Signale statt Generatoren!

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Lehrstuhl Programmiersprachen und Compilerbau

27. Workshop der GI-Fachgruppe
Programmiersprachen und Rechenkonzepte
3. bis 5. Mai 2010

Funktionale Reaktive Programmierung

- funktionale Programmierung:

  Problembeschreibung statt Ablaufplan
- Funktionale Reaktive Programmierung (FRP):

  Anwendung dieses Prinzips auf reaktive Systeme
- in diesem Vortrag: Haskell-Bibliothek Grapefruit

  `http://haskell.org/haskellwiki/Grapefruit`

Signale

- das Herzstück der FRP
- beschreiben zeitliches Verhalten
- verschiedene Arten:

DSignal diskret

Inhalt diskrete Zeitpunkte mit assoziierten Werten

Beispiel *packets* :: *DSignal Packet*

SSignal segmentiert

Inhalt Größe, die sich an diskreten Zeitpunkten ändert

Beispiel *volumes* :: *SSignal Int*

Einfache Signalkombinatoren (1)

- Mischen diskreter Signale:

  $union :: DSignal\ val \rightarrow DSignal\ val \rightarrow DSignal\ val$

  Beispiel:

  $packets :: DSignal\ Packet$
  $packets = union\ inPackets\ outPackets$

Einfache Signalkombinatoren (2)

- Akkumulieren von Signalwerten:

$$
\begin{aligned}
&\mathit{scan} :: \mathit{accu} \rightarrow \\
&\quad (\mathit{accu} \rightarrow \mathit{val} \rightarrow \mathit{accu}) \rightarrow \\
&\quad \mathit{DSignal}\ \mathit{val} \rightarrow \\
&\quad \mathit{SSignal}\ \ \mathit{accu}
\end{aligned}
$$

Beispiel:

$$
\begin{aligned}
&\mathit{volumes} :: \mathit{DSignal}\ \mathit{Packet} \rightarrow \mathit{SSignal}\ \mathit{Int} \\
&\mathit{volumes}\ \mathit{packets} = \mathit{scan}\ 0\ \mathit{next}\ \mathit{packets}\ \textbf{where} \\
&\quad \mathit{next}\ \mathit{volume}\ \mathit{packet} = \mathit{volume} + \mathit{size}\ \mathit{packet}
\end{aligned}
$$

Umschalten zwischen Signalen

- Kombinator zum Umschalten zwischen Signalen:

  $switch :: (Signal\ sig) \Rightarrow SSignal\ (sig\ val) \rightarrow sig\ val$

  instance *Signal DSignal* **where** ...

  instance *Signal SSignal* **where** ...

- Anwendung:

  Anzeige des Datenvolumens eingegangener oder ausgegangener Pakete in Abhängigkeit von Benutzerauswahl

Ereignisgetriebene Ausführung

- ereignisgetriebene Zustandsänderungen
- Signalkonsumenten registrieren Event-Handler
- typische Implementierung von Signalen:

  DSignal v Signal ist Registrierungsaktion:

  $$(v \to \mathit{IO}\ ()) \to \mathit{IO}\ ()$$

  SSignal v Signal ist Initialwert plus Aktualisierungen:

  $$(v, \mathit{DSignal}\ v)$$

Keine Signale, sondern Generatoren

- Registrierungsaktionen werden einmal pro Konsument ausgeführt
- Folge bei Benutzung von *scan*:
  - jeder Konsument erzeugt veränderliche Variable, die akkumulierten Wert speichert
  - jeder Konsument registriert Event-Handler, der diese Variable aktualisiert
- zwei Probleme:
  1. Mehrfachberechnungen
  2. Signalwerte können vom Zeitpunkt des Konsumierens abhängen
- Anschauung:

  keine Signale, sondern Signalgeneratoren

Nutzung nativer Memoization

- berechnete Werte werden gespeichert,
  wenn Datenstruktur an Variable gebunden ist
- Problem:

  Datenstruktur diskreter Signale enthält Signalwerte nicht
- Änderung der Datenstruktur:

$$([(\mathit{Time}, v)], \ldots)$$

Problem mit Mischen von Signalen

- Vereiningung von Listen von Zeit-Wert-Paaren:

$$\begin{aligned} &\mathit{occsUnion}\ ((\mathit{time}_1, \mathit{val}_1) : \mathit{occs}_1) \\ &\qquad ((\mathit{time}_2, \mathit{val}_2) : \mathit{occs}_2) = \mathit{occs}' \ \textbf{where} \\ &\mathit{occs}' = \textbf{case}\ \mathit{compare}\ \mathit{time}_1\ \mathit{time}_2\ \textbf{of}\ \ldots \end{aligned}$$

- Problem:

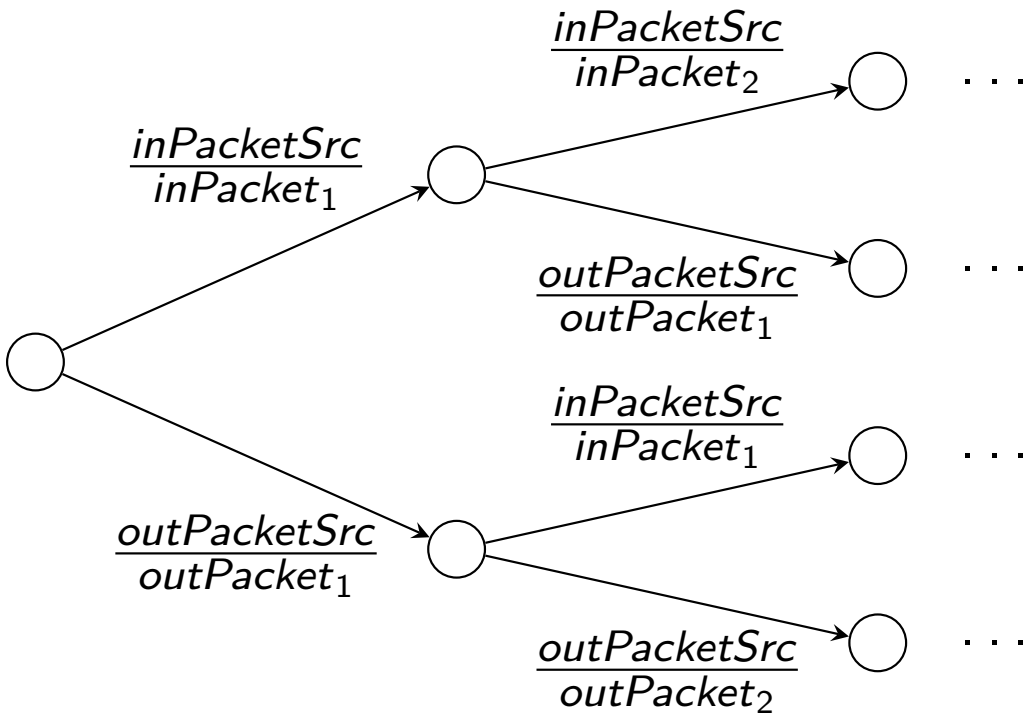
  Vergleich von Zeiten geschieht zu zeitig

- Lösung:

  Entscheidung vertagen

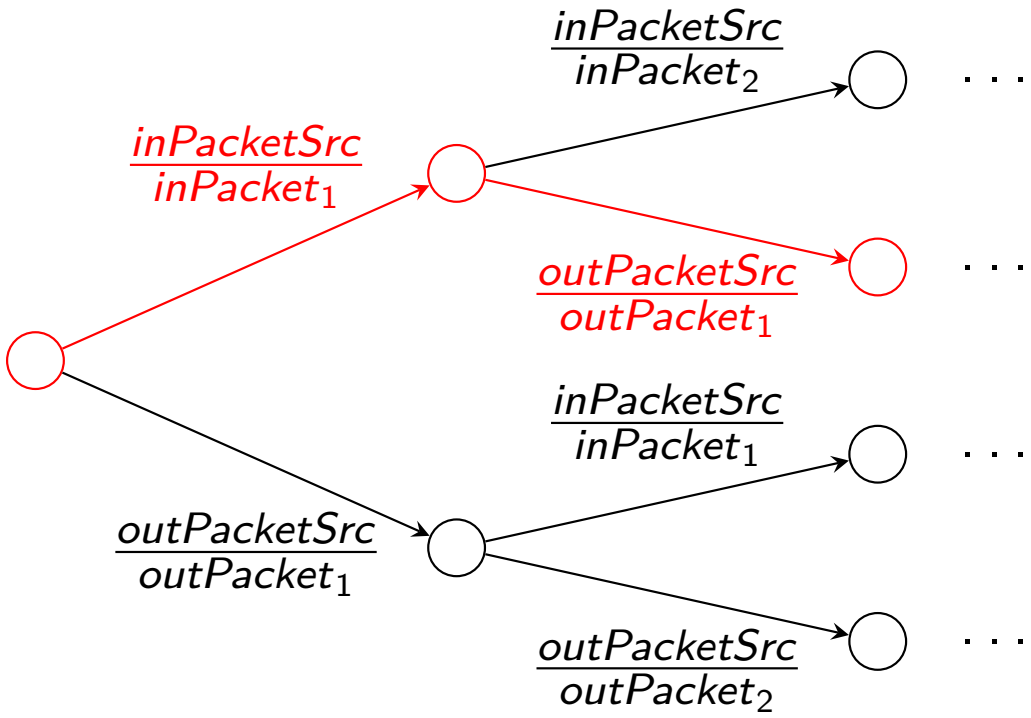
Vistas

- Vista enthält für jeden Zeitpunkt mehrere mögliche „Zukünfte“ in Abhängigkeit davon, welche Ereignisquelle als nächstes feuert
- Vista für *union inPackets outPackets*:



Konsumieren von Vistas

- Konsument wertet nur relevanten Pfad aus:



Fixieren von Startzeiten (1)

- Signaltyp erhält einen zusätzlichen (Phantom-)Typparameter, der die Lebenszeit („Ära“) des Signals repräsentiert
- einfache Signalkombinatoren erzwingen Gleichheit von Ären:

$$
\begin{aligned}
\mathit{union} :: & \; \mathit{DSignal\ era\ val} \rightarrow \\
& \; \mathit{DSignal\ era\ val} \rightarrow \\
& \; \mathit{DSignal\ era\ val}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{scan} :: & \; \mathit{accu} \rightarrow \\
& \; (\mathit{accu} \rightarrow \mathit{val} \rightarrow \mathit{accu}) \rightarrow \\
& \; \mathit{DSignal\ era\ val} \rightarrow \\
& \; \mathit{SSignal\ era\ accu}
\end{aligned}
$$

Fixieren von Startzeiten (2)

- reaktive Aktionen mit Ära-Parametern:

$$\textbf{newtype}\ \mathit{Reactive\ era\ val} = \mathit{Reactive}\ (\mathit{IO\ val})$$

- Signalproduktion und -konsumierung erzwingt Gleichheit von Signal-Ära mit der Ära der reaktiven Aktion
- Ausführung von reaktiven Aktionen nur bei Ära-Unabhängigkeit möglich:

$$\mathit{toIO} :: (\forall \mathit{era}.\mathit{Reactive\ era\ val}) \rightarrow \mathit{IO\ val}$$

Umschalten zwischen Signalfunktionen

- Funktionen über Signalen mit identischer Ära:

$$\begin{aligned} &\mathit{SignalFun\ era\ (sig_1\ `Of`\ val_1} \mapsto \\ &\quad \ldots \qquad\qquad\quad \mapsto \\ &\mathit{sig_n\ `Of`\ val_n} \mapsto \\ &\mathit{sig'\ `Of`\ val')} \end{aligned}$$

- Typ von *switch*:

$$\mathit{SSignal\ era\ (\forall era'.SignalFun\ era'\ shape)} \rightarrow \mathit{SignalFun\ era\ shape}$$

- Arbeitsweise des Kombinators:
  - Argumente der Resultatfunktion werden auf die Segmentintervalle „zurechtgestutzt“
  - jede Funktion des Argumentsignals wird auf entsprechend gestutzte Argumente angewendet
  - Resultate werden aneinander gehängt

Signale statt Generatoren!

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Lehrstuhl Programmiersprachen und Compilerbau

27. Workshop der GI-Fachgruppe
Programmiersprachen und Rechenkonzepte
3. bis 5. Mai 2010

Implementierung von *scan*

```
scan :: accu                          →
        (accu → val → accu)           →
        DSignal val                   →
        SSignal  accu
scan init next (DSignal reg) = SSignal init (DSignal reg') where
   reg' hdlr' = do
                  accuRef ← newIORef init
                  reg (λval → do
                                 accu ← readIORef accuRef
                                 let
                                    accu' = next accu val
                                 writeIORef accuRef accu'
                                 hdlr' accu')
```