CFRP issues: Sharing

Consider:

let x = 1 + integral (x * x) in x

The recursively defined behavior, a *function*, is applied over and over to the *same* stream of sample times.

- Causes recomputation
- Laziness does *not* help
- Memoization needed to get acceptable performance. But with care to avoid memory leaks.

CFRP issues: Restart (1)

Consider:

let c = hold 0 (count (repeatedly 0.5)) in c 'until' after 5 --> c * 2

What happened at the time of the switch?

- CFRP behaviors and events are *signal generators*: they will start from scratch when switched in.
- But what if we just want to continue observing an evolving signal?

An alternative

By adopting *signal functions* as the central notion, these problems are side stepped:

- Sharing amounts to sharing computations of signal samples: lazy evaluation handles that just fine.
- Observation of externally originating signals is inherent in the notion of a signal function.
- Implementation is straightforward.

CFRP issues: Restart (2)

A version of *until* that starts new behaviors from time 0.

*Time and space leak!*

- Support signals as well, e.g. through some variant of *runningIn*:

```haskell
runningIn :: B a -> (B a -> B b) -> B b
```

Idea: apply behavior to start time once and for all, then wrap up the resulting signal as a signal generator that ignores its starting time.

Yampa

What is *Yampa*?

- FRP implementation structured using *arrows*.
- Realised as an *Embedded Domain-Specific Language* (EDSL), i.e. a combinator library.
- Continuous-time signals (conceptually)
- Discrete-time signals represented by continuous-time signal carrying option type *Event*.
- Functions on signals, *Signal Functions*, is the central abstraction, forming the arrows.

Yampa

- Signal functions are first-class entities, signals a secondary notion, only existing indirectly through the signal functions.
- Advanced *switching constructs* to describe systems with highly dynamic structure.
- People:
  - Antony Courtney
  - Paul Hudak
  - Henrik Nilsson
  - John 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 (1)

Key concept: functions on signals.

```
\( f : \text{SF } \alpha \beta \) \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta
```

Intuition:

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

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

\[ x :: \text{Signal } T1 \]
\[ y :: \text{Signal } T2 \]
\[ f :: \text{SF } T1 T2 \]

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 \([0, t]\).

Signal functions and state

Alternative view:

Signal functions can encapsulate state.

\[ x(t) \]

\[ f \left[ \text{state}(t) \right] \]

\[ x(t) \]

\( \text{state}(t) \) summarizes input history \( x(t'), t' \in [0, t] \).
Thus, really a kind of process.

From this perspective, signal functions are:
- stateful if \( y(t) \) depends on \( x(t) \) and \( \text{state}(t) \)
- stateless if \( y(t) \) depends only on \( x(t) \)

Yampa and arrows (1)

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:

\[ f \gg g \]

A combinator can be defined that captures this idea:

\[ (+++) :: \text{SF } a b \rightarrow \text{SF } b c \rightarrow \text{SF } a c \]

Yampa and arrows (2)

But systems can be complex:

How many and what combinators do we need to be able to describe arbitrary systems?

Yampa and arrows (3)

John Hughes’ arrow framework:
- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.
- Related to monads, since arrows are (effectful) computations, but more general: any monad \( m \) induces an arrow, the Kleisli arrow, \( \alpha \rightarrow m \beta \), but not vice versa.
- Provides a minimal set of “wiring” combinators.

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:
    \[ (+++) :: a \ b \ c \rightarrow a \ c \ d \rightarrow a \ b \ d \]
  - 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:

\[ a \rightarrow b \rightarrow c \rightarrow d \]

\[ f \gg g \]

\[ f \gg g \rightarrow h \]

\[ \text{first } f \]
The Arrow class

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

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)

Arrow laws

(f >>> g) >>> h = f >>> (g >>> h)
arr (g . f) = arr f >>> arr g
arr id >>> f = f
first (arr f) = arr (f x id)
first (f >>> g) = first f >>> first g
first f >>> arr (id x g) = arr (id x g) >>> first f
first f >>> arr (f x g) = arr (f x g) >>> first f
first (arr f) >>> arr assoc = arr assoc >>> first f

Functions are arrows (1)

Functions are a simple example of arrows. The arrow type constructor is just (->) in that case.

Exercise 1: Suggest suitable definitions of
  • arr
  • (>>>)
  • first
  for this case!

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) or
  • f >>> g = g . f 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)

The arrow laws reformulated

Exploiting that functions are arrows, some of the laws can be formulated more neatly. E.g:

arr (f >>> g) = arr f >>> arr g
first (arr f) = arr (first f)

The loop combinator (1)

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

The loop combinator (2)

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

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

Remarkably, the four combinators arr, (>>>, first, and 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:

Exercise 2: Describe the following circuit using arrow combinators:

```
  a1 -> a2 -> a3
```

Exercise 3: The combinators `second`, `(***)`, and `(&&&)` are not primitive, but defined in terms of `arr`, `(>>>)`, and `first`. Suggest suitable definitions!

Exercise 2: One solution

```
circuit_v1 :: A Double Double
circuit_v1 = (a1 &&& arr id) >>> (a2 *** a3) >>> arr (uncurry (+))
```

Exercise 2: Another solution

```
circuit_v2 :: A Double Double
circuit_v2 = arr (\x -> (x,x)) >>> first a1 >>> (a2 *** a3) >>> arr (uncurry (+))
```

Exercise 3: Solution

```
  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)
```

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

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

- `f *** g = first f >>> second g`
- `f *** g = second g >>> first f`

No, in general,

- `first f >>> second g ≠ second g >>> 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) ≠ (f >>> h) *** (g >>> k)`

since the order of `f` and `g` differs.

However, Yampa's signal functions have no effectful interaction: they are Causal Commutative Arrows (Liu, Cheng, Hudak 2009)

Both considered identities actually hold.

Yet another attempt at exercise 2

```
circuit_v3 :: A Double Double
circuit_v3 = (a1 &&& a3) >>> first a2 >>> arr (uncurry (+))
```
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.

\[
\text{proc pat -> do [ rec]}
\]

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

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

The arrow do notation (2)

Let us redo exercise 3 using this notation:

\[
\begin{align*}
\text{circuit_v4 :: A Double Double} \\
\text{circuit_v4 = proc x -> do} \\
& y_1 \leftarrow a_1 \leftarrow x \\
& y_2 \leftarrow a_2 \leftarrow y_1 \\
& y_3 \leftarrow a_3 \leftarrow x \\
& \text{returnA} \leftarrow y_2 + y_3
\end{align*}
\]

The arrow do notation (3)

We can also mix and match:

\[
\begin{align*}
\text{circuit_v5 :: A Double Double} \\
\text{circuit_v5 = proc x -> do} \\
& y_2 \leftarrow a_2 \leftarrow a_1 \leftarrow x \\
& y_3 \leftarrow a_3 \leftarrow x \\
& \text{returnA} \leftarrow y_2 + y_3
\end{align*}
\]

The arrow do notation (4)

Exercise 4: Describe the following circuit using the arrow do-notation:

\[
\begin{align*}
\text{a1, a2 :: A Double Double} \\
\text{a3 :: A (Double,Double) Double} \\
\text{Exercise 5: As 4, but directly using only the arrow combinators.}
\end{align*}
\]

Some basic signal functions (1)

• \( \text{id} \)entity :: SF a a
  \( \text{id} \)entity = arr id
• constant :: b -> SF a b
  constant b = arr (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)

• \( \text{iPre} \) :: a -> SF a a
  \( \text{iPre} \) = \text{arr id}
• \( (^{<<}) \) :: (b->c) -> SF a b -> SF a c
  \( f (^{<<}) \) sf = sf >> arr f
• \( \text{time} \) :: SF a Time
  \( \text{time} \) = constant 1.0 >> integral

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

\( \text{time} = \text{constant 1.0} \ggg \text{integral} \)

A bouncing ball

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

\[
v = v_0 + \int_{-9.81} \text{on impact:}
\]

Note: there is no built-in notion of global time in Yampa: time is always local, measured from when a signal function started.
Modelling the bouncing ball: part 1

Free-falling ball:

```haskell
type Pos = Double
type Vel = Double
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)
```

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:

```haskell
data Event a = NoEvent | Event a
```

Discrete-time signal = Signal (Event a).

Associating information with an event occurrence:

```haskell
tag :: Event a -> b -> Event b
```

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 (a))

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 6: Define an event counter `countFrom`

```haskell
countFrom :: Int -> SF (Event a) Int
countFrom n =
  switch
    (constant n &&& identity)
    (const (notYet >>> countFrom (n+1)))
```

Solution exercise 6
Solution exercise 6

Another version that ignores any event at time 0 also from the very start:

```haskell
countFrom :: Int -> SF (Event a) Int
countFrom n = switch
  (constant n &&& notYet)
  (const (countFrom (n+1)))
```

Modelling the bouncing ball: part 3

Making the ball bounce:

```haskell
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 (1)

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 *Proceedings of ICFP 2003*.

Modelling using impulses (2)

However, Yampa does provide a derived version of integral capturing the basic idea:

```haskell
impulseIntegral ::
  VectorSpace a k =>
  SF (a, Event a) a
```

The decoupled switch

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

- Output at the point of switch is taken from the old subordinate signal function, *not* the new residual signal function.
- This means the output at the current point in time is independent of whether or not the switching event occurs at that point in time. Hence decoupled.

The recurring switch

```haskell
rSwitch, drSwitch ::
  SF a b -> SF (a, Event (SF a b)) b
```

- Switching events received on the signal function input, carrying signal function to switch into.
- Switching occurs whenever an event occurs, not just once.
- Can be defined in terms of `switch/dSwitch`.

Reading (1)


Reading (2)

Reading (3)
