Monadic Functional Reactive Programming

Haskell Symposium '13

Atze van der Ploeg

Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

23 September 2013

What is Functional Reactive Programming (FRP)?

Reactive program

Engages in a dialogue with its environment, responding to events.
Examples: IDE, Spreadsheet, Anything with a GUI

Reactive programming means...

Dealing with external events which may occur in in any order.

Traditional approaches:

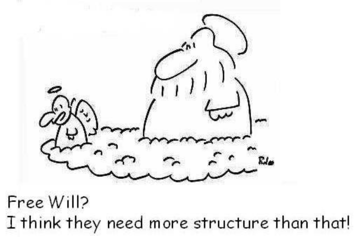
	Approach	*Downside*
■	Concurrency	non-determinism
■	Callbacks (observer pattern)	inversion of control
■	I/O multiplexing	non-composable

Functional reactive programming

Umbrella term for functional ways of reactive programming, which are deterministic, composable and without inversion of control.

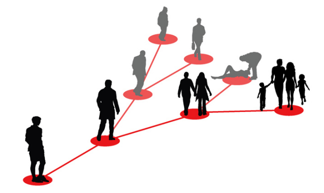
Two schools of Functional Reactive Programming

Single future



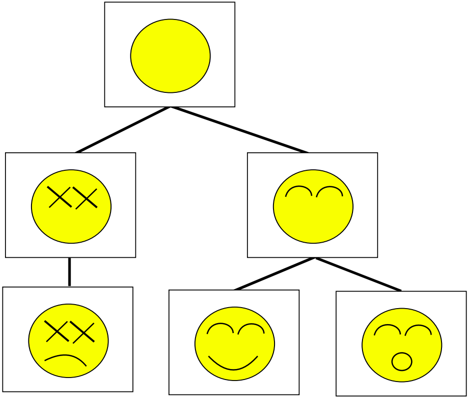
Examples: Fran, Reactive, FrTime, Scala.React

Time branching



Example: Yampa

Time branching



“Freeze” a running FRP expression, convert it to a value.

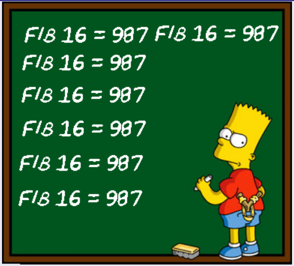
Example usages:

- Duplicate drawing in drawing program into two tabs
- Undo
- What-if?

Efficiency considerations: We do not want ...

Polling

Redundant re-evaluations



Single future evaluation mechanisms

	Evaluation mechanism
Reactive	Push-pull
FrTime, Scala.React	Self-adjusting computation

Both efficient: prevent polling & redundant re-evaluations

Time branching evaluation mechanisms

	Evaluation mechanism
Yampa	Continuation based + GADT-based optimizations

Not as efficient as single future mechanisms:

- Does not prevent polling
- Prevents a large class, but not all, redundant re-evaluations

Monadic FRP

We introduce Monadic Functional Reactive Programming.

- A new programming interface for time-branching FRP
- This talk: Efficient evaluation mechanism for time-branching FRP:
  Continuation based + event requests
  Prevents:
  - Polling
  - Redundant re-evaluations

Reactive computations

Reactive computation

A monadic computation which may require the occurrence of external events to continue.

All Monadic FRP expressions are built using:

- Basic events.

  Example: $mouseDown :: React\ GUIEv\ MBtn$

- Sequential composition:

  $(\ggg\!=) :: React\ e\ a \rightarrow (a \rightarrow React\ e\ b) \rightarrow React\ e\ b$

- Parallel composition:

  $parR :: React\ e\ a \rightarrow React\ e\ b \rightarrow React\ e\ (React\ e\ a, React\ e\ b)$

  Notice: $flip\ parR \equiv parR$

Signal computation

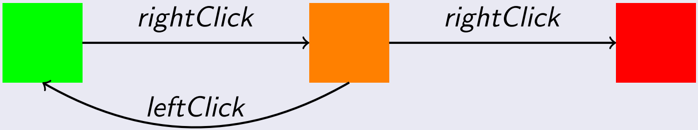
Signal computation

A reactive computation that may also emit values.
Defined in terms of reactive computations.

instance *Monad* (*Sig e f*) **where** ...
$waitFor :: React\ e\ r \to Sig\ e\ f\ r$
$emit \quad :: f \to Sig\ e\ f\ ()$

Signal computation example

Example




```
traffic :: Sig GUIEv Color Void
traffic =
  do emit green
     waitFor rightClick
     emit orange
     (r, _) ← waitFor (rightClick `parR` leftClick)
     case r of
       Done _ → emit red ≫ waitFor hellFreezesOver
       _      → traffic
```

Idea for efficient evaluation

Continuation based evaluation

$$\textbf{data}\ \mathit{React}\ e\ a = \mathit{Done}\ a \mid \mathit{Later}\ (e \to \mathit{React}\ e\ a)$$

We do not know if an event will have any effect.

Continuation based + requests

$$\textbf{data}\ \mathit{React}\ e\ a = \mathit{Done}\ a \mid \mathit{Later}\ (\mathit{Requests}\ e)\ (e \to \mathit{React}\ e\ a)$$

Add which events the reactive computation requests
(i.e. is interested in).

Interpreting Monadic FRP expressions

Interpret Reactive computation

Handle event requests in some Monad (for example IO):

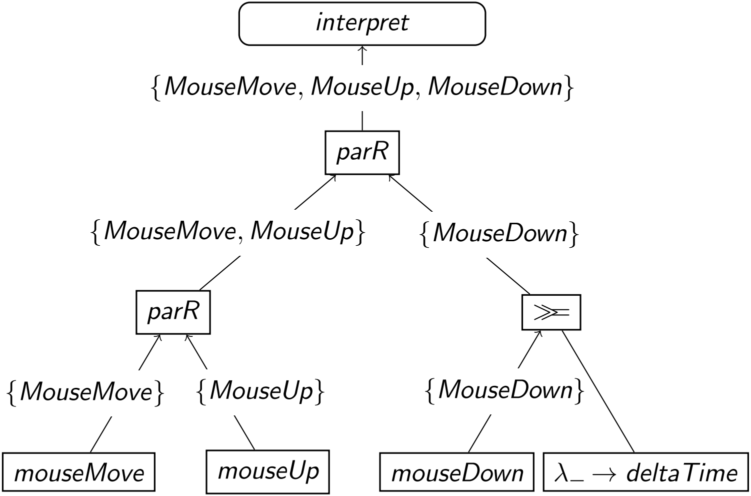
$$
\begin{aligned}
&\mathit{interpret} :: \mathit{Monad}\ m \Rightarrow (\mathit{Requests}\ e \to m\ e)\\
&\qquad\qquad \to \mathit{React}\ e\ a \to m\ a\\
&\mathit{interpret}\ p\ (\mathit{Done}\ a) \quad = \mathit{return}\ a\\
&\mathit{interpret}\ p\ (\mathit{Later}\ r\ c) = p\ r \ggg\!= \mathit{interpret}\ p \circ c
\end{aligned}
$$

Interpret Signal computations

Also handle emissions.

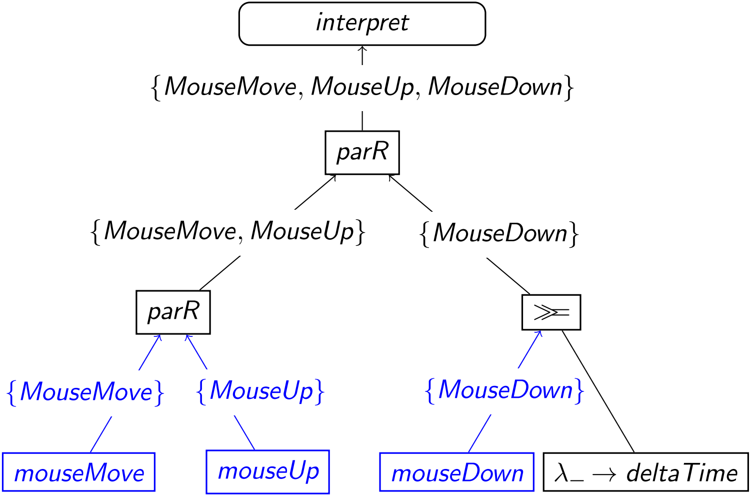
$$
\begin{aligned}
&\mathit{interpretSig} :: \mathit{Monad}\ m \Rightarrow (\mathit{Requests}\ e \to m\ e)\\
&\qquad\qquad\quad \to (a \to m\ ()) \to \mathit{Sig}\ e\ a\ b \to m\ b
\end{aligned}
$$

Event requests propagate upwards



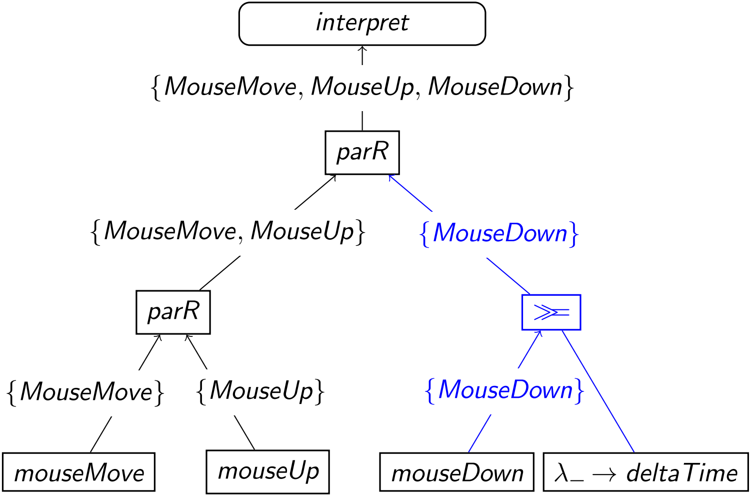
$$first\ (first\ mouseMove\ mouseUp)\ (mouseDown \gg deltaTime)$$

Event requests propagate upwards



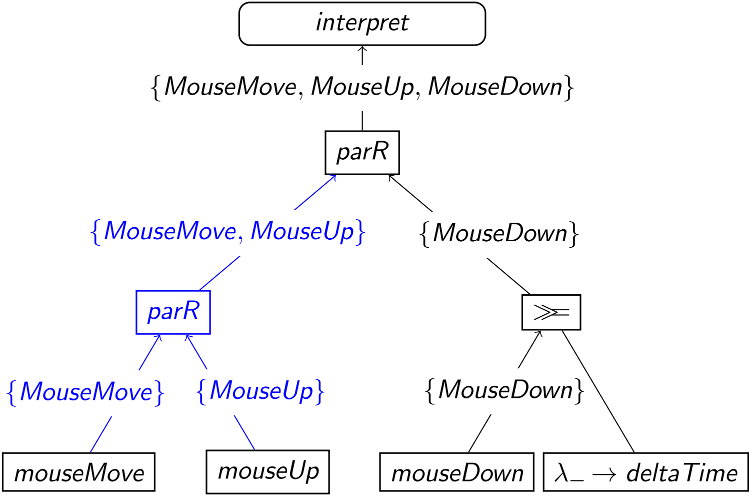
$$\mathit{first}\ (\mathit{first}\ \mathit{mouseMove}\ \mathit{mouseUp})\ (\mathit{mouseDown} \gg \mathit{deltaTime})$$

Event requests propagate upwards



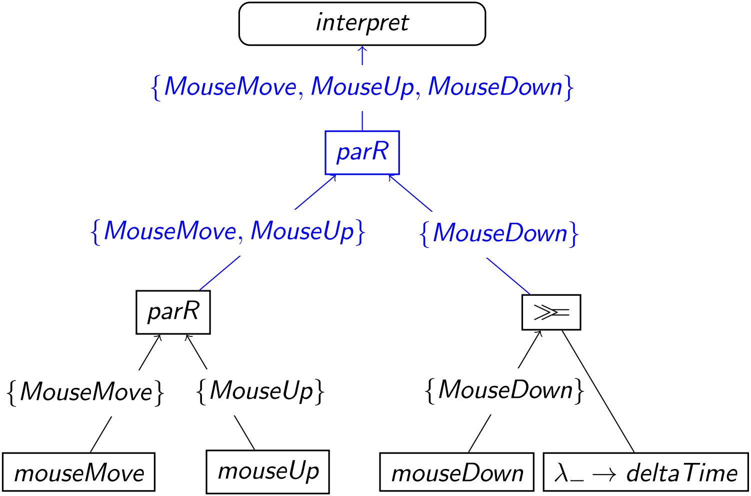
first (*first* *mouseMove* *mouseUp*) (*mouseDown* $\gg$ *deltaTime*)

Event requests propagate upwards



$$\textit{first}\ (\textit{first}\ \textit{mouseMove}\ \textit{mouseUp})\ (\textit{mouseDown} \gg \textit{deltaTime})$$

Event requests propagate upwards



$$first\ (first\ mouseMove\ mouseUp)\ (mouseDown \gg deltaTime)$$

Set of event requests → No polling

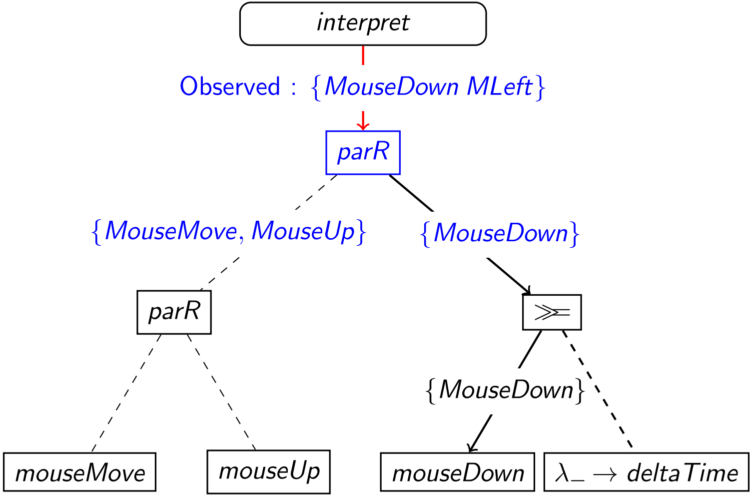
The reactive computation requests an events from the set:

$$\{MouseMove, MouseUp, MouseDown\}$$

This set is passed to the function p, which handles event requests. p can block until such an event arrives.

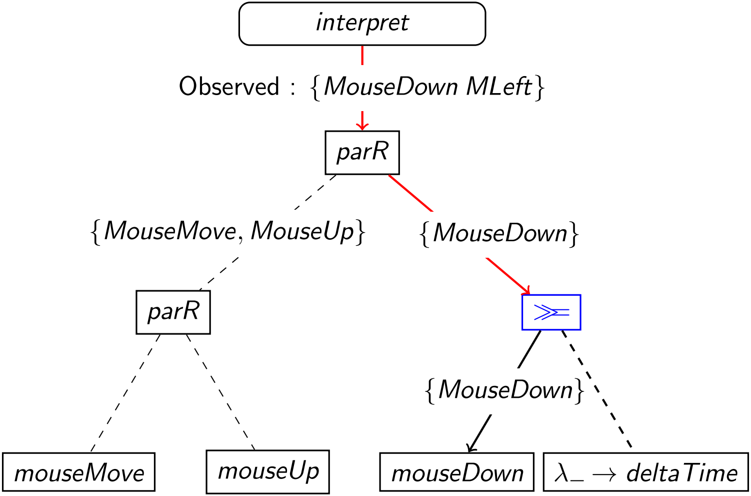
$$\begin{aligned}
&interpret :: Monad\ m \Rightarrow (Requests\ e \to m\ e) \\
&\qquad\qquad\quad \to React\ e\ a \to m\ a \\
&interpret\ p\ (Done\ a) \quad = return\ a \\
&interpret\ p\ (Later\ r\ c) = p\ r \gg\!= interpret\ p \circ c
\end{aligned}$$

Events propagate downwards: no redundant re-evalutions



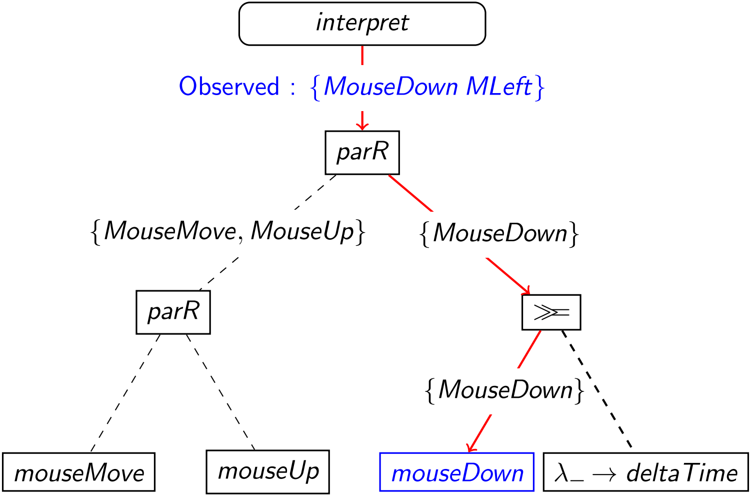
parR (parR mouseMove mouseUp) (mouseDown ≫ deltaTime)

Events propagate downwards: no redundant re-evalutions



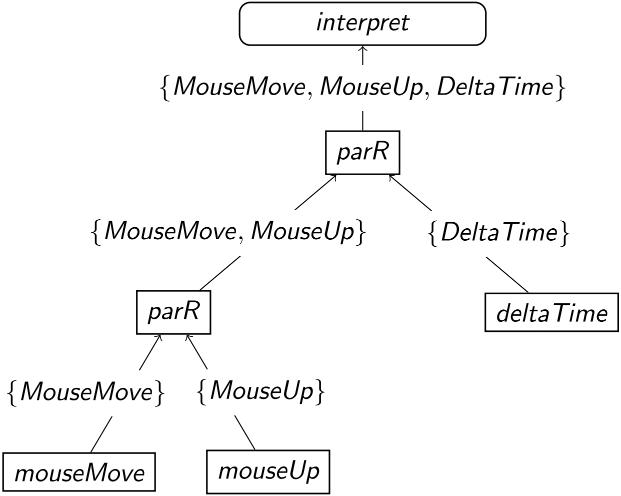
parR (*parR* *mouseMove* *mouseUp*) (*mouseDown* $\gg$ *deltaTime*)

Events propagate downwards: no redundant re-evalutions



$$parR\ (parR\ mouseMove\ mouseUp)\ (mouseDown \gg deltaTime)$$

Next state



$$\mathit{first}\ (\mathit{first}\ \mathit{mouseMove}\ \mathit{mouseUp})\ \mathit{deltaTime}$$

Evaluation mechanism summary

Event requests $\rightarrow$

- Know which events to wait for
  $\rightarrow$ No polling
- Know the event requests of each subexpression
  $\rightarrow$ No redundant re-evaluations

Future work

Implementation of evaluation mechanism currently relies on *closed* union of possible basic events.
Downsides:

- requires explicit memoization for reactive level sharing.
- reactive level recursion is (very) problematic.

Continuation based + GADT-based optimizations mechanism (Yampa) does not have these problems.

Possible solution: Use open union instead.

In the paper...

- Monadic FRP programming interface: sequencing phases more primitive than other approaches.
- Drawing program example.
- Comparison with other FRP programming interfaces & evaluation mechanisms.
- Exact time semantics (*sleep* 1.0 occurs strictly before *sleep* 1.1).
- ... and all the details.

Conclusion

Purely functional evaluation mechanism for time-branching FRP:

Continuation based + event requests

Benefits:

- No polling
- No redundant re-evaluations.

Hackage package: `drClickOn`

Question: Purity of other mechanisms?

Single future

Evaluation mechanism	*Pure?*
Push-pull	Lazy IO-like
Self-adjusting computation	No, mutable data

Time branching

Evaluation mechanism	*Pure?*
Continuation based + GADT based optimizations	Yes
Continuation based + Event requests	Yes

Question: Cost of requests?

- Bounded by number of basic events m.
- $< mn$, where n is the number of nodes in the expression tree.
- Sets along expression tree very structured
  $\rightarrow$ more efficient representation may exist.

Question: Comparison with Fudgets-style stream processors?

- In Fudgets: Multiple inputs (streams) are joined into one input $\rightarrow$ Lose information on what is new.
- In Fudgets: Routing is static $\rightarrow$ we do not know which subset of static events we are interested in at runtime.
  In Monadic FRP: routing is dynamic.

Question: Which class of redundant computations?

"Dynamic Optimization for Functional Reactive Programming using GADTs", Hendrik Nilsson, ICFP '05.
Covered by GADT based optimizations & event requests:

- Implicitly memoize SFs of type $SF\ (Event\ a)\ b$ for $NoEvent$
- Implicitly memoize stateful SFs using $sfsScan :: (c \to a \to Maybe\ (c, b)) \to c \to SF\ a\ b$ Examples: $filter, scan, map$

Not covered by GADT based optimizations, Covered by event requests:

- Stateful custom signal function.
- Stateless signal functions of another form.

In Arrowized FRP, each signal function must emit on each time step, even if output did not change.
Not true in Monadic FRP.

Question: Continuous vs. discrete time?

- Monadic FRP: No strong distinction between continuous and discrete.
- Continuous time signal computations depend on event *DeltaTime*.
- *DeltaTime* is conceptually infinitesimal small change in time.
- This may give continuous time semantics, see *"Hyperstream processing systems: nonstandard modeling of continuous-time signals"* K. Suenaga, H. Sekine, and I. Hasuo. POPL '13

Question: Time branching and Logic?

- Single future semantics FRP $\rightarrow$ Linear Temporal Logic. See: *"LTL types FRP: Linear-time Temporal Logic Propositions as Types, Proofs as Functional Reactive Programs"* Alan Jeffrey. PLPV '12.
- Time branching FRP $\rightarrow$ Computation Tree Logic ?