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

\[
\begin{align*}
\llbracket \text{Behavior}_\alpha \rrbracket &= \mathcal{T} \rightarrow \alpha \\
\llbracket \text{Event}_\alpha \rrbracket &= \left[ \hat{T} \times \alpha \right] \quad \text{-- monotonic}
\end{align*}
\]
Behaviors compose

\[
\text{time} :: \text{Behavior}_\tau \\
\llbracket \text{time} \rrbracket = \text{id}
\]

\[
\text{pure} :: \alpha \rightarrow \text{Behavior}_\alpha \\
\llbracket \text{pure}\ a \rrbracket = \lambda t \rightarrow a \\
= \text{pure}\ a
\]

\[
(\langle\star\rangle) :: \text{Behavior}_\alpha \rightarrow \beta \rightarrow \text{Behavior}_\alpha \rightarrow \text{Behavior}_\beta \\
\llbracket \text{fs} \langle\star\rangle \text{as} \rrbracket = \lambda t \rightarrow ((\llbracket\text{fs}\rrbracket\ t) (\llbracket\text{as}\rrbracket\ t)) \\
= \llbracket\text{fs}\rrbracket \langle\star\rangle \llbracket\text{as}\rrbracket
\]
Events compose

\[ \emptyset :: Event_\alpha \]
\[ \semantics{\emptyset} = [] \]

\[ (\oplus) :: Event_\alpha \rightarrow Event_\alpha \rightarrow Event_\alpha \]
\[ \semantics{e \oplus e'} = \semantics{e} \text{ 'merge' } \semantics{e'} \]

\[ fmap n :: (\alpha \rightarrow \beta) \rightarrow Event_\alpha \rightarrow Event_\beta \]
\[ \semantics{fmap f e} = \text{map } \lambda(t, a) \rightarrow (t, f a) \semantics{e} \]
\[ = fmap (fmap f) \semantics{e} \]
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:

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

Catch: We need lazy expiration times.
Generalize/simplify – Reactive values

\[ \text{Behavior}_\alpha = (\mathcal{T} \to \alpha) \times (\hat{\mathcal{T}} \times \text{Behavior}_\alpha) \]

Generalize:

\[ \text{Reactive}_\beta = \beta \times (\hat{\mathcal{T}} \times \text{Reactive}_\beta) \quad \text{-- discrete reactive} \]

And specialize:

\[ [\text{TFun}_\alpha] = \mathcal{T} \to \alpha \quad \text{-- continuous non-reactive} \]

\[ \text{Behavior} = \text{Reactive} \circ \text{TFun} \]

This representation provides \textit{Functor} and \textit{Applicative} instances.
**TFun constant-folds**

\[
\text{data } Fun \ t \ a = K \ a \mid Fun \ (t \to a) \\
[Fun \ t \ a] = t \to a
\]

\[
\text{data } TFun = Fun \ T \\
[K \ a] = \text{const } a \\
[Fun \ f] = f
\]

\[
\text{instance } \text{Functor} \ (TFun \ t) \ \text{where} \\
fmap \ f \ (K \ a) = K \ (f \ a) \\
fmap \ f \ (Fun \ g) = Fun \ (f \circ g)
\]

etc
Generalize/simplify – Future values

\[ \text{Reactive}_\beta = \beta \times (\widehat{T} \times \text{Reactive}_\beta) \]

becomes

\[ \text{Future}_\gamma = \widehat{T} \times \gamma \]

\[ \text{Reactive}_\beta = \beta \times \text{Future}_{\text{Reactive}_\beta} \]
Events are future reactives

\[
Reactive_\beta = \beta \times Future_{Reactive_\beta}
\]

becomes

\[
Event_\alpha = Future_{Reactive_\alpha}
\]

\[
Reactive_\beta = \beta \times Event_\beta
\]
Summarizing

\[
\begin{align*}
\text{Future}_\gamma &= \hat{T} \times \gamma \\
\text{Event}_\alpha &= \text{Future}_{\text{Reactive}_\alpha} \\
\text{Reactive}_\beta &= \beta \times \text{Event}_\beta \\
\text{Behavior} &= \text{Reactive} \circ \text{TFun} \\
\text{data } \text{Fun } t a &= K a \mid \text{Fun } (t \to a)
\end{align*}
\]
Future values are mostly easy

newtype Future α = Fut ( TInt , α )

deriving ( Functor, Applicative, Monad )

For Applicative and Monad, the TInt monoid uses max and $-\infty$. 
What about Monoid?
A first try:

\[(\oplus)\] chooses the earlier of two futures:

\[
\text{instance } \texttt{Monoid} \ (\texttt{Future } \alpha) \ \texttt{where}
\]

\[
\emptyset = \texttt{Fut} \left( \infty, \bot \right)
\]

\[
u_a @ (\texttt{Fut} \left( \hat{t}_a, \_ \right)) \oplus u_b @ (\texttt{Fut} \left( \hat{t}_b, \_ \right)) =
\]

\[
\text{if } \hat{t}_a \leq \hat{t}_b \ \text{then } u_a \ \text{else } u_b
\]

We’ll have to compare future times without knowing both fully. Even so, there’s a problem ...
First try:

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

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. \textit{Oops}.

Solution:

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

Can be optimized.
What are future times?

\[
\text{type } \hat{T} = \text{Max} \ (\text{AddBounds} \ (\text{Improving} \ T))
\]

- \(\text{Max}\) monoid for derived \text{Applicative Future} (and \text{Monad})
- \(\text{AddBounds}\) for the \text{Future} and \(\text{Max}\) monoids
- \(\text{Improving}\) 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* :: $\text{Improving}_a \rightarrow a$
- *compare*$_I$ :: $\text{Improving}_a \rightarrow a \rightarrow \text{Ordering}$
- *min* :: $\text{Improving}_a \rightarrow \text{Improving}_a \rightarrow \text{Improving}_a$
- $(\leq)$ :: $\text{Improving}_a \rightarrow \text{Improving}_a \rightarrow \text{Bool}$

Puzzle: Can *exact* and *compare*$_I$ implement *min* and $(\leq)$? If so,

```haskell
data Improving a =
    Imp { exact :: a, compareI :: a \rightarrow \text{Ordering} }
```
Comparing improving values – dilemma

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

\[
ab = compare_1 \hat{t}_a \ (\text{exact } \hat{t}_b) \not\equiv GT \\
ba = compare_1 \hat{t}_b \ (\text{exact } \hat{t}_a) \not\equiv LT
\]

Which to try first? We can’t know beforehand.
Same answer when defined, so try in parallel and take first answer

\[ ab \ 'unamb' \ ba \]

Referentially transparent? Yes:

\[ ab \ 'unamb' \ ba \equiv ab \sqcup ba \]

Crucial: \( ab \) and \( ba \) agree when defined.
unamb is handy

parCommute op x y = (x ‘op‘ y) ‘unamb‘ (y ‘op‘ x)

por = parCommute (∨)
pand = parCommute (∧)

-- handy with unamb
assuming :: Bool → a → a
assuming True a = a
assuming False _ = ⊥
Symmetric short-circuiting

\[
\text{parAnnihilator } \text{op} \ a \times y = \\
\quad \text{assuming } (x \equiv a) \ a \text{‘unamb‘} \\
\quad \text{assuming } (y \equiv a) \ a \text{‘unamb‘} \\
\quad (x \text{‘op‘} y)
\]

\[
\text{por} = \text{parAnnihilator } (\lor) \ True \\
\text{pand} = \text{parAnnihilator } (\land) \ False \\
\text{pmul} = \text{parAnnihilator } (\times) \ 0 \\
\text{pmin} = \text{parAnnihilator } \text{min} \ \text{minBound} \\
\text{pmax} = \text{parAnnihilator } \text{max} \ \text{maxBound}
\]
**min** is simple – almost

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

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

Similarly,

```

```
```

Too strict. Consider exact \( \hat{t}_a < t < \text{exact} \hat{t}_b \). Easy fix, via unamb:

```
```
```
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.