A basic implementation: SF (1)

Each signal function is essentially represented by a *transition function*. Arguments:
- Time passed since the previous time step.
- The current input value.

Returns:
- A (possibly) updated representation of the signal function, the *continuation*.
- The current value of the output signal.

A basic implementation: SF (2)

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 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: arr

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 implementation: >>>

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

>>> :: SF a b -> SF b c -> SF a c
(SF {sfTF = tf1}) >>> (SF {sfTF=tf2}) = SF {sfTF = tf}
where
    tf 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.

Optimizing >>>: First Attempt (1)

The arrow identity law:

\[
\text{arr id} \rightarrow \text{arr id}
\]

How can this be exploited?

1. Introduce a constructor representing the identity:

\[
\text{data SF a b} = \ldots
\]

2. Make SF abstract by hiding all its constructors.

A basic impl.: How to get started? (2)

data SF a b =
\[
\text{SF \{sfTF :: a -> Transition a b\}}
\]

data SF’ a b =
\[
\text{SF’ \{sfTF’ :: DTime -> a -> Transition a b\}}
\]

type Transition a b = (SF’ a b, b)

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

Optimizing >>>: First Attempt (2)

3. Ensure SFId only gets used at intended type:

\[
\text{id} :: SF a a
\]

\[
\text{id} = \text{SFId}
\]

4. Define optimizing version of >>>:

\[
\text{(||)} :: SF a b \rightarrow SF b c \rightarrow SF a c
\]

\[
\ldots
\]
No optimization possible?

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

\[
\text{constant} :: b \rightarrow SF\ a\ b \\
\text{constant}\ b = \text{arr}\ (\text{const}\ b)
\]

the following laws can readily be exploited:

\[
sf >>= \text{constant}\ c = \text{constant}\ c \\
\text{constant}\ c >>= \text{arr}\ f = \text{constant}\ (f\ c)
\]

But to do better, we need GADTs.

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.

Optimizing \(\gg\gg\gg\): Second Attempt (1)

Instead of

\[
\text{data}\ SF\ a\ b = \ldots
\]

we define

\[
\text{data}\ SF\ a\ b\ \text{where} \\
\ldots \\
\text{SFId} :: SF\ a\ a \\
\ldots
\]

Optimizing \(\gg\gg\gg\): Second Attempt (2)

Define optimizing version of \(\gg\gg\gg\) exactly as before:

\[
(\gg\gg\gg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c \\
\ldots
\]
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:

\[
\text{arr } g \ggg \text{switch } (...) (\_ \rightarrow \text{arr } f) \\
\text{switch} \rightarrow \text{arr } g \ggg \text{arr } f = \text{arr } (f . g)
\]

Laws Exploited for Optimizations

General arrow laws:

\[
(f >>> g) >>> h = f >>> (g >>> h) \\
\text{arr } (g . f) = \text{arr } f >>> \text{arr } g \\
\text{arr } \text{id} >>> f = f \\
f = f >>> \text{arr } \text{id}
\]

Laws involving \text{const} (the first is Yampa-specific):

\[
sf >>> \text{arr } (\text{const } k) = \text{arr } (\text{const } k) \\
\text{arr } (\text{const } k)>>>\text{arr } f = \text{arr } (\text{const } f . k)
\]

Implementation (1)

\[
data \ SF \ a \ b \ where \ \\
\text{SFArr ::} \\
(DTime \rightarrow a \rightarrow (SF \ a \ b, b)) \rightarrow \text{FunDesc } a \ b \rightarrow \text{SF } a \ b \\
\text{SFCpAXA ::} \\
(DTime \rightarrow a \rightarrow (SF \ a \ d, d)) \rightarrow \text{FunDesc } a \ b \rightarrow \text{SF } b \ c \rightarrow \text{FunDesc } c \ d \rightarrow \text{SF } a \ d \\
\text{SF ::} \\
(DTime \rightarrow a \rightarrow (SF \ a \ b, b)) \rightarrow \text{SF } a \ b
\]

Implementation (2)

\[
data \text{ FunDesc } a \ b \ where \\
\text{FDI :: FunDesc } a \ a \\
\text{FDC :: } b \rightarrow \text{FunDesc } a \ b \\
\text{FDG :: } (a \rightarrow b) \rightarrow \text{FunDesc } a \ b \\
\text{Recovering the function from a FunDesc:} \\
\text{fdFun :: FunDesc } a \ b \rightarrow (a \rightarrow b) \\
\text{fdFun FDI } = \text{id} \\
\text{fdFun (FDC } b) = \text{const } k \\
\text{fdFun (FDG } f) = f
\]
Implementation (3)

\[
\text{fdComp} :: \text{FunDesc } a \to b \rightarrow \text{FunDesc } b \to c \\
\rightarrow \text{FunDesc } a \to c \\
\text{fdComp } \text{FDI } fd2 = fd2 \\
\text{fdComp } fd1 \text{FDI } = fd1 \\
\text{fdComp } (\text{FDC } b) \text{fd2} = \\
\quad \text{FDC } ((\text{fdFun } fd2) \ b) \\
\text{fdComp } _- (\text{FDC } c) = \text{FDC } c \\
\text{fdComp } (\text{FDG } f1) \text{fd2} = \\
\quad \text{FDG } (\text{fdFun } fd2 . f1)
\]

Events

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

\[
\text{data Event } a = \text{NoEvent } | \text{Event } a
\]

Discrete-time signal = Signal (Event a).

Consider composition of pure event processing:

\[
f :: \text{Event } a \to \text{Event } b \\
g :: \text{Event } b \to \text{Event } c
\]

arr f >>> arr g

Optimizing Event Processing (1)

Additional function descriptor:

\[
\text{data FunDesc } a \ b \text{ where } \\
\quad \ldots \\
\quad \quad \text{FDE } :: \text{(Event } a \to b) \to b \\
\quad \quad \quad \to \text{FunDesc } (\text{Event } a) \ b
\]

Extend the composition function:

\[
\text{fdComp } (\text{FDE } f1 \ f1ne) \text{fd2} = \\
\quad \text{FDE } (f2 . f1) \ (f2 \ f1ne) \\
\quad \text{where} \\
\quad f2 = \text{fdFun } fd2
\]

Optimizing Event Processing (2)

Extend the composition function:

\[
\text{fdComp } (\text{FDG } f1) \ (\text{FDE } f2 \ f2ne) = \text{FDG } f \\
\text{where} \\
\quad f \ a = \\
\quad \quad \text{case } f1 \ a \ of \\
\quad \quad \quad \text{NoEvent } \rightarrow f2ne \\
\quad \quad \quad f1a \quad \rightarrow f2 \ f1a
\]
Optimizing Stateful Event Processing

A general stateful event processor:
\[
ep :: (c -> a -> (c,b,b)) -> c -> b
\rightarrow SF (Event a) b
\]
Composes nicely with stateful and stateless event processors!
Introduce explicit representation:
\[
data SF a b where
\]
\[
...SFEP :: ...
\rightarrow (c -> a -> (c, b, b)) -> c -> b
\rightarrow SF (Event a) b
\]

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??*
**Benchmark 1: Space Invaders**

![Space Invaders](image1)

**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:

![MIDI Event Processor](image2)

**The MEP4**

![MEP4](image3)