Implementing and Optimising Functional Reactive Programming

Big-O Meetup, 14 Dec. 2016

Henrik Nilsson

School of Computer Science

University of Nottingham, UK

Functional Reactive Programming (1)

Functional Reactive Programming (1)

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.

Functional Reactive Programming (1)

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran). (Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.)

Functional Reactive Programming (1)

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran). (Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.)
- FRP has evolved in a number of directions and into different concrete implementations.

Functional Reactive Programming (1)

- Key idea: Don’t program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak’s work on Functional Reactive Animation (Fran). (Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.)
- FRP has evolved in a number of directions and into different concrete implementations.
- This talk considers Yampa: an arrows-based FRP system embedded in Haskell.

Functional Reactive Programming (2)

- Yampa: pure and principled implementation in a pure setting.

Functional Reactive Programming (2)

- Yampa: pure and principled implementation in a pure setting.
- In particular: many algebraic laws hold.

Functional Reactive Programming (2)

- Yampa: pure and principled implementation in a pure setting.
- In particular: many algebraic laws hold.
- These guide the implementation and optimisations: a theme of this talk.

FRP Applications (1)

Some domains where FRP or FRP-inspired approaches have been used:

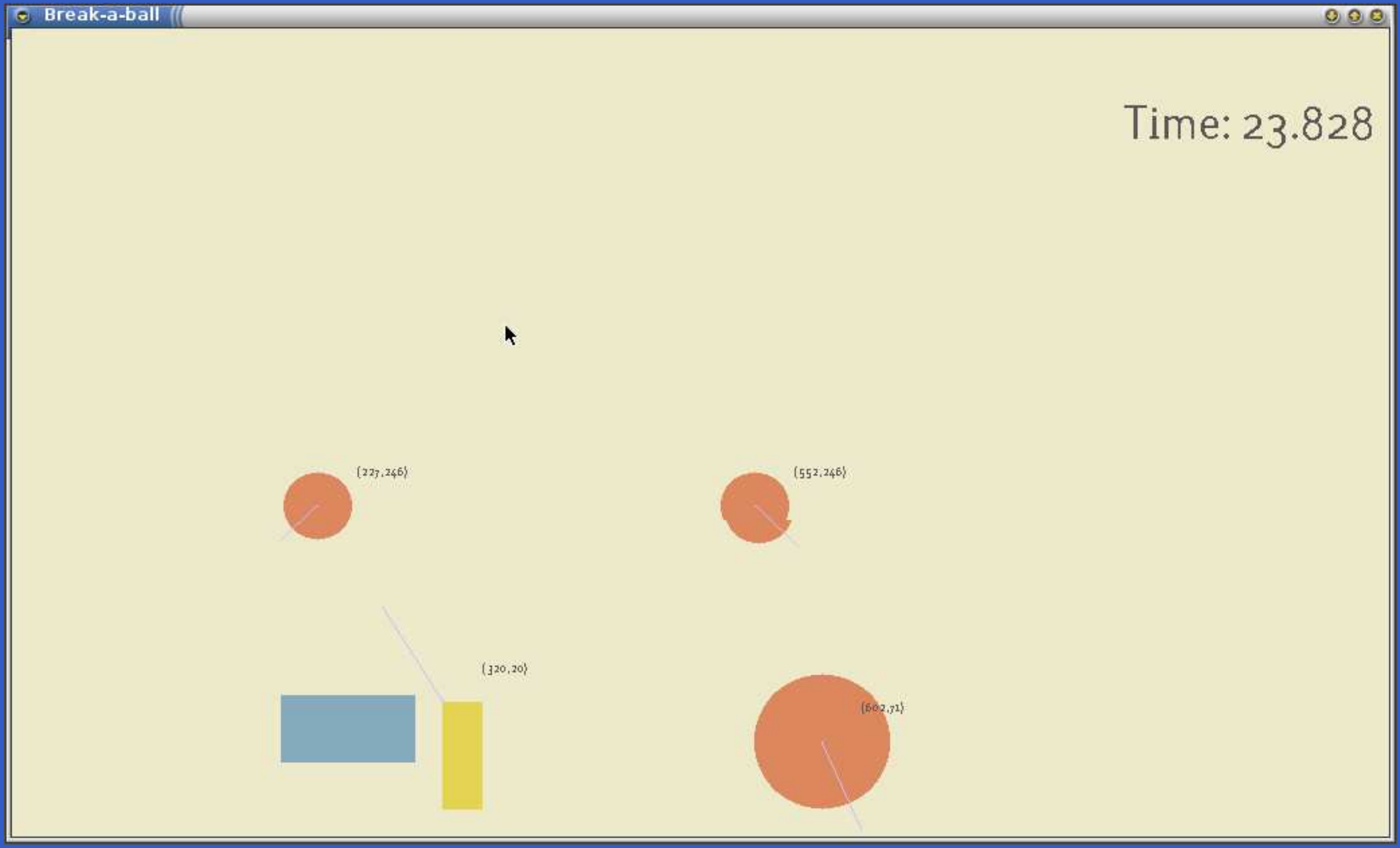
- Robotics
- Vision
- Sound synthesis
- GUIs
- Virtual Reality Environments
- Games
- Distributed Event-based Systems

FRP Applications (2)

Example: Breakout in Yampa (and SDL) on a tablet:

Take-home Game!

Or download one for free to your Android device!



Play Store: Pang-a-lambda (Keera Studios)

Arrows?

- A ***notion of computation***: function-like entities that may have effects.

Arrows?

- A ***notion of computation***: function-like entities that may have effects.
- Examples:

Arrows?

- A ***notion of computation***: function-like entities that may have effects.
- Examples:
  - Pure functions

Arrows?

- A ***notion of computation***: function-like entities that may have effects.
- Examples:
  - Pure functions
  - “Functions” with internal state

Arrows?

- A ***notion of computation***: function-like entities that may have effects.
- Examples:
  - Pure functions
  - “Functions” with internal state
  - Conditional probabilities

Arrows?

- A ***notion of computation***: function-like entities that may have effects.
- Examples:
  - Pure functions
  - “Functions” with internal state
  - Conditional probabilities
  - Any function of the form $a \rightarrow M\ b$ where M is a monad (the “Kleisli construction”).

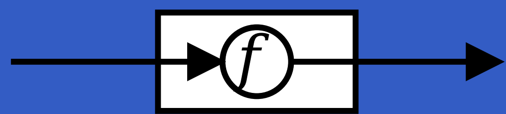
Arrows?

- A ***notion of computation***: function-like entities that may have effects.
- Examples:
  - Pure functions
  - “Functions” with internal state
  - Conditional probabilities
  - Any function of the form $a \to M\ b$ where M is a monad (the “Kleisli construction”).
- A number of ***algebraic laws*** must be satisfied: we will come back to those.

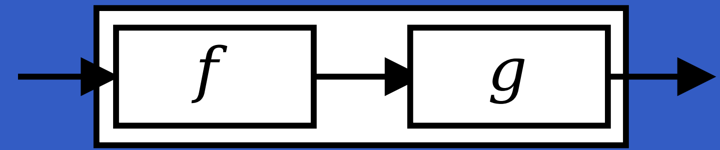
Arrows?

- A ***notion of computation***: function-like entities that may have effects.
- Examples:
  - Pure functions
  - “Functions” with internal state
  - Conditional probabilities
  - Any function of the form $a \to M\ b$ where M is a monad (the “Kleisli construction”).
- A number of ***algebraic laws*** must be satisfied: we will come back to those.
- Arrows due to Prof. John Hughes.

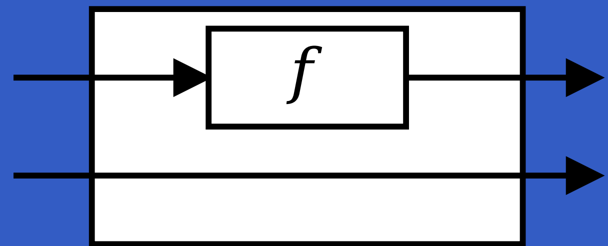
The Arrow framework (1)



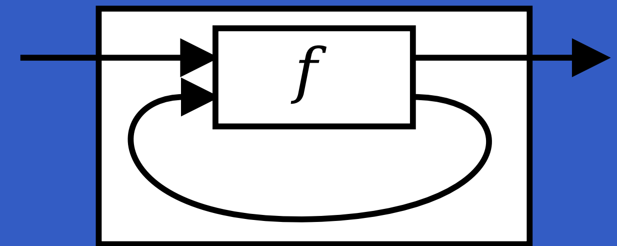
`arr f`



`f >>> g`



`first f`



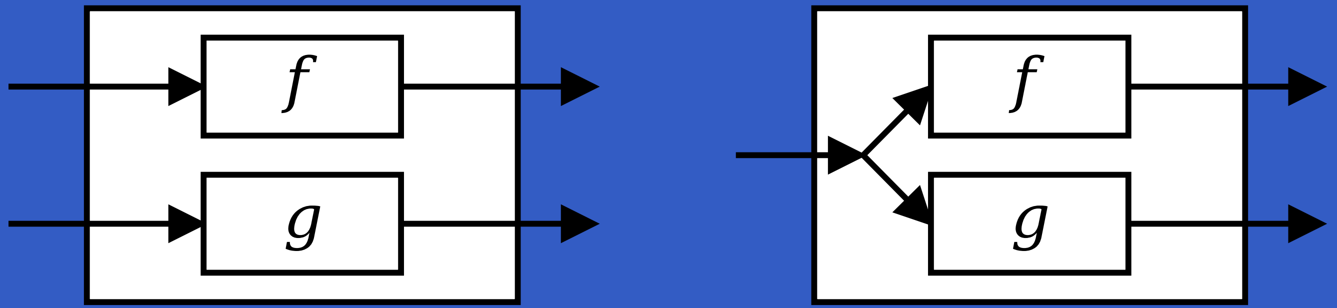
`loop f`

Types signatures for some arrow `F`:

```
arr    :: (a -> b) -> F a b
(>>>)  :: F a b -> F b c -> F a c
first  :: F a b -> F (a,c) (b,c)
loop   :: F (a, c) (b, c) -> F a b
```

The Arrow framework (2)

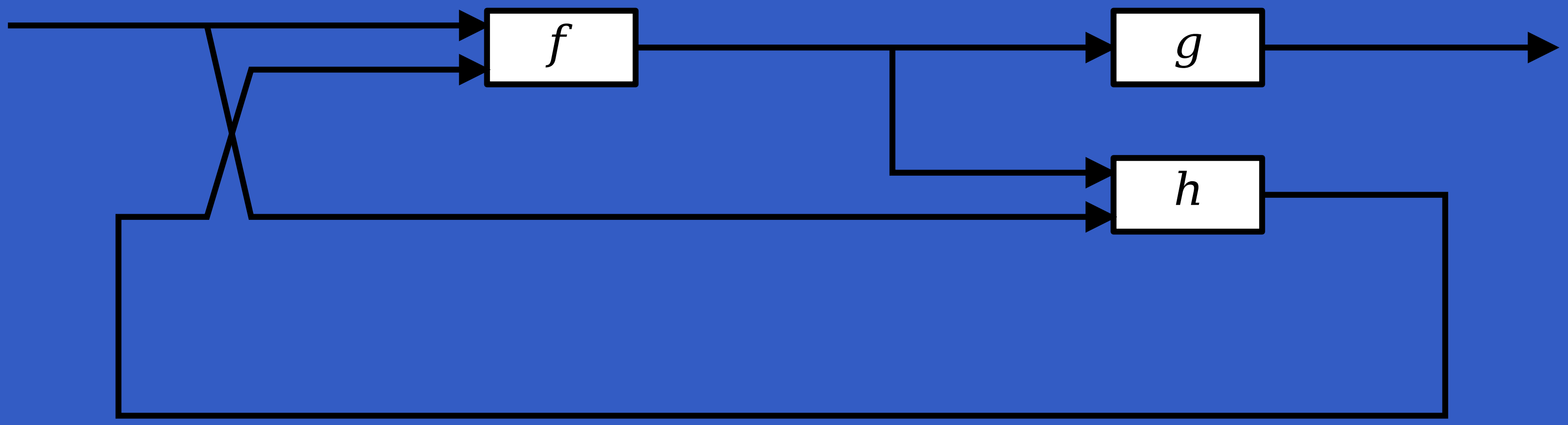
Some derived combinators:



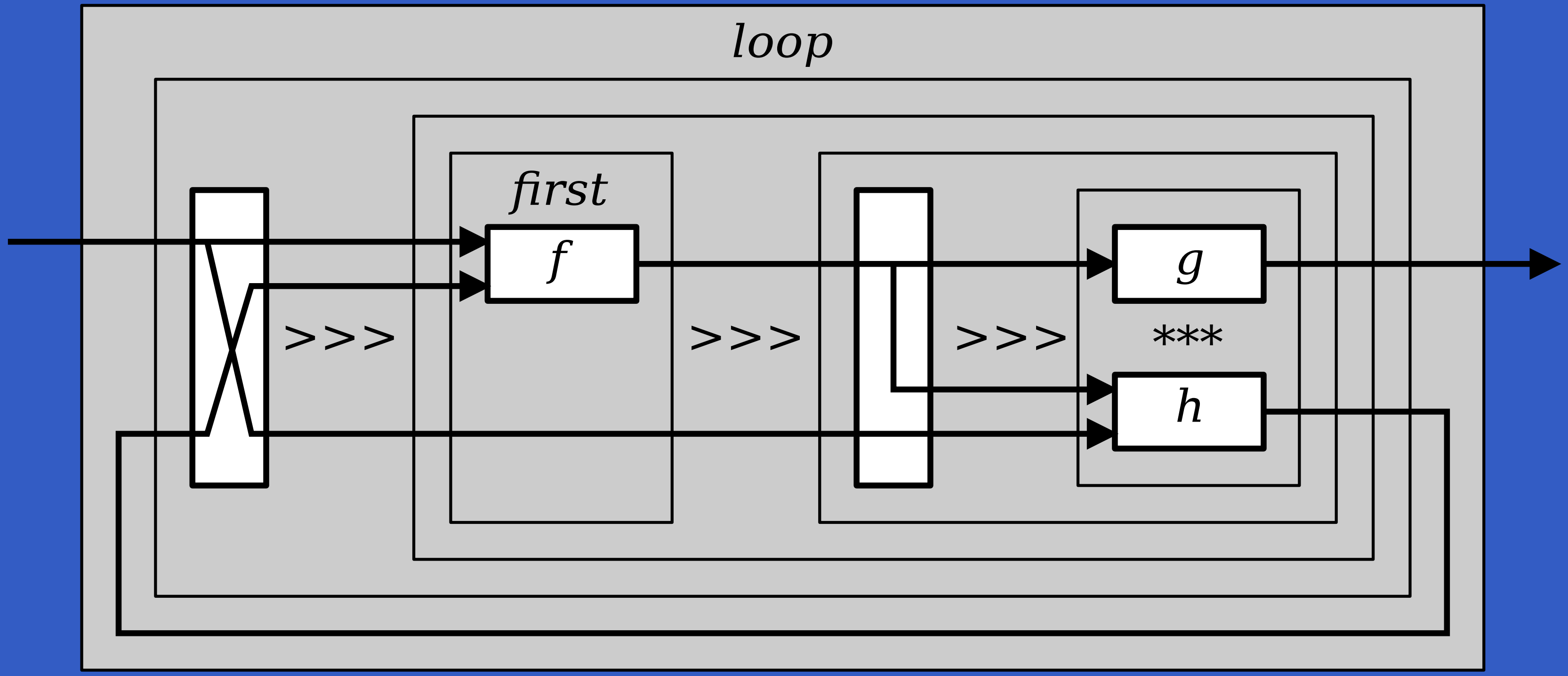
`f *** g` `f &&& g`

```
(***) :: F a b -> F c d -> F (a,c) (b,d)
(&&&) :: F a b -> F a c -> F a (b,c)
```

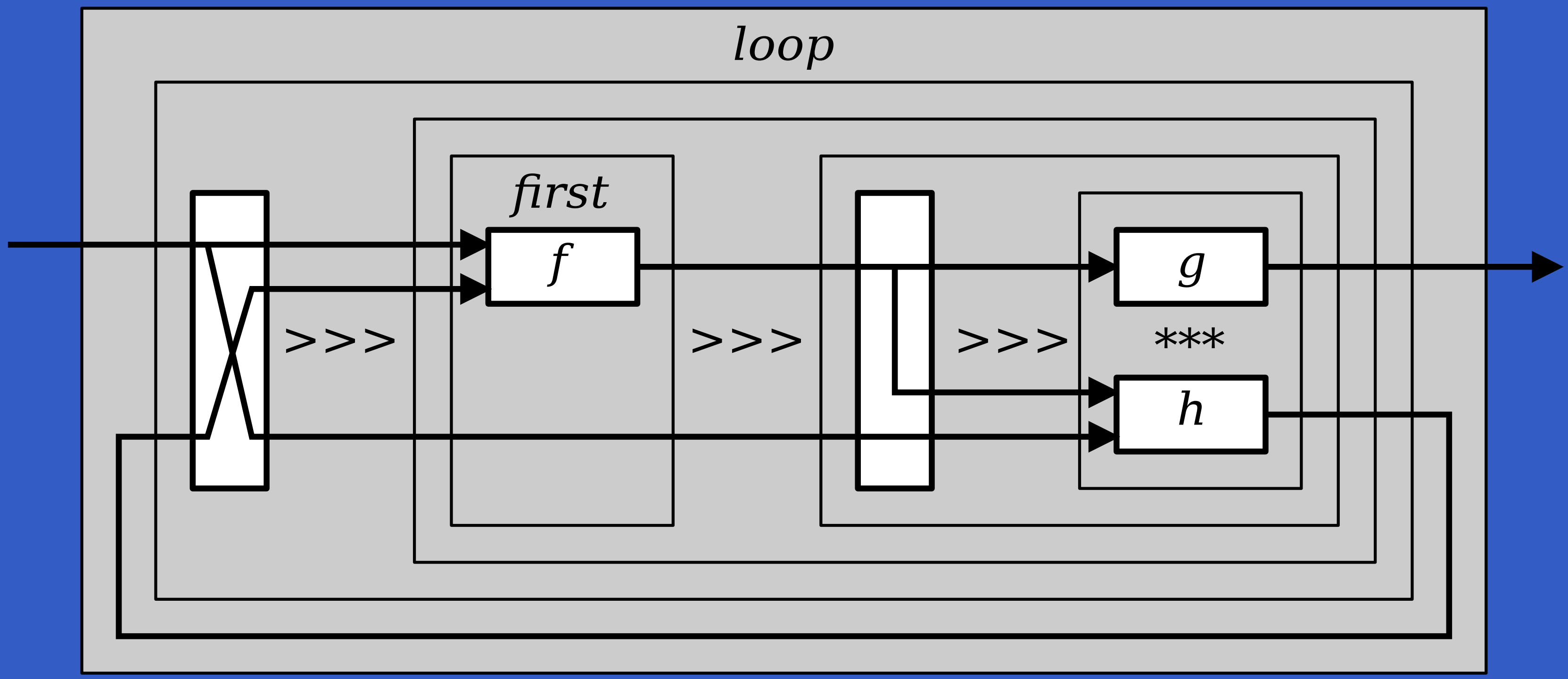
Constructing a network



Constructing a network

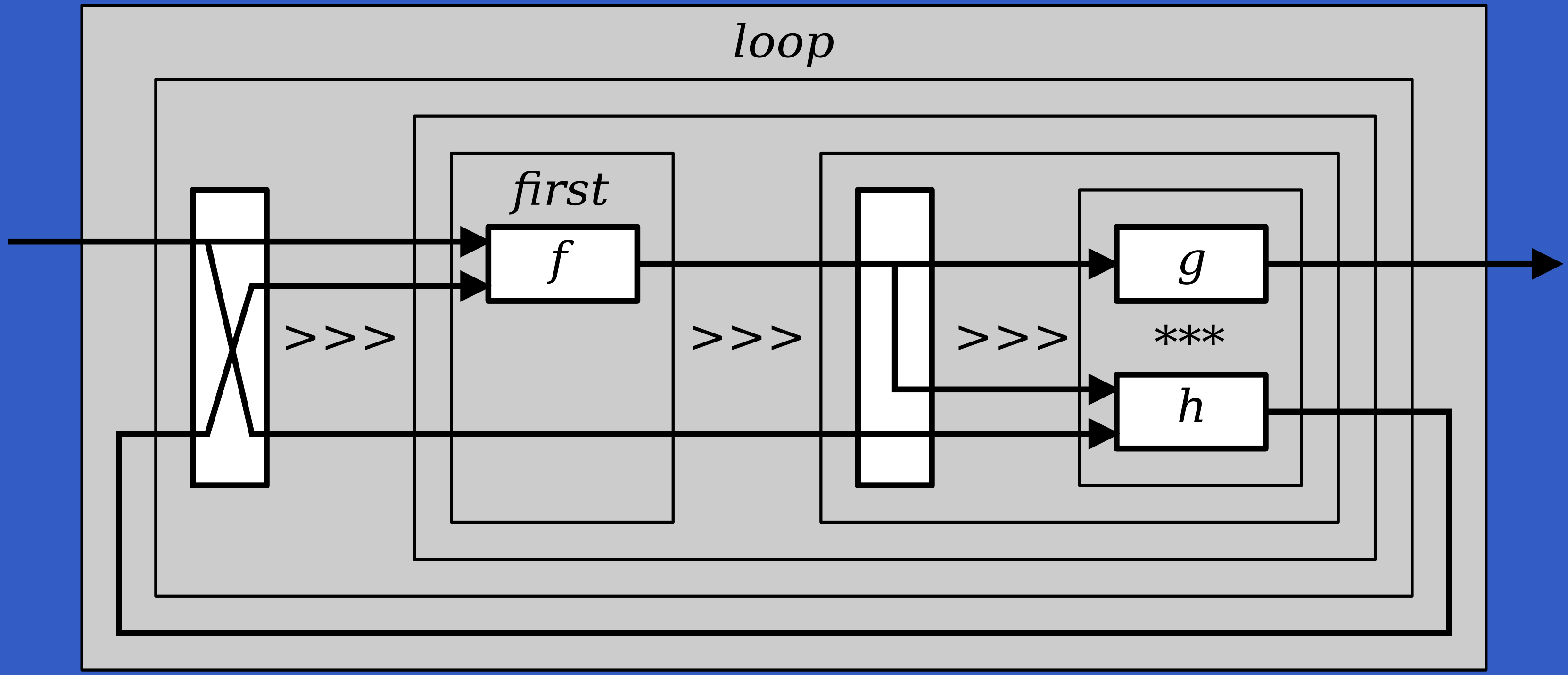


Constructing a network



Tedious way to program?

Constructing a network



Tedious way to program?

Yes, can be. But syntactic support can be provided.

Key FRP Features

Combines conceptual simplicity of the ***synchronous data flow*** approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time
- Dynamic system structure

Key FRP Features

Combines conceptual simplicity of the ***synchronous data flow*** approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time
- Dynamic system structure

(But not everything labelled “FRP” supports them all.)

Yampa (1)

Yampa (1)

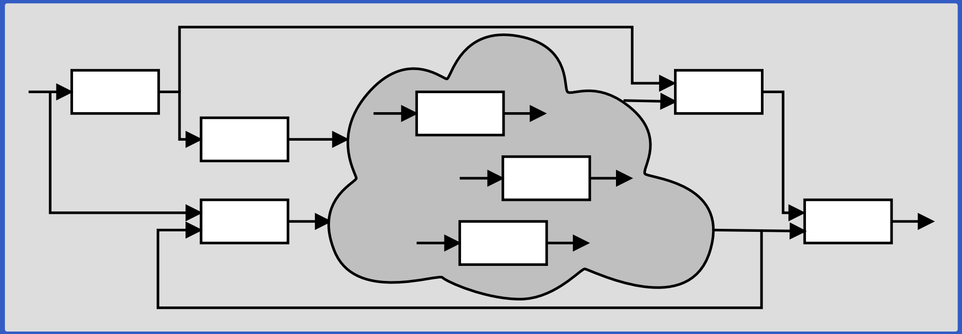
- FRP implemenattion embedded in Haskell

Yampa (1)

- FRP implemenattion embedded in Haskell
- Key concepts:
  - ***Signals***: time-varying values
  - ***Signal Functions***: functions on signals
  - ***Switching*** between signal functions

Yampa (1)

- FRP implemenattion embedded in Haskell
- Key concepts:
  - ***Signals***: time-varying values
  - ***Signal Functions***: functions on signals
  - ***Switching*** between signal functions
- Programming model:



Yampa (2)

- Signal functions are the primary notion: first-class entities.

Yampa (2)

- Signal functions are the primary notion: first-class entities.
- Signals are a secondary notion: only exist indirectly.

Yampa (2)

- Signal functions are the primary notion: first-class entities.
- Signals are a secondary notion: only exist indirectly.
- This is a key aspect allowing for a fundamentally simple, pure, implementation.

Yampa (2)

- Signal functions are the primary notion: first-class entities.
- Signals are a secondary notion: only exist indirectly.
- This is a key aspect allowing for a fundamentally simple, pure, implementation.
- Of course, FRP does not have to be implemented purely, and many FRP implementations are indeed not pure. But keeping it pure makes it easier to get correct. Good for reference if nothing else.

Yampa?

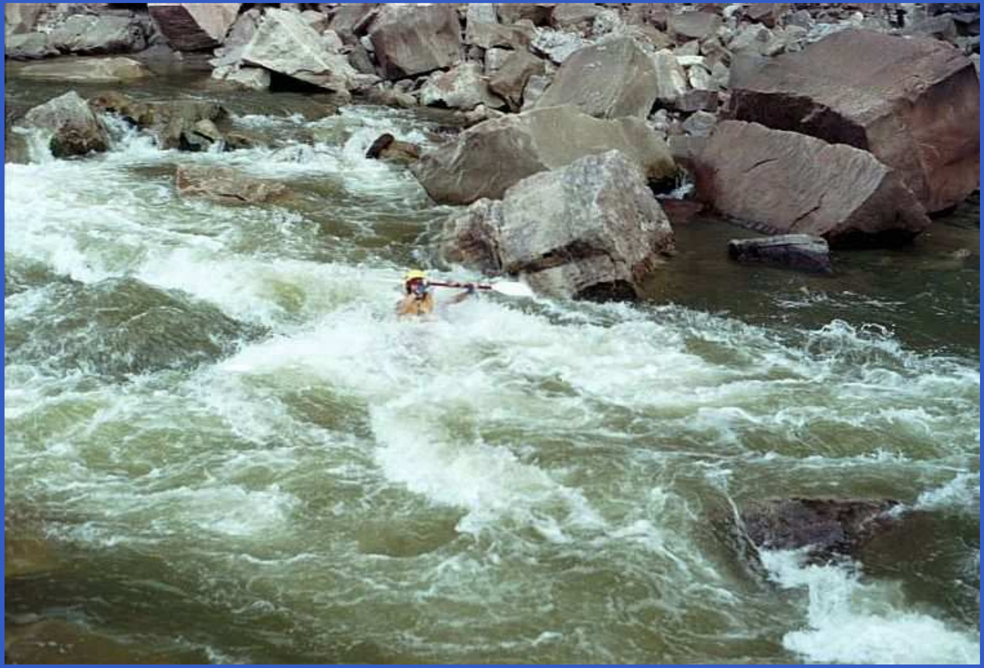
Yampa?

Yet ***A***nother ***M***ostly ***P***ointless ***A***cronym?

Yampa?

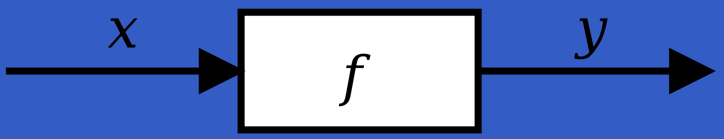
***Y**et **A**nother **M**ostly **P**ointless **A**cronym?*

Yampa is a river with long calmly flowing sections and abrupt whitewater transitions in between.

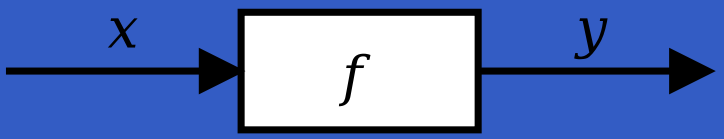


A good metaphor for hybrid systems!

Signal Functions

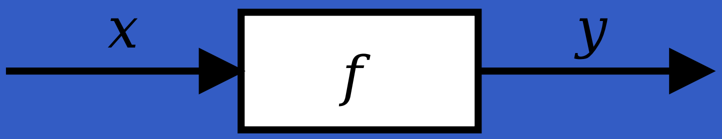


Signal Functions



Intuition:

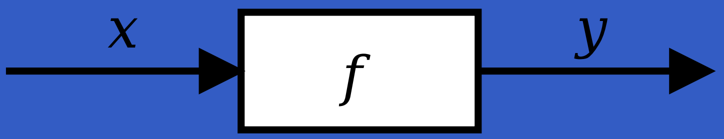
Signal Functions



Intuition:

$\texttt{Time} \approx \mathbb{R}$

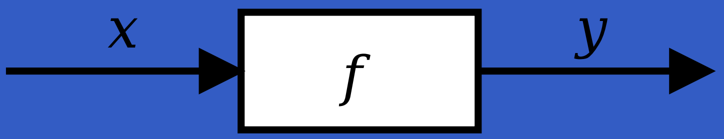
Signal Functions



Intuition:

```
Time ≈ ℝ
Signal a ≈ Time -> a
x :: Signal T1
y :: Signal T2
```

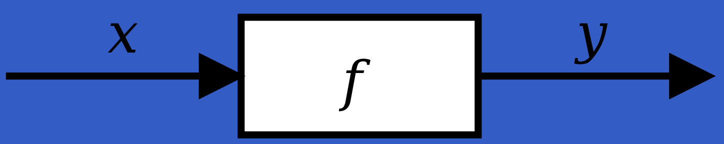
Signal Functions



Intuition:

```
Time ≈ ℝ
Signal a ≈ Time -> a
x :: Signal T1
y :: Signal T2
SF a b ≈ Signal a -> Signal b
f :: SF T1 T2
```

Signal Functions



Intuition:

```
Time ≈ ℝ
Signal a ≈ Time -> a
x :: Signal T1
y :: Signal T2
SF a b ≈ Signal a -> Signal b
f :: SF T1 T2
```

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

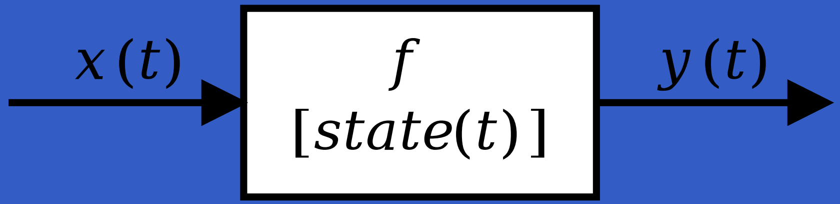
Signal Functions and State

Alternative view:

Signal Functions and State

Alternative view:

Signal functions can encapsulate ***state***.

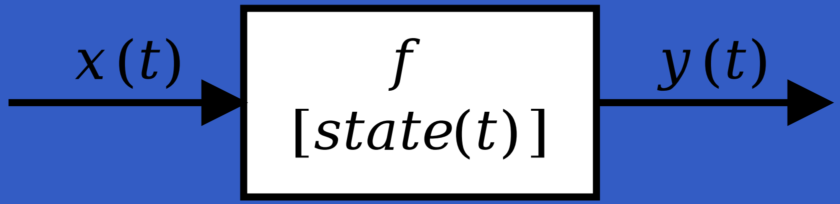


$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

Signal Functions and State

Alternative view:

Signal functions can encapsulate ***state***.



$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

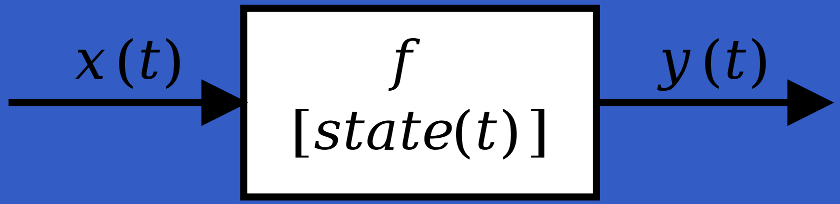
From this perspective, signal functions are:

- ***stateful*** if $y(t)$ depends on $x(t)$ and $state(t)$
- ***stateless*** if $y(t)$ depends only on $x(t)$

Signal Functions and State

Alternative view:

Signal functions can encapsulate ***state***.



$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

From this perspective, signal functions are:

- ***stateful*** if $y(t)$ depends on $x(t)$ and $state(t)$
- ***stateless*** if $y(t)$ depends only on $x(t)$

Signal functions form an arrow.

Some Basic Signal Functions

```
identity :: SF a a
```

Some Basic Signal Functions

```
identity :: SF a a

constant :: b -> SF a b
```

Some Basic Signal Functions

```
identity :: SF a a

constant :: b -> SF a b

iPre :: a -> SF a a
```

Some Basic Signal Functions

```
identity :: SF a a

constant :: b -> SF a b

iPre :: a -> SF a a

integral :: VectorSpace a s=>SF a a
```

$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau$$

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:

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: `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?

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.

Optmimizing >>>: First Attempt (1)

The arrow identity law:

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

Optmimizing >>>: First Attempt (1)

The arrow identity law:

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

How can this be exploited?

Optmimizing >>>: 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
            | ...
```

Optmimizing >>>: 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.

Optmimizing >>>: First Attempt (2)

3. Ensure `SFId` only gets used at intended type:

```
identity :: SF a a
identity = SFId
```

Optmimizing >>>: First Attempt (2)

3. Ensure `SFId` only gets used at intended type:

```
identity :: SF a a
identity = SFId
```

4. Define optimizing version of `>>>`:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
  SFId >>> sf = sf
...
```

Optmimizing >>>: First Attempt (2)

3. Ensure `SFId` only gets used at intended type:

```
identity :: SF a a
identity = SFId
```

4. Define optimizing version of `>>>`:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
  SFId >>> sf = sf
...
```

$:: \texttt{SF b c} \neq \texttt{SF a c}$

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.

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.

Optmimizing >>>: Second Attempt (1)

Instead of

```
data SF a b = ...
            | SFId
            | ...
```

Optmimizing >>>: Second Attempt (1)

Instead of

```
data SF a b = ...
            | SFId        :: SF a b
            | ...
```

Optmimizing >>>: Second Attempt (1)

Instead of

```
data SF a b = ...
            | SFId
            | ...          :: SF a b
```

we define

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

Optmimizing >>>: Second Attempt (2)

Define optimizing version of `>>>` ***exactly*** as before:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
```

Optmimizing >>>: Second Attempt (2)

Define optimizing version of `>>>` ***exactly*** as before:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
    SFId >>> sf = sf
...
```

Optmimizing >>>: Second Attempt (2)

Define optimizing version of `>>>` ***exactly*** as before:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
  SFId >>> sf = sf
...

:: SF a a
```

Optmimizing >>>: Second Attempt (2)

Define optimizing version of `>>>` ***exactly*** as before:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
  SFId >>> sf = sf
...
```

```
:: SF a a
```

```
:: SF a c
```

Other Ways? Statically?

- Other (typed) approaches include keeping coercion functions around as “evidence” for use at runtime (Hughes 2004). But imposes an overhead.

Other Ways? Statically?

- Other (typed) approaches include keeping coercion functions around as “evidence” for use at runtime (Hughes 2004). But imposes an overhead.
- When network structure is static, optimizations can be carried out once and for all. But Yampa networks may evolve over time.

Other Ways? Statically?

- Other (typed) approaches include keeping coercion functions around as “evidence” for use at runtime (Hughes 2004). But imposes an overhead.
- When network structure is static, optimizations can be carried out once and for all. But Yampa networks may evolve over time.

$$\texttt{arr g >>> switch (...) (\textbackslash\_ -> arr f)}$$
$$\xRightarrow{switch} \texttt{arr g >>> arr f} = \texttt{arr (f . g)}$$

Laws Exploited for Optimizations

General arrow laws:

```
(f >>> g) >>> h = f >>> (g >>> h)
    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))
```

Causal Commutative Arrows

- The Yampa arrow satisfies additional laws: in particular it is ***commutative***, meaning ordering between signal functions composed in parallel is irrelevant.

Causal Commutative Arrows

- The Yampa arrow satisfies additional laws: in particular it is ***commutative***, meaning ordering between signal functions composed in parallel is irrelevant.
- This can be exploited (Liu, Cheng, Hudak 2009) to define a ***Causal Commutative Normal Form (CCNF)*** for ***switch-free*** networks.

Causal Commutative Arrows

- The Yampa arrow satisfies additional laws: in particular it is ***commutative***, meaning ordering between signal functions composed in parallel is irrelevant.
- This can be exploited (Liu, Cheng, Hudak 2009) to define a ***Causal Commutative Normal Form (CCNF)*** for ***switch-free*** networks.
- Essentially CCNF is a ***Mealy Machine***.

Causal Commutative Arrows

- The Yampa arrow satisfies additional laws: in particular it is ***commutative***, meaning ordering between signal functions composed in parallel is irrelevant.
- This can be exploited (Liu, Cheng, Hudak 2009) to define a ***Causal Commutative Normal Form (CCNF)*** for ***switch-free*** networks.
- Essentially CCNF is a ***Mealy Machine***.
- Not exploited in Yampa, but this optimization has been used to obtain performance gains of two orders of magnitude (over Yampa-like performance).

Implementation (1)

```
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
```

Implementation (2)

```
data FunDesc a b where
    FDI :: FunDesc a a
    FDC :: b -> FunDesc a b
    FDG :: (a -> b) -> FunDesc a b
```

Implementation (2)

```
data FunDesc a b where
    FDI :: FunDesc a a
    FDC :: b -> FunDesc a b
    FDG :: (a -> b) -> FunDesc a b
```

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 (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:

```
data Event a = NoEvent | Event a
```

Discrete-time signal $= \texttt{Signal}\,(\texttt{Event}\,\alpha)$.

Events

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

```
data Event a = NoEvent | Event a
```

Discrete-time signal $= \texttt{Signal}\,(\texttt{Event}\,\alpha)$.

Consider composition of pure event processing:

```
f :: Event a -> Event b
g :: Event b -> Event c

arr f >>> arr g
```

Optimizing Event Processing (1)

Additional function descriptor:

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

Optimizing Event Processing (1)

Additional function descriptor:

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

Optimizing Event Processing (1)

Additional function descriptor:

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

Extend the composition function:

```
fdComp (FDE f1 f1ne) fd2 =
    FDE (f2 . f1) (f2 f1ne)
    where
        f2 = fdFun fd2
```

Optimizing Event Processing (2)

Extend the composition function:

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

Optimizing Event Processing (2)

Extend the composition function:

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

Optimizing Stateful Event Processing

A general stateful event processor:

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

Optimizing Stateful Event Processing

A general stateful event processor:

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

Composes nicely with stateful and stateless event processors!

Optimizing Stateful Event Processing

A general stateful event processor:

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

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

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

Cause for Concern

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

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

Is the result really a performance improvement?

Cause for Concern

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

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

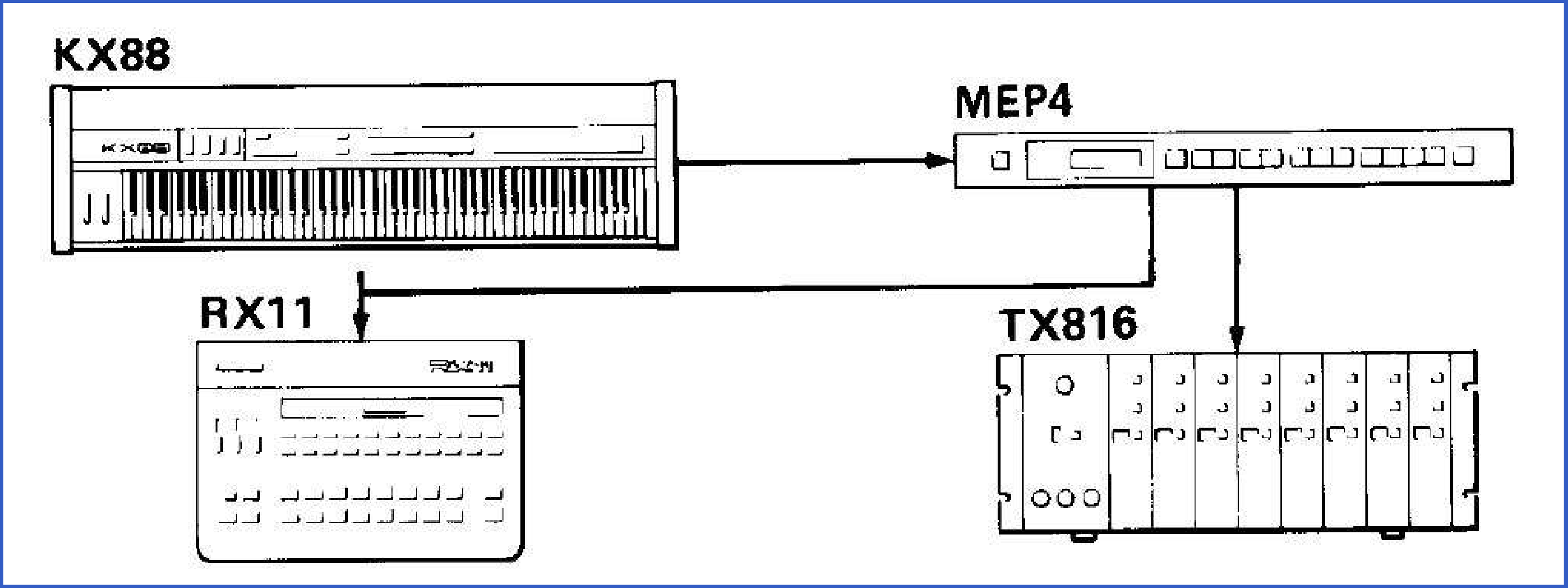
Is the result really a performance improvement? A number of Micro Benchmarks were carried out to verify that individual optimizations worked as intended, including:

- Space Invaders
- MIDI Event Processor

Benchmark 1: Space Invaders

Benchmark 2: MIDI Event Processor

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



The MEP4


YAMAHA
MEP4
Oberheim
midi express
Cheetah
MS6 MULTI TIMBRAL SYNTHESIZER MODULE
YAMAHA

Results

Benchmark	T_{U} [s]	T_{O} [s]	$T_{\mathrm{O}}/T_{\mathrm{U}}$
Space Inv.	0.95	0.86	0.93
MEP	19.39	10.31	0.48

Results

Benchmark	T_{U} [s]	T_{O} [s]	$T_{\mathrm{O}}/T_{\mathrm{U}}$
Space Inv.	0.95	0.86	0.93
MEP	19.39	10.31	0.48

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.