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

Appendix

Abstract

This file supplies more detailed discussions, such as the somewhat technical derivations of the **Applicative** and **Monad** type classes. It is not necessary to read it in order to appreciate the main paper, but some readers may want to satisfy their curiosity.

ACM Reference Format:

. 2020. The essence of live coding: Change the program, keep the state!: Appendix. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 4 pages. https://doi.org/10. 1145/nnnnnnnnnn

1 Arrows and typeclasses

The **Arrow** type class also allows for data-parallel composition:

(***)									
:: Mona	ad m								
=> Cel]	L m	а		b					
-> Cell	L m		С		d				
-> Cell	L m	(a,	c)	(b,	d)				

As for (>>>), the state type of the composed cell is the product type of the constituent states. In the resulting cell cell1 *** cell2, two inputs are received. First, cell1 is stepped with the input a, then cell2 is stepped with b.

The parallel composition operator has a dual, supplied by the **ArrowChoice** type class, which **Cells** implement: (+++)

,+++)		
:: Monad m		
=> Cell m	а	b
-> Cell m	С	d
-> Cell m (Either	• a c)	(Either b d)

Like cell1 *** cell2, its dual cell1 +++ cell2 holds the state of both cells. But while the former executes both cells, and consumes input and produces output for both of them,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-x/YY/MM...\$15.00

https://doi.org/10.1145/nnnnnnnnnnnn

the latter steps only one of them forward each time, depending on which input was provided. This enables basic control flow in arrow expressions, such as **if**- and **case**-statements. We can momentarily switch from one cell to another, depending on live input. For example, the following two cells are equal:

cellLR = proc lr -> do
 case lr of
 Left () -> returnA -< "Left!"
 Right () -> returnA -< "Right!"</pre>

cellLR'

= arr (const "Left!")
+++ arr (const "Right!")

The **ArrowLoop** class exists to enable recursive definitions in arrow expressions, and once again **Cells** implement it:

loop						
::	MonadFix	m				
=>	Cell	m	(a,	s)	(b,	s)
->	Cell	m	а		b	

A word of caution has to be issued here: The instance is implemented using the monadic fixed point operator mfix [1], and can thus crash at runtime if the current output of the intermediate value s is calculated strictly from the current input s.

We would like to have all basic primitives needed to develop standard synchronous signal processing components, without touching the **Cell** constructor anymore. One crucial bit is missing to achieve this goal: Encapsulating state. The most general such construction is the feedback loop:

feedback

::	(Monad	m,	Data	s)		
=>				s		
->	Cell	m	(a,	s)	(b,	s)
->	Cell	m	а		b	

Let us have a look at its internal state:

data Feedback sPrevious sAdditional = Feedback

```
{ sPrevious :: sPrevious
, sAdditional :: sAdditional
} deriving Data
```

In feedback sAdditional cell, the cell has state sPrevious, and to this state we add sAdditional. The additional state is received by cell as explicit input, and feedback hides it.

Note that feedback and loop are different. While loop provides immediate recursion, it doesn't add new state. feedback requires an initial state and delays it, but in turn it is always safe to use since it does not use mfix.

It enables us to write delays:

```
delay :: (Data s, Monad m) => s -> Cell m s s
delay s = feedback s $ arr swap
where
```

swap (sNew, s0ld) = (s0ld, sNew)

feedback can be used for accumulation of data. For example, sumC now becomes:

sumFeedback
:: (Monad m, Num a, Data a)
=> Cell m a a
sumFeedback = feedback 0 \$ arr
\$ \(a, accum) -> (accum, a + accum)

2 Monadic stream functions and final coalgebras

Cells mimick Dunai's [3] monadic stream functions (**MSF**s) closely. But can they fill their footsteps completely in terms of expressiveness? If not, which programs exactly can be represented as **MSF**s and which can't? To find the answer to these questions, let us reexamine both types.

With the help of a simple type synonym, the **MSF** definition can be recast in explicit fixpoint form:

type StateTransition m a b s = a -> m (b, s)

data MSF m a b = MSF

{ unMSF :: StateTransition m a b (MSF m a b)
}

This definition tells us that monadic stream functions are so-called *final coalgebras* of the **StateTransition** functor (for fixed m, a, and b). An ordinary coalgebra for this functor is given by some type s and a coalgebra structure map:

```
data Coalg m a b where
   Coalg
   :: s
   -> (s -> StateTransition m a b s)
   -> Coalg m a b
```

But hold on, the astute reader will intercept, is this not simply the definition of **Cell**? Alas, it is not, for it lacks the type class restriction **Data** s, which we need so dearly for the type migration. Any cell is a coalgebra, but only those coalgebras that satisfy this type class are a cell.

Oh, if only there were no such distinction. By the very property of the final coalgebra, we can embed every coalgebra therein:

```
finality :: Monad m => Coalg m a b -> MSF m a b
finality (Coalg state step) = MSF $ \a -> do
  (b, state') <- step state a
  return (b, finality $ Coalg state' step)</pre>
```

And analogously, every cell can be easily made into an **MSF** without loss of information:

```
finalityC :: Monad m => Cell m a b -> MSF m a b
finalityC Cell { ... } = MSF $ \a -> do
  (b, cellState') <- cellStep cellState a
  return (b, finalityC $ Cell cellState' cellStep)</pre>
```

And the final coalgebra is of course a mere coalgebra itself:

```
coalgebra :: MSF m a b -> Coalg m a b
coalgebra msf = Coalg msf unMSF
```

But we miss the abilty to encode **MSF**s as **Cells** by just the **Data** type class:

coalgebraC

:: Data (MSF m a b)
=> MSF m a b
-> Cell m a b
coalgebraC msf = Cell msf unMSF

We are out of luck if we would want to derive an instance of **Data** (**MSF** m a b). Monadic stream functions are, well, functions, and therefore have no **Data** instance. The price of **Data** is loss of higher-order state. Just how big this loss is will be demonstrated in the following section.

We would like to adopt this approach here, but we are forewarned: **Cells** are slightly less expressive than Dunai's stream functions, due to the **Data** constraint on the internal state.

3 Monads for control flow

Recall the definition of **CellExcept** from the main article. The goal is to define a **Monad** instance for it.

An existential crisis After having done away with return already, we want to implement the holy grail of Haskell, *bind*: (>>=)

```
:: Monad m
=> CellExcept m a b e1
-> (e1 -> CellExcept m a b e2)
-> CellExcept m a b e2
```

Unwrapped from the **newtype**, it would have a type signature like this:

```
bindCell
```

::	Monad	m		
=>	Cell (ExceptT e1	m)	а	b
->	(e1 -> Cell (ExceptT e2	m)	а	b)
->	Cell (ExceptT e2	m)	а	b

Its intended semantics is straightforward: Execute the first cell until it throws an exception, then use this exception to choose the second cell, which is to be executed subsequently.

But what is the state type of the result? When implementing cell `bindCell` handler, we would need to specify some type of internal state. Before the exception is thrown, this should certainly be the state of cell, but what afterwards? Worse, the state type of handler e1 depends on the The essence of live coding: Change the program, keep the state!

value of the exception e1! Without having ordered them, dependent types suddenly jump in our faces, in the disguise of existential quantification.¹ Impulsively, we want to shove the existential state type back where it came from. Why not simply store handler e1 as state once the exception e1 was thrown, and use the aptly named step from Section 2 in the main article as step function? (This is basically the final encoding from Section 2, and exactly how Dunai implements this feature.) But it is not possible, because **Cells** are not **Data**.

Live bind Accepting a setback, but not final defeat, we note that the fundamental issue is the inability to typecheck the state of the cell we would like to switch to, at least not if this cell has to depend on the thrown exception.

If we were able to offer the typechecker a state type immediately, and defer the actual choice to a later moment, we can succeed. What if we were to supply the thrown exception not when instantiating the new cell, but while it is running, as a live input?

This is exactly what *live bind* does:

(>>>=) ::	(Data e1, Me	onad	m)			
=> Cell	(ExceptT e1		m)		а	b
-> Cell	(ExceptT	e2	m)	(e1,	a)	b
-> Cell	(ExceptT	e2	m)		а	b

Its syntax is a combination of the monadic bind >>= and the sequential composition operator >>>. Its semantics is described as follows: Before an exception is thrown, it is initialised with the initial state of both cells. If no exception occurs, only the state of the first cell is stepped. As soon as an exception is thrown, the state is switched to containing just the exception and the state of the second cell. The first cell is discarded, all information in it relevant to the rest of the live program must be passed into the exception. The thrown exception e1 is passed as an additional input to the second cell, which is then executed indefinitely. The resulting cell may throw an exception of its own, which can in turn be handled again. The state of cell1 >>>= cell2 not only holds the state of the individual cells, but also the *control flow state*, that is, it designates which cell currently has control.

Applying it to Applicative If we are allowed to read the first exception during the execution of the second cell, we can simply re-raise it once the second exception is thrown: andThen

::	(Data	a e1, <mark>Mona</mark>	d m)				
=>	Cell	(ExceptT	e1		m)	а	b
->	Cell	(ExceptT		e2	m)	а	b

¹Unfortunately, we cannot achieve the goal by reverting to the preliminary definition of live programs, which did not make the state type existential. The corresponding **Cell** definition would not be an instance of **Arrow** anymore, and the type signatures would bloat indefinitely. But worst of all, bindCell would restrict the state of all cells the handler could output to the same type! Except in very simple cases, we could not branch between different cells at all.

```
-> Cell (ExceptT (e1, e2) m) a b
cell1 `andThen` Cell { ... } = cell1 >>>= Cell
{ cellStep = \state (e1, a) ->
    withExceptT (e1, ) $ cellStep state a
, ...
```

}

Given two **Cells**, the first may throw an exception, upon which the second cell gains control. As soon as it throws a second exception, both exceptions are thrown as a tuple.

At this point, we unfortunately have to give up the efficient **newtype**. The spoilsport is, again the type class **Data**, to which the exception type e1 is subjected (since the exception must be stored during the execution of the second cell). But the issue is minor, it is fixed by defining the *free functor*, or *Co-Yoneda construction*:

```
data CellExcept m a b e = forall e' .
Data e' => CellExcept
{ fmapExcept :: e' -> e
, cellExcept :: Cell (ExceptT e' m) a b
}
```

While ensuring that we only store cells with exceptions that can be *bound*, we do not restrict the parameter type e.

It is known that this construction gives rise to a **Functor** instance for free:

```
instance Functor (CellExcept m a b) where
fmap f CellExcept { ... } = CellExcept
{ fmapExcept = f . fmapExcept
, ...
}
```

The **Applicative** instance arises from the work we have done so far. pure is implemented by throwing a unit and transforming it to the required exception, while sequential application is a bookkeeping exercise around the previously defined function andThen:

```
instance Monad m
=> Applicative (CellExcept m a b) where
pure e = CellExcept
{ fmapExcept = const e
, cellExcept = constM $ throwE ()
}
CellExcept fmap1 cell1 <*>
CellExcept fmap2 cell2 = CellExcept { ... }
where
fmapExcept (e1, e2) = fmap1 e1
$ fmap2 e2
cellExcept = cell1 `andThen` cell2
```

3.1 Finite patience with monads

While **Applicative** control flow is certainly appreciated, and the live bind combinator >>>= is even more expressive, it still encourages boilerplate code like the following:

```
throwBool >>>= proc (bool, a) -> do
  if bool
  then foo1 -< a
  else foo2 -< a</pre>
```

The annoyed library user will promptly abbreviate this pattern:

bindBool

```
:: Monad m
=> Cell (ExceptT Bool m) a b
-> (Bool -> Cell (ExceptT e m) a b)
-> Cell (ExceptT e m) a b
bindBool cell handler
= cell >>>= proc (bool, a) -> do
        if bool
        then handler True -< a
        else handler False -< a</pre>
```

But, behold! Up to the **CellExcept** wrapper, we have just implemented bind, the holy grail which we assumed to be denied! The bound type is restricted to **Bool**, admitted, but if it is possible to bind **Bool**, then it is certainly possible to bind (**Bool**, **Bool**), by nesting two **if**-statements. By the same logic, we can bind (**Bool**, **Bool**, **Bool**) and so on (and of course any isomorphic type as well). In fact, *any finite type* can be bound in principle, by embedding it in such a binary vector. For what follows, we will only consider finite algebraic datatypes. These are essentially the unit type (or any single constructor type), sum types (or multiple constructor types) of other finite types, and product types (or multiple argument constructors). Recursive datatypes are infinite in Haskell (consider, e.g., the list type).

How can it be that a general bind function does not typecheck, but we can implement one for any finite type? If the exception type e is finite, the type checker can inspect the state type of the cell handler e for every possible exception value, *at compile time*. All that is needed is a little help to spell out all the possible cases, as has been done for **Bool**.

But certainly, we don't want to write out all possible values of a type before we can bind it. Again, the Haskellers' aversion to boilerplate has created a solution that can be tailored to our needs: Generic deriving [2]. We simply need to implement a bind function for generic sum types and product types, then this function can be abstracted into a type class, and GHC can infer a default instance for every algebraic data type by adding a single line of boilerplate. Since the type class is defined for all finite algebraic datatypes, we will call it **Finite**. Any user-contributed or standard type can be an instance this type class, given that it is not recursive.

It is possible to restrict the previous **CellExcept** definition by the typeclass:

```
data CellExcept m a b e = forall e' .
  (Data e', Finite e') => CellExcept
  { fmapExcept :: e' -> e
```

, cellExcept :: Cell (ExceptT e' m) a b }

Implementing the individual bind functions for sums and products, and finally writing down the complete **Monad** instance is a tedious exercise in Generic deriving.

We can save on boiler plate by dropping the Coyoneda embedding for an "operational" monad:

```
data CellExcept m a b e where
  Return :: e -> CellExcept m a b e
  Bind
    :: CellExcept m a b e1
    -> (e1 -> CellExcept m a b e2)
    -> CellExcept m a b e2
  Try
    :: (Data e, Finite e)
    => Cell (ExceptT e m) a b
```

-> CellExcept m a b e

The Monad instance is now trivial:

instance	Monad	m =>	Monad	(CellExcept	m	а	b)	where
return	= Retu	urn						
(>>=) =	Bind							

As is typical for operational monads, all of the effort now goes into the interpretation function:

```
runCellExcept
:: Monad m
=> CellExcept m a b e
-> Cell (ExceptT e m) a b
runCellExcept (Bind (Try cell) g)
= cell >>>= commute (runCellExcept . g)
runCellExcept ... = ...
```

As a slight restriction of the framework, throwing exceptions is now only allowed for finite types:

try
 :: (Data e, Finite e)
 => Cell (ExceptT e m) a b
 -> CellExcept m a b e
try = Try

In practice however, this is less often a limitation than first assumed, since in the monad context, calculations with all types are allowed again.

References

- Levent Erkök. 2002. Value recursion in monadic computations. Ph.D. Dissertation. OGI School of Science & Engineering at OHSU.
- [2] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A generic deriving mechanism for Haskell. In ACM Sigplan Notices, Vol. 45. ACM, 37–48.
- [3] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In Proceedings of the 9th International Symposium on Haskell (Nara, Japan) (Haskell 2016). ACM, 33–44.