MGS 2005
Functional Reactive Programming

Lecture 1: Introduction to FRP, Yampa, and Arrows

Henrik Nilsson

School of Computer Science and Information Technology

University of Nottingham, UK

Outline

- Brief introduction to FRP and Yampa
- Signal functions
- Arrows

Reactive programming

Reactive systems:

Reactive programming

Reactive systems:

- Input arrives ***incrementally*** while system is running.

Reactive programming

Reactive systems:

- Input arrives ***incrementally*** while system is running.
- Output is generated in response to input in an interleaved and ***timely*** fashion.

Reactive programming

Reactive systems:

- Input arrives ***incrementally*** while system is running.
- Output is generated in response to input in an interleaved and ***timely*** fashion.

Contrast ***transformational systems***.

Reactive programming

Reactive systems:

- Input arrives ***incrementally*** while system is running.
- Output is generated in response to input in an interleaved and ***timely*** fashion.

Contrast ***transformational systems***.

The notions of

- time
- time-varying values, or ***signals***

are inherent and central for reactive systems.

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.

FRP applications

Some domains where FRP has been used:

- Graphical Animation (Fran: Elliott, Hudak)
- Robotics (Frob: Peterson, Hager, Hudak, Elliott, Pembeci, Nilsson)
- Vision (FVision: Peterson, Hudak, Reid, Hager)
- GUIs (Fruit: Courtney)
- Hybrid modeling (Nilsson, Hudak, Peterson)

Key FRP features

- First class reactive components.

Key FRP features

- First class reactive components.
- Synchronous: all system parts operate in synchrony.

Key FRP features

- First class reactive components.
- Synchronous: all system parts operate in synchrony.
- Support for hybrid (mixed continuous and discrete time) systems.

Key FRP features

- First class reactive components.
- Synchronous: all system parts operate in synchrony.
- Support for hybrid (mixed continuous and discrete time) systems.
- Allows dynamic system structure.

Related languages and paradigms

FRP related to:

- Synchronous languages, like Esterel, Lucid Synchrone.

Related languages and paradigms

FRP related to:

- Synchronous languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink, Modelica.

Yampa

What is ***Yampa***?

- The most recent Yale FRP implementation. People:
  - Antony Courtney
  - Paul Hudak
  - Henrik Nilsson
  - John Peterson

Yampa

What is ***Yampa***?

- The most recent Yale FRP implementation. People:
  - Antony Courtney
  - Paul Hudak
  - Henrik Nilsson
  - John Peterson
- A Haskell ***combinator library***, a.k.a. ***Domain-Specific Embedded Language*** (DSEL).

Yampa

What is ***Yampa***?

- Structured using ***arrows***.

Yampa

What is ***Yampa***?

- Structured using ***arrows***.
- ***Continuous-time*** signals (conceptually)

Yampa

What is ***Yampa***?

- Structured using ***arrows***.
- ***Continuous-time*** signals (conceptually)
- Option type ***Event*** to handle discrete-time signals.

Yampa

What is ***Yampa***?

- Structured using ***arrows***.
- ***Continuous-time*** signals (conceptually)
- Option type ***Event*** to handle discrete-time signals.
- Advanced ***switching constructs*** to describe systems with dynamic structure.

Yampa?

Yampa?

Y et

A nother

M ostly

P ointless

A cronym

Yampa?

Yet

Another

Mostly

Pointless

Acronym

???

Yampa?

Yet
Another
Mostly
Pointless
Acronym

???

No . . .

Yampa?

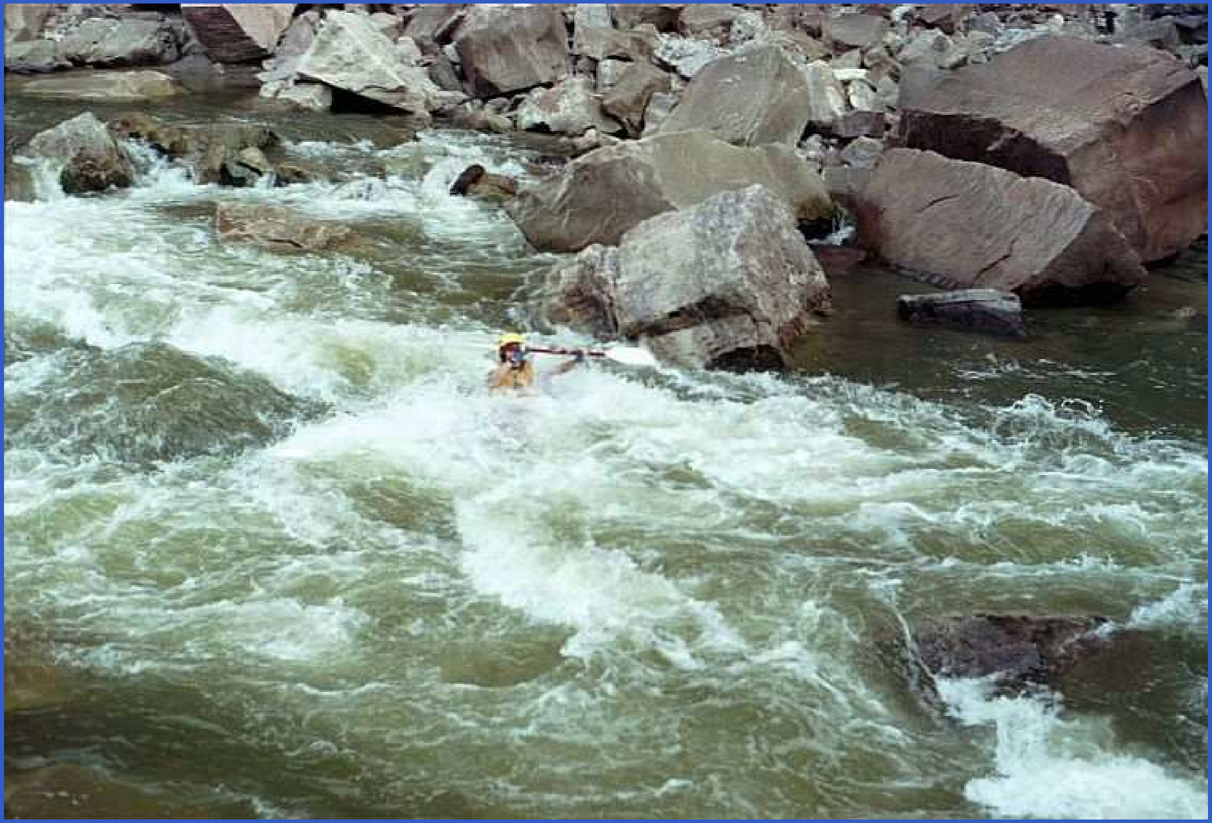
Yampa is a river . . .

Yampa?

. . . with long calmly flowing sections . . .

Yampa?

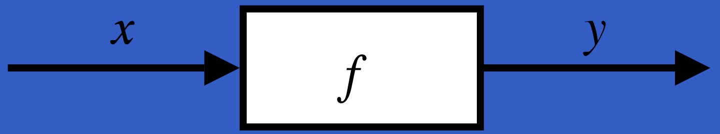
. . . and abrupt whitewater transitions in between.



A good metaphor for hybrid systems!

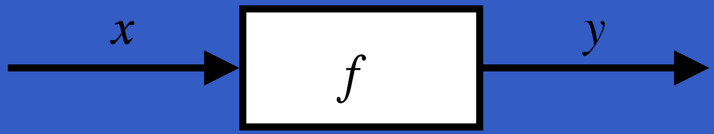
Signal functions (1)

Key concept: ***functions on signals***.



Signal functions (1)

Key concept: ***functions on signals***.



Intuition:

$$\texttt{Signal}\ \alpha \approx \texttt{Time}\rightarrow\alpha$$
$$x :: \texttt{Signal T1}$$
$$y :: \texttt{Signal T2}$$
$$f :: \texttt{Signal T1} \rightarrow \texttt{Signal T2}$$

Signal functions (2)

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

Signal functions (2)

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

Signal functions are said to be

- ***pure*** or ***stateless*** if output at time t only depends on input at time t

Signal functions (2)

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

Signal functions are said to be

- ***pure*** or ***stateless*** if output at time t only depends on input at time t
- ***impure*** or ***stateful*** if output at time t depends on input over the interval $[0, t]$.

Signal functions in Yampa

- ***Signal functions*** are ***first class entities***.
  Intuition: $\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$

Signal functions in Yampa

- ***Signal functions*** are ***first class entities***.
  Intuition: $\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$
- ***Signals*** are ***not*** first class entities: they only exist indirectly through signal functions.

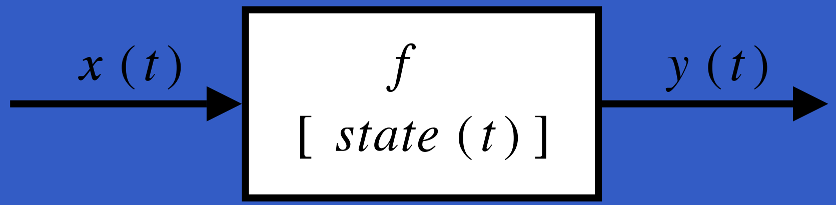
Signal functions and state

Alternative view:

Signal functions and state

Alternative view:

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

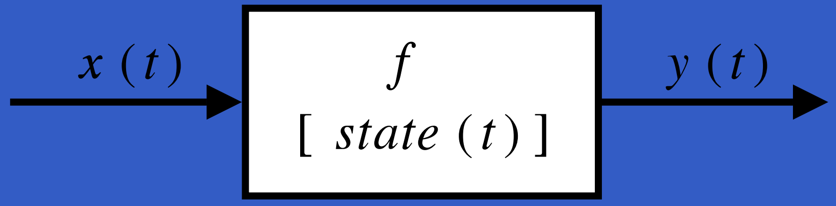


$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.
Thus, really a kind of ***process***.

Signal functions and state

Alternative view:

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



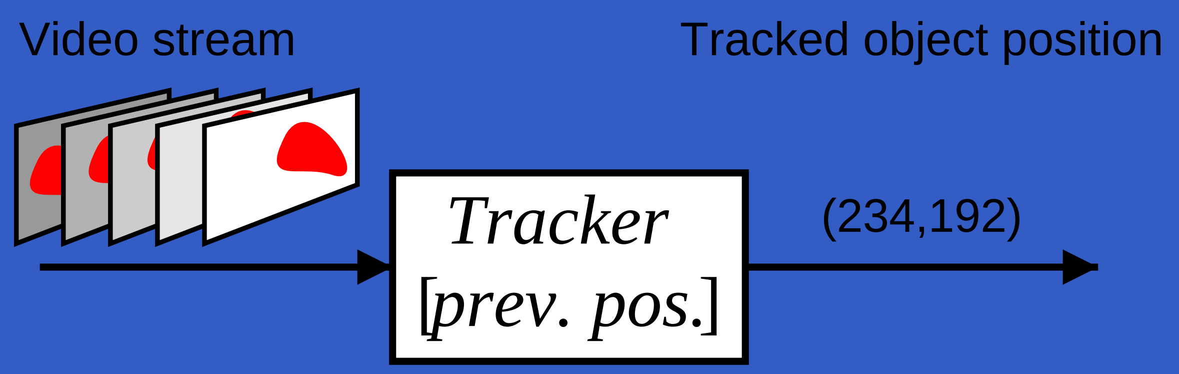
$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.
Thus, really a kind of ***process***.

From this perspective, signal functions are:

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

Example: Video tracker

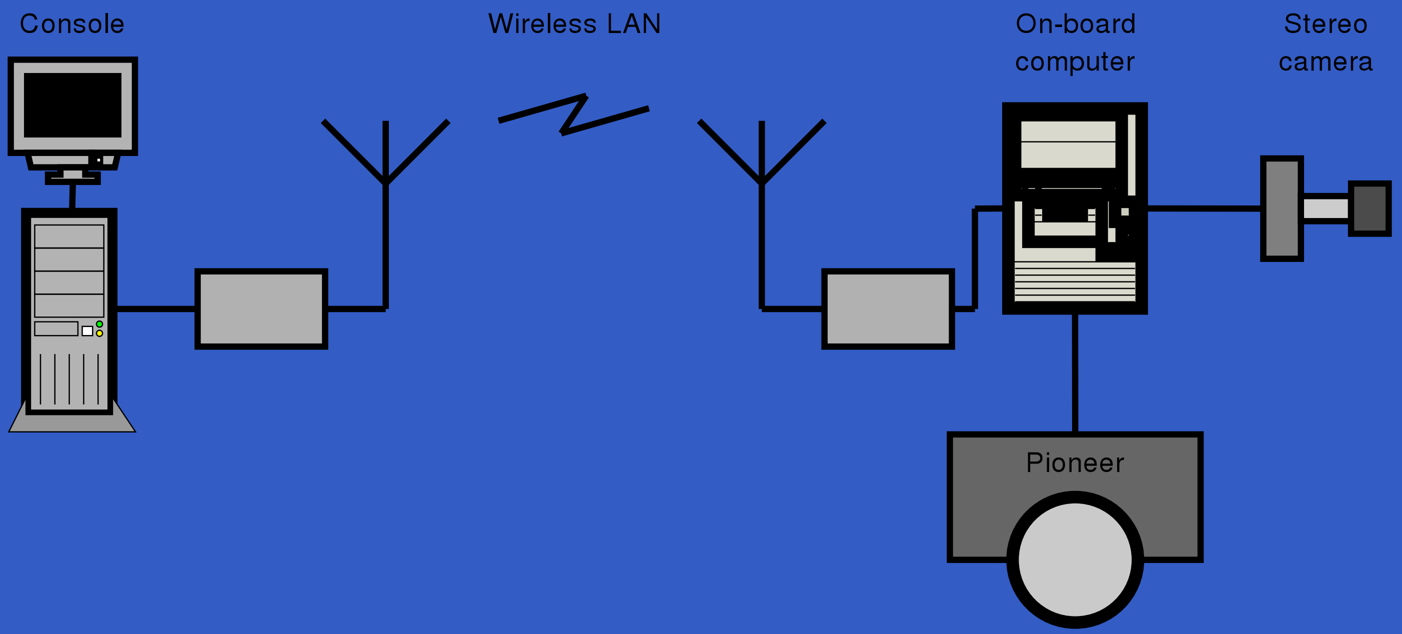
Video trackers are typically stateful signal functions:



Example: Robotics (1)

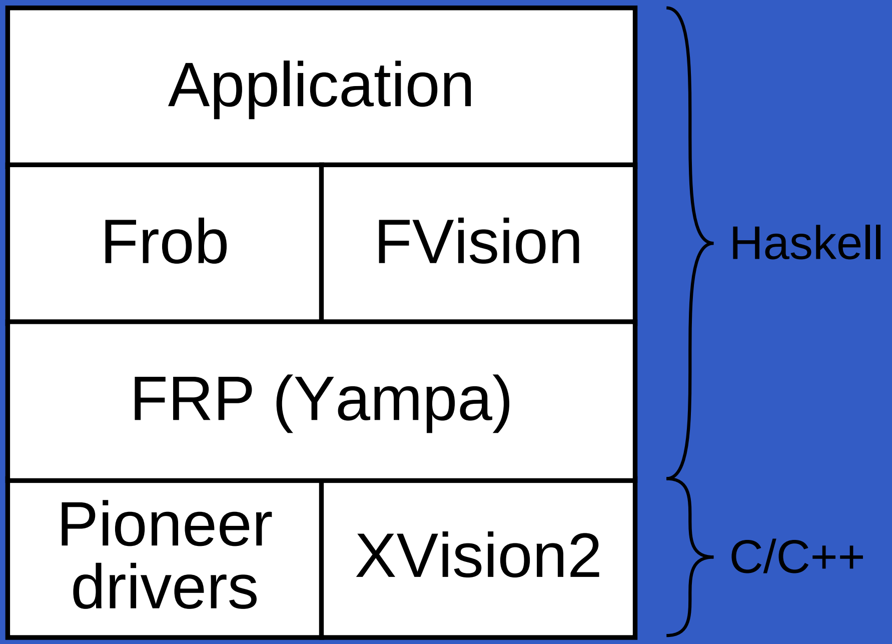
[PPDP'02, with Izzet Pembeci and Greg Hager, Johns Hopkins University]

Hardware setup:



Example: Robotics (2)

Software architecture:



Example: Robotics (3)

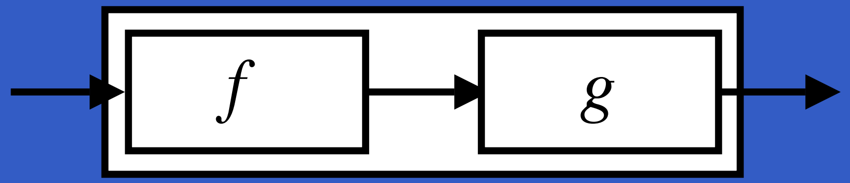
Yampa and Arrows (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

Yampa and Arrows (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

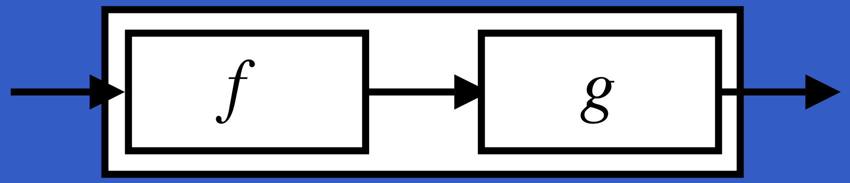
For example, serial composition:



Yampa and Arrows (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:

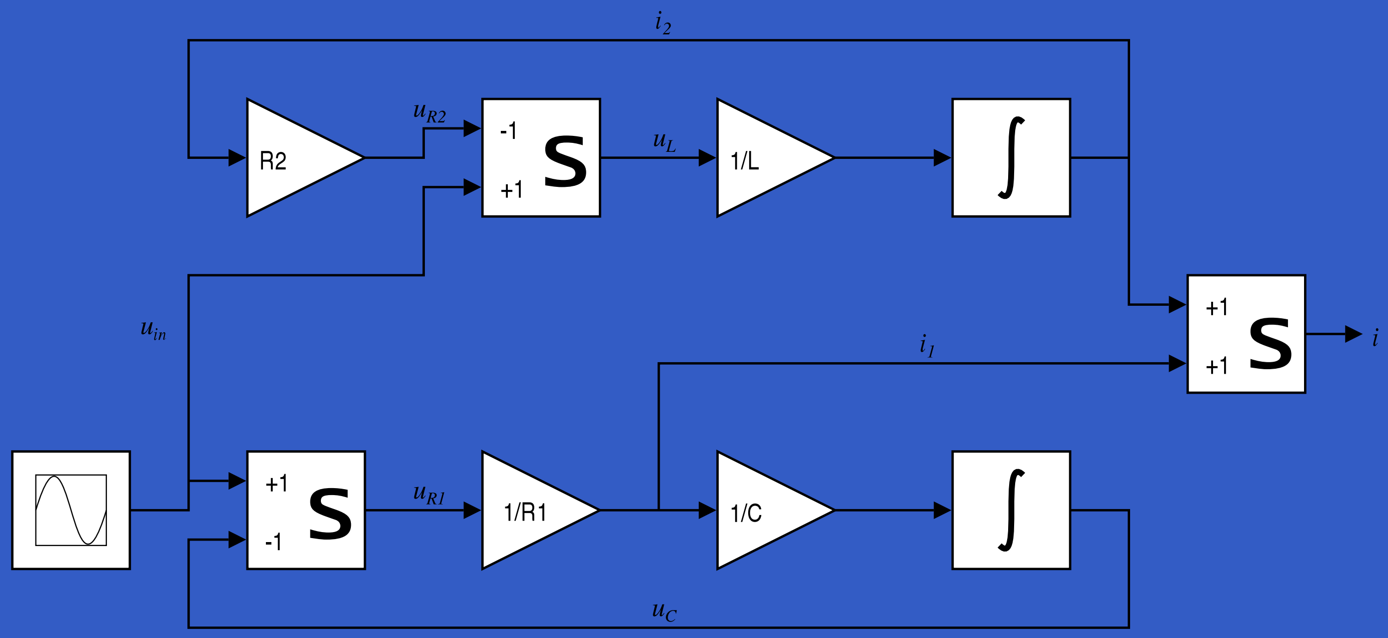


A *combinator* can be defined that captures this idea:

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

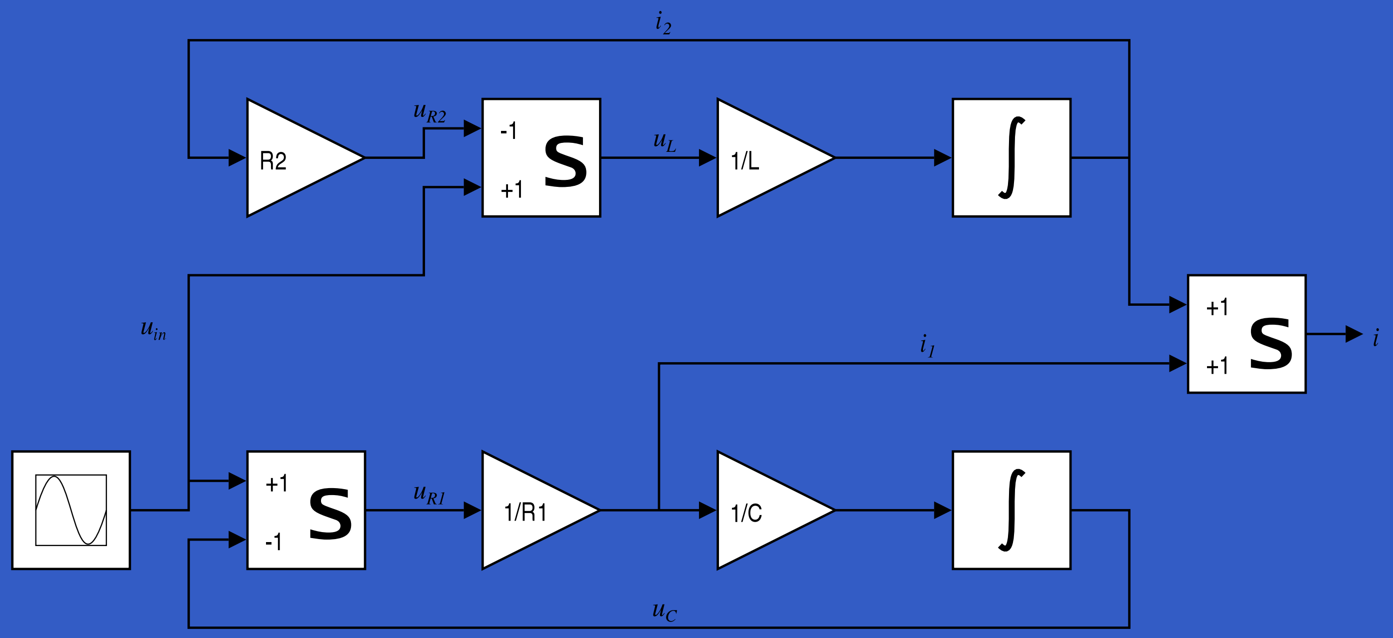
Yampa and Arrows (2)

But systems can be complex:



Yampa and Arrows (2)

But systems can be complex:



How many and what combinators do we need to be able to describe arbitrary systems?

Yampa and Arrows (3)

John Hughes' ***arrow*** framework:

- Abstract data type interface for function-like types.

Yampa and Arrows (3)

John Hughes' ***arrow*** framework:

- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.

Yampa and Arrows (3)

John Hughes' ***arrow*** framework:

- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.
- Related to ***monads***, since arrows are computations, but more general.

Yampa and Arrows (3)

John Hughes' ***arrow*** framework:

- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.
- Related to ***monads***, since arrows are computations, but more general.
- Provides a minimal set of "wiring" combinators.

What is an arrow? (1)

- A ***type constructor*** a of arity two.

What is an arrow? (1)

- A ***type constructor*** a of arity two.
- Three operators:

What is an arrow? (1)

- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:

    ```
    arr :: (b->c) -> a b c
    ```

What is an arrow? (1)

- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:
    ```
    arr :: (b->c) -> a b c
    ```
  - ***composition***:
    ```
    (>>>) :: a b c -> a c d -> a b d
    ```

What is an arrow? (1)

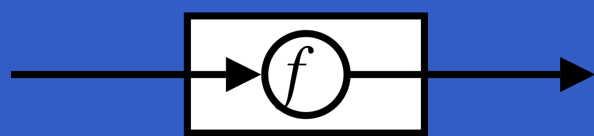
- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:
    `arr :: (b->c) -> a b c`
  - ***composition***:
    `(>>>) :: a b c -> a c d -> a b d`
  - ***widening***:
    `first :: a b c -> a (b,d) (c,d)`

What is an arrow? (1)

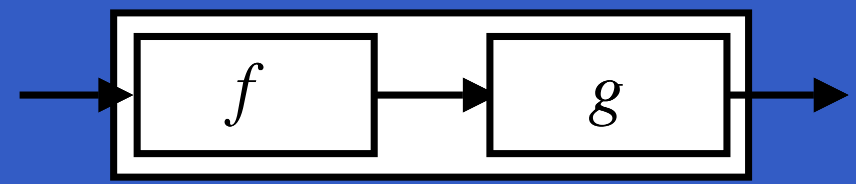
- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:
    `arr :: (b->c) -> a b c`
  - ***composition***:
    `(>>>) :: a b c -> a c d -> a b d`
  - ***widening***:
    `first :: a b c -> a (b,d) (c,d)`
- A set of ***algebraic laws*** that must hold.

What is an arrow? (2)

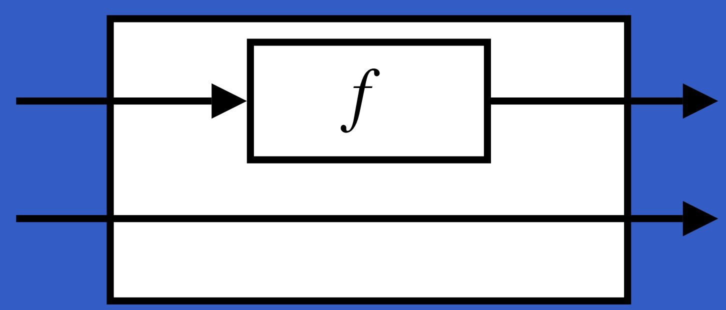
These diagrams convey the general idea:



`arr` f



f `>>>` g



`first` f

The `Arrow` class

In Haskell, a ***type class*** is used to capture these ideas (except for the laws):

```
class Arrow a where
    arr   :: (b -> c) -> a b c
    (>>>) :: a b c -> a c d -> a b d
    first :: a b c -> a (b,d) (c,d)
```

Functions are arrows (1)

Functions are a simple example of arrows. The arrow type constructor is just `(->)` in that case.

Exercise 1: Suggest suitable definitions of

- `arr`
- `(>>>)`
- `first`

for this case!

(We have not looked at what the laws are yet, but they are “natural”.)

Functions are arrows (2)

Solution:

- `arr = id`

Functions are arrows (2)

Solution:

- `arr = id`
  To see this, recall

```
id :: t -> t
arr :: (b->c) -> a b c
```

Functions are arrows (2)

Solution:

- `arr = id`

  To see this, recall

  ```
  id :: t -> t
  arr :: (b->c) -> a b c
  ```

  Instantiate with

  ```
  a = (->)
  t = b->c = (->) b c
  ```

Functions are arrows (3)

- `f >>> g = \a -> g (f a)`

Functions are arrows (3)

- `f >>> g = \a -> g (f a)` ***or***
- `f >>> g = g . f`

Functions are arrows (3)

- `f >>> g = \a -> g (f a)` ***or***
- `f >>> g = g . f` ***or even***
- `(>>>) = flip (.)`

Functions are arrows (3)

- `f >>> g = \a -> g (f a)` ***or***
- `f >>> g = g . f` ***or even***
- `(>>>) = flip (.)`
- `first f = \(b,d) -> (f b,d)`

Functions are arrows (4)

`Arrow` instance declaration for functions:

```
instance Arrow (->) where
        arr     = id
        (>>>)   = flip (.)
        first f = \(b,d) -> (f b,d)
```

Arrow laws

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

Arrow laws

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

Arrow laws

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

Arrow laws

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

Arrow laws

```
(f >>> g) >>> h = f >>> (g >>> h)
  arr (f >>> g) = arr f >>> arr g
   arr id >>> f = f
              f = f >>> arr id
  first (arr f) = arr (first f)
```

Arrow laws

```
(f >>> g) >>> h = f >>> (g >>> h)
  arr (f >>> g) = arr f >>> arr g
   arr id >>> f = f
              f = f >>> arr id
  first (arr f) = arr (first f)
first (f >>> g) = first f >>> first g
```

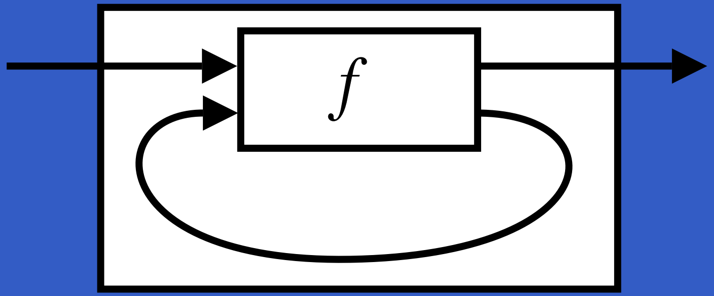
Arrow laws

```
(f >>> g) >>> h = f >>> (g >>> h)
  arr (f >>> g) = arr f >>> arr g
   arr id >>> f = f
              f = f >>> arr id
  first (arr f) = arr (first f)
first (f >>> g) = first f >>> first g
```

Exercise 2: Draw diagrams illustrating the first and last law!

The `loop` combinator (1)

Another important operator is `loop`: a fixed-point operator used to express recursive arrows or ***feedback***:



`loop` f

The `loop` combinator (2)

Not all arrow instances support `loop`. It is thus a method of a separate class:

```
class Arrow a => ArrowLoop a where
    loop :: a (b, d) (c, d) -> a b c
```

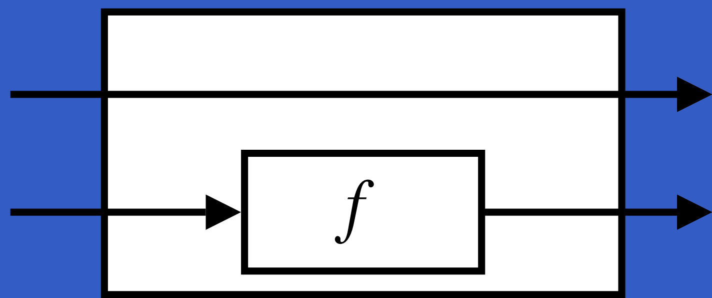
Remarkably, the four combinators `arr`, `>>>`, `first`, and `loop` are sufficient to express any conceivable wiring!

Some more arrow combinators (1)

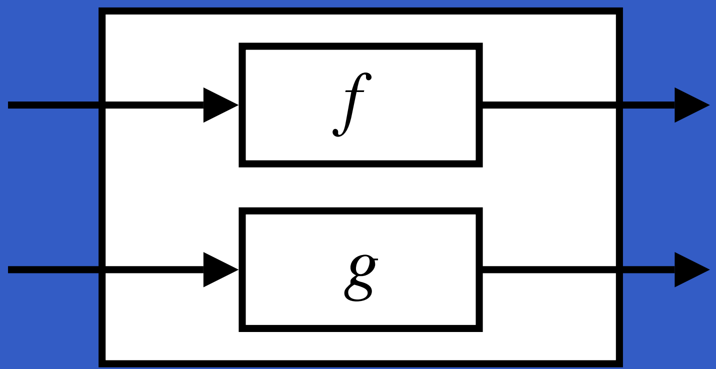
```
second :: Arrow a =>
    a b c -> a (d,b) (d,c)

(***) :: Arrow a =>
    a b c -> a d e -> a (b,d) (c,e)

(&&&) :: Arrow a =>
    a b c -> a b d -> a b (c,d)
```

Some more arrow combinators (2)

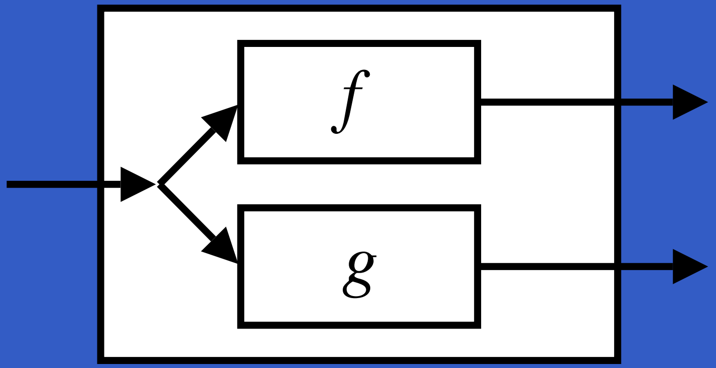
As diagrams:



second f



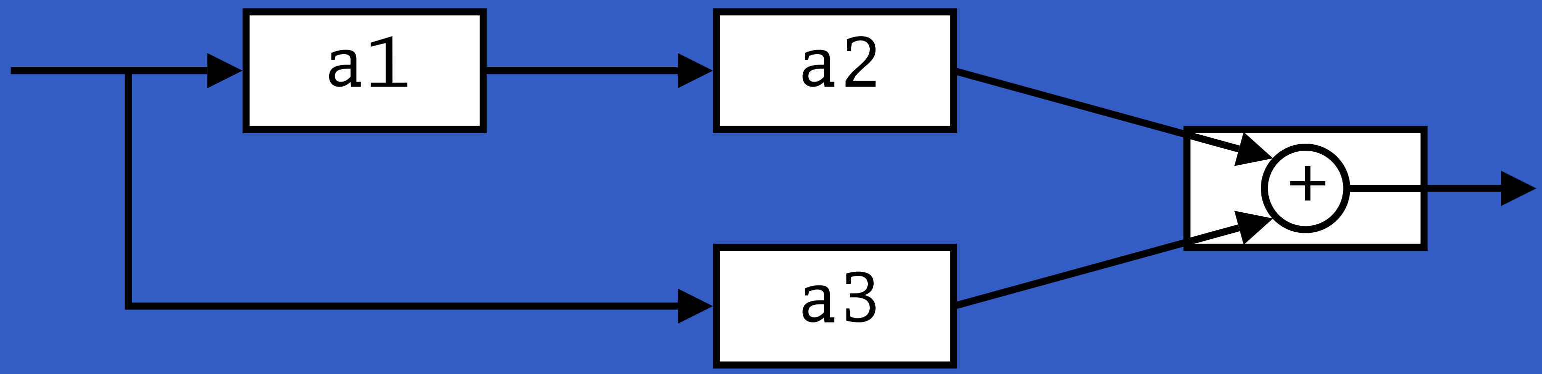
f *** g



f &&& g

Some more arrow combinators (3)

Exercise 3: Describe the following circuit using arrow combinators:



`a1, a2, a3 :: A Double Double`

Exercise 4: The combinators `second`, `(***)`, and `(&&&)` are not primitive, but defined in terms of `arr`, `(>>>)`, and `first`. Suggest suitable definitions!

Reading (1)

- John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000
- John Hughes. Programming with arrows. In *Advanced Functional Programming*, 2004. To be published by Springer Verlag.
- Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 Haskell Workshop*, pp. 51–64, October 2002.

Reading (2)

- Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, 2002. LNCS 2638, pp. 159–187.