First-Class Signals for Functional Reactive Programming

Wolfgang Jeltsch

TTÜ Küberneetika Instituut

Teooriaseminar
October 13, 2011

Overview

Introduction

FRP concepts

Generators

Memoization

Start time consistency

Overview

Introduction

FRP concepts

Generators

Memoization

Start time consistency

Functional Reactive Programming

- the ideal reactive system:
  - continuous change
  - immediate, atomic reactions on events
- not reflected by imperative implementations:
  - discretization visible
  - inconsistent intermediate states visible
- programmer confronted with technical details:
  - polling loops
  - event handlers
- goal of functional programming:

  problem description instead of execution plan
- Functional Reactive Programming (FRP):

  applying this principle to reactive systems

Implementations

- two ways of implementing FRP:

pull-based	system state is repeatedly recomputed
push-based	state changes are triggered by events

- many Haskell EDSLs:
  - pull-based:
    - Fran
    - Yampa

    etc.
  - push-based:
    - FranTk
    - Reactive
    - Grapefruit

    etc.
- EDSLs for other programming languages (all push-based):

Java	Frappé
Scheme	FrTime
JavaScript	Flapjax

Grapefruit

- originally geared towards GUI programming
- push-based, because change is rare in GUIs
- problem with existing push-based implementations:
  - no first-class descriptions of temporal behavior
  - performance problems
- a new implementation for solving these issues

Overview

Signals

- the heart of FRP
- describe temporal behavior
- three flavors:

discrete: values associated with discrete times:

$$[\![\mathit{DSignal}]\!]\ \alpha \approx [(\mathit{Time}, \alpha)]$$

continuous: arbitrary time-varying values:

$$[\![\mathit{CSignal}]\!]\ \alpha \approx \mathit{Time} \to \alpha$$

segmented: step functions over time:

$$[\![\mathit{SSignal}]\!]\ \alpha = (\alpha, [\![\mathit{DSignal}]\!]\ \alpha)$$

Examples of signals

discrete incoming network packets:

$$\mathit{DSignal}\ \mathit{Packet}$$

continuous time since the program has started:

$$\mathit{CSignal}\ \mathit{DiffTime}$$

segmented amount of network traffic:

$$\mathit{SSignal}\ \mathit{Integer}$$

Signal combinators

- functions for constructing signals
- some examples:

$$\begin{aligned} \mathit{union} &:: \mathit{DSignal}\ \alpha \to \mathit{DSignal}\ \alpha \to \mathit{DSignal}\ \alpha \\ \mathit{filter} &:: (\alpha \to \mathit{Bool}) \to \mathit{DSignal}\ \alpha \to \mathit{DSignal}\ \alpha \\ \mathit{scanl} &:: (\beta \to \alpha \to \beta) \to \beta \to \mathit{DSignal}\ \alpha \\ &\to \mathit{SSignal}\ \beta \end{aligned}$$

- application of these combinators:

$$\begin{aligned} &\ddot{p} :: \mathit{DSignal}\ \mathit{Packet} \\ &\ddot{p} = \mathit{union}\ \ddot{p}_{\mathit{In}}\ \ddot{p}_{\mathit{Out}} \end{aligned}$$

$$\begin{aligned} &\ddot{p}_{\mathit{TCP}} :: \mathit{DSignal}\ \mathit{Packet} \\ &\ddot{p}_{\mathit{TCP}} = \mathit{filter}\ \mathit{isTCPPacket}\ \ddot{p} \end{aligned}$$

$$\begin{aligned} &\bar{v} :: \mathit{SSignal}\ \mathit{Integer} \\ &\bar{v} = \mathit{scanl}\ (\lambda v\ p \to v + \mathit{size}\ p)\ 0\ \ddot{p} \end{aligned}$$

Switching

- class *Signal* of all signal types
- switching combinator:

  $$\mathit{switch} :: (\mathit{Signal}\ \sigma) \Rightarrow \mathit{SSignal}\ (\sigma\ \alpha) \rightarrow \sigma\ \alpha$$
- possible application:
  - two segmented signals that represent amount of incoming and outgoing traffic:

    $$\bar{v}_{\mathit{In}}, \bar{v}_{\mathit{Out}} :: \mathit{SSignal}\ \mathit{Integer}$$
  - segmented signal that toggles between these two, depending on user selection:

    $$\bar{\bar{v}} :: \mathit{SSignal}\ (\mathit{SSignal}\ \mathit{Integer})$$
  - switching creates a signal that always gives the respective amount of traffic:

    $$\bar{v}_{\mathit{Sel}} :: \mathit{SSignal}\ \mathit{Integer}$$
    $$\bar{v}_{\mathit{Sel}} = \mathit{switch}\ \bar{\bar{v}}$$
  - $\bar{v}_{\mathit{Sel}}$ used as the input of a display component

Overview

Introduction

FRP concepts

Generators

Memoization

Start time consistency

A straightforward push-based implementation

- updates shall be event-driven
- signal consumers register event handlers
- discrete signal is registration action
  (which yields unregistration action):

  $$\textbf{type}\ \mathit{Handler}\ \alpha = \alpha \to \mathit{IO}\ ()$$

  $$\textbf{type}\ \mathit{DSignal}\ \alpha = \mathit{Handler}\ \alpha \to \mathit{IO}\ (\mathit{IO}\ ())$$

- *SSignal* implementation directly mirrors the semantics:

  $$\textbf{type}\ \mathit{SSignal}\ \alpha = (\alpha, \mathit{DSignal}\ \alpha)$$

Implementation of *scanl*

$$
\begin{array}{l}
\mathit{scanl} :: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow \mathit{DSignal}\ \alpha \rightarrow \mathit{SSignal}\ \beta \\
\mathit{scanl}\ f\ y_0\ \ddot{x} = (y_0, \ddot{y})\ \textbf{where} \\
\quad \ddot{y} = \lambda h \rightarrow \textbf{do} \\
\qquad\qquad\qquad\qquad \vec{y} \leftarrow \mathit{newIORef}\ y_0 \\
\qquad\qquad\qquad\qquad \ddot{x}\ (\lambda x \rightarrow \textbf{do} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad y \leftarrow \mathit{readIORef}\ \vec{y} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad y' = f\ y\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{writeIORef}\ \vec{y}\ y' \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad h\ y')
\end{array}
$$

Generators, not signals

- registration actions executed once per consumer
- when using *scanl*, every consumer
  - creates a mutable variable, initialized at consumption time
  - registers a handler that updates this variable
- two problems:
  1. duplication of computations
  2. signal values depending on consumption time
- intuition:
  - values of signal types are in fact generators
  - generator yields a new signal when consumed
  - signals are not first-class anymore

Overview

Introduction

FRP concepts

Generators

Memoization

Start time consistency

Using native memoization

- Haskell caches computed parts of a data structure if a variable is bound to the structure
- problem:

  values of *DSignal* do not contain event values
- changing the data structure:

  type *DSignal* $\alpha = [(\mathit{Time}, \alpha)]$
- event streams must be interleaved when computing signal unions:

$$\begin{aligned} \mathit{union}\ ((t_1, x_1) : \ddot{x}_1)\ ((t_2, x_2) : \ddot{x}_2)\ &|\ t_1 < t_2 = \cdots \\ &|\ t_1 \equiv t_2 = \cdots \\ &|\ t_1 > t_2 = \cdots \end{aligned}$$

- problem:

  comparison of occurrence times must succeed when the first event occurs
- our solution:

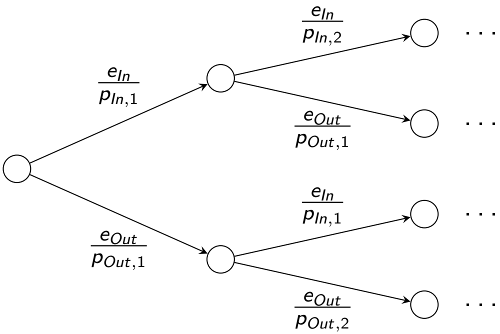
  delegate event ordering to the consumers

Representing discrete signals by vistas

- vista covers every possible event stream interleaving
- future behavior depends on which external event source fires next:

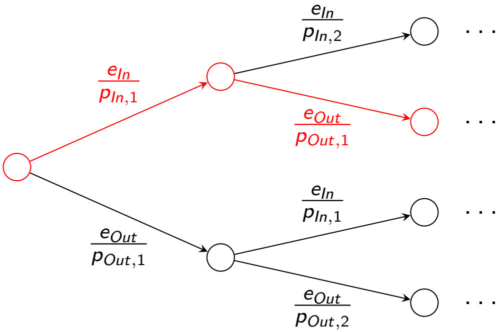
$$\begin{aligned} &\textbf{type}\ \mathit{Vista}\ \ \alpha = \mathit{Map}\ \mathit{EventSrc}\ (\mathit{Variant}\ \alpha) \\ &\textbf{type}\ \mathit{Variant}\ \alpha = (\alpha, \mathit{Vista}\ \alpha) \end{aligned}$$

- vista for *union* $\ddot{p}_{In}$ $\ddot{p}_{Out}$:



Consuming vistas

- consumer knows about the order in which event sources fire
- evaluates only the relevant path:



Implementation of combinators

- functional representation of discrete signals leads to functional implementations of combinators
- implementation of *scanl*:

$$\begin{aligned} &\mathit{scanl} :: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow \mathit{DSignal}\ \alpha \\ &\qquad \rightarrow \mathit{SSignal}\ \beta \\ &\mathit{scanl}\ f\ y_0\ \ddot{x} = (y_0, a\ y_0\ \ddot{x})\ \textbf{where} \\ &\quad a\ y = \mathit{fmap}\ (\lambda(x, \ddot{x}) \rightarrow \textbf{let} \\ &\qquad\qquad\qquad\qquad\qquad y' = f\ y\ x \\ &\qquad\qquad\qquad\qquad\quad \textbf{in}\ (y', a\ y'\ \ddot{x})) \end{aligned}$$

- problem with *filter*:

  removing nodes would destroy structure of the vista
- solution:

  make event values optional
- modified *Variant* type:

$$\textbf{type}\ \mathit{Variant}\ \alpha = (\mathit{Maybe}\ \alpha, \mathit{DSignal}\ \alpha)$$

Overview

Introduction

FRP concepts

Generators

Memoization

Start time consistency

Fixing start times

- technique inspired by Haskell's *ST* monad
- signal types get an extra (phantom) type parameter that represents signal start times
- signal combinators enforce start time equality:

$$\begin{aligned} \mathit{union} &:: \mathit{DSignal}\ t_0\ \alpha \to \mathit{DSignal}\ t_0\ \alpha \\ &\to \mathit{DSignal}\ t_0\ \alpha \\ \mathit{scanl} &:: (\beta \to \alpha \to \beta) \to \beta \to \mathit{DSignal}\ t_0\ \alpha \\ &\to \mathit{SSignal}\ t_0\ \beta \end{aligned}$$

- actions for producing and consuming signals have a parameter representing execution time:

$$\textbf{newtype}\ \mathit{Reactive}\ t_0\ \alpha = \mathit{Reactive}\ (\mathit{IO}\ \alpha)$$

- signal production and consumption enforce start time equality
- conversion to *IO* uses universal quantification:

$$\mathit{toIO} :: (\forall t_0 . \mathit{Reactive}\ t_0\ \alpha) \to \mathit{IO}\ \alpha$$

Safe switching

- safe switching combinator:

$$\mathit{switch} :: (\mathit{Signal}\ \sigma) \Rightarrow \mathit{SSignal}\ t_0\ (\forall t. \sigma\ t\ \alpha) \to \sigma\ t_0\ \alpha$$

- switches only to signals that don't depend on external events:
  - empty discrete signal
  - constant continuous signals
  - constant segmented signals
- useless
- idea:

  switching between signal functions
  instead of signals

Signal functions to the rescue

- functions on signals with identical start time:

$$\mathit{SigFun}\ t_0\ (\sigma_1\ \text{`}\mathit{Of}\text{`}\ \alpha_1 \mapsto \cdots \mapsto \sigma_n\ \text{`}\mathit{Of}\text{`}\ \alpha_n \mapsto \sigma\ \text{`}\mathit{Of}\text{`}\ \alpha)$$

- empty data types for type indices:

$$\textbf{data}\ \varphi \mapsto \varphi'$$

$$\textbf{data}\ \sigma\ \text{`}\mathit{Of}\text{`}\ \alpha$$

- *SigFun* defined as a GADT:

$$\begin{aligned}
&\textbf{data}\ \mathit{SigFun}\ t_0\ \varphi\ \textbf{where}\\
&\quad \mathit{SigFun}_0 \quad :: (\mathit{Signal}\ \sigma)\\
&\quad\quad \Rightarrow \sigma\ t_0\ \alpha\\
&\quad\quad \rightarrow \mathit{SigFun}\ t_0\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha)\\
&\quad \mathit{SigFun}_{succ} :: (\mathit{Signal}\ \sigma)\\
&\quad\quad \Rightarrow (\sigma\ t_0\ \alpha \rightarrow \mathit{SigFun}\ t_0\ \varphi')\\
&\quad\quad \rightarrow \mathit{SigFun}\ t_0\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha \mapsto \varphi')
\end{aligned}$$

Switching between signal functions

- type of the switching combinator:

$$\begin{aligned} \mathit{switch} &:: \mathit{SSignal}\ t_0\ (\forall t.\mathit{SigFun}\ t\ \varphi) \\ &\quad \to \mathit{SigFun}\ t_0\ \varphi \end{aligned}$$

- how the combinator works (conceptionally):
  - arguments of the result function are pruned to fit the segments of the argument signal (ageing)
  - each function from the argument signal is applied to its corresponding slices
  - resulting segments are concatenated

The traffic volume example again

- type of binary signal functions over a single signal type:

$$\textbf{type}\ \mathit{BinSigFun}\ t_0\ \sigma\ \alpha = \mathit{SigFun}\ t_0\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha \mapsto \sigma\ \text{`}\mathit{Of}\text{`}\ \alpha \mapsto \sigma\ \text{`}\mathit{Of}\text{`}\ \alpha)$$

- projection functions:

$$\begin{aligned}
&\pi_1, \pi_2 :: \mathit{BinSigFun}\ t_0\ \sigma\ \alpha \\
&\pi_1 = \mathit{SigFun}_{succ}\ \$\ \lambda s_1 \rightarrow \\
&\qquad \mathit{SigFun}_{succ}\ \$\ \lambda\_ \rightarrow \mathit{SigFun}_0\ s_1 \\
&\pi_2 = \mathit{SigFun}_{succ}\ \$\ \lambda\_ \rightarrow \\
&\qquad \mathit{SigFun}_{succ}\ \$\ \lambda s_2 \rightarrow \mathit{SigFun}_0\ s_2
\end{aligned}$$

- segmented signal that toggles between these functions:

$$\bar{f} :: \mathit{SSignal}\ t_0\ (\forall t.\mathit{BinSigFun}\ t\ \sigma\ \alpha)$$

- switching yields time-varying projection:

$$\begin{aligned}
&f :: \mathit{BinSigFun}\ t_0\ \sigma\ \alpha \\
&f = \mathit{switch}\ \bar{f}
\end{aligned}$$

- unpacking and applying to $\bar{v}_{In}$ and $\bar{v}_{Out}$ yields $\bar{v}_{Sel}$

First-Class Signals for Functional Reactive Programming

Wolfgang Jeltsch

TTÜ Küberneetika Instituut

Teooriaseminar
October 13, 2011