Signals, Not Generators!

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

Tenth Symposium on Trends in Functional Programming
June 2–4, 2009
Signals

- the heart of Functional Reactive Programming
- describe behavior over time
- three flavors:
  - discrete values associated with discrete times
  - continuous arbitrary time-varying values (not in this talk)
  - segmented time-varying values, changing at discrete times
- examples:
  - discrete incoming network packets
    \[ D\text{Signal} \text{ Packet} \]
  - segmented amount of network traffic
    \[ S\text{Signal} \text{ Int} \]
Signal combinators (1)

- some combinators:

  \[
  \text{union} :: \text{DSignal val} \rightarrow \text{DSignal val} \rightarrow \text{DSignal val}
  \]

  \[
  \text{scan} :: \text{accu} \rightarrow \\
  (\text{accu} \rightarrow \text{val} \rightarrow \text{accu}) \rightarrow \\
  (\text{DSignal val} \rightarrow \text{SSignal accu})
  \]

- application:

  \[
  \text{packets} :: \text{DSignal Packet} \rightarrow \\
  \text{packets} = \text{union inPackets outPackets}
  \]

  \[
  \text{amounts} :: \text{DSignal Packet} \rightarrow \text{SSignal Int} \rightarrow \\
  \text{amounts} = \text{scan 0 next where}
  \]

  \[
  \text{next amount packet} = \text{amount} + \text{size packet}
  \]
Signal combinators (2)

- switching combinator:

  \[
  \text{switch} :: (\text{Signal} \ \text{signal}) \Rightarrow \text{SSignal} (\text{signal} \ \text{val}) \rightarrow \text{signal} \ \text{val}
  \]

  \text{instance} \ \text{Signal} \ \text{DSignal} \ \text{where} \ \ldots

  \text{instance} \ \text{Signal} \ \text{SSignal} \ \text{where} \ \ldots

- application:

  showing the amount of either incoming or outgoing traffic, depending on user selection
Push-based evaluation

- event-driven updates
- signal consumers register event handlers
- typical implementation of signals:
  - **discrete** signal is registration action:
    
    \[
    \text{DSignal \ val} \cong (\text{val} \to \text{IO}()) \to \text{IO}((\text{IO}()))
    \]
  - **segmented** signal is initial value plus update signal:
    
    \[
    \text{SSignal \ val} \cong (\text{val}, \text{DSignal \ val})
    \]
Generators, not signals

- registration actions executed once per consumer
- when using `scan`, every consumer
  - creates a mutable variable holding the accumulated value
  - registers a handler that updates this variable
- two problems:
  1. duplication of computations
  2. signal values depending on consumption time
- explanation:
  - signal generators instead of signals
Using native memoization

- discrete signal contains list of value occurrences:

  \( DSignal\ val \cong [(Time, val)], \ldots \)

- signal union involves union of occurrence lists:

  \[
  \text{occsUnion}\ ((time_1, val_1) : occs_1))
  \quad ((time_2, val_2) : occs_2)) = occs' \text{ where}
  \quad occs' = \text{case compare time}_1 \ time_2 \text{ of } \ldots
  \]

- problem:
  comparison happens at the earlier one of both times

- our solution:
  postpone the decision
always gives several possible “futures” depending on what event source fires next

**vista for union inPackets outPackets:**

```
start
  in inPacket_1
  out outPacket_1
  in inPacket_1
  out outPacket_1
  in inPacket_2
  out outPacket_1
  . . .
  . . .
  . . .
```
Consuming vistas

- consumer evaluates only the relevant path:
Fixing start times (1)

- signal types get an extra (phantom) type parameter (similar to \textit{STRef}) that represents their lifetime (era):

  \[DSignal \text{ era } \text{ val}\]
  \[SSignal \text{ era } \text{ val}\]

- signal combinators enforce era equality:

  \[
  \text{union} :: DSignal \text{ era } \text{ val} \\
  DSignal \text{ era } \text{ val} \rightarrow \\
  DSignal \text{ era } \text{ val} \rightarrow \\
  \text{scan} :: \text{accu} \\
  (\text{accu} \rightarrow \text{val} \rightarrow \text{accu}) \rightarrow \\
  (DSignal \text{ era } \text{ val} \rightarrow SSignal \text{ era } \text{ accu})
  \]
reactive actions with era parameters (similar to $ST$):

```
newtype Reactive era val = Reactive (IO val)
```

guaranteed production and consumption enforce era equality
(similar to $newSTRef$, $readSTRef$ and $writeSTRef$)

running reactive actions uses universal quantification (similar

``` to $runST$):

toIO :: (\era. Reactive era val) → IO val
Safe switching

- safe switching combinator:

  \[\text{switch} :: (\text{Signal} \ \text{signal}) \Rightarrow \text{SSignal} \ \text{era} (\forall \text{era}' . \text{signal} \ \text{era}' \ \text{val}) \Rightarrow \text{signal} \ \text{era} \ \text{val}\]

- switches only to signals that don’t depend on external events:
  - empty discrete signal
  - constant segmented signals

- useless

- idea:

  switching between signal functions instead of signals
Signal functions (1)

- functions over signals with identical era:

\[
\text{SignalFun era} \ (\text{signal}_1 \ ‘\text{Of}’ \ \text{val}_1 \mapsto \\
\ldots \mapsto \\
\text{signal}_n \ ‘\text{Of}’ \ \text{val}_n \mapsto \\
\text{signal’} \ ‘\text{Of}’ \ \text{val’})
\]

- empty data types for type indices:

\[
\textbf{data} \ \text{shape} \mapsto \ \text{shape’} \\
\textbf{data} \ \text{signal} \ ‘\text{Of}’ \ \text{val}
\]
Signal functions (2)

- SignalFun defined as a GADT:

```haskell
data SignalFun era shape where

  OSF :: (Signal signal) ⇒
  signal era val

  SSF :: (Signal signal) ⇒
  (signal era val → SignalFun era shape') →
  SignalFun era (signal 'Of' val ↦ shape')
```
Switching between signal functions

- type of the switching combinator:

\[
\text{switch} :: \text{SSignal } \text{era} (\forall \text{era}' . \text{SignalFun } \text{era}' \text{ shape}) \rightarrow \text{SignalFun } \text{era shape}
\]

- how the combinator works (conceptionally):
  - arguments of the result function are cut up to fit the segments of the argument signal (ageing)
  - each function from the argument signal is applied to its corresponding slice
  - result function combines the results of these applications

- important:

  ageing is enforced
Signals, Not Generators!

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

Tenth Symposium on Trends in Functional Programming
June 2–4, 2009
Traditional implementation of \textit{scan}

\[
\begin{align*}
\text{scan} :: accu & \rightarrow \\
& (accu \rightarrow val \rightarrow accu) \rightarrow \\
& (DSignal\ val \rightarrow SSignal\ accu) \\
\text{scan init next (DSignal\ reg)} & = SSignal\ init\ (DSignal\ reg') \text{ where} \\
\text{reg'}\ hdlr & = \text{do} \\
accuRef & \leftarrow \text{newIORef init} \\
reg & (\lambda val \rightarrow \text{do} \\
accu & \leftarrow \text{readIORef accuRef} \\
\text{let} \\
accu' & = \text{next accu val} \\
\text{writeIORef accuRef accu'} \\
hdlr & \text{ accu'})
\end{align*}
\]