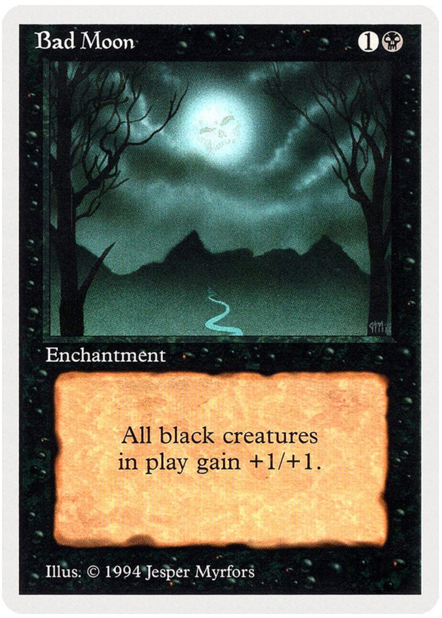
Reflex Outside the Browser

Jack Kelly
http://jackkelly.name/talks

Queensland Functional Programming Lab
CSIRO's Data61

September 2, 2019

Thought Experiment: Implement a Card Game

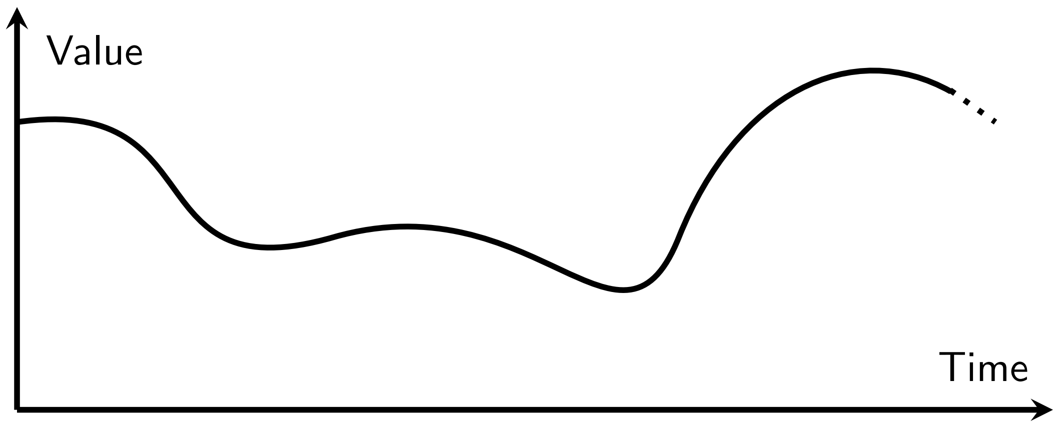

Bad Moon
Enchantment
All black creatures in play gain +1/+1.
Illus. © 1994 Jesper Myrfors

- How to implement?
- Imperative/OO:
  - Enters play: twiddle all the black creatures
  - Leaves play: untwiddle all the black creatures
  - Black creature enters play: twiddle it
  - ...
- What about creatures that change colour?
- What if the text box changes?
- Want a better way to handle time-varying state

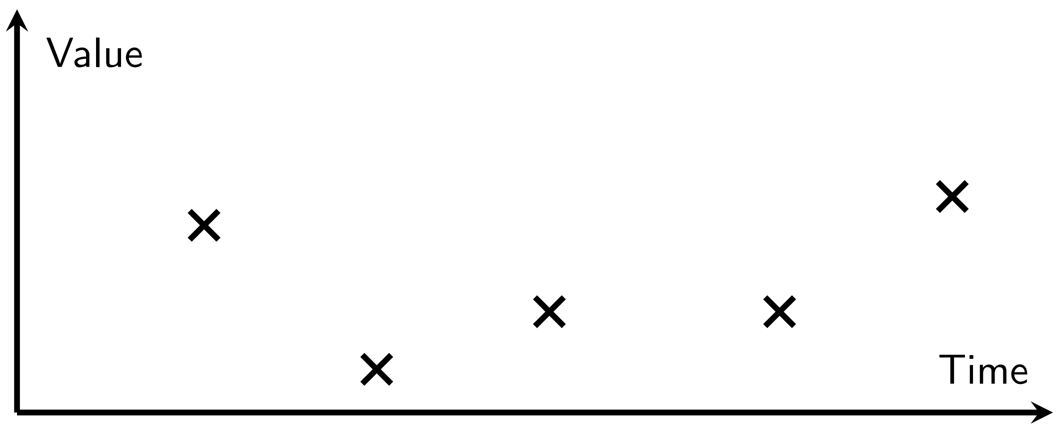
What is Reflex?

- Functional Reactive Programming (FRP) is a solid theory for talking about time-varying values and instantaneous phenomena
- Reflex is an implementation of this theory*
    - (* FRP specifies continuous time, seldom implemented)
- Primitives:
    - `Behavior a`: a time-varying a
    - `Event a`: instantaneous occurrences of a
    - `Dynamic a`: like `Behavior a`, but also signals its updates

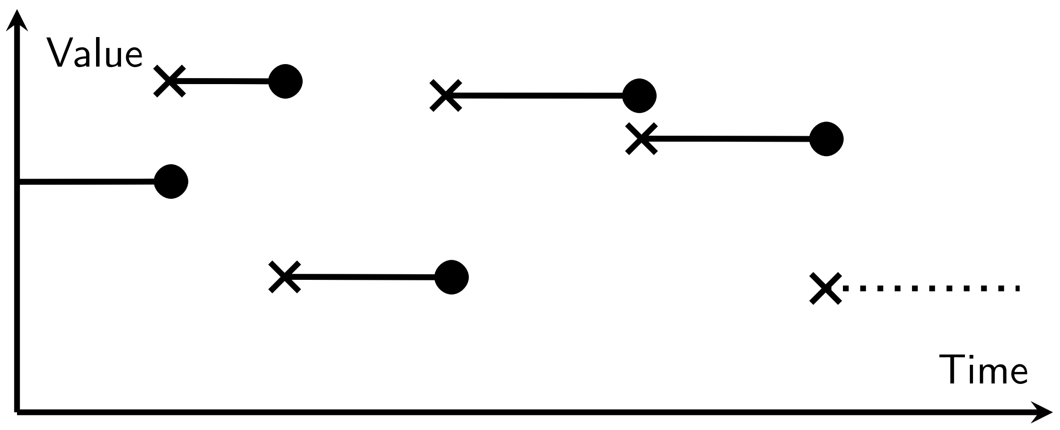
`Behavior a`: a time-varying a



`Event a`: instantaneous occurrences of `a`



`Dynamic a`: like `Behavior a`, but also signals its updates



Typeclasses

- What does a typeclass define?
- What does a typeclass *mean*?
- `Behavior` has `Functor`, `Applicative`, and `Monad` instances
- `Dynamic` has `Functor`, `Applicative`, and `Monad` instances
- `Event` has a `Functor` instance but isn't even `Applicative`!
  - but it is `Filterable` (from `witherable`)
  - and `Semialign` (from `these`/`semialign`)

`Filterable` and `Semialign`

- `Filterable` (from `witherable`)

```
class Functor f => Filterable f where
  mapMaybe :: (a -> Maybe b) -> f a -> f b
  catMaybes :: f (Maybe a) -> f a
  filter :: (a -> Bool) -> f a -> f a
```

- `Semialign` (from `these`/`semialign`)

```
data These a b = This a | That b | These a b

class Functor f => Semialign f where
  align :: f a -> f b -> f (These a b)
```

Laws! (not discussed)

- For `Filterable`:

```
mapMaybe (Just . f) = fmap f
mapMaybe f . mapMaybe g = mapMaybe (f <=< g)
```

- For `Semialign`:

```
-- (N.B.: join f = f x x):
join align = fmap (join These)
align (f <$> x) (g <$> y)
  = bimap f g <$> align x y
alignWith f a b = f <$> align a b
align x (align y z)
  = fmap assoc (align (align x y) z)
```

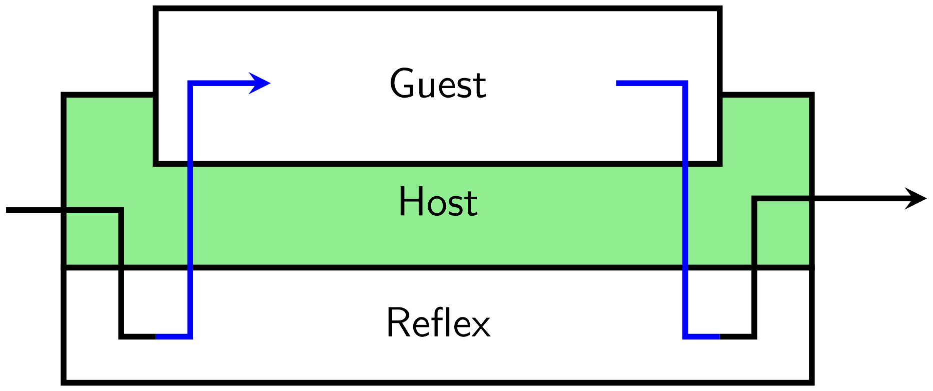
- For `Foldable Semialign`s:

```
toList x
  = toListOf (folded . here) (align x y)
  = mapMaybe justHere (toList (align x y))
```

Challenges of Reflex

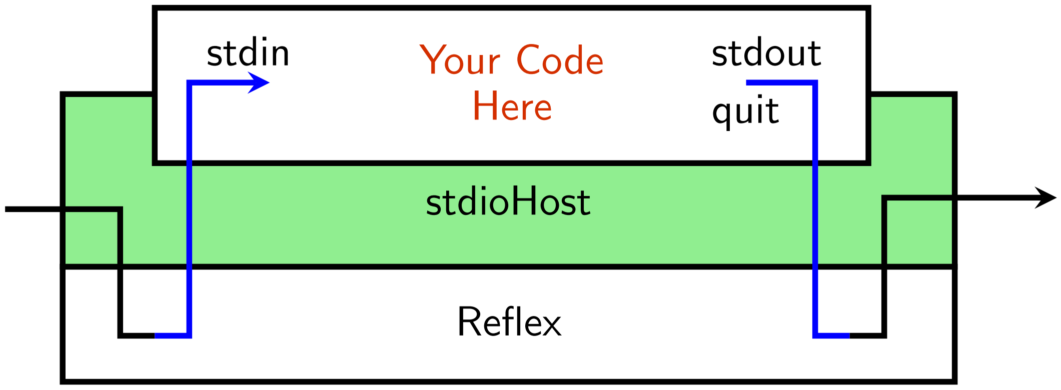
- Feels like a big jump:
  - Spectacular type signatures
  - Pigeonholed as frontend tech (GHCjs)
  - Reflex-platform (nix)
- For today:
  - Simplified type signatures:
    - Reflex: `Event t a`
    - These slides: `Event a`
  - Native binaries
  - Recent versions of Reflex are on Hackage

Hosts and Guests



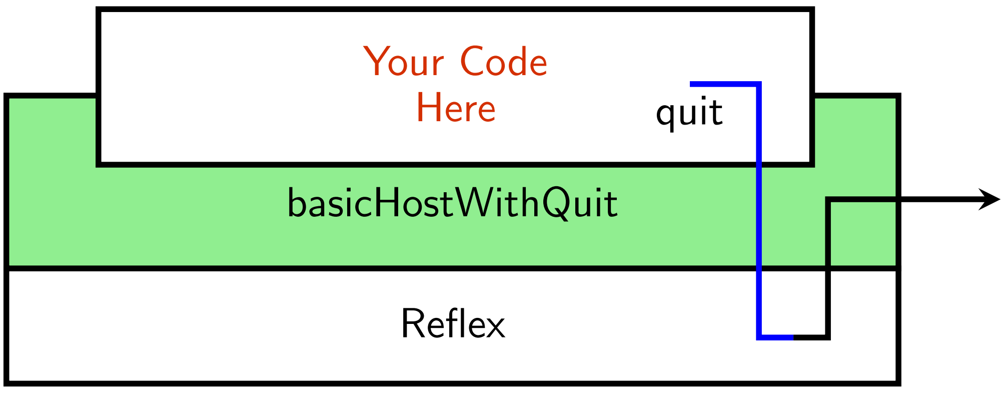
- Guests ask for features, classy MTL-style:
  - `(PostBuild m, TriggerEvent m) => ... -> m ()`
- This lets us switch out the FRP runtime
- Extend the runtime with `PostBuildT`, `TriggerEventT`, `PerformEventT`, ...

Example Host: String I/O



```
stdioHost
  :: (Event String -> m (Event String, Event ()))
  --   ~~~stdin~~~~         ~~~stdout~~~   ~~quit~~
  -> IO ()
```

Basic Host



```
basicHostWithQuit :: m (Event ()) -> IO ()
                  --     ~~~~~~~~~ quit
```

- Provided by `reflex-basic-host`
- Run until the returned event fires
- You connect your guest to the outside world

`class PostBuild (Reflex.PostBuild.Class)`

```
class PostBuild m where
  getPostBuild :: m (Event ())
```

- Morally: “Here’s an event that fires when the network is built”

class TriggerEvent (Reflex.TriggerEvent.Class)

```
class TriggerEvent m where
  -- And a couple of others
  newTriggerEvent :: m (Event a, a -> IO ())
                  --    ~~~~~~~   ~~~~~~~~~~
                  --     |          '- Its trigger
                  --     '- The event
```

- Morally: "`m` can create new events"
- Usually pass the trigger to another thread

class PerformEvent (Reflex.PerformEvent.Class)

```
class PerformEvent m where
  type Performable m :: Type -> Type
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- '- Associated type (from -XTypeFamilies)

  -- And a couple of others
  performEvent
    :: Event (Performable m a)
    --        ~~~~~~~~~~~~~~~ Actions to perform
    -> m (Event a)
    --    ~~~~~~~ Results of actions
```

- Morally: "Perform each action as it happens, and fire off the results"
- `Performable m` is often `MonadIO`
- Not always (e.g., frontend/backend in webapps)

Recreating stdio: Standard Output

```
performEvent_
  :: PerformEvent m
  => Event (Performable m ())
  -> m ()

stdout :: PerformEvent m => Event String -> m ()
stdout eStrings = performEvent_
  (liftIO . putStrLn <$> eStrings)
-- ~~~~~~~~~~~~~~~~~     ~~~~~~~~ Event String
--  '- MonadIO io => String -> io ()
--
-- ~~~~~~~Event-of-actions~~~~~~~
```

Recreating stdio: Standard Input

- After the network is built, create an event, and...
- ...kick off a thread, which...
- ...loops forever, feeding lines into the trigger

```
performEventAsync
  :: (TriggerEvent m, PerformEvent m)
  => Event ((a -> IO ()) -> Performable m ())
--           ~Trigger~~     ~~~~~Action~~~~~
  -> m (Event a)

stdin :: (...) => m (Event String)
stdin = do
  ePostBuild <- getPostBuild
  let loop fire = void $ liftIO $ forkIO
        (forever $ getLine >>= fire)
     -- ~~~~~~~~~~~~~~~~~~~~~~~~~~ Loop forever
  performEventAsync (loop <$ ePostBuild)
--~~~~Perform the loop on PostBuild~~~~~
```

Recompiling OpenGL Shaders: fsnotify

- Callback-oriented libraries work well with `TriggerEvent`
- `fsnotify` watches a directory for file changes and calls your callback when that happens
- We want an `Event (FSNotify.Event)`

```
watchDir
  :: WatchManager
  -> FilePath
  -> (FSNotify.Event -> Bool) -- ActionPredicate
  -> (FSNotify.Event -> IO ()) -- Action
  -> IO (IO ()) -- IO StopListening
```

Recompiling OpenGL Shaders: fsnotify

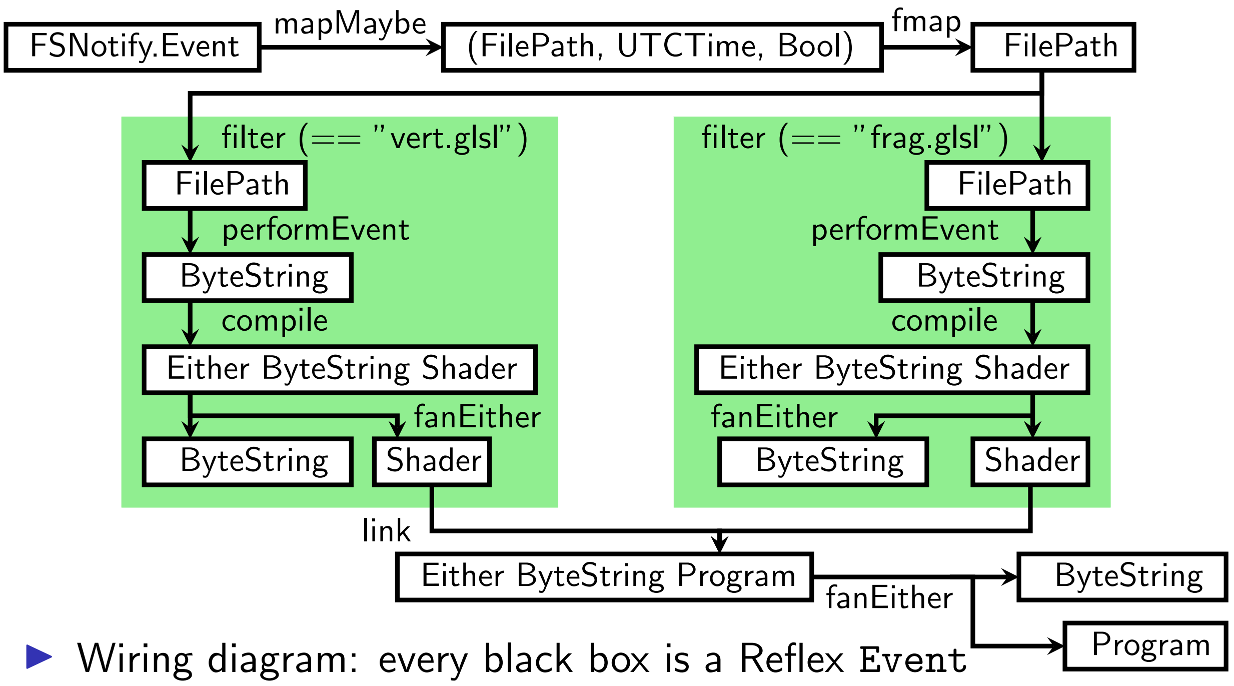
```
watchDir
  :: WatchManager
  -> FilePath
  -> (FSNotify.Event -> Bool) -- ActionPredicate
  -> (FSNotify.Event -> IO ()) -- Action
  -> IO (IO ()) -- IO StopListening
```

```
newEventWithLazyTriggerWithOnComplete
  :: TriggerEvent m
  => ((a -> IO () -> IO ()) -> IO (IO ()))
  --        ~~~~~ On-complete      ~~~~~
  --   ~~~~~~Trigger~~~~~~          '- Teardown
  -> m (Event a)
```

Recompiling OpenGL Shaders: `fsnotify`

```
watchDir
  :: TriggerEvent m
  => WatchManager
  -> FilePath
  -> m (Event FSNotify.Event)
watchDir manager dir
  = newEventWithLazyTriggerWithOnComplete $
      \fire -> FSNotify.watchDir
        manager         -- Passed through
        dir             -- Passed through
        (\_ -> True)    -- Collect all events
        (\fsEvent -> fire fsEvent (pure ()))
    --                  ~~~~           ~~~~~~~~
    -- Reflex trigger -'     Do nothing -'
    --                        (on-complete
    --                         callback)
```

Recompiling OpenGL Shaders: Shader Wiring Diagram



- Wiring diagram: every black box is a Reflex `Event`
- Begin with the `Event FSNotify.Event`...
- ...and transform it into `Event Program` of OpenGL shader programs, with a new program on each change
- See `watchShaderProgram` in `Shader.hs`

Takeaways

- Learn by doing
- FRP first, web stuff later
- Start with `reflex-basic-host`
- Wiring diagrams!

Links

- Demo code: `https://github.com/qfpl/reflex-gl-demo`
- `reflex`: `https://hackage.haskell.org/package/reflex`
- `reflex-basic-host`: `https://github.com/qfpl/reflex-basic-host`
- `glow`: `https://github.com/ekmett/codex/tree/master/glow`