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 >>> g
first f
loop f
```

arr,>>, first, and loop 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 d e -> a (b,d) (c,e)
(&&& :: Arrow a => a b c -> a b d -> a b (c,d)
```

Recap: Further arrow combinators (2)

As diagrams:

```
second f
f *** g
```

Exercise 3: One solution

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

a1, a2, a3 :: A Double Double

a1, a2, a3 :: A Double Double

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

Exercise 3: Another solution

```
Exercise 3: Describe the following circuit:

a1, a2, a3 :: A Double Double

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

Exercise 4: Suggestion 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 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 (uncurry (+))
```
Note on the definition of (***)(1)

Are the following two definitions of (***)
equivalent?
- \( f \circledast g = \text{first } f \circledcirc \text{second } g \)
- \( f \circledast g = \text{second } g \circledcirc \text{first } f \)

No, in general
\( \text{first } f \circledcirc \text{second } g \neq \text{second } g \circledcirc \text{first } f \)
since the order of the two possibly effectful computations \( f \) and \( g \) are different.

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.

\[
\text{proc pat} \to \text{do } [\text{rec}]
\]

\[
\begin{align*}
\text{pat}_1 & \leftarrow \text{sfexp}_1 -< \text{exp}_1 \\
\text{pat}_2 & \leftarrow \text{sfexp}_2 -< \text{exp}_2 \\
\ldots & \leftarrow \text{sfexp}_n -< \text{exp}_n \\
\text{returnA} & \leftarrow \text{exp}
\end{align*}
\]

Also: \( \text{let } \text{pat} = \text{exp} \equiv \text{pat} \leftarrow \text{id} -< \text{exp} \)

The arrow do notation (2)

Let us redo exercise 3 using this notation:

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

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

\[
\begin{align*}
y_1 & \leftarrow a_1 -< x \\
y_2 & \leftarrow a_2 -< y_1 \\
y_3 & \leftarrow a_3 -< x \\
\text{returnA} & \leftarrow y_2 + y_3
\end{align*}
\]

The arrow do notation (3)

We can also mix and match:

\[
\begin{array}{c}
\text{circuit}_v5 :: \text{A Double Double} \\
\text{circuit}_v5 = \text{proc } x \to \text{do}
\end{array}
\]

\[
\begin{align*}
y_2 & \leftarrow a_2 <<< a_1 -< x \\
y_3 & \leftarrow a_3 -< x \\
\text{returnA} & \leftarrow y_2 + y_3
\end{align*}
\]

The arrow do notation (4)

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

\[
\begin{array}{c}
\text{circuit} = \text{proc } x \to \text{do}
\end{array}
\]

\[
\begin{align*}
y_1 & \leftarrow a_1 -< x \\
y_2 & \leftarrow a_2 -< y_1 \\
y_3 & \leftarrow a_3 -< (x, y) \\
\text{let } y & = y_2 + y_3 \\
\text{returnA} & \leftarrow y
\end{align*}
\]

Solution exercise 5

\[
\begin{array}{c}
\text{circuit}_v5 :: \text{A Double Double} \\
\text{circuit}_v5 = \text{proc } x \to \text{do}
\end{array}
\]

\[
\begin{align*}
y_2 & \leftarrow a_2 <<< a_1 -< x \\
y_3 & \leftarrow a_3 -< y_2 + y_3
\end{align*}
\]
Some More Reading


Recap: Signal functions (1)

Key concept: **functions on signals.**

Intuition:

\[ \text{Signal } \alpha \approx \text{Time} \rightarrow \alpha \]

\[ \text{SF } \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta \]

\[ x :: \text{Signal } T1 \]

\[ y :: \text{Signal } T2 \]

\[ f :: \text{SF } T1 \rightarrow T2 \]

**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 (1)

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

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

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

  It is defined through:

  \[ y(t) = \int_0^t x(\tau) d\tau \]

Some basic signal functions (2)

- **iPre :: a -> SF a a**

- \((^<<) :: (b->c) -> SF a b -> SF a c\)

  \[ f (^<<) \text{sf} = \text{sf} >>> \text{arr } f \]

- **time :: SF a Time**

Quick Exercise: Define \( \text{time} \):

\[ \text{time} = \text{constant } 1.0 >>> \text{integral} \]

A bouncing ball

\[ y = y_0 + \int_0^t v dt \]

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

On impact:

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

(fully elastic collision)

Modelling the bouncing ball: part 1

Free-falling ball:

```haskell```
```
free_falling Ball :: Pos -> Vel -> SF () (Pos, Vel)
free_falling Ball y0 v0 = proc () -> do
v <- (v0 +) (^<< integral) <- -9.81
y <- (y0 +) (^<< integral) <- v
returnA <- (y, v)
```
Stateful event suppression

- notYet :: SF (Event a) (Event a)
- once :: SF (Event a) (Event a)

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.

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

Modelling the bouncing ball: part 2

Detecting when the ball goes through the floor:

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)

The basic switch (2)

Exercise 7: Define an event counter countFrom

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

Modelling the bouncing ball: part 3

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)

Simulation of bouncing ball

Switching

Q: How and when do signal functions “start”?
A: Switchers “apply” a signal functions 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.

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