

Plugging a Space Leak with an Arrow



Paul Hudak and Paul Liu
Yale University
Department of Computer Science

July 2007
IFIP WG2.8
MiddleOfNowhere, Iceland

Background: FRP and Yampa

- *Functional Reactive Programming* (FRP) is based on two simple ideas:
 - *Continuous time-varying values*, and
 - *Discrete streams of events*.
- *Yampa* is an “arrowized” version of FRP.
- Besides foundational issues, we (and others) have applied FRP and Yampa to:
 - Animation and video games.
 - Robotics and other control applications.
 - Graphical user interfaces.
 - Models of biological cell development.
 - Music and signal processing.
 - Scripting parallel processes.

Behaviors in FRP



- Continuous behaviors capture any time-varying quantity, whether:
 - **input** (sonar, temperature, video, etc.),
 - **output** (actuator voltage, velocity vector, etc.), or
 - intermediate **values** internal to a program.
- Operations on behaviors include:
 - **Generic operations** such as arithmetic, integration, differentiation, and time-transformation.
 - **Domain-specific operations** such as edge-detection and filtering for vision, scaling and rotation for animation and graphics, etc.



Events in FRP

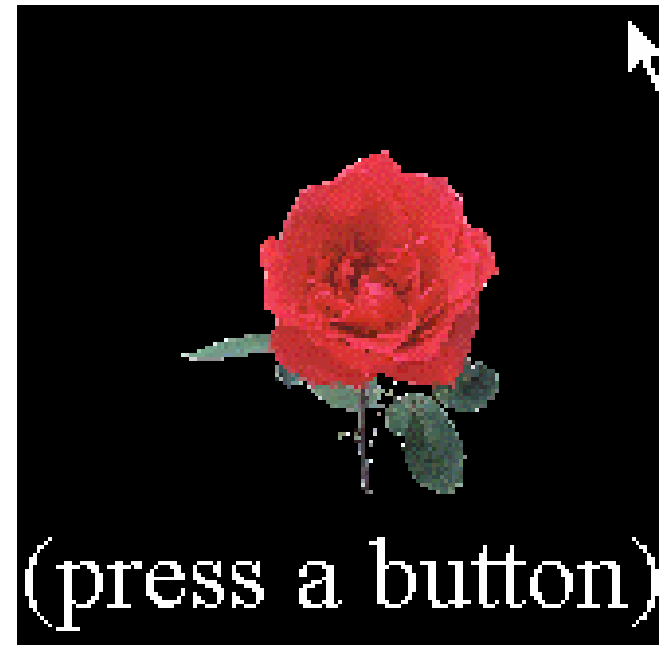
- **Discrete event streams** include user input as well as domain-specific sensors, asynchronous messages, interrupts, etc.
- They also include **tests for dynamic constraints** on behaviors (temperature too high, level too low, etc.)
- **Operations on event streams include:**
 - Mapping, filtering, reduction, etc.
 - Reactive behavior modification (next slide).

An Example from Graphics (Fran)

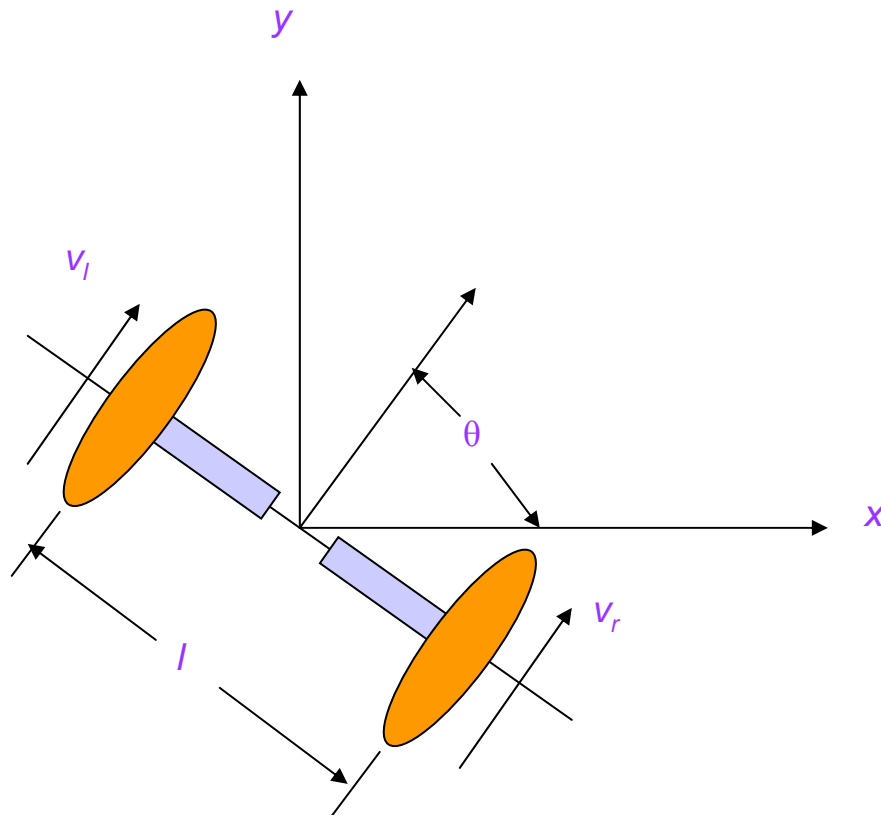
A single animation example that demonstrates key aspects of FRP:

```
growFlower = stretch size flower
  where size = 1 + integral bSign

bSign =
  0 `until`
  (lbp ==> -1 `until` lbr ==> bSign) .|.
  (rbp ==> 1 `until` rbr ==> bSign)
```



Differential Drive Mobile Robot



An Example from Robotics

- The equations governing the x position of a differential drive robot are:

$$x(t) = \frac{1}{2} \int_0^t (v_r(t) + v_l(t)) \cos(\theta(t)) dt$$

$$\theta(t) = \frac{1}{l} \int_0^t (v_r(t) - v_l(t)) dt$$

- The corresponding FRP code is:

```
x = (1/2) * (integral ((vr + vl) * cos theta))
theta = (1/l) * (integral (vr - vl))
```

(Note the lack of explicit time.)

Time and Space Leaks



- Behaviors in FRP are what we now call *signals*, whose (abstract) type is:

`Signal a = Time -> a`

- Unfortunately, unrestricted access to signals makes it far too easy to generate both *time* and *space leaks*.
- (Time leaks occur in real-time systems when a computation does not “keep up” with the current time, thus requiring “catching up” at a later time.)
- Fran, Frob, and FRP all suffered from this problem to some degree.

Solution: no signals!



- To minimize time and space leaks, *do not provide signals as first-class values*.
- Instead, provide *signal transformers*, or what we prefer to call *signal functions*:
$$\text{SF } a \ b = \text{Signal } a \ \rightarrow \ \text{Signal } b$$
- *SF is an abstract type*. Operations on it provide a *disciplined* way to compose signals.
- This also provides a more *modular* design.
- SF is an arrow – so we use *arrow combinators* to structure the composition of signal functions, and *domain-specific* operations for standard FRP concepts.

A Larger Example

- Recall this FRP definition:

$x = (1/2) \int (v_r + v_l) \cos \theta$

- Assume that:

v_r SF, v_l SF :: SF SimbotInput Speed

θ :: SF SimbotInput Angle

then we can rewrite x in Yampa like this:

```
xSF :: SF SimbotInput Distance
```

```
xSF = let v = (vrSF && vlSF) >>> arr2 (+)
```

```
        t = thetaSF >>> arr cos
```

```
        in (v && t) >>> arr2 (*) >>> integral >>> arr (/2)
```

- Yikes!!! Is this as clear as the original code??

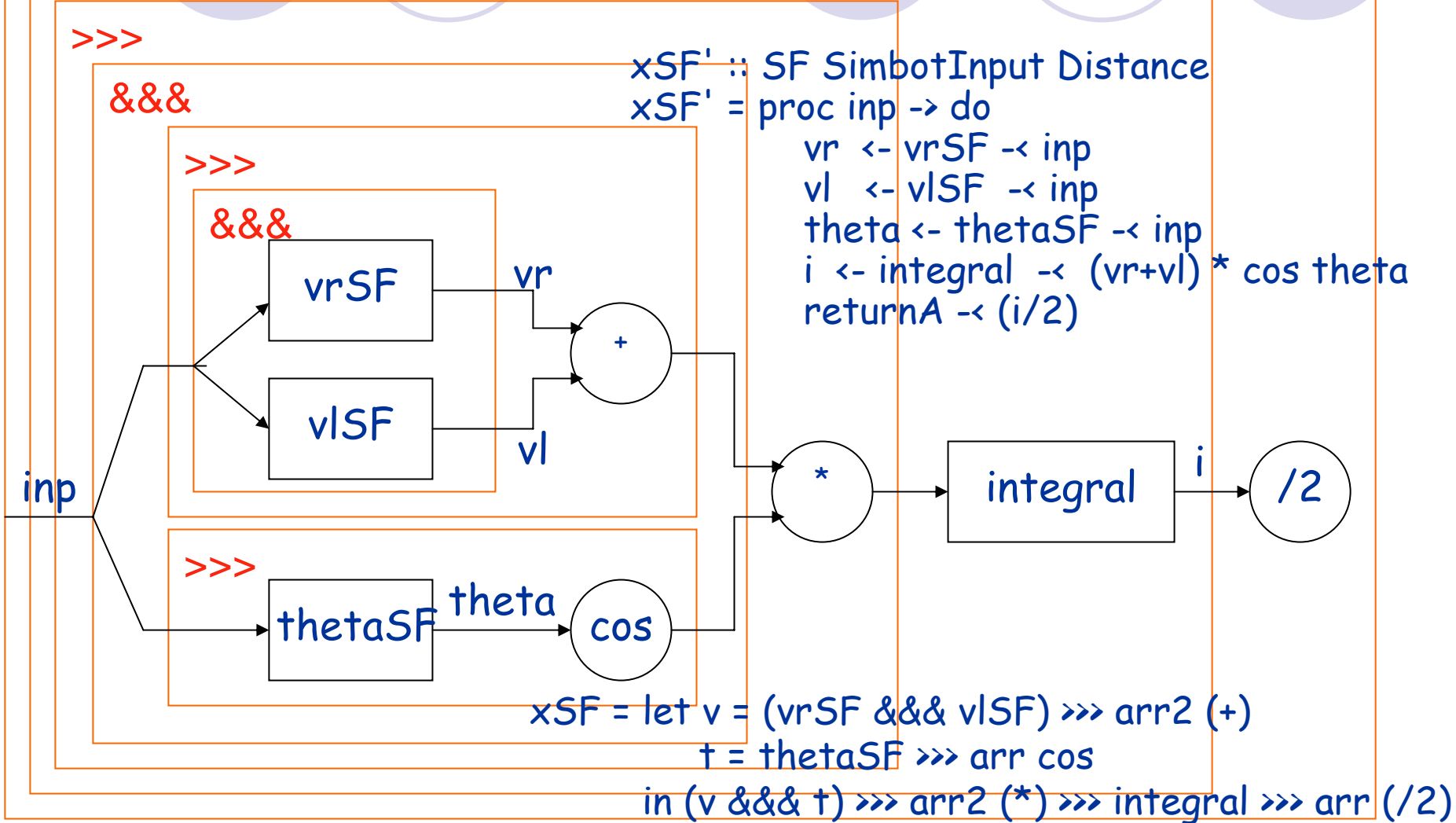
Arrow Syntax

- Using Paterson's *arrow syntax*, we can instead write:

```
xSF' :: SF SimbotInput Distance
xSF' = proc inp -> do
  vr    <- vrSF    -< inp
  v1    <- v1SF    -< inp
  theta <- thetaSF -< inp
  i     <- integral -< (vr+v1) * cos theta
  returnA -< (i/2)
```

- Feel better? 😊
- Note that *vr*, *v1*, *theta*, and *i* are *signal samples*, and not the signals themselves. Similarly, expressions to the right of “-<” denote signal samples.
- Read “`proc inp -> ...`” as “`\ inp -> ...`” in Haskell.
Read “`vr <- vrSF -< inp`” as “`vr = vrSF inp`” in Haskell.

>>> Graphical Depiction



A Recursive Mystery



- Our use of arrows was motivated by *performance* and *modularity*.
- But the improvement in performance seemed *better than expected*, and happened for FRP programs that looked Ok to us.
- Many of the problems seemed to occur with *recursive* signals, and had nothing to do with signals not being abstract enough.
- Further investigation of recursive signals is what the rest of this talk is about.
- We will see that arrows do indeed improve performance, but not just for the reasons that we first imagined!

Representing Signals

- Conceptually, *signals* are represented by:
 $\text{Signal } a \approx \text{Time} \rightarrow a$
- Pragmatically, this will not do: *stateful* signals could require re-computation at every time-step.

- Two possible alternatives:

- *Stream-based* implementation:

```
newtype S a = S ([DTime] -> [a])
```

(similar to that used in SOE and original FRP)

- *Continuation-based* implementation:

```
newtype C a = C (a, DTime -> C a)
```

(similar to that used in later FRP and Yampa)

(`DTime` is the domain of time intervals, or “delta times”.)

Integration: A Stateful Computation

- For convenience, we include an initialization argument:

```
integral :: a -> Signal a -> Signal a
```

- Concrete definitions:

```
integrals :: Double -> S Double -> S Double
```

```
integrals i (S f) =
```

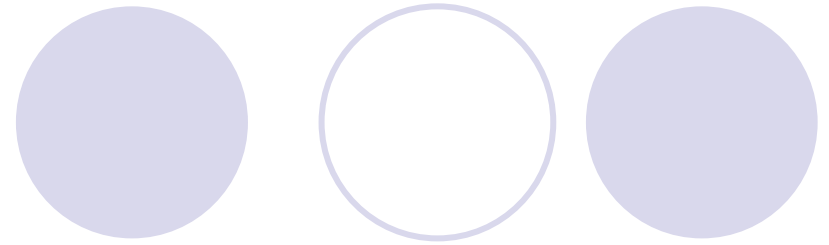
```
  S (\dts -> scanl (+) i (zipwith (*) dts (f dts)))
```

```
integralC :: Double -> C Double -> C Double
```

```
integralC i (C p) =
```

```
  C (i, \dt -> integralC (i + fst p * dt) (snd p dt))
```

“Running” a Signal



- Need a function to produce results:

```
run :: Signal a -> [a]
```

- For simplicity, we fix the delta time `dt` -- but this is not true in practice!

- Concretely:

```
runS :: S a -> [a]
```

```
runS (S f) = f (repeat dt)
```

```
runC :: C a -> [a]
```

```
runC (C p) = first p : runC (snd p dt)
```

```
dt = 0.001
```

- So far so good...

Example: The Exponential Function

- Consider this definition:

$$e(t) = 1 + \int_0^t e(t) dt$$

- Or, in our Haskell framework:

```
eS :: S Double
eS = integrals 1 eS

eC :: C Double
eC = integralC 1 eC
```

- Looks good... but is it really?

Space/Time Leak!

- Let $\text{int} = \text{integralC}$, $\text{run} = \text{runc}$, and recall:

```
int i (C p) = C (i, \dt-> int (i+fst p*dt) (snd p dt))
run (C p)   = first p : run (snd p dt)
```

- Then we can unwind ec :

```
ec = int 1 ec
    = C (1, \dt-> int (1+fst p*dt) (snd p dt) )
      ↑      └──────────────────┬──────────────────┘
      p                          |
    = C (1, \dt-> int (1+1*dt) (. dt) )
      ↑      └──────────┬──────────┘
      q
```

```
run ec
= run (C (1,q))
= 1 : run (q dt)
= 1 : run (int (1+dt) (q dt))
= 1 : run (C (1+dt, \dt-> int (1+dt*(1+dt)*dt) (. dt)))
= ...
```

- This leads to $O(n)$ space and $O(n^2)$ time to compute n elements! (Instead of $O(1)$ and $O(n)$.)


Streams are no better

- Recall:

```
int i (s f) =  
  s (\dts -> scan1 (+) i (zipwith (*) dts (f dts)))
```

- Therefore:

```
eS = int 1 eS  
    = S (\dts -> scan1 (+) 1 (zipwith (*) dts (. dts)))
```



- This leads to the same $O(n^2)$ behavior as before.

Signal Functions

- Instead of signals, suppose we focus on *signal functions*.
Conceptually:

```
SigFun a b = Signal a -> Signal b
```

- Concretely using continuations:

```
newtype CF a b = CF (a -> (b, DTime -> CF a b))
```

- Integration over CF:

```
integralCF :: Double -> CF Double Double  
integralCF i = CF (\x -> (i, \dt -> integralCF (i+dt*x)))
```

- Composition over CF:

```
(^.) :: CF b c -> CF a b -> CF a c  
CF f2 ^. CF f1 = CF (\a -> let (b,g1) = f1 a  
                               (c,g2) = f2 b  
                               in (c, \dt -> comp (g2 dt) (g1 dt)))
```

- Running a CF:

```
runCF :: CF () Double -> [Double]  
runCF (CF f) = let (i,g) = f ()  
                  in i : runCF (g dt)
```

Look Ma, No Leaks!

- This program still leaks:

```
eCF = integralCF 1 ^. eCF
```

- But suppose we define:

```
fixCF :: CF a a -> CF () a
fixCF (CF f) =
  CF (\() -> let (y, c) = f y
               in (y, \dt -> fixCF (c dt)))
```

- Then this program:

```
eCF = fixCF (integralCF 1)
```

does not leak!! It runs in constant space and linear time.

- To see why...

- Recall:

```
int i = CF (\x -> (i, \dt -> int (i+dt*x)))
fix (CF f) = CF (\() -> let (y, c) = f y
```

```
                        in (y, \dt -> fix (c dt)))
run (CF f) = let (i,g) = f () in i : run (g dt)
```

- Unwinding eCF:

```
fix (int 1)
= fix (CF (\x-> (1, \dt-> int (1+dt*x))))
= CF (\()-> let (y,c) = (1, \dt-> int (1+dt*y))
              in (y, \dt-> fix (c dt)))
= CF (\()-> (1, \dt-> fix (int (1+dt))))
```

```
run (↑)
= let (i,g) = (1, \dt-> fix (int (1+dt)))
  in i : run (g dt)
= 1 : run (fix (int (1+dt*y)))
```

- In short, `fixCF` creates a “tighter” loop than Haskell’s `fix`.

Mystery Solved

- Casting all this into the arrow framework reveals why Yampa is better behaved than FRP. In particular:

```
instance ArrowLoop CF where
  loop :: CF (b,d) (c,d) -> CF b c
  loop (CF f) = CF (\x -> let ((y,z), f') = f (x,z)
                           in (y, loop . f'))

e = proc () -> do rec
  e <- integral 1 -< e
  returnA -< e
```

- Compare loop to:

```
fixCF :: CF a a -> CF () a
fixCF (CF f) = CF (\() -> let (y, f') = f y
                          in (y, fixCF . f'))
```

Alternative Solution

- Recall this unwinding:

$$\begin{aligned} eC &= \text{int } 1 \text{ } eC \\ &= C (1, \underbrace{\backslash dt \rightarrow \text{int } (1+1*dt) \text{ } (. \text{ } dt) \text{ })}_q \end{aligned}$$

- The problem is that $(q \text{ } dt)$ is not recognized as being the same as q . What we'd really like is:

$$\begin{aligned} eC &= \dots \\ &= \dot{C} (1, \underbrace{\backslash dt \rightarrow \text{int } (1+1*dt) \text{ } .)} \end{aligned}$$

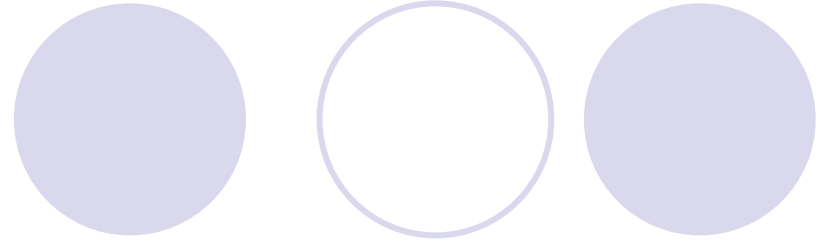
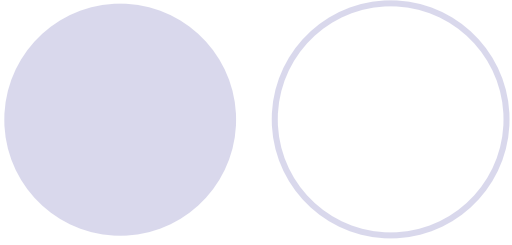
$$= C (1, \backslash dt \rightarrow \text{let loop} = \text{int } (1+dt) \text{ loop in loop})$$

- But this needs to happen on *each* step in the computation, and thus needs to be part of the evaluation strategy.
- Indeed, both *optimal reduction* [Levy, Lamping] and (interestingly) *completely lazy evaluation* [Sinot] do this, and the space / time leak goes away!

Final Thoughts

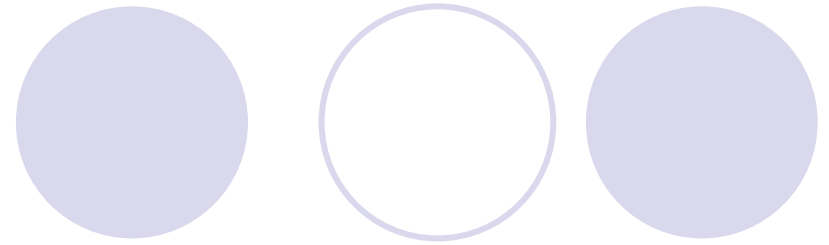


- Being able to *redefine recursion* (via fix) is a Good Thing!
- What is the “*correct*” evaluation strategy for a compiler?
- John Hughes’ original motivation for arrows arose out of the desire to plug a space leak in monadic parsers – is this just a coincidence?
- There are many other performance issues involving arrows (e.g. excessive tupling) and we are exploring *optimization methods* (e.g. using arrows laws, zip/unzip fusion, etc).
- An ambitious goal: *real-time* sound generation for Haskore / HasSound on stock hardware.



The End

Monadic Parsers



- Need *failure* and *choice*:

```
class Monad m => MonadZero m where
  zero :: m a
class MonadZero m => MonadPlus m where
  (++) :: m a -> m a -> m a
```

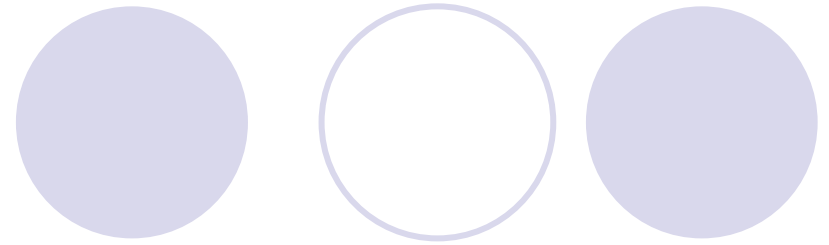
- $p1 ++ p2$ means “try parse $p1$ – if it fails, then try $p2$.”
- A monadic parser based on:

```
data Parser s a = P ([s] -> Maybe (a, [s]))
```

leads to a space leak:

processing $p1 ++ p2$ requires holding on to the stream being parsed by $p1$.

Plugging the Leak



- This problem can be fixed through some cleverness that leads to this representation of parsers:

```
data Parser s a = P (StaticP s) (DynamicP s a)
```
- The cleverness requires that `(++)` see the static part of *both* of its arguments – but there’s no way to achieve this with `bind`:

```
(>>=) :: Parser s a -> (a -> Parser s b) -> Parser s b)
```
- What to do? Make “`(a -> Parser s b)`” *abstract* – i.e. define an arrow `Parser a b`.

Arrows

- **A b c** is the arrow type of computations that take inputs of type **b** and produce outputs of type **c**.
- The arrow combinators impose a point-free programming style:

`arr :: (b -> c) -> A b c`

`(>>>) :: A b c -> A c d -> A b d`

`first :: A b c -> A (b,d) (c,d)`

`(***) :: A b d -> A c e -> A (b,c) (d,e)`

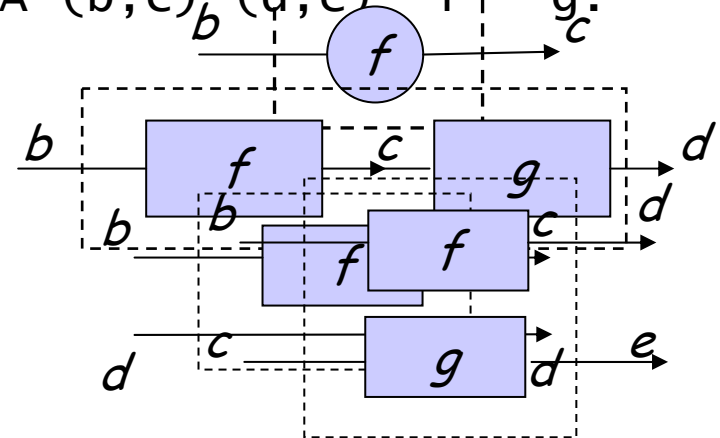
`arr f:`

`f >>> g:`

`first f:`

`f***g:`

Every pure function may be treated as a computation
 Computations can be composed sequentially
 A computation may be applied to part of the input
 Two computations can be composed in parallel



Arrow and ArrowLoop classes

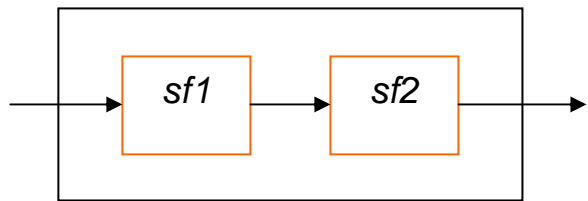
- As with monads, we use *type classes* to capture the arrow combinators.

```
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)
```

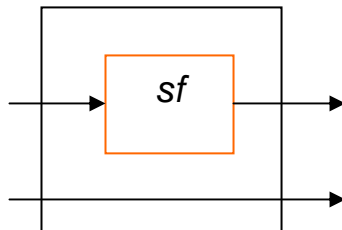
```
class Arrow a => ArrowLoop a where
  loop :: a (b,d) (c,d) -> a b c
```

(`loop` can be thought of as a *fixpoint operator* for arrows.)

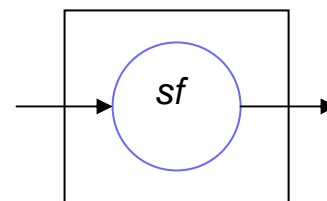
Graphical Depiction of Arrow Combinators



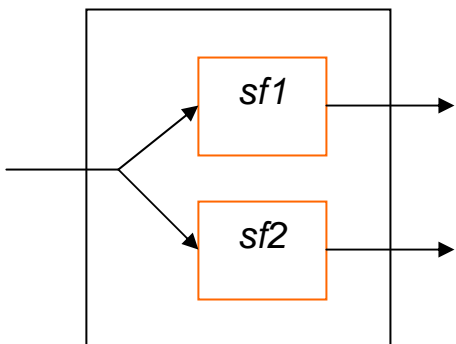
sf1 >>> sf2



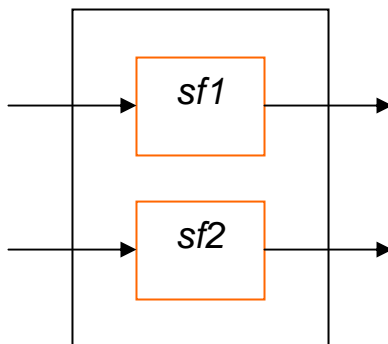
first sf



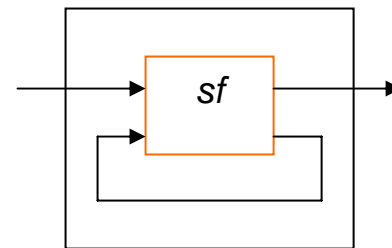
arr sf



sf1 &&& sf2



*sf1 *** sf2*



loop sf

Signal Functions in Yampa

- Conceptually: $SF\ a\ b = Signal\ a \rightarrow Signal\ b$
- But it is more efficient to design from scratch:

```
data SF a b = SF (a -> (b, DTime -> SF a b))
```

```
instance Arrow SF where
```

```
  arr f x      = (f x, \dt -> arr f)
```

```
  first f (x, z) = ((y, z), first . f')
```

```
                where (y, f') = f x
```

```
  (f >>> g) x  = (z, \dt -> f' dt >>> g' dt)
```

```
                where (y, f') = f x
```

```
                      (z, g') = g y
```

```
instance ArrowLoop SF where
```

```
  loop f x = (y, loop . f')
```

```
            where ((y, z), f') = f (x, z)
```

(Note “tight” recursion.)