Funktionale reaktive Programmierung

Wolfgang Jeltsch

Kybernetisches Institut
Technische Universität Tallinn
Tallinn, Estland

Vortrag am IHP, Leibniz-Institut für innovative Mikroelektronik

15. Oktober 2015

Motivation

- reaktive Systeme sind überall:
  - eingebettete Systeme
  - Kommunikationssysteme
  - Benutzerschnittstellen

    usw.
- deklarativer Ansatz zur Programmierung reaktiver Systeme wünschenswert:
  - verständlicherer Code
  - weniger Programmierfehler
  - höhere Wiederverwendbarkeit von Code
  - höhere Entwicklereffizienz
  - bessere Wartbarkeit
  - bessere Möglichkeiten zur Verifikation

Funktionale Programmierung

- deklaratives Programmierparadigma
- Funktionen sind normale Werte:
  - können als Argumente und Resultate von Funktionen auftreten
  - können Teil von Datenstrukturen sein
- keine Seiteneffekte während der Auswertung von Ausdrücken (referenzielle Transparenz):
  - wiederholte Auswertung des gleichen Ausdrucks führt zu gleichem Ergebnis
  - Ersetzen einer Variable durch den sie definierenden Ausdruck ändert Programmverhalten nicht
  - Werte spezieller Datentypen zur Beschreibung seiteneffekthaltiger Aktionen
- oft leistungsfähiges statisches Typsystem

Funktionale Programmiersprachen

- rein funktionale Sprachen:
  - Haskell
  - Clean
  - Agda

  usw.
- Sprachen ohne referenzielle Transparenz:
  - Standard ML
  - OCaml
  - F#
  - Scala
  - Lisp
  - Scheme

  usw.

Funktionale reaktive Programmierung (FRP)

- Erweiterung der funktionalen Programmierung um Unterstützung für reaktive Systeme
- Werte spezieller Datentypen zur Beschreibung des zeitlichen Systemverhaltens
- Konstruktion komplexer Beschreibungen aus einfacheren mittels spezieller Operationen
- Initialzündung 1997 mit Conal Elliotts Arbeit zu „funktionaler reaktiver Animation“
- seitdem aktive Forschung zu Theorie und Praxis der FRP

Funktionale reaktive Programmiersprachen

- Implementierung üblicherweise als eingebettete domänenspezifische Sprache (Bibliothek)
- oft auf Basis von Haskell:
  - FRP-Zoo: `https://github.com/gelisam/frp-zoo`
  - eigene Bibliothek Grapefruit, Version 0
- aber auch andere Programmiersprachen als Basis:
  - Scheme
  - Java
  - JavaScript

  usw.

Dieser Vortrag

- FRP-Konzepte gemäß eigener Arbeiten
- Stand dieser Arbeiten:
  - Theorie im Wesentlichen fertig
  - Implementierung im Anfangsstadium (Grapefruit, Version 1)

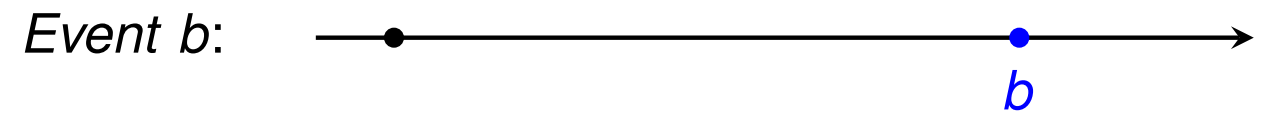
Grundlagen

- Zeit:
  - linear
  - nicht notwendigerweise diskret
- Grundbausteine zum Beschreiben zeitlichen Verhaltens:
  - Signale
  - Ereignisse

Signale und Ereignisse

- Signal ist zeitveränderlicher Wert:

- Ereignis ist Zeitpunkt mit zugehörigem Wert:



- Beispiele:

  Signal Double Audiokanal in einer Multimedia-Anwendung

  Event KeyCode Tastendruck

Zeitabhängige Typzugehörigkeit

- gewisse temporale Bedingungen sollen zur Compile-Zeit geprüft werden
- Einführung von zeitabhängiger Typzugehörigkeit
- Werte zu einem Zeitpunkt t:

  Signal a Signale, welche unmittelbar nach t starten

  Event b Ereignisse, welche nach t auftreten
- Auswirkung auf Inhalte von Signalen und Ereignissen:

  Signal a Wert des Signals zu einem Zeitpunkt t' ist ein Wert vom Typ a zum Zeitpunkt t'

  Event b Wert des Ereignisses ist ein Wert vom Typ b zum Zeitpunkt t

Anwendung zeitabhängiger Typzugehörigkeit

Signal (*Signal a*)	Wert des Signals zu einem Zeitpunkt *t* ist ein Signal, dass unmittelbar nach *t* startet
Event (*Event b*)	inneres Ereignis tritt nach dem äußeren Ereignis auf
Event (*Signal a*)	Signal startet unmittelbar nach dem Auftreten des Ereignisses

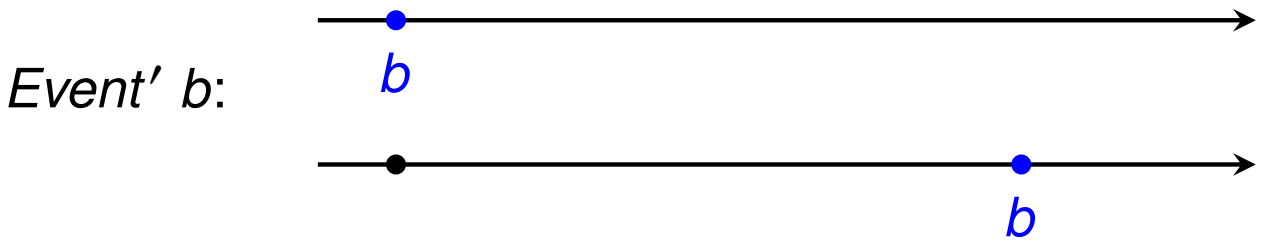
Einbeziehen der Gegenwart

- Werte der Typen *Signal* und *Event* beziehen sich nur auf die Zukunft
- Ableiten von Varianten, die die Gegenwart einbeziehen
- Signale, welche in der Gegenwart starten:

$$\textbf{data}\ \mathit{Signal'}\ a = \mathit{Signal'}\ a\ (\mathit{Signal}\ a)$$

- Ereignisse, welche in der Gegenwart auftreten können:

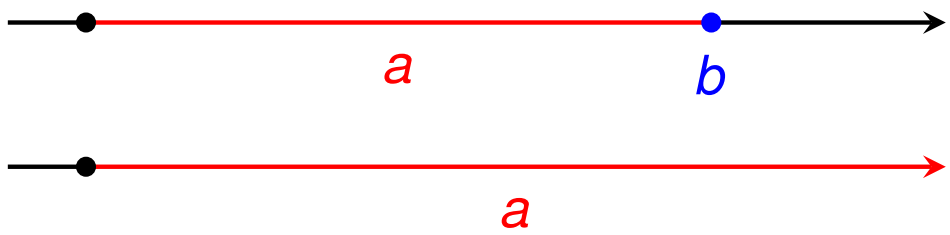
$$\textbf{data}\ \mathit{Event'}\ b = \mathit{Now}\ b \mid \mathit{Later}\ (\mathit{Event}\ b)$$



Prozesse

- Prozess besteht aus einem kontinuierlichem Teil und optional einem Abschlussereignis:

Process a b:



- unterschiedliche Prozesstypen mit verschiedenen Terminationsgarantien möglich:

$\mathit{Process}_{\leqslant\infty}$	Nicht-Termination möglich
$\mathit{Process}_{<\infty}$	Termination garantiert
$\mathit{Process}_{\leqslant t_b}$	inklusive Obergrenze für Terminationszeit
$\mathit{Process}_{<t_b}$	exklusive Obergrenze für Terminationszeit

Ableiten von Typen für Signale und Ereignisse

- Signale entsprechen nicht-terminierenden Prozessen:

  type *Signal a* = $Process_{\leqslant\infty}$ *a Void*

  data *Void* -- empty type

- Ereignisse entsprechen terminierenden Prozessen, bei denen die Werte im kontinuierlichen Teil keine Information beinhalten:

  type *Event b* = $Process_{<\infty}$ () *b*

Einbeziehen der Gegenwart

- Prozesse, deren kontinuierlicher Teil in der Gegenwart startet:

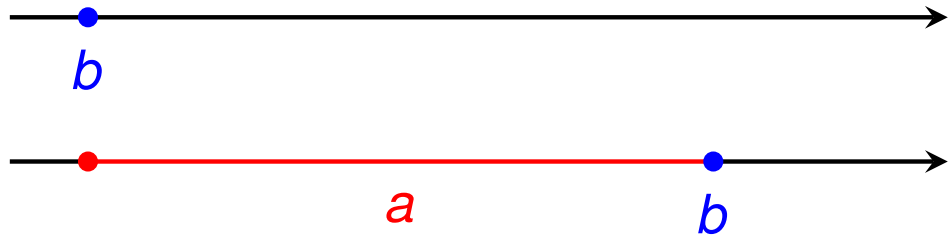
$$\textbf{data}\ \mathit{Process'}\ a\ b = \mathit{Process'}\ a\ (\mathit{Process}\ a\ b)$$

Process′ a b:

- Prozesse, die in der Gegenwart terminieren können:

$$\begin{aligned}\textbf{data}\ \mathit{Process''}\ a\ b = {} & \mathit{StoppingNow}\ b \\ \mid {} & \mathit{StoppingLater}\ (\mathit{Process'}\ a\ b)\end{aligned}$$

Process″ a b:



Anwendungsbeispiele für Prozesse

- möglicherweise terminierende Signale:

$$\textbf{type}\ \mathit{AudioChannel} = \mathit{Process'}\ \mathit{Double}\ ()$$

- Information über Termination:

$$\textbf{type}\ \mathit{AudioChannel} = \mathit{Process'}\ \mathit{Double}\ \mathit{DeletionReason}$$

$$\textbf{data}\ \mathit{DeletionReason} = \mathit{DeviceRemoval} \mid \mathit{DeletionByUser}$$

- zeitabhängige Typen:

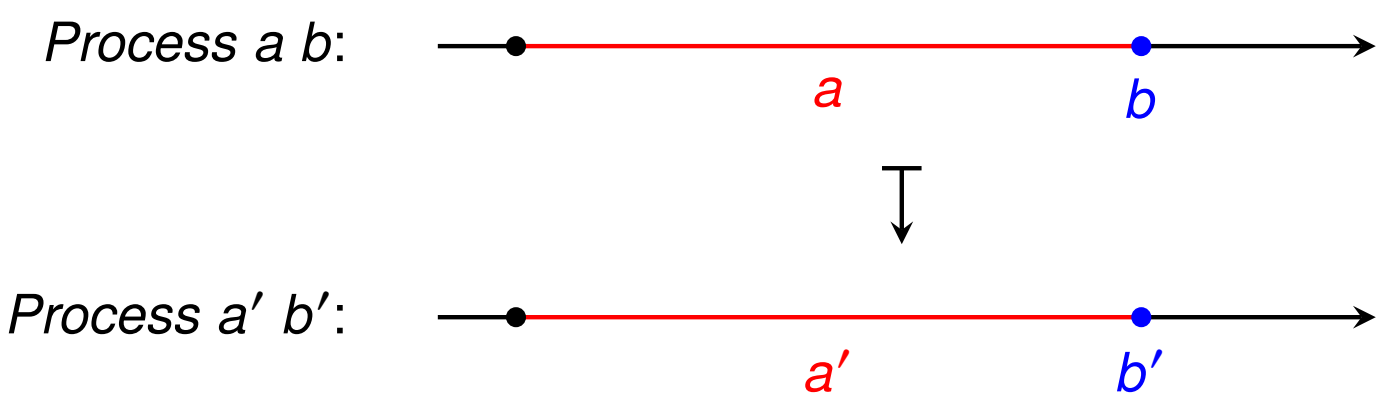
$$\textbf{type}\ \mathit{StereoMono} = \mathit{Process'}\ (\mathit{Double}, \mathit{Double})\ \mathit{MonoStereo}$$

$$\textbf{type}\ \mathit{MonoStereo} = \mathit{Process'}\ \mathit{Double}\ \mathit{StereoMono}$$

Werteumwandlung

- Konvertieren der Werte des kontinuierlichen Teils und des Wertes des Abschlussereignisses eines Prozesses:

$$\begin{aligned} \mathit{map} \;::\; & (a \longrightarrow a') \\ & \to (b \longrightarrow b') \\ & \to (\mathit{Process}\ a\ b \longrightarrow \mathit{Process}\ a'\ b') \end{aligned}$$

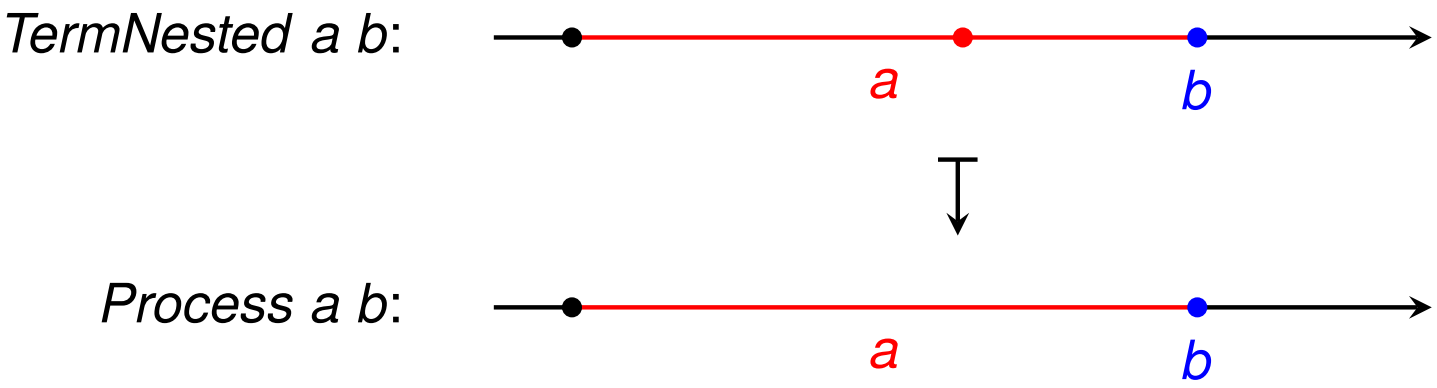


Verbinden

- Verketten eines kontinuierlichen Teils mit einem Folgeprozess:

$$join :: TermNested\ a\ b \longrightarrow Process\ a\ b$$

$$\textbf{type}\ TermNested\ a\ b = Process\ a\ (Process''\ a\ b)$$

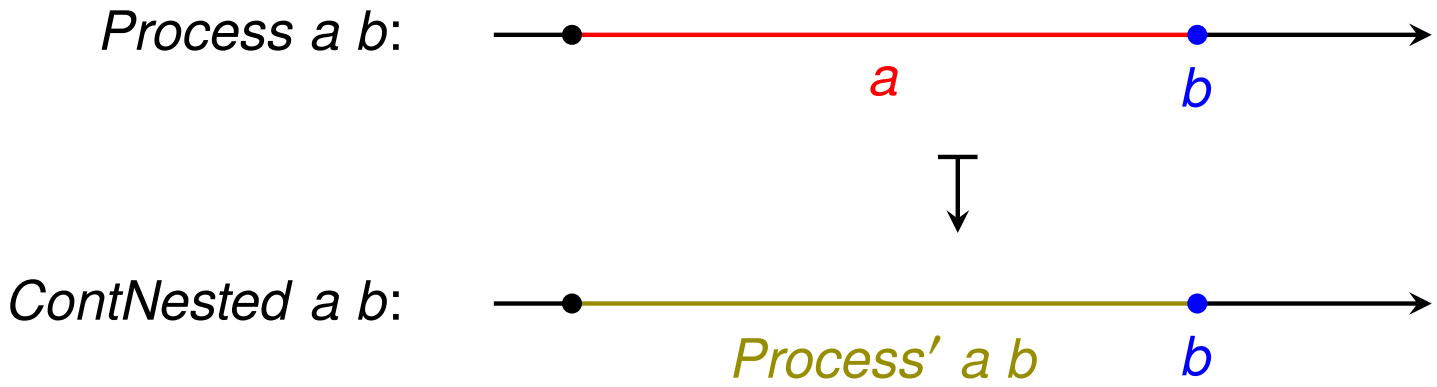


Expandieren

- Erzeugen eines kontinuierlichen Teils von immer kürzer werdenden Suffixen:

$$expand :: Process\ a\ b \longrightarrow ContNested\ a\ b$$

$$\textbf{type}\ ContNested = Process\ (Process'\ a\ b)\ b$$

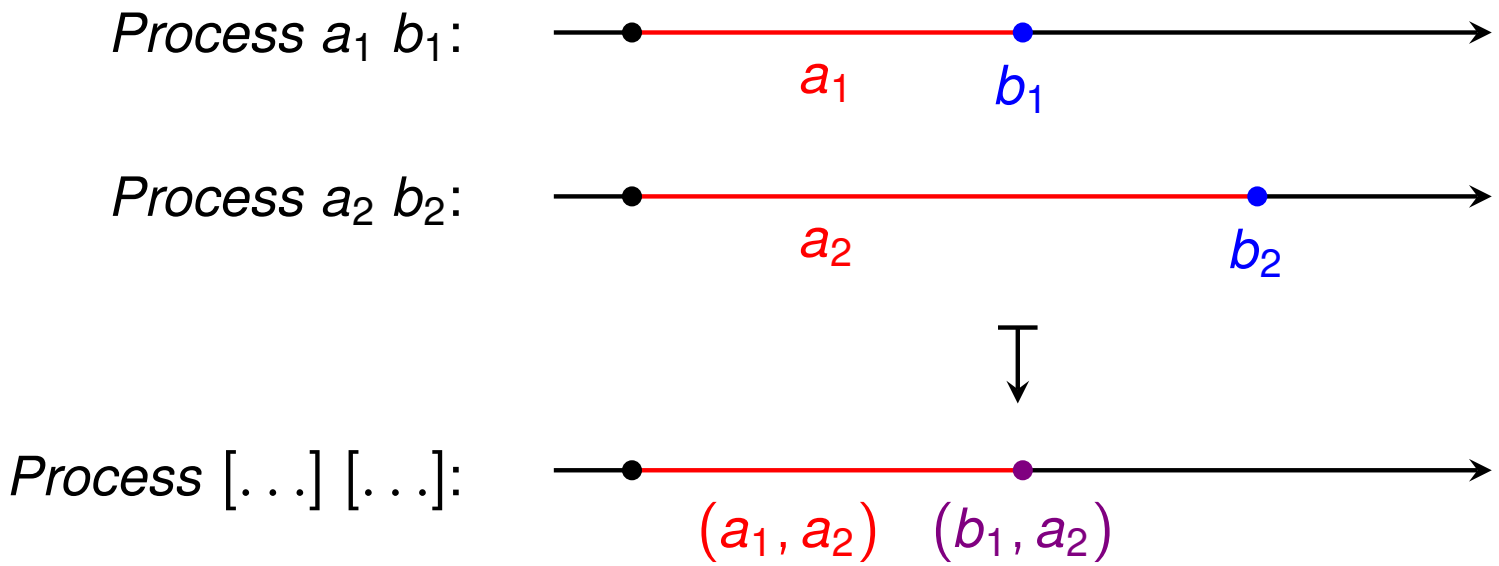


Vereinigen

- Vereinigen zweier Prozesse:

$$\begin{aligned} \mathit{merge} \;::\; & (\mathit{Process}\ a_1\ b_1, \mathit{Process}\ a_2\ b_2) \\ & \longrightarrow \mathit{Process}\ (a_1, a_2)\ (\mathit{RaceResult}\ a_1\ b_1\ a_2\ b_2) \end{aligned}$$

$$\begin{aligned} \textbf{data}\ \mathit{RaceResult}\ a_1\ b_1\ a_2\ b_2 = {} & \mathit{FstWins}\ \ b_1\ a_2 \\ \mid {} & \mathit{SndWins}\ a_1\ b_2 \\ \mid {} & \mathit{Tie}\ \ \ \ \ \ b_1\ b_2 \end{aligned}$$

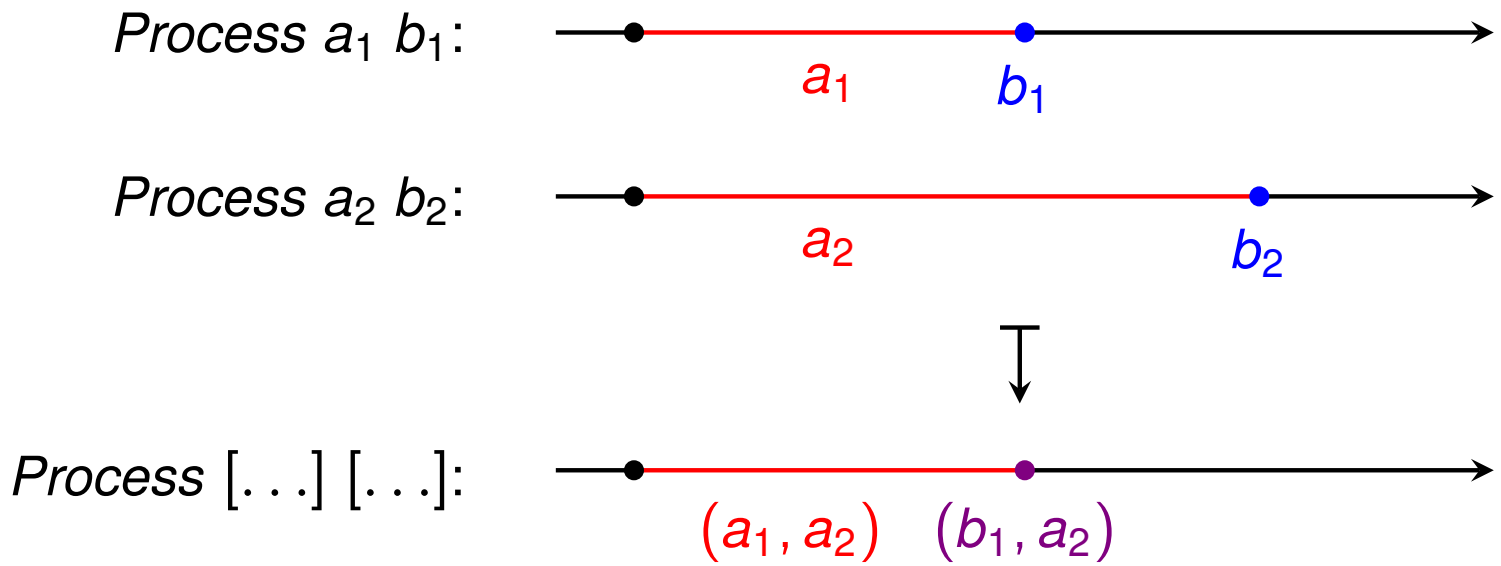


Vereinigen

- Vereinigen zweier Prozesse:

$$\begin{aligned} \mathit{merge} \;::\; & (\mathit{Process}\ a_1\ b_1, \mathit{Process}\ a_2\ b_2) \\ & \longrightarrow \mathit{Process}\ (a_1, a_2)\ (\mathit{RaceResult}\ a_1\ b_1\ a_2\ b_2) \end{aligned}$$

$$\begin{aligned} \textbf{data}\ \mathit{RaceResult}\ a_1\ b_1\ a_2\ b_2 = {} & \mathit{FstWins}\ \ b_1\ a_2 \\ \mid {} & \mathit{SndWins}\ a_1\ b_2 \\ \mid {} & \mathit{Tie}\ \ \ \ \ \ b_1\ b_2 \end{aligned}$$

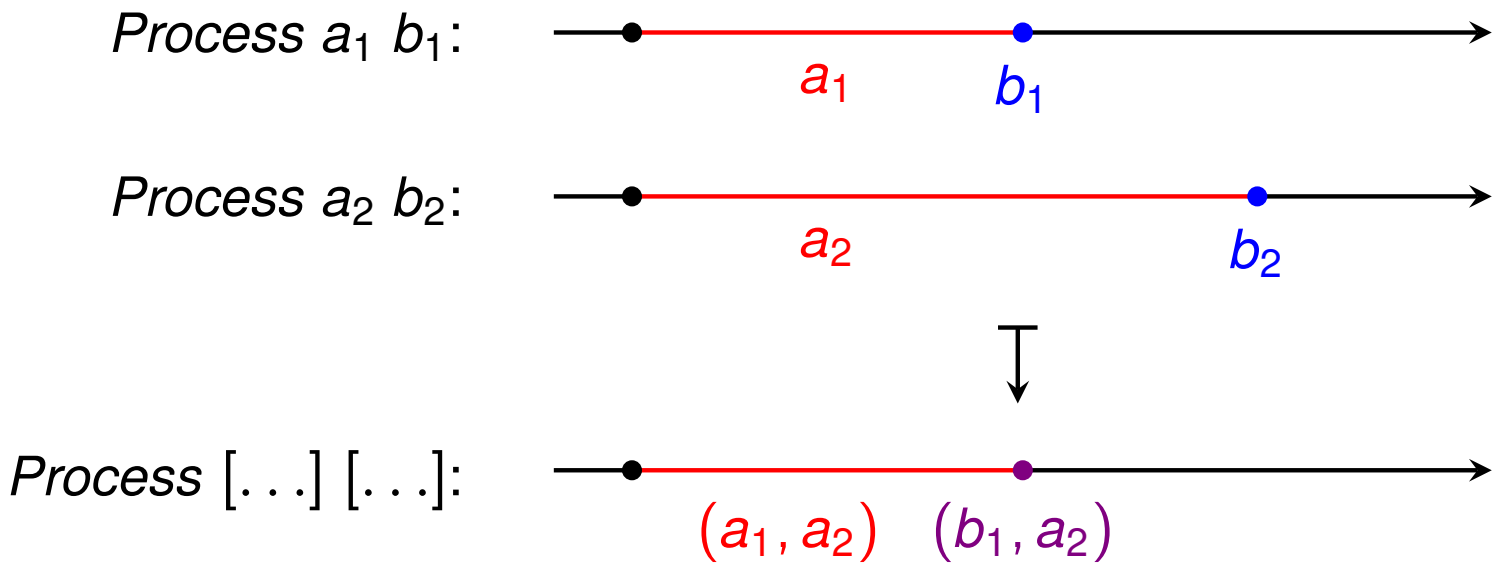


Vereinigen

- Vereinigen zweier Prozesse:

$$\begin{aligned} \mathit{merge} \;::\; & (\mathit{Process}\ a_1\ b_1, \mathit{Process}\ a_2\ b_2) \\ & \longrightarrow \mathit{Process}\ (a_1, a_2)\ (\mathit{RaceResult}\ a_1\ b_1\ a_2\ b_2) \end{aligned}$$

$$\begin{aligned} \textbf{data}\ \mathit{RaceResult}\ a_1\ b_1\ a_2\ b_2 = {} & \mathit{FstWins}\ b_1\ a_2 \\ \mid {} & \mathit{SndWins}\ a_1\ b_2 \\ \mid {} & \mathit{Tie}\ b_1\ b_2 \end{aligned}$$

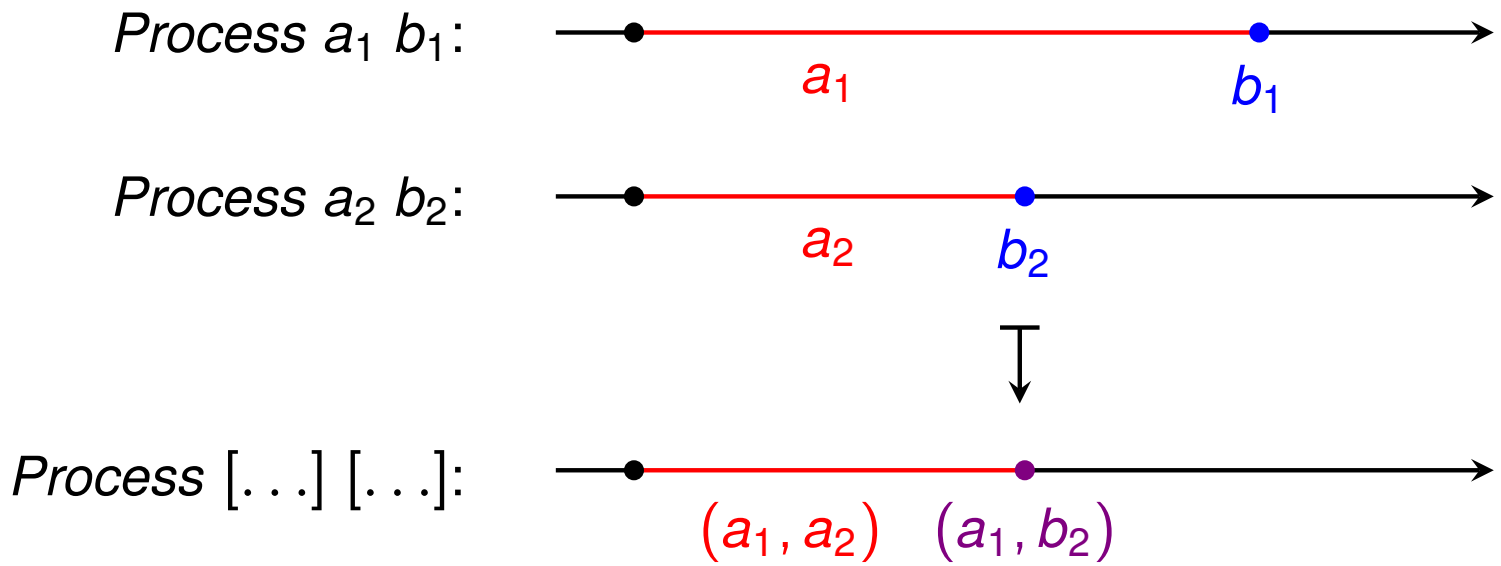


Vereinigen

- Vereinigen zweier Prozesse:

$$\begin{aligned} \mathit{merge} \;::\; & (\mathit{Process}\ a_1\ b_1, \mathit{Process}\ a_2\ b_2) \\ & \longrightarrow \mathit{Process}\ (a_1, a_2)\ (\mathit{RaceResult}\ a_1\ b_1\ a_2\ b_2) \end{aligned}$$

$$\begin{aligned} \textbf{data}\ \mathit{RaceResult}\ a_1\ b_1\ a_2\ b_2 = {} & \mathit{FstWins}\ \ b_1\ a_2 \\ \mid {} & \mathit{SndWins}\ a_1\ b_2 \\ \mid {} & \mathit{Tie}\ \ \ b_1\ b_2 \end{aligned}$$

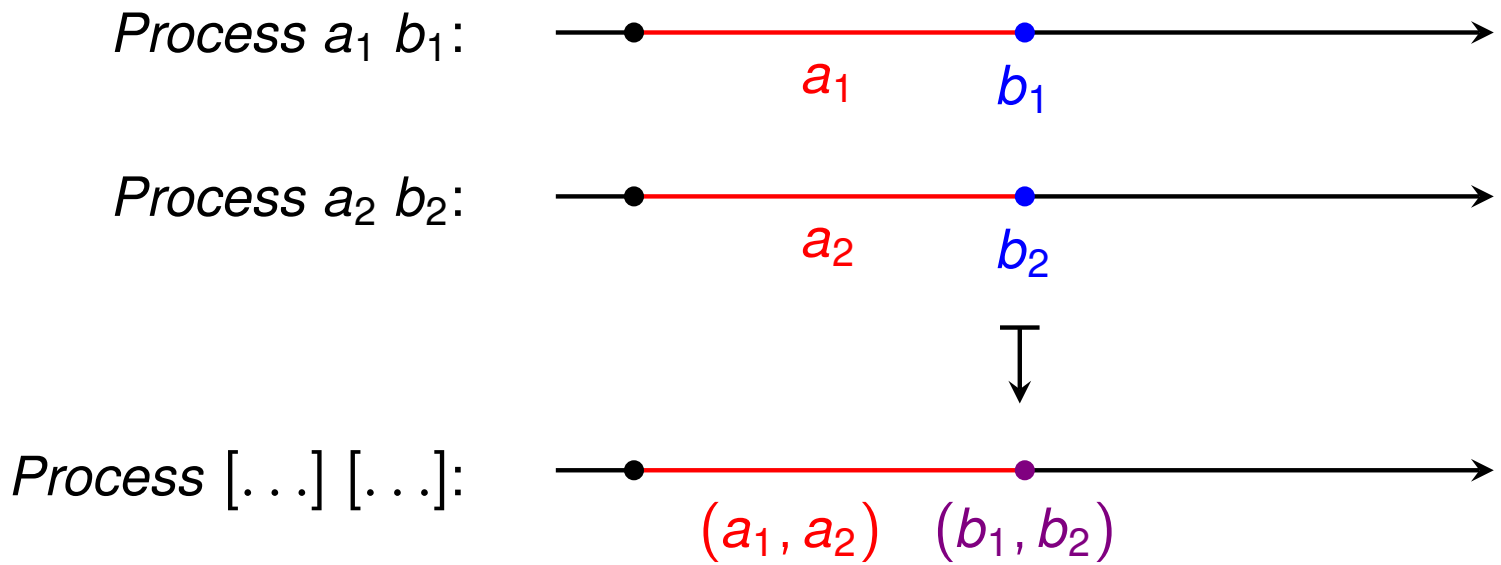


Vereinigen

- Vereinigen zweier Prozesse:

$$\begin{aligned} \mathit{merge} \;::\; & (\mathit{Process}\ a_1\ b_1, \mathit{Process}\ a_2\ b_2) \\ & \longrightarrow \mathit{Process}\ (a_1, a_2)\ (\mathit{RaceResult}\ a_1\ b_1\ a_2\ b_2) \end{aligned}$$

$$\begin{aligned} \textbf{data}\ \mathit{RaceResult}\ a_1\ b_1\ a_2\ b_2 = {} & \mathit{FstWins}\ \ b_1\ a_2 \\ \mid {} & \mathit{SndWins}\ a_1\ b_2 \\ \mid {} & \mathit{Tie}\ \ \ \ \ \ b_1\ b_2 \end{aligned}$$

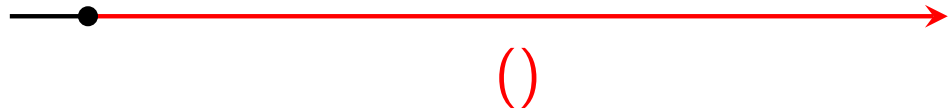


Kanonischer nicht-terminierender Prozess

- Konstruktion eines kanonischen nicht-terminierenden Prozesses:

$$cnp :: () \longrightarrow Process\ ()\ Void$$

Process () *Void*:



- nullstellige Variante des (zweistelligen) Vereinigens

Zusammenfassung

- FRP als deklarativer Ansatz zur Programmierung reaktiver Systeme
- theoretisch fundierte API
- zeitabhängige Typzugehörigkeit zentral für Prüfung temporaler Bedingungen zur Compile-Zeit

Ausblick

- Integration von Seiteneffekten
- Implementierung als Haskell-Bibliothek
- Anwendung in verschiedenen Bereichen:
  - Signalverarbeitung
  - Benutzerschnittstellen
  - Webanwendungen

  usw.
- verteiltes FRP