Functional reactive programming and clock calculus in Haskell

Manuel Bärenz (Bamberg) and Ivan Perez (Nottingham)

December 4, 2015

Objective

A framework for reactive programming, that. . .

. . . is functional (here: Haskell), so we could have

- reasoning about reactive programs,
- determinism, automatic test generation, monads, . . .

. . . can model sideeffects easily

. . . has explicit clocks

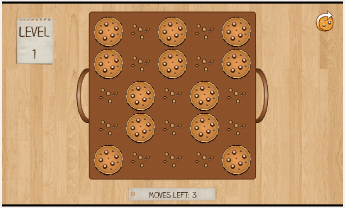
- in the type system!

. . . gives a decent API for implementing multi-rate systems and resampling

- with separation of data and synchronisation aspects.

Yampa

- Precursor: Conal Elliot's Fran
- Henrik Nilson, Paul Hudak et al.
- Signal flow language embedded in Haskell
- Real time or simulation
- No space or time leaks (in the framework)



Why Haskell?

- Use existing compilers
- Use existing, very flexible type system
- Lots of libraries

Disadvantages

- Realtime < 1 ms..?
- $\sim$ Garbage collector
- Compile to microprocessors & embedded systems?
  (Executable size ~ 1 MB)

Definition: Causal stream function

$$\begin{array}{llllllll} \textbf{data} & StreamF & & a & & b & & \\ = & \underbrace{StreamF}_{\text{Constructor}} & (& \underbrace{a}_{\text{Input}} & \to (& \underbrace{b,}_{\text{Output}} & \underbrace{StreamF\ a\ b}_{\text{New state}} &)) \end{array}$$

Example

$$streamFunction :: StreamF\ a\ b$$

$$a \longrightarrow \boxed{streamFunction} \longrightarrow b$$

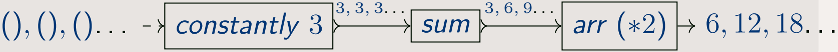
Composability

Use composition $\rightarrowtail$ for data flow:

$$(\rightarrowtail) :: \mathit{StreamF}\ a\ b \to \mathit{StreamF}\ b\ c \to \mathit{StreamF}\ a\ c$$

Example

$$\begin{aligned}&\mathit{streamFunction} :: \mathit{StreamF}\ ()\ \mathit{Int}\\&\mathit{streamFunction} = \mathit{constantly}\ 3 \rightarrowtail \mathit{sum} \rightarrowtail \mathit{arr}\ (*2)\end{aligned}$$



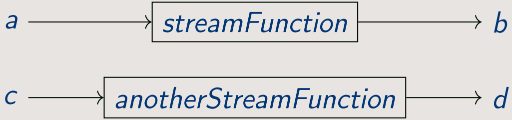
Modularity

Combine several stream functions parallely:

$$(***) :: \mathit{StreamF}\ a\ b \rightarrow \mathit{StreamF}\ c\ d \rightarrow \mathit{StreamF}\ (a, c)\ (c, d)$$

Example

$$\mathit{streamFunction} *** \mathit{anotherStreamFunction}$$

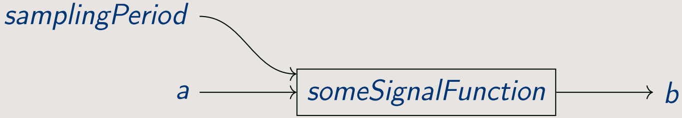


$a \longrightarrow \boxed{\mathit{streamFunction}} \longrightarrow b$

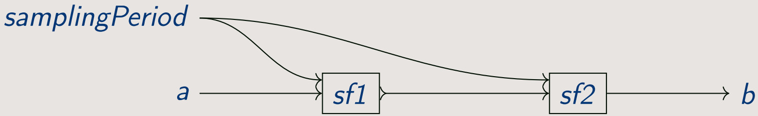
$c \longrightarrow \boxed{\mathit{anotherStreamFunction}} \longrightarrow d$

Signals as streams with a time input

$\textbf{type}\ SignalFunction\ a\ b = StreamF\ (Double, a)\ b$



Composition



Main loop of Yampa (simplified pseudocode)

```
reactimateYampa sensor signalF actuator = do
  samplingPeriod ← measureSamplingPeriod
  a ← sensor
  let (b, newSignalF) = signalF (samplingPeriod, a)
  actuator b
  reactimateYampa sensor newSignalF actuator
```

- Sensors and actuators aren't modular
- Sensors and actuators are static
- No side effects in the signal functions (no debugging, global state, exception handling etc.)
- One global single clock at only one speed (as fast as possible)

$sensor :: m\ a$	A side effect with result a
$actor :: b \to m\ ()$	A side effect depending on b, with result void

m controls the strength of the side effect:

Monad m	Side effect
IO	Any possible side effect ("Input/Output")
$Identity$	No side effect
$State$	Global state variable
. . .	Your favourite embedded DSL

. . . exceptions, debugging, logging,. . .

Actual definition: Monadic stream function

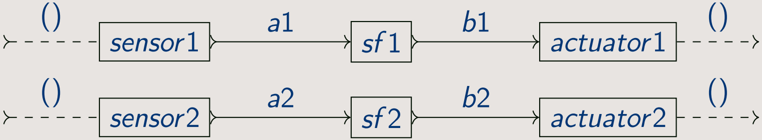
```
data MStreamF m a b
  = MStreamF (a → m (b, MStreamF a b))
type MSignalF  m a b = MStreamF m (Double, a) b
type Sensor    m a   = MSignalF m  ()          a
type Actuator  m a   = MSignalF m  a           ()
```

Treat sensors, signal functions and actuators the same:

```
reactiveProgram ::  MSignalF m () ()
reactiveProgram = sensor ↣ signalFunction ↣ actuator
```

Parallel composition

$$\begin{aligned} \mathit{bigProgram} = {} & (\mathit{sensor1} \rightarrowtail \mathit{sf1} \rightarrowtail \mathit{actuator1}) \\ & \qquad\qquad *** \\ & (\mathit{sensor2} \rightarrowtail \mathit{sf2} \rightarrowtail \mathit{actuator2}) \end{aligned}$$



Debugging as a side effect "in the middle"

Trace (pseudocode)

```
tracingExample =
  someSensor ↣ trace "Sensor signal: "
   ↣ furtherProcessing ↣ actuator
```

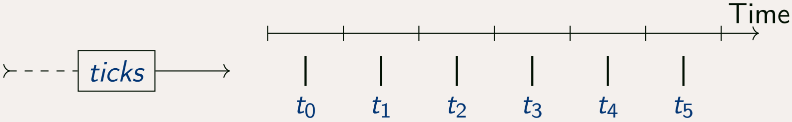
Debugger pause (actual code)

```
debuggingExample = reactimateR $
  constantly (3 :: Double) ↣ integral
   ↣ pauseOn (λx → x ⩾ 5 ∧ x ⩽ 6) "between 5 and 6: "
   ↣ liftSF print @@ TenPerSecond
```

Definition: Monadic streams and clocks (simplified)

```haskell
type MStream m a = MStreamF m () a
class Clock c where
  ticks :: MStream m Time
```

- *Time* is some type representing a time domain, say *UTCTime* (real time) or *Double* (simulation).
- *ticks* is a sideeffectful stream of time stamps. Repeat:
  1. Wait until the tick is due
  2. Return the current time stamp



Constant sample rate (“pull”)

```
runEasyExample = reactimateR $
  constantly 3 ↣ liftSF print @@ FivePerSecond
```

reactimateR	The main loop
@@	Specify clock
liftSF	Lifts a “sideeffectful function” to a (sideeffectful) signal function.

Events are clocks, too!

```
pushExample = reactimateR $
  tickInfo ↣ arr length ↣ liftSF print @@ KeyboardClock
```

Some different kinds of clocks:

- Constant rate
- “As fast as possible”
- Event-based (user-input, web server etc.)

Putting the clock in the type signature

```
easyExample    :: SF FivePerSecond IO () ()
easyExample    = constantly 3 ↣ liftSF print
runEasyExample = reactimateR $
  easyExample @@ FivePerSecond
```

SF	clocked **S**ignal **F**unction
FivePerSecond	Type specifying the clock (think: sampling speed) at which the *SF* has to run

Lots of clocks are singleton types ⟹ choose the same name for the clock type and the single inhabitant.

Type level clocks as safety measure

This won't compile

```
lifeCriticalPart :: SF CriticalRealTime m a b
lifeCriticalPart = ...
main = reactimateR $ lifeCriticalPart @@ SomeSlowClock
```

Neither will this

```
slowPart :: SF SomeSlowClock m b c
slowPart = ...
invalidComposition = slowPart ↣ lifeCriticalPart
```

... but if *SF*s with different clocks can't be composed with $\rightarrowtail$, how will they communicate?

- There is no single, general solution. Need framework for resampling!
- Separate two aspects:
  - Resampling of data streams (bounded FIFO, interpolation, . . .)
  - Scheduling of clocks (static schedule, synchronous, asynchronous, concurrently, . . .)

Actual code from the library

```
data ResBuffer m a b =
       ResBuffer { put :: a → m (   ResBuffer m a b)
                 , get ::       m (b, ResBuffer m a b)
                 }
```

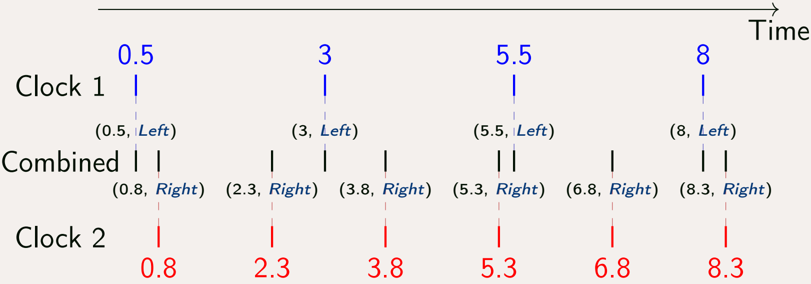
Some example implementations from the library

- `freshestValue :: a → ResBuffer m a a`
- `fifo :: a → ResBuffer m a a`
- `collect :: ResBuffer m a [a]`
- `mealy :: Mealy s (Maybe a) b → s → ResBuffer m a b`
- Could implement interpolation easily

Use bounded versions of *fifo* and *collect* to avoid space leaks!

Actual code from the library, simplified

$$\textbf{type}\ \mathit{Schedule\ c1\ c2\ m} = \mathit{c1} \to \mathit{c2} \to \mathit{MStream\ m}\ (\mathit{Time}, \mathit{Either}\ (\mathit{Tick\ c1})\ (\mathit{Tick\ c2}))$$



Combined clock ∼ Lustre-like base clock!

Some example implementations from the library

- $concurrently :: Schedule\ c1\ c2\ IO$
  - Launch two threads, run one clock in each thread.
  - Side effect in IO

  $\Longrightarrow$ nondeterministic (in naive implementation)
- $ratioHalf :: Monad\ m \Rightarrow Schedule\ c\ (HalfFrequency\ c\ m)\ m$
  - After two ticks of c, do one tick of $HalfFrequency\ c$ immediately.
  - Arbitrary monad: no side effects necessary

  $\Longrightarrow$ deterministic

```
resamplingTest = reactimateR $
   timeSinceStart ↣ trace "Putting " @@ Second
   ≻— freshestValue 0 −@− ratioHalf ⟶
   trace "Getting " ↣ constantly () @@ (HalfFrequency Second)
```

```
resamplingTest2 = reactimateR $
   tickInfo @@ KeyboardClock
   ≻— fifo "(empty)" −@− concurrently ⟶
   liftSF print @@ Second
```

Main loop

```
reactimateR $
  count ↣ arr even ↣ trace "Gravity: " @@ mouseDownClock
  ⤚ freshestValue True –@– concurrently ⟶
  physics ↣
  model @@ FourtyFPS
```

Add additional resampling easily!

QuickCheck

Formulate hypothesis to test

$$\begin{aligned}&\mathit{noBulletThroughPaper} :: \mathit{SF}\ \mathit{TestClock}\ \mathit{Identity}\ \mathit{Bool}\ \mathit{Bool}\\&\mathit{noBulletThroughPaper} = \mathit{physics} \rightarrowtail \mathit{arr}\ (\lambda(v, x) \rightarrow x \leqslant 20)\end{aligned}$$

Let QuickCheck automatically generate test data

$$\begin{aligned}&\mathit{testReflect} = \mathit{quickCheckWith}\\&\quad (\mathit{Args}\ \mathit{Nothing}\ 300\ 300\ 300\ \mathit{True})\\&\quad \mathit{noBulletThroughPaper}\end{aligned}$$

Thank you for your attention!