Appendix E

Denotational Semantics of Sodium

Revision 1.1 – 24 July 2015

Introduction

This document is the formal specification of the semantics of Sodium, an FRP system based on the concepts from Conal Elliott's paper Push-Pull FRP. The code in this document is in Haskell.

The executable version of this specification can be found at
https://github.com/SodiumFRP/sodium/blob/master/denotational/

Revision History

- 1.0 – 19 May 2015 – first version
- 1.1 – 24 July 2015 – the times for streams changed to increasing instead of non-decreasing so that multiple events per time are no longer representable.

Data types

Sodium has two data types:

- *Stream a*–a sequence of events, equivalent to Conal's *Event*
- *Cell a*–a value that changes over time, equivalent to Conal's *Behavior*

We replace Conal's term *event occurrence* with *event*.

Primitives

We define a type *T* representing time that is a total order. For the *Split* primitive, we need to extend that definition to be hierarchical such that for any time *t* we can add children numbered with natural numbers, that are all greater than t but smaller than any greater sibling of *t*. In the executable version we have used the type

type T = [Int]

with comparison defined such that early list elements have have precedence over later ones.

Sodium has sixteen primitives. Primitives marked with * are actually non-primitive, because they can be defined in terms of other primitives:

- *Never :: Stream a*
- *MapS :: (a → b) → Stream a → Stream b*
- *Snapshot* :: (a → b → c) → Stream a → Cell b → Stream c*
- *Merge :: Stream a → Stream a → (a → a → a) → Stream a*
- *Filter :: (a → Bool) → Stream a → Stream a*
- *SwitchS :: Cell (Stream a) → Stream a*
- *Execute :: Stream (Reactive a) → Stream a*
- *Updates :: Cell a → Stream a*
- *Value :: Cell a → T → Stream a*
- *Split :: Stream [a] → Stream a*
- *Constant* :: a → Cell a*
- *Hold :: a → Stream a → T → Cell a*
- *MapC :: (a → b) → Cell a → Cell b*
- *Apply :: Cell (a → b) → Cell a → Cell b*
- *SwitchC :: Cell (Cell a) → T → Cell a*
- *Sample :: Behavior a → T → a*

Reactive is a helper monad that is equivalent to *Reader T*. It represents a computation that is executed at a particular instant in time. Its declaration:

data Reactive a = Reactive { run :: T → a }

Execute works with this monad. *Reactive* is part of the public interface of Sodium used to construct the four primitives that take a *T* argument representing the time when that primitive was constructed, namely *Value*, *Hold*, *SwitchB* and *Sample*. The output values of those four primitives can never be sampled before the time *t* they were constructed with for these reasons:

- The public interface only allows *Value*, *Hold*, *SwitchB* and *Sample* to be constructed through *Reactive*.
- The time at which the simulation is sampled is always increasing.

- The public interface only allows *Reactive* to be resolved once the simulation has reached time *t*.
- The public interface only allows streams and cells to be sampled at the current simulation time.

I define semantic domains *S a* and *C a* for streams and cells:

type S a = [(T, a)] for increasing T values

type C a = (a, [(T, a)]) for increasing T values

S a represents a list of time/value pairs describing the events of the stream. *C a* represents (initial value, steps) for the cell: the initial value pertains to all times before the first step, and the time/value pairs give the discrete steps in the cell's value.

I define these semantic functions to transform streams and cells to their semantic domains:

occs :: Stream a → S a

steps :: Cell a → C a

C a is different to Conal Elliott's semantic domain for *behavior*, which was

type B a = T → a

The reason for this choice is that it makes *Updates* and *Value* possible, and it allows the cell variant of switch to take Cell (Cell a) as its argument instead of Cell a → Stream (Cell a), effectively decoupling it from stepper/hold functionality. Something roughly equivalent to Conal's *switcher* can be defined as follows, if we posit that [0] is the smallest possible value of *T*:

switcher :: Cell a → Stream (Cell a) → Cell a

switcher c s = SwitchC (Hold c s [0]) [0]

We can derive Conal's *B a* from *C a* with an *at* function:

at :: C a → T → a

at (a, sts) t = last (a : map snd (filter (\(tt, a) → tt < t) sts))

Test cases

Now I will give the definitions of the semantic functions *occs* and *steps* for each primitive, with test cases to show things are working as expected. *MkStream* is the inverse of *occs*, constructing a *Stream a* from an *S a*. We use it to feed input into our test cases.

Never

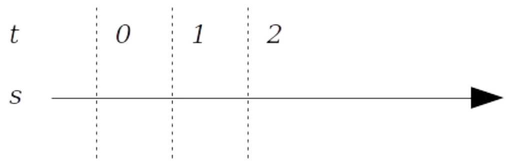
Never :: Stream a

An stream that never fires.

occs Never = []

Test cases

let s = Never



MapS

MapS :: (a → b) → Stream a → Stream b

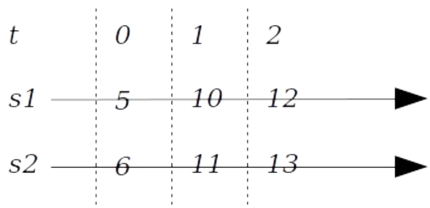
Map a function over a stream.

occs (MapS f s) = map (\(t, a) → (t, f a)) (occs s)

TEST CASES

let s1 = MkStream [([0], 5), ([1], 10), ([2], 12)]

let s2 = MapS (1+) s1



Snapshot

Snapshot :: (a → b → c) → Stream a → Cell b → Stream c

Capture the cell's observable value as at the time when the stream fires.

occs (Snapshot f s c) = map (\(t, a) → (t, f a (at stsb t))) (occs s)

where stsb = steps c

Note: Snapshot is non-primitive. It can be defined in terms of *MapS*, *Snapshot* and *Execute* thus:

snapshot2 f s c = Execute (MapS (\a → f a <$> sample c) s)

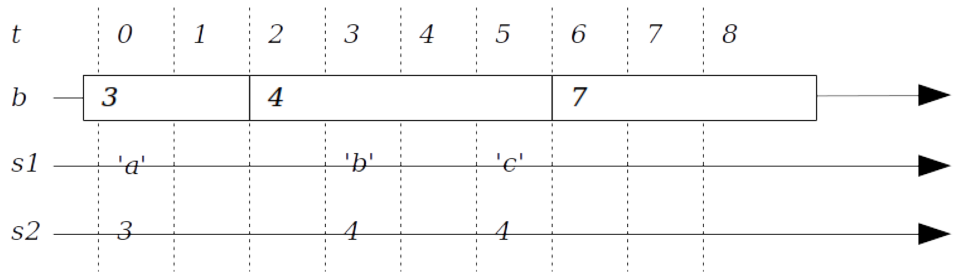
NOTE To make it easier to see the underlying meaning, we're diagramming cells in their "cooked" form with the observable values at would give us and vertical lines to indicates the steps, not directly in their B a representation of initial value and steps.

TEST CASES

let c = Hold 3 (MkStream [([1], 4), ([5], 7)]) [0]

let s1 = MkStream [([0], 'a'), ([3], 'b'), ([5], 'c')]

let s2 = Snapshot (flip const) s1 c



Merge

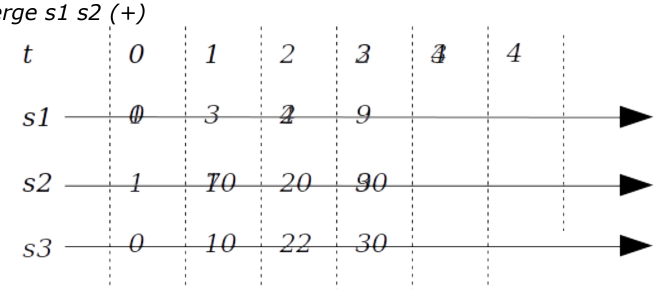
Merge :: Stream a → Stream a → (a → a → a) → Stream a

Merge the events from two streams into one. A stream can have simultaneous events, meaning two or more events with the same value *t*, which have an order. s3 in the diagram below gives an example. Merge is left-biased, meaning that for time *t*, events originating in the left input event are output before ones from the right.

```
occs (Merge sa sb) = coalesce f (knit (occs sa) (occs sb))
  where knit ((ta, a):as) bs@((tb, _):_) | ta <= tb = (ta, a) : knit as bs
        knit as@((ta, _):_) ((tb, b):bs) = (tb, b) : knit as bs
        knit as bs = as ++ bs
coalesce :: (a → a → a) → S a → S a
coalesce f ((t1, a1):(t2, a2):as) | t1 == t2 = coalesce f ((t1, f a1 a2):as)
coalesce f (ta:as) = ta : coalesce f as
coalesce f [] = []
```

Test cases

```
let s1 = MkStream [([0], 0), ([2], 2)]
let s2 = MkStream [([1], 10), ([2], 20), ([3], 30)]
let s3 = Merge s1 s2 (+)
```



Filter

Filter :: (a → Bool) → Stream a → Stream a

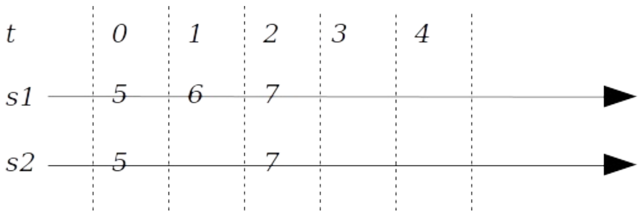
Filter events by a predicate.

occs (Filter pred s) = filter (\(t, a) → pred a) (occs s)

TEST CASES

let s1 = MkStream [([0], 5), ([1], 6), ([2], 7)]

let s2 = Filter odd s1



SwitchS

SwitchS :: Cell (Stream a) → Stream a

Act like the stream that is the current value of the cell.

```
occs (SwitchS c) = scan Nothing a sts
  where (a, sts) = steps c
        scan mt0 a0 ((t1, a1):as) =
           filter (\(t, a) → maybe True (t >) mt0 && t <= t1) (occs a0)
           ++ scan (Just t1) a1 as
        scan mt0 a0 [] =
           filter (\(t, a) → maybe True (t >) mt0) (occs a0)
```

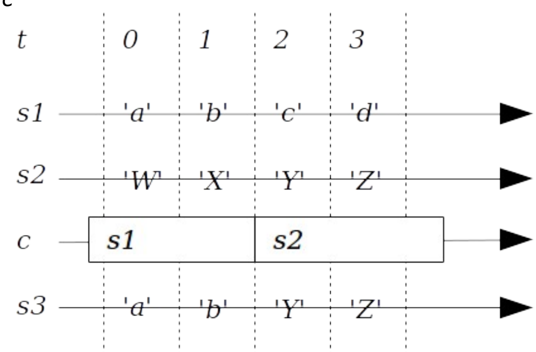
TEST CASES

let s1 = MkStream [([0], 'a'), ([1], 'b'), ([2], 'c'), ([3], 'd')]

let s2 = MkStream [([0], 'W'), ([1], 'X'), ([2], 'Y'), ([3], 'Z')]

let c = Hold s1 (MkStream [([1], s2)]) [0]

let s3 = SwitchS c



Execute

Execute :: Stream (Reactive a) → Stream a

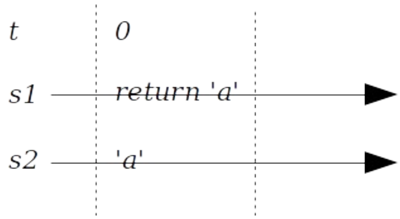
Unwrap the Reactive helper monad value of the occurrences passing it the time of the occurence. This is commonly used with when we want to construct new logic to activate with SwitchC or SwitchS.

occs (Execute s) = map (\(t, ma) → (t, run ma t)) (occs s)

TEST CASES

let s1 = MkStream [([0], return 'a')]

let s2 = Execute s1



Updates

Updates :: Cell a → Stream a

A stream representing the steps in a cell, which breaks the principle of non-detectability of cell steps. Updates must therefore be treated as an operational primitive, for use only in defining functions that don't expose cell steps to the caller. If the cell had been the Hold of stream s, it would be equivalent to Coalesce (flip const) s.

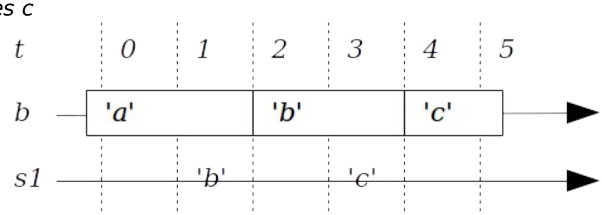
occs (Updates c) = sts

  where (_, sts) = steps c

TEST CASES

let c = Hold 'a' (MkStream [([1], 'b'), ([3], 'c')]) [0]

let s = Updates c



Value

Value :: Cell a → T → Stream a

Like *Updates* except it also fires once with the current cell value at the time *t0* when it is constructed. Also like *Updates*, Value breaks the non-detectability of cell steps so is treated as an operational primitive.

occs (Value c t0) = coalesce (flip const) ((t0, a) : sts)

  where (a, sts) = chopFront (steps c) t0

chopFront :: C a → T → C a

chopFront (i, sts) t0 = (at (i, sts) t0, filter (\(t, a) → t >= t0) sts)

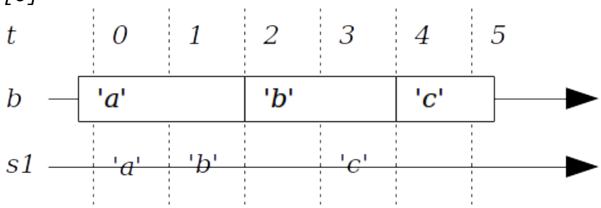
Note there is a boundary case: where a step occurs at *t0*, the initial value and the first step value are output as simultaneous events.

Note: Value has the property that it can create an event occurrence out of nothing. It is possible to argue that it is re-constructing an event occurrence that we can prove exists – the one that drives the Execute that must have executed this instance of Value. It is the same event occurrence that *Sample* implies the existence of if it is seen as being based on *Snapshot*.

TEST CASES

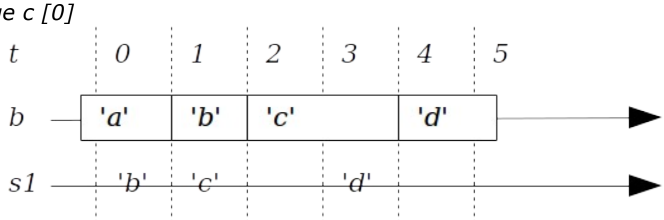
let c = Hold 'a' (MkStream [([1], 'b'), ([3], 'c')]) [0]

let s = Value c [0]



let c = Hold 'a' (MkStream [([0], 'b'), ([1], 'c'), ([3], 'd')]) [0]

let s = Value c [0]



Split

Split :: Stream [a] → Stream a

Put the values into newly created child time steps.

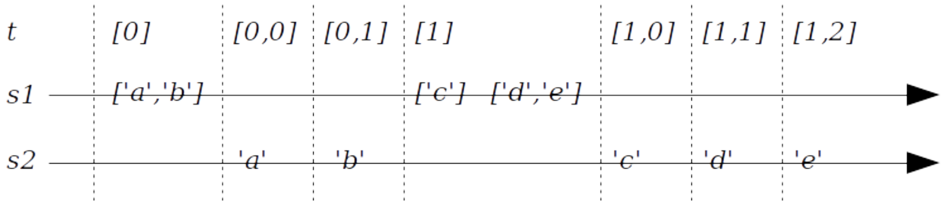
occs (Split s) = concatMap split (coalesce (++) (occs s))

where split (t, as) = zipWith (\n a → (t++[n], a)) [0..] as

TEST CASES

let s1 = MkStream [([0], ['a', 'b']), ([1],['c']),([1],['d','e'])]

let s2 = Split s1



Constant

Constant :: a → Cell a

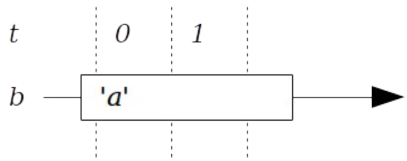
A cell with an initial value but no steps.

steps (Constant a) = (a, [])

Note: Constant is non-primitive. It can be defined in terms of Hold and Never.

Test cases

let c = Constant 'a'



Hold

Hold :: a → Stream a → T → Cell a

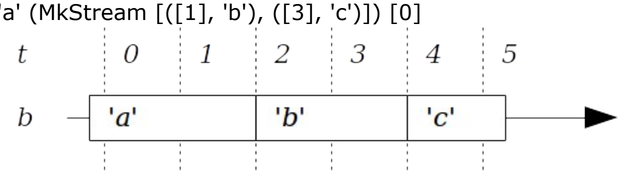
A cell with an initial value of a and the specified steps, ignoring any steps before specified *t0*.

steps (Hold a s t0) = (a, coalesce (flip const)
(filter (\(t, a) → t >= t0) (occs s)))

We coalesce to maintain the invariant that step times in *C a* are increasing. Where input events are simultaneous, the last is taken. Events before *t0* are discarded.

Test cases

let c = Hold 'a' (MkStream [([1], 'b'), ([3], 'c')]) [0]



MapC

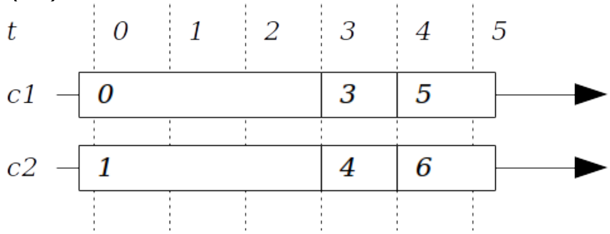
MapC :: (a → b) → Cell a → Cell b

Map a function over a cell.

steps (MapC f c) = (f a, map (\(t, a) → (t, f a)) sts)
where (a, sts) = steps c

TEST CASES

```
let c1 = Hold 0 (MkStream [([2], 3), ([3], 5)]) [0]
let c2 = MapC (1+) c1
```



Apply

Apply :: Cell (a → b) → Cell a → Cell b

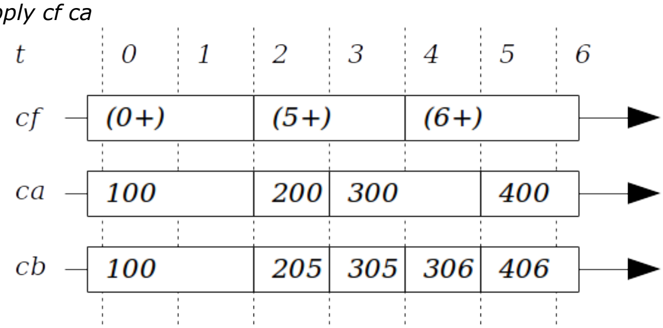
Applicative “apply” operation, as the basis for function lifting.

```
steps (Apply cf ca) = (f a, knit f fsts a asts)
   where (f, fsts) = steps cf
        (a, asts) = steps ca
        knit _ ((tf, f):fs) a as@((ta, _):_) | tf < ta = (tf, f a) : knit f fs a as
        knit f fs@((tf, _):_) _ ((ta, a):as) | tf > ta = (ta, f a) : knit f fs a as
        knit _ ((tf, f):fs) _ ((ta, a):as) | tf == ta = (tf, f a) : knit f fs a as
        knit _ ((tf, f):fs) a [] = (tf, f a) : knit f fs a []
        knit f [] _ ((ta, a):as) = (ta, f a) : knit f [] a as
        knit _ [] _ [] = []
```

Note the “no glitch” rule: Where both cells are updated in the same time t we output only one output step.

TEST CASES

```
let cf = Hold (0+) (MkStream [([1], (5+)), ([3], (6+))]) [0]
let ca = Hold (100 :: Int) (MkStream [([1], 200), ([2], 300), ([4], 400)]) [0]
let cb = Apply cf ca
```



SwitchC

SwitchC :: Cell (Cell a) → T → Cell a

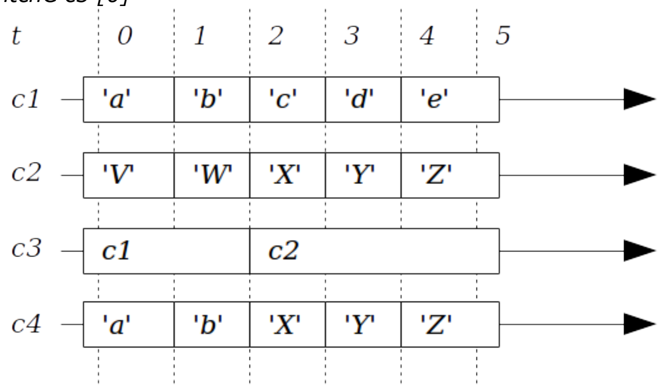
Act like the current cell in the cell.

```
steps (SwitchC c t0) = (at (steps (at (steps c) t0)) t0,
        coalesce (flip const) (scan t0 a sts))
    where (a, sts) = steps c
          scan t0 a0 ((t1, a1):as) =
              let (b, stsb) = normalize (chopBack (chopFront (steps a0) t0) t1)
              in  ((t0, b) : stsb) ++ scan t1 a1 as
          scan t0 a0 [] =
              let (b, stsb) = normalize (chopFront (steps a0) t0)
              in  ((t0, b) : stsb)
          normalize :: C a → C a
          normalize (_, (t1, a) : as) | t1 == t0 = (a, as)
          normalize as = as
          chopBack :: C a → T → C a
          chopBack (i, sts) tEnd = (i, filter (\(t, a) → t < tEnd) sts)
```

The purpose of normalize is to get rid of simultaneousness returned by *chopFront* where the first step occurs at the chop point *t0*. It discards the initial value and replaces that with the first step value. This is different to how *Value* uses *chopFront*: in that case we keep the simultaneous events.

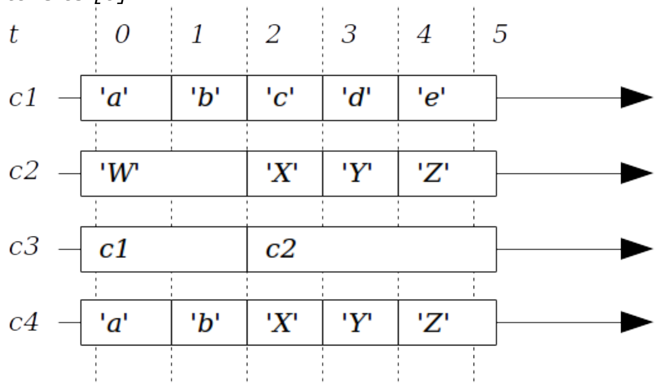
Test cases

```
let c1 = Hold 'a' (MkStream [([0], 'b'), ([1], 'c'), ([2], 'd'), ([3], 'e')]) [0]
let c2 = Hold 'V' (MkStream [([0], 'W'), ([1], 'X'), ([2], 'Y'), ([3], 'Z')]) [0]
let c3 = Hold c1 (MkStream [([1], c2)]) [0]
let c4 = SwitchC c3 [0]
```

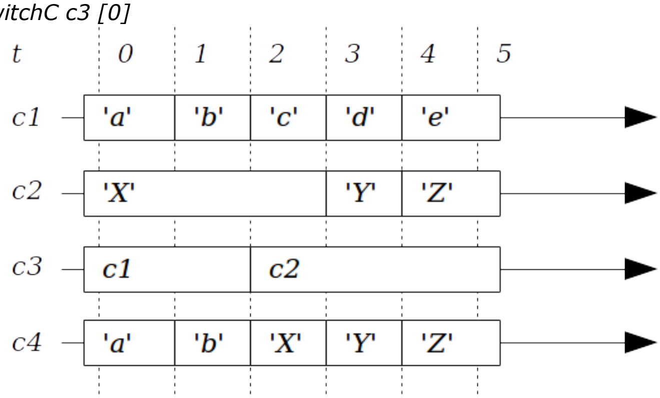



```
let c1 = Hold 'a' (MkStream [([0], 'b'), ([1], 'c'), ([2], 'd'), ([3], 'e')]) [0]
let c2 = Hold 'W' (MkStream [([1], 'X'), ([2], 'Y'), ([3], 'Z')]) [0]
let c3 = Hold c1 (MkStream [([1], c2)]) [0]
```

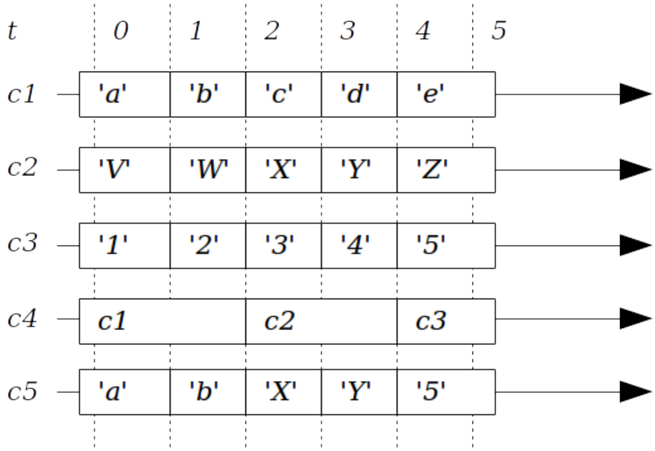
let c4 = SwitchC c3 [0]



let c1 = Hold 'a' (MkStream [([0], 'b'), ([1], 'c'), ([2], 'd'), ([3], 'e')]) [0]
let c2 = Hold 'X' (MkStream [([2], 'Y'), ([3], 'Z')]) [0]
let c3 = Hold c1 (MkStream [([1], c2)]) [0]
let c4 = SwitchC c3 [0]



let c1 = Hold 'a' (MkStream [([0], 'b'), ([1], 'c'), ([2], 'd'), ([3], 'e')]) [0]
let c2 = Hold 'V' (MkStream [([0], 'W'), ([1], 'X'), ([2], 'Y'), ([3], 'Z')]) [0]
let c3 = Hold '1' (MkStream [([0], '2'), ([1], '3'), ([2], '4'), ([3], '5')]) [0]
let c4 = Hold c1 (MkStream [([1], c2), ([3], c3)]) [0]
let c5 = SwitchC c4 [0]



Sample

Sample :: Cell a → T → a

Extract the observable value of the cell at time t.

sample :: Cell a → Reactive a

sample c = Reactive (at (steps c))

TEST CASES

let c = Hold 'a' (MkStream [([1], 'b')]) [0]

let a1 = run (sample c) [1]

let a2 = run (sample c) [2]