Functional reactive programming and clock calculus in Haskell

Manuel Bärenz (Bamberg) and Ivan Perez (Nottingham)

December 4, 2015
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..?
- Garbage collector
- Compile to microprocessors & embedded systems?
  (Executable size \(\sim\) 1 MB)
Definition: Causal stream function

```
data StreamF a b = StreamF (a -> (b, StreamF a b))
```

Constructor: Input: Output: New state

Example

```
streamFunction :: StreamF a b
```

```
a -> streamFunction -> b
```
Composability

Use composition $\rightarrow$ for data flow:

$\left(\rightarrow\right) :: \text{StreamF } a \ b \rightarrow \text{StreamF } b \ c \rightarrow \text{StreamF } a \ c$

Example

$\text{streamFunction} :: \text{StreamF } () \ \text{Int}$
$\text{streamFunction} = \text{constantly } 3 \rightarrow \text{sum} \rightarrow \text{arr } (*2)$

$(, , , ) \ldots \rightarrow \text{constantly } 3 \rightarrow \text{sum} \rightarrow \text{arr } (*2) \rightarrow 6, 12, 18, \ldots$
Modularity

Combine several stream functions parallely:

\((***) :: StreamF \ a \ b \rightarrow StreamF \ c \ d \rightarrow StreamF \ (a, c) \ (c, d)\)

Example

\texttt{streamFunction} \ **\ * another\texttt{StreamFunction}
Signals as streams with a time input

\[
\text{type } \text{SignalFunction } a \ b = \text{StreamF} \ (\text{Double}, a) \ b
\]

Composition
Main loop of Yampa (simplified pseudocode)

```plaintext
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 → m ()  
A side effect depending on b, with result void

m controls the strength of the side effect:

<table>
<thead>
<tr>
<th>Monad</th>
<th>Side effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>Any possible side effect (“Input/Output”)</td>
</tr>
<tr>
<td>Identity</td>
<td>No side effect</td>
</tr>
<tr>
<td>State</td>
<td>Global state variable</td>
</tr>
<tr>
<td>…</td>
<td>Your favourite embedded DSL</td>
</tr>
</tbody>
</table>

...exceptions, debugging, logging,...
Actual definition: Monadic stream function

```haskell
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:

```haskell
reactiveProgram :: MSignalF m () ()
reactiveProgram = sensor ↦ signalFunction ↦ actuator
```
Parallel composition

\[
\text{bigProgram} = (\text{sensor1} \xrightarrow{a_1} \text{sf1} \xrightarrow{b_1} \text{actuator1}) \\
\quad \ast \ast \ast \\
(\text{sensor2} \xrightarrow{a_2} \text{sf2} \xrightarrow{b_2} \text{actuator2})
\]
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)

\[
\begin{align*}
type \ MStream \ m \ a & = MStreamF \ m \ () \ a \\
class \ Clock \ c \ where \\
& \text{ticks :: } MStream \ m \ Time
\end{align*}
\]

- \textit{Time} is some type representing a time domain, say \texttt{UTCTime} (real time) or \texttt{Double} (simulation).
- \texttt{ticks} is a side-effectful 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

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

\(SF\) 
\(\text{clocked Signal Function}\)

\(FivePerSecond\) 
\(\text{Type specifying the clock (think: sampling speed) at which the } SF \text{ has to run}\)

Lots of clocks are singleton types \(\Rightarrow\) choose the same name for the clock type and the single inhabitant.
Type level clocks as safety measure

This won't compile

\[
\text{lifeCriticalPart} :: \ SF \ \text{CriticalRealTime} \ m \ a \ b \\
\text{lifeCriticalPart} = \ldots \\
\text{main} = \text{reactimateR} \ \$ \ \text{lifeCriticalPart} \ \odot \odot \ \text{SomeSlowClock}
\]

Neither will this

\[
\text{slowPart} :: \ SF \ \text{SomeSlowClock} \ m \ b \ c \\
\text{slowPart} = \ldots \\
\text{invalidComposition} = \text{slowPart} \ \rightarrow \ \text{lifeCriticalPart}
\]
but if $SF$s with different clocks can’t be composed with $↣$, 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

```haskell
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!
Yampa in a nutshell
Side effects and explicit clock calculus
Multi-rate systems and resampling

Separation of aspects
Data: Resampling buffers
Clocks: Scheduling
Examples
In-depth example: A small game

Actual code from the library, simplified

```haskell
type Schedule c1 c2 m = c1 → c2 → MStream m (Time, Either (Tick c1) (Tick c2))
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Clock 1</th>
<th>Combined</th>
<th>Clock 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>3</td>
<td>2.3</td>
<td>2.3</td>
<td>2.3</td>
</tr>
<tr>
<td>5.5</td>
<td>3.8</td>
<td>3.8</td>
<td>3.8</td>
</tr>
<tr>
<td>8</td>
<td>5.3</td>
<td>5.3</td>
<td>5.3</td>
</tr>
<tr>
<td></td>
<td>6.8</td>
<td>6.8</td>
<td>6.8</td>
</tr>
<tr>
<td></td>
<td>8.3</td>
<td>8.3</td>
<td>8.3</td>
</tr>
</tbody>
</table>

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*
  - \( \Rightarrow \) nondeterministic (in naive implementation)

- **ratioHalf :: Monad m ⇒ Schedule c (HalfFrequency c m) m**
  - After two ticks of *c*, do one tick of *HalfFrequency c* immediately.
  - Arbitrary monad: no side effects necessary
  - \( \Rightarrow \) 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

```haskell
reactimateR $
  \text{count} \mapsto \text{arr even} \mapsto \text{trace "Gravity: " @@ mouseDownClock}
  \mapsto \text{freshestValue True } \mathord{-@-} \text{ concurrently } \mapsto
  \text{physics} \mapsto
  \text{model } \mathord{@@} \text{ FourtyFPS}
```

Add additional resampling easily!
QuickCheck

Formulate hypothesis to test

\[\text{noBulletThroughPaper} :: \; SF \; \text{TestClock} \; \text{Identity} \; \text{Bool} \; \text{Bool}\]

\[\text{noBulletThroughPaper} = \text{physics} \leadsto \text{arr} (\lambda(v, x) \rightarrow x \leq 20)\]

Let QuickCheck automatically generate test data

\[\text{testReflect} = \text{quickCheckWith}\]

\[(\text{Args Nothing} \; 300 \; 300 \; 300 \; \text{True})\]

\[\text{noBulletThroughPaper}\]
Thank you for your attention!