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).
A single animation example that demonstrates key aspects of FRP:

\[
growFlower = \text{stretch size flower} \\
\text{where size} = 1 + \text{integral bSign}
\]

\[
bSign = \\
0 \text{ `until`} \\
(\text{lbp ==> -1 `until` lbr ==> bSign}) .|. \\
(\text{rbp ==> 1 `until` rbr ==> bSign})
\]
Differential Drive Mobile Robot

\[ \theta \]

\[ v_l \]

\[ v_r \]

\[ l \]
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:

\[ x = (1/2) * (\text{integral} \ ((v_r + v_l) * \cos \ theta)) \]
\[ \theta = (1/l) * (\text{integral} \ (v_r - v_l)) \]

(Note the lack of explicit time.)
Behaviors in FRP are what we now call *signals*, whose (abstract) type is:

\[
\text{Signal } a = \text{Time } \rightarrow a
\]

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*:
  \[ SF \ a \ b = \text{Signal} \ a \to \text{Signal} \ b \]
- *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:
  \[ x = \frac{1}{2} \int ((v_r + v_l) \times \cos \theta) \]

- Assume that:
  \[ v_rSF, v_lSF :: SF \text{ SimbotInput Speed} \]
  \[ \theta :: SF \text{ SimbotInput Angle} \]
  then we can rewrite \( x \) in Yampa like this:

  \[ xSF :: SF \text{ SimbotInput Distance} \]
  \[ xSF = \text{let } v = (v_rSF \&\& v_lSF) >>\!
  \text{arr2 (+)}
  \]
  \[ t = \theta SF >>\!
  \text{arr cos}
  \]
  \[ \text{in } (v \&\& t) >>\!
  \text{arr2 (*)} >>\!
  \text{integral} >>\!
  \text{arr (}/2) \]

- Yikes!!! Is this as clear as the original code??
Using Paterson’s *arrow syntax*, we can instead write:

\[
xSF' :: SF \text{ SimbotInput Distance}
\]

\[
xSF' = \text{proc inp -> do}
\]

\[
vr <- vrSF \text{ inp} \\
v1 <- v1SF \text{ inp} \\
\theta <- \theta\text{SF} \text{ inp} \\
i <- \text{integral} \text{ inp} (vr+v1) \times \cos \theta \\
\text{returnA} \text{ inp} \frac{i}{2}
\]

Feel better? 😊

Note that \(vr\), \(v1\), \(\theta\), and \(i\) are *signal samples*, and not the signals themselves. Similarly, expressions to the right of “\text{--}” denote signal samples.

Read “\text{proc inp -> ...}” as “\text{\ inp -> ...}” in Haskell. Read “\(vr <- vr\text{SF} \text{ inp}\)” as “\(vr = vr\text{SF} \text{ inp}\)” in Haskell.
Graphical Depiction

$xSF' :: SF \text{SimbotInput Distance} \rightarrow A$

$xSF' = \text{proc inp} -> \text{do}$

\begin{align*}
vr & \leftarrow vrSF \leftarrow \text{inp} \\
vl & \leftarrow vlSF \leftarrow \text{inp} \\
\theta & \leftarrow \thetaSF \leftarrow \text{inp} \\
i & \leftarrow \text{integral} \leftarrow (vr+vl) \cdot \cos \theta \\
\text{return}A & \leftarrow (i/2)
\end{align*}

$xSF = \text{let } v = (vrSF \&\& vlSF) \ggg \text{arr2 (+)}$

\begin{align*}
\theta & = \thetaSF \ggg \text{arr cos} \\
in (v \&\& \theta) & \ggg \text{arr2 (*)} \ggg \text{integral} \ggg \text{arr (/2)}
\end{align*}
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:
  \[ \text{Signal } a \approx \text{Time } \to a \]

- Pragmatically, this will not do: **stateful** signals could require re-computation at every time-step.

- Two possible alternatives:
  - **Stream-based** implementation:
    \[ \text{newtype } S a = S ([\text{DTime}] \to [a]) \]
    (similar to that used in SOE and original FRP)
  - **Continuation-based** implementation:
    \[ \text{newtype } C a = C (a, \text{DTime } \to C a) \]
    (similar to that used in later FRP and Yampa)
    \(\text{DTime}\) is the domain of time intervals, or “delta times”.)
**Integration: A Stateful Computation**

- For convenience, we include an initialization argument:
  
  \[ \text{integral} :: a \rightarrow \text{Signal } a \rightarrow \text{Signal } a \]

- Concrete definitions:

  \[
  \begin{align*}
  \text{integralS} :: \text{Double} \rightarrow \text{S Double} \rightarrow \text{S Double} \\
  \text{integralS} \ i \ (S \ f) &= \\
  &S (\text{scanl } (+) \ i \ (\text{zipWith } (*) \ \text{dts} \ (f \ \text{dts}))
  \\
  \\
  \text{integralC} :: \text{Double} \rightarrow \text{C Double} \rightarrow \text{C Double} \\
  \text{integralC} \ i \ (C \ p) &= \\
  &C (i, \text{\ dt } \rightarrow \text{integralC} \ (i + \text{fst } p \ \ast \ \text{dt}) \ (\text{snd } p \ \text{dt}))
  \end{align*}
  \]
“Running” a Signal

- Need a function to produce results:
  \[
  \text{run} :: \text{Signal} \ a \to \ [a]
  \]

- For simplicity, we fix the delta time \( \text{dt} \) -- but this is not true in practice!

- Concretely:
  \[
  \text{runS} :: \text{S} \ a \to [a]
  \]
  \[
  \text{runS} \ (\text{S} \ f) = f \ (\text{repeat} \ \text{dt})
  \]

  \[
  \text{runC} :: \text{C} \ a \to [a]
  \]
  \[
  \text{runC} \ (\text{C} \ p) = \text{first} \ p : \ \text{runC} \ (\text{snd} \ p \ \text{dt})
  \]

- 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:
  ```haskell
  eS :: S Double
  eS = integralS 1 eS
  
  eC :: C Double
  eC = integralC 1 eC
  ```

- Looks good... but is it really?
Space/ Time Leak!

Let $\text{int} = \text{integral}_C$, $\text{run} = \text{run}_C$, and recall:

\[
\text{int } i (C \ p) = C (i, \ \lambda dt -> \text{int} (i + \text{fst} \ p \cdot dt) (\text{snd} \ p \ dt))
\]
\[
\text{run} (C \ p) = \text{first} \ p : \text{run} (\text{snd} \ p \ dt)
\]

Then we can unwind $e_C$:

\[
e_C = \text{int} 1 \ e_C
\]
\[
= C (1, \ \lambda dt -> \text{int} (1 + \text{fst} \ p \cdot dt) (\text{snd} \ p \ dt))
\]
\[
= C (1, \ \lambda dt -> \text{int} (1 + 1 \cdot dt) (\cdot dt))
\]

\[
\text{run} \ e_C
\]
\[
= \text{run} (C (1, q))
\]
\[
= 1 : \text{run} (q \ dt)
\]
\[
= 1 : \text{run} (\text{int} (1 + dt) (q \ dt))
\]
\[
= 1 : \text{run} (C (1 + dt, \ \lambda dt -> \text{int} (1 + dt \cdot (1 + dt) \cdot dt) (\cdot dt)))
\]
\[
= \ldots
\]

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:
  \[
  \text{int } i \ (S \ f) = S \ (\lambda dts \rightarrow \text{scanl } (+) \ i \ (\text{zipWith } (*) \ dts \ (f \ dts))
  \]

- Therefore:
  \[
  eS = \text{int } 1 \ eS
  = S \ (\lambda dts \rightarrow \text{scanl } (+) \ 1 \ (\text{zipWith } (*) \ dts \ (\cdot \ dts))
  \]

- This leads to the same \(O(n^2)\) behavior as before.
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:
  \[ e_{CF} = \text{integral}_{CF} 1 \ \&\&.\ e_{CF} \]

- But suppose we define:
  \[ \text{fix}_{CF} :: \text{CF} \ a \ a \to \text{CF} \ () \ a \]
  \[ \text{fix}_{CF} \ (\text{CF} \ f) = \]
  \[ \text{CF} \ () \to \text{let} \ (y, \ c) = f \ y \]
  \[ \text{in} \ (y, \ \delta t \to \text{fix}_{CF} \ (c \ \delta t)) \]

- Then this program:
  \[ e_{CF} = \text{fix}_{CF} \ (\text{integral}_{CF} 1) \]
  does not leak!! It runs in constant space and linear time.

- To see why...
Recall:

\[ \text{int } i = \text{CF } (\lambda x \rightarrow (i, \lambda dt \rightarrow \text{int } (i+dt \times x))) \]

\[ \text{fix } (\text{CF } f) = \text{CF } (\lambda () \rightarrow \text{let } (y, c) = f y \]
\[ \text{in } (y, \lambda dt \rightarrow \text{fix } (c dt))) \]

\[ \text{run } (\text{CF } f) = \text{let } (i, g) = f () \text{ in } i : \text{run } (g dt) \]

Unwinding eCF:

\[ \text{fix } (\text{int } 1) \]
\[ = \text{fix } (\text{CF } (\lambda x \rightarrow (1, \lambda dt \rightarrow \text{int } (1+dt \times x)))) \]
\[ = \text{CF } (\lambda () \rightarrow \text{let } (y, c) = (1, \lambda dt \rightarrow \text{int } (1+dt \times y)) \]
\[ \text{in } (y, \lambda dt \rightarrow \text{fix } (c dt))) \]
\[ = \text{CF } (\lambda () \rightarrow (1, \lambda dt \rightarrow \text{fix } (\text{int } (1+dt)))) \]

\[ \text{run } () \]
\[ = \text{let } (i, g) = (1, \lambda dt \rightarrow \text{fix } (\text{int } (1+dt))) \]
\[ \text{in } i : \text{run } (g dt) \]
\[ = 1 : \text{run } (\text{fix } (\text{int } (1+dt \times y))) \]

In short, fixCF creates a “tighter” loop than Haskell’s fix.
Casting all this into the arrow framework reveals why Yampa is better behaved than FRP. In particular:

```haskell
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:

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

- Recall this unwinding:
  \[ eC = \text{int } 1 \ eC \]
  \[ = C \ (1, \ \lambda dt \to \text{int } (1+1\times dt) \cdot dt) \]

- The problem is that \((q \ dt)\) is not recognized as being the same as \(q\). What we’d really like is:
  \[ eC = \ldots \]
  \[ = C \ (1, \ \lambda dt \to \text{int } (1+1\times dt) \cdot) \]
  \[ = C \ (1, \ \lambda dt \to \text{let loop } = \text{int } (1+dt) \text{ loop in loop} \]

- 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*:

  ```haskell
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:

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

```haskell
data Parser s a = P (StaticP s) (DynamicP s a)
```

The cleverness requires that `s ++` see the static part of both of its arguments – but there’s no way to achieve this with bind:

```haskell
(>>=) :: 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)
  ```

  Every pure function may be treated as a computation. Computations can be composed sequentially. Two computations may be applied to part of the input. They can also be composed in parallel.
Arrow and ArrowLoop classes

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

```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)

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:

```haskell
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.)