A basic implementation: SF (2)

```haskell
type DTime = Double

data SF a b =
  SF { sfTF :: DTime -> a -> Transition a b }

type Transition a b = (SF a b, b)

The continuation encapsulates any internal state of the signal function. The type synonym DTime is the type used for the time deltas, > 0.
```

A basic implementation: arr

```haskell
arr :: (a -> b) -> SF a b
arr f = sf
  where
    sf = SF { sfTF = \_ a -> (sf, f a) }

Note: It is obvious that arr constructs a stateless signal function since the returned continuation is exactly the signal function being defined, i.e. it never changes.
```

A basic impl.: reactimate (1)

The function reactimate is responsible for animating a signal function:
- Loops over the sampling points.
- At each sampling point:
  - reads input sample and time from the external environment (typically I/O action)
  - feeds sample and time passed since previous sampling into the signal function's transition function
  - writes the resulting output sample to the environment (typically I/O action).

A basic impl.: reactimate (2)

- The loop then repeats, but uses the continuation returned from the transition function on the next iteration, thus ensuring any internal state is maintained.

A basic implementation: >>>

For >>>, we have to combine their continuations into updated continuation for the composed arrow:

```haskell
(>>>) :: SF a b -> SF b c -> SF a c
(>>>) = \ (sf1, tf1) -> \ (sf2, tf2) ->
  SF { sfTF = \ dt a -> (sf1' >>> sf2', c) }
  where
    (sf1', b) = tf1 dt a
    (sf2', c) = tf2 dt b

Note how same time delta is fed to both subordinate signal functions, thus ensuring synchrony.
```

A basic impl.: How to get started? (1)

What should the very first time delta be?
- Could use 0, but that would violate the assumption of positive time deltas (time always progressing), and is a bit of a hack.
- Instead:
  - Initial SF representation makes a first transition given just an input sample.
  - Makes that transition into a representation that expects time deltas from then on.
A basic impl.: How to get started? (2)

data SF a b =
    SF {sfTF :: a -> Transition a b}
data SF' a b =
    SF' {sfTF' :: DTime -> a -> Transition a b}
type Transition a b = (SF' a b, b)SF'

is internal, can be thought of as representing a “running” signal function.

No optimization possible?

The type system does not get in the way of all optimizations. For example, for:

constant :: b -> SF a b
constant b = arr (const b)

the following laws can readily be exploited:

sf >>> constant c = constant c
constant c >>> arr f = constant (f c)

But to do better, we need GADTs.

Optimizing >>>: First Attempt (1)

The arrow identity law:

arr id >>> a = a = a >>> arr id

How can this be exploited?

1. Introduce a constructor representing arr id
   data SF a b = ...
   SFId
2. Make SF abstract by hiding all its constructors.

Optimizing >>>: Second Attempt (1)

Instead of

data SF a b = ...

we define

data SF a b where
    ...
SFId :: SF a a
    ...

Other Ways?

There are other ways to implement this kind of optimisation (e.g. Hughes 2004). However:

• GADTs offer a completely straightforward solution
• absolutely no run-time overhead.

The latter is important for Yampa, since the signal function network constantly must be monitored for emerging optimization opportunities:

arr g >>> switch (...) (\_ -> arr f)

Generalized Algebraic Data Types

GADTs allow

• individual specification of return type of constructors
• the more precise type information to be taken into account during case analysis.

Laws Exploited for Optimizations

General arrow laws:

(arr g >>> switch (...) (\_ -> arr f) >>> arr id)

arr (g . f) = arr f >>> arr g

(arr id >>> f = f

f = f >>> arr id

Laws involving const (the first is Yampa-specific):

sf >>> arr (const k) = arr (const k)
arr (const k) >>> arr f = arr (const(f k))
data SF a b where
  SFArr ::
    (DTime -> a -> (SF a b, b)) -> FunDesc a b
  -> SF a b
SFcPAXA ::
  (DTime -> a -> (SF a d, d))
  -> FunDesc a bSF c FunDesc c d
de -> SF a d
  SF ::
    (DTime -> a -> (SF a b, b)) -> SF a b

**Implementation (2)**

data FunDesc a b where

  FDI :: FunDesc a a
  FDC :: b -> FunDesc a b
  FDG :: (a -> b) -> FunDesc a b

**Optimizing Event Processing (1)**

data FunDesc a b where

  FDE :: (Event a -> b) -> FunDesc (Event a) b

**Optimizing Event Processing (2)**

fdComp (FDG f1) (FDE f2 f2ne) = FDG f
  where
    f a =  
      case f1 a of  
        NoEvent -> f2ne
        f1a -> f2 f1a

Events

Yampa models *discrete-time* signals by lifting the range of continuous-time signals:

```haskell
data Event a = NoEvent | Event a
Discrete-time signal = Signal(Event a).
```

Consider composition of pure event processing:

```haskell
f :: Event a -> Event b
f :: Event a -> Event b
arr f >>> arr g
```

Optimizing Stateful Event Processing

A general stateful event processor:

```haskell
ep :: (c -> a -> (c, b, b)) -> SF (Event a) b
```

Composes nicely with stateful and stateless event processors!

Introduce explicit representation:

```haskell
data SF a b where
  SFEP :: ...
  -> (c -> a -> (c, b, b)) -> SF (Event a) b
```

Cause for Concern

Code with GADT-based optimizations is getting large and complicated:

- Many more cases to consider.
- Larger size of signal function representation.

Example: Size of `>>>`:

- Completely unoptimized: 15 lines
- Some optimizations (current): 45 lines
- GADT-based optimizations: 240 lines

Is the result really a performance improvement?

Micro Benchmarks (1)

A number of Micro Benchmarks were carried out to verify that individual optimizations worked as intended:

- Yes, works as expected.
- No significant performance overhead.
- Particularly successful for optimizing event processing: additional stages can be added to event-processing pipelines with almost no overhead.
Micro Benchmarks (2)

Most important gains:
- Insensitive to bracketing.
- A number of “pre-composed” combinators no longer needed, thus simplifying the Yampa API (and implementation).
- Much better event processing.
But what about overall, system-wide performance impact? Does it make a difference???

The MEP4

Benchmark 1: Space Invaders

Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$T_U$ [s]</th>
<th>$T_S$ [s]</th>
<th>$T_G$ [s]</th>
<th>$T_S/T_U$</th>
<th>$T_G/T_S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space Inv.</td>
<td>0.95</td>
<td>0.86</td>
<td>0.88</td>
<td>0.91</td>
<td>1.02</td>
</tr>
<tr>
<td>MEP</td>
<td>19.39</td>
<td>10.31</td>
<td>9.36</td>
<td>0.53</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Benchmark 2: MIDI Event Processor

High-level model of a MIDI event processor programmed to perform typical duties:

The MEP4

Reading