LaTeX Error: Missing documentSee the LaTeX manual or LaTeX Companion for explanation.You're in trouble here. Try typing ¡return¿ to proceed.If that doesn't work, type X ¡return¿ to quit.911c
NTASYS
Madrid, Spain
NTASYS
Madrid, Spain
LaTeX Error: Missing documentSee the LaTeX manual or LaTeX Companion for explanation.You're in trouble here. Try typing ¡return¿ to proceed.If that doesn't work, type X ¡return¿ to quit.911c
New Technologies Dept.
New Technologies Dept.
agocorona@gmail.com, alberto.gomez@ntasys.com

# MFlow: a Web framework with intuitive, composable page navigation which is compile-time safe

Most of the problems that makes special the coding and debugging of web applications comes from what makes them scalable and flexible: its stateless, its untyped nature, and the use of a few elements: URIs MIME types and HTTP and representation languages like HTML or XML. Despite the fact that the Web frameworks add features such are sesin state and validation, the stateless nature imposes an event model where the HTTP requests invoke different code procedures that handle a global state. however in terms of code readability and maintainability the event model is a design level goto, that make the code hard to understand and prone to be broken by out or order request that do not match with the state of the application, so that the code becomes polluted with checking conditions and pages with error messages.

Here I present MFlow, a framework for the functional, strongly typed language Haskell, that is free from the event model problems but still agree broadly with the REST style. It can express page flows in a procedural way like a console application and it is not based on continuations. It handles out of order requests and the back button and each page is fully addressable by an URL. It is based on self contained, fully composable, reusable, type safe components that manage user data and has a test and traffic load generator that make use of Haskell testing techniques.

## 1 Introduction

The section 2 of this article describes the main problem that MFlow tries to address that is, the stateful nature of applications in contrast with the stateless, typeless nature of the Web architecture and the event model that this architectures imposes in Web applications. It is argued about the essential identity between the event model and the procedural model -in the form of continuations- for managing stateful applications. Then, it is shown that the latter is a higher level form of stateful application programming, although it has problems in terms of scalability and navigability when the navigation tree and the number of users grows from console applications to GUI applications to Web applications, where the event model shine. Then I expose the use of continuation passing style and their problems.

In the section 3, it is explained how MFlow solves the problems of the imperative model, so that the the application code remains high level, with type safety, navigability and scalability. Moreover, the navigation is verified at compile time.

In the section 4, the modular, type safe user interface that addresses the lack of type safety and componentization of HTML is described. The MFlow components -widgets- have their own server handling code, their HTML, AJAX, HTML formatting and their requirements in terms of javaSscript and CSS included in a single Haskell procedure.

In the section 5, a brief description of the transactional and persistence mechanism is described.

In the section 6 the testing facilities are described. They make use of the navigability and type safety to generate automatic test cases that include real user navigations.

Submitted to:
WWV 2013

## 2 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, login, checking in for a flight, applying for a loan, shopping cart checkout, or even adding a confirmation step to a form. They are all examples of a flow.

Readers that are familiar with business process management (BPM) will realize that a Web Flow is a specialized case of a general purpose workflow, so it could theoretically be implemented using a general purpose BPM system.

In these workflow systems, a Web site need to store state, and this state must extend across browser sessions, and across clients. Moreover, storing all the state in the browser, and to use the hyperlinked nature of HTML as the navigation as some REST advocates recommend, is vulnerable to hacking. So to store the state in the client is usually not an option. Thats why many world web sites are not allowed to do so for functional or security reasons.

In the server, the application state can not be managed as a independent resource, to be accessed alone, since the state is an auxiliary resource that must be accessed together with the applicaion data. For example, when the user proceed to check out, the state of the shopping -the shopping cart- and the stock of the product must be accessed at the same time. The management of the state as a resource is meaningful in a Web Service, where the execution flow is managed by a client program, but not in the case of a Web application when for functional or security reasons, the state can not be located in the browser.

To cover this mismatch, the event model includes a session state, that every event handler can access and modify. The server becomes a state transition machine. Such state transitions are hard to code. There are tools that translate flow chart drawings into some form of standard XML flow configuration that contains the state transitions. When a such tool is not available, the first task of the designer is to draw its own flowchart before manually translate it to state transitions in XML, java, Ruby or javascript code. This is the methodology used for web programming in the mainstream web frameworks, that agree more or or less with the Model-View-Controller (MVC) pattern, from CGI server extensions to more sophisticated web frameworks, like ASPX, Rails, Faces etc. To summarize, here are the main problems of this model:

- 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 causes 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, like a console application. but, unlike a console applications it is necessary to keep the addressability of each element (by means of a URL) when this element is meaningful to be addressable.

Additonally if the user returns to the flow, it must continue at the point when it was left unless the time for completion has expired. This is equivalent to how the humans execute an ordinary task: We the humans describe to others the work to do as sequences in flowchart drawings or in textual descriptions,

with loops, conditionals etc instead of state transition rules. But this sequencing does not force everyone everyday to start from the beginning if he did not complete the workflow in a single day. A company just keep the job done by each worker. When the worker -or any other worker- in charge of the same work-return to the office in the morning, he read the workflow instructions, revise their state and continue at the correct point. The representation of the flow as a sequence do not force an imperative way to execute it.

## 2.1 Inverting back the inversion of control

There is an apparent incompatibility between the event model and his stateful version -the state transition machine model- enforced by the Web architecture, and a sequential description [**?**]. As a first difference, the event model is indeed an inversion of control, where the IO handling element, the Web server, invoke the user code and not the other way around as happens in imperative programming.

But a sequence of events which update an state is a form of implicit continuation programming [**?**] where each event handler triggered by each request continues the flow -event by event- taking as imput the state generated by all the previous events.

It is possible to invert back the inversion of control so that the user program can express the flow directly in the form of a procedure which contains invocations to IO actions to/from the Web browser. Just like a console application. This does not change the way the web server and the program interact: It is the server which invokes program events, as we will see, but the programmer can express such event sequences at a higher level.

That implicit form of continuation can be expressed more explicitly in a direct, not inverted way, as it could be in a console application. This is the style of this schema program, that can be applied to a console as well to a web application:

```
(set! n1 (read-first-number))

(display-result-page n1 (read-second-number ))
```

This is a simplified form of an example in [**?**]. Here `read-first-number` is a web page which ask for the first number. If the second line is a closure and the web server has a table which matches the URLs invoked in the post of this page with the closure of the second line, then the server can invoke it, then `read-second-number` will be presented as a page asking the second number, after which `display-result-page` will show the result. The mechanism that generate the invocation machinery dispose each line in the imperative sequence as closures or continuations that can be invoked by the server. This make explicit the flux of events that in the event model was implicit and renders the event model implicit, but it is evident that the same events are working in the deep. The state in this case is in the ordinary procedure variables, that may be typed. Instead, in a event driven application, the session state is in the form of hashtables of untyped strings, since the events do not share variable scopes. Additionally the programmer does not care about creating ad-hoc identifiers for links, event handlers, state identifiers etc that must match across program files, HTML files and configuration files. He does not need to detect navigation errors produced by out of order requests, create page errors with instructions

etc. (more on that later). All this manual work inherent to the event model can introduce errors and mismatches that will only be discovered at runtime.

But, even more important, now the flow is clearly explicit because it uses the intuitions of plain imperative programming. The programmer and the maintainer is free of the uncertainty that the false freedom of the event handling style produces, when what is intended is the codification of a fixed and precise sequence of things to do.

Additionally, because each continuation will construct the event handlers of the next possible interactions and will attach them to the server, and the event handlers of past interactions will be still waiting, just in case the user goes back, everything will go well even if the user goes to the previous page in the browser, or if he clone the browser window and perform two concurrent navigations,

What is the price to pay for that? For once, since there is a continuation waiting for each alternative page, the navigation may make the memory consumption grow fast. This may force to delete continuations from memory after a certain time. That may endanger the navigability.

For some usage scenarios, we need to make the continuation state persistent, so we need it to be serializable and deserializable in order to restore them after machine restart. This is hard to do, since, even if the language has serializable closures, a continuation includes program code which is linked with many libraries -that may render the serialization huge- and it may be linked to non serializable objects of the Operating system [**?**].

The size of the serialization state is a concern, especially when we think on scalability in cluster environments, where any server can answer any request, so the state must be synchronized across machines. The input-output load involved in state sharing is the reason why REST recommends to use as little state as possible. The problem of portability of serialized continuations among machines is still harder, although it has been some advances on this area in the Scala language[**?**] or storing a portable state in the client [**?**]. This solution limites 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 workflows, specially when it exceed the mean time between failures, this can end up in loss of state. These problems have maintained the sequential style of programming Web applications out of mainstream use.

## 3 MFlow state management

In the Haskell language it is not possible, to date, to serialize a closure. But it is possible to recover the state of a application by creating a log of execution and then recover the state from it. The log contain the intermediate results of each step in the computation, not the computation itself. The machinery for log creation and recovery can be hidden in a monad. A Monad is a form of semicolon programming implemented in Haskell that introduces additional aspects in a sequence of statements[**?**]. Monads have a solid mathematical foundation in Category Theory.

When some additional aspects are needed in a plain IO procedure, then a monad transformer can be used to add it. This is what the `step` call and the `Workflow` monad transformer does[**?**] for automatic logging and recovery.

This Haskell computation below is a ordinary console application. The first line is the type or the signature of the procedure, which is a sequence of IO actions that return void:

```
proc :: IO ()
```

```
proc = do

   s <- getLine

   print $ hello  ++ s

   t <- getLine

   print $ hello2  ++ t
```

Then, the transformed procedure:

```
include Control.Workflow

include Control.Monad.IO.Class


procWF :: Workflow IO ()

proc = do

   s <- step getLine

   liftIO . print $ hello  ++ s

   t <- step getLine

   liftIO . print $ hello2  ++ t}
```

Includes added effects of logging and recovery. It is said that this new procedure is lifted to the `Workflow` monad. If we interrupt the program in the second getLine, then, at restart, the program will not ask for the first hello but will get it from the log. Then, it will print the first hello and will wait for the second line since the log ends there. The variables and the execution point are the same than when the program was interrupted. The state has been recovered from the log.

A flow of this kind can perform the same de-inversion of control than the above continuation based flow in a web application, if instead of getLine, it has a sentence `ask` that send HTML to the server and wait for the response. This procedure can be killed if the user do not respond after a certain time, since the state can be recovered from the log. Only It is necessary an application server that re-spawn the process when a request for this flow arrive.

To register a flow in the flow scheduler and to star it, MFlow uses the following statements:

```
import MFlow.Wai.Blaze.Html.All


addMessageFlows  [(URL verb string,runFlow flowName)]

wait $ run 8081 waiMessageFlow


flowName= do

          -- here the definition

          ....
```

This example uses WAI[**?**] a Web Application interface, (that has bindings for different Web server extensions, such is FastCGI). It also uses the warp server[**?**] and the Blaze-html[**?**] rendering engine. The MFlow core does not assume a particular server interface neither a particular formatting.

But this does not solve the back button problem. To do so, a form of backtracking is necessary, so that when the browser send the result of a previous page in the navigation, the procedure can go back until a previous ask sentence match with the data sent. From this moment on, the flow proceed normally. This is the purpose of the fail-back monad[**?**].

Both monads with an additional third state monad are stacked to create the Flow monad. Both aspects of state recovery and backtracking are available. These complexities are hidden to the programmer. He creates a simple imperative-looking program.

Here is how the same above program is expressed in MFlow:

```
proc=  do

     n1<- step $ ask readFirstNumber

     n2<- step $ ask readSecondNumber

     ask $ displayResultPage n1 n2
```

By means of the backtracking mechanism, this flow can respond to any number of back buttons pressed. By the logging mechanism, the state is persistent so it can be safely killed and restarted whenever the user delay the response. Since there is a single thread of execution, the memory footprint is minimal. In the example above, the log just stores the numbers asked for. Additionally the log may be human readable, and can be very useful for tracking navigation and debugging in the production environment.

### 3.1 Scalability

Because the logs grows by small update events, it is easy for two or more servers to synchronize state by interchanging update events, instead of entire states. This makes MFlow state persistence theoretically ideal for highly scalable web applications, distributed in the cloud without the problems of continuation-based flows.

The state can even be stored in the browser. That would render the server applications stateless and, hence, very scalable and full REST compliant. However to achieve this scalability is necessary to get rid of the backtracking mechanism and substitute it by a restart of the flow with the log state attached to every request, in order to synchronize on-request. That would make long navigations problematic. That is the solution adopted by Peter Thiemann in the Haskell WASH[**?**] framework.

Instead, in-server event sharing and backtracking is a general mechanism for synchronization of processes and data in a scalable, fault tolerant cloud configuration, where the web navigation is one more stateful process. The author rediscovered the logging mechanism in trying to solve the problem of saving and sharing state of any computation. There are many studies on recovery and synchronization based on event systems and logging, but they assume a global state and an event-driven program style[**?**]. The Workflow approach does not suffer these limitations.

### 3.2 Page addressability and navigation maps

The difference between a navigation map and a flow is that a navigation map usually has many branches, connected by links. There are no state changes and each page can be addressed by its own URL directly. If the Web framework each page has an URLs and the links return back to the flow (more on that later), then we can express a navigation map in a flow procedure. So there may be a unification of navigation maps and flows, so all the navigation tree can be expressed in a single sequence with branches depending on the links pressed. This routing, in the forrm of a tree in code is a more structured, flexible and expressive to express a navigation than a plain map table of URLs versus event handlers.

Consider this navigation:

```
data Pages= MyPage | OtherPage ...

pages=  do

     r<- ask landingPage

     case r of

       MyPage  -> do

             user <- ask loginPage

             ask $ userPage user

             ...
```

```
       OtherPage -> ...
```

Here there is no state persistence, since `step` is not used. It is supposed that `landingPage` has a link to `myPage` . If he presses this link, the `loginPage` is sent and, after login, the `userPage` is presented. The links are typed and the compiler will show an error if the programer does use a link not of the correct type.

A machine state transition system based on rules such is Javaserver Faces Flow, would express this as follows:

```
<navigation-rule>

    <from-view-id>/landingPage.jsp</from-view-id>

    <navigation-case>

        <from-outcome>login</from-outcome>

        <to-view-id>/login.jsp</to-view-id>

    </navigation-case>

        <from-outcome>userPage</from-outcome>

        <to-view-id>/userPage.jsp</to-view-id>

    </navigation-case>

</navigation-rule>

<navigation-rule>

    <from-view-id>/login.jsp</from-view-id>

    <navigation-case>

         <from-action>#{GetNameBean.helloAction}</from-action>

           <from-outcome>success</from-outcome>

           <to-view-id>/userPage.jsp</to-view-id>

    </navigation-case>
```

```
</navigation-rule>
```

In the jsp file, this must appear:

```
<h:commandButton id="submit"

    action="#{someBean.eitherLoginOrUserPage}" value="Submit" />
```

`someBean.java` file must contain de code of `eitherLoginOrUserPage`. This is only the navigation configuration, not the java code neither the rendering. Note that the specification is full of identifier strings that are filenames, outcome string and so on which references java identifiers, java files, jsp files and other configurations. There is plenty of opportunities for errors.

However, the advantage of this event model is that `landingPage`, `loginPage` and `personalPage` can be invoked out of the flow. I can store a link to `userPage` in my notepad. when I press the link, the page is invoked. The disadvantage is that if the login session has expired or if the user passes the link to a friend who has not logged, the state expected by `userPage` is not what is expected by the page handler (already logged) so the programmer has to take care of detecting and present every kind of error of this type and redirect the user to the appropriate page. This means more error pages, more java code and more navigation rules in the XML.

Instead, the MFlow code is far more simple. But It seems, from the imperative look, that every navigation has to start from the beginning in a sequence to get the userPage.But this is not necessary. It is possible to express a direct link to userPage in MFlow. The URL is:

```
http:://server/proc?p1=MyPage
```

And there are no page errors; If the user is logged (the browser has the user cookie) `userPage` will be presented. Otherwise, it will show the login page that make sure to log in correctly before going to `userPage`. The login page will appear again and again until a valid login/password is entered. Just the behaviour that we would expect by default.

The reason why ask pass trough `landingPage` is because, before sending anything, it check for the parameters in the GET request. If he find the respose that he needs, he return inmediately without sending anything to the user. In this case, the first ask find `p1=myPage`, that is a parameter and a value of the correct type that `landingPage` expect, that is what it was asking for. Then the next ask is for the login, so it proceed as explained above. The procedure this time works like a type safe route selector instead of as a flow procedure.

A single URL request can satisfy various ask statements if the parameters codified in the request satisfy what is expected. So any linked page can be addressed in a single URL.

However, currently the names of the parameters in the URL are automatically generated so they are not meaningful. Additionally, currently only URLs with a single parameter are generated automatically,

more parameters can be added by hand. REST style addresses without abitrary parameter names would be a better solution. This is an area for improvement.

As before, the backtracking mechanism works as well; When the flow receives an HTTP request that do not match with the current execution state, the process automatically backtrack to the ask statement of the flow that handle it. Then the flow proceed from this point on. So if the user introduces an URL of a previous step or another point in the navigation, the procedure will backtrack the navigation tree until some parameters match, then will advance to the point in the navigation until all the parameters match.

However, if the URL has no parameters, the user receives the page corresponding to the current state instead of going to the root of the navigation. This is intended to avoid undoing workflows that are not pure navigations, but stateful, and involve state data that may be onerous for the user to redo if they are undone inadvertedly. For example, shopping carts, multi step reservation wizards etc.

### 3.3 Roll back state on back button pressed

The MFlow strategy for handling request is to match the request with the ask statements in the execution tree, starting from the current state. In the matching process, the state is modified accordingly.

This automatic rollback of state when the navigation goes back, thanks to the fail-back monad, and the update of it from request paramenters when the application goes forward is a nice property that avoid the creation of ad hoc state management. This may be the skeleton of a flow with a shopping page that manage a shopping cart

```
shop products= do

   setHeader $ \html -> html << body << p << ("A persistent shopping cart skeleton <> html)

   setTimeouts 120 (30*24*60*60)

   loop emptyCart

   where

   loop cart=  do

      r <- step . ask $ products <** showCart cart <+> wlink () << p <<  exit shop

    case r of

             (Just bought,_) -> loop cart

             _ -> breturn cart
```

Here there are some interface elements that will be explained later. `setTimeouts` establishes how long the process is running in memory and how long the serialized session data is recorded, respectively.

each product selected on each ask request is stored in the log. If, after two minutes, the user select another product, the process will be restarted and will recover the shopping cart state by re-executing the loop, taking as imput the log content until the log is finished. Unless the user does not enter for a month, in which case, the log will be deleted and the shopping cart will appear empty. When the user press the `"exit shop"` link, the flow will return the shop cart to the calling flow.

In the previous example, it is noteworthy that, if the user is adding products to the shopping cart, when he press the back button, the previous page will appear, with one product less in the cart. In this page, when he select other product and send the request, the application will backtrack one step in the loop, so the shopping cart will roll back from the last transaction and the shopping cart will reflect the page that the user is seeing. This synchronization of state and pages occurs on every case thanks to the backtracking mechanism.

### 3.4 Configuration

Sometimes a navigation is preceded by a configuration step. For example, this flow ask to choose the prefered skin before going to the rest of the navigation:

```
skins= do

   setHeader $  html . body

   setTimeouts 120 (365*24*60*60)

   ask loginPage

   skin <- step . ask $   p << "choose skin"

                      ++> wlink Normal << p << "normal"

                     <|> wlink Blue    << p << "blue"

                     <|> wlink Red     << p << "red"



   step $ restOfTheFlow skin


   where


   restOfTheFlow skin = do

        r  <- ask $  p << ("you choosen the skin " ++ show skin)
```

```
            ++> wlink "change" << MF.div << "Change the skin"

            <++ br

            <|> wlink "doOther" << p << "other things"


      case r of

        "change" -> breturn ()

        _ ->do

               ask $   p << "other things"

                   ++> a ! href (fromString "http://www.google.com") << p << "google"

                   ++> br

                   ++> wlink () << p << "press here to return to the salutation"
```

In the navigation, if the user does not click anything after 120 seconds , the process will shut down. But because the skin value is stored in the log (by `step`) when the flow restart in response to a user request, it will recover the skin from the log. So the first page presented will show the skin chosen the first time. Or else, the adequate page will be sent, given the parameters in the request.

If the user want to choose another skin, he will press the "Change the skin" button. Then `restOfTheFlow` will exit and the flow will be executed again, since the flows are executed in a loop, without being restarted. Then the log will reflect two elections of skin and two void values (from the second step). When restart, the process will loop two times and recover the last skin. Unless the flow is not invoked during a year, in which case the state will be deleted and the process will restart anew.

Note the use of `breturn` instead of `return`. The former tell the fail-back monad that, when back-tracking, restOfTheflow must be called back, because it contains ask statements which correspond to previous pages towards which the browser would navigate back.

| **Web framework** | **how it works** | **how state/navigation works** | **Problems** |
|---|---|---|---|
| Event model - State transition machine model | | | |
| MVC: ASPX, JSP, Rails, node.js | MVC, mostly stateless. Page oriented | Add session state as hash tables. flash objects(Rails) | Event-based: spaguetty code. back button usually does not work (roll-back state hardly possible) |
| Struts, JSF | MVC page and session oriented | Add basic configurable navigation (XML) | Event based, complex, no checks at compile time. No back button support |
| Seam, Spring Web Flow | Flow oriented via XML config and language extensions | add conversations, subflows, back button support | Event based, complexity,No checks at compile time |
| Apache wicket | MVC with interface in java objects+HTML | add serializable page state. support back button (with appropriate browser configuration) | Event based. No back button detection,serialized state may be big |
| Procedural, sequential style | | | |
| Coccon, seaside, ocsigen | Continuation-based, flow in a single procedude (javascript,smalltalk, Ocaml respectively) | serializable execution state, support back button | compile time checks vary. serialized state may be big. no bookmarkable URLs, scalability problems |
| WASH | Monad for creation-replay of the log, flow in a single procedure (haskell) | Log recreate the execution state | Each request implies a replay of the log. log in the browser can be hackedv |
| MFlow | Monad transformers for logging/recovery and navigation backtracking, flow in a single Procedure(Haskell), checked at runtime. | Log may recreate the execution state. Execution state stay in memory. Back button supported by backtracking. bookmarkable URLs | Under test |

**Flow state management on different Web Framewors**

# 4 User interface in MFlow

The document oriented interface or the Web architecture, HTML, make it simple, flexible and powerful but his typelessness and the lack of modularity renders difficult the development of safe interactive applications.

This would be a minimal version of the pages of the example that appears in the navigation section:

```
landingPage= p << "hi, this is the landing page"

               ++> wlink MyPage << "press here to go to your page"


loginPage= b << "please login" ++> userWidget Nothing userLogin


userPage user = p << ("this is your user page for " ++ user)

                  ++> wlink () << p << "press to go to landingPage"
```

Here p is the the HTML tag <p> and the text in the right is the HTML enclosed on it. In this case, a string. It is possible to use plain HTML, for the formatting using a XML preprocessing extension of Haskell or in general any formatting since MFlow does not assume a particular formatting library. However this example, like most of the cases it is used the blaze-html library, a fast Html combinator library.

The operator ++> adds rendering to a widget. A widget is a element rendered in the `view` format, generated in the monad `m` which return to the server data of type `a`.

The signature of a widget is: `View view m a`. Therefore ++> has the signature:

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

The first argument is the rendering. The second is the typed element (widget) and the result is a widget

The types of the three pages above (deduced by the debugger ghci) are:

```
>:t landingPage

landingPage :: View Html IO Pages
```

```
>:t loginPage

loginPage :: View Html IO String


>:t userPage

userPage  :: View Html IO ()
```

*View v m a* is a parameterized data structure that contains formatting of type v, has been generated by running a monad m and return data of type a. *View v m* is itself an instance of the Haskell *Applicative* class. This is a structure close to a monad with nice compositional operators. that means that the widgets can be combined using applicative combinators. The idea of applicative combinator to generate composable, type safe User interfaces is based on formlets[**?**][**?**] . The MFlow implementation add embedded formatting, modifiers, additional combinators, link management and callbacks, besides validation.

For example `landingPage` can have various links instead of one:

```
data Page= MyPage | OtherPage | AndYetAnother deriving(Typeable, Read, Show)


landingPage= p << "hi, this is the landing page"

          ++> wlink MyPage        << p << "press here to go to your page"

          <|> wlink OtherPage     << p << or press here if you like to go to other page

         <|> wlink AndYetAnother << p << or this other
```

the `<|>` applicative operator return either the left expression or the right one. The signature is:
`(<|>) ::  Alternative f => f a -> f a -> f a`
Which is the most general form, but applied to View v m a data, it get two alternatives and return the rendering of both, but only one of the two possible values. This behaviour is defined in the Alternative instance of `View v m` :

```
(<|>) :: View v m a -> View v m a -> View v m a
```

In this case, the type variable f is `View v m`

`wlink` is a typed link. It generates a normal HTML link where the first argument is the value returned and the second is the content. Unlike a normal link which may point to any page or even any site, wlink ever return the value to the current flow, so after the link, we stay in the local scope.

The returned value is typed. it may be of any kind as long as it can be read from a String and shown as String.

Since the pages are combinations of typed elements called widgets, their combinations are also widgets. This page has a form field above and show some links below:

```
variousStuff=  p << give me a number ++> getInt Nothing `validate` lessThan5

            <+> p << or else

           ++> wlink True << p << press this link

           <** p << or this other

           ++> (wlink () << p << or this other with a callback `waction` someFlow)



            where


            lessThan5 n=

               return if n < 5

                        then Nothing

                        else Just (p << "Please enter a number less than 5"))
```

This widget has the type:

```
variousStuff :: View Html m  (Maybe Int, Maybe Bool)
```

Since it may return either the `Int` of the input formulary generated by `getInt` in a POST request or a Boolean `True` if he press the fist link, in a get request. The third link, when pressed, its result will be ignored.

The reason of this combination is because the binary operator `<+>` join the result of two widgets in a 2-tuple, while `<**` ignores the result of the element on its right, the third widget (since the rest is formatting added with `++>` and `<++` operators). But this last element is a link that has an `waction` which adds a callback that will be executed if the link is pressed.

The signature of waction is:

```
waction :: View view m a -> (a -> FlowM view m b) -> View view m b
```

The first argument is a widget that return a value of type a, the second is a Flow that get the value returned by the first widget and execute itself. In this case the result of the flow is ignored because it is composed trough the `<**` combinator.

With `waction` it is possible the creation of self contained widgets that have their own server handler code. A callback defined with `waction`, is invoked when the widget is activated, in the case above, when the link is clicked. This callback could perform simple server actions or, additionally it can contain further user interactions with their own `ask` statements. At the end of these interactions, the user interface return to the page where the action was placed.

The first link has a validation `validate`. If the link is pressed the validator will get the return of the widget on its left (`getInt` in this case) and execute the expression on the right (check that is less than 5) . If the result is a error message, the page will be presented again with the message. If the validation return Nothing, `ask` will return the number entered.

Note that the form field does not need an explicit Form-action statemment as is usual in HTML formatting libraries. It is added automatically by the user interface when needed.

## 4.1 Other user interface features

The current version of MFlow support AJAX. It can be combined AJAX with widgets to generate widgets with auto-completion forms, or widgets that act as containers of a variable numbers of other widgets. There are widgets for content management and multilanguage edition facilities. There is a module which implement these dynamic widgets (`MFlow.Forms.Widgets`) that show how composable the interface architecture is.

Some of these widgets require additional resources like CSS styles, javaScript source code etc. To express these requirements, MFlow has a simple management of these requirements that make sure that the final composed page includes each resource once.

The rendering of a widget can be cached. This speed up and reduces the load produced for page generation (The whole page is a widget). The statement:

```
wcached key time widget
```

Caches a widget during a time in seconds identified with a key.

## 5 Database and persistence

MFlow uses TCache[**?**], a transactional cache with configurable persistence, which handle haskell records in memory and support STM transactional updates.This is much faster than database transactions. It also write coherent states in persistent storage. Each record can have its own mechanism of persistence, defined by the programmer. I can be a SQL query or a file. Writing to files is supported by default to every serializable data. This speed up the prototyping. TCache support full text search and has a relationa-like query language based on field record names besides search by key. The log mechanism uses also TCache, so the user sessions state and other data state may be coherent, if TCache is used for the application data.

## 6 Testing MFlow applications

The strong typing allows for static checking at compile time, but also can be used to generate test cases and different load conditions.

There is a test version of ask that instead of interacting with the user, generates a aleatory response of this type, so that the whole flow can be executed to check for property assertion failures. in the style of QuickCheck[**?**]. Suppose that we have this program:

```
module Main where

import MFlow.Wai.Blaze.Html.All

main= do

   addMessageFlows  [("" , runFlow flow)]

   wait $ run 8081 waiMessageFlow

#endif



flow= do

    r <- ask userInput

    s <- somethingComplex r

    t <- somethingComplex s

    doOther t
```

We need to test somethingComplex and somethingComplex since we know some invariants that they must accomplish between the input and the output: This would be the code:

```
{-# OPTIONS  -XCPP #-}

#define TEST

module Main where

import MFlow.Wai.Blaze.Html.All

#ifdef TEST

  hiding (ask)

import MFlow.Forms.Test

import Control.Monad

#endif

main= do

#ifdef TEST

   replicateM_ 100 . runTest1 $ runFlowOnce flow

#else

   addMessageFlows  [("" , runFlow flow)]

   wait $ run 8081 waiMessageFlow

#endif

flow= do

    r <- ask userInput

    s <- somethingComplex r

    t <- somethingComplex s  `verify` (invariant  r s, 1 not satisfied)

    doOther t                `verify` (invariant s t, 2 not satisfied)
```

Here the transformed program can run in test mode or in normal mode simply by not defining TEST in the second line.

In test mode, ask is changed by the definition in the module MFlow.Forms.Text, which fabricate aleatory values for most of the standard datatypes. and containers, just like the package quickcheck. For this purpose any datatype that is instance of the classes *Bounded* and *Enum* are instances of the *Generate* class, which is the class used to produce random values.

Verify check the invariant assertions. When they return false, varify generate a user exception with the error message.

the initiation of the test is done in the line:

```
replicateM 100 . runTest1 $ runFlowOnce flow
```

Here, replicateM repeat the test for 100 times. runTest1 executes it in a single thread, sequentially, and

runFlowOnce just executes the flow and exit, while runFlow would restart again the flow forever.

If we want to execute the 100 threads simultaneously, the execution line should be:

```
runTest [(100 ,runFlowOnce flow)]
```

*runTest* may be used to check various flows of the application in different load conditions.

Sometimes, instead of a random value, it is necessary to enter a concrete value.

*askt* return a value depending on the number of times that the flow has been executed. The first parameter is the programmer-defined expression that get the execution number and generate the value to be returned.

```
askt :: (Int ->a) -> View v m a -> FlowM v m a
```

This allows for the orchestration of complex testing scenarios where different simulated users perform different actions.

instead of askt, inject can override the user interaction of ask:

```
MFlow.Forms.ask somePage          `inject` someValues
```

Is semantically equivalent to

```
askt someValues somePage
```

However, askt , like ask, forces the rendering of the page, while inject dont. This is important in load testing

One example of this is the user login widget and in general any widget with actions, They execute code that has side effects in the execution of the flow. For this purpose waction and all the primitives in MFlow.Forms that use actions are redefined in the MFlow.Forms.Test module.

In general *MFlow.Forms.Test.ask* can be used to produce random executions, a kind of monkey tests to check the invariants and simulate different loads in dry-dock conditions. Alternatively, askt and inject can be used to perform more controlled paths of execution, in which different simulated users return different responses according with a predefined sequence.

The GHC profiler can be used to obtain the coverage of the tests performed. If there is not enough coverage with the random tests, then askt and inject can be used to generate more appropriate test data.

## 7 Conclusions and future work

The MFlow development demonstrates the power of monadic computation for creating Web flows and Web navigations in a concise, intuitive and maintainable way while reducing drastically the noise and the error ratio in web programming. This monadic style, together with the composability of the pages, in a strongly typed language such is Haskell, makes the navigation verifiable at compile time, and this facilitates the dynamic testing of the whole navigation too, while the scalability and navigability is not compromised.

One important task in future work is to realize the theoretical scalability of the architecture by developping a synchronization mechanism for clustering MFlow servers. Another inprovement would be better URLs in the REST style.

Some small issues of navigation remain to be fixed. Some more experience is needed to realize the full potential and reduce the complexity exposed to the programmer.

## 8 Bibliography