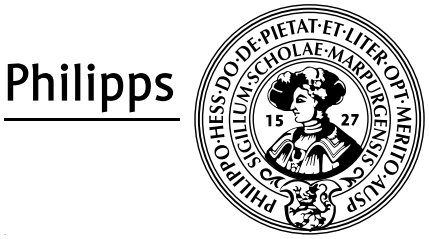


Fachbereich Mathematik & Informatik
AG Programmiersprachen und Parallelität
Prof. Dr. R. Loogen

Seminar "Fun of Haskell Programming"

Ausarbeitung zum Thema

Funktionales reaktives Programmieren mit Yampa

von

Florian Pfeiffer
Matr.-Nr. 2143593

Betreuerin: Prof. Dr. R. Loogen

Basierend auf:
Paul Hudak, Antony Courtney, Henrik Nilsson, John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159-187. Springer-Verlag, 2003.

1 Einleitung

Diese Arbeit befasst sich mit der Realisierung *reaktiver Systeme* in funktionalen Sprachen am Beispiel von *Yampa*[1], einer in Haskell eingebundenen, domänenspezifischen Sprache, als eine Instanz von FRP.

FRP, oder Functional Reactive Programming (funktionelles, reaktives Programmieren) ist hierbei ein Modell für das deklarative Programmieren von *hybriden Systemen*, also Systemen mit kontinuierlichen sowie diskreten Komponenten. Yampa spezialisierte sich auf die Domäne der Robotik, also das Programmieren von mobilen Robotern – inzwischen ist aber auch z.B. ein auf OpenGL basierter First Person Shooter[2] in Yampa implementiert worden (siehe [Che05]).

Die ursprüngliche Frage hinter der Entstehung von FRP war, ob der hohe Abstraktionsgrad funktionaler Sprachen in der realen Welt von Nutzen sein kann; spezifischer, ob

1. funktionale Sprachen wie Haskell in der realen Welt anwendbar sind,
2. diese Anwendung darüber hinaus sogar Vorteile hat, und schließlich,
3. ob Performanzprobleme überwunden werden können.

Die erste Manifestation von FRP (und eigentlich sogar die Sprache, aus der FRP letztendlich hervorging) war *Fran*, eine DSL für Graphik und Animation. Hudak, Courtney, Nilsson und Peterson beschreiben in [HCNP03] FRP als die "Essenz" von Fran, jedoch ohne Spezialisierung auf bestimmte Themengebiete. Aus FRP selbst gingen wiederum verschiedene DSLs hervor, ob nun für Verkehrsüberwachung per UAV (Unmanned Aerial Vehicle, unbemanntes Luftfahrzeug), für GUIs oder eben für Robotik (siehe Abbildung 1).

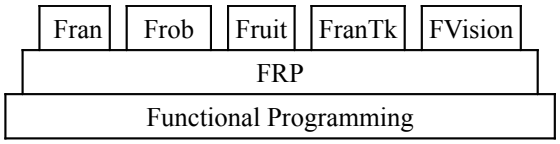


Abbildung 1: Eingebundene DSLs

Der bedeutendste Unterschied in Yampa zum ursprünglichen FRP ist wohl die Repräsentation durch *Arrows*[3]. Somit ist Yampa auch für Systeme, die Echtzeitberechnungen benötigen, praktikabel. Tatsächlich wurde die DSL bereits erfolgreich zur Programmierung von Industrierobotern verwendet. Für diejenigen, die keinen Zugang zu derartiger Technik haben, gibt es allerdings auch einen Simulator, der auch in dieser Arbeit verwendet werden wird. *Caveat:* Der Simulator stellt natürlich

[1] http://www.haskell.org/yampa

[2] Als First Person Shooter (FPS) bezeichnet man ein Computerspiel, in dem der Spieler aus der Egoperspektive in einer dreidimensionalen Spielwelt mit verschiedenen Schusswaffen vom Computer oder anderen Spielern gesteuerte Gegner bekämpft.

[3] Arrows sind eine Generalisierung von Monaden, die, wie später deutlich wird, die Programmierung konsistent macht.

eine Idealumgebung dar, in der die Hindernisse und Schwierigkeiten, mit denen in der realen Welt zu rechnen ist, nicht auftreten.

2 Yampa und FRP

2.1 Yampa-Grundlagen

Das wichtigste Konzept hinter FRP (und damit auch hinter Yampa) ist das eines *Signals*. Ein Signal ist ein beliebiger variabler Wert, der sich mit der Zeit verändert. Somit stellt es quasi eine Implementation von aus der Schulphysik bekannten Funktionen der Zeit, wie z.B. Beschleunigung oder Geschwindigkeit, dar. Man kann sich ein Signal mit einem polymorphen Typ vorstellen, z.B. `Signal a = Time → a`.

Ein Signal vom Typ `a`, also ein Wert vom Typ `Signal a`, ist damit eine Funktion, die die Zeit auf einen Typ `a` abbildet. In der Yampa-Implementation wird für `Time` der Typ `Double` verwendet.

Beispiel: Die Geschwindigkeit eines Differential Drive Robots ist ein Paar der beiden Geschwindigkeiten der linken und der rechten Räder. In Yampa ist ein Typ `Speed` als Synonym von `Double` definiert, und somit läßt sich die Geschwindigkeit des Roboters mit dem Typ `Signal (Speed, Speed)`, in unserer Vorstellung also `Time → (Speed, Speed)` repräsentieren. ◁

Das Fundamentale an diesem Konzept ist die Ausdruckskraft, die dem Programmierer damit verliehen wird. Die Möglichkeit, stetige Werte zu definieren und zu manipulieren, verleiht dem Programmierer ein riesiges Potential.

Beispiel: Laut [DJ00] lassen sich die Gleichungen zur Steuerung eines Differential Drive Robots folgendermaßen als Funktionen der Zeit darstellen:

$$\begin{aligned} x(t) &= \frac{1}{2}\int_0^t (v_r(t) + v_l(t))\cos(\Theta(t))\,dt \\ y(t) &= \frac{1}{2}\int_0^t (v_r(t) + v_l(t))\sin(\Theta(t))\,dt \\ \Theta(t) &= \frac{1}{l}\int_0^t (v_r(t) - v_l(t))\,dt \end{aligned}$$

$x(t)$ und $y(t)$ sind hierbei die Koordinaten des Roboters, $\Theta(t)$ die Orientierung, $v_l(t)$ und $v_r(t)$ sind die Geschwindigkeiten der jeweils linken und rechten Räder, und l der Abstand zwischen den linken und rechten Rädern.

Durch `Signal`s lassen sich diese Gleichungen direkt in FRP ausdrücken (vorausgesetzt, `(*)`, `(+)`, `sin`, `cos` etc. sind für `Signal`s passend überladen):

```
x     = (1/2) * integral ((vr + vl) * cos theta)
y     = (1/2) * integral ((vr + vl) * sin theta)
theta = (1/l) * integral (vr - vl)
```

Da `Signal`s bereits implizit von der Zeit abhängig sind, kann man auf ein explizites `t` verzichten – die Zuweisungen sind quasi "point free". Trotzdem kann man erkennen, dass für die Portierung von physikalischen Gleichungen zu Zuweisungen in FRP fast kein Aufwand nötig ist. ◁

2.2 Arrows, AFRP und Signalfunktionen

> *"Wouldn't a Haskell robot just sit there and be lazy? Or actually, I guess if you told it to do something it would delay doing it until you actually demanded the results."*
>
> — A Sceptic

Dieses Zitat findet sich auf der alten Website der *Frob*-Bibliothek – und der "Skeptiker" hat nicht ganz unrecht. Techniken wie Lazy Evaluation, die sonst viele Vorteile mit sich bringen, können bei hybriden Systemen jede Hoffnung auf Echtzeitverhalten zerstören. Obwohl also das Konzept von `Signal`s ein sehr allgemeines ist, muss der Programmierer entweder seine Roboter von Anfang an sehr präzise und aufmerksam designen, oder aber bei jedem Testlauf mit frustrierenden (und eigentlich unnötigen) Problemen kämpfen – sogenannten "time- and space-leaks".

Um dieses Problem zu überwinden, gab es verschiedene Ansätze, die von den Autoren von Yampa nicht genauer benannt wurden (siehe [HCNP03]). Diese Ansätze brachten aber nicht den gewünschten Erfolg, da entweder das Problem nur verlagert wurde oder neue Probleme auftraten. In Yampa selbst ist das Problem einfach "verboten" worden: `Signal`s sind nicht mehr als First Class-Werte erlaubt. Der Programmierer arbeitet auf der Ebene von *"Signal Transformers"*, wie sie noch in AFRP (Arrowized FRP) genannt wurden, oder *"Signal Functions"*, die Bezeichnung in Yampa. Eine "Signal Function" (Signalfunktion) ist eine Funktion, die Signale auf Signale abbildet, also:

```
SF a b = Signal a → Signal b
```

`SF` selbst ist ein abstrakter Datentyp. Das heißt für den Programmierer, dass er nicht nach Belieben Signalfunktionen kreieren kann (was das oben angesprochene Problem wieder nur verlagert hätte). Es gibt eine Menge primitiver Signalfunktionen und eine weitere Menge von Kompositionsoperatoren (oder "Combinators"), womit die Generierung komplexerer Signalfunktionen möglich wird. Allerdings vermeidet auch dieser streng disziplinierte Ansatz time- und space-leaks erst durch die Strukturierung von Yampa: die Bibliothek basiert auf *"Arrows"*. Arrows sind eine von John Hughes in [Hug00] vorgestellte und von Ross Paterson implementierte Generalisierung von Monaden. Yampa nutzt dies aus, indem `SF` eine Instanz der `Arrow`-Klasse ist.

Ein fehlerfreier `RobotController` ist also effektiv nichts anderes als eine komplexe Signalfunktion, die aus primitiven Signalfunktionen zusammengesetzt ist und dann von einem passenden Interpreter (in unserem Fall `AFrob.RobotSim`) ausgeführt wird.

Wer sich hier an *IO-Monaden*, wie sie in der Vorlesung *"CS 310: Praktische Informatik III: Konzepte von Programmiersprachen"* behandelt wurden, oder an *State-Monaden* erinnert fühlt, liegt nicht ganz falsch: in [HCNP03] wird das Konzept, dass der Status selbst nicht sichtbar ist, und ein Programm aus einer Befehlssequenz besteht, die wiederum von einem Interpreter oder dem Betriebssystem ausgeführt wird, als eine gute Analogie beschrieben.

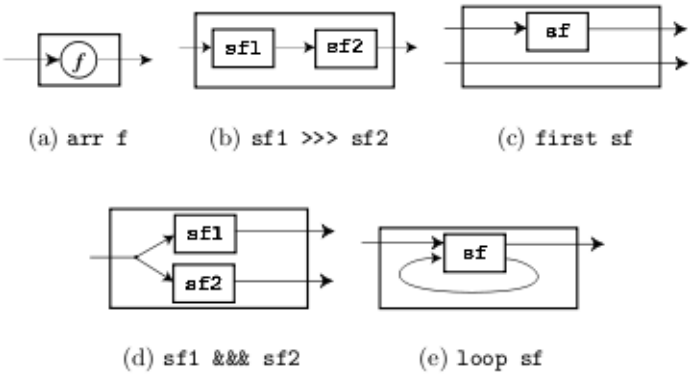
Caveat: Trotzdem gibt es einen grundlegenden Unterschied: wie bereits erwähnt, sind `Arrow`s allgemeiner als Monaden. Insbesondere sind nicht alle Kombinatoren

komplett linear (was auch nicht wünschenswert wäre, denn in den seltensten Fällen möchte man einen sequentiellen `RobotController`).

2.2.1 Kombinatoren und Library Functions

Yampa bietet eine ganze Reihe von Kombinatoren an. Eine Auswahl findet sich in Abbildung 2.

```
arr    :: Arrow a ⇒ (b → c) → a b c
arr2   :: (a → b → c) → SF (a,b) c
(>>>)  :: Arrow a ⇒ a b c → a c d → a b d
(<<<)  :: Arrow a ⇒ a c d → a b c → a b d
first  :: Arrow a ⇒ a b c → a (b,d) (c,d)
second :: Arrow a ⇒ a b c → a (d,b) (d,c)
(***)  :: Arrow a ⇒ a b c → a b' c' → a (b,b') (c,c')
(&&&)  :: Arrow a ⇒ a b c → a b c' → a b (c,c')
loop   :: Arrow a ⇒ a (b,d) (c,d) → a b c
```



(a) arr f (b) sf1 >>> sf2 (c) first sf

(d) sf1 &&& sf2 (e) loop sf

Abbildung 2: Typische Kombinatoren in Yampa

Im Folgenden werden einige dieser Kombinatoren vorgestellt. Die Funktionen der meisten Kombinatoren sollten durch die Typen und die graphische Darstellung selbsterklärend sein; der Vollständigkeit halber folgt hier trotzdem eine kurze Auflistung der obigen Kombinatoren inklusive Funktionalität:

- Der wahrscheinlich wichtigste Kombinator ist `arr`. Wie die Typdefinition vermuten läßt, ist dies ein einstelliger Operator, der eine normale Funktion in eine Signalfunktion umwandelt.
- `arr2` funktioniert analog zu `arr`, nur dass `arr2` für binäre Funktionen gedacht ist. Der Kombinator ist äquivalent zu `arr ∘ uncurry`.
- `(<<<)` und `(>>>)` stellen die normale und die inverse Komposition von Signalfunktionen dar.
- `first` und `second` wenden eine Signalfunktion nur auf den ersten bzw. zweiten Wert in einem Paar an.

- `(&&&)` leitet ein Argument an zwei Signalfunktionen weiter. Die Ergebnisse werden als Paar zurückgegeben.
- `(***)` realisiert das sogenannte *"Argument Wiring"*. Das Paar, welches die beiden Argumente enthält, wird "aufgeteilt", das erste Argument von der ersten und das zweite von der zweiten Signalfunktion ausgewertet. Die Ergebnisse werden wieder als Paar zurückgegeben.
- `loop` wird für rekursiv definierte Signalfunktionen gebraucht. Aus Gründen, die später noch klar werden, wird hier noch nicht weiter darauf eingegangen.

Nota bene: diese Menge von Kombinatoren enthält bereits einige minimale Untermengen, beispielsweise `{arr, (>>>), (&&&)}` und `{arr, (>>>), first}`. Die jeweils anderen Kombinatoren lassen sich dadurch ausdrücken.

Beispiel:

`(***)` mit `{arr, (>>>), (&&&)}`:

```
(***) :: SF b c → SF b' c' → SF (b,b') (c,c')
f *** g = (arr fst >>> f) &&& (arr snd >>> g)
```

`((&&&), second)` mit `{arr, (>>>), first}`:

```
second :: Arrow a ⇒ a b c → a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
           where swap pr = (snd pr, fst pr)

(&&&) :: Arrow a ⇒ a b c → a b d → a b (c,d) \\
f &&& g = arr (λb → (b,b)) >>> (first f >>> second g)
```

◁

Abgesehen von den Kombinatoren liefert Yampa noch viele vordefinierte Signalfunktionen, von denen in Abbildung 3 ein paar vorgestellt werden. Dabei sind zwei Dinge zu beachten:

1. `derivative` und `integral` sind für sämtliche Vektorräume überladen. Der Einfachheit halber (und weil es sonst den Rahmen sprengen würde) sind sie hier aber auf `Double` spezialisiert.
2. Einige Signalfunktionen sind "zustandsbasiert". Hier ist `integral` ein sehr gutes Beispiel: per Definition addiert die Funktion immer wieder neue Werte zu ihrem jeweiligen Ergebnis. Sie ist also stark von der Zeit abhängig. Da `arr` und `arr2` nur "reine" Funktionen ("pure functions") transformiert, ist es unmöglich, auf diese Art zustandsbasierte Funktionen zu erzeugen. Diese Funktionen müssen entweder vordefiniert, also bereits in der Yampa-Bibliothek, sein, oder aber mittels der vordefinierten zustandsbasierten Funktionen erzeugt werden. Vice versa existieren zu zustandsbasierten Signalfunktionen keine normalen Haskell-Funktionen.

```
identity   :: SF a a
constant   :: b → SF a b
time       :: SF a Time
integral   :: SF Double Double
derivative :: SF Double Double
```

Abbildung 3: Vordefinierte Signalfunktionen in Yampa

2.2.2 Der Arrow-Präprozessor

Mit dem gesammelten Wissen über Signalfunktionen ist es jetzt möglich, den im Beispiel gezeigten Code für `x` in Yampa zu schreiben:

Beispiel:

Angenommen, es existieren die folgenden Signalfunktionen:

```
vrSF, vlSF :: SF RobotInput Speed  -- Variablen vr, vl...
thetaSF :: SF RobotInput Angle     -- ...und theta aus dem Beispiel
```

Mit der zur Verfügung stehenden Menge an Signalfunktionen und Kombinatoren läßt sich der Code als Signalfunktion schreiben:

```
xSF :: SF RobotInput Distance
xSF = let v = (vrSF &&& vlSF) >>> arr2 (+)
          t = thetaSF >>> arr cos
      in (v &&& t) >>> arr2 (*) >>> integral >>> arr(/2)
```

◁

Wie man schnell feststellt, ist die Funktion durch die vielfache Anwendung der Kombinatoren recht unübersichtlich geworden. Aus genau diesem Grund hat Ross Paterson einen Präprozessor für die sogenannte *"arrow syntax"* geschrieben, die den Code lesbarer machen soll[4] (siehe [Pat01]):

Beispiel:

```
xSF' :: SF RobotInput Distance
xSF' = proc inp → do
  vr    ← vrSF    -< inp
  vl    ← vlSF    -< inp
  theta ← thetaSF -< inp
  i     ← integral -< (vr+vl) * cos theta
  returnA -< (i/2)
```

◁

Dem erfahrenen Haskell-Programmierer wird eine gewisse Analogie zu λ-Ausdrücken nicht entgangen sein. In der Tat ist die Syntax `proc pat →` analog zu einem Ausdruck der Form λ `pat →` – nur, dass hier natürlich eine Signalfunktion anstelle einer normalen Haskell-Funktion definiert wird. Gilt pat_0 `::` `a` und $expr_n$ `::` `b`, so ist die Funktion vom Typ `SF a b`.

Die in Abbildung 4 vorgestellte *arrow syntax* ist recht einprägsam, wobei, wenn $SFexpr_i$ `::` `SF T1 T2` gilt,

[4]Dieser Präprozessor ist seit Version 6.2 in GHC integriert und das Paket damit nicht mehr benötigt – man kann ghc oder ghci einfach mit der Option *"-farrows"* starten.

```
proc pat0 → do
  pat1 ← SFexpr1 -< expr1
  pat2 ← SFexpr2 -< expr2
  -- etc.
  returnA -< exprn
```

Abbildung 4: Arrow Syntax

- $expr_i$ vom Typ `T1` und
- pat_i vom Typ `T2` sein muss.

Das ist wiederum analog zu `let pat = expr1 expr2`, wo, wenn $expr_1$ vom Typ `T1 → T2` ist, $expr_2$ `:: T1` und pat `:: T2` gelten muss. Weiterhin ist eine Variable, die an pat_i gebunden wird, erst danach verfügbar, kann also nur von $expr_j, i < j \leq n$ verwendet werden. Insbesondere können gebundene Variablen *nie* in $SFexpr_{1 \leq i \leq n-1}$ benutzt werden.

Mit obiger Vorstellung von `Signal`s ist es hier leicht (aber wichtig), festzustellen, dass wenn `SF a b` gerade der Typ `Signal a → Signal b` ist, was wiederum für den Typ `(Time → a) → (Time → b)` steht, der Programmierer zwar Zugriff auf die Werte vom Typ `a` und `b` hat, nicht aber auf die vom Typ `Time → a` oder `Time → b`.

Als letztes ist noch festzustellen, dass der Code in *arrow syntax* sehr nah an den korrespondierenden *Signal Flow Diagrams* ist. Abbildung 5 zeigt als Beispiel das Diagramm für `xSF'`. Sollte der Programmierer also doch den Überblick verlieren, kann er sich seine `RobotController`s vorzeichnen.

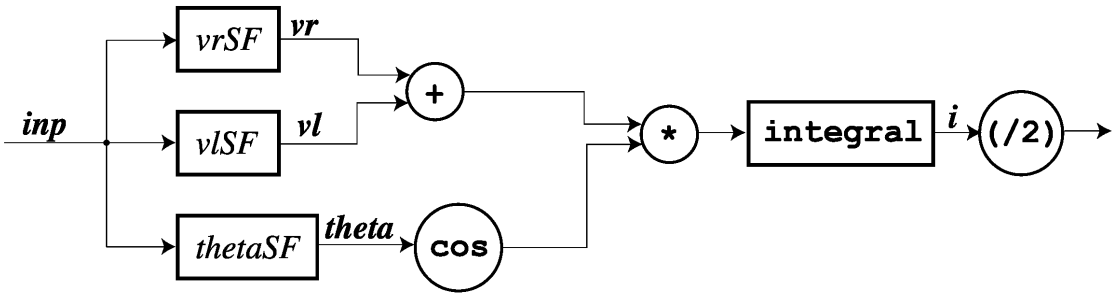


Abbildung 5: Signal Flow Diagram für `xSF'`

2.3 Ereignisse und Switches

Mit den bisher vorgestellten Techniken lassen sich nur lineare Signalfunktionen programmieren. Jedoch wird man sich damit nur in den seltensten Fällen zufrieden geben. Insbesondere sind die meisten Roboter ohne irgendeine Art von Verzweigung nutzlos.

Die wohl einfachste Art, Verzweigung in Yampa zu erzeugen, ist mit einer einfachen `if`-Klausel. Angenommen, es existieren die Funktionen

```
type Behavior a = Time → a  --simplified
type Event    a = [(Time,a)] --simplified
```

Abbildung 6: Ursprüngliche Unterscheidung zwischen `Behavior` und `Event` in *Frob*

```
flag     :: SF a Bool
sfx, sfy :: SF a b
```

Dann ließe sich eine Signalfunktion wie folgt schreiben:

```
sf :: SF a b
sf = proc i → do
  x ← sfx  -< i
  y ← sfy  -< i
  b ← flag -< i
  returnA -< if b then x else y
```

Diese Signalfunktion wird sich immer dann wie `sfx` verhalten, wenn `flag` als Wert `True` zurückgibt; ansonsten wie `sfy`.

Dies liefert zwar eine Möglichkeit zur bedingten Verzweigung, aber in vielen Situationen wird das nicht reichen, vor allem wenn man mehrere Signalfunktionen auf unterschiedliche Weise miteinander verknüpfen will. Dies wäre zwar auch möglich, aber nur mit einem unproportional großen Overhead an Workarounds.

Was für den Programmierer wünschenswert wäre, läßt sich mit einem NFA aus der Automatentheorie vergleichen, in dem die Zustände einzelne Controller (also lineare Verhaltensweisen eines Roboters) repräsentieren, und die Transitionen durch bestimmte Ereignisse ausgelöst werden.[5] Aber auch das ist in Yampa möglich.

2.3.1 Diskrete und kontinuierliche Werte

Wie Abbildung 6 zeigt, wurde in früheren FRP-Versionen wie z.B. *Frob* sehr stark zwischen zwei Ereignistypen unterschieden (siehe [PHE99]):

1. **diskrete Ereignisse**, welche zu einem bestimmten Zeitpunkt eintreten und keine meßbare Dauer haben (beispielsweise ein Mausklick oder der genaue Zeitpunkt, an dem ein stetig steigender Wert eine festgelegte Grenze überschreitet), und

2. **kontinuierliche Werte**, prinzipiell Funktionen, die immer einen bestimmten Wert zurückgeben. Hierzu würde bei einem Roboter beispielsweise ein Sonar, Radar oder seine Temperatur zählen, aber auch unabhängige Werte wie z.B. Zeit.

Spätestens seit 2002 verfolgt man einen anderen Ansatz: Der `Behavior`-Typ ist inzwischen offensichtlich in Signalfunktionen gekapselt, und `Event a` ist inzwischen isomorph zu `Maybe a`[6], wie Abbildung 7 zeigt. Für ein Ereignis vom Typ `Event T`

[5] In der Tat wird ein derartiges Konzept in der künstlichen Intelligenz eingesetzt, z.B. zur Programmierung von Gegnern oder generell computergesteuerten Charakteren in Computerspielen.

[6] `Event` ist genauso überladen wie `Maybe`, also z.B. eine Instanz der Typklasse `Functor`.

nennt sich der Datentyp `Signal (Event T)` *"Event Stream"*, der zu jeder Zeit einen Wert hat, entweder `NoEvent` oder eben `Event T`. Eine Signalfunktion, die diesen Stream erzeugt, nennt sich "Event Source" und hat den Typ `SF a (Event b)`.

Damit ist die Unterscheidung zwischen diskret und kontinuierlich längst nicht mehr so signifikant: ein *Event Source* ist genauso kontinuierlich abrufbar wie jede andere Signalfunktion – nur eben für die Zeit, in der das Ereignis nicht eintrifft, hat der gelieferte Stream keinen Wert (`NoEvent`).

```
data Event a = Event a -- an event occurence
             | NoEvent -- a non-occurrence
```

Abbildung 7: Ereignisse in AFRP und Yampa

Auch hier bietet Yampa wieder einige recht nützliche vordefinierte Event Sources. Der Ausdruck `boolSF >>> edge` generiert beispielsweise jedes Mal, wenn das Signal von `boolSF` von `False` auf `True` springt, ein `Event`. Dies bietet z.B. eine einfache Methode, Warn- oder Alarmsignale als Event Sources zu generieren. In Abbildung 8 findet sich ein Beispiel mit der (fiktiven) Signalfunktion `tempSF :: SF RobotInput Temp`, die die Temperatur eines Roboters zurückgibt. Die restlichen Event Sources sind selbsterklärend.

```
edge        :: SF Bool (Event ())
never       :: SF a (Event b)
now         :: b → SF a (Event b)
after       :: Time → b → SF a (Event b)
repeatedly :: Time → b → SF a (Event b)

-- Beispiel für edge mit einer (fiktiven) Signalfunktion
alarmSF :: SF RobotInput (Event ())
alarmSF = tempSF >>> arr (>100) >>> edge
```

Abbildung 8: Vordefinierte Event Sources in Yampa mit Beispiel für `edge`

2.3.2 Simultane Ereignisse

Will man einen Controller auf mehrere verschiedene Ereignisse vorbereiten, muss man in Betracht ziehen, dass mehrere dieser Ereignisse gleichzeitig auftreten könnten.

Beispiel:

Angenommen, man hat einen Controller vom Typ

```
rcThruMaze :: Velocity → Distance → RobotController
```

Dieser Controller ist folgendermaßen konfiguriert:

1. Richte Dich an der Wand links von Dir aus, so dass Du d m von ihr entfernt bist und parallel zu ihr stehst.

2. Fahre mit der Geschwindigkeit v geradeaus (also parallel zur Wand), bis

- Dein Sonar die Wand links von Dir nicht mehr findet. In diesem Fall drehe Dich um 90 Grad nach links.
- Dein Sonar eine Wand direkt vor Dir (i.e. in weniger als d m Entfernung sieht. In diesem Fall drehe Dich um 90 Grad nach rechts.

O.B.d.A. bestehe die Welt aus einem sehr einfachen Labyrinth: in ein Raster unterteilt, ist ein "Kästchen" entweder eine Mauer oder Weg. Dann reicht es schon, eine T-Kreuzung zu betrachten, um ein fundamentales Problem aufzuzeigen:

Wie man schnell sieht, sind beide Fälle eingetreten: der Sonar findet links gar keine Wand mehr, aber sieht direkt vor dem Roboter eine neue Wand in weniger als d m Entfernung. Der Roboter müßte nun versuchen, sich gleichzeitig nach rechts und nach links zu drehen.

◁

Man sieht also, dass es nicht reicht, einfach nur jedes `Event` abzufangen. Yampa kennt für solche Aufgaben insgesamt vier verschiedene `merge`-Funktionen, die in Abbildung 9 aufgelistet sind.

```
mergeBy :: (a → a → a) → Event a → Event a → Event a
lMerge, rmerge, merge  :: Event a → Event a → Event a
```

Abbildung 9: `merge`-Funktionen für `Event`s

`mergeBy` ist wohl die generellste der vier Funktionen: der Programmierer selbst gibt eine Funktion an, mit der gemergt wird. `lMerge` und `rMerge` geben dem "linken" bzw. dem "rechten" (i.e. dem ersten bzw. zweiten) `Event` den Vorzug. `merge` darf nur dann verwendet werden, wenn es keine Möglichkeit gibt, dass beide `Event`s simultan auftreten. Sollte dies trotzdem passieren, gibt die Funktion eine Fehlermeldung aus.

2.3.3 Switches

Switches sind die fundamentalste und wichtigste Art, mit `Event`s zu arbeiten. Mit ihnen kann man eine Signalfunktion in eine andere überführen. Dies erfüllt auch die oben angesprochenen Erwartungen, mehrere Signalfunktionen auf mehrere (möglicherweise recht komplexe) Arten zu verbinden.

Die vier wichtigsten Switches (zumindest für diese Arbeit) unterscheiden sich in zwei Aspekten:

1. ***D*elayed** Switches passieren nicht gleichzeitig mit dem `Event`, sondern direkt danach. Das ist vor allem für Rekursion von enormer Bedeutung (vgl. hierzu die rekursiven Listendefinitionen `ones = 1:ones` (delayed) und `ones = ones` (not delayed – und auch nicht brauchbar).

2. ***R*ecurring** gibt an, dass der Switch nicht nur für das erste, sondern für alle `Event`s im Event Stream gilt. Siehe dazu das Beispiel auf Seite 12.

```
switch,  dSwitch  :: SF a (b, Event c) → (c → SF a b) → SF a b
rSwitch, drSwitch :: SF a b → SF (a, Event (SF a b)) b
```

Abbildung 10: Vier der Möglichkeiten, in Yampa zu "switchen"

Somit hat der Programmierer mindestens vier Möglichkeiten, zu "switchen". Die korrespondierenden Funktionen sind in Abbildung 10 aufgelistet.

Die wichtigste Eigenschaft von Switches ist, dass die Zeitmessung mit jedem Switch wieder von neu beginnt. So ist es z.B. möglich, eine Sinuskurve zu generieren und bei einem bestimmten `Event`, unabhängig von der Zeit, das Signal wieder neu (also bei 0) zu starten:

```
let sinSF = time >>> arr sin
in (sinSF &&& rsStuck) 'switch' const sinSF
```

Außerdem sind noch `pSwitch` und `rpSwitch` verfügbar. Das `p` steht hierbei für "parallel", die Switches ermöglichen die Manipulation dynamischer Collections von Signalfunktionen (siehe [NCP02]).

2.3.4 Sonstige nützliche Funktionen

Wie bereits erwähnt, ist `Event` eine Instanz der Klasse `Functor`. Das bedeutet z.B., dass die Funktion

```
fmap :: (Functor f) ⇒ (a → b) → f a → f b
```

benutzt werden kann, um den Wert eines `Event`s zu modifizieren. So kann man beispielsweise den Wert eines `Event`s vom Typ `e :: Event Double` einfach mit dem Ausdruck `fmap (+1) e` inkrementieren.

Für den Fall, dass der alte Wert eines `Event`s für die Signalfunktion und/oder den Programmierer nicht von Belang ist, gibt es in Yampa die Funktion "`tag`":

```
tag :: Event a → b → Event b
e 'tag' b = fmap (const b) e
```

Diese Funktion überschreibt den alten Wert eines `Event`s mit einem beliebigen neuen Wert. Somit ist es z.B. möglich, vordefinierten `Event`s (auch solchen vom Typ `Event ()`) einen `String` zuzuweisen, mit dem man das `Event` besser identifizieren kann.

Man sieht also, dass die diskrete und die kontinuierliche Welt eng miteinander verknüpft sind. Auch wenn die wichtigsten Arten der Interaktion genannt wurden; drei Funktionen gilt es hier doch noch vorzustellen (hierzu siehe Abbildung 11).

`hold` generiert anfangs ein Signal, dessen konstanter Wert von dem ersten Parameter bestimmt wird. Jedes Mal, wenn dann ein `Event` eintritt, nimmt `hold` den Wert dieses `Event`s an und behält ihn bis zum nächsten `Event`.

`accum` ist quasi eine Version von `map` für Event Streams: jedes Input Event generiert ein Output Event. Dabei gilt:

$$\begin{aligned} accum\, v_0\ \_ &= v_0 \\ accum\, v_n\, f_n &= f_n\, v_{n-1},\ n \geq 1 \end{aligned}$$

```
hold  :: a → SF (Event a) a
accum :: a → SF (Event (a → a)) (Event a)
accumHold :: a → SF (Event (a → a)) a
```

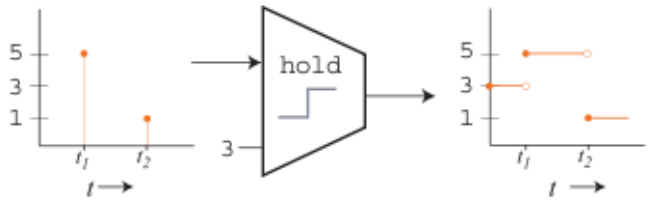


Abbildung 11: graphische Darstellung von `hold`

Da sich herausgestellt hat, dass die Verknüpfung von `accum` und `hold` sehr häufig von Nutzen ist, wurde die Funktion als `accumHold` in Yampa vordefiniert.

2.4 Rekursive Signale

Der aufmerksame Leser mag sich an den `loop`-Kombinator erinnern, der auf Seite 5 auf-, aber nicht eingeführt wurde. Nachdem jetzt die Arrow Syntax, `Event`s und Switches bekannt sind, kann das hier nachgeholt werden.

Wie bereits erwähnt, dient der `loop`-Kombinator der Definition rekursiver Signalfunktionen (er ist ein sogenannter "fixpoint operator"). Auch in der Arrow Syntax ist es möglich, rekursive Signalfunktionen zu schreiben. Mehr noch, die Syntax erlaubt es dem Programmierer, direkt rekursiv zu definieren. Der Präprozessor übersetzt dies dann in "traditionellen" Yampa-Code mit dem `loop`-Kombinator. Allerdings ist dafür das Schlüsselwort `rec` vor den rekursiven Bindings anzugeben.

Eine recht übliche Anwendung ist die Übertragung eines "Schnappschusses" im Falle eines Switches, also eine Übertragung des Signalstatus zum Zeitpunkt des Switches in ein neues Signal, um dessen Wert zu berechnen.

Beispiel:

Angenommen, es existiert ein Event Source vom Typ `SF RobotInput (Event ())` namens `incrVelEvs`, dessen `Event`s bei Eingabe von Kommandos zur Beschleunigung generiert werden. Dann lässt sich die folgende rekursive Signalfunktion definieren:

```
vel :: Velocity → SF RobotInput Velocity
vel v0 = proc inp → do
  rec e ← incrWelEvs -< inp
      v ← drSwitch (constant v0) -< (inp, e 'tag' constant (v+1))
  returnA -< v
```

Diese Funktion reagiert direkt auf den Event Source und damit auch auf besagte Kommandos zur Beschleunigung. Die rekursiv definierte Variable ist hierbei `v`. ◁

3 Der Robot Simulator

3.1 Die Simbots

Der in dieser Arbeit verwendete Simulator kennt nur einen Typ von Roboter (im folgenden "Simbot" genannt). Es handelt sich um einen "differential drive robot", also einen mobilen Roboter mit zwei Motoren, die die linken und die rechten Räder getrennt steuern. Durch zwei unterschiedliche Geschwindigkeiten wird somit eine Drehbewegung simuliert. Weiterhin verfügen die Simbots über

- *"bumper switches"*, die aktiviert werden, sobald der Simbot in ein Hindernis fährt, den Simbot stoppen und ein Signal an das Programm senden;
- *"range finders"*, die das jeweils nächste Objekt nördlich, östlich, südlich und westlich von dem Simbot erkennt;
- einen *"animate object tracker"* (oder *"aot"*), der die Positionen anderer Simbots und bspw. einiger Bälle zurückgibt[7], und
- einige andere Komponenten, die bei Bedarf später vorgestellt werden.

3.1.1 Exkurs in Robotik: Typen, Komponenten und Steuerung

In der Realität gibt es natürlich viele verschiedene Arten von Robotern mit ebenso vielen unterschiedlichen Kombinationen von Komponenten, wie z.B. Lautsprechern, diversen Sensoren, Sonar etc. Dies wird durch die Ein- und Ausgabetypen des Roboters dargestellt. Für die Simbots sind dies beispielsweise die Typen `SimbotInput` und `SimbotOutput`. Da die AFrob-Bibliothek so generisch wie möglich gestaltet wurde, werden Roboter als Instanzen von Teilmengen einer Menge von Typklassen realisiert, die wiederum verschiedene Typen von Funktionalität repräsentieren (siehe Abbildung 12).

Desweiteren gibt es eine Vielzahl von Möglichkeiten, Roboter zu steuern: mit einer Menge von als Vektor dargestellten Kräften läßt sich die Bewegung eines Roboters ebenso gut modellieren wie mit X-, Y-, und Θ-Werten oder -Funktionen. Diese Arbeit wird sich allerdings auf eine direkte Angabe der Geschwindigkeiten als `(Velocity, Velocity)` bzw `(Velocity, RotVel)` beschränken.

3.2 Der Robot Simulator

Der Simulator, auf dem die in dieser Arbeit programmierten Beispiele getestet wurden, ist quasi eine "Sandbox", also eine geschlossene Umgebung, in der Idealbedingungen für die Tests herrschen. Insbesondere sind die Positions- und Distanzwerte von einer Genauigkeit, die man in der realen Welt mit an Sicherheit grenzender Wahrscheinlichkeit nicht bekommen wird. Aus diesem Grund wird im Rahmen dieser Arbeit auch nach Möglichkeit auf Anwendung der Odometrie-Funktionen, also Informationen über die Position und Ausrichtung der Simbots, verzichtet.

[7] Dies wird allerdings nicht garantiert, um wenigstens für ein Minimum an Realismus zu sorgen.

```
-- Input Classes And Related Functions
------------------------------------------------------------------
class HasRobotStatus i where
    rsBattStat :: i → BatteryStatus    -- Current battery status
    rsIsStuck  :: i → Bool             -- Currently stuck or not

data BatteryStatus = BSHigh | BSLow | BSCritical
     deriving (Eq, Show)

-- derived event sources:
rsBattStatChanged  :: HasRobotStatus i ⇒ SF i (Event BatteryStatus)
rsBattStatLow      :: HasRobotStatus i ⇒ SF i (Event ())
rsBattStatCritical :: HasRobotStatus i ⇒ SF i (Event ())
rsStuck            :: HasRobotStatus i ⇒ SF i (Event ())

class HasOdometry i where
    odometryPosition :: i → Position2  -- Current position
    odometryHeading  :: i → Heading    -- Current heading

class HasRangeFinder i where
    rfRange    :: i → Angle → Distance
    rfMaxRange :: i → Distance

-- derived range finders:
rfFront :: HasRangeFinder i ⇒ i → Distance
rfBack  :: HasRangeFinder i ⇒ i → Distance
rfLeft  :: HasRangeFinder i ⇒ i → Distance
rfRight :: HasRangeFinder i ⇒ i → Distance

class HasAnimateObjectTracker i where
    aotOtherRobots :: i → [(RobotType, Angle, Distance)]
    aotBalls       :: i → [(Angle, Distance)]

class HasTextualConsoleInput i where
    tciKey :: i → Maybe Char

tciNewKeyDown :: HasTextualConsoleInput i ⇒
                                 Maybe Char → SF i (Event Char)
tciKeyDown    :: HasTextualConsoleInput i ⇒ SF i (Event Char)

-- Output Classes And Related Functions
------------------------------------------------------------------
class MergeableRecord o ⇒ HasDiffDrive o where
  ddBrake   :: MR o                          -- Brake both wheels
  ddVelDiff :: Velocity → Velocity → MR o  -- set wheel velocities
  ddVelTR   :: Velocity → RotVel   → MR o  -- set vel. and rot.

class MergeableRecord o ⇒ HasTextConsoleOutput o where
  tcoPrintMessage :: Event String → MR o
```

Abbildung 12: Typklassen für die Ein- und Ausgabe der Roboter

3.2.1 Die "Welt" des Simulators

Innerhalb des Simulators arbeitet man mit festen Größen. Obwohl die Einheiten für diese Größen laut Hudak, Courtney, Nilsson und Peterson (siehe [HCNP03]) allesamt Meter sind, ist davon natürlich durch die Skalierung im Simulator nichts sichtbar. Die "Welt" (also die Karten für die Testszenarios) hat beispielsweise eine Größe von $10 * 10$ m, wobei die Koordinaten auf dieser Welt von [-5|-5] bis [5|5] anzugeben sind ([0|0] ist also die Koordinate für das Zentrum). An dieser Stelle seien noch die Konstanten `worldXMin`, `worldXMax`, `worldYMin` und `worldYMax` erwähnt, die wie erwartet die Werte -5 und 5 enthalten. Eine Welt ist vom Typ `WorldTemplate`, was wiederum eine Liste vom Typ `[ObjectTemplate]` ist.

Der Simulator kennt nur wenige (für seine Zwecke aber völlig ausreichende) Objekte (siehe Abbildung 13):

- **Objekte mit fixer Position,** namentlich Wände und Kubi (Blöcke). Hierbei kann man die Wände horizontal oder vertikal ausrichten, die Größe beträgt $0,1 * 1$ m. Die Blöcke haben eine Größe von $0,5 * 0,5$ m.
- **Bälle** sind neben den Simbots die einzigen beweglichen Objekte. Sie sind mit dem *AOT* der Simbots auffindbar und "rollen" weg, wenn sie von einem Simbot oder anderen Ball angestoßen werden.
- **Simbots** sind natürlich der wichtigste Bestandteil im Simulator. Obwohl, wie oben bereits erwähnt, der Simulator nur einen Typus von Roboter "kennt", simuliert er doch zwei verschiedene, indem einfach der String `rpType` entweder die Zeichenkette "`SimbotA`" oder "`SimbotB`" erhält. Die beiden Typen erhalten eine unterschiedliche Darstellung, `SimbotA` ist ein Kreis, `SimbotB` ein Dreieck. Beide Typen haben einen Durchmesser von $0,5$ m.

```
type WorldTemplate = [ObjectTemplate]

data ObjectTemplate =
    OTVWall   { otPos  :: Position2 } -- Vertical wall
  | OTHWall   { otPos  :: Position2 } -- Horizontal wall
  | OTBlock   { otPos  :: Position2 } -- Square obstacle
  | OTBall    { otPos  :: Position2 } -- Ball
  | OTSimbotA { otRId  :: RobotId,    -- Simbot A robot
                otPos  :: Position2,
                otHdng :: Heading   }
  | OTSimbotB { otRId  :: RobotId,    -- Simbot B robot
                otPos  :: Position2,
                otHdng :: Heading   }
```

Abbildung 13: Die "bekannte Welt" des Simulators

`Position2` ist hierbei ein Synonym für `RealFloat a ⇒ Point2 !a !a`, `RobotId` für `Int` und `Heading` für `Double`. Bei letzterem ist zu beachten, dass die Angabe im Bogenmaß ist, also im Bereich $(-\pi, \pi]$ liegt, wobei 0 für Osten, $\pi/2$ für Norden, π für Westen und $-\pi/2$ für Süden steht.[8]

[8] Yampa liefert hierfür dankenswerterweise eine Signalfunktion namens `normalizeAngle` mit.

Die Simbots benutzen alle den Datentyp `SimbotProperties`, welcher wiederum eine Instanz der Typklasse `HasRobotProperties` ist (siehe Abbildung 14). Somit kann man relativ problemlos auf einfache Eigenschaften wie Typus und Nummer des Simbots zugreifen.

```
class HasRobotProperties i where
  rpType     :: i → RobotType    -- Type of robot
  rpId       :: i → RobotId      -- Identity of robot
  rpDiameter :: i → Length       -- Distance between wheels
  rpAccMax   :: i → Accerelation -- Max translational acc
  rpWSMax    :: i → Speed        -- Max wheel speed

type RobotType = String
type RobotId = Int
```

Abbildung 14: Die Typklasse `HasRobotProperties`

3.2.2 Aufbau eines Testszenarios

Um mit dieser doch recht umfangreichen Menge von Funktionen und Klassen nun auch etwas anfangen zu können, wird natürlich ein Testszenario benötigt. Das entsprechende Programm hat die in Abbildung 15 aufgeführte Grundstruktur.

```
module MyRobotShow where

import AFrob              -- Yampa import
import AFrob.RobotSim     -- robot simulator

main :: IO ()
main = runSim (Just world) rcA rcB

world :: WorldTemplate
world =

rcA :: SimbotController  -- controller for simbot A's
rcA =

rcB :: SimbotController  -- controller for simbot B's
rcB =
```

Abbildung 15: Grundstruktur eines AFrob-Programms

Caveat: man kann zwar beliebig viele Simbots vom gleichen Typ erstellen, allerdings gibt es nur zwei `SimbotController`: den für alle Simbots vom Typ `SimbotA` und den für alle Simbots vom Typ `SimbotB`. Möchte man mehr als 2 verschiedene Roboter programmieren, empfiehlt sich der in Abbildung 16 gezeigte Workaround.

```
rcA :: SimbotController
rcA rProps =
  case rpId rProps of
    1 → rcA1 rProps
    2 → rcA2 rProps
    3 → rcA3 rProps

rcA1, rcA2, rcA3 :: SimbotController
rcA1 =
rcA2 =
rcA3 =
```

Abbildung 16: Workaround für mehr als 2 verschiedene `SimbotController`

4 Die `SimbotController`

4.1 `MergeableRecord`s

Erinnert man sich an die in Abbildung 12 gezeigten Output-Klassen, stellt man fest, dass dort der Typ `MR a` und die Einschränkung `MergeableRecord a` auftaucht. Damit ist es möglich, Datensätze einzeln zu spezifizieren und danach zusammenzufassen. Praktischerweise ist `SimbotOutput` eine Instanz von `MergeableRecord`. Somit lassen sich auch die Output-Werte zusammenfassen.

`MergeableRecord` besitzt drei grundlegende Operationen, von denen eine auch als Operator verfügbar ist. Die korrespondierenden Funktionen sind in Abbildung 17 aufgeführt.

```
mrMake     :: MergeableRecord a ⇒ (a → a) → MR a
mrFinalize :: MergeableRecord a ⇒ MR a → a
(~+~)      :: MergeableRecord a ⇒ MR a → MR a → MR a
mrMerge    :: MergeableRecord a ⇒ MR a → MR a → MR a
```

Abbildung 17: grundlegende Operationen des Moduls `MergeableRecord`

Beispiel:

```
sbo :: SimbotOutput
sbo = mrFinalize (ddVelDiff vel1 vel2
      'mrMerge' tcoPrintMessage stringEvent)
```

Diese Funktion fügt dem Befehl zur Festlegung der Geschwindigkeit einen zweiten Befehl zur Ausgabe einer Nachricht auf der Konsole an. ◁

Nota bene: für Simbots sind diese Befehle auch die einzigen, die sich zusammenfügen lassen.

4.2 Einfache `SimbotController`

Ein `SimbotController` hat folgenden Datentyp:

```
type SimbotController = SimbotProperties → SF SimbotInput SimbotOutput
```

`SimbotProperties` ist hierbei, wie in Kapitel 3 erwähnt, eine Instanz der in Abbildung 14 vorgestellten Typklasse `HasRobotProperties`. Beispielhaft werden in Abbildung 18 drei Controller vorgestellt:

```
rcStop :: SimbotController
rcStop _ = constant (mrFinalize ddBrake)

rcForward :: Velocity → SimbotController
rcForward v _ = constant (mrFinalize $ ddVelDiff v v)

rcForward' :: Velocity → SimbotController
rcForward' v rps = constant (mrFinalize $ ddVelDiff v' v')
                   where v'   = max (-vMax) (min vMax v)
                         vMax = rpWSMax rps
```

Abbildung 18: Einfache `SimbotController`

Hierbei ist `rcStop` ein stationärer Simbot, er tut also gar nichts. `rcForward` lässt den Simbot mit einer definierten Geschwindigkeit geradeaus (oder rückwärts, wenn $v < 0$) fahren. `rcForward'` ist eine Verbesserung, die die maximale Geschwindigkeit des Simbots in Betracht zieht.

4.2.1 Drehgeschwindigkeit mittels `ddVelTR`

Eine weitere Möglichkeit zur Angabe der Bewegung eines Simbots bietet die Funktion `ddVelTR`, die anstatt der beiden Geschwindigkeiten eine Angabe zur Vorwärts- und eine zur Drehgeschwindigkeit akzeptiert. Es gilt z.B.:

```
ddVelDiff v v = ddVelTR v 0
```

Für Differential Drive Robots gilt hier eine Einschränkung: je schneller die Vorwärtsgeschwindigkeit, desto langsamer kann der Roboter sich drehen. Insbesondere ist es unmöglich, eine Drehbewegung bei maximaler Vorwärtsgeschwindigkeit auszuführen. Es läßt sich zeigen, dass für maximale Geschwindigkeit v_{max} und eine Vorwärtsgeschwindigkeit v_f die maximale Drehgeschwindigkeit folgendermaßen berechnet werden kann:

$$\omega_{max} = \frac{2(v_{max} - v_f)}{l}$$

l ist hierbei der Abstand zwischen den rechten und linken Rädern.

Beispiel:

```
rcTurn :: Velocity → SimbotController
rcTurn vel rps =
    let rMax = 2 * ((rpWSMax rps) - vel) / rpDiameter rps
    in constant (mrFinalize $ ddVelTr vel rMax)
```

Dieser Simbot dreht sich bei einer festgelegten Geschwindigkeit so schnell er kann. ◁

4.3 Navigation in einem Labyrinth

Als etwas größeres Beispiel soll hier ein Simbot modelliert werden, der sich durch ein aus horizontalen und vertikalen Wänden bestehenden Labyrinth bewegt. Ein einfacher Algorithmus lautet: "folge der linken Wand, bis Du am Ziel bist." Hierzu werden die Range Finder benutzt.

4.3.1 Orientierung an einer Mauer

Die Idee ist hier, bei konstanter Vorwärtsgeschwindigkeit die Distanz zum nächsten linken Objekt gegen die gewünschte Distanz d zu prüfen und, wenn nötig, die Drehgeschwindigkeit anzupassen. Für die Bestimmung dieser Drehgeschwindigkeit kann man sich einer Formel aus der Regelungstheorie bedienen:

$$\omega = K_p(r - d) + K_d(\frac{dr}{dt})$$

r ist hierbei die gemessene und d die gewünschte Distanz zum nächsten linken Objekt und dr seine Ablei-tung. K_p ist die proportionale Zunahme, K_d die abgeleitete.[9] Weiterhin kann gezeigt werden, dass die optimale Beziehung zwischen K_p und K_d durch folgende Gleichung beschrieben werden kann:

$$K_p = vK_d^2/4$$

In [HCNP03] wird $K_p = 5$ gesetzt und aus pragmatischen Gründen $|\omega| \leq 0.2$ gefordert. Mit diesen Formeln ist der Controller sehr einfach zu schreiben (siehe Abbildung 19).

```
lim :: (Ord a) ⇒ a → a → a
lim m y = max (-m) (min m y)

rcFollowLeftWall :: Velocity → Distance → SimbotController
rcFollowLeftWall v d _ = proc sbi → do
  let r = rfLeft sbi
  dr ← derivative -< r
  let omega  = kp*(r-d) + kd*dr
      kd     = 5
      kp     = v*(kd^2)/4
  returnA -< mrFinalize (ddVelTR v (lim 0.2 omega))
```

Abbildung 19: Controller zum Ausrichten an der linken Wand

Caveat: Der Controller funktioniert zwar, aber nur, wenn der Simbot bereits einigermaßen ausgerichtet ist. In einem Labyrinth wird das der Fall sein, daher kann man sich in diesem Fall damit begnügen.

Was jetzt noch fehlt, ist die Möglichkeit, sich um Kurven zu bewegen. Genauer:

- Erkennt der Range Finder ein Objekt direkt vor sich, soll er sich um 90 Grad nach rechts drehen und dann der Wand folgen, die sich jetzt links von ihm befinden sollte (*Rechtskurve*).

[9] Ein derartiger Controller wird im Allgemeinen als "PD-Controller" bezeichnet.

- Erkennt der Range Finder links keine Wand mehr in der Nähe, soll er einen Bogen fahren, bis er sich um 90 Grad nach links gedreht hat, und dann der Wand folgen, die sich jetzt links von ihm befinden sollte (*Linkskurve*).

Die Event Sources hierfür sind relativ schnell geschrieben. *Nota bene:* Um sicherzustellen, dass der Simbot nach der Drehung tatsächlich nach den vier Himmelsrichtungen ausgerichtet ist, empfiehlt es sich, die 90-Grad-Drehung nicht hardzucoden, sondern dann ein Event zu generieren, wenn die Funktion `odometryHeading` einen Wert zurückgibt, der durch $\pi/2$ teilbar ist. Durch die Ungenauigkeit derartiger Funktionen ist hier allerdings nicht auf genaue, sondern nur auf ungefähre Gleichheit zu testen. Die Event Sources und zwei Hilfsfunktionen sind in Abbildung 20 aufgeführt.

```
turnRSF :: Distance → SF Distance (Event String)
turnRSF d = arr (<(d+0.3)) >>> (edgeTag "right")

turnLSF :: Distance → SF Distance (Event String)
turnLSF d = arr (>(d*3)) >>> (edgeTag "left")

alignedSF :: SF SimbotInput (Event ())
alignedSF = arr odometryHeading >>> arr aligned >>> edge

aligned :: (Ord a, Floating a) ⇒ a → Bool
aligned hdg = let hdg' = abs(2*hdg/pi)
              in any (λx → abs(x - hdg') < 0.01) [0.0,1.0,2.0]

lim :: (Ord a, Num a) ⇒ a → a → a
lim m y = max (-m) (min m y)
```

Abbildung 20: Event Sources und Hilfsfunktionen

Hierbei werden Events mit der Funktion `edgeTag` direkt getaggt, damit für die Testfälle eine Unterscheidung möglich wird. Der fertige Controller ist dann lediglich eine Kombination dieser Signalfunktionen (siehe Abbildung 21).

Dabei ist es ratsam, während der Drehung die Geschwindigkeit zu drosseln. Die Odometrie-Funktionen geben zwar akkurate Werte zurück, jedoch werden diese nicht kontinuierlich ausgelesen (auch wenn durch sehr kleine Zeitintervalle Kontinuität simuliert werden soll). Daher ist es möglich, bei einer zu schnellen Drehgeschwindigkeit den Moment zu verpassen, in dem der Simbot seine gewünschte Ausrichtung erreicht. Die Faustregel hierfür: je weniger Abweichung toleriert wird, umso langsamer sollte sich der Simbot drehen. Auch ist wegen der engen Gänge eine Verschärfung der Einschränkung des Wertebereichs von ω auf $|\omega| \leq 0.1$ sinnvoll.

5 Fazit

Am Beispiel von Yampa ist festzustellen, dass es gerade in der Robotik von sehr großem Vorteil sein kann, die traditionellen Low-Level-Sprachen hinter sich zurückzulassen. Anstelle der üblichen, schnell unverständlich werdenden imperativen Programme, bei denen man sich stark an der Hardware (i.e. den verschieden Robotern)

```
rcMaze :: Velocity → SimbotController
rcMaze v rc = rcAlign `switch` (λx → rcTrn x)
  where d       = 0.7
        lSF     = turnLSF d
        rSF     = turnRSF d
        rcAlign = proc sbi → do
          let hdg = (odometryHeading sbi)*2/pi
              dir | hdg < (-1.5) = pi
                  | hdg < (-0.5) = (-pi/2)
                  | hdg <   0.5  = 0.0
                  | hdg <   1.5  = (pi/2)
                  | otherwise    = pi
              r   = rfRange sbi (normalizeHeading (dir))
          dr ← derivative -< r
          let omega = kp*(r-d) + kd*dr
              kd    = 5
              kp    = v*(kd^2)/4
          turnLE ← lSF -< r
          turnRE ← rSF -< rfRange sbi (normalizeHeading (dir-(pi/2)))
          let turnE = turnLE `lMerge` turnRE
              mr    = (ddVelTR v (lim 0.1 omega)) `mrMerge`
                      tcoPrintMessage turnE
          returnA -< (mrFinalize mr, turnE)
        rcTrn x = ((constant (mrFinalize (ddVelDiff (v/4)
            (if x == "left" then (v/2) else (-v/4))))) &&& alignedSF)
              `switch` (λ_ → rcMaze v rc)
```

Abbildung 21: Fertiger Controller für das Labyrinth

orientieren muss, erreicht man durch FRP ein höheres Abstraktionsniveau, indem man sich an logischen Komponenten orientieren kann. Das resultiert einerseits in weniger unnötigem und repetitivem Programmieraufwand, andererseits verleiht es dem Programmierer eine in imperativen Sprachen nicht mögliche Ausdruckskraft.

Natürlich gibt es noch einige "Kinderkrankheiten". Insbesondere ist nicht immer offensichtlich, wie die unterschiedlichen Signalfunktionen verknüpft werden müssen, um das gewünschte Ergebnis zu erzielen, und das Programmieren komplexerer Signalfunktionen erfordert recht viel Denkarbeit. Allerdings wird genau an diesen Problemen gearbeitet, so dass Yampa bereits eine hervorragende Möglichkeit bietet, reaktive Systeme mit den Vorteilen funktionaler Sprachen zu programmieren.

Yampa wurde zwar bereits zur Programmierung von Industrierobotern verwendet, allerdings bietet es im Bereich hybrider Systeme noch viele weitere Möglichkeiten: Bereits 2003 wurde mit Yampa eine Variation des bekannten Spieleklassikers "Space Invaders" geschrieben (beschrieben in [CNP03]), 2005 sogar ein First Person Shooter namens "Frag" (siehe [Che05]). Gerade dort fällt allerdings auf, dass bei einer größeren Menge von Charakteren recht viele Ressourcen verbraucht werden. Da größere Firmen im Spielebereich häufig derartige Charaktere parallelisieren bzw. auf Threads verteilen, bietet sich eine Verkettung mit einer parallelen Sprache wie z.B. "Eden" (hierzu siehe [LOMP05]) an.

Aber selbst ohne Parallelität hat man den entscheidenden Vorteil, dass man mathematische und physikalische Gleichungen (wie z.B. aus der Regelungstheorie,

wenn es um Robotik geht) ohne Probleme in Yampa-Code übertragen kann, und sich damit voll und ganz auf seine Kreativität und die Programmierung dessen, was die Signalfunktion eigentlich machen soll, konzentrieren kann. Dies schlägt sich dann natürlich auch in der Produktivität nieder.

Literatur

[Che05] Mun Hon Cheong. Functional programming and 3d games. Master's thesis, University of New South Wales, Sydney, Australia, November 2005.

[CNP03] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18, New York, NY, USA, 2003. ACM.

[DJ00] Gregory Dudek and Michael Jenkin. *Computational Principles of Mobile Robots*. Cambridge University Press, New York, 2000.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[Hug00] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[LOMP05] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[Pat01] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[PHE99] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *First International Workshop on Practical Aspects of' Declarative Languages (PADL)*, January 1999.