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

Seminararbeit

Praktische typisierte bedarfsorientierte Zusicherungen in Haskell

Sven Gundlach

WS 2012/2013

Inhaltsverzeichnis

1 Einführung

Vorbedingungen und Nachbedingungen sind wichtig Hilfsmittel zur Entwicklung von Programmen. Im *Design by contract* Konzept werden beide Bedingungen als Zusicherung zusammengefasst. Dabei werden Vorbedingungen als vom Aufrufenden einzuhaltende Zusicherungen und Nachbedingungen als vom Aufgerufenen einzuhaltende Zusicherungen gesehen. Dieses Konzept ist in der objektorientierten Programmierung wie z.B. in Eiffel [Mey92] verbreitet, wird aber in funktionalen Programmiersprachen nur wenig eingesetzt [Chi12].

Die Arbeit von Findler und Felleisen [FF02] hat praktische Möglichkeiten gezeigt, wie man Zusicherungen für Programmiersprachen mit Funktionen höherer Ordnung einsetzen kann. Findler und Felleisen haben diese auch in die Racket/Scheme Implementierung umgesetzt. Für andere Programmiersprachen wie zum Beispiel Haskell ist das aber bisher nicht der Fall gewesen. In der Arbeit von Chitil [Chi12] wurde daher eine Bibliothek für Haskell entwickelt, mit der Zusicherungen auf einfache Art in Haskell Programmen eingesetzt werden können. Diese Seminararbeit bafasst sich mit der Arbeit von Chitil [Chi12] und der dabei entwickelten Bibliothek.

1.1 Aufbau dieser Arbeit

In Kapitel 2 werden notwendige Grundlagen zum Verständnis der Arbeit erklärt bzw. vorgestellt.

Kapitel 3 befasst sich mit der von Chitil [Chi12] entwickelten Bibliothek und den dabei behandelten Problemstellungen. Die Strukturierung des Kapitels orientiert sich an den in der Arbeit von Chitil [Chi12] festgelegten Eigenschaften der Bibliothek.

Kapitel 4 fasst die Ergebnisse zusammen und weist auf mögliche Erweiterungen hin.

2 Grundlagen

In diesem Kapitel werden die Grundlagen zu den in der Arbeit von Chitil [Chi12] behandelten Themen dargestellt. Die Arbeit von Chitil baut auf Bedarfsauswertung auf. Es werden die grundsätzlichen Eigenschaften der Bedarfsauswertung und eine Abgrenzung zur strikten Auswertung angeben.

Für Zusicherungen ist in Haskell eine einheitliche Typisierung wünschenswert. Hierfür werden die grundlegenden verschiedenen Möglichkeiten zur Typisierung funktionaler Sprachen beschrieben.

Die letzten drei Abschnitte befassen sich mit Zusicherungen und *Contract Monitoring*, sowie den darauf aufbauenden Konzept *Design by Contract*. Es werden der Aufbau von Zusicherungen und die sich daraus ergebenden Notwendigkeiten und Möglichkeiten beschrieben.

2.1 Bedarfsauswertung

Die Auswertung von Ausdrücken kann grundsätzlich auf zwei Arten durchgeführt werden. Ausdrücke können ausgewertet werden, indem alle ihre Argumente ausgewertet und daraus folgend die Werte der Ausdrücke berechnet werden. Diese Strategie heißt strikte Auswertung und entspricht einer Leftmost-Innermost Reduktion. Vorteil dieser Strategie ist eine einfache Implementierung. Nachteilig ist, dass zur Berechnung des Wertes eines Ausdrucks alle Werte seiner Argumente bekannt sein müssen. Dies schließt auch zur Berechnung des Ausdrucks unnötige und nicht terminierende Argumente ein.

Dieser Nachteil kann durch eine faule Auswertung oder Bedarfsauswertung (*lazy evaluation*) vermieden werden. Bei dieser Strategie wird ein Ausdruck ohne Wissen über seine Teilausdrücke soweit wie möglich ausgewertet und danach nur Werte notwendiger Teilausdrücke berechnet. Vorteil ist eine bedarfsgesteuerte und damit möglicherweise einfachere Berechnung des Ausdrucks. Außerdem ist es möglich, unendliche Datenstrukturen zu verwenden. Der Nachteil ist eine aufwendigere Implementierung der Auswertung. Zudem ist die Analyse der benötigten Ressourcen, wie z.B. Speicherplatz oder Rechenzeit, komplizierter.

Will man beispielsweise ein Minimum aus einer Liste von Zahlen berechnen, kann man die Liste aufsteigend sortieren und danach das erste Element aus der Liste nehmen. In Haskell tut dies die folgende Funktion.

```
minimum :: [a] -> a
minimum ls = head (sort ls)
```

Mit einer strikten Auswertung muss zuerst die gesamte Liste sortiert werden, um danach das erste Element der Liste zurückzugeben. Wird als Sortieralgorithmus zum Beispiel Quicksort verwendet, kann das Minimum aller Zahlen schon feststehen, bevor die gesamte Liste sortiert ist. Dies ist der Fall, sobald die kleinste einelementige Teilliste feststeht. In diesem Fall wird bei bedarfsgesteuerter Auswertung der Rest der Liste nicht sortiert, sondern das kleinste Element zurückgegeben.

2.2 Typisierung

Funktionale Programmiersprachen können in statisch und dynamisch typisierte Sprachen unterteilt werden. Bei statisch getypten Programmiersprachen wird die Typprüfung während der Übersetzungszeit vom Compiler durchgeführt.

Die Prüfung zu diesem Zeitpunkt hat mehrere Vorteile. Erstens können Typfehler früh erkannt werden, ohne dass diese zur Laufzeit auftreten. Zweitens kann ein Compiler durch zusätzliche Informationen zur Übersetzungszeit Optimierungen vornehmen.

Nachteile ergeben sich zum einen bei der Wiederverwendung des Codes. Die Typen müssen gegebenenfalls angepasst werden. Durch das Einhalten eines Typsystems müssen entweder inhärente Typkonversionen genutzt oder Funktionen für mehrere Typen definiert werden.

Manche statisch getypte funktionale Programmiersprachen, wie zB. Haskell oder SML, bieten zusätzlich polymorphe Typen an, wodurch Funktionen unabhängig von konkreten Typen definiert werden können. Außerdem können durch Techniken wie Typinferenz Wiederholungen von Typangaben in Funktionsdefinitionen vermieden werden.

Bei dynamisch getypten Programmiersprachen werden die Typprüfungen zur Laufzeit durchgeführt. Dies hat den Vorteil, dass Funktionen einfacher definiert werden können. Die Funktionen sind dadurch in ihrer Anwendung flexibler. Dies ermöglicht die einfachere Wiederverwendung von Code, wenn dieser ausreichend dokumentiert ist.

Typisierung in Haskell Das *Hindley-Milner Typsystem* (HM) bildet die Grundlage vom Typsystem in Haskell und für viele weitere funktionale Programmiersprachen. Es ist ein statisches Typsystem. In HM könne Typen ohne zusätzliche Annotationen bestimmt werden. Der Typ eines Wertes oder einer Funktion wird per Typinferenz ermittelt.

Außerdem sind in Haskell verschiedene Arten von Polymorphismus möglich.

- Parametrischer Polymorphismus: Hierbei werden für Typen allgemeine Typen angegeben, die durch einen beliebigen konkreten Typ zur Laufzeit ersetzt werden können. solange nicht weitere Bedingungen wie z.B. `Eq` angegeben sind. Funktionen mit entsprechenden allgemeinen Typen können mit mehreren konkreten Typen verwendet werden.

Beispielsweise ist die Berechnung der Länge einer Liste unabhängig von den konkreten Typen der Listenelemente. In Haskell könnte eine Funktion daher folgendermaßen mit einem allgemeinen Typen `a` implementiert werden.

```
listLength :: [a] -> Int
listLength [] = 0
listLength (x:xs) = 1 + length xs
```

- Ad-hoc Polymorphismus: Beim Ad-hoc Polymorphismus wird ein Wert für mehrere Typen unterschiedlich implementiert. Zur Laufzeit wird über den konkreten Typ des Wertes die entsprechende Implementierung ausgewählt. In Haskell geschieht dies über Typklassen und deren Instanzen. Dazu wird ein Typ als Instanz einer Typklasse definiert und das Verhalten von vorgegebenen Funktionen für den Typ implementiert. Dadurch ist der Typ in allen die Typklasse voraussetzenden Funktionen verwendbar. [WB89]

  Beispielsweise kann eine Funktion zum Inkrementieren von Zahlen die Typklasse `Num` voraussetzen.

  ```
  simpleInc :: Num a => a -> a
  simpleInc x = x + 1
  ```

  Alle mit dieser Funktion verwendeten Typen müssen Instanzen der Typklasse `Num` sein und dazu unter anderem die Funktion `+` implementieren, wie z.B. der Typ `Int`. Eine selbst definierte Instanz `Zweiundvierzig` der Typklasse `Num` kann die Funktion `+` anders definieren, indem die Konstante `42` berechnet wird. Zusätzlich müssen alle anderen Funktion der Typklasse `Num` implementiert werden, solange keine Standard-Implementierungen der Typklasse existieren.

  ```
  instance Num Zweiundvierzig where
  (+) = 42
  (-) = 42
  (*) = 42
  ...etc...
  ```

  Für die Instanz `Zweiundvierzig` wird `simpleInc` das Ergebnis `42` ergeben.

Haskell bietet noch weitere Arten von Polymorphismus wie z.B. *rank-N types* und *impredicative types* an. Diese werden jedoch nicht speziell zur Typisierung der Zusicherungen in der Bibliothek [Chi12] verwendet.

2.3 Zusicherungen

Zusicherungen bestehen aus Vor- und Nachbedingungen. Vor- und Nachbedingungen müssen erfüllt sein, damit ein Programmablauf als korrekt angesehen wird [AH12, Chi12, DTW09]. In funktionalen Sprachen können Zusicherungen auf einfache Werte und auf Funktionen höherer Ordnung angewendet werden [FF02].

Zusicherungen ermöglichen eine stärkere Ausdrucksweise als Typsysteme, da sie unterschiedliche Bedingungen über gleiche Werte an unterschiedlichen Stellen im Programm zulassen [AH12, Chi12]. Zum Vergleich gelten bei Typsystemen dieselben Bedingungen für Werte eines Typs im gesamten Programm.

Zusicherungen ermöglichen auch Annahmen über die korrekte Verwendung von Schnittstellen, indem sie für aufrufende Funktionen Verpflichtungen festlegen, durch die eine korrekte Ausführung der aufgerufenen Funktion sichergestellt wird. Eine solche Zusicherung ist eine Funktion höherer Ordnung, die die Zusicherung des Ergebnisses von den Argumenten abhängig macht [Chi12].

Beispielsweise würde eine Zusicherung mit der Vorbedingung "positiver Integer" und Nachbedingung "positiver Integer größer gleich 10" in der Syntax der Bibliothek von Chitil [Chi12] wie folgt sein.

```
prop :: (a -> Bool) -> Contract a
prop p = \x -> if p x then Just x else Nothing

nat :: Contract Integer
nat = prop (>=0)

g x = assert (nat >-> (prop (>=10))) x + 10
```

Hierbei ist `g` eine zu sichernde Funktion, `nat` eine Bedingung für positive Integer, `prop` eine Funktion die Boolesche Prädikate in Bedingungen umwandelt, `>->` ein Kombinator für den Typ `Contract` und `Contract` der Typ für Bedingungen und Zusicherungen. `assert` stellt für die Funktion `g` sicher, dass wenn `x` ein positiver Integer ist, das Ergebnis eine Zahl größer gleich `10` ist.

2.4 *Design by Contract*

Design by contract ist ein Programmierkonzept, bei dem Softwareentwickler durch formale und präzise Zusicherungen die korrekte Verwendung von Funktion (als Teil eines Programms) sicherstellen sollen. Ein Client ruft eine Funktion auf und sichert dabei eine Vorbedingung zu. Ein Server stellt eine Funktion zur Verfügung und garantiert eine Nachbedingung. Zusicherungen fassen Vorbedingungen, Invarianten und Nachbedingungen zusammen, die vor, während und nach einem Programmablauf gelten. Eine Invariante gilt, bevor und nachdem eine Funktion angewendet wird. Invarianten können daher durch Vorbedingungen und Nachbedingungen konstruiert werden. Daher können Zusicherungen auf Vorbedingungen und Nachbedingungen reduziert werden.

Bei *Design by Contract* soll der Server nicht eine Verletzung der Vorbedingung behandeln, sondern den Client über die Verletzung informieren. Der Client ist dann für die Verletzung der Zusicherung verantwortlich. Genauso soll der Client nicht eine Verletzung der Nachbedingung behandeln, sondern der Server soll für die Nachbedingung und damit für die Verletzung der Zusicherung verantwortlich gemacht werden [DTW08, DTW09, Mey92].

Das *Design by Contract* Konzept vereinfacht so einen komponentenbasierten Aufbau von Programmen. Dazu werden die Zusicherungen als Teil der Schnittstellen der Komponenten implementiert. Das *Debuggen* eines Programms wird vereinfacht, indem die Quelle eines Fehlers und nicht sein Auftreten bestimmt wird [DTW08, DTW09, Mey92].

Das Verifizieren einer Zusicherung erfolgt durch *Contract Monitoring*. Hierbei ist es notwendig, dass die Semantik einer Funktion durch die Verifizierung von Zusicherungen nicht verändert wird [DTW08, DTW09, Mey92].

2.5 *Contract monitoring*

Zusicherungen können statisch oder dynamisch verifiziert werden. In der Praxis werden Zusicherungen meist durch *Contract Monitoring* dynamisch verifiziert.

Contract monitoring wird in der zu sichernden Funktion implementiert. Es wird die Vorbedingung vor der Ausführung einer Funktion geprüft. Falls die Vorbedingung verletzt ist, wird eine Fehlermeldung an den Client gegeben. Falls nach der Ausführung der Funktion die Nachbedingung verletzt ist, wird eine Fehlermeldung an die Funktion selbst gegeben[Chi12].

Nach den Arbeiten von Degen, Thierman und Wehr [DTW08, DTW09] sollten für *Contract Monitoring* die vier Eigenschaften *Meaning Reflection*(MR), *Violations Faithfully*(F), *Meaning Preservation* (MP) und *Idempotently* (IP) erfüllt sein.

- MR ist gegeben, wenn eine Funktion mit *Contract Monitoring* dasselbe Ergebnis liefert wie ohne *Contract Monitoring*.
- MP ist gegeben, wenn eine Funktion mit *Contract Monitoring* und ohne Verletzung der Zusicherung sich genauso verhält wie ohne *Contract Monitoring* oder bei Verletzung der Zusicherung die Verletzung angibt.
- F ist gegeben, wenn die durch *Contract Monitoring* verifizierten Zusicherungen nur wahre Eigenschaften enthalten.
- IP ist gegeben, wenn dieselbe Zusicherung mehrmals angewendet äquivalent zur einmaligen Anwendung der Zusicherung ist.

Für funktionale Sprachen kann *Contract Monitoring* in zwei Arten unterschieden werden.

Eager monitoring orientiert sich an der strikten Auswertung von Ausdrücken. Zur Verifizierung einer Zusicherung werden Vorbedingung und Nachbedingung strikt ausgewertet. Dabei kann entsprechend der strikten Auswertung Code unnötig ausgeführt werden [AH12, DTW08, DTW09, FF02].

Delayed monitoring orientiert sich an der Bedarfsauswertung von Ausdrücken. Die Prüfung von Vorbedingung und Nachbedingung werden solange verzögert bis alle notwendigen Argumente durch den Code der Anwendung berechnet sind [AH12, DTW08, DTW09, FF02].

3 Bibliothek

In diesem Kapitel wird ein Überblick über die in der Arbeit von Chitil [Chi12] entwickelte Bibliothek gegeben und die wichtigen Eigenschaften der Bibliothek beschrieben, sowie einige Teile der Implementierung vorgestellt.

Alle Abschnitte dieses Kapitels beziehen sich auf die Arbeit von Chitil [Chi12] und zitieren inhaltlich daraus. Referenzen auf die Arbeit und die Implementierung der Bibliothek werden daher nicht explizit angeben. Die Bibliothek kann auf der Webseite der *University of Kent* mit Beispielen frei heruntergeladene werden.

Die Bibliothek ist so aufgebaut, dass das Hinzufügen von Zusicherungen nicht den Typ oder die Semantik der Funktionen verändert. Dazu werden für Zusicherungen Kombinatoren über Funktionen implementiert, die parametrisch polymorphe Typen haben. Weiterhin werden Kombinatoren als Funktionen bereitgestellt, mit denen eine oder mehrere Vor- bzw. Nachbedingungen zu komplexeren Zusicherungen zusammengefasst werden können. Diese Funktionen werden nach Bedarf ausgewertet. Dadurch bleibt die Auswertungsstrategie der Haskell Programme erhalten.

Als zweiten wichtigen Punkt bietet die Bibliothek ein erweitertes Fehlerbehandlungssystem. Dieses stellt nicht nur das Auftreten eines Fehlers fest, sondern stellt weitere Informationen zu den Ursachen und Umständen bereit.

3.1 Parametrische polymorphe Typen und Aufbau der Zusicherungen

Damit Zusicherungen und darauf arbeitende Kombinatoren universell eingesetzt werden können, ist es notwendig, dass für Zusicherungen und Kombinatoren keine konkreten Typen verwendet werden. Typen von Zusicherungen und Kombinatoren müssen den Typen der Funktionen angepasst werden. Daher eignen sich parametrisch polymorphe Typen für die Typisierung. Es ist zusätzlich notwendig, für Typen von Zusicherungen und Kombinatoren keine klassenspezifischen Voraussetzungen zu machen.

Durch die polymorphe Typisierung können Zusicherungen in Funktionen implementiert werden, ohne dass die Typen der Funktionen anpasst werden müssen. Auch müssen in anderen Programmteilen keine Aufrufe von Funktion anpasst werden.

In der in Chitil entwickelten Bibliothek werden Zusicherungen zunächst als Identitätsfunktion implementiert. Es wird der Typ *Contract a* verwendet, wobei a den parametrischen Typ angibt. Dies ermöglicht es, eine Reihe von Kombinatoren korrekt zu implementieren.

```
type Contract a = a -> a
assert :: Contract a -> (a -> a)
assert c = c

true :: Contract a
true = id

false :: Contract a
false = const (error "...")

(&) :: Contract a -> Contract a -> Contract a
c1 & c2 = c2 . c1

(>->) :: Contract a -> Contract b -> Contract (a -> b)
pre >-> post = \f -> post . f . pre
```

Der Kombinator `&` verbindet zwei Zusicherungen zu einer Zusicherung, die verletzt wird, falls eine der enthaltenen Zusicherungen verletzt wird. `true` ist eine Zusicherung, die immer erfüllt ist. Dies ist nützlich, falls ausgedrückt werden soll, dass Werte in einer anderen Zusicherung irrelevant sind. Als Gegenstück gibt `false` an, dass die Auswertung von Werten nie erfolgen soll. Der Kombinator `>->` fasst Vor- und Nachbedingungen zu komplexeren Zusicherungen zusammen. Der Kombinator `>->` gibt keine logische Implikation an. Er gibt an, wenn eine Vorbedingung erfüllt ist, muss auch die Nachbedingung erfüllt sein. Falls die Vorbedingung nicht erfüllt ist, ist die Zusicherung verletzt und der Aufruf ist falsch.

Ein Problem dieser Implementierung ist es, dass nicht mehrere Zusicherungen disjunktiv getestet werden können. Ein disjunktiver Kombinator `|>` müsste mehrere Zusicherungen separat auswerten und die Ergebnisse kombinieren. Fehlermeldungen müssten dabei zurückgehalten und nur ausgegeben werden, wenn alle Zusicherungen verletzt werden. Es ist aber nicht möglich, auf Fehlermeldungen zu testen und diese dann zurückzuhalten. Daher lässt sich mit der bisherigen Implementierung ein disjunktiver Kombinator nicht implementieren.

Das Problem kann durch den in Haskell verfügbaren Typ `Maybe` gelöst werden. `Maybe` stellt die Konstruktoren `Just` und `Nothing` bereit. Auf diese Konstruktoren kann per *Pattern Matching* getestet werden. `Nothing` kann als Verletzung und `Just` für die Erfüllung einer Zusicherung verwendet werden. Daher kann durch eine Umwandlung des Ergebnisbereichs der Zusicherung in den Typ `Maybe` und einer Fallunterscheidung für `Nothing` und `Just` auf Verletzungen getestet werden.

Durch diese Implementierung ist es möglich, den disjunktiven Kombinator und die bisherigen Kombinatoren zu implementieren.

```
type Contract a = a -> Maybe a
assert :: Contract a -> (a -> a)
assert c x = case c x of
```

```
  Just y -> y
  Nothing -> error "Contract violated."

true :: Contract a
true = Just

false :: Contract a
false = const Nothing

(|>) :: Contract a -> Contract a -> Contract a
c1 |> c2 = \x -> c1 x `mplus` c2 x

(&) :: Contract a -> Contract a -> Contract a
c1 & c2 = \x -> c1 x >>= c2

(>->) :: Contract a -> Contract b -> Contract (a -> b)
pre >-> post = \f -> Just (f `seq` (assert post . f . assert
   pre))
```

Die Kombinatoren `>->`, `&`, `true` und `false` verhalten sich wie bereits in der ersten Implementierung

Der disjunktive Kombinator `|>` testet Bedingungen und und gibt eventuelle Verletzungen nur aus, wenn alle Bedingungen verletzt wurden.

Mit dieser Implementierung können Zusicherungen mit dem `>->` Kombinator aus Vor- und Nachbedingungen erstellt werden. Vorbedingungen und Nachbedingungen wiederum werden als Bedingungen für Werte und Zusicherungen als Bedingungen für Funktionen definiert. Komplexere Zusicherungen können durch Disjunktion und Konjunktion konstruiert werden. Der Typ `Contract` ermöglicht es, für Vorbedingungen und Nachbedingungen denselben Typ wie für Zusicherungen zu verwenden und darauf aufbauend polymorphe Kombinatoren wie `&` oder `|>` zu verwenden.

3.2 Bedarfsauswertung von Zusicherungen

Bedarfsauswertung ist eine wichtige Eigenschaft in der funktionalen Programmierung. Daher ist es notwendig, dass Kombinatoren dies unterstützen. Ein nach Bedarf auswertender Kombinator prüft nur die Teile einer Zusicherung, die auch notwendig sind. Es wird dadurch das Auftreten eines Fehlers und nicht die Möglichkeit des Auftretens erkannt.

In Haskell werden Funktionen nach Bedarf ausgewertet. Daher bieten sich Funktionen für die Implementierung von Zusicherungen an.

Im vorherigen Abschnitt wurde der Aufbau von Zusicherungen und Vor- bzw. Nachbedingungen als Funktionen beschrieben. Um entscheiden zu können, ob eine Vor- oder

Nachbedingung erfüllt ist, werden diese aus Booleschen Prädikaten konstruiert. Boolesche Prädikate werden als parametrisch polymorphe Funktionen mit Booleschem Ergebnis definiert.

```
prop :: (a -> Bool) -> Contract a
prop p = \x -> if p x then Just x else Nothing
```

Außerdem sind Kombinatoren als Funktion definiert, die Prädikate in Vor- und Nachbedingungen umwandeln. Alle Kombinatoren sind Funktionen und können daher nach Bedarf ausgewertet werden.

Ein Problem sind strikte Prädikate ,wie `all`, in Verbindung mit Listen. Strikte Prädikate können in Vorbedingungen oder Nachbedingungen auftreten und eine komplette Auswertung von Listen erfordern. Um dieses Problem zu lösen, können Typen in flache und nicht flache Typen unterteilt werden. Strikte Prädikate stellen für flache Typen kein Problem dar, da in diesen keine verketteten Datenstrukturen vorkommen. Jedes Element eines flachen Typs steht nur mit $\perp$ und sich selbst in Relation. Flache Typen sind in Haskell z.B. die Typen `Integer`, `Float`, `Bool` und `Char`. Für nicht flache Typen wie Listen können zusätzliche Konstruktoren bereitgestellt werden. Diese Konstruktoren können Listen punktweise auswerten und testen.

```
pNil :: Contract [a]
pNil [] = Just []
pNil ( : ) = Nothing

pCons :: Contract a -> Contract [a] -> Contract [a]
pCons c cs [] = Nothing
pCons c cs (x:xs) = Just (assert c x : assert cs xs)
```

Durch diese Konstruktoren kann eine Liste nach Bedarf ausgewertet werden. Die Konstruktoren werden durch *Pattern Matching* ausgewählt. *Pattern matching* testet nur die obersten Elemente eines Konstruktors. Der Programmierer muss daher bei Disjunktion mehrerer Konstruktoren die Reihenfolge beachten. Zum Beispiel wird folgende Zusicherung verletzt, obwohl nur eine der beiden inneren Zusicherungen verletzt wird.

```
assert (pCons nat pNil |> pCons true pNil) [-3]
```

Die erste Zusicherung `pCons nat pNil` passt auf das Argument `[-3]` und wird daher geprüft und durch `nat` verletzt. Daher wird die zweite Zusicherung nie geprüft, obwohl diese nicht verletzt wird.

Für andere nicht flache Typen müssen eigene Konstruktoren anhand der Konstruktion der Daten implementiert werden. In der Bibliothek wird ein Automatismus zur Ableitung solcher Konstruktoren anhand von Templates bereitgestellt.

Mit den vorgestellten Kombinatoren und Konstruktoren können Zusicherungen mit Bedarfsauswertung erstellt werden. Zum Beispiel kann folgendermaßen zugesichert werden, dass beliebig viele Elemente von einer unendlichen Liste genommen werden können.

```
nat :: Contract Integer
nat = prop (>=0)

infinite :: Contract [a]
infinite = pCons true infinite

takeFromInf :: a -> [b] -> [b]
takeFromInf = assert (nat >-> infinite >-> true) takeFromInf'

takeFromInf' n xs = take n xs
```

Hierbei wird durch das Prädikat `nat` angegeben, dass das erste Argument ein positiver Integer ist. Das Prädikat `infinite` stellt sicher, dass das zweite Argument eine unendliche Liste ist. Die Liste wird durch Bedarfsauswertung nur soweit wie notwendig ausgewertet.

3.3 Algebraische Eigenschaften der Kombinatoren

Mit Kombinatoren sollen aus einfachen Zusicherungen komplexere Zusicherungen aufgebaut werden können, hierfür eignen sich grundlegende algebraische Funktionen wie *Und* bzw. *Oder*. Mit diesen grundlegenden Kombinatoren ist es möglich, komplexe Bedingungen anhand des *Divide and Conquer* Konzeptes in einfache Bedingungen aufzuteilen. Der für Zusicherungen gewählte Typ `Maybe` gewährleistet dazu die notwendigen algebraischen Eigenschaften wie Disjunktion, Konjunktion, Assoziation und Idempotenz. Die Beweise dazu sind wie folgt.

- Disjunktion von Prädikaten.

  `prop` p_1 `|>` `prop` p_2 `=` `prop (\x ->` p_1 `x ||` p_2 `x)`

  Beweis. `prop` p_1 `|>` `prop` p_2

  ```
  Die Auswertungsstrategie ist leftmost.
  =\x -> case prop p1 x of
    Just y -> Just y
    Nothing -> prop p2 x
  Prädikate sind boolesche Funktion.
  =\x -> case (if p1 x then Just x else Nothing) of
    Just y -> Just y
    Nothing -> prop p2 x
  =\x -> if p1 x then Just x else prop p2 x
  =\x -> if p1 x then Just x else
    (if p2 x then Just x else Nothing)
  =\x -> if p1 x || p2 x then Just x else Nothing
  =prop (\x -> p1 x || p2)
  ```

  □

  Der Beweis zu Konjunktion von Prädikaten ist analog.

- Assoziativität der Konjunktion.
  c_1 & (c_2 & c_3) = (c_1 & c_2) & c_3

  Beweis. c1 & (c2 & c3)
  Definition der Konjunktion
  =\x -> c1 x >>= (c2 & c3)
  =\x -> c1 x >>= (\y -> c2 y >>= c3)
  Assoziativität der Maybe Monade
  =\x -> (c1 x >>= c2) >>= c3
  Substitution von x durch y
  =\x -> (\y -> c1 y >>= c2) x >>= c3
  Definition der Konjunktion auf c_1 und c_2 angewendet.
  =\x -> (c1 & c2) >>= c3
  Definition der Konjunktion auf (c_1,c_2) und c_3 angewendet.
  =(c1 & c2) & c3

  □

  Der Beweis zur Assoziativität der Disjunktion ist analog.
- Neutrales Element der Konjunktion.
  true & c = c

  Beweis. true & c
  Definition der Konjunktion
  =\x -> true x >>= c
  Definition von true
  =\x -> Just x >> c
  =(Eigenschaft der Maybe Monade)
  =\x -> c x
  =c

  □

  Die Beweise zum neutralem Element der Disjunktion, sowie zur Rechten Seite sind analog.
- Idempotenz der Konjunktion.
  c |> c = c

  Beweis. c & c
  =\x -> c x >>= c
  Die Auswertungsstrategie ist leftmost.
  =\x -> case c x of
  Just y -> c y
  Nothing -> Nothing
  Falls Just y dann ist c y auch Just y
  =\x -> c x
  Ist Konstante Funktion nach c
  =c

  □

  Der Beweis zur Idempotenz der Disjunktion ist analog.

Beweise zu weiteren Eigenschaften wie Absorption von Konjunktion und Disjunktion,

sowie dem Erhalt der Gültigkeit beim Konjunktion und Disjunktion sind in der Arbeit von Chitil aufgeführt.

3.4 Fehlerbehandlung

Zur Fehlerbehandlung wird meist der Fehler und die Position im Code, an dem der Fehler aufgetreten ist, angegeben. Bei Funktionen höherer Ordnung ist dies jedoch nur teilweise hilfreich. Funktionen höherer Ordnung können an zwei Positionen verletzt werden. Der Server, der eine Funktion bereitstellt, oder der Client, der die Funktion aufruft, können einen Fehler verursachen [DTW08, FF02, HJL06].

Das Erkennen des Zustandes, in dem ein Fehler auftritt, ist allein nicht immer ausreichend, um einen Fehler zu beheben. In solchen Fällen kann es erforderlich sein, den Ausführungspfad zu einem Fehler-Zustand zu ermittelt. Fehler sind durch diese Informationen besser nachvollziehbar.

Durch eine Zusicherung wird die Entscheidung, ob Server oder Client für einen Fehler verantwortlich sind, vereinfacht. Wird eine Zusicherung verletzt, sind entweder die Argumente der Funktion falsch oder die Funktion selber hat einen Fehler. Im ersten Fall ist der Client für einen Fehler verantwortlich. Die Vorbedingung ist verletzt. Im zweiten Fall ist der Server für einen Fehler verantwortlich. Die Nachbedingung ist verletzt.[DTW08, DTW09]

Ist das Argument selbst eine Funktion mit Vorbedingung, so ist es komplizierter, die Fehlerquelle zu bestimmen. Wird die Vorbedingung der Argumentfunktion verletzt, so ist nicht der Client verantwortlich sondern der Server. Der Server ist in diesem Fall Client der Argumentfunktion. Die *even-odd rule* ist eine einfache Methode, die Rolle der beteiligten Funktionen zu bestimmen [FF02].

Die *even-odd rule* zählt die Anzahl der Aufrufe einer Argumentfunktion in einem Ausführungspfad, in dem ein Fehler auftritt. Ist die Anzahl gerade, so ist der Server für den Fehler verantwortlich. Ist die Anzahl ungerade, so ist der Client für den Fehler verantwortlich.

Im folgenden Beispiel wird eine Funktion `f` in einer Funktion `g` aufgerufen.

```
g f = assert (((prop (>=1)) >-> (prop (>=0))) >-> true) f 0
```

Im Beispiel wird die Argumentfunktion `f` mit 0 von `g` aufgerufen, wodurch die Vorbedingung (größer gleich 1) von `f` verletzt ist. `f` steht in diesem Fall einmal in der Definition von g und in dem Aufruf von g, daher zweimal. Nach der *even-odd rule* ist daher g für den Fehler verantwortlich. Dies stimmt offensichtlich, da das Argument 0 in der Definition von `g` steht [FF02].

Angenommen in `g` wird die Funktion `f` mit 1 aufgerufen, aber die übergebene Funktion `f` liefert -1. In diesem Fall wurde der Aufruf von `g` bereits ausgewertet. `f` steht nur in der Definition von `g`. Nach der *even-odd rule* ist daher der Fehler im Aufruf von `g`. Auch dies stimmt offensichtlich, da der Fehler durch die übergebene Funktion entsteht [FF02].

In der Bibliothek wird diese Technik verwendet, indem zunächst der Server für einen Fehler verantwortlich gemacht wird. Der Zusicherung-Kombinator (`>->`) negiert die Verantwortung für den Fehler in der Vorbedingung und lässt sie in der Nachbedingung gleich. Durch Abzählen des Negierens der Verantwortung wird die *even-odd rule* verwendet.

3.5 Zusätzliche Funktionen

Die bisherigen Eigenschaften der Bibliothek reichen aus, um Zusicherungen zu implementieren und zur Laufzeit zu prüfen. Zur einfacheren Implementierbarkeit der Zusicherungen bietet die Bibliothek weitere Funktionen. Es werden Templates bereitgestellt, über die sich Konstruktoren für nicht flache Typen generieren lassen. Ebenfalls wird ein Negations-Kombinator für spezielle Typen ermöglicht. Abschließend bietet die Bibliothek Kombinatoren für strikte Typen an.

3.5.1 Templates

Um Konstruktoren für alle Datentypen anzubieten, muss für jeden Datenkonstruktor ein Konstruktor geschrieben werden. Da dies aufwendig ist, bietet die Bibliothek eine Möglichkeit, Code automatisch zu generieren. Dafür werden Haskell Templates , die vom *Glasgow Haskell compiler*(GHC) bereitgestellt werden, verwendet. Haskell Templates ermöglichen es, Code zur Compilezeit zu generieren und ihn auf Typkorrektheit zu prüfen. Zur Erstellung von Konstruktoren muss ein Ableitungsbaum vom Programmierer angegeben werden. Dieser Ableitungsbaum gibt die Datenstruktur an, woraus Datentypen für Haskell erzeugt werden können.

3.5.2 Negation

Neben Konjunktion und Disjunktion wäre zur einfacheren Ausdrucksweise ein Negations-Kombinator hilfreich. Ein allgemeiner Negations-Kombinator würde die Zusicherungen aber in eine boolesche Algebra umwandeln. Dies würde Probleme mit der Auswertung von `c |> ¬ c` und `c & ¬ c` verursachen [Chi11]. Daher werden analog zu den Konstruktoren für nicht flache Typen Kombinatoren für die Negation von Typen bereitgestellt.

3.5.3 strikte Erweiterungen

Haskell erlaubt die Verwendung von strikten Typen. Diese Typen können analog zu nicht strikten Typen in Zusicherungen verwendet werden. Eine Zusicherung prüft dabei den gesamten Typ, bevor ein Ergebnis zurückgeliefert wird. Daher verhalten sich bedarfsgesteuerte Zusicherungen auf strikten Typen wie strikte Zusicherungen (*eager contracts*).

4 Zusammenfassung und Ausblick

Die Bibliothek von Chitil [Chi12] bietet ein umfassendes System zur Laufzeitprüfung von Haskellprogrammen auf vielfältige Eigenschaften. Zu testende Programme müssen dazu nur erweitert, nicht umgeschrieben werden. Dabei wird die Semantik des Programms nur durch die in den Zusicherungen gewünschten Eigenschaften erweitert. Es gibt Erweiterungsmöglichkeiten, die den Einsatz der Zusicherungen vereinfachen, wie z.B. Negation oder Templates.

Der Aufbau von zusätzlichen Konstruktoren für nicht flache Typen ist durch den GHC einfach gestaltet und damit praktisch anwendbar. Die Bibliothek und Anwendungsbeispiele können auf der Webseite der *University of Kent* frei heruntergeladen werden.

Probleme gibt es beim *Pattern Matching* von Konstruktoren (vgl. Kapitel 3.2). Diese müssen durch den Programmierer beachtet werden. Der Einsatz von strikten Typen führt zur strikten Auswertung von Zusicherungen. Auch dies muss von Programmierer im Zusammenhang mit abstrakten Typen beachtet werden.

Literaturverzeichnis

[AH12] Antoy, S. ; Hanus, M.: Contracts and Specifications for Functional Logic Programming. In: *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, Springer LNCS 7149, 2012, S. 33–47

[Chi11] Chitil, Olaf: A semantics for lazy assertions. In: *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*. New York, NY, USA : ACM, 2011 (PEPM '11). – ISBN 978–1–4503–0485–6, 141–150

[Chi12] Chitil, Olaf: Practical typed lazy contracts. In: *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*. New York, NY, USA : ACM, 2012 (ICFP '12). – ISBN 978–1–4503–1054–3, 67–76

[DTW08] Degen, Markus ; Thiemann, Peter ; Wehr, Stefan: Contract Monitoring and Call-by-name Evaluation (extended abstract). In: *20th Nordic Workshop on Programming Theory*. Tallinn, Estonia, November 2008

[DTW09] Degen, Markus ; Thiemann, Peter ; Wehr, Stefan: True Lies: Lazy Contracts for Lazy Languages. In: *GI Jahrestagung*, 2009, S. 2946–2959

[FF02] Findler, Robert B. ; Felleisen, Matthias: Contracts for higher-order functions. In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. New York, NY, USA : ACM, 2002 (ICFP '02). – ISBN 1–58113–487–8, 48–59

[HJL06] Hinze, Ralf ; Jeuring, Johan ; Löh, Andres: Typed contracts for functional programming. In: *In FLOPS '06: Functional and Logic Programming: 8th International Symposium*, Springer-Verlag, 2006, S. 208–225

[Mey92] Meyer, Bertrand: *Eiffel: The Language*. Hemel Hempstead : Prentice Hall, 1992

[WB89] Wadler, P. ; Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: *Proc. POPL'89*, 1989, S. 60–76