The essence of live coding: Change the program, keep the state!

Abstract

One rarely encounters programming languages and frameworks that provide general-purpose and type-safe hot code swap. It is demonstrated here that this is entirely possible in Haskell, by faithfully following the motto of live coding: "Change the program, keep the state."

With generic programming, one easily arrives at an automatic state migration function. The approach can be generalised to an arrowized Functional Reactive Programming framework that is parametrized by its side effects. It allows for building up complete live programs from reusable, modular components, and to separate data flow cleanly from control flow. Useful utilities for debugging and quickchecking are presented.

1 Introduction

Live coding has come to denote various concepts, from hot code swap on a production server to artistic performances with code that is written, executed and updated live. From the implementor's perspective, the common denominator is a framework to update the definition of a program while not interrupting its execution.

In a dynamically typed, process-oriented language, such as Erlang [1], hot code swap is conceptually simple: A new server process is started, the state of the old server is transferred to the new one, and the old server is shut down. The challenge lies in ensuring that the state is correctly migrated, as corrupt state might crash the server. This is a nontrivial task if no type checker is available to verify if the migrated state conforms to the schema required by the new server.

Domain specific live coding frameworks usually have a central server that generates the desired effect – be it audio, video or device communication – and offers an API through which users can create "cells", such as oscillators or signal processors in the case of audio (such as [21]), and connect, reorder, update or destroy them during execution. The central server guarantees safe updates, and strongly typed user-side libraries (such as Tidal [14] for audio applications) exist, but the user is restricted to a domain specific language.

In this article, we implement a lightweight general purpose live coding framework in Haskell from scratch. It is not only type-safe, but also type-driven, in that boilerplate code for state migrations (which is hard to get right without a static type checker) is automatically derived from the type. It is not restricted to a particular domain, by virtue of being parametrised over an arbitrary monad: Any domain- or library-specific effect can be incorporated effortlessly, and handled with standard Haskell functions. The framework follows the tradition of monadic arrowized Functional Reactive Programming (FRP) as developed in [18] and [16]. To run live programs created in it, a runtime environment in the `IO` monad is supplied, but since the framework does not hide `IO` in its abstraction (such as many other FRP frameworks do), it is an easy exercise to execute the live programs in e.g. the `STM` monad [8] or any other concurrency context such as an external main loop. The state of the live program can be inspected and debugged safely during the execution.

There is no subtly clever trick. We simply follow, dutifully and without compromise, the mantra of live coding:

Change the program. Keep the state.

This is not a new idea in itself. Hot code swap in Erlang realises this motto, and similar views are expressed about live coding in Elm[1] (a domain specific web frontend language inspired by Haskell). What is new about this approach is the consequential application of this motto to create a general purpose, type-safe FRP framework[2] with *automatic state migration*.

Arriving at a simple *state migration function* by faithfully following the live coding mantra is a manageable task, carried out in Section 2. Upon the migration function, a live coding framework is constructed in Section 3. It is much more rewarding to recast this framework in the form of functional reactive programming, which allows us to reuse modular, functional components and separate data flow from control flow. Crucially, the state of our live programs is built up automatically by using FRP idioms. The result is presented in Section 4, which heavily draws inspiration from Dunai, a monadic arrowized FRP framework. After having implemented the data flow aspects of our framework, we turn to control flow in Section 5. A monadic interface to our live programs is presented. In Section 6, several useful tools such as debuggers and quickchecking utilities are shown.

This article is written in literate Haskell and supplies the library presented here. The source code will be made openly available upon publication.

2 Change the program. Keep the state (as far as possible).

Our model of a live program will consist of a state and an effectful state transition function. A preliminary version is

[1] Compare https://elm-lang.org/blog/interactive-programming.

[2] It shall be remarked that FRP is long past niche applications in the video and audio domains. It is possible to write web servers and frontends, simulations and games in it.

```haskell
data LiveProgram m s = LiveProgram
  { liveState :: s
  , liveStep  :: s -> m s
  }
stepProgram
  :: Monad m
  => LiveProgram m s -> m (LiveProgram m s)
stepProgram liveProgram@LiveProgram { .. } = do
  liveState' <- liveStep liveState
  return liveProgram { liveState = liveState' }
stepProgramMVar
  :: MVar (LiveProgram IO s)
  -> IO ()
stepProgramMVar var = do
  currentProgram <- takeMVar var
  nextProgram <- stepProgram currentProgram
  putMVar var nextProgram
launch
  ::           LiveProgram IO s
  -> IO (MVar (LiveProgram IO s))
launch liveProgram = do
  var <- newMVar liveProgram
  forkIO $ forever $ stepProgramMVar var
  return var
```

Figure 1. `LiveProgramPreliminary.lhs`

shown in Figure 1. The program is initialised at a certain state, and from there its behaviour is defined by repeatedly applying the function `liveStep` to advance the state and produce effects. This is implemented in `stepProgram`. Since we want to run the program in a separate thread while compiling a new version of the program in the foreground, we have to store the program in a a concurrent variable, here an `MVar`. Given this variable, stepping the program it contains is a simple `IO` action, implemented in `stepProgramMVar`. To run the program, we fork a background thread and repeatedly call `stepProgramMVar` there.

In a dynamically typed language, such a setup is in principle enough to implement hot code swap. At some point, the execution will be paused, and the function `liveStep` is simply exchanged for a new one. Then the execution is resumed with the new transition function, which operates on the old state. Of course, the new step function has to take care of migrating the state to a new format, should this be necessary. The difficulties arise (apart from the practicalities of the implementation), from the inherent unsafety of this operation: Even if the old transition function behaved correctly, and the old state is in a valid format, the new transition function may crash or otherwise misbehave on the old state. It is very hard to reduce the probability of such a failure with tests since the current state constantly changes

```haskell
hotCodeSwap
  :: (s -> s')
  -> LiveProgram m s'
  -> LiveProgram m s
  -> LiveProgram m s'
hotCodeSwap migrate newProgram oldProgram
  = LiveProgram
    { liveState = migrate $ liveState oldProgram
    , liveStep  = liveStep newProgram
    }
```

Figure 2. `Preliminary/HotCodeSwap.lhs`

(by design). A static typechecker is missing, to guarantee the safety of this operation.

But let us return to Haskell, where we have such a typechecker. It immediately points out the unsafety of the migration: There is no guarantee that the new transition function will typecheck with the old state! In fact, in many situations, the state type needs to be extended or modified.

This kind of problem is not unknown. In the world of databases, it is commonplace that a table lives much longer than its initial schema. The services accessing the data can change, and thus also the requirements to the data format. The solution to this problem is a *schema migration*, an update to the database schema that alters the table in such a way that as little data as possible is lost, and that the table adheres to the new schema afterwards. Data loss is not entirely preventable, though. If a column has to be deleted, its data will not be recoverable. In turn, if a column is created, one has to supply a sensible default value (often `NULL` will suffice).

2.1 Migrating the state

We can straightforwardly adopt this solution by thinking of the program state as a small database table with a single row. Its schema is the type `s`. Given a *type migration* function, we can perform hot code swap, as shown in Figure 2.

This may be an acceptable solution to perform a planned, well-prepared intervention, but it does spoil the fun in a musical live coding performance if the programmer has to write a migration function after every single edit. What a live performer actually needs, is a function with this mysterious type signature:

```haskell
hotCodeSwap
  :: LiveProgram m s'
  -> LiveProgram m s
  -> LiveProgram m s'
```

It is the same type signature as in Figure 2, but with the first argument, the manual migration function, removed. The new program, including its initial state, has just been compiled, and the old program is still stored in a concurrent variable. Can we possibly derive the new state by simply looking at the initial state of the new program and the old state? Is

there a magical, universal migration function? If there were, it would have this type:

```
migrate :: s' -> s -> s'
```

A theoretician will probably invoke a free theorem [20] here, and infer that there is in fact a unique such function: `const`! But it is not what we were hoping for. `const s' s` will throw away the old state and run the program with the new initial state – effectively restarting our program from a blank slate.

In this generality, we cannot hope for any other solution. But in the following, we are going to see how to tweak the live program definition by only twenty characters, and arrive at an effective migration function.

2.2 Type-driven migrations

In many cases, knowing the old state and the new initial state is sufficient to derive the new, migrated state safely. As an example, imagine the internal state of a simple webserver that counts the number of visitors to a page.

```
data State = State { nVisitors :: Int }
```

The server is initialised at 0, and increments the number of visitors every step. (For a full-fledged webserver, the reader is asked to patiently wait until the next section.)

```
server = LiveProgram (State 0) $ \State { .. }
  -> State $ return $ nVisitors + 1
```

We extend the state by the name of the last user agent to access the server (initially not present):

```
data State = State Int (Maybe ByteString)
initState = State 0 Nothing
```

From just comparing the two datatype definitions, it is apparent that we would want to keep the number of visitors, of type `Int`, when migrating. For the new argument of type `Maybe ByteString`, we cannot infer any sensible value from the old state, but we can take the value `Nothing` from the new initial state, and interpret it as a default value. A general state migration function should specialise to:

```
migrate (Server1.State nVisitors)
        (Server2.State _         mUserAgent)
     =  Server2.State nVisitors mUserAgent
```

Our task was less obvious if we would have extended the state by the last access time, encoded as a UNIX timestamp:

```
data State = State Int Int
```

Here it is unclear to which of the `Int`s the old value should be migrated. It is obvious again if the datatype was defined as a record as well:

```
data State = State
  { nVisitors      :: Int
  , lastAccessUNIX :: Int
  }
```

We need to copy the `nVisitors` field from the old state, and initialise the `lastAccessUNIX` field from the new state. (Conversely, if we were to migrate back to the original definition, there is no way but to lose the data stored in `lastAccessUNIX`.) Clearly, the record labels enabled us to identify the correct target field. The solution lies in the datatype definition.

We can meta-program a migration function by reasoning about the structure of the type definition. This is possible with the techniques presented in the seminal, now classic article "Scrap Your Boilerplate" [13]. It supplies a typeclass `Typeable` which enables us to compare types and safely type-cast at runtime, and a typeclass `Data` which allows us to inspect constructor names and record field labels. Using the package `syb`, which supplies common utilities when working with `Data`, our migration function is implemented in under 50 lines of code, with the following signature:

```
migrate :: (Data a, Data b) => a -> b -> a
```

It handles the two previously mentioned cases: Constructors with the same names, but some mismatching arguments, and records with some coinciding field labels, but possibly a different order. In nested datatype definitions, the function recurses into all children of the data tree. Needless to say, if the types do match, then the old state is identically copied.

Sometimes it is necessary to manually migrate some part of the state. Assume, for the sake of the example, that our webserver has become wildly popular, and `nVisitors` is close to `maxInt`. We need to migrate this value to an arbitrary precision `Integer`. It is easy to extend `migrate` by a special case provided by the user:

```
userMigrate
  :: (Data a, Data b, Typeable c, Typeable d)
  => (c -> d)
  -> a -> b -> a

intToInteger :: Int -> Integer
intToInteger = toInteger
```

In our example, we would use `userMigrate intToInteger` to migrate the state.

To use the automatic migration function, we only need to update the live program definition to include the `Data` constraint, as shown in Figure 3. This is a small restriction. The `Data` typeclass can be automatically derived for every algebraic data type, except those that incorporate *functions*. We have to refactor our live program such that all functions are contained in `liveStep` (and can consequently not be migrated), and all data is contained in `liveState`.

Now that we have a universal migration function, it is not necessary to carry the type of the state around in the type signature. In fact it would be cumbersome in combination with `MVar`s (which can't change their type), and a real burden when later modularising the state. Consequently, the type is made existential. The only necessary information is that it is an instance of `Data`.

```haskell
data LiveProgram m = forall s . Data s
  => LiveProgram
  { liveState :: s
  , liveStep  :: s -> m s
  }
hotCodeSwap
  :: LiveProgram m
  -> LiveProgram m
  -> LiveProgram m
hotCodeSwap
  (LiveProgram newState newStep)
  (LiveProgram oldState _)
  = LiveProgram
  { liveState = migrate newState oldState
  , liveStep  = newStep
  }
```

Figure 3. `LiveProgram.lhs`

3 The runtime

3.1 Hands on interaction

Enough declaration. Let us get semantic and run some live programs! In the preliminary version, a function `stepProgram` implemented a single execution step, and it can be reused here, up to removing the explicit state type. The runtime behaviour of a live program is defined by calling this function repeatedly. We could of course run the program in the foreground thread:

```haskell
foreground :: Monad m => LiveProgram m -> m ()
foreground liveProgram
  =   stepProgram liveProgram
  >>= foreground
```

But this would leave no possibility to exchange the program with a new one. Instead, we can store the program in an `MVar` and call `stepProgramMVar` on it. Now that we can migrate any `Data`, we can follow the original plan of exchanging the live program in mid-execution:

```haskell
update
  :: MVar (LiveProgram IO)
  ->       LiveProgram IO
  -> IO ()
update var newProg = do
  oldProg <- takeMVar var
  putMVar var $ hotCodeSwap newProg oldProg
```

The old program is retrieved from the concurrent variable, migrated to the new state, and put back for further execution. And so begins our first live coding session in GHCi (line breaks added for readability):

```
 > var <- newMVar $ LiveProgram 0
    $ \s -> print s >> return (s + 1)
 > stepProgramMVar var
0
```

```haskell
load :: IO (MVar (LiveProgram IO))
load = readStore (Store 0)
save :: MVar (LiveProgram IO) -> IO ()
save var = writeStore (Store 0) var
liveinit _ = return $ unlines
  [ "var <- newMVar liveProgram"
  , "save var"
  ]
livereload _ = return $ unlines
  [ ":reload"
  , "var <- load"
  , "update var liveProgram"
  ]
livestep _ = return "stepProgramMVar var"
```

Figure 4. `GHCi.lhs`

```
 > stepProgramMVar var
1
 > update var $ LiveProgram 0
    $ \s -> print s >> return (s - 1)
 > stepProgramMVar var
2
 > stepProgramMVar var
1
 > stepProgramMVar var
0
```

Upon updating, the state was correctly preserved. The programs were specified in the interactive session here, but of course we will want to load the program from a file, and use GHCi's `:reload` functionality when we have edited it. But as soon as we do this, the local binding `var` is lost. The package `foreign-store` [5] offers a remedy: `var` can be stored persistently across reloads. To facilitate its usage, GHCi macros are defined for the initialisation and reload operations (Figure 4). They assume the main live program and the `MVar` to be called `liveProgram` and `var`, respectively, but this can of course be generalised. With the macros loaded, the session simplifies to:

```
 > :liveinit
 > :livestep
0
 > :livestep
1
 > :livereload
[1 of 1] Compiling Main ( ... )
Ok, one module loaded.
 > :livestep
2
 > :livestep
1
 > :livestep
```

```
data Env = Env
  { requestVar  :: MVar Request
  , responseVar :: MVar ByteString
  }
```

Figure 5. DemoWai.lhs

```
0
```

Before entering `:livereload`, the main file was edited in place and reloaded. Of course, it is not intended to enter `:livestep` repeatedly when coding. We want to launch a separate thread which executes the steps in the background. Again, we can reuse the function `launch`. (Only the type signature needs updating.) In the next subsection, a full example is shown.

3.2 Live coding a webserver

To show that live coding can be applied to domains outside audio and video applications, let us realise the example from the previous section and create a tiny webserver using the WAI/Warp framework [22]. It is supposed to count the number of visitors, and keep this state in memory when we change the implementation.

The boiler plate code, which is suppressed here, initialises the Warp server, uses `launch` to start our live program in a separate thread and waits for user input to update it.

To save ourselves an introduction to Warp, we will communicate to it via two **`MVar`**s, which we need to share with the live program. The textbook solution is to supply the variables through a **`Reader`** environment, which needs to supplied to the live program before execution. This can be done by transporting the program along the `runReaderT` monad morphism. A function `hoistLiveProgram` does this (borrowing nomenclature from the `mmorph` [7] package).

The server logic is shown in Figure 6. It is initialised at 0 visitors. The step function receives the number of past visitors and blocks on an **`MVar`** until a request (which is discarded) to the server arrives. The number of visitors is incremented by 1, and baked into a response, which is in another **`MVar`**. Finally, the updated state (the incremented number of visitors) is returned, and passed to the next step.

We then modify[3] the server logic as in Figure 7. Additionally to the number of visitors, we also store the last user agent name in the state, if it was sent. For this, one more record field is added to the state type.

Let us run the old server, and switch to the new one during execution. From a console, we access the running server:

```
$ curl localhost:8080
This is Ye Olde Server.
```

[3]The functions `fromStrict` and `pack` might be unfamiliar. They are from the `bytestring` package and convert between different kinds of strings. `requestHeaders` from the `wai` package extracts the HTTP headers, such as the user agent name, from a request, as a list of tuples.

```
data State = State
  { nVisitors :: Integer
  } deriving Data

oldServer :: LiveProgram (ReaderT Env IO)
oldServer = LiveProgram
  { liveState = State 0
  , liveStep  = \State { .. } -> do
      Env { .. } <- ask
      _ <- lift $ takeMVar requestVar
      let nVisitorsNew = nVisitors + 1
      lift $ putMVar responseVar $ unlines
        [ "This is Ye Olde Server."
        , "You are visitor #"
        <> (pack $ show nVisitorsNew) <> "."
        ]
      return $ State nVisitorsNew
  }
```

Figure 6. DemoWai1.lhs

```
You are visitor #1.
$ curl localhost:8080
This is Fancy Nu $3rv3r!
You are visitor #2.
$ curl localhost:8080
This is Fancy Nu $3rv3r!
You are visitor #3.
Last agent: curl/7.64.0
```

It correctly remembered the number of past visitors upon reload and initialised the last user agent with the value **`Nothing`**. When accessing the new server again, it stored the user agent as expected.

4 Live coding as arrowized FRP

Writing out the complete state of the live program explicitly is tedious. We have to plan the whole program in advance and artificially separate its state from the step function. Such a development approach prevents us from writing programs in a modular fashion. The purpose of this section is to show that we can develop live programs modularly by extending the approach presented so far to an arrowized FRP framework.

In ordinary functional programming, the smallest building blocks are functions. It stands to reason that in live coding, they should also be some flavour of functions, in fact, **`Arrow`**s [10]. We will see that it is possible to define bigger live programs from reusable components. Crucially, the library user is disburdened from separating state and step function. The state type is built up behind the scenes, in a manner compatible with the automatic state migration.

```
data State = State
  { nVisitors :: Integer
  , lastAgent :: Maybe ByteString
  } deriving Data
newServer :: LiveProgram (ReaderT Env IO)
newServer = LiveProgram
  { liveState = State 0 Nothing
  , liveStep  = \State { .. } -> do
      Env { .. } <- ask
      request <- lift $ takeMVar requestVar
      let nVisitorsNew = nVisitors + 1
          lastAgentNew = fmap fromStrict
            $ lookup "User-Agent"
            $ requestHeaders request
      lift $ putMVar responseVar $ unlines $
        [ "This is Fancy Nu $3rv3r!"
        , "You are visitor #"
        <> (pack $ show nVisitorsNew) <> "."
        ] ++ maybeToList
        (("Last agent: " <>) <$> lastAgent)
      return $ State nVisitorsNew lastAgentNew
  }
```

Figure 7. DemoWai2.lhs

4.1 Cells

In our definition of live programs as pairs of state and state steppers, we can generalise the step functions to an additional input and output type. Live programs are thus generalised to effectful *Mealy machines* [15]. Let us call them cells, the building blocks of everything live:

```
data Cell m a b = forall s . Data s => Cell
  { cellState :: s
  , cellStep  :: s -> a -> m (b, s)
  }
```

Such a cell may progress by one step, consuming an `a` as input, and producing, by means of an effect in some monad `m`, not only the updated cell, but also an output datum `b`:

```
step
  :: Monad m
  => Cell m a b
  -> a -> m (b, Cell m a b)
step Cell { .. } a = do
  (b, cellState') <- cellStep cellState a
  return (b, Cell { cellState = cellState', .. })
```

As a simple example, consider the following `Cell` which adds all input and returns the delayed sum each step:

```
sumC :: (Monad m, Num a, Data a) => Cell m a a
sumC = Cell { .. }
  where
    cellState = 0
    cellStep accum a = return (accum, accum + a)
```

We recover live programs as the special case of trivial input and output:

```
liveCell
  :: Functor     m
  => Cell        m () ()
  -> LiveProgram m
liveCell Cell { .. } = LiveProgram
  { liveState = cellState
  , liveStep  = fmap snd . flip cellStep ()
  }
```

4.2 FRP for automata-based programming

Effectful Mealy machines, here cells, offer a wide variety of applications in FRP. The essential parts of the API, which is heavily inspired by the FRP library `dunai` [18], is shown here. We will address the data flow aspects in this section, investigating control flow later in Section 5.

Composition By being an instance of the type class `Category` for any monad `m`, cells implement sequential composition:

```
(>>>)
  :: Monad m
  => Cell  m a b
  -> Cell  m   b c
  -> Cell  m a   c
```

For two cells `cell1` and `cell2` with state types `state1` and `state2`, the composite `cell1 >>> cell2` holds a pair of both states:

```
data Composition state1 state2 = Composition
  { state1 :: state1
  , state2 :: state2
  } deriving Data
```

The step function executes the steps of both cells after each other. They only touch their individual state variable, the state stays encapsulated.

Composing `Cell`s sequentially allows us to form live programs out of *sensors*, pure signal functions and *actuators*:

```
type Sensor   a   = Cell   IO         () a
type SF       a b = forall m . Cell m    a b
type Actuator   b = Cell   IO           b ()

buildLiveProg
  :: Sensor   a
  -> SF       a b
  -> Actuator   b
  -> LiveProgram IO
buildLiveProg sensor sf actuator = liveCell
  $ sensor >>> sf >>> actuator
```

This will conveniently allow us to build a whole live program from smaller components. It is never necessary to specify a big state type manually, it will be composed from basic building blocks like `Composition`.

Arrowized FRP `Cell`s can be made an instance of the `Arrow` type class, which allows us to lift pure functions to `Cell`s:

```
arr
  :: Monad m
  ->         (a -> b)
  -> Cell  m  a    b
```

Together with the `ArrowChoice` and `ArrowLoop` classes (discussed in the appendix), cells can be used in *arrow notation* [17] with `case`-expressions, `if then else` constructs and recursion. The next subsection gives some examples.

An essential aspect of an FRP framework is some notion of *time*. As this approach essentially uses the `dunai` API, a detailed treatment of time domains and clocks as in [2] can be readily applied here. But let us, for simplicity and explicitness, assume that we will execute all `Cell`s at a certain fixed step rate, say a thousand steps per second. Then an Euler integration cell can be defined:

```
stepRate :: Num a => a
stepRate = 25

integrate
  :: (Data a, Fractional a, Monad m)
  => Cell m a a
integrate = arr (/ stepRate) >>> sumC
```

The time since activation of a cell is then famously [16, Section 2.4] defined as:

```
localTime
  :: (Data a, Fractional a, Monad m)
  => Cell m b a
localTime = arr (const 1) >>> integrate
```

Monads and their morphisms Beyond standard arrows, a `Cell` can encode effects in a monad, so it is not surprising that Kleisli arrows can be lifted:

```
arrM
  :: Monad m
  ->         (a -> m b)
  -> Cell  m  a      b
```

In case our `Cell` is in another monad than `IO`, one can define a function that transports a cell along a monad morphism:

```
hoistCell
  :: (forall x . m1 x   ->      m2 x)
  ->        Cell m1 a b -> Cell m2 a b
```

For example, we may eliminate a `ReaderT r` context by supplying the environment through the `runReaderT` monad morphism, or lift into a monad transformer:

```
liftCell
  :: (Monad m, MonadTrans t)
  => Cell         m  a b
  -> Cell      (t m) a b
liftCell = hoistCell lift
```

As described in [18, Section 4], we can successively handle effects (such as global state, read-only variables, logging, exceptions, and others) until we arrive at `IO`. Then we can execute the live program in the same way as before.

4.3 A sine generator

Making use of the `Arrows` syntax extension, we can implement a harmonic oscillator that will produce a sine wave with amplitude 10 and given period length:

```
sine
  :: MonadFix m
  => Double -> Cell m () Double
sine t = proc () -> do
  rec
    let acc = - (2 * pi / t) ^ 2 * (pos - 10)
    vel <- integrate -< acc
    pos <- integrate -< vel
  returnA -< pos
```

By the laws of physics, velocity is the integral of accelleration, and position is the integral of velocity. In a harmonic oscillator, the acceleration is in the negative direction of the position, multiplied by a spring factor depending on the period length, which can be given as an argument. The integration arrow encapsulates the current position and velocity of the oscillator as internal state, and returns the position.

The sine generator could in principle be used in an audio or video application. For simplicity, we choose to visualise the signal on the console instead, with our favourite Haskell operator varying its horizontal position:

```
asciiArt :: Double -> String
asciiArt n = replicate (round n) ' ' ++ ">>="

printEverySecond :: Cell IO String ()
printEverySecond = proc string -> do
  count <- sumC -< 1 :: Integer
  if count `mod` stepRate == 0
    then arrM putStrLn -< string
    else returnA       -< ()
```

Our first live program written in FRP is ready:

```
printSine :: Double -> LiveProgram IO
printSine t = liveCell
  $   sine t
  >>> arr asciiArt
  >>> printEverySecond
```

What if we would run it, and change the period in mid-execution? We execute the program such that after a certain time, the live environment inserts `printSine` with a different period. Let us execute it:[4]

```
>>=
     >>=
               >>=
                     >>=
```

[4] From now on, the GHCi commands will be suppressed.

```
                >>=
    >>=
>>=
 >>=
       >>=
                >>=
                     >>=
                       >>=
                    >>=
               >>=
     >>=
>>=
>>=
```

It is clearly visible how the period of the oscillator changed, while its position (or, in terms of signal processing, its phase) did not jump. If we use the oscillator in an audio application, we can retune it without hearing a glitch; if we use it in a video application, the widget will smoothly change its oscillating velocity without a jolt.

5 Control flow

Although we now have the tools to build big signal pathways from single cells, we have no way yet to let the incoming data decide which of several offered pathways to take for the rest of the execution. While we can (due to `ArrowChoice`) temporarily branch between two cells using `if then else`, the branching is reevaluated (and the previous choice forgotten) every step. We are lacking permanent *control flow*.

The primeval arrowized FRP framework Yampa [16] caters for this requirement by means of switching from a signal function to another if an event occurs. Dunai [18, Section 5.3], taking the monadic aspect seriously, rediscovers switching as effect handling in the `Either` monad.

5.1 Exceptions

In Dunai, we can switch from executing one stream function to another by *throwing an exception*. Whenever we wish to hand over control to another component, we throw an exception as an effect in the `ExceptT` monad (which is simply the `Either` monad beefed up as a monad transformer). This exception has to be handled by choosing a new component based on the exception value. The type checker can verify at the end that all exceptions have been handled.

Dunai offers a `Monad` interface where the values in the context are the *thrown exceptions* (and not the output data). This offers a comfortable and idiomatic way of separating data flow and control flow, resulting in well-structured code. It will turn out that we can implement a `Functor` instance effortlessly, and an `Applicative` instance with a little work, but the `Monad` instance will be quite a high bar to clear.

Throwing exceptions No new concepts beyond the function `throwE :: Monad m => e -> ExceptT e m a` from the package `transformers` [6, 11] are needed:

```
throwC
  :: Monad m
  => Cell (ExceptT e m) e arbitrary
throwC = arrM throwE
```

The above function simply throws the incoming exception. To do this only if a certain condition is satisfied, `if`-constructs can be used. For example, this cell forwards its input for a given number of seconds, and then throws an exception:

```
wait
  :: Monad           m
  => Double
  -> Cell (ExceptT () m) a a
wait tMax = proc a -> do
  t <- localTime -< ()
  if t >= tMax
    then throwC  -< ()
    else returnA -< a
```

Handling exceptions In usual Haskell, the `ExceptT` monad transformer is handled by running it:

```
runExceptT :: ExceptT e m b -> m (Either e b)
```

The caller can now decide how to handle the value `e`, should it occur. This approach can be adapted to cells. A function is supplied that runs the `ExceptT e` layer:

```
runExceptC
  :: (Data e, Monad m)
  => Cell (ExceptT e m) a           b
  -> Cell            m  a (Either e b)
```

To appreciate its inner workings, let us again look at the state it encapsulates:

```
data ExceptState state e
  = NotThrown state
  | Exception e
  deriving Data
```

As long as no exception occurred, `runExceptC cell` simply stores the state of `cell`, wrapped in the constructor `NotThrown`. The output value `b` is passed on. As soon as the exception `e` is thrown, the state switches to `Exception e`, and the exception is output forever.

As soon as the exception is thrown, we can "live bind" it to further cells as an extra input:

```
(>>>=) :: (Data e1, Monad m)
  => Cell (ExceptT e1    m)      a  b
  -> Cell (ExceptT    e2 m) (e1, a) b
  -> Cell (ExceptT    e2 m)      a  b
(>>>=) cell1 cell2 = proc a -> do
  eb <- liftCell $ runExceptC cell1 -< a
  case eb of
    Right b -> returnA -< b
    Left e  -> cell2   -< (e, a)
```

We run the exception effect of the first cell. Before it has thrown an exception, its output is simply forwarded. As

soon as the exception is thrown, the second cell is activated and fed with the input and the thrown exception.

5.2 Control flow context

Inspired by [2, Section 2, "Control Flow through Exceptions"], we introduce a newtype:

```
newtype CellExcept m a b e = CellExcept
  { runCellExcept :: Cell (ExceptT e m) a b }
```

We can enter the `CellExcept` context from an exception-throwing cell, trying to execute it until the exception occurs:

```
try
  :: Cell (ExceptT e m) a b
  -> CellExcept      m  a b e
try = CellExcept
```

And we can leave it safely once we have proven that there are no exceptions left to throw, i.e. the exception type is empty:

```
safely
  :: Monad      m
  => CellExcept m a b Void
  -> Cell       m a b
safely = hoistCell discardVoid . runCellExcept
  where
    discardVoid
      = fmap (either absurd id) . runExceptT
```

One way to prove the absence of further exceptions is, of course, to run an exception-free cell:

```
safe
  :: Monad      m
  => Cell       m a b
  -> CellExcept m a b Void
safe cell = CellExcept $ liftCell cell
```

The return of the monad Our new hope is to give `Functor`, `Applicative` and `Monad` instances to `CellExcept`. We will explore now how this allows for rich control flow.

The `Functor` instance is not too hard. When an exception is raised, we simply apply a given function to it:

```
instance Functor m
  => Functor (CellExcept m a b) where
  fmap f (CellExcept cell) = CellExcept
    $ hoistCell (withExceptT f) cell
```

The `pure` function of the `Applicative` class (or equivalently, `return` of the `Monad`), is simply throwing an exception, wrapped in the newtype:

```
pure
  :: Monad      m
  =>                  e
  -> CellExcept m a b e
pure e = CellExcept $ arr (const e) >>> throwC
```

Like the sequential application operator `<*>` from the `Applicative` class can be defined from the bind operator `>>=`, it can also be defined from the *live bind* operator `>>>=` introduced previously. As a technical tour-de-force, even a `Monad` instance for `CellExcept` can be derived with some modifications. This is shown at length in the appendix.

But how can `Applicative` and `Monad` be put to use? The foreground value of `CellExcept` is the thrown exception. With `pure`, such values are created, and `Functor` allows us to perform computations with them. The classes `Applicative` and `Monad` allow us to *chain* the execution of exception throwing cells:

```
sineWait
  :: Double
  -> CellExcept IO () String Void
sineWait t = do
  try  $   arr (const "Waiting...")
       >>> wait 2
  safe $   sine t
       >>> arr asciiArt
```

This `do`-block can be read intuitively. Initially, the first cell is executed, which returns the message `"Waiting..."` every second. After three seconds, it throws an exception, which is handled by activating the sine generator. Since all exceptions have been handled, we leave the `CellExcept` context and run the resulting program:

```
printSineWait :: LiveProgram IO
printSineWait = liveCell
  $   safely (sineWait 10)
  >>> printEverySecond
```

```
Waiting...
Waiting...
>>=
  >>=
       >>=
             >>=
                  >>=
                    >>=
                  >>=
             >>=
       >>=
 >>=
>>=
 >>=
       >>=
             >>=
                   >>=
```

The crucial advantage of handling control flow this way is that the *control state* – that is, the information which exceptions have been thrown and which cell is currently active – is encoded completely in the overall state of the live program, and can thus be migrated automatically. Let us rerun the above example, but after the first `try` statement has already passed control to the sine generator we shorten the period length of the sine wave and reload:

```
Waiting...
Waiting...
>>=
  >>=
       >>=
            >>=
                 >>=
                   >>=
                 >>=
            >>=
       >>=
 >>=
>>=
      >>=
                  >>=
                   >>=
       >>=
>>=
```

The migrated program did not restart and wait again, but remembered to immediately continue executing the sine generator from the same phase as before. This is in contrast to simplistic approaches to live coding in which the control flow state is forgotten upon reload, and restarted each time.

5.3 Exceptions forever

What if we want to change between the oscillator and a waiting period indefinitely? In other words, how do we repeatedly execute this action:

```
sinesWaitAndTry
  :: MonadFix   m
  => CellExcept m () String ()
sinesWaitAndTry = do
  try $   arr (const "Waiting...")
      >>> wait 1
  try $   sine 5
      >>> arr asciiArt
      >>> wait 5
```

The one temptation we have to resist is to recurse in the `CellExcept` context to prove the absence of exceptions:

```
sinesForever'
  :: MonadFix   m
  => CellExcept m () String Void
sinesForever' = do
  sinesWaitAndTry
  sinesForever'
```

It typechecks, but it does *not* execute correctly. As the initial state is built up, the definition of `sinesForever'` inquires about the initial state of all cells in the `do`-expression, but last one is again `foo`, and thus already initialising such a cell hangs in an infinite loop. Using the standard function `forever :: Applicative f => f a -> f ()` has the same deficiency, as it is defined in essentially the same way.

The resolution is an explicit loop operator, and faith in the library user to remember to employ it.

```
foreverE
  :: (Monad m, Data e)
  =>                e
  -> Cell (ReaderT  e (ExceptT e m)) a b
  -> Cell                       m   a b
```

The loop function receives as arguments an initial exception, and a cell that is to be executed forever[5]. Again, it is instructive to look at the internal state of the looped cell:

```
data ForeverE e s = ForeverE
  { lastException :: e
  , initState     :: s
  , currentState  :: s
  }
  deriving Data
```

`foreverE e cell` starts with the initial state of `cell`, and a given value `e`. Then `cell` is stepped, mutating `currentState`, until it encounters an exception. This new exception is stored, and the cell is restarted with the original initial state. The cell may use the additional input `e` to ask for the last thrown exception (or the initial value, if none was thrown yet). The exception is thus the only method of passing on data to the next loop iteration.[6] In our example, we need not pass on any data, so a simpler version of the loop operator is defined:

```
foreverC
  :: (Data e, Monad m)
  => Cell (ExceptT e m) a b
  -> Cell            m  a b
foreverC = foreverE () . liftCell
  . hoistCell (withExceptT $ const ())
```

Now we can finally implement our cell:

```
sinesForever :: MonadFix m => Cell m () String
sinesForever = foreverC
  $ runCellExcept
  $ sinesWaitAndTry

printSinesForever :: LiveProgram IO
printSinesForever = liveCell
  $   sinesForever
  >>> printEverySecond
```

Let us run it:

```
Waiting...
>>=
       >>=
                 >>=
                 >>=
       >>=
```

[5] Of course, the monad `m` may again contain exceptions that can be used to break from this loop.

[6] It is the user's responsibility to ensure that it does not introduce a space leak, for example through a lazy calculation that builds up bigger and bigger thunks.

```
Waiting...
>>=
       >>=
                    >>=
                    >>=
       >>=
Waiting...
```

6 Tooling

6.1 Debugging the live state

Having the complete state of the program in one place allows us to inspect and debug it in a central place. We might want to interact with the user, display aspects of the state and possibly even change it in place. In short, a debugger is a program that can read and modify, as an additional effect, the state of an arbitrary live program:

```
newtype Debugger m = Debugger
  { getDebugger :: forall s .
      Data s => LiveProgram (StateT s m)
  }
```

A simple debugger prints the unmodified state to the console:

```
gshowDebugger :: Debugger IO
gshowDebugger = Debugger
  $ liveCell $ arrM $ const $ do
    state <- get
    lift $ putStrLn $ gshow state
```

Thanks to the `Data` typeclass, the state does not need to be an instance of `Show` for this to work: `syb` offers a generic `gshow` function. A more sophisticated debugger could connect to a GUI and display the state there, even offering the user to pause the execution and edit the state live. We can bake a debugger into a live program:

```
withDebugger
  :: Monad       m
  => LiveProgram m
  -> Debugger    m
  -> LiveProgram m
```

Again, let us understand the function through its state type:

```
data Debugging dbgState state = Debugging
  { state    :: state
  , dbgState :: dbgState
  } deriving (Data, Eq, Show)
```

On every step, the debugger becomes active after the cell steps, and is fed the current `state` of the main program. Depending on `dbgState`, it may execute some side effects or mutate the `state`, or do nothing at all[7].

Live programs with debuggers are started just as usual. Let us inspect the state of the example `printSineWait` from Section 5.2. It is a simple, albeit lengthy exercise in generic programming to prune all irrelevant parts of the state when printing it, resulting in a tidy output like:

```
Waiting...
NotThrown: (1.0e-3)
 >>> +(0.0) >>> (0.0)+ >>> (1)
NotThrown: (2.0e-3)
 >>> +(0.0) >>> (0.0)+ >>> (2)
[...]
Waiting...
NotThrown: (2.0009999999998906)
 >>> +(0.0) >>> (0.0)+ >>> (2001)
Exception:
 >>> +(3.9478417604357436e-3) >>> (0.0)+
 >>> (2002)
[...]
```

The cell is initialised in a state where the exception hasn't been thrown yet, and the `localTime` is `1.0e-3` seconds. The next line corresponds to the initial state (position and velocity) of the sine generator which will be activated after the exception has been thrown, followed by the internal counter of `printEverySecond`. In the next step, local time and counter have progressed. Two thousand steps later, the exception is finally thrown, and the sine wave starts.

6.2 Testing with `QuickCheck`

Often, some cells in a live program should satisfy certain correctness properties. It is good practice in Haskell to build up a program from functions, and ensure their correctness with property-based testing. `QuickCheck` [3] is the primeval framework for this. It generates arbitrary input for a function, and checks whether given assertions are valid.

Unit tests In our live coding approach, programs are not composed of mere functions, but of cells, and of course we wish to test them in a similar way before reloading. As a simple example, we wish to assure that `sumC` will never output negative numbers if only positive numbers are fed into it. Our test cell is thus defined as:

```
testCell :: Monad m => Cell m (Positive Int) Bool
testCell
  = arr getPositive >>> sumC >>> arr (>= 0)
```

We begin by running a cell repeatedly against a list of inputs, collecting its outputs:

```
embed
  :: Monad m
  =>        [a]
  -> Cell   m a b
  ->        m  [b]
embed [] _ = return []
embed (a : as) cell = do
  (b, cell') <- step cell a
  bs <- embed as cell'
  return $ b : bs
```

[7] This option is important for performance: E.g. for an audio application, a side effect on every sample can slow down unbearably.

If the input type `a` can be generated arbitrarily, then so can a list of `a`s. After running the cell with all inputs, we form the conjunction of all properties, with `QuickCheck`'s `conjoin`. Effects in `IO` can be embedded in `QuickCheck` [4] with the monad morphism `run`, and executed with `monadicIO`. Cobbling all those pieces together makes cells testable:

```
instance (Arbitrary a, Show a, Testable prop)
  => Testable (Cell IO a prop) where
  property cell = property
    $ \as -> monadicIO $ fmap conjoin
    $ embed as $ hoistCell run cell
```

Let us execute our test:

```
 > quickCheck testCell
+++ OK, passed 100 tests.
```

A large class of properties can be tested this way. We can unit test all components of a new version of our live program before reloading it. To go further, one could set up *stateful property-based testing* [9] for the live coding environment.

Migration tests Even better, we can test *before reloading* whether the newly migrated state would be valid. Given some tests on intermediate values in the computation, we collect all test properties in a `Writer` effect:

```
logTest
  :: Monad m
  => Cell m a prop
  -> Cell (WriterT [prop] m) a ()
logTest cell
  =   liftCell cell
  >>> arrM (return >>> tell)
```

Now the tests can be included in the definition of the whole live program without adding new outputs. When the program is built, we can optionally test the properties:

```
liveCheck
  :: Testable prop
  => Bool
  -> LiveProgram (WriterT [prop] IO)
  -> LiveProgram                 IO
liveCheck test = hoistLiveProgram performTests
  where
    performTests action = do
      (s, props) <- runWriterT action
      when test $ quickCheck $ conjoin props
      return s
```

The function `liveCheck True` will run `quickCheck` on all properties, while `liveCheck False` gives the "production" version of our program, with tests disabled. We launch two separate threads and run the test version in one of them and the production version in the other. Always reloading into the test version first, we can ensure that the migration will create valid state before migrating the live system.

6.3 External main loops

7 Conclusion

General purpose live coding can be simple and free from boilerplate. It is most naturally cast in the form of a Functional Reactive Programming (FRP) framework, and conforms well with the synchronous, arrowized paradigm in the tradition of Yampa and Dunai. The state migration is type-safe, and type-driven, in that it is derived generically from the datatype definition. By parametrizing the cells over arbitrary monads, and leveraging the exception monad, we can reason about effects and separate data flow aspects from control flow. The approach is extensible as debugging and testing methods can be added easily.

Further directions To use the framework in any setting beyond a toy application, wrappers have to be written that explicitly integrate it in the external loops of existing frameworks, such as web frontends and backends, OpenGL, `gloss` [12], or audio libraries. As a start, the multi-clock FRP library `rhine` [2] could be adapted to this approach.

The automatic migration only guarantees that the new state will typecheck. However, if further properties beyond the reach of Haskell's type system are expected to hold for the old state, those are not guaranteed for the new state. Within Haskell, quickchecking is our only hope. An extension such as refinement types (see e.g. [19] about LiquidHaskell) can automatically verify certain algebraic constraints, though. It would be a great enrichment to generalise automatic migration to such a type system.

The author thanks Iván Pérez for his work on Yampa, Dunai, and numerous other projects in the FRP world; the reviewers of the Haskell Symposium for very helpful comments that enriched and streamlined this work; Paolo Capriotti for the initial idea that led to monadic exception control flow; and the sonnen VPP team, especially Fabian Linges, for helpful discussions about hot code swap in Erlang.

The essence of live coding: Change the program, keep the state!

References

[1] Joe Armstrong. 2013. *Programming Erlang: software for a concurrent world.* Pragmatic Bookshelf.

[2] Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with type-level clocks. In *Proceedings of the 11th International Symposium on Haskell*. ACM, 145–157.

[3] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the International Conference on Functional Programming (ICFP 2000)*. ACM.

[4] Koen Claessen and John Hughes. 2002. Testing monadic code with QuickCheck. http://www.cse.chalmers.se/ rjmh/Papers/QuickCheckST.ps. *ACM SIGPLAN Notices* 37, 12 (2002), 47–59.

[5] Chris Done. 2014. foreign-store. https://github.com/esoeylemez/foreign-store.

[6] Andy Gill and Ross Paterson. 2004. transformers. https://hub.darcs.net/ross/transformers/.

[7] Gabriel Gonzalez. 2013. mmorph. https://github.com/Gabriel439/Haskell-MMorph-Library.

[8] Tim Harris, Simon Marlow, and Simon Peyton Jones. 2005. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (ppopp '05: proceedings of the tenth acm sigplan symposium on principles and practice of parallel programming ed.). ACM Press, 48–60. https://www.microsoft.com/en-us/research/publication/composable-memory-transactions/

[9] Fred Hebert. 2019. *Property-Based Testing with PropEr, Erlang, and Elixir.* Pragmatic Bookshelf.

[10] John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1-3 (2000), 67–111.

[11] Mark P Jones. 1995. Functional programming with overloading and higher-order polymorphism. In *International School on Advanced Functional Programming*. Springer, 97–136.

[12] Ben Lippmeier. 2010. Gloss. https://github.com/benl23x5/gloss.

[13] Ralf Lämmel and Simon Peyton Jones. 2003. Scrap your boilerplate: a practical approach to generic programming. ACM Press, 26–37. https://www.microsoft.com/en-us/research/publication/scrap-your-boilerplate-a-practical-approach-to-generic-programming/

[14] Alex McLean. 2014. Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*. ACM, 63–70.

[15] George H Mealy. 1955. A method for synthesizing sequential circuits. *The Bell System Technical Journal* 34, 5 (1955), 1045–1079.

[16] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, 51–64.

[17] Ross Paterson. 2001. A new notation for arrows. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 229–240.

[18] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, 33–44.

[19] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 269–282.

[20] Philip Wadler. 1989. Theorems for free!. In *FPCA*, Vol. 89. 347–359.

[21] Scott Wilson, David Cottle, and Nick Collins. 2011. *The SuperCollider Book.* The MIT Press.

[22] Kazu Yamamoto, Michael Snoyman, and Andreas Voellmy. [n. d.]. Warp. http://www.aosabook.org/en/posa/warp.html. In *The Performance of Open Source Applications.*