Push-Pull Signal-Function Functional Reactive Programming

Edward Amsden

Rochester Institute of Technology

IFL 2012
August 29, 2012

R·I·T

B. Thomas Golisano
College of Computing and Information Sciences

Functional Reactive Programming (FRP)

- *Functional* First-class and higher-order functions.
- *Reactive* Behavior changes in response to temporal inputs.
- Basic abstractions:
    - Signals - Functions of time[1].
    - Events - Sequences of temporally ordered and labeled discrete values.

[1]Often also called behaviors.

Signal-Function FRP

- Signal functions are transformers of events and signals.
- Signal functions are first class in Signal-Function FRP.
- Signals and events are not first-class in Signal Function FRP.
- This approach is more composable than first-class signals and events (since input may be transformed)[2].

[2] Courtney and Elliott, "Genuinely functional user interfaces".

Push vs. Pull Evaluation

- When to evaluate what?
- Pull evaluation ("demand-driven"):
  Evaluate when output is needed.
- Push evaluation ("data-driven"):
  Evaluate when input is available.
- Ideally, an FRP system uses push for events and pull for signals[3].

[3] Elliott, "Push-pull functional reactive programming".

Separating Events and Signals

```
class (Category a) => Arrow a where
  arr :: (b -> c) -> a b c
  ...
```

- Traditional Signal-Function FRP encodes signal functions as a Haskell Arrow.

```
data SF a b = ...

instance Arrow SF where
  ...
```

Separating Events and Signals

- Necessary to lift any function to a corresponding signal function, without input/output type annotation
- Events encoded as optional signals:

  ```
  type Event a = Maybe a
  ```

- Combined signals and events encoded as one signal with pair type:

Separating Events and Signals

```
class (Category a) => Arrow a where
  ...
  first :: a b c -> a (b, d) (c, d)
  ...

capture :: SF (a, Event b) (Event a)

ghci> :t first . arr
first . arr
    :: (Arrow a) => (b -> c) -> a (b, d) (c, d)
```

Type Signal Functions with Signal Vectors

- Described by Sculthorpe[4]
- Describe separation of individual signals and events
- In Haskell (with `-XEmptyDataDecls`):

```
data SVEmpty
data SVSignal a
data SVEvent a
data SVAppend svl svr
type :^: svl svr = SVAppend svl svr
```

- Would like to use `-XDataKinds` but it's not working well yet.

[4] Schulthorpe, "Towards Safe and Efficient Functional Reactive Programming".

Type Signal Functions with Signal Vectors

- Two combinator examples:
- Lifting a pure function to transform a signal:

```
pureSignal ::      (a -> b)
                -> SF (SVSignal a) (SVSignal b)
```

- Passing through input on the right:

```
first ::     SF svIn svOut
          -> SF (svIn :^: svRight) (svOut :^: svRight)
```

- Composing these leads to:

```
ghci> :t first . pureSignal
first . pureSignal
    :: (a -> b) -> SF (SVSignal a :^: sv)
                      (SVSignal b :^: sv)
```

Partial Representations of Signal Vectors

- To evaluate signals and events differently we must represent them separately.
- Signal vectors enable this by distinguishing them in the types.
- We can construct several representations of a signal vector.
  - Represent signal leaves, event leaves, or both.
  - Represent one leaf, a subset of leaves, or all applicable leaves.
  - Transform the type at the leaf, or don't.

Signal Representation

- Represent an entire signal sample
  (for initializing a signal function at time zero):

```
data SVSample where
    SVSample      ::    a -> SVSample (SVSignal a)
    SVSampleEvent ::    SVSample (SVEvent a)
    SVSampleEmpty ::    SVSample SVEmpty
    SVSampleBoth  ::    SVSample svLeft
                     -> SVSample svRight
                     -> SVSample (svLeft :^: svRight)
```

Signal Representation

- Represent a signal delta
  (replacement values for a subset of signals):

```
data SVDelta where
     SVDelta         ::   a -> SVDelta (SVSignal a)
     SVDeltaNothing  ::   SVDelta sv
     SVDeltaBoth     ::   SVDelta svLeft
                       -> SVDelta svRight
                       -> SVDelta (svLeft :^: svRight)
```

Event Representation

- Represent an event occurrence:

```
data SVOccurrence where
     SVOccurrence ::   a -> SVOccurrence (SVEvent a)
     SVOccLeft    ::   SVOccurrence svLeft
                    -> SVOccurrence
                         (svLeft :^: svRight)
     SVOccRight   ::   SVOccurrence svRight
                    -> SVOccurrence
                         (svLeft :^: svRight)
```

Signal Function Representation

- Separate continuations for time advancement and event occurrences:

```
data Initialized
data NonInitialized

data SF init svIn svOut where
  SF ::   (SVSample svIn
            -> (SVSample svOut,
                 SF Initialized svIn svOut))
        -> SF NonInitialized svIn svOut
...
```

Signal Function Representation

- Separate continuations for time advancement and event occurrences.

```
...
  SFInit ::   (Double
               -> SVDelta svIn
               -> (SVDelta svOut,
                   [SVOccurrence svOut],
                   SF Initialized svIn svOut))
           -> (SVOccurrence svIn
               -> ([SVOccurrence svOut],
                   SF Initialized svIn svOut)
           -> SF Initialized svIn svOut
```

Evaluation

- Yampa/AFRP: Supply SF and input/output actions to an evaluation loop (`reactimate`).
- Here: initialize an evaluation state with:
  - A signal function.
  - Initial values for all input signals.
  - Handlers for all outputs.
- Then, the evaluation monad carries this state and provides the actions:
  - `push` Push an event (which will be immediately reacted to).
  - `update` Update the value of an input signal (with no immediate effect).
  - `step` Update the time and evaluate new values of signals.

Further Work

- Dynamic collections: Dynamically switch between collections of signal functions.
- Semantics/correctness proof (especially for event merging).
- Time-independence optimization.

Questions?

Time-independence optimization

- Inefficiency: running every time continuation every sample step.
- This happens even if the signal delta is empty (no change)
- Currently this is necessary because the *time* delta might cause some output.
- Optimization:
  - Mark time-independent SFs using a separate constructor.
  - "Smart" composition-routing to make a composite SF time-independent if all of its components are.
  - Time-independent time continuation has no time input, only delta.
  - Only called if non-empty delta.

Event Merging (Semantics Idea)

- Push-based events are not tied to samples.
- We thus know only what the last sample time was for an event, not the actual time.
- Can we hide event merging in this loss of precision?