

Push-pull functional reactive programming

Conal Elliott

3 September, 2009

Haskell Symposium

1 Functional reactive programming

- Semantics
- Building blocks
- Refactoring

2 Future values

- Class instances
- Future times

3 Improving values

- Description and problems
- Improving

4 Unambiguous choice

What is Functional Reactive Programming?

- Composable dynamic values,
- ... with simple & precise semantics.
- *Continuous* time (zoomable).
- Fine-grain, *deterministic* concurrency.

Classic FRP – semantic model

Behaviors (signals) are flows of values, punctuated by event occurrences.

$$\llbracket Behavior_{\alpha} \rrbracket = \mathcal{T} \rightarrow \alpha$$

$$\llbracket Event_{\alpha} \rrbracket = [\hat{\mathcal{T}} \times \alpha] \text{ -- monotonic}$$

Behaviors compose

$time :: Behavior_{\mathcal{T}}$

$\llbracket time \rrbracket = id$

$pure :: \alpha \rightarrow Behavior_{\alpha}$

$\llbracket pure a \rrbracket = \lambda t \rightarrow a$
 $= pure a$

$(\langle * \rangle) :: Behavior_{\alpha \rightarrow \beta} \rightarrow Behavior_{\alpha} \rightarrow Behavior_{\beta}$

$\llbracket fs \langle * \rangle as \rrbracket = \lambda t \rightarrow ((\llbracket fs \rrbracket t) (\llbracket as \rrbracket t))$
 $= \llbracket fs \rrbracket \langle * \rangle \llbracket as \rrbracket$

Events compose

$\emptyset :: Event_{\alpha}$
 $\llbracket \emptyset \rrbracket = []$

$(\oplus) :: Event_{\alpha} \rightarrow Event_{\alpha} \rightarrow Event_{\alpha}$
 $\llbracket e \oplus e' \rrbracket = \llbracket e \rrbracket \text{ 'merge' } \llbracket e' \rrbracket$

$fmap\ n :: (\alpha \rightarrow \beta) \rightarrow Event_{\alpha} \rightarrow Event_{\beta}$
 $\llbracket fmap\ f\ e \rrbracket = map\ (\lambda(t, a) \rightarrow (t, f\ a))\ \llbracket e \rrbracket$
 $\quad = fmap\ (fmap\ f)\ \llbracket e \rrbracket$

Events punctuate behaviors

$$\text{stepper} :: \alpha \rightarrow \text{Event}_{\alpha} \rightarrow \text{Behavior}_{\alpha}$$

More generally,

$$\text{switcher} :: \text{Behavior}_{\alpha} \rightarrow \text{Event}_{\text{Behavior}_{\alpha}} \rightarrow \text{Behavior}_{\alpha}$$

Main idea of the paper: Behaviors are chains of simple phases

So represent as such:

$$Behavior_a = (\mathcal{T} \rightarrow a) \times (\widehat{\mathcal{T}} \times Behavior_a)$$

Catch: We need *lazy expiration times*.

Generalize/simplify – Reactive values

$$Behavior_{\alpha} = (\mathcal{T} \rightarrow \alpha) \times (\widehat{\mathcal{T}} \times Behavior_{\alpha})$$

Generalize:

$$Reactive_{\beta} = \beta \times (\widehat{\mathcal{T}} \times Reactive_{\beta}) \quad \text{-- discrete reactive}$$

And specialize:

$$\llbracket TFun_{\alpha} \rrbracket = \mathcal{T} \rightarrow \alpha \quad \text{-- continuous non-reactive}$$

$$Behavior = Reactive \circ TFun$$

This representation provides *Functor* and *Applicative* instances.

TFun constant-folds

```
data Fun t a = K a | Fun (t → a)  
[[Fun t a]] = t → a
```

```
data TFun = Fun  $\mathcal{T}$ 
```

```
[[K a]] = const a  
[[Fun f]] = f
```

```
instance Functor (TFun t) where  
  fmap f (K a) = K (f a)  
  fmap f (Fun g) = Fun (f ∘ g)
```

etc

Generalize/simplify – Future values

$$\text{Reactive}_\beta = \beta \times (\widehat{\mathcal{T}} \times \text{Reactive}_\beta)$$

becomes

$$\text{Future}_\gamma = \widehat{\mathcal{T}} \times \gamma$$

$$\text{Reactive}_\beta = \beta \times \text{Future}_{\text{Reactive}_\beta}$$

Events are future reactives

$$\text{Reactive}_{\beta} = \beta \times \text{Future}_{\text{Reactive}_{\beta}}$$

becomes

$$\text{Event}_{\alpha} = \text{Future}_{\text{Reactive}_{\alpha}}$$

$$\text{Reactive}_{\beta} = \beta \times \text{Event}_{\beta}$$

Summarizing

$$\mathit{Future}_\gamma = \widehat{\mathcal{T}} \times \gamma$$

$$\mathit{Event}_\alpha = \mathit{Future}_{\mathit{Reactive}_\alpha}$$

$$\mathit{Reactive}_\beta = \beta \times \mathit{Event}_\beta$$

$$\mathit{Behavior} = \mathit{Reactive} \circ \mathit{TFun}$$

$$\mathbf{data} \mathit{Fun} \mathit{t} \mathit{a} = \mathit{K} \mathit{a} \mid \mathit{Fun} (\mathit{t} \rightarrow \mathit{a})$$

Future values are mostly easy

newtype *Future* $\alpha = \text{Fut } (\widehat{\mathcal{T}}, \alpha)$

deriving (*Functor*, *Applicative*, *Monad*)

For *Applicative* and *Monad*, the $\widehat{\mathcal{T}}$ monoid uses *max* and $-\infty$.

What about Monoid?

A first try:

(\oplus) chooses the earlier of two futures:

instance *Monoid* (*Future* α) **where**

$$\emptyset = \text{Fut } (\infty, \perp)$$

$$u_a @ (\text{Fut } (\hat{t}_a, -)) \oplus u_b @ (\text{Fut } (\hat{t}_b, -)) = \\ \text{if } \hat{t}_a \leq \hat{t}_b \text{ then } u_a \text{ else } u_b$$

We'll have to compare future times without knowing both fully.
Even so, there's a problem ...

(\oplus) must be even lazier.

First try:

$$u_a @ (Fut (\hat{t}_a, -)) \oplus u_b @ (Fut (\hat{t}_b, -)) = \\ \text{if } \hat{t}_a \leq \hat{t}_b \text{ then } u_a \text{ else } u_b$$

Produces *no* information until resolving $\hat{t}_a \leq \hat{t}_b$.

Consider $(u_a \oplus u_b) \oplus u_c$, where u_c is earliest. *Oops.*

Solution:

$$Fut (\hat{t}_a, a) \oplus Fut (\hat{t}_b, b) = \\ Fut (\hat{t}_a \text{ 'min' } \hat{t}_b, \text{if } \hat{t}_a \leq \hat{t}_b \text{ then } a \text{ else } b)$$

Can be optimized.

What are future times?

type $\widehat{T} = \text{Max} (\text{AddBounds} (\text{Improving } T))$

- *Max* monoid for derived *Applicative Future* (and *Monad*)
- *AddBounds* for the *Future* and *Max* monoids
- *Improving* for partiality

What are improving values?

- Lazy values, as a monotonic sequence of lower bounds
- Invented by Warren Burton in the 1980s
- Operations for *min* and *max*
- Purely functional implementation

Improving values have some problems.

- Expensive to step through accumulated lower bounds
- Redundant traversal
- Needs a generator of lower bounds

Can we improve on improving values?

What operations do we use on \hat{T} ?

exact :: *Improving*_a → *a*
*compare*_l :: *Improving*_a → *a* → *Ordering*
min :: *Improving*_a → *Improving*_a → *Improving*_a
 (≤) :: *Improving*_a → *Improving*_a → *Bool*

Puzzle: Can *exact* and *compare*_l implement *min* and (≤)?
 If so,

data *Improving* *a* =
 Imp { *exact* :: *a*, *compare*_l :: *a* → *Ordering* }

Comparing improving values – dilemma

How to compare future times: $\hat{t}_a \leq \hat{t}_b$? Two ideas:

$$ab = \text{compare}_I \hat{t}_a (\text{exact } \hat{t}_b) \neq GT$$

$$ba = \text{compare}_I \hat{t}_b (\text{exact } \hat{t}_a) \neq LT$$

Which to try first? We can't know beforehand.

Try both

Same answer when defined, so try in parallel and take first answer

$$ab \text{ 'unamb' } ba$$

Referentially transparent? Yes:

$$ab \text{ 'unamb' } ba \equiv ab \sqcup ba$$

Crucial: ab and ba agree when defined.

unamb is handy

$parCommute\ op\ x\ y = (x\ 'op'\ y)\ 'unamb'\ (y\ 'op'\ x)$

$por = parCommute\ (\vee)$

$pand = parCommute\ (\wedge)$

-- handy with unamb

$assuming :: Bool \rightarrow a \rightarrow a$

$assuming\ True\ a = a$

$assuming\ False\ _ = \perp$

Symmetric short-circuiting

parAnnihilator op a x y =
assuming (x ≡ a) a 'unamb'
assuming (y ≡ a) a 'unamb'
(x 'op' y)

por = parAnnihilator (∨) True

pand = parAnnihilator (∧) False

pmul = parAnnihilator (×) 0

pmin = parAnnihilator min minBound

pmax = parAnnihilator max maxBound

min is simple – almost

data *Ordering* = *LT* | *EQ* | *GT* **deriving** (*Eq*, *Ord*, *Bounded*, ...)

compare (a 'min' b) x = *compare* a x 'min' *compare* b x

Similarly,

*compare*_l (\hat{t}_a 'min' \hat{t}_b) t = *compare*_l \hat{t}_a t 'min' *compare*_l \hat{t}_b t

Too strict. Consider *exact* $\hat{t}_a < t < \textit{exact}$ \hat{t}_b . Easy fix, via *unamb*:

*compare*_l (\hat{t}_a 'min' \hat{t}_b) t = *compare*_l \hat{t}_a t 'pmin' *compare*_l \hat{t}_b t

Summary

- Refactored FRP suggests hybrid data/function representation
- Data-driven but still pull-based, due to blocking threads
- Semantic determinacy saved, thanks to unambiguous choice

Future work

- Subtle RTS and/or laziness.
- Measure and tune, esp improving values & *unamb*.
- Useful for arrow-based FRP?
- More fun with *unamb*
- *Event* is fishy. No semantic TCM, monad assoc can fail.
- Extend caching to *TFun*.