Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Seminararbeit

Functional Reactive Programming with Liveness Guarantees

Thorben Droste

WS 2015/2016

Inhaltsverzeichnis

1 Einführung

Viele aktuelle Softwaresysteme sind reaktiv und müssen somit jederzeit auf Anfragen von außen reagieren können. Jeder Server, der Anfragen bearbeitet, oder jedes Programm mit einer grafischen Benutzeroberfläche ist somit reaktiv. Reaktive Systeme werden häufig durch Events gesteuert und können sehr komplex werden. Die Funktionale Reaktive Programmierung (FRP) kapselt den eventgesteuerten Teil des Systems und bietet eine rein funktionale Schnittstelle zur Entwicklung von reaktiven Systemen.

Diese Seminararbeit befasst sich mit dem Thema Funktionale Reaktive Programmierung und einer auf Mengen und Continuations basierenden Haskell-Implementierung. Außerdem wird diskutiert, wie durch ein geeignetes Typsystem sichergestellt werden kann, dass für jedes eingehende Event immer auch ein ausgehendes Event erzeugt wird (Liveness). Die Arbeit befasst sich inhaltlich mit dem Artikel „Functional Reactive Programming with Liveness Guarantees“ von Alan Jeffrey [Jef13]. Der von Jeffrey diskutierte FRP-Ansatz wird in einer Haskell-Implementierung konkretisiert. Dabei werden entstehende Typprobleme anhand von konkreten Beispielen diskutiert und Lösungsvorschläge gemacht.

1.1 Motivation

Komplexe Softwarearchitekturen bestehen aus vielen Komponenten, die über definierte Schnittstellen miteinander kommunizieren. Das Ziel ist es, diese Komponenten möglichst gut voneinander zu entkoppeln und die Kommunikation zu minimieren. In der imperativen Programmierung ist die Kommunikation über Ereignisse (Events) ein weit verbreitetes Konzept. Die teilweise auch in eigenen Threads laufenden Komponenten lösen dabei jedes Mal ein Event aus, wenn sich der innere Zustand der Komponente ändert. Andere Komponenten können Callback-Funktionen registrieren, um auf die Events zu reagieren. Durch die Verwendung vieler Callback-Funktionen wird der Quellcode häufig sehr unübersichtlich und die Wartung wird aufwendiger. Außerdem lässt sich nicht mehr unbedingt erkennen, welche Abhängigkeiten zwischen den Komponenten bestehen. Die Aufgabe der Funktionalen Reaktiven Programmierung ist es, die eventgesteuerte Kommunikation zwischen den Komponenten zu kapseln, einen Mechanismus zur Behandlung von konkurrierenden Zugriffen und eine rein funktionale Schnittstelle zur Verfügung zu stellen. Über die funktionale Schnittstelle können die Abhängigkeiten zwischen den Komponenten in deklarativer Form beschrieben werden. Das Konzept der FRP wurde in der Vergangenheit viel diskutiert und stellt zusammen mit einem Typsystem, welches den korrekten Einsatz sicherstellt, einen guten Ansatz dar, um komplexe Softwaresysteme übersichtlich und wartbar zu halten. Aus diesem Grund wird in dieser Arbeit eine konkrete Implementierung näher betrachtet.

1.2 Funktionale Reaktive Programmierung

Reaktive Systeme sind Systeme, die immer wieder auf externe Ereignisse reagieren müssen, ohne diese vorher explizit auszulösen [HP85, S. 479]. Ein Ereignis kann z. B. ein Mausklick, eine Tastatureingabe oder eine Anfrage an einen Server sein. Somit sind viele Systeme reaktiv und eigenen sich für den Einsatz von FRP. Die Funktionale Reaktive Programmierung ist eine Technik um reaktive Systeme mit einem rein funktionalen Interface zu entwickeln [Jef13, S. 1] und wurde zum ersten Mal 1997 in dem Artikel „Functional Reactive Animation“ von Conal Elliott und Paul Hudak [EH97] behandelt.

Ein FRP-Framework besteht grundsätzlich aus Behaviors und Events sowie einer Reihe von Kombinatoren. Ein Behavior oder auch Signal ist ein sich über die Zeit verändernder Wert, wie z. B. der Wert eines Sensors oder der Inhalt eines Textfeldes. Ein Event ist eine potentiell unendliche Folge von Werten gepaart mit den Zeitpunkten, zu denen das Event aufgetreten ist, wie z. B. ein Mausklick oder ein Tastendruck. In dieser Arbeit wird der Begriff Signal als Oberbegriff für Behaviors und Events verwendet. Signale können mithilfe der Kombinatoren zu neuen komplexeren Signalen kombiniert werden. Die Kombination der Signale wird auch als Signal-Graph oder Signal-Netzwerk bezeichnet [NCP02].

Das folgende Beispiel zeigt einen einfachen Taschenrechner und soll das Verhalten von Behaviors verdeutlichen. Die Benutzeroberfläche des Rechners besteht aus zwei Eingabefeldern für Zahlen und einem Ausgabefeld für die Summe der beiden Zahlen.

Mithilfe von FRP kann nun ausgedrückt werden, dass in dem Ausgabefeld immer die Summe der beiden Werte aus den Eingabefeldern steht. Sobald sich eines der beiden Eingabefelder ändert, passt sich auch die berechnete Summe im Ausgabefeld an. Die Eingabefelder lassen sich jeweils an ein Behavior binden, welches immer den aktuellen Wert des entsprechenden Feldes hat. Aus den beiden Behaviors wird durch die Anwendung der Funktion `+` ein neues Behavior `bSum` erzeugt, welches als Wert immer die Summe der Behaviors `num1` und `num2` hat. Dieses Behavior kann dann mit `bind` an das Ausgabefeld gebunden werden.

```
createNetwork inputField1 inputField2 outputField = do
    num1 <- behavior inputField1 0
    num2 <- behavior inputField2 0
    let bSum = (+) <$> num1 <*> num2
    bind outputField bSum
```

Das oben gezeigte Haskell-Beispiel[1] verwendet die Funktionen eines fiktiven FRP-Frameworks. Je nachdem, welches FRP-Framework eingesetzt wird, gibt es verschiedene Möglichkeiten, wie sich das Programm verhält.

- Push-basierte Systeme (data-driven) stoßen die Berechnung der Nachfolger im Signal-Netzwerk an, wenn sich ein Wert ändert. Ein Nachteil dieser Systeme ist, dass sich Behaviors gegebenenfalls häufig ändern und abhängige Werte neu berechnet werden, obwohl diese in diesem Moment nicht benötigt werden [Ell09, S. 1].
- Pull-basierte Systeme (demand-driven) berechnen einen Wert erst dann neu, wenn dieser benötigt wird. Das bedeutet, wenn ein Wert angefragt wird, dann werden wiederum die Vorgänger im Signal-Netzwerk neu berechnet. Der Nachteil von pull-basierten Systemen ist, dass jeder Vorgänger im Signal-Netzwerk neu berechnet wird, auch wenn sich dieser nicht geändert hat [Ell09, S. 1].

Mit einem pull-basierten Taschenrechner müsste die Berechnung z. B. durch ein Event angestoßen werden, während in einem push-basierten Taschenrechner die Berechnung der Summe beginnt, sobald sich einer der Eingangswerte ändert.

FRP ist ein in der Vergangenheit aber auch aktuell viel diskutiertes Thema und es gibt eine Vielzahl an Implementierungen. Die für diese Arbeit wichtigste Implementierung ist Reactive Banana [Apf15]. Reactive Banana orientiert sich an den Konzepten, die von C. Elliot in seiner Arbeit zum Thema Push-Pull-FRP vorgestellt wurden [Apf11]. Ein weiteres FRP-Framework ist Yampa [Gro]. Yampa basiert auf Arrows und der Originalarbeit zum Thema FRP von C. Elliott [EH97]. Schließlich gibt es auch in der Java-Welt eine FRP-Implementierung. JavaFX [Hom13] stellt Properties und Bindings zur Verfügung, welche den Behaviors und ihren Kombinatoren entsprechen.

[1] `https://www.haskell.org/`

2 Konzept für ein FRP-Framework

In diesem Kapitel wird ein Konzept für ein FRP-Framework nach Jeffrey[Jef13] vorgestellt. Der Fokus steht dabei auf einer Continuation-basierten Implementierung und den Liveness-Eigenschaften des Frameworks. Liveness-Eigenschaften sind Eigenschaften, welche garantieren, dass ein gewünschter Zustand irgendwann eintritt [OL82, S. 486]. In diesem Fall soll garantiert werden, dass für jedes eingehende Event irgendwann ein ausgehendes Event erzeugt wird [Jef13, S. 1]. Dazu wird in den folgenden Kapiteln ein passendes Typsystem entwickelt, das dem Compiler ermöglicht, die geforderten Eigenschaften sicherzustellen. In einem ersten Schritt wird ein Continuation-Modell in Haskell implementiert und darauf aufbauend eine Mengen-Implementierung aufgesetzt, welche als Abstraktion des Continuation-Modells dient. Auf der Basis dieser Abstraktion werden dann Events und Behaviors definiert.

2.1 Continuations

Eine Continuation ist eine Funktion, die bestimmt wie ein Ergebnis einer Operation weiterverarbeitet wird [App92, S. 2]. Eine Continuation kann allgemein als eine Funktion vom Typ $a \to b$ definiert werden, welche einen Wert vom Typ `a` verarbeitet und einen Wert vom Typ `b` zurück gibt. In diesem Fall dient das Continuation-Modell als Basis für die Implementierung von Mengen und jede Continuation liefert als Ergebnis einen Typ vom Wert `Bool`.

```
type Cont a = a -> Bool
```

Aufbauend auf diesen Typ werden verschiedene Funktionen definiert, um den Einsatz der Continuations zu vereinfachen. Die Funktion `discard` liefert eine Continuation, die den übergebenen Wert ignoriert und als Ergebnis immer `False` zurück liefert.

Angenommen ein Client liefert Anfragen mit Werten vom Typ `a`. Die Funktion `process` verarbeitet jede Anfrage mit der übergebenden Continuation und beantwortet die Anfrage entsprechend. Der Aufruf von `process client discard` ignoriert alle Anfragen und beantwortet sie mit `False`.

```
process :: Client a -> Cont a -> IO ()

discard :: Cont a
discard _ = False
```

Angenommen der Client produziert selbst Arbeitsanweisungen in Form von Continuations und alle Continuations, die der Client produziert, sollen mit dem Wert `x` vom

Typ `b` aufgerufen werden. Dann wäre der Typ $a = Cont\ b$ und als Continuation müsste der Funktion `process` eine Continuation vom Typ `Cont (Cont b)` übergeben werden. Mithilfe der Funktion `invokeWith` wird eine Continuation erzeugt, die alle übergebenen Continuations mit dem Wert von `x` aufruft.

```
process client (invokeWith b)

invokeWith :: b -> Cont (Cont b)
invokeWith x c = c x
```

Mit der Funktion `andThen` können zwei Continuations kombiniert werden, indem eine neue Continuation erzeugt wird, welche die Ergebnisse über eine Disjunktion verknüpft. Dabei bricht die Berechnung ab, sobald eine Continuation den Wert `True` liefert.

```
andThen :: Cont a -> Cont a -> Cont a
andThen c1 c2 x = c1 x || c2 x
```

Von Interesse wäre auch die Definition einer Funktion `map`, um Funktionen vom Typ $a \to b$ auf den Parameter einer Continuation vom Typ `Cont a` anzuwenden. Diese kann jedoch nicht definiert werden, da sich aus der übergebenen Continuation vom Typ `Cont a` kein Wert mit dem Typ `a` ableiten lässt. Stattdessen lässt sich eine Funktion `comap` definieren.

```
comap :: (b -> a) -> Cont a -> Cont b
comap f ca b = ca $ f b
```

Die Funktion `comap` bekommt als ersten Parameter eine Funktion vom Typ $b \to a$ statt einer Funktion $a \to b$. Mit `comap` kann z. B. aus einer Continuation `c = (<5)`, die für alle Werte vom Typ Int, die kleiner als 5 sind, den Wert `True` zurückgibt, eine Continuation `c' = comap length c` erzeugt werden, die für alle Listen mit weniger als 5 Elementen den Wert `True` zurückgibt.

2.2 Mengen und Relationen

In diesem Kapitel werden Mengen und Relationen auf der Basis des Continuations-Modells implementiert. Dabei ist eine Relation eine Menge von Paaren, und eine Menge wird als Continuation einer Continuation implementiert.

```
data Set a = Set { unSet : Cont (Cont a) }
type Relation a b = Set (a, b)
```

Eine Menge ist also eine Funktion von $Cont\ a \to Bool$, in der ein Wert genau dann enthalten ist, wenn die übergebene Continuation mit dem Wert aufgerufen wird. Dieses Verhalten lässt sich nutzen, um eine Funktion `contains` zu definieren. Übergibt man beispielsweise eine Continuation `(==42)` an eine Menge, dann wird diese Continuation mit allen in der Menge enthaltenden Werten aufgerufen. Existiert in der Menge ein Wert `42`, dann ergibt sich der Wert `True`.

```
contains :: Eq a => Set a -> a -> Bool
contains s v = unSet s (==v)
```

Die Funktion `empty` erzeugt eine leere Menge und verwendet dabei die auf Continuations definierte `discard` Funktion. Die leere Menge ignoriert jede übergebene Continuation und gibt immer den Wert `False` zurück. Die Funktion `singleton` erzeugt aus einem Wert eine Menge, welche genau diesen einen Wert enthält. Mit der Funktion `union` können zwei Mengen vereinigt werden. Dabei wird eine übergebende Continuation zuerst mit allen Werten aus der ersten Menge und dann mit allen Werten aus der zweiten Menge aufgerufen. Die Funktion `union` verwendet dafür den Contiuation-Kombinator `andThen`, welcher genau die beschriebene Semantik hat.

```
empty :: Set a
empty = Set discard

singleton :: a -> Set a
singleton = Set $ invokeWith a

union :: Set a -> Set a -> Set a
union (Set s1) (Set s2) = Set $ andThen s1 s2
```

Der Typ einer Menge entspricht dem Typ `Cont (Cont a)`, auf dem sich im Gegensatz zu `Cont a` eine Funktion `map` und somit eine Funktorinstanz[1] definieren lässt. Neben `fmap` wird eine weitere Funktion `pmap` definiert, welche für jeden Wert aus einer Menge eine Funktion `f` mit dem Typ $a \rightarrow Set\ b$ aufruft und die Ergebnisse in einer neuen Menge vereinigt. Dabei ist `copmap` eine Hilfsfunktion, die mit einer Funktion vom Typ $b \rightarrow Cont\ (Cont\ a)$ aus einer Continuation vom Typ `Cont a` eine Continuation vom Typ `Cont b` erzeugt. Mithilfe der Funktion `pmap` kann eine applikative Funktorinstanz[2] definiert werden. Die Funktion `pure` entspricht der bereits gezeigten Funktion `singleton` und erzeugt eine Menge mit genau einem Element. Der Applikationsoperator `<*>` auf Mengen verhält sich ähnlich wie der Applikationsoperator auf Listen. Jede Funktion aus der ersten Menge wird auf jedes Element aus der zweiten Menge angewandt, und die Ergebnisse werden vereinigt.

```
instance Functor Set where
    fmap f s = Set (\cb -> unSet s (\a -> cb $ f a))

instance Applicative Set where
    pure      = singleton
    sf <*> sv = pmap (\f -> f <$> sv) sf

pmap :: (a -> Set b) -> Set a -> Set b
pmap f (Set xs) = Set $ (comap (copmap (unSet . f))) xs
```

[1] vgl. Funktor `https://en.wikibooks.org/wiki/Haskell/The_Functor_class`
[2] vgl. Applikativer Funktor `https://en.wikibooks.org/wiki/Haskell/Applicative_functors`

```
copmap :: (b -> Cont (Cont a)) -> Cont a -> Cont b
copmap f ca b = f b ca
```

Eine weitere wichtige Funktion, die sich leicht mit `pmap` definieren lässt, ist die Funktion `filter`. Die Funktion filter nimmt ein Prädikat und eine Menge und entfernt alle Elemente aus der Menge, für die das Prädikat `False` liefert. Die Funktion `domain` bestimmt den Definitionsbereich einer Relation, indem die Funktion `fst`, die das erste Element eines Paares liefert, auf alle Paare der Relation angewandt wird.

```
filter :: (a -> Bool) -> Set a -> Set a
filter p s = pmap (\x -> if (p x) then singleton x else empty) s

domain :: Relation a b -> Set a
domain = fmap fst
```

Abschließend werden zwei Funktionen definiert, um eine Relation `R` vom Typ `Relation a b` auf eine Funktion `f` vom Typ $a \rightarrow Set\ b$ abzubilden und umgekehrt. Dabei bildet die Funktion `f` jedes `a` aus dem Definitionsbereich der Relation `R` auf eine Menge `B` ab, für die gilt: $B = \{b \mid (a,b) \in R\}$. Wenn `R` eine eindeutige und totale Menge ist, dann enthält `B` immer genau ein Element. Die Funktionen `relation` und `buffer` bilden zusammen einen Isomorphismus zwischen Relationen und mengenwertigen Abbildungen[3] (set-valued functions).

```
relation :: Set a -> (a -> Set b) -> Relation a b
relation s f = pmap (\a -> ((,)a) <$> f a) s

buffer :: Eq a => Relation a b -> a -> Set b
buffer (Set r) x = Set $ \cb -> r $ \(a,b) -> (a == x) && cb b
```

Die Anwendung der Funktionen wird im nächsten Beispiel noch einmal verdeutlicht. Die Relation $r = \{(1, Eins), (1, One), (2, Two)\}$ aus dem unten gezeigten Beispiel wird mit der Funktion `buffer` auf eine Funktion `f` mit dem Typ $Int \rightarrow Set\ String$ abgebildet. Durch die Anwendung der Funktion `f` auf den Wert `1` ergibt sich die Menge $\{One, Eins\}$. Diese Funktion `f` kann mit der Funktion `relation` wieder zurück auf die ursprüngliche Relation `r` abgebildet werden.

```
-- Ausgabe:
-- ["One","Eins"]
-- ["Two"]
example = do
    showSet (f 1)
    showSet (f 2) where
```

[3] vgl. Multivalued Function `http://mathworld.wolfram.com/MultivaluedFunction.html`

```
r1  = singleton (1, "Eins")
r1' = singleton (1, "One")
r2  = singleton (2, "Two")
r   = r1 `union` r1' `union` r2
dom = domain r
f   = buffer r
r'  = relation dom f
```

2.3 FRP-Modell

Im letzten Abschnitt wurde gezeigt, wie auf der Basis von Continuations eine Implementierung für Mengen und Relationen angegeben werden kann. Mithilfe dieser Abstraktion wird in diesem Abschnitt ein FRP-Modell definiert.

2.3.1 Events und Behaviors

Die Definition von Events und Behaviors wurde bereits im Kapitel 1.2 erläutert. Events treten zu einem bestimmten Zeitpunkt auf und liefern einen Wert, z. B. durch das Drücken der Taste 'a' zum Zeitpunkt t_0. Das Verhalten von Events lässt sich durch eine partielle Funktion $Time \to A$ mit `A` als Typ des referenzierten Wertes beschreiben. Behaviors verhalten sich anders und haben zu jedem Zeitpunkt einen definierten Wert, wie z. B. die Position des Mauszeigers. Ihr Verhalten entspricht einer totalen Funktion $Time \to A$.

Um zunächst zu verstehen, wie das FRP-Modell funktioniert, werden Events und Behaviors, wie im folgenden Codebeispiel gezeigt, definiert. Die tatsächlichen Implementierungen von Events und Behaviors unterscheiden sich und sind komplizierter als hier angegeben.

```
type Event a = Time -> a
type Behavior a = Time -> a
```

Eine FRP-Anwendung kombiniert Events und Behaviors mithilfe von geeigneten Kombinatoren, um das gewünschte Programmverhalten zu erreichen. Die einfachsten Kombinatoren sind die folgenden:

- `map :: (a -> b) -> Event a -> Event b` Die Funktion `map` nimmt eine Funktion `f` vom Typ $a \to b$ und ein Event `e` vom Typ `a` und wendet die Funktion `f` auf jeden Wert von `e` an.
- `filter :: (a -> Bool) -> Event a -> Event a` Die Funktion `filter` nimmt ein Prädikat `p` und ein Event `e` und erzeugt ein neues Event, welches nur zu den Zeitpunkten `t` auftritt, für die `p (e t)` gilt.
- `partition :: (a -> Bool) -> Event a -> (Event a, Event a)` Die Funktion `partition` nimmt ein Prädikat und ein Event und teilt die Vorkommen des Events je nachdem, welchen Wert das Prädikat liefert, auf.

- `union :: Event a -> Event a -> Event a` Die Funktion `union` vereinigt die Vorkommen zweier Events in einem neuen Event. Treten zwei Events gleichzeitig auf, dann wird das erste Event bevorzugt behandelt.

Mit diesen Kombinatoren kann die unten gezeigte `encryptedKeyEvent` Funktion implementiert werden. Die Funktion verschlüsselt, wenn ein Tastaturevent ausgelöst wird, alle Großbuchstaben mit einem einfachen Algorithmus und gibt das veränderte Event zurück. Aus dem Event $\{0 \to A, 1 \to B, 2 \to c\}$ wird das Event $\{0 \to C, 1 \to D, 2 \to c\}$. Dazu wird das Event zunächst mit dem `partition`-Kombinator in ein Event für Großbuchstaben und ein Event für alle anderen Zeichen aufgeteilt. Mit `map` wird die Verschlüsselungsfunktion auf das Großbuchstabenevent angewandt. Schließlich werden beide Events wieder miteinander kombiniert.

```
encryptedKeyEvent :: Event Char -> Event Char
encryptedKeyEvent keyEvt = union cAlphas numbers where
    (alphas, numbers) = partition isCapitel keyEvt
    cAlphas = map (encrypt 'C') alphas

    isCapitel c = ord c >= 65 && ord c <= 90
    encrypt c key = chrInAbc $ inAbc c + inAbc key
    inAbc c = ord c - 65
    chrInAbc i = chr (i `mod` 26 + 65)
```

Beispiel 2.1: Beispiel für die Verwendung von FRP-Kombinatoren

Der `union`-Kombinator hat ein paar Schwachpunkte. Werden zwei Events aus unterschiedlichen Quellen kombiniert, dann kann es passieren, dass zwei Events gleichzeitig auftreten. Die Semantik von `union` sieht vor, dass in diesem Fall das Event aus der ersten Quelle bevorzugt behandelt wird. Diese Entscheidung führt dazu, dass der `union`-Kombinator nicht kommutativ ist. Außerdem ist die Implementierung schwierig, da beim Eintreffen eines Events aus der zweiten Eventquelle zum Zeitpunkt t_i sichergestellt sein muss, dass die erste Eventquelle nicht auch ein Event zum Zeitpunkt t_i produziert. Erst wenn dies sichergestellt ist, kann das entsprechende Event weitergegeben werden und braucht nicht länger gepuffert werden.

Für die Implementierung von `union` wäre es hilfreich, wenn die Menge der Zeitpunkte zu denen das erste Event auftritt und die Menge der Zeitpunkte zu denen das zweite Event auftritt, disjunkt wären. In diesem Fall könnten die Events einfach vereinigt werden, der Kombinator wäre kommutativ und kein Event müsste gepuffert werden.

Ein weiteres Problem ist, dass der Typ von `encryptedKeyEvent` jede Form der Eventverarbeitung erlaubt. Das bedeutet, dass für jedes eingehende Event nicht unbedingt auch ein ausgehendes Event erzeugt wird. Dies steht im Widerspruch zu den geforderten Liveness-Eigenschaften.

Die genannten Probleme können durch ein geeignetes Typsystem gelöst werden. Zunächst wird der Definitionsbereich der Eventfunktion auf die Menge `T` mit $T \subseteq Time$ beschränkt. Die Menge `T` enthält genau die Zeitpunkte, zu denen das Event auftritt. Als

Ergebnis entsteht eine totale Funktion $T \to a$, die das Verhalten eines Events beschreibt. Aus Symmetriegründen werden auch Behaviors nur noch auf `T` definiert. Die Definition der Typen Event und Behavior wird um einen weiteren Typparameter erweitert:

```
type Event t a = t -> a
type Behavior t a = t -> a
```

2.3.2 Dependent Types

Der Typ `t` ist ein wenig problematisch. Ein Event `e` mit $e = \{0 \to A, 1 \to B, 2 \to c\}$ ist auf den Zeitpunkten $\{0, 1, 2\}$ definiert. Der Typ `t` muss genau diese Menge repräsentieren. Das bedeutet, dass der Typ `t` von seinen konkreten Werten abhängt. Eine solche Abhängigkeit lässt sich mithilfe von Dependent Types ausdrücken. Dependent Types sind Typen, welche auf den von ihnen enthaltenen Werten basieren [Tho91, S. 214]. Eine Sprache, welche Dependent Types unterstützt, ist z. B. Agda. In Agda lässt sich ausdrücken, dass die Konkatenation eines Vektors der Länge `m` und eines Vektors der Länge `n` einen neuen Vektor der Länge `m+n` erzeugt für alle $m, n \in \mathbb{N}$. Ob Dependent Types mächtig genug sind, um einen Typ wie `t` auszudrücken, konnte im Rahmen dieser Arbeit nicht geklärt werden. Das folgende Beispiel zeigt die Implementierung der Vektorkonkatenation aus der Vektorbibliothek von Agda [Dan11].

```
++ : ∀ {a m n}
   -> Vec A m
   -> Vec A n
   -> Vec A (m + n)
```

2.3.3 Existential Types

Durch die Verwendung des neuen Typs `Event t a` in dem in Beispiel 2.1 wird garantiert, dass die Funktion `encryptedKeyEvent` für jedes eingehende Event auch ein ausgehendes Event erzeugt. Aus dem gleichen Grund muss die Funktion `filter` angepasst werden. Wird auf das Event $e = \{0 \to A, 1 \to B, 2 \to c\}$ mit dem Typ `Event t a` mit $t = \{0, 1, 2\}$ die Funktion `filter` mit `isCapitel` als Prädikat angewandt, dann ergibt sich ein neues Event `e'` mit dem Typ `Event s a` mit $s = \{0, 1\}$ und $e' = \{0 \to A, 1 \to B\}$. Das neue Event ist nicht mehr vom Typ `t`, sondern vom Typ `s` mit $s \subseteq t$. Diese Abhängigkeit lässt sich mit Bounded Existential Types [CW85] beschreiben. Die Signatur von `filter` wird entsprechend angepasst und bedeutet, dass es einen Typ `s` gibt, welcher ein Subtyp von `t` ist.

```
filter :: (a -> Bool) -> Event t a -> (∃s ⊆ t)Event s a
```

Bei der Funktion `partition` verhält es sich ähnlich. Als Parameter erhält die Funktion ein Event vom Typ `Event t a` und liefert zwei Events mit den Typen `Event s a` und `Event r a` mit $r \subseteq t, s \subseteq t$ aber insbesondere auch $s = t \backslash r$. Um für den Typ `s` auszudrücken, dass `s` eine Menge repräsentiert, die alle Elemente enthält, die in `t` aber nicht in `r` enthalten sind, wird ein neuer Typ `Minus a b` definiert, welcher ausdrückt, dass in

einem Event vom Typ `Event (Minus a b) c` nur Werte vorkommen, die in `a` aber nicht in `b` liegen. Insgesamt verändert sich die Signatur von `partition` zu:

```
partition::(a -> Bool)
    -> Event t a
    -> ∃(s ⊆ t)(Event s a, Event (Minus t s) a)
```

Als Alternative zu neuen Typen, wie der Typ `Minus`, kann das korrekte Verhalten des Systems auch durch den Einsatz von Refinement Types [VRJ13] sichergestellt werden. Mithilfe von Refinement Types kann ein Typ an bestimmte Bedingungen geknüpft werden[4].

2.3.4 Rekursive Behaviors

Ein wichtiges Feature von FRP-Systemen sind rekursive Behaviors. Sei beispielsweise `vals` ein Behavior mit $vals = \{0 \to x_0, 1 \to x_1, 2 \to x_2\}$ und `sum` ein rekursives Behavior mit der Semantik $sum\ t_{i+1} = vals\ t_{i+1} + sum\ t_i$. Dann ist der Wert des Behaviors `sum` zum Zeitpunkt t_{i+1} von dem Wert des Behaviors `vals` zum Zeitpunkt t_{i+1} und von dem Wert des Behaviors `sum` zum Zeitpunkt t_i abhängig. Das Behavior $sum = \{0 \to x_0, 1 \to x_0 + x_1, 2 \to x_0 + x_1 + x_2\}$ liefert also zum Zeitpunkt t_i die Summe aller Werte von `vals` bis zum Zeitpunkt t_i.

Um sicherzustellen, dass jeder rekursive Aufruf irgendwann terminiert, werden mehrere Invarianten gefordert.

- Jedes rekursiv definierte Behavior `b` darf zum Zeitpunkt t_{i+1} nur von Werten von `b` bis zum Zeitpunkt t_i abhängig sein. Zusammen mit den beiden folgenden Invarianten terminiert damit jeder rekursive Aufruf.

- Die Menge der Zeitpunkte muss wohlgeordnet sein. Das heißt, sei `T` eine total geordnete Menge von Zeitwerten, dann ist `T` wohlgeordnet genau dann, wenn jede nicht leere Teilemenge `T` ein kleinstes Element besitzt. Das heißt, sei $S \subseteq T$ mit $S \neq \emptyset$ und $s \in S$, dann ist `s` kleinstes Element von `S` genau dann, wenn $\forall s' \in S : s \leq s'$ gilt. Zum Beispiel ist die Menge $\{\mathbb{Z}, \leq\}$ nicht wohlgeordnet, da für jedes $z \in \mathbb{Z}$ ein $z' \in \mathbb{Z}$ existiert mit $z' \leq z$. Die Menge $\{\mathbb{N}, \leq\}$ ist hingegen wohlgeordnet [Dei09, S. 225].

- Die Menge der Zeitpunkte muss lokal endlich sein. Das heißt, sei `T` eine Menge, dann ist $\{T, \leq\}$ eine lokal endliche Halbordnung genau dann, wenn jedes Intervall $[x, y] = t \in T | x \leq t \leq y$ eine endliche Menge ist. Daher gilt, $\{\mathbb{R}, \leq\}$ ist keine lokal endliche Halbordnung und $\{\mathbb{N}, \leq\}$ ist eine lokal endliche Halbordnung [Sta99, S. 98].

Rekursive Behaviors können mithilfe eines `fix`-Kombinators definiert werden. Der `fix`-Kombinator nimmt eine Funktion `f` vom Typ $Behavior\ t\ a \to Behavior\ t\ a$ und gibt den Fixpunkt von `f` zurück. Der Fixpunkt von `f` ist ein Behavior `b`, für das gilt:

[4] Beispiel für die Implementierung der Funktion filter `http://goto.ucsd.edu:8090/index.html`

$b = f(b)$ [Dei09, S. 295]. Mit den aktuell definierten Kombinatoren lassen sich keine Funktionen definieren, welche ein Behavior zum Zeitpunkt t_{i+1} auf ein Behavior zum Zeitpunkt t_i abbilden. Daher wird ein neuer Kombinator `delay` eingeführt, welcher ein Behavior um eine Zeiteinheit verzögert. Die Semantik von `delay` ist dabei $delay\ b\ t_{i+1} = b\ t_i$. Für ein Behavior $b = \{0 \to A, 1 \to B, 2 \to c\}$ liefert $delay\ b = \{1 \to A, 2 \to B\}$. Dabei verändert `delay` den Definitionsbereich des Behaviors von $\{0, 1, 2\}$ zu $\{1, 2\}$. Allgemeiner gilt, dass `delay` eine Funktion von einem `Behavior t a` zu einem `Behavior s a` mit $s = t \backslash \{t_0\}$ ist. Um diese Einschränkung auszudrücken, wird ein neuer Typ `Tail` definiert. Der Typ `Tail t` repräsentiert dabei die Menge $t \backslash \{t_0\}$. Der `delay`-Kombinator wird verwendet, um beim `fix`-Kombinator eine explizite Verzögerung um eine Zeiteinheit zu erzwingen und damit sicherzustellen, dass die erste Invariante immer erfüllt ist.

```
fix :: (Event (Tail t) a -> Behavior t a) -> Behavior t a
fix f = b where b = f (delay b)
```

Für die Implementierung von `delay` muss sich zu jedem Zeitpunkt t_{i+1} der Vorgänger t_i, sowie das kleinste Element t_0 bestimmen lassen. Um dies zu ermöglichen, wird ein abstrakter Datentyp `Clock t` implementiert, welche die folgenden Elemente enthält:

- `times :: Set Time` umfasst die Zeitpunkte auf denen das entsprechende Event oder Behavior definiert ist.
- `first :: Set Time` enthält das kleinste Element $t_0 \in times$ oder ist die leere Menge, falls $times = \emptyset$ gilt.
- `prev :: Relation Time Time` ist die Vorgängerrelation mit $(t_i, t_j) \in prev$ genau dann, wenn $t_i > t_j$ und $t_i, t_j \in times$.
- `parent :: t` liefert den einzig möglichen Wert vom Typ `t` und wird dazu verwendet, um z. B. nach einer `delay`-Operation aus einem Wert vom Typ `Clock (Tail t)` wieder einen Wert vom Typ `Clock t` zu bestimmen.

Die Typen `Event` und `Behavior` werden jeweils um ein Element vom Typ `Clock t` erweitert. Ein korrekter Wert des Typs `Clock t` kann nur dann erzeugt werden, wenn $t \subseteq Time$ und $\{t, \leq\}$ eine lokal endliche und wohlgeordnete Menge ist. Daher lassen sich Events und Behaviors nur noch mit Typen für `t` erzeugen, welche die geforderten Invarianten erfüllen. Außerdem kann für jedes Event oder Behavior der Definitionsbereich, das kleinste Element des Definitionsbereichs und für jeden Zeitpunkt ein Vorgänger bestimmt werden.

3 FRP-Implementierung

In den letzten Abschnitten wurden bis jetzt die FRP-Kombinatoren und ihre Semantik beschrieben, sowie ein Typsystem entwickelt, welches Liveness-Eigenschaften zusichert. In diesem Abschnitt wird die auf Mengen und Relationen basierende Implementierung des FRP-Modells näher betrachtet. Die auf Mengen definierten Kombinatoren haben häufig eine ähnliche Semantik, wie die entsprechenden Kombinatoren auf Events und Behaviors. Daher lassen sich die Mengen-Kombinatoren einfach in den Kontext von Events und Behaviors übertragen.

3.1 Events und Behaviors

Ein Behavior wird als Funktion mit dem Typ $Time \to Set\ a$ definiert und ein Event als Relation mit dem Typ `Relation Time a`.

```
data Behavior t a = Beh {
    behClock :: Clock t,
    at :: Time -> Set a
}
data Event t a = Evt {
    evtClock :: Clock t,
    occurs :: Relation Time a
}
```

Diese Implementierung liefert pull-basierte Behaviors, das heißt, dass der Wert eines Behaviors `b` erst durch den Aufruf der Funktion `at b` mit einem Wert `t` angefragt werden muss. Als Ergebnis wird eine Menge zurückgegeben. Der Menge kann eine Continuation übergeben werden, welche aufgerufen wird, wenn der Wert zum Zeitpunkt `t` zur Verfügung steht.

Im Gegensatz dazu ist die Eventimplementierung push-basiert. Der Eventrelation kann eine Continuation übergeben werden, die jedes Mal, wenn ein Event auftritt, mit einem Paar $(Time, a)$ aufgerufen wird.

Damit sich die Events und Behaviors wie oben beschrieben verhalten, müssen die übergebenen Continuations gespeichert werden. Die in Kapitel 2.2 beschriebene Mengenimplementierung, würde eine übergebene Continuation nur für alle bereits in der Menge enthaltenden Werte aufrufen, jedoch nicht für neu hinzukommende Werte. Um das korrekte Verhalten zu erreichen, müssten zunächst Seiteneffekte innerhalb einer Continuation erlaubt werden. Danach könnten mithilfe einer `IORef` alle Continuations einer Menge gespeichert werden und jedes Mal aufgerufen werden, wenn sich die Menge verändert.

Dieser Ansatz wird von Jeffrey in seiner Java-Implementierung verfolgt. An dieser Stelle wird auf eine konkrete Implementierung verzichtet, um die folgenden Beispiele nicht zu kompliziert zu machen.

Mit der aktuellen Implementierung der Events und Behaviors lassen sich Zustände erzeugen, welche nicht in der Semantik der Signale vorgesehen sind. Daher werden die folgenden Invarianten gefordert:

- Für jedes `t` aus dem Definitionsbereich eines Behaviors `b` liefert `at b t` eine Menge mit genau einem Wert. An dieser Stelle wird ein Wert vom Typ `Set` verwendet, weil sich eine Menge durch die CPS-Implementierung wie ein Future-Value verhält[1], d.h. der Wert ist noch nicht unbedingt verfügbar, kann jedoch schon über die Mengenkombinatoren manipuliert werden. In dem Moment, in dem der Wert verfügbar wird, wird eine Continuation aufgerufen und alle Operationen auf dem konkreten Wert durchgeführt.
- Für ein Event `e` mit dem Definitionsbereich `T` und dem Wertebereich `A` ist der Quellbereich der Relation `occurs e` gleich `T` und $\forall t \in T \forall x, y \in A : (t, x) \in occurs\ e \wedge (t, y) \in occurs\ e \Rightarrow x = y$. Die Relation `occurs e` ist also eindeutig und total auf `T`.

Die Funktion `at b` für ein Behavior `b` ist eine mengenwertige Abbildung und `occurs e` für ein Event `e` ist eine Relation. Damit lässt sich der in Kapitel 2.2 vorgestellte Isomorphismus direkt auf Behaviors und Events übertragen. Die Funktion `buffer` bildet ein Event auf ein Behavior ab und die Funktion `event` ein Behavior auf ein Event, dabei kann über das im Behavior enthaltenden `Clock`-Element der Definitionsbereich (`times`) des Behaviors bestimmt werden.

```
buffer :: Event t a -> Behavior t a
buffer (Evt c e) = Beh c $ buffer e

event :: Behavior t a -> Event t a
event (Beh c b) = Evt c $ relation (times c) b
```

3.2 Funktorinstanz

Jeffrey definiert in seiner Arbeit verschiedene Formen der Funktion `map`. Statt viele spezielle Varianten von `map` zu implementieren, lassen sich auch jeweils eine Funktorinstanz und gegebenenfalls auch eine applikative Funktorinstanz angeben. Dabei könnte die Arbeit von C. Elliott zum Thema Push-Pull FRP[Ell09] als Vorbild dienen, da hier ähnliche Implementierungsansätze zu erkennen sind. Basierend auf einer gültigen Funktorinstanz für Mengen, lassen sich auch Funktorinstanzen für Behaviors und Events angeben. Das `Clock`-Element wird dabei in beiden Fällen einfach durchgeschleift, da sich der Definitionsbereich der Signale nicht ändert. In der abstrakten Sichtweise auf ein Behavior `b` als

[1] vgl. `http://conal.net/blog/posts/another-angle-on-functional-future-values`

totale Funktion wäre `fmap f b` einfach die Funktionskomposition von `f` und `b`. In der konkreten Implementierung liefert `at` eine Menge. Daher muss die Funktion `f` zunächst mit `fmap` in den Kontext der Mengen gehoben werden. Die `fmap` Implementierung für ein Event hebt Funktion `f` in den Paar-Kontext und wendet diese dann auf die Elemente der `occurs`-Relation an.

```
instance Functor (Behavior t) where
    fmap f (Beh c at) = Beh c $ fmap f . at

instance Functor (Event t) where
    fmap f (Evt c occurs) = Evt c $ fmap f <$> occurs
```

3.3 Applikative Funktorinstanz

Mehrstellige `map` Funktionen wie `map2`, welche eine zweistellige Funktion auf zwei Behaviors anwendet, könnten elegant mithilfe einer applikativen Funktorinstanz definiert werden. Die Funktion `map2` wäre dann durch `map2 f b1 b2 = f <$> b1 <*> b2` definiert. Außerdem könnten auf die gleiche Weise `map3`, `map4`, usw. definiert werden. Dabei gibt es einige Punkte, die näher betrachtet werden müssen. Zum einen wird bei der Erzeugung eines Signals immer ein `Clock`-Element benötigt. Elliott löst dieses Problem, indem er den Definitionsbereich eines Signals um einen Wert $-\infty$ erweitert, sodass jedes Signal einen immer vorhandenen Startwert hat [Ell09, S. 3]. Ein anderer Ansatz wäre die Definition einer globalen Uhr, mit der dann jederzeit ein `Clock`-Element erzeugt werden kann. Jeffrey selbst macht in seiner Arbeit keine Angaben dazu, auf welche Art und Weise ein Signal erzeugt werden kann.

Unter der Annahme, dass ein globales Zeitelement existiert, wird die applikative Funktorinstanz definiert. Die Funktion `pure` erzeugt ein Behavior, welches für jeden Wert `t` eine Menge mit dem Element `x` zurückgibt. Der Applikationsoperator `<*>` verwendet die applikative Funktorinstanz des Typs `Set`, um die Funktion aus dem ersten Behavior zum Zeitpunkt `t` auf den Wert des zweiten Behaviors zum Zeitpunkt `t` anzuwenden. Diese Implementierung ist nur korrekt, da beide Behaviors den Definitionsbereich `t` haben und die Invariante, dass für ein Behavior `b` und ein Zeitpunkt `t` der Funktionswert von `b ‘at‘ t` immer eine Menge mit genau einem Element ist, erfüllt sein muss.

```
instance Applicative (Behavior t) where
    pure x = Beh c xs where
        t = getTime
        c = getClock t
        xs = \t -> singleton x
    Beh c bf <*> Beh _ bx = Beh c $ \t -> bf t <*> bx t
```

Ein mehrstelliges `map` kann für Events nicht ohne Weiteres implementiert werden, da die einzelnen Werte gegebenenfalls zu unterschiedlichen Zeiten zur Verfügung stehen. Seien z. B. e_1 und e_2 zwei Events mit den gleichen Zeitstempeln aber unterschiedlicher

Verarbeitungsdauer, das heißt, dass der Wert mit dem Zeitstempel t_i aus dem Event e_1 später als der Wert mit dem Zeitstempel t_i aus dem Event e_2 zur Verfügung steht. Werden diese Events mit einer Funktion `map2` kombiniert, dann muss das zweite Event gepuffert werden.[2] Jeffrey definiert eine Funktion `map2` auf einem Event und einem Behavior, sodass wenn eine Funktion mit `map2` auf zwei Events angewandt werden soll, ein Event durch die Umwandlung in ein Behavior explizit gepuffert werden muss. Somit könnte auch eine applikative Funktorinstanz für Events definiert werden, indem der Applikationsoperator immer implizit das zweite Event puffert. In der unten gezeigten Implementierung ist `bx` das gepufferte Event und `fApp` wendet die Funktion `f` zum Zeitpunkt `t` aus dem Event `ef` auf den entsprechenden Wert zum Zeitpunkt `t` aus dem Event `ex` an.

```
instance Applicative (Event t) where
    pure x = Evt c xs where
        t  = getTime
        c  = getClock t
        xs = singleton (t, x)
    Evt c ef <*> ex = Evt c $ pmap fApp ef where
        bx   = at (buffer ex)
        fApp = \(t,f) -> (,)t <$> f <$> bx t
```

3.4 Filter Implementierung

Alle Signalkombinatoren lassen sich einfach durch die definierten Mengenkombinatoren ausdrücken. Der meiste Aufwand entsteht durch die Verwaltung des `Clock`-Elements. Das folgende Codebeispiel zeigt die Implementierung der Funktion `filter`. Die eigentliche Filteroperation ist einfach die Anwendung der `filter` Funktion auf die zweite Komponente der Werte aus der Eventrelation. Der Definitionsbereich (`nTimes`) des neuen Events ist die Quellmenge der gefilterten Eventrelation. Um die neue Ordnungsrelation (`nPrev`) zu ermitteln, wird die Transitivität ausgenutzt und zu jedem Zeitpunkt `t` das nächste Element `t'` mit $t \leq t' \wedge t' \in nTimes$ gesucht. Das neue kleinste Element (`nFirst`) ist das kleinste Element des neuen Definitionsbereichs bzw. das einzige Element, für das kein Vorgänger in der neuen Ordnungsrelation definiert ist.

Dem `parent`-Element wird in diesem Fall der Wert `undefined` zugewiesen. Jeffrey definiert in seiner Arbeit nicht genau, welcher Wert dem `parent`-Element in diesem Fall zugewiesen werden soll. Der Idee nach müsste das `parent`-Element den einzigen Wert vom Typ `s`, also die Menge, die `s` repräsentiert, zugewiesen bekommen. In einem späteren Schritt würde sich dann das ursprüngliche `Clock`-Element über diesen Wert vom Typ `s` wiederherstellen lassen. Betrachtet man die Verwendung des `parent`-Elements genauer, dann stellt man fest, dass das Element tatsächlich nur in zwei speziellen Fällen genutzt wird. In beiden Fällen wird aus dem `parent`-Element das ursprüngliche `Clock`-Element wiederhergestellt.

[2] Das FRP Framework Reactive Banana definiert keine Applicative-Instanz für Events vgl. `http://hackage.haskell.org/package/reactive-banana`

- Der erste Fall ist die Funktion `union`. In diesem Fall erzwingt der Typ von `union`, dass die zu vereinigenden Events vorher mit der Funktion `partition` erzeugt wurden. Die Funktion `partition` erzeugt insbesondere ein Event vom Typ `Event (Minus s t) a`. Für den Typ `Minus` lässt sich ein konkreter Wert angeben und somit das `parent`-Element entsprechend setzen.
- Der zweite Fall ist die Funktion `cons :: a -> Event (Tail t) a -> Event t a`, welche eine Wert an ein Event anfügt. Die Typsignatur von `cons` erzwingt, dass der erste Wert des Events vorher durch die Funktionen `delay` oder `tail` entfernt wurde. Die Funktionen `delay` und `tail` erzeugen ein Event vom Typ `Event (Tail t) a`. Wie für den Typ `Minus` lässt sich für den Typ `Tail` ein konkreter Wert angeben und somit das `parent`-Element korrekt setzten.

Grundsätzlich ist in dem FRP-Modell von Jeffrey keine Funktion vorgesehen, welche aus einem gefilterten Event wieder das ursprüngliche Event herstellt. Daher wird auch das, in einer `filter` Funktion gesetzte, `parent`-Element nie ausgewertet. Eine mögliche Überlegung wäre es, ob nicht anstelle des Typs `t` der Typ `Maybe t` für das `parent`-Element verwendet werden sollte.

```
filter :: (a -> Bool) -> Event t a -> Event s a
filter p (Evt c xs) = Evt c' xs' where
    c'      = Clock nTimes nFirst nPrev nParent
    xs'     = filter (p . snd) e
    nTimes  = domain xs'
    nFirst  = filter (isFirst nPrev) nTimes
    nPrev   = getPrev nTimes (prev c)
    nParent = undefined

isFirst :: Eq a => Relation a a -> a -> Bool
isFirst ord t = isEmpty (bOrd t) where
    bOrd = buffer (swap <$> ord)

getPrev :: Eq a => Set a -> Relation a a -> Relation a a
getPrev def ord = pmap succPair defInit where
    bOrd       = buffer ord
    succPair t = (,)t <$> getNext t
    defInit    = filter (not . isLast) def
    isLast t   = isEmpty (bOrd t) || isEmpty (getNext t)
    getNext t  = if containsAll def (bOrd t)
        then bOrd t
        else pmap getNext (bOrd t)
```

Alle weiteren Kombinatoren lassen sich ähnlich implementieren und werden an dieser Stelle nicht weiter betrachtet.

4 Zusammenfassung und Fazit

Diese Arbeit gibt einen kurzen Einstieg in die Funktionale Reaktive Programmierung (FRP). Es wird gezeigt, was FRP ist und in welchen Fällen FRP angewandt werden kann. Es werden die Grundkonzepte von FRP diskutiert und gezeigt, welche Implementierungen es bereits gibt. Im Hauptteil der Arbeit wird eine auf Mengen und Continuations basierende FRP-Implementierung in Haskell vorgestellt, in der der Fokus auf der Liveness-Eigenschaft, dass für jedes eingehende Event immer auch ein ausgehendes Event erzeugt wird, liegt. Dies wird in erste Linie dadurch erreicht, dass Events und Behaviors als totale Funktionen über die Zeit modelliert werden. In der Arbeit wird thematisiert, welche Folgen diese Modellierung hat und wie die entstehenden Typprobleme gelöst werden können. Es wird der Unterschied zwischen pull und push-basierten Lösungen behandelt und gezeigt, wie die vorgestellte FRP-Implementierung beide Ansätze miteinander kombiniert. Außerdem wird ein Isomorphismus zwischen Relationen und mengenwertigen Funktionen definiert und auf die push-basierten Events und pull-basierten Behaviors übertragen.

4.1 Ergebnis

Im Zuge dieser Arbeit wird das vorgestellte FRP-Framework unter anderem in Haskell implementiert. Dazu wird zunächst ein Modul zur Verwendung und Manipulation von Continuations implementiert. Basierend auf den Continuations kann dann ein Typ für Mengen definiert werden. Die Mengenimplementierung abstrahiert das Continuations-Modul und bildet die Basis für die Implementierung von Events und Behaviors. Durch die geschickte Verwendung der Mengenabstraktion entstehen push-basierte Events und pull-basierte Behaviors. Die konkrete Implementierung der Signale wird durch applikative Funktorinstanzen erweitert, sodass FRP-Programme in einem applikativen Stil geschrieben werden können. Trotz intensiver Bemühungen zeigt sich, dass sich nicht alle Anforderungen des FRP-Frameworks mit dem Standard Typsystem von Haskell umsetzen lassen. Außerdem resultiert aus dem hohen Grad an Typsicherheit eine gewisse Inflexibilität des Frameworks. So können beispielsweise zwei Events nur vereinigt werden, wenn die Events vorher durch die Partitionierung einer gemeinsamen Eventquelle entstanden sind. Es können also keine Events aus verschiedenen Quellen kombiniert werden.

4.2 Empfehlungen für zukünftige Arbeiten

Das vorgestellte FRP-Framework ist nur ein Prototyp und bedarf einer Reihe von Erweiterungen, bevor es tatsächlich eingesetzt werden kann. Ein wichtiger Punkt ist dabei die

Implementierung des FRP-Frameworks in einer Sprache wie z. B. Agda, die alle Anforderungen an das Typsystem erfüllt. Außerdem muss eine Schnittstelle für die Erzeugung von Events und Behaviors definiert werden, damit das Framework z. B. zusammen mit einer GUI genutzt werden kann. Zwei weitere Punkte, die zu einem späteren Zeitpunkt genauer untersucht werden könnten, sind der Speicherverbrauch und die Laufzeitkomplexität des Frameworks. Das Problem mit dem Speicherverbrauch ist, dass ein Event zu jedem Zeitpunkt, zu dem das Event auftritt, einen Wert speichert und somit der Speicherverbrauch eines Events kontinuierlich wächst. Bei der Laufzeitkomplexität verhält es sich so, dass durch die Continuation basierte Mengenimplementierung viele Mengenoperationen eine lineare Laufzeit in der Anzahl der Elemente haben. Hier könnte überprüft werden, ob die Laufzeiten der einzelnen Funktionen verbessert werden könnten. Abschließend wäre es interessant, das FRP-Framework so zu erweitern, dass es in einem verteilten System transparent eingesetzt werden kann. Das heißt, dass Events und Behaviors innerhalb eines verteilten Systems genutzt werden können, als würden sie in einem zentralisierten System verwendet werden.

Literaturverzeichnis

[Apf11] Appelmus, Heinrich: *FRP - Push-driven Implementations and Sharing.* `http://apfelmus.nfshost.com/blog/2011/04/24-frp-push-driven-sharing.html`. Version: 2011, Abruf: 22.12.2015

[Apf15] Appelmus, Heinrich: *Reactive-banana.* `https://wiki.haskell.org/Reactive-banana`. Version: 2015, Abruf: 22.12.2015

[App92] Appel, Andrew W.: *Compiling with Continuations.* New York, NY, USA : Cambridge University Press, 1992. – ISBN 0–521–41695–7

[CW85] Cardelli, Luca ; Wegner, Peter: On Understanding Types, Data Abstraction, and Polymorphism. In: *ACM Comput. Surv.* 17 (1985), Dezember, Nr. 4, S. 471–523. `http://dx.doi.org/10.1145/6041.6042`. – DOI 10.1145/6041.6042. – ISSN 0360–0300

[Dan11] Danielsson, Nils A.: *Agda Vector Source.* `https://github.com/agda/agda-stdlib/blob/master/src/Data/Star/Vec.agda`. Version: 2011, Abruf: 22.12.2015

[Dei09] Deiser, O.: *Einführung in die Mengenlehre.* Springer Berlin Heidelberg, 2009 (Springer-Lehrbuch). `https://books.google.de/books?id=J8TBXux80G0C`. – ISBN 9783642014451

[EH97] Elliott, Conal ; Hudak, Paul: Functional Reactive Animation. In: *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming.* New York, NY, USA : ACM, 1997 (ICFP '97). – ISBN 0–89791–918–1, S. 263–273

[Ell09] Elliott, Conal M.: Push-pull Functional Reactive Programming. In: *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell.* New York, NY, USA : ACM, 2009 (Haskell '09). – ISBN 978–1–60558–508–6, S. 25–36

[Gro] Group, Yale H.: *Yampa.* `https://wiki.haskell.org/Yampa`, Abruf: 22.12.2015

[Hom13] Hommel, Scott: *JavaFX Properties and Binding.* `https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm`. Version: 2013, Abruf: 22.12.2015

[HP85] Harel, D. ; Pnueli, A.: On the Development of Reactive Systems. In: Apt, Krzysztof R. (Hrsg.): *Logics and Models of Concurrent Systems.* New York,

NY, USA : Springer-Verlag New York, Inc., 1985. – ISBN 0–387–15181–8, S. 477–498

[Jef13] Jeffrey, Alan: Functional Reactive Programming with Liveness Guarantees. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA : ACM, 2013 (ICFP '13). – ISBN 978–1–4503–2326–0, S. 233–244

[NCP02] Nilsson, Henrik ; Courtney, Antony ; Peterson, John: Functional Reactive Programming, Continued. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. New York, NY, USA : ACM, 2002 (Haskell '02). – ISBN 1–58113–605–6, S. 51–64

[OL82] Owicki, Susan ; Lamport, Leslie: Proving Liveness Properties of Concurrent Programs. In: *ACM Trans. Program. Lang. Syst.* 4 (1982), Juli, Nr. 3, S. 455–495. `http://dx.doi.org/10.1145/357172.357178`. – DOI 10.1145/357172.357178. – ISSN 0164–0925

[Sta99] Stanley, R.P.: *Enumerative Combinatorics:*. Cambridge University Press, 1999 (Cambridge Studies in Advanced Mathematics). `https://books.google.de/books?id=EvJg1VjIGyMC`. – ISBN 9780521663519

[Tho91] Thompson, Simon: *Type theory and functional programming*. Addison-Wesley, 1991 (International computer science series). – 214–225 S. – ISBN 978–0–201–41667–1

[VRJ13] Vazou, Niki ; Rondon, PatrickM. ; Jhala, Ranjit: Abstract Refinement Types. In: Felleisen, Matthias (Hrsg.) ; Gardner, Philippa (Hrsg.): *Programming Languages and Systems* Bd. 7792. Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–37035–9, S. 209–228