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:

\[
\begin{array}{c}
\text{f} \rightarrow \text{g}
\end{array}
\]

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

\[
(\gg\gg) :: \text{B} \ a \ b \rightarrow \text{B} \ b \ c \rightarrow \text{B} \ a \ c
\]

But systems can be complex:

\[
\text{How many and what combinators do we need to be able to describe arbitrary systems?}
\]

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** `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.

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? (2)

These diagrams convey the general idea:

```
arr f
f >>> g
```

Functions are arrows (1)

Functions are a simple example of arrows, with `(-->)` as the arrow type constructor.

**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
  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) \textbf{or}
- f >>> g = g . f \textbf{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)
```

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!
The loop combinator (1)

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

\[ \text{loop } f \]

Some more arrow combinators (1)

\[
\begin{align*}
\text{second} :: & \text{ Arrow } a \Rightarrow \\
& a \ b \ c \Rightarrow a \ (d, b) \ (d, c) \\
\text{(***)} :: & \text{ Arrow } a \Rightarrow \\
& a \ b \ c \Rightarrow a \ d \ e \Rightarrow a \ (b, d) \ (c, e) \\
\text{(&&)} :: & \text{ Arrow } a \Rightarrow \\
& a \ b \ c \Rightarrow a \ b \ d \Rightarrow a \ b \ (c, d)
\end{align*}
\]

The loop combinator (2)

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

\[
\text{class Arrow } a \Rightarrow \text{ ArrowLoop } a \text{ where} \\
\text{loop} :: a \ (b, d) \ (c, d) \Rightarrow a \ b \ c
\]

Remarkably, the four combinators \texttt{arr}, \texttt{>>>}, \texttt{first}, and \texttt{loop} are sufficient to express any conceivable wiring!

Some more arrow combinators (2)

As diagrams:

\[
\begin{align*}
\text{second } f \\
\text{f *** g}
\end{align*}
\]

\[
\begin{align*}
\text{f & & g}
\end{align*}
\]
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:

\[ a1 \rightarrow a2 \rightarrow a3 \]

Exercise 3: One solution

Exercise 3: Describe the following circuit using arrow combinators:

\[ a1 \rightarrow a2 \rightarrow a3 \]

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

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

\[ \text{circuit}_v1 = (a1 &&& arr \text{id}) \]

\[ \rightarrow (a2 *** a3) \]

\[ \rightarrow arr (\text{uncurry (+)}) \]

Exercise 3: Another solution

Exercise 3: Describe the following circuit:

\[ a1 \rightarrow a2 \rightarrow a3 \]

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

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

\[ \text{circuit}_v2 = arr (\text{x} -> (\text{x},\text{x})) \]

\[ \rightarrow \text{first a1} \]

\[ \rightarrow (a2 *** a3) \]

\[ \rightarrow arr (\text{uncurry (+)}) \]
Note on the definition of (***)(1)

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

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

No, in general

\( \text{first } f \gg \gg \text{second } g \neq \text{second } g \gg \gg \text{first } f \)

since the order of the two possibly effectful computations \( f \) and \( g \) are different.

Yet an attempt at exercise 3

\[
\begin{array}{c}
  \text{circuit}_v3 :: \text{A } \text{Double Double} \\
  \text{circuit}_v3 = (a1 \&\& a3) \\
  \quad \gg \gg \text{first } a2 \\
  \quad \gg \gg \text{arr (uncurry (+))}
\end{array}
\]

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

Note on the definition of (***)(2)

Similarly

\( (f *** g) \gg \gg (h *** k) \neq (f \gg \gg h) *** (g \gg \gg k) \)

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

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

\[
\begin{align*}
\text{first } f \gg \gg \text{second (arr } g) \\
= \text{second (arr } g) \gg \gg \text{first } f
\end{align*}
\]

However, for certain arrow instances equalites like the ones above do hold.

The arrow do notation (1)

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

\[
\begin{align*}
\text{proc } \text{pat} & \to \text{do [ rec]}
\quad \text{pat}_1 \leftarrow \text{sfexp}_1 \leftarrow \text{exp}_1 \\
\quad \text{pat}_2 \leftarrow \text{sfexp}_2 \leftarrow \text{exp}_2 \\
\ldots
\quad \text{pat}_n \leftarrow \text{sfexp}_n \leftarrow \text{exp}_n \\
\text{returnA} \leftarrow \text{exp}
\end{align*}
\]

Also: \( \text{let } \text{pat} = \text{exp} \equiv \text{pat} \leftarrow \text{arr id} \leftarrow \text{exp} \)
The arrow do notation (2)

Let us redo exercise 3 using this notation:

```
circuit_v4 :: A Double Double
  circuit_v4 = proc x -> do
    y1 <- a1 -< x
    y2 <- a2 -< y1
    y3 <- a3 -< x
    returnA -< y2 + y3
```

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
```

The arrow do notation (4)

Recursive networks: do-notation:

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

Exercise 5: Describe this using only the arrow combinators.
Arrows and Monads (1)

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

```haskell
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
```

Arrows and Monads (2)

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

```haskell
apply :: Arrow a => a (a b c, b) c
```

Exercise 6: Verify that

```haskell
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).

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.
- **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.

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

A good metaphor for hybrid systems!

Signal functions

Key concept: functions on signals.

Intuition:
- $\text{Signal } \alpha \approx \text{Time} \to \alpha$
- $x :: \text{Signal } T1$
- $y :: \text{Signal } T2$
- $SF \alpha \beta \approx \text{Signal } \alpha \to \text{Signal } \beta$
- $f :: SF T1 T2$

Additionally: causality requirement.
Signal functions and state

Alternative view:

Signal functions can encapsulate \textit{state}.

\[ x(t) \xrightarrow{f} y(t) \]

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

Functions on signals are either:

- \textbf{Stateful}: \(y(t)\) depends on \(x(t)\) and \textit{state}(t)
- \textbf{Stateless}: \(y(t)\) depends only on \(x(t)\)

Some further basic signal functions

- \textbf{identity} :: SF a a
  \[ \text{identity} = \text{arr } \text{id} \]
- \textbf{constant} :: b \rightarrow SF a b
  \[ \text{constant } b = \text{arr } (\text{const } b) \]
- \textbf{integral} :: \text{VectorSpace a s} \Rightarrow SF a a
- \textbf{time} :: SF a Time
  \[ \text{time} = \text{constant } 1.0 \ggg \text{integral} \]
- \textbf{(}^\ll\text{)} :: (b\rightarrow c) \rightarrow SF a b \rightarrow SF a c
  \[ f (^\ll) \text{ sf} = \text{sf} \ggg \text{arr } f \]

Yampa and Arrows

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

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

But apply has no useful meaning. Hence SF is \textbf{not} a monad.

Example: A bouncing ball

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

On impact:
\[ v = -v(t-) \]

(fully elastic collision)
Part of a model of the bouncing ball

Free-falling ball:

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)

Example: Space Invaders

Dynamic system structure

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

Overall game structure
Reading


Reading (2)