Recap: The arrow framework (1)

The following two Haskell type classes capture the notion of an arrow and of an arrow supporting feedback:

```
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)

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

Recap: The arrow framework (2)

arr \( f \), \( f >>> g \), first \( f \), and loop \( f \) are sufficient to express any conceivable "wiring"!
Recap: Further 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)

Exercise 3: One solution

**Exercise 3:** Describe the following circuit using arrow combinators:

```
 a1      a2
  |      |
  +----+----+
  |    |    |
  |    v    |
  +--------+
```

a1, a2, a3 :: A Double Double

circuit_v1 :: A Double Double

circuit_v1 = (a1 &&& arr id)
  >>> (a2 *** a3)
  >>> arr (uncurry (+))

Exercise 3: Another solution

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

```
 a1     a2     a3
  |      |      |
  |      |      |
  |      +----+
  |         |    |
  +---------+----+
```

a1, a2, a3 :: A Double Double

circuit_v2 :: A Double Double

circuit_v2 = arr (\x -> (x,x))
  >>> first a1
  >>> (a2 *** a3)
  >>> arr (uncurry (+))
Exercise 4: Solution

Exercise 4: Suggest definitions of second, (***) and (&&&).

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)

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.

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

Similarly

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

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

However, the following is true

(an additional arrow law):

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

Yet an attempt at exercise 3

\[
\begin{align*}
circuit_v3 :: & \text{ A Double Double} \\
\text{circuit_v3} &= (a1 &\& a3) \\
& >>> \text{first } a2 \\
& >>> \text{arr } (\text{uncurry } (+))
\end{align*}
\]

Are circuit_v1, circuit_v2, and circuit_v3 all equivalent?
Point-free vs. pointed programming

What we have seen thus far is an example of **point-free** programming: the values being manipulated are not given any names.

This is often appropriate, especially for small definitions, and it facilitates equational reasoning as shown by Bird & Meertens (Bird 1990).

However, large programs are much better expressed in a **pointed** style, where names can be given to values being manipulated.

The arrow do notation (1)

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

```haskell
proc pat -> do [ rec ]
  pat_1 <- sfexp_1 <<< exp_1
  pat_2 <- sfexp_2 <<< exp_2
  ...
  pat_n <- sfexp_n <<< exp_n
returnA <<< exp
```

Also: let pat = exp \equiv pat <- arr id <<< exp

The arrow do notation (2)

Let us redo exercise 3 using this notation:

```haskell
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:

```haskell
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)

Exercise 5: Describe the following circuit using the arrow do-notation:

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

Exercise 6: As 5, but directly using only the arrow combinators.

Solution exercise 5

\[
\text{circuit = proc x \rightarrow do}
\]
\[\text{rec}
\]
\[\text{y1 <- a1 -< x}
\]
\[\text{y2 <- a2 -< y1}
\]
\[\text{y3 <- a3 -< (x, y)}
\]
\[\text{let y = y2 + y3}
\]
\[\text{returnA -< y}
\]

Some More Reading


Recap: Signal functions (1)

Key concept: functions on signals.

Intuition:

\[\text{Signal } \alpha \approx \text{Time} \rightarrow \alpha\]
\[\text{SF } \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta\]
\[x :: \text{Signal T1}\]
\[y :: \text{Signal T2}\]
\[f :: \text{SF T1 T2}\]

\text{SF is an instance of Arrow and ArrowLoop.}
Recap: 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 $[0, t]$.

Some basic signal functions (2)

- *iPre* :: $a \rightarrow SF\ a\ a$
- $(^<<) :: (b\rightarrow)c \rightarrow SF\ a\ b \rightarrow SF\ a\ c$
- $f\ (^<<)\ sf = sf\ >>>\ arr\ f$
- *time* :: $SF\ a\ Time$

Quick Exercise: Define *time*!

\[ time = constant\ 1.0\ >>>\ integral \]

Some basic signal functions (1)

- **identity** :: $SF\ a\ a$
  \[
  identity = arr\ id 
  \]
- **constant** :: $b \rightarrow SF\ a\ b$
  \[
  constant\ b = arr\ (const\ b) 
  \]
- **integral** :: $VectorSpace\ a\ s\Rightarrow SF\ a\ a$

It is defined through:

\[
y(t) = \int_{0}^{t} x(\tau)\ d\tau
\]

A bouncing ball

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

On impact:

\[
v = -v(t^-)\] (fully elastic collision)
Modelling the bouncing ball: part 1

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)

Some basic event sources

• never :: SF a (Event b)
• now :: b -> SF a (Event b)
• after :: Time -> b -> SF a (Event b)
• repeatedly ::
    Time -> b -> SF a (Event b)
• edge :: SF Bool (Event ())

Events

Conceptually, **discrete-time** signals are only defined at discrete points in time, often associated with the occurrence of some **event**.

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

data Event a = NoEvent | Event a

**Discrete-time signal** = Signal (Event a).

Associating information with an event occurrence:

tag :: Event a -> b -> Event b

Stateful event suppression

• notYet :: SF (Event a) (Event a)
• once :: SF (Event a) (Event a)
### Modelling the bouncing ball: part 2

Detecting when the ball goes through the floor:

```haskell
fallingBall' ::
    Pos -> Vel
    -> SF () ((Pos, Vel), Event (Pos, Vel))
fallingBall' y0 v0 = proc () -> do
    yv@(y, _) <- fallingBall y0 v0 -< ()
    hit <- edge -< y <= 0
    returnA -< (yv, hit `tag` yv)
```

### Switching

**Q:** How and when do signal functions “start”?

**A:**

- **Switchers** “apply” a signal function to its input signal at some point in time.
- This creates a “running” signal function instance.
- The new signal function instance often replaces the previously running instance.

Switchers thus allow systems with **varying structure** to be described.

### The basic switch (1)

**Idea:**

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

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

### The basic switch (2)

**Exercise 7:** Define an event counter `countFrom`

```haskell
countFrom ::
    Int -> SF (Event a) Int
using

    switch :: SF a (b, Event c)
            -> (c -> SF a b)
            -> SF a b
    constant :: b -> SF a b
    tag :: Event a -> b -> Event b
```
Solution exercise 7

countFrom :: Int -> SF (Event a) Int
countFrom n = 
  switch 
    (constant n 
      &&& arr (\e -> e `tag` (n+1)))
countFrom

Simulating the bouncing ball

Making the ball bounce:

bouncingBall :: Pos -> SF () (Pos, Vel)
bouncingBall y0 = bbAux y0 0.0
  where
    bbAux y0 v0 =
      switch (fallingBall' y0 v0) $ \(y,v) ->
        bbAux y (-v)

Modelling using impulses

From a modelling perspective, using a device like
switch to model the interaction between the ball
and the floor may seem rather unnatural.

A more appropriate account of what is going on
is that an impulsive force is acting on the ball for
a short time.

This can be abstracted into Dirac Impulses:
impulses that act instantaneously. See

Henrik Nilsson. Functional Automatic
Differentiation with Dirac Impulses. In