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.

$$[\![Behavior_\alpha]\!] = \mathcal{T} \rightarrow \alpha$$

$$[\![Event_\alpha]\!] \quad = [\widehat{\mathcal{T}} \times \alpha] \quad \text{-- monotonic}$$

Behaviors compose

$$time :: Behavior_{\mathcal{T}}$$
$$[\![time]\!] = id$$

$$pure :: \alpha \to Behavior_{\alpha}$$
$$\begin{aligned}[\![pure\ a]\!] &= \lambda t \to a \\ &= pure\ a\end{aligned}$$

$$(\circledast) :: Behavior_{\alpha \to \beta} \to Behavior_{\alpha} \to Behavior_{\beta}$$
$$\begin{aligned}[\![fs \circledast as]\!] &= \lambda t \to (([\![fs]\!]\ t)\ ([\![as]\!]\ t)) \\ &= [\![fs]\!] \circledast [\![as]\!]\end{aligned}$$

Events compose

$$\emptyset :: \mathit{Event}_\alpha$$
$$[\![\emptyset]\!] = [\,]$$

$$(\oplus) :: \mathit{Event}_\alpha \rightarrow \mathit{Event}_\alpha \rightarrow \mathit{Event}_\alpha$$
$$[\![e \oplus e']\!] = [\![e]\!] \text{ `}\mathit{merge}\text{` } [\![e']\!]$$

$$\mathit{fmap}\ n :: (\alpha \rightarrow \beta) \rightarrow \mathit{Event}_\alpha \rightarrow \mathit{Event}_\beta$$
$$\begin{aligned} [\![\mathit{fmap}\ f\ e]\!] &= \mathit{map}\ (\lambda(t,a) \rightarrow (t, f\ a))\ [\![e]\!] \\ &= \mathit{fmap}\ (\mathit{fmap}\ f)\ [\![e]\!] \end{aligned}$$

Events punctuate behaviors

$$stepper :: \alpha \rightarrow Event_{\alpha} \rightarrow Behavior_{\alpha}$$

More generally,

$$switcher :: Behavior_{\alpha} \rightarrow Event_{Behavior_{\alpha}} \rightarrow Behavior_{\alpha}$$

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

So represent as such:

$$\mathit{Behavior}_a = (\mathcal{T} \to a) \times (\widehat{\mathcal{T}} \times \mathit{Behavior}_a)$$

Catch: We need *lazy expiration times*.

Generalize/simplify – Reactive values

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

Generalize:

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

And specialize:

$$[\![TFun_\alpha]\!] = \mathcal{T} \to \alpha \quad \text{-- continuous non-reactive}$$

$$Behavior = Reactive \circ TFun$$

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

TFun constant-folds

$$\textbf{data}\ \mathit{Fun}\ t\ a = K\ a \mid \mathit{Fun}\ (t \rightarrow a)$$
$$[\![\mathit{Fun}\ t\ a]\!] = t \rightarrow a$$

$$\textbf{data}\ \mathit{TFun} = \mathit{Fun}\ \mathcal{T}$$

$$[\![K\ \ \ a]\!] = \mathit{const}\ a$$
$$[\![\mathit{Fun}\ f]\!] = f$$

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ (\mathit{TFun}\ t)\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ (K\ a) \quad\ = K\ \ \ (f\ a)\\ &\quad \mathit{fmap}\ f\ (\mathit{Fun}\ g) = \mathit{Fun}\ (f \circ g)\end{aligned}$$

etc

Generalize/simplify – Future values

$$Reactive_\beta = \beta \times (\widehat{\mathcal{T}} \times Reactive_\beta)$$

becomes

$$\begin{aligned} Future_\gamma &= \widehat{\mathcal{T}} \times \gamma \\ Reactive_\beta &= \beta \times Future_{Reactive_\beta} \end{aligned}$$

Events are future reactives

$$Reactive_\beta = \beta \times Future_{Reactive_\beta}$$

becomes

$$Event_\alpha = Future_{Reactive_\alpha}$$

$$Reactive_\beta = \beta \times Event_\beta$$

Summarizing

$$
\begin{aligned}
\mathit{Future}_{\gamma} &= \widehat{\mathcal{T}} \times \gamma \\
\mathit{Event}_{\alpha} &= \mathit{Future}_{\mathit{Reactive}_{\alpha}} \\
\mathit{Reactive}_{\beta} &= \beta \times \mathit{Event}_{\beta} \\
\mathit{Behavior} &= \mathit{Reactive} \circ \mathit{TFun}
\end{aligned}
$$

$$
\textbf{data}\ \mathit{Fun}\ t\ a = K\ a \mid \mathit{Fun}\ (t \to a)
$$

Future values are mostly easy

newtype *Future* α = *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**

$$\emptyset = \mathit{Fut}\ (\infty, \bot)$$

$$u_a@(\mathit{Fut}\ (\hat{t}_a, \_)) \oplus u_b@(\mathit{Fut}\ (\hat{t}_b, \_)) = \\ \quad \mathbf{if}\ \hat{t}_a \leqslant \hat{t}_b\ \mathbf{then}\ u_a\ \mathbf{else}\ u_b$$

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

$(\oplus)$ must be even lazier.

First try:

$$u_a@(Fut\ (\hat{t}_a, \_)) \oplus u_b@(Fut\ (\hat{t}_b, \_)) =$$
$$\quad \mathbf{if}\ \hat{t}_a \leqslant \hat{t}_b\ \mathbf{then}\ u_a\ \mathbf{else}\ u_b$$

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

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

Solution:

$$Fut\ (\hat{t}_a, a) \oplus Fut\ (\hat{t}_b, b) =$$
$$\quad Fut\ (\hat{t}_a\ \text{`}min\text{`}\ \hat{t}_b, \mathbf{if}\ \hat{t}_a \leqslant \hat{t}_b\ \mathbf{then}\ a\ \mathbf{else}\ b)$$

Can be optimized.

What are future times?

$$\textbf{type}\ \widehat{\mathcal{T}} = \mathit{Max}\ (\mathit{AddBounds}\ (\mathit{Improving}\ \mathcal{T}))$$

- *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 $\widehat{\mathcal{T}}$?

$$\begin{aligned}
exact &:: Improving_a \rightarrow a \\
compare_I &:: Improving_a \rightarrow a \rightarrow Ordering \\
min &:: Improving_a \rightarrow Improving_a \rightarrow Improving_a \\
(\leqslant) &:: Improving_a \rightarrow Improving_a \rightarrow Bool
\end{aligned}$$

Puzzle: Can *exact* and $compare_I$ implement *min* and $(\leqslant)$?
If so,

$$\begin{aligned}
&\textbf{data}\ Improving\ a = \\
&\quad Imp\ \{\, exact :: a, compare_I :: a \rightarrow Ordering \,\}
\end{aligned}$$

Comparing improving values – dilemma

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

$$ab = compare_I\ \hat{t}_a\ (exact\ \hat{t}_b) \not\equiv GT$$
$$ba = compare_I\ \hat{t}_b\ (exact\ \hat{t}_a) \not\equiv LT$$

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

Try both

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

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

Referentially transparent? Yes:

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

Crucial: *ab* and *ba* agree when defined.

unamb is handy

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

$$\begin{aligned} por &= parCommute\ (\vee) \\ pand &= parCommute\ (\wedge) \end{aligned}$$

-- handy with unamb

$$\begin{aligned} &assuming :: Bool \rightarrow a \rightarrow a \\ &assuming\ True\ a = a \\ &assuming\ False\ \_ = \bot \end{aligned}$$

Symmetric short-circuiting

$$
\begin{aligned}
&\mathit{parAnnihilator}\ \mathit{op}\ a\ x\ y = \\
&\quad \mathit{assuming}\ (x \equiv a)\ a\ `\mathit{unamb}` \\
&\quad \mathit{assuming}\ (y \equiv a)\ a\ `\mathit{unamb}` \\
&\quad (x\ `\mathit{op}`\ y)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{por} &= \mathit{parAnnihilator}\ (\vee)\ \ \mathit{True} \\
\mathit{pand} &= \mathit{parAnnihilator}\ (\wedge)\ \ \mathit{False} \\
\mathit{pmul} &= \mathit{parAnnihilator}\ (\times)\ \ 0 \\
\mathit{pmin} &= \mathit{parAnnihilator}\ \mathit{min}\ \ \mathit{minBound} \\
\mathit{pmax} &= \mathit{parAnnihilator}\ \mathit{max}\ \mathit{maxBound}
\end{aligned}
$$

min is simple – almost

$$\textbf{data}\ Ordering = LT \mid EQ \mid GT\ \textbf{deriving}\ (Eq, Ord, Bounded, ...)$$

$$compare\ (a\ \text{`}min\text{`}\ b)\ x = compare\ a\ x\ \text{`}min\text{`}\ compare\ b\ x$$

Similarly,

$$compare_I\ (\hat{t}_a\ \text{`}min\text{`}\ \hat{t}_b)\ t = compare_I\ \hat{t}_a\ t\ \text{`}min\text{`}\ compare_I\ \hat{t}_b\ t$$

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

$$compare_I\ (\hat{t}_a\ \text{`}min\text{`}\ \hat{t}_b)\ t = compare_I\ \hat{t}_a\ t\ \text{`}pmin\text{`}\ compare_I\ \hat{t}_b\ t$$

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*.