# Funktionale GUI-Programmierung in Haskell mit Grapefruit

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Lehrstuhl Programmiersprachen und Compilerbau

Haskell in Leipzig 2, Juli 2007

# GUI-Programmierung in Haskell heute

- in der Praxis imperative Bibliotheken mit etwas funktionalem Flair:
  - Gtk2Hs
  - wxHaskell

  und andere
- daher nur unzureichende Nutzung der Vorteile von Haskell

# Rein funktionale GUI-Programmierung

- Forschung zu rein funktionaler GUI-Programmierung seit über 15 Jahren
- viele Bibliotheken:
  - fortschrittliche Konzepte umgesetzt
  - i.A. nicht für den produktiven Einsatz geeignet
  - viele Projekte auch wieder eingeschlafen

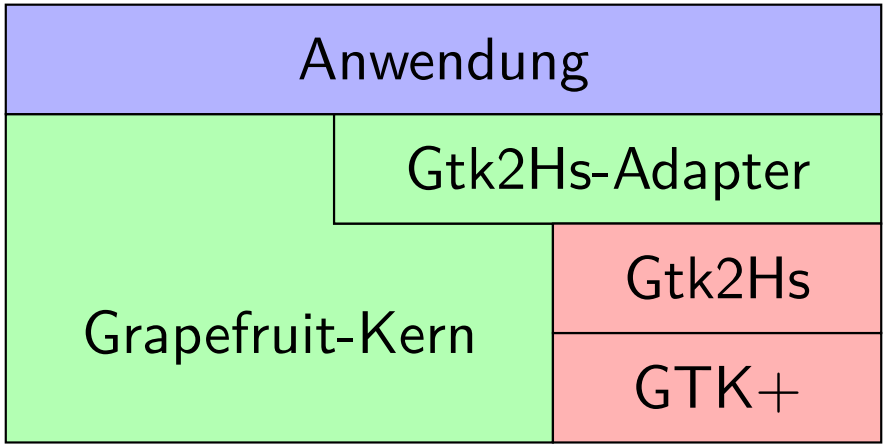
# Grapefruit-Entwicklungsziele

- Aufgreifen bewährter Ideen und Entwicklung neuer Techniken
- Praxistauglichkeit
- natives Look-and-Feel
- Plattform für Forschung zu Spezifikation und Verifikation grafischer Oberflächen

# Eckdaten zur Entwicklung

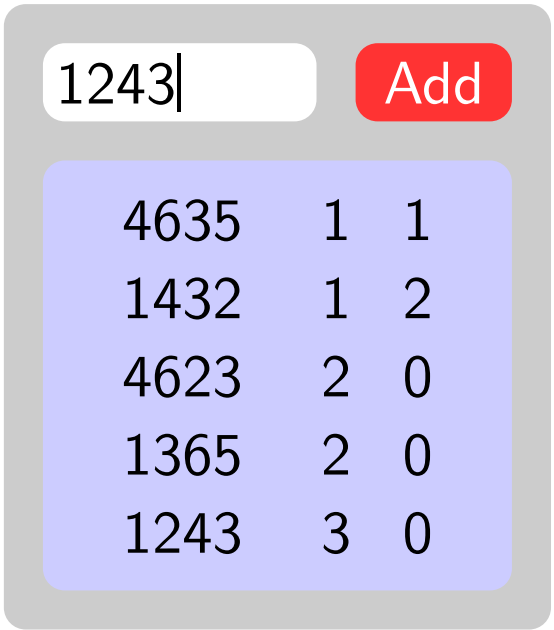
- Entwicklung seit Ende Februar 2007 an der BTU Cottbus
- derzeitige Entwickler:
  - Wolfgang Jeltsch (wissenschaftlicher Mitarbeiter)
  - Matthias Reisner (Student)
- momentaner Stand:
  - zentrale Konzepte umgesetzt
  - noch sehr kleine Auswahl an GUI-Komponenten

# Architektur




- Nutzung des Gtk2Hs-Toolkits als Basis
- Trennung von allgemeinem und toolkitspezifischem Code (Portierung auf andere Toolkits geplant)

# Beispiel: Mastermind™-ähnliches Spiel Codebreaker




- Hinzufügeknopf aktiv, wenn gültige Kombination im Eingabefeld
- beim Drücken dieses Knopfes Einfügen der Kombination in die Versuchsliste
- Anzeige stellt immer Versuche zusammen mit Feedbacks dar
- Spielfeld ist aktiv, wenn die Versuchsliste nicht den Code enthält

# GUI-Programmierung in Grapefruit

- zeitliche Abläufe als First-Class-Objekte:
  - Ereignisströme
  - Signale
- Netzwerke miteinander kommunizierender GUI-Komponenten:
  - Arrows
- Effizienz durch datengetriebene Implementierung

# Ereignisströme

## Definition

Ein Ereignisstrom ist eine Folge von Ereignissen, wobei folgendes gilt:

- Jedes Ereignis tritt zu einer bestimmten Zeit auf.
- Jedes Ereignis transportiert einen Wert.

## Intuition

**data** EventStream *value* = EventStream [(Time, *value*)]

# Funktionen zur Arbeit mit Ereignisströmen (Auswahl)

$$
\begin{aligned}
\mathit{mempty} &:: \text{EventStream } \mathit{value} \\
\mathit{mappend} &:: \text{EventStream } \mathit{value} \\
&\rightarrow \text{EventStream } \mathit{value} \\
&\rightarrow \text{EventStream } \mathit{value} \\
\mathit{fmap} &:: (\mathit{value1} \rightarrow \mathit{value2}) \\
&\rightarrow (\text{EventStream } \mathit{value1} \rightarrow \text{EventStream } \mathit{value2}) \\
\mathit{filter} &:: (\mathit{value} \rightarrow \text{Bool}) \\
&\rightarrow (\text{EventStream } \mathit{value} \rightarrow \text{EventStream } \mathit{value}) \\
\mathit{scan} &:: \mathit{accu} \\
&\rightarrow (\mathit{accu} \rightarrow \mathit{eventValue} \rightarrow \mathit{accu}) \\
&\rightarrow (\text{EventStream } \mathit{eventValue} \rightarrow \text{EventStream } \mathit{accu})
\end{aligned}
$$

# Signale

## Definition

Ein Signal ist eine Zuordnung von Werten zu Zeitpunkten, beschreibt also zeitabhängige Werte.

## Intuition

**data** Signal *value* = Signal (Time → *value*)

# Funktionen zur Arbeit mit Signalen (Auswahl)

$$
\begin{aligned}
\mathit{signal} &:: \mathit{value} \rightarrow \text{EventStream } \mathit{value} \rightarrow \text{Signal } \mathit{value} \\
(\langle\$\rangle) &:: (\mathit{value1} \rightarrow \mathit{value2}) \\
&\rightarrow (\text{Signal } \mathit{value1} \rightarrow \text{Signal } \mathit{value2}) \\
\mathit{liftA2} &:: (\mathit{value1} \rightarrow \mathit{value2} \rightarrow \mathit{value3}) \\
&\rightarrow (\text{Signal } \mathit{value1} \rightarrow \text{Signal } \mathit{value2} \rightarrow \text{Signal } \mathit{value3}) \\
\mathit{sample} &:: \text{EventStream } () \\
&\rightarrow \text{Signal } \mathit{value} \\
&\rightarrow \text{EventStream } \mathit{value}
\end{aligned}
$$

# Arrows

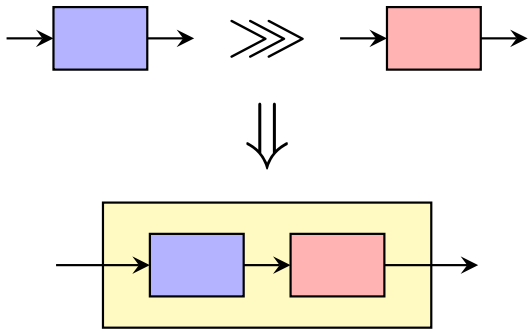
## Intuition

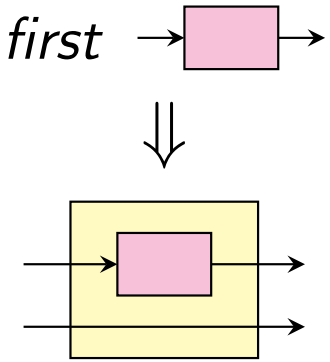
Arrows sind Komponenten mit je einer Eingabe und einer Ausgabe, welche miteinander vernetzt werden können, um neue Arrows zu formen.

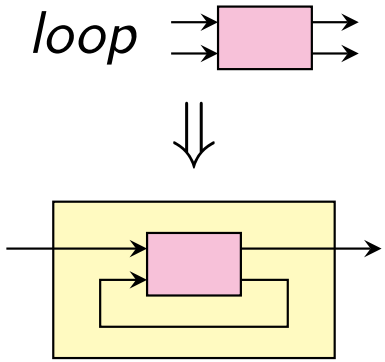
# Arrowklassen (Auswahl)

- für uns wichtige Arrowklassen:
  - Basisklasse Arrow mit Methoden *pure*, $(\ggg)$ und *first*
  - Subklasse ArrowLoop mit Methode *loop*
- Zweck der Methoden:
  - *pure* für Konstruktion effektloser Arrows
  - $(\ggg)$, *first* und *loop* für Konstruktion zusammengesetzter Arrows

# Konstruktion zusammengesetzter Arrows





# Arrow-Syntax (vereinfacht)

- **proc**-Ausdrücke für komfortable Konstruktion von Arrows:

$$
\begin{aligned}
&\textbf{proc}\ \langle input \rangle \rightarrow \textbf{do} \\
&\quad \langle output_1 \rangle \leftarrow \langle arrow_1 \rangle \rightarrowtail \langle input_1 \rangle \\
&\qquad\qquad \vdots \qquad\qquad \vdots \\
&\quad \langle output_n \rangle \leftarrow \langle arrow_n \rangle \rightarrowtail \langle input_n \rangle \\
&\quad returnA \rightarrowtail \langle output \rangle
\end{aligned}
$$

- **rec**-Blöcke zur Darstellung von Arrows mit Zyklen

# Beispielcode: Eingabekomponente in Codebreaker (vereinfacht)

$$
\begin{aligned}
&\textbf{proc}\ () \rightarrow \textbf{do} \\
&\quad \mathit{guess} \leftarrow \mathit{guessInputField} \dashv () \\
&\quad \mathit{sendReq} \leftarrow \mathit{addButton} \dashv \mathit{isOk} \langle\$\rangle \mathit{guess} \\
&\quad \mathit{returnA} \dashv \mathit{sample}\ \mathit{sendReq}\ \mathit{guess}
\end{aligned}
$$

# Umsetzung dynamischer Oberflächen

- Dynamikkomponente:
  - erhält zeitabhängige Collection von Komponenten als Eingabe
  - enthält immer genau die Elemente der aktuellen Collection
- Probleme bei der Verwendung von Signalen:
  - linearer Zeitaufwand für Aktualisierungen
  - keine Elementidentitäten
- Lösung durch spezielle Containersignal-Typen:
  - gegenwärtig für Listen implementiert:

**data** ListSignal *element*

# Erzeugung elementarer Listensignale

- Konstruktion durch Initalwert und Ereignisstrom mit Änderungen:

$$
\begin{aligned}
\mathit{listSignal} \;::\; & [\mathit{element}] \\
& \rightarrow \text{EventStream (Change } \mathit{element}\text{)} \\
& \rightarrow \text{ListSignal } \mathit{element}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{data}\ \text{Change } \mathit{element} = \; & \text{Addition Int } \mathit{element} \\
\mid\; & \text{Removal Int} \\
\mid\; & \text{Move Int Int}
\end{aligned}
$$

# Weitere Funktionen zur Arbeit mit Listensignalen (Auswahl)

$$
\begin{aligned}
\mathit{elem} &:: \mathit{element} \to \text{ListSignal } \mathit{element} \to \text{Signal Bool} \\
\mathit{length} &:: \text{ListSignal } \mathit{element} \to \text{Signal Int} \\
\mathit{reverse} &:: \text{ListSignal } \mathit{element} \to \text{ListSignal } \mathit{element} \\
\mathit{filter} &:: (\mathit{element} \to \text{Bool}) \\
&\quad \to (\text{ListSignal } \mathit{element} \to \text{ListSignal } \mathit{element}) \\
\mathit{sort} &:: \text{Ord } \mathit{element} \Rightarrow \\
&\quad \text{ListSignal } \mathit{element} \to \text{ListSignal } \mathit{element}
\end{aligned}
$$

# Effizienz von Listensignalen

- Platzaufwand: $O(1)$ bzw. $O(n \log n)$
- Zeitaufwand:
  - für Initialisierung: $O(1)$ bzw. $O(n \log n)$
  - für Aktualisierung: $O(1)$, $O(\log n)$ bzw. $O\left(\log^2 n\right)$

# Zusammenfassung

- problemnahe Beschreibung von Oberflächen:
  - Beschreibung zeitlicher Abläufe als zentrales Konzept
  - Arrows und Arrow-Syntax zur Beschreibung der Interaktion von Komponenten
- trotzdem effiziente Umsetzung:
  - datengetriebene Implementierung

# Ausblick

- größere Auswahl an Komponenten
- weitere Containersignaltypen
- weitere Adapter
- Model-View-Controller-Prinzip
- sich kontinuierlich verändernde Signale
- Grafik (animiert, ggf. interaktiv)
- Einbindung externer Objekte (z.B. Dateisystem)
- Klangsynthese

# Schluss

- weiterführende Ressourcen:
  - Grapefruit-Homepage unter `http://haskell.org/haskellwiki/Grapefruit`
  - Kontaktierung des Autors unter `jeltsch@informatik.tu-cottbus.de`
- Danke für die Aufmerksamkeit!
- Fragen?