Reflex Documentation

Release 0.5

Divam

Dec 09, 2018

Contents

If you are new to reflex, then check out some *Tutorials*

This documentation is a work-in-progress, so some of the sections are incomplete or empty.

Please feel free to contribute to this Documentation by opening a pull-request here

Contents:

CHAPTER 1

Installation

1.1 Overview

The essential components required for developing `reflex` based application are

1. GHC or GHCJS

   If you are building a web application with `reflex-dom` then you need `ghcjs` to create JavaScript output. With `ghc` you can use the `reflex-dom` to create a `webkit` based desktop or mobile app.

2. Reflex library

   The current supported `reflex` and `reflex-dom` packages (version 0.5 and 0.4 respectively) are available only through Github, as they are not yet published on Hackage.

To quickly get started with developing full-stack web apps using reflex, Obelisk is the recommended method.

For a more advanced usage, `reflex-platform` is the recommended method.

1.2 Obelisk

Obelisk is a command line tool and a set of libraries to make it easy to get started with full-stack web development with `reflex`. It includes features like

- Automatic installation of latest `reflex`, `ghc`, `ghcjs` and haskell dependencies/libraries using `nix`.
- Create a skeleton project with
  - frontend using `reflex-dom`
  - backend using `snap`, with pre-rendering support.
- Development workflow related commands like
  - `ob run` to automatically rebuild your application on a file write. It also serves the frontend using `jsaddle-warp`, to help in faster development.

  - `ob repl` to provide a `ghci` shell.
  - `ob deploy` to help in deployment to EC2, and create optimised/minified `js`.
  - Create android app `.apk` file, and `iOS` app with `nix` commands.
- Routing library `obelisk-route` to create type safe routes

  see *obelisk-route*

1.3 reflex-platform

reflex-platform is a collection of `nix` expressions and scripts to provide `ghc`, `ghcjs` and a curated set of packages for use with `reflex-dom`.

This includes a specially modified `text` package which internally uses the JS string. The performance of this `text` package is significantly better on the browser.

Note: GHCJS uses a lot of memory during compilation. 16GB of memory is recommended, with 8GB being pretty close to bare minimum.

1.4 reflex-project-skeleton

reflex-project-skeleton is a bare repository which uses `reflex-platform` to provide a nice development enviroment, with both the power of `nix` (for binary cache of dependencies) and `cabal new-*` commands (for incremental builds).

For a project with both a `backend` and `frontend` components, this is the recommended setup.

See README and `reflex-project-skeleton/reflex-platform/project/default.nix` for usage details.

This also supports cross-compiling the `frontend` part to android and iOS platforms!

The following contains information of creating a project-skeleton from scratch and also more details about its working.

https://github.com/reflex-frp/reflex-platform/blob/develop/docs/project-development.md

1.5 Minimal dev-env using reflex-platform

Please refer to reflex-platform' README

The `try-reflex` script can create a development environment with `ghc` and `ghcjs`. You can use this to have a quick starting setup to compile code-snippets and smaller projects.

When using this for the first time, setup can take considerable time to download all the dependencies from the binary cache.

But for a non-trivial project it is recommended to use `cabal`.

1.5.1 Using cabal with `reflex-platform`

If you dont have a project with cabal file then use `cabal init` to create one.

Then use the `workon` script from reflex-platform to create a development environment (nix-shell) according to the dependencies specified in cabal file.

```
$ ~/reflex-platform/scripts/work-on ghcjs ./your-project

# or just "cabal configure" if working on ghc
<nix-shell> $ cabal configure --ghcjs
<nix-shell> $ cabal build
```

Note: The `cabal update` and `cabal install` commands should not be used, as the task of fetching and installing dependecies is done by `nix`.

This will use your package's cabal file to determine dependencies. If you have a `default.nix`, it will use that instead. Note that your project's path must include at least one slash (`/`) so that work-on can detect that it is a path, rather than a package name.

This will give you the exact environment needed to work with the given package and platform, rather than the general-purpose environment provided by the Reflex Platform.

You can replace ghcjs with ghc to hack on the native GHC version of the package (including with GHCi if you want). You can also use a package name instead of a path, which will drop you into the standard build environment of that package; this works even if you don't yet have the source for that package.

1.5.2 Add reflex-platform to project

Since the build environment is dependent on the reflex-platform, it is important to keep this dependency as a part of the project. Moreover the version of libraries will change with time in the reflex-platform so it is important to keep a reference to the reflex-platform' "version" which has been used to build the project.

The simplest way to do this is to create a submodule in your project, and use the `workon` script from it to create a shell with proper build dependencies.

Assuming you are using git for versioning:

```
git submodule add https://github.com/reflex-frp/reflex-platform

# Then use the workon script to get the nix-shell
./reflex-platform/scripts/work-on ghcjs ./.
```

A better way is to use the `nix` commands, see *reflex-project-skeleton* or project-development.md

1.6 Local Haddock and Hoogle

Local hoogle server can be run from the shell created for development environment by

```
$ hoogle server --local
```

To obtain a shell; if you are using

- `reflex-project-skeleton` or `obelisk` then do:

```
$ nix-shell -A shells.ghc
```

- `reflex-platform`: Create a shell from either `try-reflex` or `workon`:

From this shell the path of local haddock documentation can also be obtained using:

```
# or use ghcjs-pkg
$ ghc-pkg field <package> haddock-html
```

1.7 GHCi / ghcid with `jsaddle-warp`

- `reflex-project-skeleton`:

  For a simple ghci repl do:

  ```
  $ ./cabal new-repl frontend
  ```

  or create a shell using nix-build:

  ```
  $ nix-shell -A shells.ghc
  $ cabal new-repl frontend
  ```

  See the README of the project for more details

  For `ghcid` you might have to run the ghcid from the frontend directory so that it detects the `src` folder correctly

  ```
  $ cd frontend; ghcid -c "cd ..; ./cabal new-repl frontend"
  ```

- `reflex-platform`:

  Create a shell from either `try-reflex` or `workon` and use the regular `cabal repl` or `ghcid` commands from your project root.

With `jsaddle-warp` package you can run your app in browser without using `ghcjs`. You need to modify the `main` like the code below. Then you can run it via `ghci` or `ghcid`, and open your application from browser via http://127.0.0.1:3911/:

```
module Main where

import Reflex.Dom.Core
import Language.Javascript.JSaddle.Warp

main = run 3911 $ mainWidget $ text "hello"
```

This should works fine on Chrome/Chromium, but might not work with firefox.

1.8 IDE tools support

Instructions for setting emacs/spacemacs are here : https://github.com/reflex-frp/reflex-platform/pull/237

1.9 Contributing to Reflex

To contribute to `reflex` or `reflex-dom` packages, it is best to use `reflex-platform`. The `hack-on` script will checkout the source of the package in your local `reflex-platform` directory as a git submodule, and use it to provide the development environment.:

```
$ ./scripts/hack-on reflex -- or reflex-dom
```

You can then patch the source code, test your changes and send a PR from the git submodule.

Todo: Add ways to use reflex without nix / reflex-plarform

CHAPTER 2

Overview

`reflex`

provides the Functional Reactive Programming (FRP) implementation.

This is the base for `reflex-dom` but is independent of the DOM / web interface design code, and can be used in many other applications.

See Quick Ref

`reflex-dom-core` and `reflex-dom`

provides a APIs for constructing DOM widgets, do websocket / XHR requests, etc.

Most of the functionality is part of the `reflex-dom-core` package.

See Quick Ref

2.1 Reflex Basics

The `reflex` package provides the foundation for the FRP architecture. It consists of many type class definitions and their implementations, and the most important type class in this package is `Reflex`.

The three main types to understand in `Reflex` are Behavior, Event, and Dynamic.

1. Behavior

   A container for a value that can change over time. 'Behavior's can be sampled at will, but it is not possible to be notified when they change

   `Behavior t a` abstracts the idea of a value `a` at all points in time. It must be defined for all points in time and at any point you can look at the behavior and sample its value. If you need to represent something that does not have a value at all points in time, you should probably use `Behavior t (Maybe a)`.

2. Event

`Event t a` abstracts the idea of something that occurs or is updated at discrete points in time. An example might be button clicks which would be `Event t ()`, or key presses which might be `Event t Char`. Events are push oriented, i.e. they tell you when the value changes.

3. Dynamic

   `Dynamic t a` is an abstraction that has a value at all points in time AND can notify you when its value is updated. They are essentially a tuple of an Event and a Behavior boxed up in a way that keeps everything consistent. They can be viewed as a step function over time, with the value changing at every occurrence.

   We use `Dynamic` in `reflex-dom` in a lot of places where you might expect to use `Behavior` in various other FRP settings because the DOM API is fundamentally push-based: you pretty much have to explicitly tell things to update, the browser isn't asking our program which DOM tree should be displayed, so we have to know when the values change.

The `t` type parameter indicates which *timeline* is in use. Timelines are fully-independent FRP contexts, and the type of the timeline determines the FRP engine to be used. This is passed to every FRP-enabled datatypes and it ensures that wires don't get crossed if a single program uses Reflex in multiple different contexts.

In reactive programming you have various sources of events which have to be utilised for providing responses. For example when user clicks a button, this event can have various different reponses depending upon the context or more specifically the state of the application.

The response to an event in most cases will do some changes like modify DOM, communicate with server or change the internal state of application.

In Reflex this response can be expressed or implemented by

1. Firing another `Event`.
2. Modification of a `Dynamic` Value.

Note that there are no explicit callbacks or function calls in response to the incoming events. Instead there is generation of new Events and modification of Dynamic values. These Event and Dynamic values are then propagated to widgets which provide the appropriate response to the event.

Since this propagation of `Event`/`Dynamic` values can be cyclic, it can be thought as an Event propagation graph.

For more details see *Event*

2.2 Architecture of a Reflex-DOM Application

A typical Reflex-DOM application consists of widgets, and some glue code to *connect* the widgets together.

Widget can be thought as a DOM Structure which has the capability to modify its contents in response to events or based on some dynamic values. It can also contain structures like input fields which can generate events. Moreover user interaction events like mouse clicks can also be captured from the widgets.

Additionally there are some pieces of code (equivalent to a controller) which does not have a Dom view, but can process input events, maintain a state and generate output events or dynamic values.

These controller can encapsulate the logic behind handling of incoming events, they can transform (using Functor) or filter (using Applicative) these events and dynamic values as per the need. This way user has the power to create custom event flows which can be either restricted/local to some widgets or span the entire app.

Reflex does not enforce a strict separation between these two, and user has the complete flexibility to choose a suitable design.

Sometimes it is a good practice to partition the code in these sub-categories, like implementing the main business logic in a pure function or a state machine, and the view in a separate module.

But many times it is better to have independent self-contained widgets, thereby reducing the complexity of propagating trivial events from view to the controller.

Also see the reddit thread how to structure a reflex application.

2.3 DOM Creation

The HTML DOM is constructed as a tree of "Objects" in which both the "sequence" of objects in the tree and their "heirarchy" has to be specified.

In `reflex-dom`, DOM creation works in a Monad `DomBuilder`. Since it is monadic, the sequence of function calls directly correspond to the sequence of DOM elements. To create heirarchy a lot of basic widgets take an addition argument of type (m a) which will be nested inside it.

For example:

```
let myText = do -- Specifies sequence
      el "h1" (text "Header") -- Nesting
      text "Content"

el "div" myText -- Nesting
```

2.4 View-Controller Architecture

Separate APIs to manage events and to render view

```
-- button_and_textvisibility.hs
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE LambdaCase #-}

-- This code demonstrates use of an event to create dynamic values
-- Simple flow of an event from one widget to another.
main = mainWidget $ do

  -- View Widget to Generate Events
  -- button widget is defined in library, it creates a simple button
  evClick <- button "Click Me!"

  -- Controller
  -- Handle events and create a 'Dynamic t Bool' value
  -- This toggles the visibility when the button is pressed
  isVisible <- foldDyn (\_ b -> not b) False evClick

  -- View
  -- This is a simple widget that takes a 'Dynamic t Bool' as input
  textWithDynamicVisibility isVisible

  return ()

-- This widget takes the input value of visibility
-- and creates a view based on that
textWithDynamicVisibility isVisible = do
  let dynAttr = ffor isVisible
                (\case
```

(continues on next page)

(continued from previous page)

```
                  True -> ("style" =: "")
                  False -> ("style" =: "display: none;"))

  elDynAttr "div" dynAttr $
    text "Click the button again to make me disappear!"
```

2.5 Widgets Interacting Together

By using the recursive-do notation we can connect the widgets together. This is a simple example of creating a cicular Event-Dynamic propagation.:

```
-- button_and_textvisibility_2.hs
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE RecursiveDo #-} -- This is important!

-- This code demonstrates use of an event to create dynamic values
-- Circular flow of Event/Dynamic using Recursive-do syntax
main = mainWidget $ do

  rec
    -- Controller
    -- Handle events and create a 'Dynamic t Bool' value
    -- This toggles the visibility when the button is pressed
    isVisible <- foldDyn (\_ b -> not b) False evClick

    -- View
    -- This widget creates the button and its click event,
    -- The click event is propagated to the controller
    evClick <- textWithDynamicVisibility isVisible

  return ()

-- This widget takes the input value of visibility
-- and creates a view based on that
textWithDynamicVisibility isVisible = do
  -- View Widget to Generate Events
  -- button widget is defined in library, it creates a simple button
  evClick <- button "Click Me!"

  let dynAttr = ffor isVisible
                (\case
                  True -> ("style" =: "")
                  False -> ("style" =: "display: none;"))

  elDynAttr "div" dynAttr $
    text "Click the button again to make me disappear!"

  return evClick
```

As you can see this helps to completely separate the View widget and controller code.

But the real power of recursive-do notation can be utilised in creating more complex *Integrated* widgets as desribed in the next section.

2.6 Integrated Widget Architecture

In Reflex it is possible to combine the view and controller part of the code to create integrated widgets which can be plugged in easily in your app.

Example of a widget which is self-contained. This widget creates a simple text field, which can be edited by clicking on it. Source:

```
editInPlace
    :: MonadWidget t m
    => Behavior t Bool
    -- ^ Whether or not click-to-edit is enabled
    -> Dynamic t String
    -- ^ The definitive value of the thing being edited
    -> m (Event t String)
    -- ^ Event that fires when the text is edited
```

Quoting mightybyte

> This defines the entire interface to this widget. What makes this example particularly interesting is that the widget has to maintain some internal state in order to implement its functionality. Namely, it has to keep track of the Viewing/Editing state. Reflex allows widgets to handle this kind of state internally without needing to add it to some top-level application-wide state object. This hugely improves composability and ultimately allows you to build GUI apps just like you would any other Haskell app–main is your overarching top-level function and then you split out whatever widgets it makes sense to split out. Your guide for splitting things will probably be that you want to find pieces that are loosely connected to everything else in terms of inputs and ouputs and make them their own function.

2.7 Overview of `ghcjs` and `jsaddle` Packages

`ghcjs`

Is the compiler, like `ghc`.

`ghcjs-dom`

Is the library which provides the interface APIs to work with DOM and Web APIs, either on a browser (by compiling with `ghcjs`) or natively using webkitgtk (when compiled with `ghc`)

Applications should use the `ghcjs-dom` package and the `GHCJS.DOM.*` modules it contains; to get the best mix of portability and performance (rather than using the `jsaddle-dom`, `ghcjs-dom-jsaddle` and `ghcjs-dom-jsffi` directly).

Note: The below package descriptions are provided for information only. For using reflex-dom in applications ghcjs-dom should be sufficient.

`ghcjs-base`

Is the base library for `ghcjs` for JavaScript interaction and marshalling

This package should be included in cabal only if using `ghcjs` by adding this

```
if impl(ghcjs)
  build-depends: ghcjs-base
```

`jsaddle`

JavaScript interface that works with `ghcjs` or `ghc`.

It provides a set of APIs to do arbitrary JS execution in a type-safe manner.

- If compiled with `ghc` on native platforms like WebKitGtk, WKWebView on iOS / macOS or Android using JNI.

  It uses a *JavaScript command interpreter* for each of the different targets.

- If compiled with `ghc` using `jsaddle-warp` and running on browser.

  The JS commands are encoded in the executable running on native platform, and sent to the browser for execution using a websocket connection.

- If compiled with `ghcjs`, it uses some JSFFI calls to execute the functions indirectly.

  Note: this has poor performance compared to calling the DOM APIs directly through `ghcjs-dom-ffi` as the DOM API calls are wrapped in an execution script.

See README for more details.

`ghcjs-base` and `jsaddle` form the base for these packages

`ghcjs-dom-ffi`

This package implements the entire DOM/Web API interface as direct JSFFI calls.

On browser this is the most optimal way to execute DOM related actions.

`ghcjs-dom-jsaddle` and `jsaddle-dom`

This provides the DOM/Web API interface using `jsaddle`

CHAPTER 3

Reflex

The `reflex` library provides the foundation Classes and their implementation APIs to do Functional Reactive Programming. This is independent of the DOM creation code, and can be used to implement FRP architecture in non-web related apps also.

The Quick Ref provides a really nice overview of its APIs.

3.1 FRP Basics

In order to leverage the full power of reflex, one has to effectively use the ability to create an Event propagation graphs, and use it to model the business logic. This guide gives an overview of basics and various useful techniques.

Also see *Reflex Basics*

3.1.1 `Event`

Creation

newTriggerEvent

Is used to inject value in the `reflex` event-propagation-graph from outside using IO action:

```
newTriggerEvent :: TriggerEvent t m
  => m (Event t a     -- Event triggered by fun
       , a -> IO ()) -- fun
```

`newTriggerEvent` can also be used to break a big `rec` block.:

```
rec
  ev1 <- widget1 evN
```

(continues on next page)

(continued from previous page)

```
  ..
  ..

  evN <- widgetN evN_1
```

In this the `widgetN` and many other widgets in-between can be pulled outside the `rec` block:

```
(evN, evNIOAction) <- newTriggerEvent

ev1 <- widget1 evN

..
..

evN' <- widgetN evN_1

performEvent $ ((\v -> liftIO $ evNIOAction v) <$> evN')
```

From `Dynamic`

By calling `updated` on a `Dynamic` value one can obtain the event when its value changes.:

```
updated :: (Reflex t) => Dynamic t a -> Event t a
```

Repeating Events

Using APIs from `Reflex.Time` one can create repeating events.:

```
tickLossy :: (_)
  => NominalDiffTime -- in seconds
  -> UTCTime
  -> m (Event t TickInfo)
```

`tickLossy` will create an `Event` every `n` seconds. Though it is not guaranteed to always fire an `Event` after the elapsed time, especially if the value `n` is very small.

There are many more APIs in this module to generate repeating events based on more complex algorithms.

From DOM widgets

When doing DOM based programming using `reflex-dom-core`, a number of widgets provide `Event` in response to the external events.

- Input fields like button, text-box, drop down, etc.

  See *DOM Input elements*

- User interaction events like mouse click, mouse over, etc.

  See *DOM Events*

- Response from XHR / AJAX / websocket requests

  See *XHR/ websocket*

- Arbitrary `on` events from the browser

  See *Web APIs and FFI*

Manipulation

Using these primary `Events` you can create secondary / derived events by

1. Manipulating the value using `Functor` / `fmap`:

```
-- inputValueEv :: Event t Int

doubledInputValueEv = ffor inputValue (* 2)
```

2. Filtering the value:

```
-- inputValueEv :: Event t Int

-- This Event will fire only if input value is even
evenOnlyEv = ffilter even inputValueEv
```

   Use `fmapMaybe fforMaybe` for similar filtering

3. Multiple events can be combined using

   Merges the value *a*

```
<>          :: Semigroup a => Event a -> Event a -> Event a
```

   This fires the *a* event only when *b* is not firing at the same time:

```
difference :: Event a -> Event b -> Event a
```

   Combine two separate events:

```
align      ::                       Event a -> Event b -> Event (These a b)
alignWith  :: (These a b -> c) -> Event a -> Event b -> Event c
```

   Combine a list of events:

```
mergeWith  :: (a -> a -> a) -> [Event a] -> Event a
mergeList  :: [Event a] -> Event (NonEmpty a)
```

   Drop all except the *leftmost* event:

```
leftmost   :: [Event a] -> Event a
```

   Other APIs:

```
mergeMap   :: Ord k => Map k (Event a) -> Event (Map k a)
merge      :: GCompare k => DMap (WrapArg Event k) -> Event (DMap k)
```

4. Tagging value of `Dynamic` or `Behavior`.

   Using these APIs, see Quick Ref

```
gate                          ::                        Behavior Bool -> Event a ->␣
↪Event a
tag                           ::                           Behavior a -> Event b ->␣
↪Event a
tagPromptlyDyn                ::                            Dynamic a -> Event b ->␣
↪Event a
attach                        ::                           Behavior a -> Event b ->␣
↪Event (a, b)
attachPromptlyDyn             ::                            Dynamic a -> Event b ->␣
↪Event (a, b)
attachWith                    :: (a -> b ->       c) -> Behavior a -> Event b ->␣
↪Event c
attachPromptlyDynWith         :: (a -> b ->       c) ->  Dynamic a -> Event b ->␣
↪Event c
attachWithMaybe               :: (a -> b -> Maybe c) -> Behavior a -> Event b ->␣
↪Event c
attachPromptlyDynWithMaybe    :: (a -> b -> Maybe c) ->  Dynamic a -> Event b ->␣
↪Event c
<@>                           ::                  Behavior (a -> b) -> Event a ->␣
↪Event b
<@                            ::                           Behavior a -> Event b ->␣
↪Event a
```

The below will create an event which will fire whenever the Dynamic changes and give the *old* value of the Dynamic.

```
tag (current dyn) $ updated dyn
```

3.1.2 Behavior

`Behavior` value can be tagged with an `Event` using `tag` or `attach`, or it can be sampled in a widget, when it is first created using `sample`.

3.1.3 Dynamic

Creation

Create a `Dynamic` which changes value when `Event` occurs:

```
holdDyn :: (MonadHold t m) => a -> Event t a -> m (Dynamic t a)
```

There are also a number of input APIs in `reflex-dom-core` which provide `Dynamic` values in the context of DOM. See *DOM Input elements*

Manipulation

Using some primary `Dynamic` values you can create secondary / derived values by

- `fmap` - Simply use `Functor` instance when only one `Dynamic` value is being manipulated.
- Combine multiple `Dynamic` values using:

```
zipDyn :: Reflex t => Dynamic t a -> Dynamic t b -> Dynamic t (a, b)

zipDynWith :: Reflex t => (a -> b -> c) -> Dynamic t a -> Dynamic t b ->␣
↪Dynamic t c
```

Zipping is useful when multiple `Dynamic` values have a common point of influence in the application.

For example if you have two variable parameters like color and font of text. Then you can construct the dynamic attributes from these parameters by simply zipping them together.:

```
-- textFont :: Dynamic t Text
-- textColor :: Dynamic t Text

getAttr (f,c) = ("style" =: ("font-family: " <> f "; color: " <> c))

elDynAttr "div" (getAttr <$> (zipDyn textFont textColor)) $ text "Text"
```

- Using `Applicative`:

```
-- dInt1, dInt2, dInt3 :: Dynamic t Int
let
  eInt :: Dynamic t (Int, Int, Int)
  eInt = (,,) <$> dInt1 <*> dInt2 <*> dInt3
```

Much more complicated things can be done using `traverse`/ `sequenceA`:

```
-- mDyn :: Map k (Dynamic t Int)
let
  dMap :: Dynamic t (Map k Int)
  dMap = sequenceA mDyn
```

Note: `zipDynWith` is more efficient than `f <$> d1 <*> d2`

3.1.4 Reflex

The `Reflex` class provides the basic functionality for FRP. It provides the basic functions to efficiently handle the `Event`, `Behavior` and `Dynamic` values. All the *pure* APIs like `tagDyn`, `zipDyn`, etc are created using the functionality provided through `Reflex` class.

The other two most important features required for FRP are maintaining some state, and doing modifications based on events. This is provided from the two classes `MonadHold` and `Adjustable`.

Also see QuickRef

3.1.5 MonadHold

This is required to create any stateful computations with Reflex. It designates monads that can create new `Behavior` s based on `Event` s.:

```
hold :: a -> Event t a -> m (Behavior t a)
```

3.1.6 Adjustable

A Monad that supports adjustment over time. After an action has been run, if the given events fire, it will adjust itself so that its net effect is as though it had originally been run with the new value.:

```
runWithReplace :: m a -> Event t (m b) -> m (a, Event t b)
```

3.2 Event Propagation Graph

3.2.1 Simple Tree

Simply pass the `Event`/`Dynamic` values to input of functions. This will create kind of an event propagation flow from top to bottom. But no feedback-loops can be created, for that use `RecursiveDo`.

3.2.2 RecursiveDo

Is used to create a cyclic event propagation graph. Because the underlying mechanism of graph creation is monadic (using `MonadHold`, etc). To create feedback-loops we need to use `MonadFix`.

The actual usage is quite simple:

```
-- Required extension for rec style blocks
-- {-# LANGUAGE RecursiveDo #-}

rec
  let
    ev1 = f2 <$> ev2
  d1 <- widgetHold (w1Init) (w1 <$> ev1)
  ev2 <- viewD1Widget d1
```

in this example the `ev1` is used to create a `Dynamic` value `d1`, which is then shown to the user using `viewD1Widget`. This widget can in turn modify the value using the `Event` `ev2`.

But there are some pitfalls too, especially if you use 'Promptly' APIs like `tagPromptlyDyn`, `switchPromptlyDyn`, `attachPromptlyDyn`, etc. All these APIs take a `Dynamic` value as input, and if used incorrectly they can cause problems like hang, stack overflow, etc.

In most cases you would want to use their corresponding APIs like `tag`, `switch`, `attach`, etc (which all work on the `Behavior` values), along with `current :: Dynamic t a -> Behavior t a`.

see debugging *Hang / Stack Overflow*

For more details checkout the articles on *MonadFix / RecursiveDo*

3.2.3 Maintaining State via fold

In order to store a state/data for your app (ie create a state machine) simply use `foldDyn`

```
-- State can be any arbitrary haskell data
stateDynVal :: Dynamic t MyState

-- ev can a collection of all events on which the state depends
-- For example all input events
```

(continues on next page)

(continued from previous page)

```
ev :: Event t Inputs

-- This is a pure API which can process the input events and current state
-- to generate a new state.
eventHandler :: (Inputs -> MyState -> MyState)

-- foldDyn :: (a -> b -> b) -> b -> Event t a -> Dynamic t b
stateDynVal <- foldDyn eventHandler initState ev
```

Even nested state machines can be designed if your have a state with nested `Dynamic` value by using `foldDynM`

Use `foldDynMaybe`, `foldDynMaybeM` in cases where you want to filter input events, such that they don't modify the state of application.

For example in a shopping cart if the user has not selected any items, the "add to cart" button should do nothing. This kind of behavior can be implemented by returning `Nothing` from the eventHandler.

3.2.4 `getPostBuild`

```
getPostBuild :: PostBuild t m => m (Event t ())
```

This `Event` will fire once at the start of an action / DOM widget is created. Also each time that part of the DOM gets re-created (like if it is created from scratch via `widgetHold`). This can be used to do communication with server or do some FFI.

Note that the `Event` fires when the build action completes, but the fragment may not yet be in the browser DOM. So you might have to add some delay to this before accessing the DOM via some FFI.

3.2.5 Doing IO via `performEvent`

Example:

```
doneEv <- performEvent (ffor triggerEv $ \val -> liftIO $ do
  putStrLn "Doing some action"
  someIOAction val)

widgetHold (text "Waiting for action to complete")
  (showResultOfAction <$> doneEv)
```

Todo: Does the doneEv always occur in the frame after triggerEv?

3.2.6 Debounce, Delay, BatchOccurence

`Reflex.Time` provides a set of useful APIs which come handy when you need to do real life event handling.:

```
debounce :: (_) => NominalDiffTime -> Event t a -> m (Event t a)

-- Wait for user to stop typing for 0.5 sec, and then send a search request to server

searchTextEv <- debounce 0.5 (_textInput_input someTextInput)
```

When doing FFI calls `delay` may be required:

```
delay :: (_) => NominalDiffTime -> Event t a -> m (Event t a)


performEvent (abort <$ stopAndRestartEv)
delayedEv <- delay 0.2 stopAndRestartEv
performEvent (start <$ delayedEv)
```

When handling a set of events from external sources many times the sequence of events is not deterministic, or perhaps we want a `debounce` kind of functionality but dont want to miss any `Event`. In such cases we need to use `batchOccurrences` to properly model the logic.

```
batchOccurrences :: (_) => NominalDiffTime -> Event t a -> m (Event t (Seq a))
```

3.3 Higher order FRP

3.3.1 Nested Values and flattening

When you model real world `Dynamic` values many times you end up with nested structures.

For example, if the value of items in a shopping cart depends on the shipping method chosen, then you can end up with a value `total' :: Dynamic t [Dynamic t Int]`:

```
selectedItems :: Dynamic t [Item]
isExpeditedShipping :: Dynamic t Bool

total' = Dynamic t [Dynamic t Int]
total' = ffor selectedItems
          (map getItemPrice)

getItemPrice :: Item -> Dynamic t Int
getItemPrice itm = ffor isExpeditedShipping
                    (\case
                      True -> (itemPrice itm) + (shippingCharges itm)
                      False -> itemPrice itm)
```

In such cases in order to get a total value `Dynamic t Int`, you need to use flattening APIs. In case of `Dynamic` it is simply `join` from `Control.Monad` (since `Dynamic` has an instance of `Monad`):

```
total'' :: Dynamic t (Dynamic t Int)
total'' = foldr1 (\a b -> (+) <$> a <*> b) <$> total'

total :: Dynamic t Int
total = join total''
```

See QuickRef for details on other flattening APIs.

3.3.2 Dynamic widgets on Dynamic Collections

In order to model complex flows of events or dynamically changing data collection, we need to use higher order containers like lists (`[]`) or Maps (`Data.Map`).

To effectively work with such `Dynamic` collections, `Reflex.Collection` provides a bunch of APIs.

See Quickref for a summary of these APIs https://github.com/reflex-frp/reflex/blob/develop/Quickref.md#collection-management-functions

3.3.3 Reflex.Network

Provides these APIs. If you look closely they are the equivalent of `dyn` and `widgetHold`, but work in non-DOM applications.:

```
networkView :: (Reflex t, NotReady t m, Adjustable t m, PostBuild t m)
  => Dynamic t (m a) -> m (Event t a)

networkHold :: (Reflex t, Adjustable t m, MonadHold t m)
  => m a -> Event t (m a) -> m (Dynamic t a)
```

3.3.4 EventWriter and DynamicWriter

`EventWriter` allows you to send events "upwards" in your widget hierarchy, much like Elm's update propagation.:

```
-- Main APIs
runEventWriterT :: (Reflex t, Monad m, Semigroup w) => EventWriterT t w m a -> m (a,␣
↪Event t w)
tellEvent :: EventWriter t w m => Event t w -> m ()

-- Example usage
body :: MonadWidget t m => m ()
body = do
  rec
    (_, ev) <- runEventWriterT ewbs
    dy <- foldDyn (:) ["bar"] ev
    simpleList dy dynText
  return ()

ewbs :: MonadWidget t m => EventWriterT t Text m ()
ewbs = do
  evClick <- button "Click Me"
  tellEvent ("foo" <$ evClick)
  return ()
```

3.3.5 Requester

`Requester` lets you make requests and receive responses anywhere within your widgets, and automatically collect/distribute them as necessary.

The primary API which will be used to initiate a request and get a response is:

```
requesting :: Event t (Request m a) -> m (Event t (Response m a))
```

This requires defining two type constructors `Request m` and `Response m`.

The API to actually collect all the requests and provide response to each request is:

```
runRequesterT :: (Reflex t, Monad m)
  => RequesterT t request response m a
```

(continues on next page)

(continued from previous page)

```
  -> Event t (RequesterData response)
  -> m (a, Event t (RequesterData request))
```

As you can see all the requests are bundled up in the `RequesterData request`, and the responses are also provided in a similar event of type `RequesterData response`.

The `RequesterData` is like a `Map` structure where the keys are some arbitrary values corresponding to the origin of request, and the values are the actual request data.

to provide a response one can use these APIs:

```
traverseRequesterData :: forall m request response. Applicative m
  => (forall a. request a -> m (response a))
  -> RequesterData request
  -> m (RequesterData response)
```

can be used to provide response to all the request by specifying a *request handler*.

But if you want access to each request separately and provide the responses in independent manner (in case you are doing XHR/ websocket requests for each request separately).

Then you can convert this into a list of key value pairs (`DSum`), provide the response to each request by using the same key with `singletonRequesterData` to recreate the `RequesterData`:

```
requesterDataToList :: RequesterData f -> [DSum RequesterDataKey f]

singletonRequesterData :: RequesterDataKey a -> f a -> RequesterData f
```

3.3.6 Workflow

`Reflex.Workflow` provides a specialised API:

```
newtype Workflow t m a = Workflow { unWorkflow :: m (a, Event t (Workflow t m a))}

workflow :: forall t m a. (Reflex t, Adjustable t m, MonadFix m, MonadHold t m)
  => Workflow t m a -> m (Dynamic t a)
```

The working of this API can be easily explained using a DOM based widget example:

```
-- A DOM based example of Workflow
page1, page2, page3 :: (MonadWidget t m) => Workflow t m Text
page1 = Workflow . el "div" $ do
  el "div" $ text "This is page 1"
  pg2 <- button "Switch to page 2"
  return ("Page 1", page2 <$ pg2)

page2 = Workflow . el "div" $ do
  el "div" $ text "This is page 2"
  pg3 <- button "Switch to page 3"
  pg1 <- button "No wait, I want to go back to page 1"
  return ("Page 2", leftmost [page3 <$ pg3, page1 <$ pg1])

page3 = Workflow . el "div" $ do
  el "div" $ text "You have arrived on page 3"
  pg1 <- button "Start over"
  return ("Page 3", page1 <$ pg1)
```

(continues on next page)

(continued from previous page)

```
main = mainWidget $ do
  r <- workflow page1
  el "div" $ do
    text "Current page is: "
    dynText r
```

3.4 Performance

3.4.1 UniqDynamic

`UniqDynamic` is useful to eliminate redundant update events from a Dynamic.:

```
uniqDynamic :: Reflex t => Dynamic t a -> UniqDynamic t a

fromUniqDynamic :: (Reflex t, Eq a) => UniqDynamic t a -> Dynamic t a
```

Internally, `UniqDynamic` uses pointer equality as a heuristic to avoid unnecessary update propagation; this is much more efficient than performing full comparisons. However, when the UniqDynamic is converted back into a regular Dynamic, a full comparison is performed.

In order to maintain this constraint, the value inside a UniqDynamic is always evaluated to weak head normal form.

Also see the documentation of `Reflex.Dynamic.Uniq`

3.4.2 Patch and Incremental

An `Incremental` is a more general form of a `Dynamic`. Instead of always fully replacing the value, only parts of it can be patched. This is only needed for performance critical code via `mergeIncremental` to make small changes to large values.

`Reflex.Patch.*` provides a number of data structures which have the ability to do incremental updates.

3.4.3 Cheap / Fast variants of APIs

3.5 Internals

3.5.1 Frames

A frame is the atomic time unit

- Frame begins with, say, a mouse click
- Mouse click event fires
- Events fmapped from that event fire
- All other events depending on those events fire
- Repeat until there are no more event firings
- Frame ends

3.5.2 Spider Timeline

CHAPTER 4

Reflex Dom

See Quick Ref

4.1 Basic Widgets

4.1.1 Static DOM

Here is a simple example of using some of the static-dom widgets:

```
-- simple_dom.hs
{-# LANGUAGE OverloadedStrings #-}

import Reflex.Dom

-- Code to showcase Reflex.Dom's APIs to create simple static DOM
main = mainWidget $ do
  simple

simple :: (DomBuilder t m) => m ()
simple = do
  el "div" $
    -- Specify attributes in a (Map Text Text)
    elAttr "span" ("style" =: "color:blue") $
      text "Text inside span"

  -- Use CSS style center-align and red-text
  -- using these specialised APIs
  divClass "center-align" $
    elClass "span" "red-text" $
      text "Div with class center-align and red text"

  el "dl" $ do
    dtdd "dt dd tags" $
```

(continues on next page)

(continued from previous page)

```
      text "Here goes the description"

    dtdd "Reflex" $ do
      text "Haskell + awesome FRP!"
      el "br" $ blank -- Add line break, blank == return ()
      -- A simple URL link
      elAttr "a" ("href" =: "http://reflexfrp.org") (text "Reflex-FRP")
```

4.1.2 Dynamic DOM

To create interactive widgets you need to do changes in DOM in response to `Events` or `Dynamic` values.

The simplest way to create a dynamic DOM is to use library APIs which take Dynamic values as input. The following section covers these APIs. Using these APIs you can create bigger widgets which can have multiple Dynamic values as input.:

```
-- Show simple text
dynText $ someDynTextValue -- :: Dynamic t Text

display $ someDynValueWithShowInstance

-- The value of input element can be modified from an external Event t text
txtInpEl <- inputElement $ def
  & inputElementConfig_setValue .~ changeValueEv
```

Also you can create dynamic widgets by using static widgets, ie the widget which don't take dynamic values as inputs (eg. `button :: Text -> m (Event t a)`). This can be done simply by mapping the Dynamic values over these widgets and using `dyn`.:

```
-- Use the library API button which accepts static Text
-- and modify its value by using a (Dynamic t Text)
dyn (button <$> (value txtInpEl))
```

The library provides a number of standard widgets which accept `Dynamic` values as input

`elDynAttr elDynClass`

> Change the attributes of a DOM element via Dynamic values.

`tableDynAttr`

> A widget to display a table with static columns and dynamic rows.:

`tabDisplay`

> A widget to construct a tabbed view that shows only one of its child widgets at a time. Creates a header bar containing a `<ul>` with one `<li>` per child; clicking a `<li>` displays the corresponding child and hides all others.

4.1.3 DOM Input elements

To create input form elements and use them to create `Event` and `Dynamic` values use the widgets provided by `Reflex.Dom.Widget.Input`

The various input elements usually contain these two values:

```
*_input :: Event t a
*_value :: Dynamic t a
```

The `_input` event will only fires when *user* modifies contents of the input field. But if you are modifying the value of the input field using reflex `Event` and you want to capture even these changes, then use `updated value`.

Tip: When using the `*_input` Events you might have to use `debounce`. See *Debounce, Delay, BatchOccurence*

4.1.4 DOM Events

`domEvent` API can be used to create `Event` on DOM elements:

```
(e,_) <- el' "span" $ text "Click Here"

clickEv :: Event t ()
clickEv <- domEvent Click e
```

For a complete list of events accepted by `domEvent` see `EventName` in Reflex.Dom.Builder.Class.Events

4.1.5 Dynamic widgets based on Events

Create a widget which updates whenever `Event` occurs.

If you have a widget which depends on some event (like server response), but you need to display something else instead of a blank.

```
-- responseEv :: Event t SomeData
-- displaySomeData :: SomeData -> m ()

-- widgetHold :: m a -> Event t (m a) -> m (Dynamic t a)
widgetHold (text "Loading...") (displaySomeData <$> responseEv)
```

Every time the `widgetHold` event fires, it removes the old DOM fragment and builds a new one in-place

4.2 Miscellaneous

4.2.1 Resize Detector

```
-- Reflex.Dom.Widget.Resize
resizeDetector :: (...) => m a -> m (Event t (), a)
```

This is useful to respond to changes in size of a widget.

4.2.2 Host / URL / Location

`Reflex.Dom.Location` contains utility functions for obtaining the host, URL, protocol, etc.

4.2.3 Client side routes

obelisk-route

Obelisk is packaged with a set of routing libraries `obelisk-route`, `obelisk-route-frontend` and `obelisk-route-backend`. These libraries provide the following features

- Type safety in routes design.
- Derive encoding/decoding of routes from a single definition.
- Share the routes between frontend and backend.
- Compile time checking of routes to static files.

For example usage of `obelisk-route` please see source code of reflex-frp.org or reflex-examples.

Apart from this the Reflex.Dom.Contrib.Router provides APIs to manipulate and track the URL.

Also checkout https://github.com/3noch/reflex-dom-nested-routing

4.2.4 SVG

To embed an SVG element use `elDynAttrNS'` along with SVG namespace:

```
elSvgns = elDynAttrNS' (Just "http://www.w3.org/2000/svg")
```

Using *canvas* element with reflex is generally not a good idea, as it is based on an imperative style of coding (vs the declarative style of svg).

Also checkout https://github.com/qfpl/reflex-dom-svg

4.3 XHR/ websocket

For usage on XHR / AJAX requests please see the haddock documentation of module `Reflex.Dom.Xhr`, it contains example usage of the APIs.

4.3.1 Websocket

Use `webSocket` API from the `Reflex.Dom.WebSocket` module.:

```
webSocket
  :: Text -- url, like "ws://localhost:3000/myWebSocketHandler"
          -- use wss for SSL connections
  -> WebSocketConfig t a -> m (WebSocket t)

data WebSocketConfig t a
  = WebSocketConfig {_webSocketConfig_send :: Event t [a],
                     _webSocketConfig_close :: Event t (Word, Text),
                     _webSocketConfig_reconnect :: Bool}

 type WebSocket t =
   RawWebSocket t ByteString

 data RawWebSocket t a
```

(continues on next page)

(continued from previous page)

```
  = RawWebSocket {_webSocket_recv :: Event t a,
                  _webSocket_open :: Event t (),
                  _webSocket_error :: Event t (),
                  _webSocket_close :: Event t (Bool, Text)}
```

To send data over WebSocket pass an event to `_webSocketConfig_send` of type `Event t [a]` where `a` is either `Text` or `ByteString`.

The return value from WebSocket is available from `_webSocket_recv :: Event t ByteString`

Here `_webSocketConfig_close` is an `Event` which can close the WebSocket connection from client side. And `_webSocket_close` is the response from server when the connection closes.

Manually closing a websocket that is configured to reconnect will cause it to reconnect. If you want to be able to close it permanently you need to set `_webSocketConfig_reconnect = False`.

See reflex-examples for an echo example.

4.3.2 Integration with Backend

One of the big strength of `reflex-dom` is that a common code can be shared between backend and frontend.

Quoting mightybyte again. See hsnippet.com source code here

> I used a very similar architecture with Reflex with HSnippet, and it's delightful to work with. Server communication was done over websockets with the wire format being a serialized version of these data types. Adding a new client/server or server/client message couldn't be more simple.

The simplest form of integration with backend is to define the message data in the `common` package, along with its serialisation functions (eg `deriving instance` of `ToJSON` and `FromJSON`).

servent-reflex

https://github.com/imalsogreg/servant-reflex

> *servant-reflex* lets you share your *servant* APIs with the frontend. See the readme for more details.

reflex-websocket-interface

Going a few steps further in this integration is the library reflex-websocket-interface

- It provides a reflex side API like this:

```
getResponse :: (_) => Event t request -> m (Event t response)
```

  This takes care of encoding and decoding of the messages (using `aeson`), do all the routing of Event behind the scenes, and provide the response at the point where request was initiated.

  This architecture of handling the request and its response at the same place in widget code is essential for self-contained widgets. It also helps greatly simplify the coding, especially when there are more than one instance of a widget, and they all use single websocket to communicate.

  Internally this uses *Requester*.

- It ensures the server has code to handle all the request types.
- It further ensures that the type of response for a request is consistent between frontend and backend.

4.4 Performance

4.4.1 Prerendering / Server side rendering

The `renderStatic` API can be used to render the DOM parts of the application to plain HTML. This way the server can serve the generated HTML, so that the page *opens* instantly for the user.:

```
renderStatic :: StaticWidget x a -> IO (a, ByteString)
```

To create widget which support static rendering, the `prerender` API will be required internally to separate the static code from the Immediate DomBuilder one.

```
prerender :: forall js m a. Prerender js m =>
  m a -> (PrerenderClientConstraint js m => m a) -> m a
```

Here the first widget supports Static rendering, and the second one has the actual JSM functionality.

See reflex-examples for example usage.

4.4.2 lazy

`Reflex.Dom.Widget.Lazy` contains widgets for creating long lists. These are scrollable element and only renders child row elements near the current scroll position.

CHAPTER 5

Application Development with Reflex-DOM

5.1 Debugging

5.1.1 Functionality

In addition to the normal `Debug.Trace` APIs, the following can be used for debugging.

The output of these APIs will be in the browser console when compiled with `ghcjs`. For `jsaddle-warp` and `webkit` based apps the output will be on the terminal.:

```
traceEvent :: (Reflex t, Show a) => String -> Event t a -> Event t a
traceEventWith :: Reflex t => (a -> String) -> Event t a -> Event t a
```

Moreover the reflex-dom-contrib package contains a bunch of utility functions. One can just copy-paste these functions, ie use them without dependency on the package.:

```
-- Reflex.Dom.Contrib.Utils
-- pops up a javascript alert dialog box
alertEvent :: (_) => (a -> String) -> Event t a -> m ()

-- pops up a javascript confirmation dialog box
confirmEvent :: (_) => (a -> String) -> Event t a -> m (Event t a)

-- | Prints a string when an event fires.  This differs slightly from
-- traceEvent because it will print even if the event is otherwise unused.
putDebugLnE :: MonadWidget t m => Event t a -> (a -> String) -> m ()
```

5.1.2 Hang / Stack Overflow

In general its possible to create a loop by mistake with this kind of code in a "pure" haskell.:

```
let
  f v = ... (f v)
```

But thanks to `MonadFix` (`RecursiveDo`) this is a very common problem, even in a "monadic" code.

Basically for doing anything useful one has to introduce a feedback in the event propagation graph. And often this can lead to either a loop or a deadlock.

To fix this

- Breaking down a big `rec` block into nested `rec` blocks or a series of `rec` blocks. Moving the code in a separate functions can also help simplify the `rec` block.

  Also see: using *newTriggerEvent* to break down a big `rec` block.

- Avoid using `switchPromptlyDyn` / `tagPromptlyDyn`, instead use `switch . current` / `tag . current`

  Many times what one really need is the previous value of a `Dynamic` to create a cyclic event propagation.

- Use `widgetHold` against `dyn`

  Separating an initial value from an update event means that the function using them doesn't have to call `sample` on a `Dynamic`, which can be unsafe when you don't know whether the MonadFix knot has been tied.

  Using `widgetHold` ensures that the user doesn't accidentally give an untied Dynamic.

For more details checkout the articles on *MonadFix / RecursiveDo*

5.1.3 Compilation Errors

These are a few common compile time errors which can occur while using the widgets

- If you define a widget but don't use it any where

```
-- 't' is not used anywhere
let t = textInput $ def

Compile error

• Couldn't match type ‘DomBuilderSpace m0’ with ‘GhcjsDomSpace’
    arising from a use of ‘textInput’
  The type variable ‘m0’ is ambiguous
• In the expression: textInput $ def
  In an equation for ‘t’: t = textInput $ def
```

  Solution: Simply comment this code or use it.

- In a `rec` block if use a "pure" API in a "monadic" context, then you can get weird type errors:

```
-- This will lead to type-checker assume the monad to be Dynamic
ev <- switchPromptlyDyn dynEv
```

  The biggest problem with such errors is that the line numbers are not correct, so it can take a while to figure out the source of error

  One possible solution is to explicitly specify the type of functions and expression in the `let` and `do` block inside of `rec`:

```
--  This is required to specify the types
--  {-# LANGUAGE ScopedTypeVariables #-}

-- This can be useful to specify types partially, just to help figure out source␣
↪of error
```

(continues on next page)

(continued from previous page)

```
--  {-# LANGUAGE PartialTypeSignatures #-}

-- Specify an explicit forall
myWidget :: forall t m k . (MonadWidget t m, Ord k)
  => Map k Text -> m ()
myWidget mapInput = do
  ..

  rec
    let
      eTabClicks :: Event t k = leftmost tabClicksList

    d :: Dynamic t k <- do
      someCodeThatIsSupposedToReturnDynamicK
```

5.2 Web APIs and FFI

- For working with DOM and using Web APIs the `ghcjs-dom` package should suffice.

  It provides APIs like `getElementById`, `getBoundingRect` to work with DOM, and many other Web APIs related to geolocation, media management, web audio, etc.

  To use the DOM related APIs for `reflex-dom` created elements, extract the *raw* element from the *reflex element*

```
import qualified GHCJS.DOM.Types as DOM
import qualified GHCJS.DOM.DOMRectReadOnly as DOM
import qualified GHCJS.DOM.Element as DOM

  (e,_) <- el' "div" $ text "Hello"

  let getCoords e = DOM.liftJSM $ do
        rect <- DOM.getBoundingClientRect (_element_raw e)
        y <- DOM.getY rect
        h <- DOM.getHeight rect
        return (y,h)

  performEvent (getCoords e <$ ev)
```

- But when using external .js files, one has to do arbitrary JS code execution.

  For doing this `jsaddle` package is preferred as it provides a type-safe way to execute the JS code.

  See documentation of `Language.Javascript.JSaddle.Object` for examples

  See *DOM-UI Libraries* for example usage.

- It is also possible to do arbitrary JS code block execution using `eval` API from `Language.Javascript.JSaddle.Evaluate`.

```
eval :: (ToJSString script) => script -> JSM JSVal

liftJSM $ eval "console.log('Hello World')"
```

- JSFFI functions

  This will only work with `ghcjs`:

```
import GHCJS.Types (JSVal)

foreign import javascript unsafe
  "try { $r = $1 / $2; } catch (e) { $r = "error"; }"
  divide :: Double -> Double -> JSVal
```

See https://github.com/ghcjs/ghcjs/blob/master/doc/foreign-function-interface.md

5.2.1 Capturing DOM events with FFI

Many of the Web APIs work on a *callback* mechanism, where a user supplied function will be called. Many of these APIs in JS code start with *on* prefix.

Example JS code for creating an AudioNode to handle audio data, Source

```
// Give the node a function to process audio events
scriptNode.onaudioprocess = function(audioProcessingEvent) {
  // The input buffer is the song we loaded earlier
  var inputBuffer = audioProcessingEvent.inputBuffer;
  ..
  }
```

Similar callback can be created by using the `on` API from `GHCJS.DOM.EventM`

```
-- here audioProcess is the equivalent "tag" for JS onaudioprocess

  myNode :: ScriptProcessorNode

  liftJSM $ on myNode audioProcess myAudioProcessHandler

myAudioProcessHandler :: EventM ScriptProcessorNode AudioProcessingEvent ()
myAudioProcessHandler = do
  -- aEv :: AudioProcessingEvent
  aEv <- ask
  buf <- getInputBuffer aEv
  ..
```

5.2.2 Exception Handling

5.3 Integrating CSS and embed in HTML

`reflex-dom` has the following entry points for embedding CSS and a head widget:

```
mainWidget :: (forall x. Widget x ()) -> IO ()

mainWidgetWithHead :: (forall x. Widget x ()) -> (forall x. Widget x ()) -> IO ()

-- Share data between head and body widgets
mainWidgetWithHead' :: (a -> Widget () b, b -> Widget () a) -> IO ()

-- import Data.FileEmbed -- from file-embed package
-- This requires TemplateHaskell
-- customCss :: ByteString
```

(continues on next page)

(continued from previous page)

```
-- customCss = $(embedFile "src/custom.css")
mainWidgetWithCss :: ByteString -> (forall x. Widget x ()) -> IO ()

mainWidgetInElementById :: Text -> (forall x. Widget x ()) -> IO ()
```

`reflex-dom-core` provides equivalent functions in `Reflex.Dom.Main` for use with `jsaddle-warp`

5.4 Deploying

5.4.1 Nix based server

If your server has `nix` installed then the steps to deploy are quite simple.

If you are using *reflex-project-skeleton* or following project-development.md follow the instructions and create the `nix-build` outputs of your backend and frontend projects.

- Frontend

  For `ghcjs` based projects the `frontend-result` will contain the *.js files which you can simply copy to the desired location on server.

  For information on the use of closure compiler to reduce the size of `all.js` see https://github.com/ghcjs/ghcjs/wiki/Deployment

- Backend

  For `backend-result` once you have the build products ready, copy them to server using:

  ```
  # or nix copy, if using nix 2.0
  $ nix-copy-closure --to someuser@server.org backend-result
  ```

  You will have to configure the server's nix configuration and add *someuser* to trusted users:

  For NixOS add this to `/etc/nixos/configuration.nix`:

  ```
  nix.trustedUsers = [ "someuser" ];
  ```

  For non NixOS, add this to `/etc/nix/nix.conf`:

  ```
  trusted-users = someuser
  ```

  On the server then use the same nix-path

5.5 Miscellaneous

5.5.1 Rendering image from `ByteString`

If you have the encoded image data as `ByteString` then you can render the image in browser using the *img* tag in combination with *createObjectURL*.

This API will create a URL which can be specified in the `img` tag's `src` attribute:

```
foreign import javascript unsafe "window['URL']['createObjectURL']($1)"␣
↪createObjectURL_ :: Blob.Blob -> IO JS.JSVal

createObjectURL :: ByteString -> IO Text
createObjectURL bs = do
  let opt :: Maybe JS.BlobPropertyBag
      opt = Nothing
  -- bsToArrayBuffer :: MonadJSM m => ByteString -> m ArrayBuffer
  ba <- bsToArrayBuffer bs
  b <- Blob.newBlob [ba] opt
  url <- createObjectURL_ b
  return $ T.pack $ JS.fromJSString $ JS.pFromJSVal url
```

5.6 Android / iOS Apps

On a mobile device the speed of a `ghcjs` based browser app can be extremely bad. But the good news is that with little effort the `reflex-dom` apps can be compiled to run as a native mobile app. The performance of these apps can be considerably faster (of the order of 10x) as the haskell runtime runs on the actual processor.

See the README of *reflex-project-skeleton* or project-development.md for instructions of creating an android or iOS app from your frontend project.

Also see: https://github.com/gonimo/gonimo

Note: Cross-compiling currently doesn't support Template Haskell, so replace all the `makeLenses`, etc code with generated splices

---

Todo: Expand this section

CHAPTER 6

Non-DOM related usage of `reflex`

The `reflex` FRP architecture (and package) can be used to create non-DOM based UI application and even some non-UI stuff like server.

6.1 `reflex-host`

Source : https://github.com/bennofs/reflex-host

This provides a set of higher-level abstractions on top of the `reflex` FRP primitives.

Using this library, you don't need to build your own event loop. You can just start registering external events and performing actions in response to FRP events.

- https://github.com/dalaing/reflex-host-examples

  This has a set of examples using this package

- https://github.com/dalaing/reflex-basic-host

  Contains an even simplified API interface

6.2 UI

- https://github.com/reflex-frp/reflex-sdl2

  Experimental SDL 2 based reflex app using sdl2 haskell bindings.

- https://github.com/deech/fltkhs-reflex-host

  An experimental code for *FLTK GUI toolkit* based applications using reflex.

- https://github.com/lspitzner/bricki-reflex

  Experimental `brick` based terminal UI.

  http://hexagoxel.de/postsforpublish/posts/2017-10-30-brick-plus-reflex.html

6.3 Other

- https://github.com/dalaing/reflex-server-websocket

CHAPTER 7

Resources

7.1 Tutorials

- Queensland FP Lab: Functional Reactive Programming with *reflex*

  https://blog.qfpl.io/projects/reflex/

  These are very well written tutorials for beginners. It also has a number of exercises.

- https://github.com/hansroland/reflex-dom-inbits/blob/master/tutorial.md

  This is a single page *long-form* introduction, which covers a lot of material for `reflex-dom` applications.

7.2 Examples

- https://github.com/gspia/reflex-examples

  A fork of the https://github.com/reflex-frp/reflex-examples, updated to use a recent reflex-platform together with an example on the new project setup (as of early 2018).

  Examples include Basic ToDo, Drag-and-Drop, file input and many more.

- https://github.com/reflex-frp/reflex-dom-contrib

  A collection is useful APIs and DOM widgets.

- https://github.com/gspia/7guis-reflex-nix

  Example of 7 types of GUI tasks from basic counter to a spreadsheet.

7.3 Applications

7.3.1 Full-Stack Haskell Apps

- http://hsnippet.com/

  A web application to try out reflex in browser.

  The code is somewhat out of date, so latest features in reflex may not be available.

  Code: https://github.com/mightybyte/hsnippet

- http://hexplore.mightybyte.net/

  An experimental interface to browse haskell packages (registered on hackage)

  Code: https://gitlab.com/mightybyte/hexplore/

- https://tenjinreader.com

  An application to read Japanese books. Uses *reflex-project-skeleton*.

  It has a web + android version of the reflex app

  Code: https://github.com/blueimpact/tenjinreader

- https://app.gonimo.com/

  The free baby monitor for smartphone, tablet or PC.

  It has a web + android version of the reflex app

  Code: https://github.com/gonimo/gonimo

7.3.2 Games

- https://mightybyte.github.io/reflex-2048/

  Code: https://github.com/mightybyte/reflex-2048

- https://rvl.github.io/flatris/

  Code: https://github.com/rvl/flatris

  A simple FE only game. This also contains an example of auto-reloading development environment

7.3.3 Other

- https://github.com/CBMM/cochleagram

  Tools for psychoacoustics.

  This captures WebAudio, and does the processing to create an audio spectogram.

7.4 Reflex Libraries

7.4.1 DOM-UI Libraries

- Semantic UI components

https://github.com/reflex-frp/reflex-dom-semui

- Bootstrap Material Design

  https://github.com/hexresearch/reflex-material-bootstrap

  See README for instructions on integrating external js and also for using closure-compiler.

- Material Components

  https://github.com/alasconnect/reflex-material

- https://github.com/TaktInc/reflex-dhtmlx

  A wrapper around *date-picker* widget from DHTMLX

- https://github.com/gspia/reflex-dom-htmlea

  This library provides short-hand names for the most common HTML elements and attributes.

  A longer term aim is to provide self contained customisable components providing reasonable default settings with examples, allowing to build demos quickly. For example, a table component gives a functionality in which it is possible to select columns, cells, rows and have other ready made functionality.

  Also see https://github.com/gspia/reflex-dom-themes and https://github.com/gspia/reflex-dom-htmlea-vs

7.4.2 Other Libraries

- https://github.com/diagrams/diagrams-reflex

  Port of the `diagrams` library with svg output. See the README for supported constructs.

  Examples http://bergey.github.io/gooey/

  https://github.com/bergey/gooey

- https://github.com/qfpl/reflex-dom-svg

  This is a work-in-progress helper library for creating svg

- https://github.com/qfpl/reflex-dom-canvas

  An experimental support for canvas element

- https://github.com/reflex-frp/reflex-dom-ace

  This package provides a Reflex wrapper around the ACE editor.

  This is also intended to serve as an example of how to structure FFI packages that rely on external JS packages.

- https://github.com/dfordivam/audiocapture

  Demo for capturing audio via WebAudio APIs

7.5 Posts / Blogs

- https://github.com/mightybyte/real-world-reflex/blob/master/index.md
- https://emmanueltouzery.github.io/reflex-presentation

7.5.1 MonadFix / RecursiveDo

- 24 Days of GHC Extensions: Recursive Do

  https://ocharles.org.uk/blog/posts/2014-12-09-recursive-do.html
- Grokking Fix

  http://www.parsonsmatt.org/2016/10/26/grokking_fix.html
- MonadFix is Time Travel

  https://elvishjerricco.github.io/2017/08/22/monadfix-is-time-travel.html
- Haskell Wiki

  https://wiki.haskell.org/MonadFix
- Typeclassopedia on MonadFix

  https://wiki.haskell.org/Typeclassopedia#MonadFix

Todo: Add Haddock documentation
