. As a result, these web frameworks have not been accepted by the mainstream, despite their advantages.

In this article, I will show how an alternative monadic structure for thread state serialization and backtracking can solve these problems. MFlow [1] is a Haskell web framework that automatically handles the back button and other out-of-order requests. Unlike continuation-based frameworks, each GET page is addressable by a REST URL, and the architecture is scalable since the state is composed of serializable events. It is based on enhanced formlets that permit the creation of fully

self-contained, composable components called widgets with high level combinators, allowing for the creation of dynamic applications with no explicit use of JavaScript.

Motivation: State management

Many web applications require the same sequence of steps to execute in different contexts. Often, these sequences are merely components of a larger task that the user is trying to accomplish. Such a reusable sequence is called a flow. User registration, log in, checking in for a flight, applying for a loan, shopping cart checkout, or even adding a confirmation step to a form: these are all examples of a flow. In many of these cases, the website needs to store state, and this state sometimes must extend across browser sessions. Storing all the state in the browser can have security issues, and the state is limited to a single web browser. To deal with this problem, web frameworks include server session state that every HTTP event handler can access and modify. The program becomes a state transition machine: such state transitions are hard to code and maintain.

The *Spring Web Flow*, a flow oriented framework on top of the Java *Spring* framework enumerates the following problems when developing web applications [2]:

- Visualizing the flow is very difficult.
- The application has a lot of code accessing the session state.
- Enforcing controlled navigation is important but not possible.
- Proper browser back button support seems unattainable.
- Browser and server get out of sync with "Back" button use.
- Multiple browser tabs cause concurrency issues with HTTP session data.

The ideal solution would be to codify the application flow in the most natural and maintainable way possible. It would be necessary to preserve, in the code, the sequencing whenever the sequence exist, in the same way as a console application. However, each individual page should be addressable by means of an URL.

There is an apparent incompatibility between the state transition machine model, enforced by the web architecture, and the sequential description proposed above. However, the request handlers triggered by each request can be considered as a continuation in the flow, with the state generated by all the previous events. It is possible to invert back the inversion of control [3] so that the program can express the flow directly in the form of a sequence which contains inputs and outputs from/to the Web browser, just like a console application.

This transformation does not change the way the web server and the program interact: the server invokes request handlers, but the handlers are sequenced inside a procedure, so that they share variable scopes. Since the state is in the form of ordinary language variables, the maintainer can observe the sequence of state changes, and a statically typed compiler can verify the consistency of the state transitions. Each request goes to the handler that has the appropriate state to handle the request. This is the basic design of continuation-based frameworks such are ocsigen (OCaml) [4] or Seaside (Smalltalk) [5].

However, serialization and sharing of continuation state across computer nodes is necessary if we want state persistence and horizontal scalability.

The input-output load involved in state sharing is the reason why REST recommends to use as little state as possible. This is hard to do: even if the language has serializable closures, a continuation includes program code which is linked with many libraries which may render the serialization huge; additionally, it may be linked to non serializable objects of the operating system [6].

The problem of the size and portability of serialized continuations is still unsolved, although there have been some advances on this area in the Scala language [7]. Another option is to store the portable state in the client [8]. This solution constrains the state to a single browser session.

Another way to deal with these problems is to redirect every user to an unique server. In long living flows, especially when it exceeds the mean time between failures, this can end up in loss of state. each page in a flow supposes the serialization of the closure that handles the request of the page, so that the problem of state management multiplies by the number of steps. Maintaining only the most recent states impairs the navigability with the back button. These problems have maintained the continuation-based frameworks out of mainstream use, despite their huge advantages.

MFlow state management

In the Haskell language, it is not possible, to date, to serialize a dynamic closure. Cloud Haskell [**?**] communicate pointers to closures whose addresses are know at compilation time. That method is not feasible for the dynamic closures generated by continuations, that include the state of the computation.

However, it is possible to create an execution log and recover the execution state by running the program taking as inputs the logged events when the procedure is re-started. The log will contain the intermediate results of each step in the computation, though not the computation itself. The machinery for log creation and recovery can be hidden in a monad.

This is the purpose of the Workflow monad, defined in the workflow package [10]. The effect of logging and recovery are managed by the `step` call. The monad is essentially a state transformer which stores intermediate values in persistent storage.

The pseudocode of the lift operation (`step`) that would do logging and recovery would be:

```
import Control.Monad.State

type Workflow = StateT

step:: m a -> Workflow m a
step mx = do
    st <- get
    if thereRemainResultsInLog st
        then getResultfromLog st
        else do
            r <- mx
            storeInLog r st
            return r
```

Since the state must be written to (read from) permanent storage, results must be serializable, so `step` is not defined for all types, but for types with Serialize instances. The real signature is:

```
step:: (MonadIO m, Serialize a, Typeable a) => m a -> Workflow m a
```

To make the logging and recovery more efficient, the workflow monad caches the read and write operations. `Serialize`, from the RefSerialize [11] package, allows incremental serializations of the modifications of a data structure in the log, rather than logging the entire structure in each step. This is very useful when the flow manages containers or large user-defined structures.

Let's see how to use the workflow monad. A computation in the IO monad, for example:

```
main = do
    n  <-  ask "give me the first number"
    n' <-  ask "give me the second number"
    print $ n + n'

ask s = do
    putStrlLn s
    r <- getLine
    return $ read r
```

Can be lifted to the workflow monad:

```
include Control.Workflow
include Control.Monad.IO.Class(liftIO)

main = exec1 "wfname" $ do
    n  <- step $ ask "give me the first number"
    n' <- step $ ask "give me the second number"
    liftIO . print $ n + n'
```

The lifted computation includes effects of logging and recovery. If we interrupt the program before answering the second question, upon restart the computation will not ask the first question. Instead, it will retrieve the answer from the log and immediately ask the second question. The variables and the execution state then will be the same than when the program was interrupted: the state has been recovered from the log.

If we define `ask` to handle HTTP requests and responses instead of console input/output, a procedure of this kind can express a web flow. In particular, the log can be stored in the browser by means of hidden form variables. If the process is finished after each response, that would make the server applications stateless and, hence, very scalable and fullly REST compliant, since each page keeps its own log state. That is the solution adopted by Peter Thiemann in WASH [12], an excellent Web framework mistreated by its syntax. But client stored state is not appropriate in many cases, for the functional and security reasons above mentioned. Moreover, a re-spawn of the process and a replay of the log are necessary on every request.

MFlow takes a different approach. In this case the server stay running waiting for the next request until a timeout is reached, instead of being stopped after each request. The non-blocking nature of the Haskell threads make the context switch for each request as efficient as event handlers.

After the timeout the process dies. When a new request arrive, the process restart using the log to recover his state. Unfortunately, there is an added problem on this new approach: the server process must synchronize in some way when the process is running and receives out of order requests instead of the next request in the flow sequence, e.g., when the user has pressed the back button.

Synchronization problem

To solve the back button problem, a form of backtracking is necessary, so that when the browser send a request from of a previous page in the navigation, the procedure can go back until a previous page handler in the flow match with the data sent. From this moment on, the flow proceed normally. This is the purpose of the fail-back monad:

```
data FailBack a = BackPoint a | NoBack a | GoBack

newtype BackT m a = BackT { runBackT :: m (FailBack a ) }


instance Monad m => Monad (BackT  m) where
    fail  _ = BackT $ return GoBack
    return x = BackT . return $ NoBack x
    x >>= f  = BackT $ loop
    where
    loop = do
        v <- runBackT x
        case v of
            NoBack y  -> runBackT (f y)
            BackPoint y  -> do
                z <- runBackT (f y)
                case z of
                  GoBack  -> loop
                  other -> return other
            GoBack -> return  GoBack


instance MonadTrans BackT where
    lift f = BackT $ f >>= \x -> return $ NoBack x

breturn = flowM . BackT . return . BackPoint
```

This monad executes as an Identity monad as long as it is handling `NoBack` results. But if `GoBack` is returned by some statement, then, if the previous one returned a `BackPoint`, it will be re-executed again.

If the previous statement was not a `BackPoint`, then this previous statement will return `GoBack` as well, so the computation will proceed further back until another previous `BackPoint` is found.

Each `BackPoint` in the backtrack is re-evaluated. If it returns `NoBack` or `BackPoint`, the process will resume forward execution. If it returns `GoBack`, it continues backtracking.

This console application is the same above program lifted to the fail-back monad:

```
main = runBackT $ do
    lift (print "will return here at most") >> breturn ()
    n  <- lift  $ ask "give me the first number"
    n'  <- lift  $ ask "give me the second number"
    lift $ print  $  n + n'

    where
    ask s = do
        putStrLn s
        s <- getLine
        if s == "back" then  fail "" else breturn $ read s
```

Here, `ask` is designed to return to the previous line when the string "back" is entered. In the last line, `ask` either fails and initiates a backtracking to the previous ask or returns with `breturn` and becomes a back point that will be called again in case of backtracking.

If `ask` is redefined to accept formlets [13] descriptions instead of text, to send/receive HTTP requests and responses, then the fail-back monad can handle gracefully the back button: it is only necessary to verify the match of the parameters with what the formlet expects. If there is no match, it triggers backtracking until some previous ask match. This solves the problem of the back button.

All that concerning the back button, but that does not solve the general problem of tracking from a branch in the navigation tree to another. I will show below how backtracking together with a careful design of link syntax make it possible to address any page in the flow *forward* by means of a REST URL.

The `FlowM` monad is essentially the same failback transformer stacked above a State transformer that carries out the formlet state. It may be stacked on top of the (`Workflow IO`) monad or the `IO` monad, depending on whether or not persistent state is desired:

```
newtype FlowM v m a = FlowM {runFlowM :: FlowMM v m a}
  deriving (Monad, MonadIO, MonadState (MFlowState v))

type WState view m = StateT (MFlowState view) m
type FlowMM view m = BackT (WState view m)
```

User Interface

The rendering and validation in MFlow are inspired by the formlets [13] concept. The View data carries out a rendering format `v` for an underlying monad `m`, which is `IO` always, except in the case of cached widgets.

As in the case of the `FlowM` monad, the View data carries the formlet state information.

The formlet information is stored in a `FormElm` data structure, which has the generated rendering as well as the result of the match of the request parameters with the formlet in a Maybe value:

```
newtype View v m a = View { runView :: WState v m (FormElm v a) }

data FormElm v a = FormElm [v] (Maybe a)
```

The `v` in these definitions is the particular rendering. A widget must have a rendering format that must be an instance of the `FormInput` class.

```
class (Monoid view,Typeable view)  => FormInput view where

    toByteString ::  view -> B.ByteString
    toHttpData ::  view -> HttpData
    fromStr ::  String -> view
    fromStrNoEncode ::  String -> view
    ftag ::  String -> view  -> view
    inred  ::  view -> view
    flink ::  String -> view -> view
    flink1::  String -> view
    flink1 verb = flink verb (fromStr  verb)
    finput ::  Name -> Type -> Value -> Checked -> OnClick -> view
    ftextarea ::  String -> T.Text -> view
    fselect ::  String -> view -> view
    foption ::  String -> view -> Bool -> view
    foption1 ::  String -> Bool -> view
    foption1  val msel = foption val (fromStr val) msel
    formAction  ::  String -> view -> view
    attrs ::  view -> Attribs -> view
```

This class describes how to create form elements, links and error messages in that particular format. Using the class methods, the widgets are not tied to a particular rendering. In most cases, blaze-html rendering is used, but there are also bindings for xhtml and HSP. It is possible to create widgets that can use any rendering, thanks to the `FormInput` abstraction. The widgets in `MFlow.Forms.Widgets` module are defined in this way.

The pseudocode of an `Int` input box would be:

```
getInt :: (FormInput view, MonadIO m) => Maybe Int -> View view m Int
getInt = View $ do
    parm <-  genNewId
    mr <- lookupParam parm
    return \$ FormElm [finput "text" parm "" False Nothing ] $  mr
```

`ask` gets a widget, displays it, gets the response from the user and returns it under the FlowM monad:

```
ask ::  (FormInput view) => View view IO a -> FlowM view IO a

page = ask
```

`page` is a synonym of `ask`

The formlets can be combined with applicative operators to create more complex widgets. A page is a widget. For this purpose, `View` has an applicative instance:

```
instance (Functor m, Monad m) => Applicative (View view m) where
  pure a  = View $  return (FormElm [] $ Just a)
  View f <*> View g = View $
                  f >>= \(FormElm form1 k) ->
                  g >>= \(FormElm form2 x) ->
                  return $ FormElm (form1 ++ form2) (k <*> x)
```

This is a complete program that uses the FlowM monad and widgets combined with applicative operators:

```
{-# LANGUAGE OverloadedStrings #-}
import MFlow.Wai.Blaze.Html.All
import Control.Applicative
```

```
main = runNavigation "sum" . step $ do
      setHeader html . body
      (n1,n2) <- page $ (,) <$> getInt Nothing <++ br
                            <*> getInt Nothing <++ br
                            <** submitButton "send"

      page $ p << (n1+n2) ++> wlink () "click to repeat"
```

runNavigation executes the FlowM computation in a endless loop.

```
runNavigation ::  String -> FlowM Html (Workflow IO) () -> IO ()
```

It expects persistent navigation, but the example is transient, so we apply a single step to the transient computation. The first page rendering is an applicative combination of getInt formlets to create a 2-tuple.

There are line breaks (br) added using blaze-html formatting. They are added with the operator:

```
(++>)    :: v -> View v m a -> View v m a
```

…that add formatting to a widget.

This operator:

```
(<<) :: (v -> v) -> v -> v
```

…encloses formatting within a tag (in this case the blaze-html tag “b”). Since ■ has less precedence that ++>, the composition does not need parentheses.

The second page presents an HTML paragraph with the sum of the numbers. The wlink widget renders an HTML link, with a URL that invokes the flow again; the page will return wlink value.

```
wlink ::  (Typeable a, Show a, MonadIO m,  FormInput view)
        => a -> view -> View  view m a
```

Because wlink is in the last FlowM statement, runNavigation will restart the computation again.

Routing

Note that `wlink` can also consume inmediately an element of a RESTful path, if it is available, without displaying anything. When a widget returns a match, `page` does not send a page response. Instead, it returns the result to the flow immediately. This is key for addressing any page in the flow in a REST URL:

```
data Option = Option1 | Option2

main = runNavigation "verb" .  step  $  do
    r <- page   $ h3 << "Alternative operator  used here"
              ++> wlink Option1 << b << "Choose Option1" <++ br
              <|> wlink Option2 << b << "Choose Option2"

    case r of
      Option1 -> proc1
      Option2 -> proc2
```

In this example, URLs which start with `http://host/verb/option1/...` will go straight to `proc1`: the menu will not be presented. As you would expect, URLs which start with `http://host/verb/option2/..` will go straight to proc2. The URL `http://host/verb` will go to the initial menu.

Thus, it is possible to express REST routes as flows using `wlink`. The code above has three implicit routes.

This is combined with the backtracking mechanism to make any GET page in the flow addressable. For example, if in the above example the flow is running the `proc1` branch, when the user enters the URL of the second option, the backtracking mechanism will backtrack to `/verb`, after which the second wlink will match the `/option2` segment of the URL, so `proc2` will be executed.

Single page applications

As seen above, the applicative instance permits the combination of widgets using applicative operators. But `View` also has an Monad instance:

```
instance (Monad m) => Monad (View view m) where
    View x >>= f = View $ do
                  FormElm form1 mk <-  x
                  case mk of
                    Just k  -> do
                        FormElm form2 mk <-  runView $ f k
                        return $ FormElm (form1 ++ form2) mk
                    Nothing ->
                        return $ FormElm form1 Nothing

    return = View .  return . FormElm  [] . Just
```

This monad instance is somewhat remarkable. In this monadic instance, the second widget consumes the output of the previous statement, provided that the latter validates. But if the previous widget return `Nothing`, then the computation is interrupted and the rendering becomes what was produced until that moment.

`page` in this case will present the rendering again and again until the monadic computation finish with a valid output that will be returned to the flow. Since each iteration read the input of the previous one and each widget can modify its presentation depending on the previous widget output, sophisticated input forms and presentations are possible:

```
data Client = Person{pname, paddress  :: String, age :: Int}
            | Company{cname,caddress :: String, numWorkers :: Int}

main = runNavigation "input" . step $ do
    page $ do
        type <- getRadio[\n -> n ++> setRadioActive v n
                        | v <-["person","company"]
        case type of
          "person" -> Person
                          <$> "name: "  ++> getString Nothing
                          <*> "address " ++> getString Nothing
                          <*> "age "    ++> getInt Nothing
          "company" -> Company
                          <$>"name: "    ++> getString Nothing
                          <*> "address " ++> getString Nothing
                          <*> "workers " ++> getInt Nothing
    ...
```

Here, two options in a radio button are presented. Because it is a monad, the second statement can use the `type` to present the appropriate applicative form.

In the first iteration, only the radio buttons are presented. When the user chooses one of the options, the appropriate formulary appears below the radio buttons. When the user completes the formulary, `page` will return the `Client` data to the flow. If the user changes the option in the radio button, the form will change accordingly.

To avoid sending the whole page at each iteration, there are some page modifiers that refresh only the elements that change using AJAX. Push is also possible using long polling. The execution of the server code is the same but only the rendering of the widget activated by the request is refreshed in the page.

Note that the single page example above can be transformed into a two pages example by translating the View monad to the FlowM monad with only a few changes.

If in the bind operation of the View monad we discard the rendering of the first formlet, the effect is that the rendering of the second formlet is substituted for the first when the former is validated. That is how MFlow implements the callback mechanism of the Seaside framework [**?**]

Using these and other combinators it is possible to create different kind of applications using basically the same above elements in a pure Haskell EDSL [15].

Web services

A widget in MFlow is essentially a request parser and a writer of responses. A page is the composition of this kind of elements.

That parser-writer combination of the View monad can be used as a general request-response mechanism. Adding a simple set of combinators it is possible to create web services in an idiomatic way without the need of special constructions.

The three services implement a sum and a product of two integers.

First, a REST web service, where the parameters are in a REST path.

```
main = runNavigation "apirest" . step $ ask $ do
         op <- getRestParam
         term1 <- getRestParam
         term2 <- getRestParam
         case (op, term1,term2) of
           (Just  "sum", Just x, Just y) ->  wrender (x + y :: Int) **> stop
           (Just "prod", Just x, Just y) ->  wrender (x * y) **> stop
           _  ->do -- blaze Html
                     h1 << "ERROR. API usage:"
                     h3 << "http://server/api/sum/[Int]/[Int]"
                     h3 << "http://server/api/prod/[Int]/[Int]"
                 ++> stop

stop = empty
```

`getRestParam` read the next REST segment in the path if there is any.

`stop` is a synomym of of the `Applicative` class `empty` method. By definition, the computation will stop, `ask` will fail and the Flow will not navigate forward. The content of the writer (the one generated by `wrender`) is sent as response. If the parameters are not the expected ones, the a HTML message is sent.

Some example of invocations:

```
> curl "http://mflowdemo.herokuapp.com/apirest/prod/4/3"
12

> curl "http://mflowdemo.herokuapp.com/apirest/sum/5/7"
12
```

The second example read key-value parameters for the numbers instead of REST parameters:

```
main = runNavigation "apikw" . step . ask $ do
         op <- getRestParam
         term1 <- getKeyValueParam "t1"
         term2 <- getKeyValueParam "t2"
         case (op, term1, term2) of
           (Just  "sum", Just x, Just y) -> wrender (x + y :: Int) **> stop
           (Just "prod", Just x, Just y) -> wrender (x * y) **> stop
           _ -> do -- blaze-html
                     h1 << "ERROR. API usage:"
                     h3 << "http://server/api/sum?t1=[Int]&t2=[Int]"
                     h3 << "http://server/api/prod?t1=[Int]&t2=[Int]"
                  ++> stop
```

Some example invocation:

```
> curl "http://mflowdemo.herokuapp.com/apikv/prod?t1=4&t2=3"
12

> curl "http://mflowdemo.herokuapp.com/apikv/sum?t1=4&t2=3"
7
```

The third service uses key-value parameters as well, but there are defined parsec-like combinators that are Widgets as well:

```
main  = runNavigation "apiparser" . step . asks $
             do rest "sum" ; disp $ (+) <$> wint "t1" <*> wint "t2"
```

```
      <|> do rest "prod"; disp $ (*) <$> wint "t1" <*> wint "t2"
      <?> do  -- blaze Html
             h1 << "ERROR. API usage:"
             h3 << "http://server/api/sum?t1=[Int]&t2=[Int]"
             h3 << "http://server/api/prod?t1=[Int]&t2=[Int]"
  where
  asks w = ask $ w >> stop
```

Some example invocation:

```
> curl "http://mflowdemo.herokuapp.com/apiparser/prod?t1=4&t2=3"
12

> curl "http://mflowdemo.herokuapp.com/apiparser/sum?t1=4&t2=3"
7
```

The combinators `rest` and `wint` are defined below, from `getRestParam` and `getKeyValueParam`. They are part of the `MFlow.Forms.WebApi` module

`rest` verify that the next rest parameter is an expected value. `wint` read a parameter of type Int which has a given key.

The operator `<?>` present a blaze-html message when the parser does not match, like in a parser combinator.

`disp` writes the result of a computation.

It uses the same applicative, alternative and monadic combinators defined for any other widget in MFlow.

```
stop = empty

restp = View $ do
   mr <- getRestParam
   return $ FormElm [] mr

rest v = do
   r <- restp
   if r == v
     then return v
     -- restore the rest index and fail
     else modify (\s -> s{mfPIndex = mfPIndex s-1}) >> stop

wparam par = View $ do
   mr <- getKeyValueParam par
   return $ FormElm [] mr

disp :: Show a => View Html IO a -> View Html IO ()
disp w = View $ do
   elm@(FormElm f mx) <- runView w
   case mx of
     Nothing -> return $ FormElm f Nothing
     justx@(Just x) -> return $ FormElm (f++[fromStr $ show x])
                            $ return ()

infixl 3 <?>
(<?>) w v = View $ do
  r@(FormElm f mx) <- runView w
  case mx of
    Nothing -> runView $ v ++> stop
    Just _ -> return r

wint p = wparam p :: View Html IO Int
```

Scalability

Because the logs grow by small update events, it is easy for two or more servers to synchronize state by interchanging `step` events instead of entire states. The small size of the step events makes MFlow state easier to synchronize and architecture independent, without the problems of continuation-based frameworks.

Execution traces

Trace logging by means of hand-made statements, although tedious and cumbersome, is a traditional way of tracking errors in web applications in production environments.

MFlow produces automatic execution traces under the `FlowM v IO` monad. These logs are produced at failure time. This log is created by the same backtracking mechanism of the fail-back monad.

Other exception-treatment monads like Error, Maybe or the Exception monad fail back to the calling method. The failback monad instead returns to the previous statement, so a trace rather than a call stack will be produced. This is a great improvement, especially for web applications where it is necessary to extract the maximum amount of information from each failure in the exploitation environment.

The trace mechanism uses the monadloc[16] package. `withLoc` is a method of the MonadLoc class, so it can be redefined for the particular needs of each monad. In this case, the instance inserts an exception handler that adds the line of error to the trace and initiates a backtracking. The backtracking proceeds back to the beginning, following the execution history in reverse order. This is the pseudocode of the instance:

```
instance  MonadLoc (FlowM v IO)  where
    withLoc loc f = do
          r <- compute f ‘catch‘ (\e ->do
                    insert (show e) in the list
                    return GoBack)  -- backtrack
          if  trace going
              prepend location (loc) info to the list
              return GoBack  -- continue backtracking
            else
              return r  -- normal return
```

The monadloc-pp preprocessor inserts a `withLoc` call behind each line of code with information about the module and line number within the first parameter.

When the backtracking is complete, the scheduler detects the trace at the root of the execution and prints it in the console.

To see an example of trace log, see [17]. In the case of persistent flows, the FlowM v (Workflow IO) monad generates its own log at execution time, which is readable and can be inspected in case of error.

Conclusions

MFlow demonstrates the power of monadic computations for creating web flows and web navigation in a concise, intuitive and maintainable way, while reducing drastically the plumbing and the error ratio in web programming. The navigation is verifiable at compile time, and this facilitates the testing and maintenance, while the scalability and navigability is not compromised.

Future work

One important task in the future is to develop a synchronization mechanism for MFlow servers to realize the theoretical scalability and failover of the architecture, using Cloud Haskel [18].

The routing example shows how the FlowM machinery can work as an event scheduler without inversion of control. As such, it can be used in very different scenarios.

The supervisor package [19] contains a enhanced version of the fail-back monad applicable to any context. It and can perform specific actions when the computation goes forward and backward thanks to a programmer-defined instance. It will be used by MFlow in the future to substitute the less general fail-back monad.

Event scheduling without inversion of control is ideal for process interaction in cloud environments [20]. Optional backtracking and persistence can make it more expressive and convenient.

Persistent flows can be used in enterprise integration scenarios [21] with long-running transactions where backtracking can be used to perform rollbacks.

References

[1] http://www.haskell.org/package/MFlow.

[2] Spring web flow. http://projects.spring.io/spring-webflow.

[3] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. **SIGPLAN Not**, 38:page 2003 (2001).

[4] http://ocsigen.org.

[5] http://en.wikipedia.org/wiki/Seaside_(software).

[6] Continuation Fest 2008 Tokyo, Japan April 13, 2008. **Clicking on Delimited Continuations** (2008). http://okmij.org/ftp/Computation/Fest2008-talk-notes.pdf.

[7] Ian Clarke. Swarm, a new approach to distributed computation. video. http://tech.slashdot.org/story/09/10/11/1738234/Swarm-mdash-a-New-Approach-To-Distributed-Computation?from=rss.

[8] Jay A. McCarthy. Automatically restful web applications: marking modular serializable continuations. **SIGPLAN Not.**, 44(9):pages 299–310 (August 2009). http://doi.acm.org/10.1145/1631687.1596594.

[9] The distributed-static package. http://hackage.haskell.org/package/distributed-static/docs/Control-Distributed-Static.html.

[10] http://www.haskell.org/package/Workflow.

[11] The refserialize package. http://www.haskell.org/package/RefSerialize.

[12] http://www.informatik.uni-freiburg.de/~thiemann/WASH.

[13] http://www.haskell.org/package/formlets.

[14] L Renggli S Ducasse, A Lienhard. Seaside: A flexible environment for building dynamic web applications. Technical report. http://citeseer.uark.edu:8080/citeseerx/showciting;jsessionid=00305F8E94C1AA992461768584AA0B7E?cid=9279754.

[15] Mflow as a dsl for web applications. https://www.fpcomplete.com/user/agocorona/MFlowDSL.

[16] http://hackage.haskell.org/package/monadloc.

[17] Demo of error traces in mflow. http://mflowdemo.herokuapp.com/noscript/errortraces/trace.

[18] Cloud haskell. http://www.haskell.org/haskellwiki/Cloud_Haskell.

[19] The supervisor package. https://hackage.haskell.org/package/supervisor.

[20] Martin Odersky Philipp Haller. Event-based programming without inversion of control. http://lampwww.epfl.ch/~odersky/papers/jmlc06.pdf.

[21] Alberto G. Corona. How haskell can solve the integration problem. https://www.fpcomplete.com/business/blog/guest-post-solve-integration-problem.