Outline

• The basic implementation approach
• Optimization
• Aggressive optimization using GADTs
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: $\text{SF}$ (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 $\text{DTime}$ is the type used for the time deltas, $> 0$.
```
A basic impl.: reactimate (1)

The function reactimate is responsible for animating a signal function:
A basic impl.: reactimate (1)

The function reactimate is responsible for animating a signal function:

- Loops over the sampling points.
A basic impl.: reactimate (1)

The function *reactimate* is responsible for animating a signal function:

- Loops over the sampling points.
- At each sampling point:
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)
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
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: \texttt{arr}

\begin{verbatim}
arr :: (a -> b) -> SF a b
arr f = sf
  where
    sf = SF {sfTF = \_ a -> (sf, f a)}
\end{verbatim}

Note: It is obvious that \texttt{arr} constructs a \textbf{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:

\[
(\triangleright\triangleright\triangleright) :: SF a b \to SF b c \to SF a c
\]

\[
(SF \{sfTF = tf1\}) \triangleright\triangleright\triangleright (SF \{sfTF=t\}) = SF \{sfTF = tf\}
\]

where

\[
tf dt a = (sf1' \triangleright\triangleright\triangleright sf2', c)
\]

where

\[
(sf1', b) = tf1 dt a
\]
\[
(sf2', c) = tf2 dt b
\]

Note how \textit{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?
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.
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.
The arrow identity law:

\[
\text{arr id } \gggg a = a = a \gggg \text{arr id}
\]
Optimizing >>>: First Attempt (1)

The arrow identity law:

\[ arr \text{ id} \triangleright	riangleright\triangleright a = a = a \triangleright	riangleright\triangleright arr \text{ id} \]

How can this be exploited?
Optmimizing >>>>: First Attempt (1)

The arrow identity law:

\[ \text{arr id} \gggg a = a = a \gggg \text{arr id} \]

How can this be exploited?

1. Introduce a constructor representing \textit{arr id}

```haskell
data SF a b = ...
  SFId
  ...
```
Optmimizing >>>>: First Attempt (1)

The arrow identity law:

\[ \text{arr id} \mapsto \text{id} = \text{id} = \text{id} \mapsto \text{arr id} \]

How can this be exploited?

1. Introduce a constructor representing \( \text{arr id} \)
   
   ```
   data SF a b = ...
   │ SFId
   │ ...
   ```

2. Make \( SF \) abstract by hiding all its constructors.
3. Ensure $\text{SFId}$ only gets used at intended type:

```haskell
identity :: SF a a
identity = SFId
```
Optimizing >>>: First Attempt (2)

3. Ensure $\text{SF}_{\text{Id}}$ only gets used at intended type:

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

   \[
   \text{identity} = \text{SF}_{\text{Id}}
   \]

4. Define optimizing version of >>>:

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

   \[
   \ldots
   \]

   \[
   \text{SF}_{\text{Id}} \text{>>> sf} = \text{sf}
   \]

   \[
   \ldots
   \]
3. Ensure $\text{SFId}$ only gets used at intended type:

\[
\text{identity} :: \text{SF} \ a \ a \\
\text{identity} = \text{SFId}
\]

4. Define optimizing version of $\ggg$:

\[
(\ggg) :: \text{SF} \ a \ b \to \text{SF} \ b \ c \to \text{SF} \ a \ c
\]

\[
\ldots
\]

\[
\text{SFId} \ggg \sf = \sf
\]

\[
\ldots
\]

\[
:: \text{SF} \ b \ c \neq \text{SF} \ a \ c
\]
No optimization possible?

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

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

the following laws can readily be exploited:

```plaintext
sf >>> constant c = constant c
constant c >>> arr f = 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.
Instead of

```haskell
data SF a b = ...
  SFId
  ...
```
Instead of

```haskell
data SF a b = ...
    SFId
    ...
    :: SF a b
```
Instead of

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

we define

\[
\text{data SF} \ a \ b \ \text{where}
\]

\[
\ldots
\]

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

\[
\ldots
\]
Define optimizing version of >>> exactly as before:

\[
(\gggg) :: SF \ a\ b \to SF\ b\ c \to SF\ a\ c
\]

\ldots
Define optimizing version of \( \gggg \) exactly as before:

\[
\gggg : \ SF \ a \ b \to \ SF \ b \ c \to \ SF \ a \ c
\]

\[
\ldots
\]

\[
\text{SFId} \gggg \ sf = sf
\]

\[
\ldots
\]
Define optimizing version of \( \gggg \) \textit{exactly} as before:

\[
(\gggg) :: SF \ a \ b \rightarrow SF \ b \ c \rightarrow SF \ a \ c
\]

\[
\text{SFId} \quad \gggg \quad sf = sf
\]

\[
:: \ SF \ a \ a
\]
Define optimizing version of \( \ggg \) exactly as before:

\[
(\ggg) :: SF\ a\ b \to SF\ b\ c \to SF\ a\ c
\]

\[
\text{SFId} \ggg \text{sf} = \text{sf}
\]

\[
:: SF\ a\ a
:: SF\ a\ c
\]
Other Ways?

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

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

- GADTs offer a completely straightforward solution
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.
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:

\[
\text{arr } g \gggg \text{switch} \ (\ldots) \ (\_ \to \text{arr } f) \\
\text{switch} \\
\implies \text{arr } g \gggg \text{arr } f = \text{arr } (f \ . \ g)
\]
Laws Exploited for Optimizations

General arrow laws:

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

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

\[
sf >>> \text{arr } (\text{const } k) = \text{arr } (\text{const } k)
\]
\[
\text{arr } (\text{const } k) >>> \text{arr } f = \text{arr } (\text{const } (f \cdot k))
\]
Laws Exploited for Optimizations

General arrow laws:

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

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

\[sf >>> \text{arr } (\text{const } k) = \text{arr } (\text{const } k)\]
\[
\text{arr } (\text{const } k) >>> \text{arr } f = \text{arr } (\text{const } (f \cdot k))\]
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 b -> SF b c -> FunDesc c d
  -> SF a d

SF ::
  (DTime -> a -> (SF a b, b))
  -> SF a b
data FunDesc a b where
  FDI :: FunDesc a a
  FDC :: b -> FunDesc a b
  FDG :: (a -> b) -> FunDesc a b
data FunDesc a b where

  FDI :: FunDesc a a

  FDC :: b \rightarrow FunDesc a b

  FDG :: (a \rightarrow b) \rightarrow FunDesc a b
data FunDesc a b where

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

Recovering the function from a FunDesc:

  fdFun :: FunDesc a b -> (a -> b)
  fdFun FDI = id
  fdFun (FDC b) = const b
  fdFun (FDG f) = f
Implementation (2)

data FunDesc a b where

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

Recovering the function from a FunDesc:

fdFun :: FunDesc a b -> (a -> b)

fdFun FDI = id
fdFun (FDC b) = const b
fdFun (FDG f) = f
Implementation (3)

```
fdComp :: FunDesc a b -> FunDesc b c -> FunDesc a c
fdComp FDI fd2 = fd2
fdComp fd1 FDI = fd1
fdComp (FDC b) fd2 = FDC ((fdFun fd2) b)
fdComp _ (FDC c) = FDC c
fdComp (FDG f1) fd2 = FDG (fdFun fd2 . f1)
```
Events

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

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

*Discrete-time signal* \(= \text{Signal(Event } a)\).
Yampa models \textit{discrete-time} signals by lifting the \textit{range} of continuous-time signals:

\begin{verbatim}
data Event a = NoEvent | Event a
\end{verbatim}

\textit{Discrete-time signal} \textit{=} Signal (Event α).

Consider composition of pure event processing:

\begin{verbatim}
f :: Event a \rightarrow Event b
g :: Event b \rightarrow Event c
\end{verbatim}

\texttt{arr f >>> arr g}
Additional function descriptor:

```haskell
data FunDesc a b where
  ...
  FDE :: (Event a -> b) -> b
        -> FunDesc (Event a) b
```

Optimizing Event Processing (1)
Optimizing Event Processing (1)

Additional function descriptor:

data FunDesc a b where

... 

FDE :: (Event a -> b) -> b -> FunDesc (Event a) b
Additional function descriptor:

```haskell
data FunDesc a b where
    ...
    FDE :: (Event a -> b) -> b -> FunDesc (Event a) b
```

Extend the composition function:

```haskell
fdComp (FDE f1 f1ne) fd2 = FDE (f2 . f1) (f2 f1ne)
where
    f2 = fdFun fd2
```
Extend the composition function:

\[ \text{fdComp } (\text{FDG } f_1) \ (\text{FDE } f_2 \ f_2\text{ne}) = \text{FDG } f \]

where

\[ f \ a = \]

\[ \text{case } f_1 \ a \ of \]

\[ \text{NoEvent} \rightarrow f_2\text{ne} \]
\[ f_1a \rightarrow f_2 \ f_1a \]
Extend the composition function:

\[
\text{fdComp } (\text{FDG } f_1) \ (\text{FDE } f_2 \ f_{2\text{ne}}) = \text{FDG } f \\
\text{where}
\]

\[
f \ a = \\
\text{case } f_1 \ a \ \text{of}
\]

NoEvent \ \rightarrow \ f_{2\text{ne}}

f_1a \ \rightarrow \ f_2 \ f_1a
\]
Optimizing Stateful Event Processing

A general stateful event processor:

\[
\text{ep} :: (c \to a \to (c, b, b)) \to c \to b \\
\to SF (Event a) b
\]
Optimizing Stateful Event Processing

A general stateful event processor:

\[ \text{ep} :: (c \to a \to (c, b, b)) \to c \to b \to \text{SF (Event } a \text{)} b \]

Composes nicely with stateful and stateless event processors!
Optimizing Stateful Event Processing

A general stateful event processor:

\[ \text{ep} :: (\text{c} \rightarrow \text{a} \rightarrow (\text{c}, \text{b}, \text{b})) \rightarrow \text{c} \rightarrow \text{b} \rightarrow \text{SF} \ (\text{Event a}) \ b \]

Composes nicely with stateful and stateless event processors!
Introduce explicit representation:

\[
\text{data SF a b where }
\]

\[ \ldots \]

\[ \text{SFEP} :: \ldots \]

\[ \rightarrow (\text{c} \rightarrow \text{a} \rightarrow (\text{c}, \text{b}, \text{b})) \rightarrow \text{c} \rightarrow \text{b} \rightarrow \text{SF} \ (\text{Event a}) \ b \]
Code with GADT-based optimizations is getting large and complicated:

- Many more cases to consider.
- Larger size of signal function representation.
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 >>>:
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
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
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
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?
A number of Micro Benchmarks were carried out to verify that individual optimizations worked as intended:
A number of Micro Benchmarks were carried out to verify that individual optimizations worked as intended:

- Yes, works as expected.
A number of Micro Benchmarks were carried out to verify that individual optimizations worked as intended:

- Yes, works as expected.
- No significant performance overhead.
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.
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
Benchmark 2: MIDI Event Processor

High-level model of a MIDI event processor programmed to perform typical duties:
The MEP4
## 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>
Reading