Arrows (1)
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:

![Block diagram]

A combinator can be defined that captures this idea:

\[
(\ggg) :: B \ a \ b \rightarrow B \ b \ c \rightarrow B \ a \ c
\]

Arrows (2)
But systems can be complex:

These diagrams convey the general idea:

![Block diagram]

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).
- 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.

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)
```

What is an arrow? (1)
• A type constructor \( a \) of arity two.
• Three operators:
  - lifting:
    \[
    \text{arr} :: (b \rightarrow c) \rightarrow a \ b \ c
    \]
  - composition:
    \[
    (\ggg) :: a \ b \ c \rightarrow a \ c \ d \rightarrow a \ b \ d
    \]
  - widening:
    \[
    \text{first} :: a \ b \ c \rightarrow a \ (b,d) \ (c,d)
    \]
• A set of algebraic laws that must hold.

What is an arrow? (2)
These diagrams convey the general idea:

![Block diagram]

Functions are arrows (1)
Functions are a simple example of arrows, with \((\rightarrow)\) as the arrow type constructor.

Exercise 1: Suggest suitable definitions of

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

for this case!

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

Functions are arrows (2)
Solution:

- \( \text{arr} = \text{id} \)
  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)

- \( f \gg g = \lambda a \rightarrow g (f \ a) \) or
- \( f \gg g = g . f \) or even
- \( >>> = \overline{flip \ (.)} \)
- \( \overline{first \ f = \lambda (b,d) \rightarrow (f \ b,d)} \)

Functions are arrows (4)

Arrow instance declaration for functions:

\[
\begin{align*}
\text{instance Arrow (->) where} \\
\quad \text{arr} & = \text{id} \\
\quad (>>>) & = \overline{flip \ (.)} \\
\quad \overline{first \ f = \lambda (b,d) \rightarrow (f \ b,d)}
\end{align*}
\]

Some arrow laws

\[
\begin{align*}
(f >>> g) >>> h & = f >>> (g >>> h) \\
\overline{arr \ (f >>> g)} & = arr \ f >>> arr \ g \\
\overline{arr \ id >>> f} & = f \\
\overline{first \ (arr \ f)} & = arr \ (\overline{first \ f}) \\
\overline{first \ (f >>> g)} & = \overline{first \ f >>> first \ g}
\end{align*}
\]

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

The loop combinator (1)

Another important operator is \( \text{loop} \): a fixed-point operator used to express recursive arrows or feedback:

\[ \text{loop} \ f \]

The loop combinator (2)

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

\[
\text{class Arrow a => ArrowLoop a where} \\
\quad \text{loop :: a (b,d) (c,d) -> a b c}
\]

Remarkably, the four combinators \( \text{arr}, \gggg, \overline{first}, \) and \( \text{loop} \) are sufficient to express any conceivable wiring!

Some more arrow combinators (1)

\[ \overline{second :: Arrow a => a b c -> a (d,b) (d,c)} \]
\[ (***) :: Arrow a => a b c -> a d e -> a (b,d) (c,e) \]
\[ (&&&) :: Arrow a => a b c -> a b d -> a b (c,d) \]

Some more arrow combinators (2)

\[ \overline{second f = \text{arr swap} >>> \overline{first f} >>> \text{arr swap}} \]
\[ \overline{swap \ (x,y) = \ (y,x)} \]
\[ (***) :: Arrow a => a b c -> a d e -> a (b,d) (c,e) \]
\[ f *** g = \overline{first f} >>> \overline{second g} \]
\[ (&&&) :: Arrow a => a b c -> a b d -> a b (c,d) \]
\[ f &&& g = \text{arr} \ (\lambda x \rightarrow (x,x)) >>> \overline{f *** g} \]

Exercise 3

Describe the following circuit using arrow combinators:

\[
\begin{array}{c}
\text{a1} \\
\text{a2} \\
\text{a3}
\end{array}
\]

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

Describe the following circuit using arrow combinators:

```
(a1 &&& arr id) >>> (a2 *** a3) >>> arr (uncurry (+))
```

Exercise 3: Another solution

Describe the following circuit:

```
(a1 &&& arr id) >>> (a2 *** a3) >>> arr (uncurry (+))
```

Note on the definition of (***) (1)

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

\[ f *** g = \text{first } f \implies \text{second } g \]
\[ f *** g = \text{second } g \implies \text{first } f \]

No, in general

since the order of \( f \) and \( g \) are different.

Exercise 4: Are \( \text{circuit\_v1} \), \( \text{circuit\_v2} \), and \( \text{circuit\_v3} \) all equivalent?

Yet an attempt at exercise 3

```
(a1 &&& a3) >>> first a2 >>> arr (uncurry (+))
```

The arrow do notation (1)

Ross Paterson's do-notation for arrows supports pointed arrow programming. Only syntactic sugar.

```
proc pat -> do [rec]
  pat_1 <- sfexp_1 <<< exp
  pat_2 <- sfexp_2 <<< exp
  ...
  pat_n <- sfexp_n <<< exp
  returnA <<< exp
```

Also:

```
let pat = exp  in  pat <- arr id <<< exp
```

The arrow do notation (2)

Let us redo exercise 3 using this notation:

```
(a1 &&& a3) >>> (a2 *** a3) >>> arr (uncurry (+))
```

The arrow do notation (3)

We can also mix and match:

```
(a1 &&& a3) >>> first a2 >>> arr (uncurry (+))
```

The arrow do notation (4)

Recursive networks: do-notation:

```
(a1 &&& a3) >>> (a2 *** a3) >>> arr (uncurry (+))
```

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:

```
newtype Kleisli m a b = K (a -> m b)

instance Monad m => Arrow (Kleisli m) where
  arr f = K (\b -> return (f b))
  K f >>> K g = K (\b -> f b >>= g)
```

Exercise 6: Verify that

```
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).

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.
- Advanced **switching constructs** allows for highly dynamic system structure.

Yampa?

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

A good metaphor for hybrid systems!

Related languages

FRP related to:

- Synchronous languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink.
- Modeling languages, like Simulink.
- Distinguishing features of FRP:
  - First class reactive components.
  - Allows highly dynamic system structure.
  - Supports hybrid (mixed continuous and discrete) systems.

Signal functions

Key concept: **functions on signals**.

```
x :: Signal T1
y :: Signal T2
f :: SF α β = Signal α -> Signal β
```

Intuition:

- `Signal α ≈ Time → α`
- `x :: Signal T1`
- `y :: Signal T2`
- `f :: SF T1 T2`

Additionally: **causality** requirement.

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)

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

A good metaphor for hybrid systems!
Signal functions and state

Alternative view:
Signal functions can encapsulate state. 

\[ state(t) \text{ summarizes input history } x(t'), t' \in [0, t]. \]
Functions on signals are either:
- **Stateful**: \( y(t) \) depends on \( x(t) \) and \( state(t) \)
- **Stateless**: \( y(t) \) depends only on \( x(t) \)

Example: A bouncing ball

\[ y = y_0 + \int v \, dt \]
\[ v = v_0 + \int -9.81 \]

On impact:
\[ v = -v(t^-) \]
(fully elastic collision)

Example: Space Invaders

Yampa and Arrows

SF is an arrow. Signal function instances of core combinators:
- \( arr :: (a \rightarrow b) \rightarrow SF a b \)
- \( >>> :: SF a b \rightarrow SF b c \rightarrow SF a c \)
- \( first :: SF a b \rightarrow SF (a,c) (b,c) \)
- \( loop :: SF (a,c) (b,c) \rightarrow SF a b \)

But apply has no useful meaning. Hence SF is not a monad.

Some further basic signal functions

- \( identity :: SF a a \)
- \( constant :: b \rightarrow SF a b \)
- \( integral :: VectorSpace a \rightarrow SF a a \)
- \( time :: SF a \text{ Time} \)
- \( (^<<) :: (b->c) \rightarrow SF a b \rightarrow SF a c \)

MGS 2007: ADV Lecture 3 – p.37/46

Part of a model of the bouncing ball

Free-falling ball:

\[ \text{type Pos = Double} \]
\[ \text{type Vel = Double} \]

\[ \text{fallingBall :: Pos -> Vel -> SF () (Pos, Vel)} \]
\[ \text{fallingBall y0 v0 = proc () \rightarrow do} \]
\[ \text{v <- (v0 +) (^<< integral) \(-9.81\)} \]
\[ \text{y <- (y0 +) (^<< integral) \(-\)} \]
\[ \text{returnA \(-\) (y, v)} \]

Overall game structure

Dynamic system structure

Switching allows the structure of the system to evolve over time:

Example: Space Invaders

Overall game structure

Reading
