System descriptions in the form of block diagrams are very common. Blocks have inputs and outputs and can be combined into larger blocks. For example, serial composition:

\[(\triangleright\triangleright\triangleright) :: B \ a \ b \rightarrow B \ b \ c \rightarrow B \ a \ c\]
System descriptions in the form of block diagrams are very common. Blocks have inputs and outputs and can be combined into larger blocks. For example, serial composition:

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

\[(\gg\gg\gg) :: B \ a \ b \rightarrow B \ b \ c \rightarrow B \ a \ c\]
But systems can be complex:
But systems can be complex:

How many and what combinators do we need to be able to describe arbitrary systems?
Arrows (3)

John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).
John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).
- Particularly suitable for types representing process-like computations.
John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).
- Particularly suitable for types representing process-like computations.
- Related to *monads*, since arrows are computations, but more general.
John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).
- 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 of arity two.

Three operators:
- Lifting: \(\text{arr} :: (b\rightarrow c) \rightarrow a \rightarrow b \rightarrow c\)
- Composition: \(\text{>>>} :: a \rightarrow b \rightarrow c \rightarrow a \rightarrow b \rightarrow d\)
- Widening: \(\text{first} :: a \rightarrow b \rightarrow c \rightarrow a \rightarrow (b,d) \rightarrow (c,d)\)

A set of algebraic laws that must hold.
What is an arrow? (1)

- A type constructor $\alpha$ 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*:
    
    \[
    \text{arr} :: (b \to c) \to a \ b \ c
    \]
What is an arrow? (1)

- A type constructor \( a \) of arity two.
- Three operators:
  - \textit{lifting}:
    \[
    \text{arr} :: (b \rightarrow c) \rightarrow a \ b \ c
    \]
  - \textit{composition}:
    \[
    (\gggg) :: a \ b \ c \rightarrow a \ c \ d \rightarrow a \ b \ d
    \]
What is an arrow? (1)

• A *type constructor* `a` of arity two.

• Three operators:
  - *lifting*:
    \[ \text{arr} :: (b \to c) \to a b c \]
  - *composition*:
    \[ (\gg\gg) :: a b c \to a c d \to a b d \]
  - *widening*:
    \[ \text{first} :: a b c \to a (b, d) (c, d) \]
What is an arrow? (1)

- A **type constructor** \( a \) of arity two.
- Three operators:
  - *lifting*:
    \[ \text{arr} :: (b \to c) \to a \ b \ c \]
  - *composition*:
    \[ (\gggg) :: a \ b \ c \to a \ c \ d \to a \ b \ d \]
  - *widening*:
    \[ \text{first} :: a \ b \ c \to a \ (b, d) \ (c, d) \]
- A set of **algebraic laws** that must hold.
What is an arrow? (2)

These diagrams convey the general idea:

arr $f$

$f \Rightarrow g$

first $f$
The *Arrow* class

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

```haskell
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)
```

MGS 2007: ADV Lecture 3 – p.7/46
Functions are arrows (1)

Functions are a simple example of arrows, with \((\to)\) as the arrow type constructor.

**Exercise 1:** Suggest suitable definitions of

- \(\text{arr}\)
- \(\text{>>>()}\)
- \(\text{first}\)

for this case!

(We have not looked at what the laws are yet, but they are “natural”.)
Functions are arrows (2)

Solution:

- \texttt{arr = id}
Solution:

- arr = id

To see this, recall

\[ id :: t \rightarrow t \]
\[ arr :: (b \rightarrow c) \rightarrow a \ b \ c \]
Functions are arrows (2)

Solution:

- \( arr = id \)
  
  To see this, recall
  
  \[
  \text{id} :: t \rightarrow t \\
  \text{arr} :: (b\rightarrow c) \rightarrow a \ b \ c
  \]

  Instantiate with

  \[
  a = (\rightarrow) \\
  t = b\rightarrow c = (\rightarrow) b \ c
  \]
Functions are arrows (3)

- \( f >>> g = \lambda a \rightarrow g (f a) \)
Functions are arrows (3)

- $f >>> g = \lambda a \rightarrow g (f a)$  \textit{or}
- $f >>> g = g \cdot f$

MGS 2007: ADV Lecture 3 – p.10/46
Functions are arrows (3)

- $f >>> g = \lambda a \rightarrow g (f a)$  \textit{or}
- $f >>> g = g . f$  \textit{or even}
- $(<<<) = \text{flip } (.)$
Functions are arrows (3)

- \( f >>> g = \lambda a \rightarrow g (f a) \)  
  - or
- \( f >>> g = g . f \)  
  - or even
- \( (>>>) = \text{flip} \ (.) \)
- \( \text{first } f = \lambda (b,d) \rightarrow (f b,d) \)
functions are arrows (4)

Arrow instance declaration for functions:

instance Arrow (->) where
   arr      = id
   (>>>)    = flip (.)
   first f = \( (b, d) \rightarrow (f b, d) \)
Some arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]
Some arrow laws

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

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

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr } (f >>> g) = \text{arr } f >>> \text{arr } g\]
\[\text{arr } \text{id} >>> f = f\]
Some arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr } (f >>> g) = \text{arr } f >>> \text{arr } g\]
\[\text{arr } \text{id} >>> f = f\]
\[f = f >>> \text{arr } \text{id}\]
Some arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr } (f >>> g) = \text{arr } f >>> \text{arr } g\]
\[\text{arr id } >>> f = f\]
\[f = f >>> \text{arr id}\]
\[\text{first } (\text{arr } f) = \text{arr } (\text{first } f)\]
Some arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr} \ (f >>> g) = \text{arr} \ f >>> \text{arr} \ g\]
\[\text{arr id} >>> f = f\]
\[f = f >>> \text{arr id}\]
\[\text{first} \ (\text{arr} \ f) = \text{arr} \ (\text{first} \ f)\]
\[\text{first} \ (f >>> g) = \text{first} \ f >>> \text{first} \ g\]
Some 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!
Another important operator is \texttt{loop}: a fixed-point operator used to express recursive arrows or \textit{feedback}:

\begin{center}
\texttt{loop} \ f
\end{center}
The loop combinator (2)

Not all arrow instances support \texttt{loop}. It is thus a method of a separate class:

\begin{verbatim}
class Arrow a => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c
\end{verbatim}

Remarkably, the four combinators \texttt{arr}, \texttt{>>>}, \texttt{first}, and \texttt{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)

second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap

swap (x,y) = (y,x)

(*** :: Arrow a => a b c -> a d e -> a (b,d) (c,e)
f *** g = first f >>> second g

(&&& :: Arrow a => a b c -> a b d -> a b (c,d)
f &&& g = arr (\x->(x,x)) >>> (f *** g)
Some more arrow combinators (3)

second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)
Some more arrow combinators (3)

second :: Arrow a => a b c -> a (d, b) (d, c)
second f = arr swap >>> first f >>> arr swap
swap (x, y) = (y, x)

(***) :: Arrow a => a b c -> a d e -> a (b, d) (c, e)
f *** g = first f >>> second g
Some more arrow combinators (3)

second :: Arrow a => a b c -> a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
swap (x,y) = (y,x)

(*** :: Arrow a =>
    a b c -> a d e -> a (b,d) (c,e)
f *** g = first f >>> second g

(&&&) :: Arrow a =>
    a b c -> a b d -> a b (c,d)
f &&& g = arr (\x->(x,x)) >>> (f *** g)
Exercise 3

Describe the following circuit using arrow combinators:

\[ \text{a1, a2, a3} :: \text{A Double Double} \]
Exercise 3: One solution

Exercise 3: Describe the following circuit using arrow combinators:

\[
\text{a1, a2, a3} :: \text{A Double Double}
\]

\[
\text{circuit} \_v1 :: \text{A Double Double}
\]

\[
\text{circuit} \_v1 = (\text{a1} \&\& \text{arr id}) \triangleright\triangleright (\text{a2} \**\** \text{a3}) \triangleright\triangleright \text{arr (uncurry (+))}
\]
Exercise 3: Describe the following circuit using arrow combinators:

```
Exercise 3: One solution

a1, a2, a3 :: A Double Double

circuit_v1 :: A Double Double
circuit_v1 = (a1 &&& arr id) >>> (a2 *** a3) >>> arr (uncurry (+))
```
Exercise 3: Describe the following circuit:

\[ a1, a2, a3 :: \text{A Double Double} \]

\[
\text{circuit}\_v2 :: \text{A Double Double}
\]

\[
\text{circuit}\_v2 = \text{arr} \ (\lambda x \rightarrow (x,x))
\]

\[
\begin{split}
&\begin{array}{c}
\text{first} \ a1\\
\text{(a2 *** a3)}\\
\text{arr} \ (\text{uncurry} \ (+))
\end{array}
\end{split}
\]
Exercise 3: Another solution

**Exercise 3:** Describe the following circuit:

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

circuit_v2 :: A Double Double

circuit_v2 = arr (\x -> (x,x))
            >>> first a1
            >>> (a2 *** a3)
            >>> arr (uncurry (+))
```
Are the following two definitions of (*** ) equivalent?

- \( f \ *** \ g = \text{first } f \ggg \text{second } g \)
- \( f \ *** \ g = \text{second } g \ggg \text{first } f \)

No, in general, \( f \ggg \text{second } g \neq \text{second } g \ggg f \) since the order of the two possibly effectful computations \( f \) and \( g \) are different.
Note on the definition of (***) (1)

Are the following two definitions of (***) equivalent?

- $f *** g = \text{first } f >>> \text{second } g$
- $f *** g = \text{second } g >>> \text{first } f$

No, in general

$\text{first } f >>> \text{second } g \neq \text{second } g >>> \text{first } f$

since the order of the two possibly effectful computations $f$ and $g$ are different.
Similarly

\[(f *** g) >>> (h *** k) \neq (f >>> h) *** (g >>> k)\]

since the order of \(f\) and \(g\) differs.
Note on the definition of (***) (2)

Similarly

\[(f *** g) >>> (h *** k) \neq (f >>> h) *** (g >>> k)\]

since the order of \(f\) and \(g\) differs.

However, the following is true (an additional law):

\[
\text{first } f >>> \text{second } (\text{arr } g) = \text{second } (\text{arr } g) >>> \text{first } f
\]

However, for certain arrow instances equalities like the ones above do hold.
Yet an attempt at exercise 3

circuit_v3 :: A Double Double
circuit_v3 = (a1 &&& a3)
    >>> first a2
    >>> arr (uncurry (+))
Yet an attempt at exercise 3

\[
\text{circuit}_v^3 :: \text{A Double Double}
\]
\[
\text{circuit}_v^3 = (a_1 \&\& a_3)
\]
\[
\text{a1} \implies \text{first a2} \\
\text{a2} \implies \text{arr (uncurry (+))}
\]

**Exercise 4**: Are \text{circuit}_v^1, \text{circuit}_v^2, and \text{circuit}_v^3 all equivalent?
Ross Paterson’s \( \text{do} \)-notation for arrows supports \textit{pointed} arrow programming. Only \textit{syntactic sugar}.

\[
\begin{align*}
\text{proc } pat & \rightarrow \text{do} [ \text{rec} ] \\
& \quad pat_1 \leftarrow \text{sfexp}_1 \leftarrow \text{exp}_1 \\
& \quad pat_2 \leftarrow \text{sfexp}_2 \leftarrow \text{exp}_2 \\
& \quad \ldots \\
& \quad pat_n \leftarrow \text{sfexp}_n \leftarrow \text{exp}_n \\
& \quad \text{return A} \leftarrow \text{exp}
\end{align*}
\]

Also: \( \text{let } pat = exp \equiv pat \leftarrow \text{arr id} \leftarrow exp \)
Let us redo exercise 3 using this notation:

\[
\text{circuit\_v4 :: A Double Double}
\]

\[
\text{circuit\_v4 = proc x -> do}
\]

\[
\begin{align*}
  y1 & \leftarrow a1 \leftarrow x \\
  y2 & \leftarrow a2 \leftarrow y1 \\
  y3 & \leftarrow a3 \leftarrow x \\
  \text{returnA} & \leftarrow y2 + y3
\end{align*}
\]
The arrow do notation (3)

We can also mix and match:

circuit_v5 :: A Double Double

circuit_v5 = proc x -> do
  y2 <- a2 <<< a1 <<< x
  y3 <- a3 <<< x
  returnA <<< y2 + y3
Recursive networks: $\mathbf{do}$-notation:

\[
\begin{align*}
\text{a1} & \rightarrow \text{a2} \\
\text{a3} & \rightarrow \text{a2} \\
\text{a3} & \rightarrow + \\
+ & \rightarrow \text{a1}
\end{align*}
\]

\[
a1, a2 :: A \text{ Double Double} \\
a3 :: A (\text{Double,Double}) \text{ Double}
\]
The arrow $\texttt{do}$ notation (4)

Recursive networks: $\texttt{do}$-notation:

- $a_1$, $a_2 :: A \, \text{Double} \, \text{Double}$
- $a_3 :: A \, (\text{Double},\text{Double}) \, \text{Double}$

**Exercise 5:** Describe this using only the arrow combinators.
The arrow do notation (5)

circuit = proc x -> do
    rec
    y1 <- a1 <-- x
    y2 <- a2 <-- y1
    y3 <- a3 <-- (x, y)
    let y = y2 + y3
    returnA <-- y
Arrows and Monads (1)

Arrows generalize monads: for every monad type there is an arrow, the *Kleisli category* for the monad:

\[
\text{newtype Kleisli m a b = K (a \rightarrow m b)}
\]

\[
\text{instance Monad m => Arrow (Kleisli m) where}
\]
\[
\text{arr f = K (\lambda b \rightarrow return (f b))}
\]
\[
\text{K f >>> K g = K (\lambda b \rightarrow f b >>= g)}
\]
Arrows and Monads (2)

But not every arrow is a monad. However, arrows that support an additional `apply` operation are effectively monads:

\[
\text{apply} :: \text{Arrow } a \Rightarrow a (a \ b \ c, \ b) \ c
\]

Exercise 6: Verify that

\[
\text{newtype } M \ b = M (A () \ b)
\]

is a monad if `A` is an arrow supporting `apply`; i.e., define `return` and `bind` in terms of the arrow operations (and verify that the monad laws hold).
Functional Reactive Programming (FRP):

- Paradigm for *reactive programming* in a functional setting:
  - Input arrives *incrementally* while system is running.
  - Output is generated in response to input in an interleaved and *timely* fashion.
Functional Reactive Programming (FRP):

- Paradigm for *reactive programming* in a functional setting:
  - Input arrives *incrementally* while system is running.
  - Output is generated in response to input in an interleaved and *timely* fashion.

- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
An application: FRP

Functional Reactive Programming (FRP):

- Paradigm for *reactive programming* in a functional setting:
  - Input arrives *incrementally* while system is running.
  - Output is generated in response to input in an interleaved and *timely* fashion.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.
Yampa

Yampa:

- The most recent Yale FRP implementation.
Yampa

*Yampa*:  
- The most recent Yale FRP implementation.  
- *Embedding* in Haskell (a Haskell library).
Yampa:

- The most recent Yale FRP implementation.
- *Embedding* in Haskell (a Haskell library).
- *Arrows* used as the basic structuring framework.
Yampa

**Yampa:**

- The most recent Yale FRP implementation.
- *Embedding* in Haskell (a Haskell library).
- *Arrows* used as the basic structuring framework.
- *Continuous time.*
Yampa

**Yampa:**

- The most recent Yale FRP implementation.
- *Embedding* in Haskell (a Haskell library).
- *Arrows* used as the basic structuring framework.
- *Continuous time*.
- Discrete-time signals modelled by continuous-time signals and an option type.
Yampa:

- The most recent Yale FRP implementation.
- *Embedding* in Haskell (a Haskell library).
- *Arrows* used as the basic structuring framework.
- *Continuous time*.
- Discrete-time signals modelled by continuous-time signals and an option type.
- Advanced *switching constructs* allows for highly dynamic system structure.
Related languages

FRP related to:

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

FRP related to:

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

Distinguishing features of FRP:

- First class reactive components.
- Allows highly dynamic system structure.
- Supports hybrid (mixed continuous and discrete) systems.
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)
Yampa?
Yampa?

Yet
Another
Mostly
Pointless
Acronym
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!
Key concept: *functions on signals*.
Key concept: *functions on signals*.

**Intuition:**

\[
\begin{align*}
\text{Signal } \alpha &\approx \text{Time} \rightarrow \alpha \\
x &:: \text{Signal} \ T1 \\
y &:: \text{Signal} \ T2 \\
\text{SF } \alpha \ \beta &\approx \text{Signal} \ \alpha \ \rightarrow \text{Signal} \ \beta \\
f &:: \text{SF} \ T1 \ T2
\end{align*}
\]
Signal functions

Key concept: *functions on signals*.

Intuition:

\[
x : \text{Signal } T_1 \\
y : \text{Signal } T_2 \\
f : \text{SF } T_1 \rightarrow T_2
\]

Additionally: *causality* requirement.
Signal functions and state

Alternative view:

Functions on signals are either:

- Stateful: \( y(t) \) depends on \( x(t) \) and state \( x(t_0) \), \( t_0 \leq t \leq t \).
- Stateless: \( y(t) \) depends only on \( x(t) \).
Signal functions and state

Alternative view:

Signal functions can encapsulate **state**.

\[ \text{state}(t) \text{ summarizes input history } x(t'), t' \in [0, t]. \]
Signal functions and state

Alternative view:

Signal functions can encapsulate state.

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

Functions on signals are either:

- **Stateful**: \( y(t) \) depends on \( x(t) \) and \( \text{state}(t) \)
- **Stateless**: \( y(t) \) depends only on \( x(t) \)
Yampa and Arrows

SF is an arrow. Signal function instances of core combinators:

- \( \text{arr} :: (a \to b) \to SF\ a\ b \)
- \( \text{>>>} :: SF\ a\ b \to SF\ b\ c \to SF\ a\ c \)
- \( \text{first} :: SF\ a\ b \to SF\ (a,c)\ (b,c) \)
- \( \text{loop} :: SF\ (a,c)\ (b,c) \to SF\ a\ b \)

But \texttt{apply} has no useful meaning. Hence \( SF \) is \textit{not} a monad.
Some further basic signal functions

- \texttt{identity} :: \texttt{SF} a a
  \[ \texttt{identity} = \texttt{arr id} \]
Some further basic signal functions

- `identity :: SF a a`
  
  `identity = arr id`

- `constant :: b -> SF a b`
  
  `constant b = arr (const b)`
Some further basic signal functions

- `identity :: SF a a`
  
  \[
  \text{identity} = \text{arr id}
  \]

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

- `integral :: VectorSpace a s=>SF a a`
  
  \[
  \text{time} = \text{constant} \ 1.0 \ggg \text{integral}
  \]

- `(^<<) :: (b->c) -> SF a b -> SF a c`
  
  \[
  f \ (^<<) \ \text{sf} = \text{sf} \ggg \text{arr} \ f
  \]
Some further basic signal functions

- `identity :: SF a a`
  `identity = arr id`

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

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

- `time :: SF a Time`
  `time = constant 1.0 >>> integral`
Some further basic signal functions

- **identity :: SF a a**
  \[
  \text{identity} = \text{arr id}
  \]

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

- **integral :: VectorSpace a s=>SF a a**

- **time :: SF a Time**
  \[
  \text{time} = \text{constant}\ 1.0 >>> \text{integral}
  \]

- **(^<<) :: (b->c) -> SF a b -> SF a c**
  \[
  f\ (^<<)\ \text{sf} = \text{sf} >>> \text{arr f}
  \]
Example: A bouncing ball

\[ y = y_0 + \int v \, dt \]

\[ v = v_0 + \int -9.81 \]

On impact:

\[ v = -v(t-) \]

(fully elastic collision)
Free-falling ball:

```hs
import Data.Semigroup (<<)

type Pos = Double

type Vel = Double

fallingBall :: Pos -> Vel -> SF () (Pos, Vel)
fallingBall y0 v0 = proc () -> do
  v <- (v0 +) ^<< integral 9.81
  y <- (y0 +) ^<< integral v
  returnA <- (y, v)
```
Dynamic system structure

**Switching** allows the structure of the system to evolve over time:
Example: Space Invaders
Overall game structure

dpSwitch

route

ObjInput

ObjOutput

alien

gun

alien

bullet

killOrSpawn

[ObjectOutput]
Reading

Reading (2)
