# The essence of live coding

Change the program, keep the state!

Manuel Bärenz

## Abstract

I present a general-purpose and type safe live coding framework in Haskell. The internal state of the live program is automatically migrated when performing hot code swap. The approach is then generalised to an easy to use FRP interface. It is parametrized by its side effects, separates data flow cleanly from control flow, and allows to develop live programs from reusable, modular components. Useful utilities for debugging and quickchecking are presented.

## 1 Introduction

For our purposes, a live coding framework has to allow a program to be updated, recompiled and reloaded, without interrupting its execution. The essential idea can be summed up in the motto of live coding:

**Change the program. Keep the state.**

In a dynamically typed language like Erlang [1], this is conceptually simple, but error-prone and tedious: There is no guarantee that the current state of the old program is compatible with the new program. Usually, one has to manually migrate the state to a new schema, but one cannot rule out failure. In webserver applications, the risk of runtime errors from incorrect state migrations may well outweigh the benefits.

The solution to avoid runtime errors is to employ a type checker. What is more, we will see how it is possible to arrive at an *automatic* migration function: By generic, type-driven programming.

Typed, automatically migrating live coding frameworks already exist [9], even for Haskell [7], but they are typically restricted to a particular domain such as audio or video. Here, an effect-polymorphic, universal live coding framework is developed. By being parametrised over arbitrary monads, it can connect to many backends and external libraries. Consequentially, the framework can be used in virtually any domain were live coding makes sense.

From an API perspective, Essence Of Livecoding follows the arrowized Functional Reactive Programming (FRP) viewpoint, in particular Dunai [8] and Rhine [2]. For the library user, this is essential in order to build programs modularly from reusable components, and to separate data flow from control flow. It is also essential from an implementation perspective, for two corresponding purposes: To build up state types modularly which greatly facilitates automatic and generic state migration, and to be able to migrate the control state[1] (the information which branch of the program is currently active).

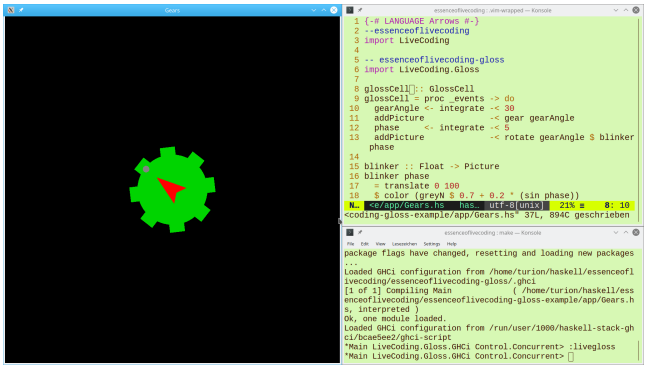

**Figure 1.** Example with Gloss and PulseAudio

```
data LiveProgram m = forall s . Data s
  => LiveProgram
  { liveState :: s
  , liveStep  :: s -> m s
  }
```

**Figure 2.** Definition of live programs

In the demonstration, I show in more detail how to live code a `gloss` program, and add or remove certain features from it. This program is then linked to a simple sound synthesizer using the PulseAudio backend.

The talk slides and a full article version are available at https://www.manuelbaerenz.de/. The source repository containing the library and examples (console, audio, video, web servers) is available at https://github.com/turion/essence-of-live-coding.

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

Our model of a live program consists of a state and an effectful state transition ("step") function, shown in Figure 2. The semantics of a live program is given by the side effects resulting from repeatedly applying `liveStep` to its current state, which is initialised with `liveState`. We hide the state type from the type signature by making it existential, since it will typically change over the course of a live coding session.

[1] A hard problem according to https://elm-lang.org/blog/interactive-programming.

The `Data` constraint will later be the crucial ingredient for a generic state migration function.

A live coding session is started via GHCi, into which we load a file containing a live program. We store it in an `MVar` and launch a separate thread that steps the state repeatedly and stores it in the `MVar` again. When we have edited the file, we reload it from GHCi, making use of the `foreign-store` [4] package to persist the concurrent variable. Now, we would like to update the state transition function `liveStep` in the live program, while keeping the current state, as far as possible. Here lies the fundamental challenge of live coding: The new program does not match the old state type.

The key insight is that we can use the datatype definition of the state to generically derive the correct migration. Assume, for example, that the state of some webserver is defined as:

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

And after reloading, a new record field was added:

```
data State = State
  { nVisitors  :: Int
  , lastAccess :: UTCTime
  }
```

Clearly, when migrating the old state to the new datatype, we want to preserve the `nVisitors` field. For `lastAccess` on the other hand, we cannot compute any sensible value. and thus have to initialise this field from the initial state of the new program. By reasoning about record fields and their types, we were able to find the best state migration. Similar generic criteria include matching constructor names, matching builtin types, and automatic newtype wrapping. All of these can be implemented in less than 100 lines using the `Data` type class from [6], as a function of this type signature:

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

It receives the new initial state and the old current state, and tries to migrate the old state as far as possible to the new state type. Wherever the automatic migration would perform suboptimally – as were the case if we wanted to migrate `nVisitors` from `Int` to `Integer` – it is possible to extend by a special case provided by the user.

## 3 Live coding as arrowized FRP

Writing out the complete state of the live program explicitly and separating its state from the step function is tedious. Instead, we want to develop modularly, and an arrowized FRP interface will allow us to do so. The live program definition is generalized to "cells"[2], shown in Figure 3. Additionally to a state and a step function, cells also have an input type `a` and an output type `b`. They can be composed sequentially, by feeding the output of one cell as input into another cell. By being instances of the `Arrow` type class, they can also be

[2]Cells are the building blocks of everything live.

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

**Figure 3.** The definition of a live coding cell

composed in parallel, giving rise to clear data flow declarations through the arrow syntax extension. The migration function has special cases for the state types of composed cells, making FRP cells suitable for live coding.

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)
```

Cells may also create side effects in a monad. A cell of type `Cell IO () a` produces data, using the `IO` monad, while `Cell IO a ()` consumes data. Composing effectful data producers with data processing cells, and finally with effectful consumers, we recover live programs as the special case of trivial input and output.

Using the `ExceptT` monad transformer, we also provide a monadic control flow interface based on type-safe exceptions. It enables the library user to permanently switch from one cell to another, triggered by events thrown anywhere within the cell. The crucial advantage of embedding control flow into cell states as an effect is that the migration function preserves the current control flow state (e.g., the information to which cell we currently switched) out of the box.

## 4 Tooling

For ease of use, custom GHCi commands are supplied that start a live program in a separate thread and allow reload it when it is edited. These cover ordinary live programs in `IO`, but also video and audio backends. Utilities for integration with other external loops are given.

It is easy to add debugging functionality to the framework, e.g. displaying the state or changing it after interacting with the user. In short, a debugger is itself a live program that can, as an effect, read and modify the state of an arbitrary other live program, i.e. it is of this type:

```
forall s . Data s => LiveProgram (StateT s m)
```

As examples, there are debuggers printing the current state to the console, displaying it graphically via `gloss` [5], or pausing the execution upon user interaction.

Testing live programs or cells with arbitrary input using QuickCheck [3] before reloading is often sensible. By collecting test results of components in a writer monad, we can modularly check properties of intermediate data. Thanks to the `Data` constraint, cells and live programs can be tested by generating arbitrary *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] Chris Done. 2014. foreign-store. https://github.com/esoeylemez/foreign-store.

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

[6] 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/

[7] 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.

[8] 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.

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