Erweiterung eines Frameworks zur Generierung freier Theoreme um Typkonstruktorklassen

Thomas Rossow
Matrikelnummer: 004045

Master-Thesis
Februar 2017

Betreut von:
Prof. Dr. Michael Hanus
M. Sc. Sandra Dylus

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

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Kiel, den 3.2.2017 _____

Inhaltsverzeichnis

1 Einleitung

Das Typsystem einer Programmiersprache hat einen erheblichen Einfluss darauf, wie sich die Arbeit mit dieser Sprache gestaltet. Natürlich wird man jedes Programm, das mit einem statischen Typsystem entwickelt wurde, auch mit dynamischen Typen schreiben können. Dennoch bietet statische Typsicherheit deutliche Vorteile – nicht nur während der Entwicklung. Ein Vorteil ist die Möglichkeit, wertvolle Erkenntnisse allein aus den verwendeten Typen des Programms zu ziehen. Und genau das ist ein zentrales Thema dieser Arbeit.

Haskell ist eine funktionale Programmiersprache mit einem statischen Typsystem. Wie es bei vielen funktionalen Programmiersprachen der Fall ist, basiert auch Haskells Typsystem auf dem Hindley/Milner System [Hin69, Mil78], was es Haskell ermöglicht, zu jedem Ausdruck automatisch eine Typsignatur herzuleiten, die den allgemeinsten Typ dieses Ausdrucks darstellt [Wad89].

Und das ist der Grundsatz: *Jeder* Ausdruck hat einen ganz bestimmten Typ. Dieser Typ steht fest und wird sich auch während der Programmausführung nicht ändern. Kann ein solcher Typ nicht hergeleitet werden, dann kompiliert das Programm gar nicht erst.

Der offensichtliche Vorteil für den Entwickler ist, dass viele unerwünschte Programmzustände gar nicht erst auftreten, weil sie bereits durch Typfehler abgefangen werden, während eine Programmiersprache mit dynamischem Typsystem solche Probleme ohne Fehlermeldung durchwinken würde, was nicht selten schwer nachvollziehbare Laufzeitfehler zur Folge hat. Betrachten wir zum Beispiel die folgende Funktion der Skriptsprache *Javascript* [Fla06].

```
function sum(a, b)
{
   return a + b;
}
```

Javascript hat ein dynamisches Typsystem, das heißt es können beim Aufruf der Funktion `sum` Parameter beliebiger Typen übergeben werden. Ein Laufzeitfehler tritt dann auf, wenn zwei Werte `a` und `b` übergeben werden, auf die der Operator + nicht angewandt werden kann (ein Beispiel hierfür wären Funktionen). Problematisch ist auch der Fall, in dem zwei Parameter übergeben werden, auf die der +-Operator zwar angewandt werden kann, jedoch nicht das erwartete Ergebnis liefert. Ein Aufruf von `sum("42", 7)` würde beispielsweise zum Ergebnis `"427"` führen.

Trotz automatisch hergeleiteten Typen bietet Haskell dem Entwickler zusätzlich die Möglichkeit, zu jedem Ausdruck eine eigene Typsignatur anzugeben. Das führt nicht dazu, dass Haskell Ausdrücke auf einmal als völlig andere Typen interpretiert. Vielmehr bedeutet das, dass Haskell Typen, die der Benutzer manuell angegeben hat, ebenfalls akzeptiert, sofern es sich dabei um Spezialisierungen des allgemeinsten Typs handelt[1].

[1] Manchmal können Typannotationen einen Unterschied machen, beispielsweise in Verbindung mit Typklassen.

Das bietet vor allem den Vorteil, dass Haskell weiß, was der Entwickler erwartet. Es ist durchaus möglich, dass Haskell trotz eines Denkfehlers des Entwicklers einen Typ herleiten kann, der zur Funktionsdeklaration passt – mit dem man aber nicht rechnet. Das folgende Beispiel soll veranschaulichen, wie ein solcher Fehler zustande kommen kann.

```
concatStrings a b = a : b
```

Die angegebene Funktion soll zwei Zeichenketten konkatenieren, verwendet aber statt des Konkatenationsoperators ++ fälschlicherweise den Listenkonstruktor `:`. Das hat zur Folge, dass Haskell als ersten Parameter a ein Listenelement statt einer Liste erwartet, es ergeben sich jedoch keine Probleme mit der Typsignatur. Problematisch wird es erst, wenn versucht wird, die Funktion aufzurufen, beispielsweise mit dem folgenden Aufruf, wobei `b` eine Zeichenkette ist.

```
concatStrings "Student#" b
```

Haskell erkennt, dass der Funktionsaufruf nicht mit der (automatisch ermittelten) Funktionssignatur kompatibel ist und erzeugt einen Kompilierfehler. Die Fehlermeldung besagt allerdings, dass der Typ `[Char]` nicht mit dem erwarteten Typ `Char` übereinstimmt und gibt als Fehlerposition den Funktionsaufruf an.

Hätte der Entwickler die erwartete Typsignatur `[Char] -> [Char] -> [Char]` angegeben, dann hätte bereits die Typsignatur einen Fehler erzeugt, da sie kein Spezialfall der automatisch hergeleiteten Signatur `a -> [a] -> [a]` ist. Zusammenfassend kann man sagen, dass das Typsystem ein wichtiger Faktor bezüglich Fehleranfälligkeit und -prävention ist.

Besonders vorteilhaft ist Haskells Typsystem aber, um Schlüsse aus Funktionssignaturen zu ziehen. Betrachtet man eine solche Signatur, offenbart sich oft schon ein Teil der Funktionalität, da die Einschränkungen des Typsystems wenig Spielraum für unerwartetes Verhalten lassen. Die folgende Zeile zeigt ein Beispiel für eine solche Funktionstypsignatur.

```
applyOnList :: (a -> b) -> [a] -> [b]
```

Es handelt sich um eine Funktion namens `applyOnList`, die als ersten Parameter eine Funktion erwartet, die einen beliebigen Typ auf einen anderen (oder den gleichen) beliebigen Typ abbildet. Beim zweiten Parameter muss es sich um eine Liste handeln, deren Listenelemente vom Typ a sind, also vom Typ, den die als erster Parameter übergebene Funktion als Eingabewert erwartet.

Allein durch diesen Typ lässt sich bereits erahnen, was sich ungefähr hinter dieser Funktion verbirgt. Es ist anzunehmen, dass die Funktion `applyOnList` die Eingabeliste mithilfe der übergebenen Funktion manipuliert und die resultierende Liste zurückgibt. Natürlich ist es nicht gesagt, dass sie das tut, es drängt sich aber die Annahme auf, dass sie etwas "Ähnliches" tun muss – einfach deshalb, weil es der Typ suggeriert.

Haskell stellt in der Prelude[2] eine Funktion `map` bereit, die die gleiche Signatur wie unsere Beispielfunktion `test` hat. Und `map` macht tatsächlich genau das, was man erwarten würde: Es wendet die übergebene Funktion auf jedes Element der übergebenen Liste an und liefert die Ergebnisliste. Dass sich `test` und `map` eine Typsignatur teilen, lässt Rückschlüsse auf eine ähnliche Funktionsweise zu. Dass diese Ähnlichkeit nicht auf vage Mutmaßungen beschränkt ist, sondern dass sogar ganz systematisch konkrete Aussagen zu beliebigen Typsignaturen hergeleitet werden können, ist Thema der sogenannten *freien Theoreme*, auf die in Kapitel 3 genauer eingegangen wird.

Da die Herleitung dieser freien Theoreme ganz systematisch und immer gleich abläuft, kann sie auch ohne Probleme programmatisch erledigt werden. Das Haskell-Paket *free-theorems* [Böh07] enthält eine Bibliothek, die genau diese Aufgabe erledigt, und für die sogar eine Weboberfläche existiert [fre]. Diese Bibliothek ermöglicht es, Haskell-Code einzulesen und aus beliebigen Typsignaturen die entsprechenden freien Theoreme zu generieren, wobei sogar eigene Datentypen in den Signaturen verwendet werden können. Die Weboberfläche enthält darüber hinaus eine Liste bekannter Deklarationen aus der Prelude.

Neben algebraischen Datentypen ist es in Haskell auch möglich, eigene Typklassen zu definieren und in Typsignaturen zu verwenden. Ein prominentes Beispiel für eine Typklasse ist `Eq`. Dabei handelt es sich um die Klasse der vergleichbaren Datentypen, die den Gleichheitsoperator == definiert.

Allein in der Prelude sind Typklassen allgegenwärtig, da sie mit Ad-Hoc-Polymorphismus ein mächtiges Werkzeug bieten, um Programme zu strukturieren. Die Bibliothek *free-theorems* unterstützt die Verwendung von Typklassen. Es ist möglich, eigene Klassen zu deklarieren und freie Theoreme aus Typsignaturen zu generieren, in denen Typklassen verwendet werden.

Allerdings unterstützt die Bibliothek lediglich Typklassen für Datentypen, die keine Typparameter haben. Haskell erlaubt es, Datentypen zu definieren, die Typen als Parameter erwarten. Haskell erlaubt es ebenso, Typklassen für parametrisierte Datentypen zu deklarieren. Dass es sich dabei nicht um eine unwichtige Kleinigkeit im Sprachumfang von Haskell handelt, wird spätestens klar, wenn man sich in Erinnerung ruft, dass auch die bekannte Typklasse `Monad` eine Typkonstruktorklasse ist, die in Haskell sehr häufig Anwendung findet.

Es ist also wünschenswert, *free-theorems* derart zu erweitern, dass es auch mit Typkonstruktorklassen zurecht kommt und freie Theoreme für entsprechende Typsignaturen generieren kann. Und genau darauf liegt der Fokus dieser Arbeit. Es wird erläutert, wie die Bibliothek und insbesondere die Anpassungen für Typkonstruktorklassen aussehen.

Dazu wird in Kapitel 2 zunächst auf die mathematischen Grundlagen eingegangen, also Notationen, Definitionen, etc., die für das Verständnis der daraufhin folgenden Abschnitte benötigt werden. Die allgemeine Herangehensweise zum Herleiten freier Theoreme ist dann das Thema von Kapitel 3, in dem das sogenannte *Parametrizitätstheorem* eingeführt wird, das dann zu freien Theoremen führt. Am Ende des Kapitels geht es dann um die Erweiterung der Theorie auf Typkonstruktorklassen. Um die Anwendbarkeit die-

[2] Die Prelude ist Haskells Standardbibliothek, die automatisch importiert wird [Jon03].

ser Theorie zu motivieren, werden daraufhin einige Anwendungsbeispiele gegeben.

Schließlich wird in Kapitel 4 die Bibliothek *free-theorems* etwas genauer beschrieben, wobei insbesondere auf den Aufbau und die grundsätzliche Aufteilung eingegangen wird. Die nötigen Anpassungen für die Erweiterung um Typkonstruktorklassen sind dann in Kapitel 5 zu finden.

In dieser Arbeit wird dabei größtenteils von einer vereinfachten Teilsprache von Haskell ausgegangen, die weder Striktheit noch den Fixpunktoperator kennt. Das reicht nicht aus, um alle realen Haskell-Programme zu beschreiben, und in Kapitel 6 wird genauer darauf eingegangen, welche Konzepte in dieser Teilsprache fehlen und wie sie von *free-theorems* behandelt werden.

Am Ende der Arbeit findet sich schließlich ein zusammenfassendes Kapitel und ein Ausblick, der Anstöße für zukünftige Arbeiten liefert.

2 Grundlagen

Bevor auf die eigentliche Thematik eingegangen wird, bietet es sich an, einige Grundlagen zu klären, die im Laufe dieser Arbeit von Bedeutung sein werden. Ein großer Teil der Arbeit baut auf dem mathematischen Konstrukt der Relation auf, weshalb neben grundsätzlichen mathematischen Notationen vor allem Definitionen und Notationen eingeführt werden, die Relationen betreffen.

Des Weiteren sind Haskell-Kenntnisse erforderlich für das Verständnis der weiteren Arbeit. Es würde an dieser Stelle zu weit führen, sämtliche Sprachkonstrukte von Haskell anzuführen, die für die folgenden Abschnitte von Bedeutung sind; dennoch wird zumindest auf Typklassen eingegangen, insbesondere Typkonstruktorklassen, da diese für die Arbeit eine signifikante Rolle spielen. Zudem bietet es sich an, von Anfang an einige Benennungen einzuführen, die im Laufe der Arbeit immer wieder auftauchen werden.

2.1 Relationen

Die mathematische Grundlage und das wichtigste Werkzeug beim Erzeugen von freien Theoremen sind Relationen.

Definition 1. *Eine binäre Relation ist auf zwei Mengen definiert, deren Elemente sie einander zuordnet. Die Schreibweise* $\mathcal{R} : T_1 \Leftrightarrow T_2$ *sagt, dass* $\mathcal{R} \subseteq T_1 \times T_2$ *eine auf den Mengen* T_1 *und* T_2 *definierte Relation ist.*

Man kann die Relation als eine Menge von Paaren auffassen, wobei jedes Paar aus einem Element aus T_1 und einem Element aus T_2 besteht. Die Aussage $(x, y) \in \mathcal{R}$ bedeutet also, dass x und y bezüglich $\mathcal{R}$ verwandt sind. Man schreibt hierfür gelegentlich auch $x\ \mathcal{R}\ y$ (hauptsächlich, wenn es sich bei $\mathcal{R}$ um ein Operatorsymbol handelt). Anders ausgedrückt ordnet eine Relation jedem Element aus der ersten Menge beliebig viele Elemente aus der zweiten Menge zu.

Eine besondere Relation, die sich aus einer beliebigen Menge erzeugen lässt, ist die sogenannte *Identitätsrelation*, die auch in freien Theoremen eine wichtige Rolle spielt.

Definition 2. *Ist* M *eine Menge, so bezeichnet* id_M *die Identitätsrelation auf dieser Menge, die definiert ist durch* $id_M = \{(x, x) | x \in M\}$*, in der also jedes Element aus* M *mit sich selbst verwandt ist.*

Ein Beispiel für Identitätsrelationen lässt sich ganz einfach für die Menge $\mathcal{B}$ der Wahrheitswerte geben. Es ist $\mathcal{B} = \{tt, ff\}$, somit ist $id_{\mathcal{B}} = \{(tt, tt), (ff, ff)\}$. Man kann die Menge der Relationen weiter unterteilen, ein Spezialfall der Relation ist beispielsweise die *partielle Ordnung*.

Definition 3. *Eine partielle Ordnung* $\sqsubseteq : M \Leftrightarrow M$ *ist eine Relation, die reflexiv, anti-*

symmetrisch und transitiv ist, d.h. es gelten die folgenden Aussagen für alle $x, y, z \in M$.

$$x \sqsubseteq x \qquad \textit{(Reflexivität)}$$
$$x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y \qquad \textit{(Antisymmetrie)}$$
$$x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z \qquad \textit{(Transitivität)}$$

Als Beispiel für eine solche partielle Ordnung lässt sich der Teilmengenoperator $\subseteq$ auf beliebigen Mengen betrachten. Wie man sich leicht klar machen kann, gilt für $\subseteq$ Reflexivität, Antisymmetrie und Transitivität.

In Abschnitt 6 wird eine partielle Ordnung genutzt, um Ausdrücke bezüglich ihrer *Definiertheit'* zu ordnen. Dazu müssen einige zusätzliche Definitionen eingeführt werden, um partielle Ordnungen weiter einzuschränken. Die folgenden Definitionen richten sich dabei nach Johann und Voigtländer [JV06].

Definition 4. *Wenn* $\sqsubseteq_X$ *ein kleinstes Element besitzt, dann ist* $\sqsubseteq_X$ *eine komplette Partialordnung und man nennt* X *geordnet. Dieses kleinste Element wird typischweise als* $\bot_X$ *(bottom) bezeichnet, wobei man den Index auch weglassen kann, wenn klar ist, welche Menge gemeint ist.*

Definition 5. *Sei* $\mathcal{R} \subseteq X \times Y$ *eine binäre Relation über zwei geordneten Mengen* X *und* Y. *Man nennt* $\mathcal{R}$ *strikt, wenn* $(\bot, \bot) \in \mathcal{R}$. $\mathcal{R}$ *wird total genannt, wenn für jedes Paar* $(x, y) \in \mathcal{R}$ *gilt, dass aus* $y = \bot$ *folgt* $x = \bot$.

Definition 6. *Man nennt* $\mathcal{R}$ *bottom-reflexiv, wenn für jedes Paar* $(x, y) \in \mathcal{R}$ *gilt:* $y = \bot \Leftrightarrow x = \bot$.

Schließlich soll *Stetigkeit* eingeführt werden, wozu zunächst die Definition einer *monotonen Sequenz* benötigt wird.

Definition 7. *Sei* X *eine Menge und* $\sqsubseteq_X$ *eine partielle Ordnung über* X. *Man nennt eine Funktion* $\bar{x} : \mathbb{N} \to X$ *eine monotone Sequenz über* X, *wenn* $\bar{x}(i) \sqsubseteq_X \bar{x}(i+1)$ *für alle* $i \in \mathbb{N}$.

Anders ausgedrückt ist eine monotone Sequenz also eine Menge von Elementen einer Ordnung, die jeder natürlichen Zahl ein Element aus der geordneten Menge zuordnet, sodass eine Erhöhung der Zahl auch eine Erhöhung des Elements bezüglich der Ordnung zur Folge hat. Auf dieser Definition von monotonen Sequenzen aufbauend lässt sich nun eine Definition für *Stetigkeit* formulieren, darüber hinaus nennen wir die Kombination aus Stetigkeit und Striktheit *zulässig*.

Definition 8. *Wenn für alle monotonen Sequenzen* $\bar{x}$ *über* X *und* $\bar{y}$ *über* Y, *die die Aussage* $(\bar{x}(i),\ \bar{y}(i)) \in \mathcal{R}$ *erfüllen, jeweils das Paar der Suprema* $(\bigsqcup \bar{x}, \bigsqcup \bar{y})$ *ebenfalls in* $\mathcal{R}$ *ist, dann nennt man* $\mathcal{R}$ *stetig.*

Definition 9. *Eine Relation ist zulässig, wenn sie strikt und stetig ist.*

Hat man zwei Relationen gegeben, lassen sich aus diesen neue Relationen konstruieren, beispielsweise durch die Relationskomposition.

Definition 10. *Sind drei Mengen X, Y und Z und zwei binäre Relationen $\mathcal{R} : X \Leftrightarrow Y$ und $\mathcal{R} : Y \Leftrightarrow Z$ gegeben, dann definiert man die Komposition von $\mathcal{R}$ und $\mathcal{S}$, geschrieben $\mathcal{R}; \mathcal{S}$, wie folgt.*

$$\mathcal{R} \ ; \ \mathcal{S} = \{(x, z) \mid \exists y \in Y.\ (x, y) \in \mathcal{R} \wedge (y, z) \in \mathcal{S}\} \subseteq X \times Z$$

In Worten ausgedrückt bedeutet das, dass man zwei Relationen zu einer neuen Relation zusammenfügt, in der zwei Elemente x und z verwandt sind, wenn es ein Bindeelement $y \in Y$ gibt, mit dem das erste Element x in der ersten Relation und das zweite Element z in der zweiten Relation verwandt ist.

Wir betrachten ein Beispiel, in dem die folgenden Mengen und Relationen definiert sind.

$$\begin{aligned}
&M = \{1, 2, 3\}, N = \{a, b, c\}, O = \{\alpha, \beta, \gamma\} \\
&\mathcal{R} : M \Leftrightarrow N, \mathcal{S} : N \Leftrightarrow O \\
&\mathcal{R} = \{(1, a), (2, b), (3, c)\} \\
&\mathcal{S} = \{(a, \alpha), (b, \beta), (c, \gamma)\}
\end{aligned}$$

Durch Komposition der beiden Relationen $\mathcal{R}$ und $\mathcal{S}$ ergibt sich die folgende Relation.

$$R; S = \{(1, \alpha), (2, \beta), (3, \gamma)\}$$

Mithilfe von Relationskomposition kann man nun sehr leicht *Linksabgeschlossenheit* definieren.

Definition 11. *Ist Y eine Menge und X eine bezüglich $\sqsubseteq_X$ geordnete Menge, dann nennt man eine Relation $\mathcal{R} \subseteq X \times Y$ linksabgeschlossen, wenn $\sqsubseteq_X; R = R$.*

Eine Relation $\mathcal{R}$ ist also linksabgeschlossen bezüglich einer Ordnung $\sqsubseteq_X$, wenn die Komposition dieser Ordnung mit $\mathcal{R}$ wieder $\mathcal{R}$ ergibt, oder in anderen Worten: wenn für jedes $(x, y) \in \mathcal{R}$ gilt, dass zu jedem $a \sqsubseteq_X x$ auch $(a, y) \in \mathcal{R}$.

Definition 12. *Das Inverse einer Relation $\mathcal{R} : X \Leftrightarrow Y$ wird definiert als $\mathcal{R}^{-1} = \{(y, x) \mid (x, y) \in \mathcal{R}\} \subseteq Y \times X$. Möchte man $\sqsubseteq^{-1}$ ausdrücken, nutzt man zuweilen auch das umgedrehte Symbol $\sqsupseteq$.*

Definition 13. *Seien X und Y geordnete Mengen und $f : X \to Y$ eine Funktion. Man nennt f monoton, wenn aus $x \sqsubseteq_X y$ folgt, dass $f(x) \sqsubseteq_Y f(y)$ für alle $x, y \in X$.*

Wie bereits erwähnt, ordnet eine Relation einem Element ein einzelnes Element, mehrere Elemente oder gar keine Elemente zu. Das unterscheidet sie von einer Funktion, oder anders ausgedrückt: Das macht Funktionen zu speziellen Relationen. Eine

Funktion, ebenfalls auf zwei Mengen definiert, ordnet jedem Element genau ein Ergebnis zu – Eingabewerte mit mehreren Funktionswerten sind genausowenig enthalten wie Eingabewerte ohne Ergebnis.

Wir betrachten eine beliebige Funktion $f : T_1 \to T_2$. Zu jedem $x \in T_1$ ist also $f(x)$ definiert, wobei $f(x) \in T_2$. Funktionen kann man auch wie folgt als Relationen darstellen.

$$\{(x, f(x)) \mid x \in T_1\}$$

Man nennt die zu f gehörige Relation auch den *Graphen* zu f. Umgekehrt heißt das natürlich, dass eine Relation unter Umständen auch eine Funktion ist. Das wird besonders interessant, wenn man über Relationen allquantifiziert, zu sehen im folgenden Beispiel.

$$\forall \mathcal{R} : T_1 \Leftrightarrow T_2 . A(\mathcal{R})$$

Hierbei ist $A(\mathcal{R})$ eine Aussage, die von $\mathcal{R}$ abhängt. Diese Aussage lässt sich verschärfen, indem man statt über Relationen über Funktionen $\forall f : T_1 \to T_2$ quantifiziert. Eine Aussage wie $(x, y) \in \mathcal{R}$ wird dann beispielsweise zu $f\ x = y$.

Man kann sich leicht klar machen, dass eine solche spezialisierte Aussage aus der ursprünglichen Aussage folgt. Gilt die ursprüngliche Aussage, gilt dementsprechend auch die speziellere Aussage.

2.2 Haskell

Die in dieser Arbeit verwendete Bibliothek generiert freie Theoreme für in Haskell geschriebene Programme. In diesem Abschnitt wird noch einmal speziell auf das Sprachkonstrukt der Typkonstruktorklassen eingegangen. Eine generelle Beschreibung der Sprache Haskell würde an dieser Stelle zu weit führen, hier sei auf [Jon03] verwiesen.

Typklassen in Haskell ermöglichen die Umsetzung von Ad-Hoc-Polymorphismus. Man kann beliebige Typklassen selbst definieren, wobei eine Typklasse immer aus einem Namen, beliebig vielen Funktionssignaturen und gegebenenfalls auch aus vordefinierten Standardimplementierungen einzelner Funktionen besteht. Haskell transformiert Programme, die Typklassen verwenden, intern in Programme ohne dieses Konstrukt – man nennt diesen Vorgang auch Wörterbuchtransformation (engl. “dictionary translation”) [Jon93].

Es folgt ein einfaches Beispiel, das auch in der Prelude von Haskell enthalten ist.

```
class Show a where
    show :: a -> String
```

Der Name dieser Beispielklasse ist `Show`. Man gibt eine Typvariable an, in diesem Fall `a`, die auch in jeder Funktionssignatur vorkommen muss. In dieser Beispielklasse wird genau eine Funktionssignatur eingeführt: Die Signatur `a -> String` für die Funktion `show`.

Im weiteren Haskellprogramm können nun für beliebige Datentypen Instanzdeklarationen angegeben werden. Der folgende Quellcode zeigt ein Beispiel für einen selbstdefinierten Datentyp.

```
data Test = Test Int String

instance Show Test where
    show (Test _ s) = "Test " ++ s
```

Kommt nun im Programm in einem Ausdruck der Funktionsaufruf `show` vor, so wird anhand des Parametertypen entschieden, welche Instanz dieser Klasse verwendet wird. Es kann also für jeden Typen eine komplett eigene Implementierung angegeben werden.

Hat man eine Typklasse definiert, kann man im Folgenden in beliebigen Typsignaturen einen Kontext angeben, der bestimmte Typvariablen auf bestimmte Klassen festlegt, wie es das folgende Beispiel zeigt.

```
test :: Functor f => f Int -> f Int
```

Hier ist `f` eine freie Typvariable, für die jedoch ein Kontext angegeben ist, der besagt, dass für f nur Instanzen der Typklasse `Functor` infrage kommen. Im Gegenzug für diese Einschränkung ist es in der Implementierung möglich, die Funktionen dieser Klasse mit dieser Variable zu nutzen, da sichergestellt ist, dass Implementierungen dieser Funktionen existieren. Die folgende Funktionsdefinition ist also eine mögliche Implementierung der obigen Beispielsignatur.

```
test x = fmap (+1) x
```

Haskell erkennt, dass `x` ein Typ sein muss, für den eine `Functor`-Instanz deklariert ist, weshalb eine `fmap`-Implementierung existiert.

Nun ist es ja auch möglich, dass ein selbstdefinierter Datentyp einen Typ als Parameter erwartet, wie das folgende Beispiel zeigt.

```
data Test a = Test a String
```

Beim Ausdruck `Test` handelt es sich in diesem Fall also nicht um einen Typ, sondern eher um einen funktionsähnlichen Ausdruck, der einen Typ als Parameter erwartet und einen Typ zurückgibt. Man spricht hier auch von *Sorten* (engl. “kinds”). Haskell führt für Sorten eine eigene Notation ein. Alle Basistypen wie `Int`, `String`, etc. sind von der Sorte *. Typkonstruktoren, die einen Typen als Parameter erwarten und daraus einen neuen Typen konstruieren, sind von der Sorte $* \to *$. Kompliziertere Sorten wie zum Beispiel $* \to (* \to *) \to *$ sind theoretisch ebenfalls möglich, werden aber in dieser Arbeit keine Anwendung finden.

Typklassen können auch für Datentypen definiert werden, die Typparameter erwarten. Das folgende Beispiel zeigt die Definition für die Typklasse `Functor`, die für einen Typen definiert ist, der wiederum einen Typparameter erwartet.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Hier ist `f` die Typvariable. Die Signatur für `fmap` lässt erkennen, dass `f` auf einen Parameter appliziert wird, f muss also von der Sorte $* \to *$ sein. Zu beachten ist hier, dass es keinen Sinn macht (und auch nicht gestattet ist), dass die Typvariable mit unterschiedlichen Argumentzahlen genutzt wird. Ebenso wäre es ungültig, eine Instanz von `Functor` zu definieren für einen Datentyp, der keine Typparameter erwartet.

Nachfolgend ist ein Beispiel für ein ungültiges Programm zu sehen, in dem ein Datentyp namens `Test` deklariert wird und in einer `instance`-Deklaration eine Instanz für die Typklasse `Functor` beschrieben wird.

```
data Test = Test String Int

instance Functor Test where
  fmap = undefined
```

Davon abgesehen, dass `fmap` als undefiniert deklariert ist (eine sinnvolle Implementierung ist hier ohnehin nicht möglich), ergibt diese Deklaration bereits zur Kompilierzeit einen Fehler, den GHC folgendermaßen darstellt.

```
The first argument of `Functor' should have kind `* -> *',
  but `Test' has kind `*'
In the instance declaration for `Functor Test'
```

Haskell meldet also, dass der Datentyp `Test` von der falschen Sorte ist: Statt der erwarteten Sorte $* \to *$ wird die Sorte $*$ vorgefunden und es kommt zum Kompilierfehler.

3 Freie Theoreme

In der Einleitung wurde bereits kurz thematisiert, dass man allein durch Betrachten der Typsignatur einer Funktion bereits eine grundlegende, intuitive Vorstellung davon erhalten kann, welche Funktionalität sich ungefähr hinter dieser Funktion verbirgt. Haskells Typsystem sorgt dafür, dass Funktionen in ihren Implementierungsmöglichkeiten teilweise stark eingeschränkt sind. Der Grund hierfür ist der parametrische Polymorphismus, der in Haskellfunktionen mit Typvariablen Anwendung findet. Nicht nur verringert das Typsystem die Fehleranfälligkeit von produziertem Code, es vereinfacht auch Rückschlüsse auf die Funktionsweise und Korrektheitsbeweise.

Außerdem wurde bereits angedeutet, dass es eine Möglichkeit gibt, systematisch Theoreme zu gegebenen Typsignaturen herzuleiten. Diese sogenannten *freien Theoreme*, ihre Herleitung und ihre Verwendung sind Thema dieses Kapitels. Zunächst wird an einem Beispiel erläutert, wie die Vorgehensweise intuitiv zu erklären ist. Der darauf folgende Abschnitt behandelt die formale Herleitung eines freien Theorems zu einer beliebigen Typsignatur nach Wadler [Wad89].

Daraufhin wird die Erweiterung des Vorgehens um Typkonstruktorklassen thematisiert, wie sie auch von Voigtländer verwendet wird [Voi09]. Am Ende des Kapitels wird schließlich an einem praktischen Beispiel gezeigt, wie ein freies Theorem für einen Beweis verwendet werden kann.

3.1 Intuition

Bevor die theoretische Herangehensweise beschrieben wird, bietet es sich zunächst an, die Vorgehensweise intuitiv an einem Beispiel zu erklären. Dabei werden noch keine Definitionen eingeführt oder Formeln aufgestellt, es geht vielmehr darum, ein Gefühl dafür zu vermitteln, warum es überhaupt möglich sein soll, Theoreme allein aus der Signatur eines Ausdrucks abzuleiten. Im nächsten Abschnitt wird dieses Vorgehen dann konkretisiert und für beliebige Typsignaturen verallgemeinert.

Als einfaches Beispiel dient die Funktion mit der folgenden Typsignatur.

$$f :: [a] \to [a] \tag{1}$$

Gegeben ist also eine Funktion f, die eine Liste auf eine Liste abbildet. Das Besondere an dieser Funktion ist, dass sie nicht für einen konkreten Typ wie beispielsweise *Integer* deklariert ist, sondern dass man sie für jeden beliebigen Typ instanziieren kann, indem man die Typvariable a durch einen Typausdruck ersetzt. Die Funktion ist also für jeden beliebigen Typ deklariert, es handelt sich um eine *polymorphe Funktion*.

Bei der Typvariable a handelt es sich erst einmal um eine ungebundene, *freie* Variable. Man kann sich aber einen impliziten Allquantor vor der eigentlichen Signatur denken. Tatsächlich existiert dieser Allquantor sogar als spezielles Schlüsselwort `forall` in Haskell, wie die folgende Schreibweise zeigt[3].

[3] Um die forall-Notation verwenden zu können, muss allerdings die `ExistentialQuantification`-Spracherweiterung aktiviert werden.

```
f :: forall a. [a] -> [a]
```

Im Normalfall wird dieses Schlüsselwort jedoch nicht explizit benötigt und deshalb meistens weggelassen. Immer, wenn freie Typvariablen in einer Signatur vorkommen, lässt sich diese Signatur also *schließen*, indem für jede freie Typvariable ein entsprechender Allquantor an den Anfang gesetzt wird.

Die Instanziierung einer polymorphen Funktion auf einen konkreten Typ wird im weiteren Verlauf der Arbeit notiert, indem der polymorphe Ausdruck mit einem Index versehen wird, der dem Typ entspricht, auf den der Ausdruck instanziiert wird. $f_{Integer}$ ist dann beispielsweise die Funktion f, konkretisiert auf den Typ Integer, d.h. es wird Integer für jedes a in die Funktionssignatur eingesetzt. $f_{Bool \to Bool}$ wäre folglich die gleiche Funktion, instanziiert auf den Typ $Bool \to Bool$, sie hätte also die Signatur $[Bool \to Bool] \to [Bool \to Bool]$.

Hat die Funktion mehrere verschiedene Typvariablen, so werden die einzelnen Typen in der Reihenfolge des Auftretens der zu ersetzenden Typvariablen angegeben. Eine Funktion mit der Signatur `g :: a -> b` instanziiert man folglich mit $g_{Integer\ String}$ auf den Typ $Integer \to String$.

Ein entscheidendes Detail bei polymorphen Funktionen in Haskell ist die Art des Polymorphismus. Man unterscheidet zwischen Ad-Hoc Polymorphismus und parametrischem Polymorphismus [Str00]. Bei Typvariablen in Haskell handelt es sich um parametrischen Polymorphismus, das heißt die entsprechende Funktion arbeitet für sämtliche eingesetzten Typen nach demselben Prinzip.

Erst einmal klingt das nicht besonders außergewöhnlich, es zieht aber wichtige Konsequenzen mit sich. “Dasselbe Prinzip” heißt hier nämlich, dass unabhängig vom eingesetzten Typ die gleichen Berechnungen stattfinden. Die Funktion kann nicht einmal durch Fallunterscheidung ermitteln, um welchen konkreten Typ es sich im jeweiligen Fall handelt.

Das ist deswegen etwas Besonderes, weil dadurch ermöglicht wird, allgemeine Aussagen zu treffen, die für sämtliche Typinstanziierungen gelten. Vergleicht man diese Methode mit der zweiten Art von Polymorphismus, dem *Ad-Hoc-Polymorphismus*, wird der Vorteil klar: Unter Letzterem versteht man, dass eine Funktion zwar für verschiedene Typen definiert wird, diese jedoch jeweils eine konkrete, typspezifische Implementierung haben.

Es handelt sich also technisch gesehen um verschiedene Funktionen, die sich einen Namen teilen können, weil anhand der Typsignatur und der Typen der Parameter, mit denen die Funktion aufgerufen wird, entschieden werden kann, welche der Funktionen jeweils gemeint ist. Ein Beispiel für Ad-Hoc-Polymorphismus bietet die Programmiersprache *C++* [Str95]: Hier ist es möglich, mehrere Funktionen mit gleichem Namen in einer Klasse zu deklarieren, solange die Parameter unterschiedliche Signaturen besitzen, wie Listing 1 zeigt.

Hier werden drei Methoden deklariert, die alle den Namen `log` haben, sich jedoch durch den Typ des Parameters unterscheiden. Die jeweilige Implementierung, die hier nicht zu sehen ist, kann sich von Funktion zu Funktion komplett unterscheiden. Auch

```
class Log
{
public:
    void log(TextObject* t);
    void log(Image* t);
    void log(Test* t);
}
```

Listing 1: Ad-Hoc-Polymorphismus in C++

Haskell bietet mit den in der Einleitung bereits eingeführten `Typklassen` Ad-Hoc-Polymorphismus [WB89].

Beim Ad-Hoc-Polymorphismus kann also jede Funktion eine eigene Implementierung haben, außerdem ist jeder Funktion bekannt, auf welchem Typ sie arbeitet, was es ihr ermöglicht, typspezifische Operationen anzuwenden. Es liegt nahe, dass dies allgemeingültige Aussagen über solche Funktionen sehr stark einschränkt.

Wir betrachten wieder die Beispielfunktion f aus (1). Durch parametrischen Polymorphismus ist sie also für jeden beliebigen Typ a definiert. Das hat Konsequenzen für die Funktionsweise dieser Funktion.

Man kann sich an dieser Stelle die Frage stellen: Wie kann die Funktion die Liste erzeugen, die sie zurückgeben muss? Sie kennt den konkreten Typ nicht, das heißt sie kann keine neuen Elemente "erfinden", weil sie keine Informationen darüber hat, welchen Typ die Listenelemente haben müssen. Und Ausdrücke, die zu jedem beliebigen Typ passen, gibt es in Haskell nicht[4]. Eine Möglichkeit hat sie aber dennoch: Sie kann die Elemente der Eingabeliste wiederverwenden. Die einzige Möglichkeit für die Funktion, die Liste zu ändern, besteht also in einer Umsortierung der Listenelemente.

Doch auch für eine Neusortierung der Liste hat die Funktion wenige Anhaltspunkte. Sie kennt zwar die Länge der Liste, doch ohne Kenntnis des Typen kann sie keine weiteren Aussagen über den Inhalt der einzelnen Listenelemente treffen (und da die Klasse Eq nicht gegeben ist, nicht mal über deren Gleichheit). Auch hat sie aufgrund der referenziellen Transparenz in Haskell keinen globalen Kontext oder Ähnliches, anhand dessen sie Entscheidungen treffen könnte. Sie wird also gezwungenermaßen je zwei Listen gleicher Länge stets gleich behandeln.

In der Einleitung wurde bereits die Haskell-Funktion `map` eingeführt, die eine Funktion auf jedes Element einer Liste anwendet und eine Liste gleicher Länge zurückgibt, in dem jedes Element durch die Eingabefunktion manipuliert wurde.

Haskell verwendet für Funktionsapplikation das Prinzip des Currying, das heißt, dass eine Funktion, die auf ein Argument angewandt wird, eine neue Funktion zurückgibt, in der dieses Argument fest gesetzt ist, die also einen Parameter weniger erwartet. Man

[4]Tatsächlich kann man undefinierte Werte als Ausdrücke ansehen, die zu einem beliebigen Typ generiert werden können – diese werden jedoch erst einmal außen vor gelassen. In Kapitel 6 wird etwas genauer darauf eingegangen.

kann die *map*-Funktion also auch anders interpretieren als Funktion, die eine Funktion als Parameter erwartet und eine neue Funktion zurückgibt. Diese neue Funktion erwartet wiederum eine Liste als Parameter und gibt eine Liste zurück.

Der folgende Ausdruck wandelt also gewissermaßen die Eingabefunktion `f` in eine neue Funktion um, die die Funktionalität von `f` für Listen bietet.

```
map f
```

Man kann auch sagen: `map` *liftet* die Funktion `f` in den Listen-Kontext. Im Folgenden nehmen wir an, dass die Funktion f auf einen beliebigen konkreten Typ instanziiert wurde. Wir betrachten jetzt eine beliebige nicht-polymorphe Funktion g, die auf diesem konkreten Typ arbeitet. *map g* liefert folglich eine Funktion, die g auf sämtliche Elemente einer Eingabeliste anwendet. Jede Liste, die durch Anwendung von *map g* entsteht, wird die gleiche Anzahl an Elementen haben wie die ursprüngliche Liste. Darüber hinaus vertauscht *map g* die Listenelemente nicht, jedes neue Element wird an derselben Stelle stehen wie sein Ursprungselement.

Auf dieser Argumentation aufbauend kann man erkennen, dass es unerheblich für die Ergebnisliste ist, ob man zuerst die Funktion f auf eine Liste appliziert und dann auf die resultierende Liste die Funktion *map g* anwendet, oder ob man in umgekehrter Reihenfolge vorgeht. Abbildung 1 veranschaulicht dies für eine Beispielfunktion f, die Listen umkehrt, und eine Funktion g, die zu jedem Listenelement den Wert 1 hinzuaddiert. Hervorgehoben ist die Position eines Elements in dieser Liste. Im linken Teil der Abbildung wird zunächst die Funktion f appliziert und als zweites *map g*, im rechten Teil wird *map g* zuerst angewendet und dann erst f.

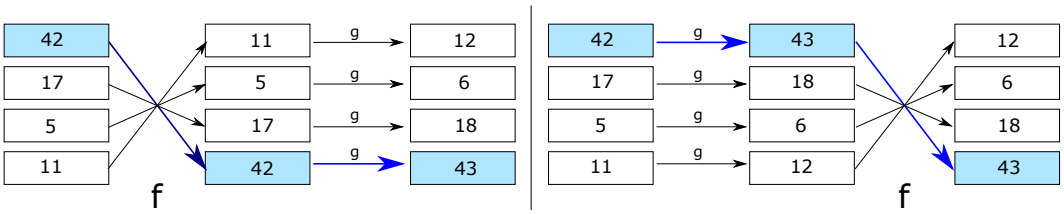

Abbildung 1: Die Reihenfolge der Anwendung von f und *map g* spielt keine Rolle für die Ergebnisliste.

Die folgende Formel fasst die Aussage der obigen Argumentation als mathematische Aussage zusammen.

$$f(map\ g\ l) = map\ g\ (f\ l)$$

Diese Aussage ist ein Theorem, das für beliebige Funktionen g und beliebige Listen l gilt. Hierbei ist zu beachten, dass g hier keine polymorphe Funktion ist, es ist eine beliebige Funktion, die auf dem *konkreten* Typ von f arbeitet d.h. auf dem Typ, auf

den f instanziiert wird. Ebenso handelt es sich bei l um eine Liste ebendieses konkreten Typs.

Das Bemerkenswerte ist jedoch nicht diese allgemeine Aussage an sich, sondern die Grundlage, auf der sie getroffen wurde. Überlegt man sich einmal, anhand welcher Informationen diese Schlussfolgerung gezogen wurde, wird klar, dass das einzige für diese Herleitung benötigte Wissen die Typsignatur der Funktion f ist. Insbesondere wurde die Funktion nicht auf einen konkreten Typ instanziiert, die Aussage gilt für beliebige Typen.

Natürlich war die vorangegangene Herleitung sehr unkonkret und nicht wirklich mathematischer Natur. Im Folgenden wird dieser intuitive Ansatz nun mathematisch konkretisiert.

3.2 Parametrizität

Der Schlüssel zur Herleitung von freien Theoremen liegt in der Betrachtung von Typen. Intuitiv werden Typen häufig als Mengen aufgefasst. So würde man den Typ `Bool` beispielsweise als Menge $\mathbb{B}$ der booleschen Werte auffassen mit $\mathbb{B} = \{tt, ff\}$, der Typ `Integer` wäre die Menge der ganzen Zahlen $\mathbb{Z}$ usw. Komplexere Typen ergeben sich dann durch Zusammensetzung aus einzelnen Basistypen, beispielsweise würde man den Typen $Integer \to Integer$ auffassen als die Menge $(\mathbb{Z} \to \mathbb{Z})$ aller Funktionen von den ganzen Zahlen in die ganzen Zahlen.

Die grundsätzliche Idee hinter freien Theoremen basiert auf der Erkenntnis von Reynolds, dass die Semantik eines beliebigen Ausdrucks verwandte Umgebungen auf verwandte Werte abbildet [Rey83]. Dieses Theorem, von Reynolds das *Abstraktionstheorem* genannt, wird von Wadler umformuliert zum sogenannten *Parametrizitätstheorem* [Wad89], das diese Erkenntnis für polymorphe Funktionssignaturen nutzbar macht und die Herleitung von Theoremen ermöglicht. Das heißt allerdings, dass man Typen als Relationen auffassen muss statt als Mengen, um dieses Theorem anwenden zu können.

Um Typen als Relationen aufzufassen, definiert man sogenannte *relationale Aktionen*, mit denen zu jedem Typ eine entsprechende Relation konstruiert werden kann. Johann und Vogitländer definieren relationale Aktionen wie folgt [JV06].

Definition 14 (Relationale Aktion)**.** *Zu jedem n-stelligen Typkonstruktor wird eine Abbildung definiert, die aus n Relationaldarstellungen eine neue Relation konstruiert. Diese Abbildung nennt man relationale Aktion.*

Man kann also zu einem beliebigen Typ eine Relation konstruieren, die diesen Typ repräsentiert, indem man zu jedem Typkonstruktor die entsprechende relationale Aktion anwendet. Die einzelnen Relationen werden nach Voigtländer [Voi09] folgendermaßen konstruiert, wobei wir die Menge aller Typmengen als $Typen$ bezeichnen.

- Zu jedem Basistyp T (`Int`, `Bool`, etc.) ist die zugehörige Relation die Identitätsrelation $\mathcal{R} = id_T = \{\ (x, x) \mid x \in T\ \}$.

- Sind $\mathcal{R} : R_1 \Leftrightarrow R_2$ und $\mathcal{S} : S_1 \Leftrightarrow S_2$ Typrelationen, so ist

$$\mathcal{R} \to \mathcal{S} = \{(f, g) \mid f \in (R_1 \to S_1), g \in (R_2 \to S_2), \\ \forall x \in R_1, y \in R_2 : (f\ x, g\ y) \in \mathcal{S}\}$$

  die zugehörige Funktionstyprelation.

- Ist $\mathcal{F}(\mathcal{X})$ eine Typrelation, die von einer Typrelation $\mathcal{X} : X \Leftrightarrow X'$ abhängt, dann definiert man

$$\forall\mathcal{X}.\mathcal{F}(\mathcal{X}) = \{\ (x, y) \mid x \in X, y \in X', \forall A_1, A_2 \in Typen, \\ \forall\mathcal{A} : A_1 \Leftrightarrow A_2.\ (x_{A_1}, y_{A_2}) \in \mathcal{F}(\mathcal{A})\ \}$$

  Hierbei sind x und y polymorphe Ausdrücke.

Die Konstruktion zu Basistypen sagt ganz einfach, dass in der resultierenden Relation jeder Wert nur mit sich selbst verwandt ist. Betrachten wir als Beispiel die Relation zum Typ `Bool`. Als Menge aufgefasst stellt sich dieser Datentyp dar wie folgt.

$$Bool = \{\ True, False\ \}$$

Die zu diesem Typ konstruierte Relation ist nun die Identitätsrelation, $True$ ist also mit $True$ verwandt, $False$ ist mit $False$ verwandt. Sonstige Beziehungen gibt es nicht zwischen den Werten.

Funktionstypkonstruktoren bauen auf zwei bereits bestehenden Typrelationen $\mathcal{R}$ und $\mathcal{S}$ auf. Wir bleiben beim einfachen Beispiel `Bool` und erweitern die Signatur zum Typ `Bool -> Bool`.

Induktiv werden nun also zunächst die Relationen zu den Basistypen konstruiert, in diesem Fall id_{Bool}. Darauf aufbauend wird dann aus den zwei Identitätsrelationen die Funktionstyprelation gebaut. Die Intuition hinter der oben gegebenen Formel ist die folgende: Es wird eine Relation von Funktionen konstruiert, in denen zwei Funktionen genau dann verwandt sind, wenn sie verwandte Parameter auf verwandte Ergebnisse abbilden.

Die Konstruktion $id_{Bool} \to id_{Bool}$, die sich aus dem Beispiel ergibt, führt also zu einer Relation $\mathcal{R} : (Bool \to Bool) \Leftrightarrow (Bool \to Bool)$, in der solche Funktionen verwandt sind, die verwandte Argumente aus der ersten Relation auf verwandte Ergebnisse in der zweiten Relation abbilden. Da es sich bei beiden Relationen um Identitätsrelationen handelt, können nur gleiche Werte verwandt sein, und somit können auch nur Funktionen verwandt sein, die gleiche Werte auf gleiche Werte abbilden – es können also auch nur gleiche Funktionen verwandt sein.

Alle Basistypen und alle Funktionen auf Basistypen sind also Identitätsrelationen. Bis hierhin bringt die Verwendung von Relationen statt Mengen noch keinen wirklichen Vorteil, da immer wieder Identitätsrelationen entstehen. Interessant wird es erst beim Auftreten von Typabstraktionen der Form $\forall\mathcal{R}.\mathcal{F}(\mathcal{R})$, wobei $\mathcal{F}(\mathcal{R})$ eine Relation ist, die

von einer Relation $\mathcal{R}$ abhängt. Wie in der obigen Definition der entsprechenden *relationalen Aktion* beschrieben, sind zwei Werte dann verwandt bezüglich der konstruierten Relation, falls sie bezüglich jeder Relation $\mathcal{A}$ auf beliebigen Typen verwandt sind, sofern sie auf deren jeweiligen Typ instanziiert werden.

Auch für Datentypen lassen sich Relationen konstruieren. Ein Beispiel ist der Listentypkonstruktor `[]`, für den man folgende relationale Aktion rekursiv definiert, wenn eine Typrelation $\mathcal{R} : T_1 \Leftrightarrow T_2$ gegeben ist.

$$[\mathcal{R}] = \{([], [])\} \cup \{(a : as, b : bs) \mid (a, b) \in \mathcal{R}, (as, bs) \in [\mathcal{R}]\}$$

Andere Standardtypkonstruktoren lassen sich auf ähnliche Weise beschreiben. Nun haben wir die Mittel, zu jeder vorhandenen Typsignatur eine Relation zu konstruieren. Eine entscheidende Erkenntnis von Wadler [Wad89], aufbauend auf Reynolds [Rey83], ist nun das Parametrizitätstheorem. Hierzu muss noch der Begriff *geschlossener Term* definiert werden.

Definition 15. *Ein Term ist geschlossen, wenn er keine freien Termvariablen enthält.*

Diese Definition ist zwar relativ selbsterklärend, ist aber wichtig für das folgende Theorem, da dieses Theorem von geschlossenen Termen ausgeht.

Theorem 1 (Parametrizität)**.** *Ist t ein geschlossener Term von Typ T und $\mathcal{T}$ die zugehörige konstruierte Typrelation, dann gilt $(t, t) \in \mathcal{T}$.*

Damit ist alles gegeben, was man zur Herleitung des freien Theorems benötigt. Wir untersuchen wieder das Beispiel $f :: [a] \to [a]$. Durch Parametrizität ist gegeben:

$$(f, f) \in \forall\mathcal{R}.[\mathcal{R}] \to [\mathcal{R}]$$

Es wurde bereits zu Beginn des Kapitels erwähnt, dass freie Typvariablen in Signaturen mit einem impliziten Allquantor quantifiziert werden, weshalb hier die entsprechende relationale Aktion Anwendung findet. Wir können nun Schritt für Schritt die Definitionen der einzelnen Relationskonstruktionen anwenden, wodurch die Formel abgerollt wird.

$$\begin{aligned} &\forall A_1, A_2 \in \mathit{Typen}, \mathcal{A} : A_1 \Leftrightarrow A_2 : \\ &\quad (f_{A_1}, f_{A_2}) \in [\mathcal{A}] \to [\mathcal{A}] \\ \\ \Leftrightarrow\ &\forall A_1, A_2 \in \mathit{Typen}, \mathcal{A} : A_1 \Leftrightarrow A_2 : \\ &\quad \forall (l, l') \in [\mathcal{A}] : \\ &\quad\quad (f_{A_1} l, f_{A_2} l') \in [\mathcal{A}] \end{aligned}$$

An diesem Punkt haben wir eine Aussage über beliebige Typrelationen $\mathcal{A}$, anschaulicher wäre aber eine Aussage über beliebige Funktionen. Da man Funktionen als Spezialfall von Relationen betrachten kann, können wir die Aussage spezifizieren für den Fall, dass es sich bei $\mathcal{A}$ um eine Funktion handelt. Wir quantifizieren dazu statt über alle Relationen $\mathcal{A} : A_1 \Leftrightarrow A_2$ über alle Funktionen $g : A_1 \to A_2$.

Aus dem abgerollten Parametrizitätstheorem haben wir ja bisher die Aussage abgeleitet, dass für alle Parameter $l, l' \in [\mathcal{A}]$ die Aussage $(f_{A_1} l, f_{A_2} l') \in [\mathcal{A}]$ gilt. Als Funktion geschrieben heißt das, es gilt die folgende Aussage, wobei der Ausdruck $lift_{[]}(g)$ bedeuten soll, dass die Funktion g in den Listenkontext gehoben wird (die Schreibweise $[g]$ würde zu unnötigen Mehrdeutigkeiten führen).

$$lift_{[]}(g)(f_{A_1}\ l) = f_{A_2}\ l'$$

Das Lifting von Funktionen in den Listenkontext kann man auch anders ausdrücken als das Mapping dieser Funktionen auf Listen, sprich: die Haskell-Funktion *map*, angewandt auf die entsprechende Funktion.

$$map\ g\ (f_{A_1}\ l) = f_{A_2}\ l'$$

Hierbei ist zu bemerken, dass nun nicht mehr über alle Paare $(l, l') \in [\mathcal{A}]$ quantifiziert werden muss, da es zu jedem $l \in [A_1]$ nur genau ein $l' \in [A_2]$ geben kann, für das gilt $map\ g\ l\ =\ l'$. Folglich kann man die Variable l' weglassen und stets den Ausdruck $map\ g\ l$ verwenden. Ersetzt man die Variable l' also entsprechend, so erhält man die folgende Aussage.

$$map\ g\ (f_{A_1} l) = f_{A_2}(map\ g\ l)$$

Und man erkennt: Es handelt sich um die gleiche Aussage, die wir vorher intuitiv konstruiert haben. Jetzt haben wir bewiesen, dass sie gültig ist, da Parametrizität gilt. Und ein Theorem dieser Art lässt sich für jede beliebige Typsignatur herleiten.

3.3 Typkonstruktorklassen und -variablen

Die bisherige Theorie basierte auf Wadler [Wad89] und sah kein Auftreten von Typklassen oder Typkonstruktorklassen vor. In Abschnitt 3.2 wurde angegeben, wie beispielsweise der Listendatentyp in eine Relationaldarstellung überführt werden kann, und auf dieselbe Art und Weise lassen sich entsprechende Repräsentationen für beliebige Datentypen herleiten, so zum Beispiel $Maybe$, wie das folgende Beispiel zeigt.

$$Maybe\ \mathcal{R} = \{\ (Nothing,\ Nothing)\ \} \cup \{\ (Just\ x, Just\ y)\ |\ (x, y) \in \mathcal{R}\}$$

Dass in Typsignaturen aber nicht nur über Typvariablen, sondern auch über Variablen allquantifiziert werden kann, die ihrerseits nicht Platzhalter für spezielle Typen, sondern für Typkonstruktoren sind, wurde bisher außer Acht gelassen. In diesem Abschnitt geht es nun also um Typkonstruktorvariablen von der Sorte $* \to *$, wie es von Voigtländer [Voi09] vorgestellt wird. In der Einleitung wurde bereits die Funktion $fmap$ als Beispiel angegeben, die die folgende Typsignatur besitzt.

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Die Variable f wird über den Kontext auf die Typklasse `Functor` eingeschränkt. Letztere ist von der Sorte $* \to *$, es handelt sich also um einen Typkonstruktor, der einen Typ auf einen neuen Typ abbildet. Daher ist es in der Typsignatur legitim, diese Variable auf Typparameter anzuwenden, in diesem Beispiel also in $f\ a$ und $f\ b$. Zwei Dinge müssen zum bisherigen Ansatz hinzugefügt werden: Allquantifizierung über Typkonstruktorvariablen muss separat betrachtet werden gegenüber Allquantifizierung über gewöhnlichen Typvariablen, und Klasseneinschränkungen müssen beachtet werden, im obigen Beispiel `Functor f`.

Die Allquantifizierung über Typkonstruktorvariablen funktioniert an sich ähnlich wie die Allquantifizierung über gewöhnliche Typvariablen. Statt als beliebige Relationen über beliebig gewählten Typen interpretiert man freie Typkonstruktorvariablen als Funktionen über beliebigen Typkonstruktoren k_1, k_2, die eine Relation auf eine neue abbilden, indem sie die Typkonstruktoren auf die Typmengen der Relation anwenden.

Es bietet sich an, an einem Beispiel zu zeigen, wie eine solche Relationsfunktion aussehen kann. Die folgende Formel zeigt eine Funktion $\mathcal{F} : Maybe \Leftrightarrow []$, die eine Relation $\mathcal{R}$ auf eine neue Relation $\mathcal{F}\ \mathcal{R}$ abbildet.

$$\mathcal{F}\ \mathcal{R} = \{(Nothing, [])\} \cup \{(Just\ a, [b]) | (a, b) \in \mathcal{R}\}$$

Diese Relation hat Ähnlichkeiten mit den Relationen, die für Basistypkonstruktoren wie Listen oder $Maybe$ konstruiert werden. Das Besondere an Typkonstruktorvariablen ist eben, dass man eine Aussage über *alle* Relationsfunktionen trifft, die infrage kommen. Und es kommen nicht alle Funktionen infrage. Im obigen Beispiel wird die Typkonstruktorvariable auf die Typkonstruktorklasse `Functor` eingeschränkt, deren Deklaration wie folgt aussieht.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Wie man sieht, hat die Klasse `Functor` einen Typparameter f und definiert die Typsignatur zu $fmap$. Stößt man jetzt in einer Typsignatur auf eine Klassenbeschränkung wie $Functor\ f$, muss man die Allquantifizierung über Typkonstruktoren weiter einschränken.

Zum einen dürfen die resultierenden Relationen natürlich nur auf Typen definiert sein, für die Instanzen der entsprechenden Klasse deklariert sind, in diesem Beispiel also nur Funktoren. Zum anderen muss man aber auch fordern, dass Parametrizität bei

Verwendung dieser Typen erhalten bleibt, das heißt die Typen der einzelnen Klassenfunktionen müssen ihrerseits mit sich selbst verwandt sind bezüglich ihrer konstruierten Relation, wenn man für die Typkonstruktorvariable die Funktion $\mathcal{F}$ einsetzt.

Formal wird eine Allquantifizierung über Typkonstruktorvariablen in eine Relationsfunktion statt in eine Relation überführt. Die zu überführende Typsignatur ist dabei von der folgenden Art.

```
f :: C t => A(t)
```

Hierbei soll `A(t)` ein Typausdruck sein, der die Typkonstruktorvariable `t` enthält. Man überführt diese Signatur wieder wie üblich in eine entsprechende Relationaldarstellung $\forall^{\{C\}}\mathcal{T}.\mathcal{A}(\mathcal{T})$, wobei $\mathcal{A}$ die Typrelation zu `A` ist, abhängig von einer Funktion auf Relationen $\mathcal{T}$. Die Relationskonstruktion ist dabei wie folgt definiert, wobei es es sich hier genau genommen um die Konstruktion einer Funktion auf Relationen handelt.

$$
\begin{aligned}
&(x, x') \in \forall^{\{C\}}\mathcal{T}.\mathcal{A}(\mathcal{T}) \\
\Leftrightarrow\ &\forall k_1, k_2 \text{ Typkonstruktoren für C, die C einhalten} \\
&\quad \forall \mathcal{F} : k_1 \Leftrightarrow k_2 \\
&\quad\quad (x_{k_1}, x'_{k_2}) \in \mathcal{A}(\mathcal{F})
\end{aligned}
$$

Es wird also über alle Typkonstruktoren und über alle Funktionen quantifiziert, die Relationen auf Relationen abbilden, wobei die ursprünglichen Typmengen auf die Typmengen abgebildet werden, die durch Anwendung der Typkonstruktoren entstehen. Die Beschränkung auf "Typkonstruktoren für C, die C einhalten" ist die bereits angesprochene Einschränkung auf diejenigen Funktionen, die für die entsprechende Typklasse überhaupt infrage kommen. Man kann diese Einschränkung zusammenfassend durch die folgenden beiden Kriterien beschreiben.

- Die entsprechende Funktion muss Relationen auf solche Relationen abbilden, die auf Typen definiert sind, für die es Instanzdeklarationen für die Klasse gibt.
- Die entsprechende Funktion muss Relationen auf solche Relationen abbilden, für die sämtliche Klassenfunktionen wiederum mit sich selbst verwandt sind bezüglich ihrer jeweiligen Relationaldarstellung, wenn für die entsprechende Typkonstruktorvariable die Funktion eingesetzt wird.

Um wieder auf das ursprüngliche Beispiel der `Functor`-Klasse zurückzukommen, ist im Folgenden die Bedingung gegeben, die gelten muss für eine Funktion $\mathcal{F}$, damit diese die Klasse `Functor` einhält.

$$(fmap_{k_1}, fmap_{k_2}) \in \forall\mathcal{R}.\ \forall\mathcal{S}.(\mathcal{R} \to \mathcal{S}) \to \mathcal{F}\mathcal{R} \to \mathcal{F}\mathcal{S}$$

Wird nun eine Typkonstruktorvariable auf einen Typ angewandt, handelt es sich nicht um eine relationale Aktion, die Anwendung findet. Letztere wäre ja um Konstruktion von Relationen aus Typen, basierend auf existierenden Relationen. Typkonstruktorvariablen sind aber keine Relationen, es handelt sich um Funktionen – und die Applikation dieser Funktionen auf Typrelationen ist auch tatsächlich einfach eine Funktionsanwendung, deren Ergebnis wieder eine Relation ist.

Es ist noch anzumerken, dass eine entsprechende Methode natürlich auch für Typklassen ohne Typparameter möglich ist. Der Unterschied zu Typkonstruktorklassen ist der, dass bei letzteren die Parametrizitätsaussagen der einzelnen Klassenfunktionen bezüglich einer Relationsfunktion $\mathcal{F}$ gelten müssen, damit diese zulässig ist. Bei simplen Typklassen muss einfach nur jede einzelne Parametrizitätsaussage für sich selbst gelten.

3.4 Anwendung

In den vorangegangenen Abschnitten wurde beschrieben, wie freie Theoreme hergeleitet werden. Das Vorgehen wurde am einfachen Beispiel $f :: [a] \to [a]$ verdeutlicht. Dieses Beispiel beschreibt die grundlegende Technik zwar, die resultierende Aussage an sich ist aber relativ simpel und darüber hinaus kaum interessant. In diesem Abschnitt sollen zwei Beispiele betrachtet werden, die ein wenig komplexer und auch etwas interessanter sind. Vor allem aber verwenden sie Typkonstruktorklassen, es kann also an Beispielen gezeigt werden, wie sich die Herleitung von freien Theoremen gestaltet, wenn Typkonstruktorvariablen eine Rolle spielen.

In Haskell ist es gebräuchliche Praxis, für Datentypen oder Typklassen zusätzliche Einschränkungen zu definieren, die nicht durch die Implementierung sichergestellt werden. Stattdessen liegt es am Anwender, diese Einschränkungen einzuhalten, und es hindert ihn nichts daran, diese Gesetze zu missachten. Ein prominentes Beispiel ist die Klasse `Functor`, die die Funktion `fmap` definiert. Für `fmap` werden die folgenden Gesetze gefordert.

```
fmap id      = id
fmap (p . q) = (fmap p) . (fmap q)
```

Es wäre nun interessant zu betrachten, welche Aussagen zu dieser Funktion das freie Theorem offenbart. Tatsächlich lässt sich nach Kmett [fma] zeigen, wie man mithilfe des freien Theorems beweisen kann, dass das zweite Gesetz in Haskell immer gelten muss, wenn das erste Gesetz gilt.

Dieser Beweis soll an dieser Stelle nicht wiedergegeben werden, weil er vor allem zu weit über die Thematik von freien Theoremen hinausgeht, aber es ist sinnvoll zu zeigen, wie man das freie Theorem für die Funktion `fmap` herleitet.

Wie bereits erwähnt, hat `fmap` die folgende Typsignatur.

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Die Besonderheit an dieser Signatur ist die auftretende Typkonstruktorvariable f. Doch nicht minder interessant ist die Tatsache, dass eine Funktion als Parameter übergeben wird – dieser Fall wurde in dieser Arbeit bisher noch nicht explizit an einem Beispiel erklärt. Im Folgenden ist zu sehen, wie das Parametrizitätstheorem angewandt wird und Schritt für Schritt die jeweiligen relationalen Aktionen eingesetzt werden, wobei $(* \to *)$ die Menge aller Typkonstruktoren der Sorte $* \to *$ sei.

$$(fmap, fmap) \in \forall \mathcal{F}.\ \forall \mathcal{R}.\ \forall \mathcal{S}.\ (\mathcal{R} \to \mathcal{S}) \to \mathcal{F}\mathcal{R} \to \mathcal{F}\mathcal{S}$$

$$\begin{aligned}
\Leftrightarrow\ & \forall k1, k2 \in (* \to *), \mathcal{K} : k_1 \Leftrightarrow k_2, \mathcal{K} \text{ beachtet Functor} \\
& (fmap_{k1}, fmap_{k2}) \in \forall \mathcal{R}.\ \forall \mathcal{S}.\ (\mathcal{R} \to \mathcal{S}) \to \mathcal{K}\mathcal{R} \to \mathcal{K}\mathcal{S}
\end{aligned}$$

$$\begin{aligned}
\Leftrightarrow\ & \forall k1, k2 \in (* \to *), \mathcal{K} : k_1 \Leftrightarrow k_2, \mathcal{K} \text{ beachtet Functor} \\
& \forall A, A' \in Typen, \mathcal{A} : A \Leftrightarrow A' \\
& \forall B, B' \in Typen, \mathcal{B} : B \Leftrightarrow B' \\
& (fmap_{k1AB}, fmap_{k2A'B'}) \in (\mathcal{A} \to \mathcal{B}) \to \mathcal{K}\mathcal{A} \to \mathcal{K}\mathcal{B}
\end{aligned}$$

$$\begin{aligned}
\Leftrightarrow\ & \forall k1, k2 \in (* \to *), \mathcal{K} : k_1 \Leftrightarrow k_2, \mathcal{K} \text{ beachtet Functor} \\
& \forall A, A' \in Typen, \mathcal{A} : A \Leftrightarrow A' \\
& \forall B, B' \in Typen, \mathcal{B} : B \Leftrightarrow B' \\
& \forall (g_1, g_2) \in (\mathcal{A} \to \mathcal{B}) \\
& (fmap_{k1AB}\ g_1, fmap_{k2A'B'}\ g_2) \in \mathcal{K}\mathcal{A} \to \mathcal{K}\mathcal{B}
\end{aligned}$$

Da der erste Parameter von $fmap$ eine Funktion ist, werden auch Elemente aus dieser Funktionstyprelation allquantifiziert, d.h. $\forall (g, g') \in (\mathcal{A} \to \mathcal{B})$. Auch an dieser Stelle kann man die Definition des Funktionstypkonstruktors einsetzen, also $(g, g') \in (\mathcal{A} \to \mathcal{B})$ genau dann, wenn $\forall (x, x') \in \mathcal{A} : (g\ x, g'\ x') \in \mathcal{B})$. Die Aussage lässt sich also wie folgt weiter umformen.

$$
\begin{aligned}
&\forall k1, k2 \in (* \to *), \mathcal{K} : k_1 \Leftrightarrow k_2, \mathcal{K} \text{ beachtet Functor} \\
&\quad \forall A, A' \in Typen, \mathcal{A} : A \Leftrightarrow A' \\
&\quad\quad \forall B, B' \in Typen, \mathcal{B} : B \Leftrightarrow B' \\
&\quad\quad\quad \forall g_1 : A \to B, g_2 : A' \to B' \\
&\quad\quad\quad\quad (\forall (x, x') \in \mathcal{A}.(g_1\ x, g_2\ x') \in \mathcal{B}) \\
&\quad\quad\quad\quad\quad \Rightarrow (fmap_{k1AB}\ g_1, fmap_{k2A'B'}\ g_2) \in \mathcal{K}\mathcal{A} \to \mathcal{K}\mathcal{B}
\end{aligned}
$$

$$
\begin{aligned}
\Leftrightarrow\ &\forall k1, k2 \in (* \to *), \mathcal{K} : k_1 \Leftrightarrow k_2, \mathcal{K} \text{ beachtet Functor} \\
&\quad \forall A, A' \in Typen, \mathcal{A} : A \Leftrightarrow A' \\
&\quad\quad \forall B, B' \in Typen, \mathcal{B} : B \Leftrightarrow B' \\
&\quad\quad\quad \forall g_1 : A \to B, g_2 : A' \to B' \\
&\quad\quad\quad\quad (\forall (x, x') \in \mathcal{A}.\ (g_1\ x, g_2\ x') \in \mathcal{B}) \\
&\quad\quad\quad\quad\quad \Rightarrow \forall (y, y') \in \mathcal{K}\mathcal{A} \\
&\quad\quad\quad\quad\quad\quad (fmap_{k1AB}\ g_1\ y, fmap_{k2A'B'}\ g_2\ y') \in \mathcal{K}\mathcal{B}
\end{aligned}
$$

An dieser Stelle ist es noch möglich, die Relationen $\mathcal{A}$ und $\mathcal{B}$ auf Funktionen zu spezialisieren. Wir werden feststellen, dass dies bei auftretenden Typkonstruktorvariablen leider oftmals keine Vereinfachung bringt, da Typkonstruktorvariablen über beliebige Typkonstruktoren allquantifiziert werden und deshalb nichts Genaues über sie bekannt ist.

Im vorangegangenen Abschnitt wurde bereits erläutert, dass man bei der Spezialisierung zu einigen gelifteten Datentypen spezielle Funktionen einführen kann, um die spezialisierten Funktionen in einen entsprechenden Kontext zu liften, beispielsweise durch Verwenden der Funktion *map*, wenn eine Relation durch den Listentypkonstruktor in den Listenkontext geliftet wird.

Es wäre wünschenswert, an dieser Stelle ebenfalls eine solche Funktion einzufügen. Im betrachteten Beispiel handelt es sich um die `Functor`-Klasse, von der wir wissen, dass sie eine Funktion `fmap` enthält, die den Inhalt des entsprechenden `Functor`-Datentyps manipuliert, und man könnte argumentieren, dass sie für diesen Zweck geeignet ist – zumindest, wenn sie korrekt implementiert wurde und die `Functor`-Gesetze einhält. Da aber der allgemeine Fall betrachtet werden soll, in dem eine solche Funktion nicht unbedingt gegeben ist, beschränken wir uns hier darauf, die Relationsfunktion $\mathcal{K}$ auf die entsprechende Funktion anzuwenden. Da sich die Funktion als Relation auffassen lässt, können wir daraus auch wieder eine Relation generieren.

$$
\begin{aligned}
&\forall k1, k2 \in (* \to *), \mathcal{K} : k_1 \Leftrightarrow k_2, \mathcal{K} \text{ beachtet Functor} \\
&\quad \forall a : A \to A' \\
&\quad\quad \forall b : B \to B' \\
&\quad\quad\quad \forall g_1 : A \to B \\
&\quad\quad\quad\quad \forall g_2 : A' \to B' \\
&\quad\quad\quad\quad\quad (\forall x \in A : b\ (g_1\ x) = g_2(ax)) \\
&\quad\quad\quad\quad\quad \Rightarrow \forall y \in k_1\ A \\
&\quad\quad\quad\quad\quad\quad\quad (fmap_{k1AB}\ g_1\ y, fmap_{k2A'B'}\ g_2) \in \mathcal{K}b
\end{aligned}
$$

Es ist natürlich schade, dass die Spezialisierung auf Funktionen durch die Anwendung der Relationsfunktion gewissermaßen wieder rückgängig gemacht wird. Das macht die Spezialisierung auf Funktionen natürlich nicht hinfällig, auf der linken Seite der entstandenen Implikation bietet die Spezialisierung einen Vorteil bei der Lesbarkeit.

Somit haben wir also das freie Theorem für `fmap` hergeleitet. Es zeigt sich, dass auch mit Typkonstruktorvariablen ganz normal gearbeitet werden kann. Durch das Auftreten der Funktion als Parameter ist auch gut zu sehen, wie eine solche Funktion zu einer Implikation führt, da für Elemente der entsprechenden Relation wiederum die Relationskonstruktion aus dem Funktionstyp gilt.

Nicht zu vergessen ist, was sich hinter der Einschränkung "$\mathcal{K}$ beachtet Functor" verbirgt: Es muss Parametrizität gelten für alle Klassenfunktionen von `Functor`, wenn für die Typkonstruktorvariable $\mathcal{K}$ eingesetzt wird (wie es zu Beginn des Abschnitts bereits erläutert wurde).

In der Arbeit, in der Voigtländer die hier verwendete Interpretation von Typkonstruktorvariablen in freien Theoremen einführt [Voi09], nutzt er diese Erweiterung der freien Theoreme, um einige Überlegungen insbesondere zu Monaden anzustellen. Es soll nachfolgend eine dieser Überlegungen dargestellt werden, um ein weiteres Beispiel für den Einsatz von freien Theoremen, insbesondere mit der Erweiterung um Typkonstruktorvariablen, vorzuführen. Der folgende Code zeigt, wie die Monadenklasse definiert ist.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Tatsächlich gibt es noch weitere Funktionen `>>` und `fail`, diese spielen jedoch in diesem Fall keine Rolle und besitzen darüber hinaus Standardimplementierungen. Wie auch für Funktoren gibt es in Haskell einige – von Haskell nicht erzwungene – Gesetze, die für Monadeninstanzen gelten sollten. Diese sind wie folgt definiert.

```
return a >>= k  =  k a
m >>= return  =  m
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Ein monadischer Wert heißt *rein*, wenn er mit keinerlei monadischem Effekt versehen ist. Ein einfaches Beispiel für monadische Effekte liefert zum Beispiel die `IO`-Monade, mit deren Hilfe Seiteneffekte wie Ein- und Ausgabe umgesetzt werden. So stellt ein Ausdruck mit dem Typ `IO a` eine Berechnung dar, bei deren Ausführung es zu Ein- und Ausgaben kommen kann, aber nicht muss.

Der Ausdruck `getLine` vom Typ `IO String` ist ein Beispiel für einen solchen effektbehafteten monadischen Ausdruck, da er nicht nur eine Zeichenkette enthält, sondern auch einen Seiteneffekt, der das Einlesen dieser Zeichenkette von der Standardeingabe verursacht. Der folgende Ausdruck wiederum ist ein Beispiel für einen reinen monadischen Wert, der keinen monadischen Effekt enthält.

```
return "Zeichenkette"
```

Man kann einen reinen Wert ansehen als den Wert, der von der `return`-Funktion der jeweiligen Monadeninstanz bzw. einer semantisch äquivalenten Funktion geliefert wird. Betrachten wir die folgende Funktion.

```
f :: Monad m => [m Int] -> m Int
```

Eine Erkenntnis von Voigtländer ist nun, dass man im Fall, dass die Eingabeliste nur reine Werte beinhaltet, davon ausgehen kann, dass das Ergebnis auch rein ist, sofern die Monade das erste Monadengesetz erfüllt: $return_k\ a >>=_k k = k\ a$. Man argumentiert hier genauso wie bei polymorphen Funktionen: Da die Funktion über beliebige Monaden m definiert ist, hat die Funktion keine Kenntnis über die jeweilige Monade, die genutzt wird. Sie kann dementsprechend keine monadenspezifischen Effekte einführen; nur bereits vorhandene Effekte können gegebenenfalls beibehalten werden.

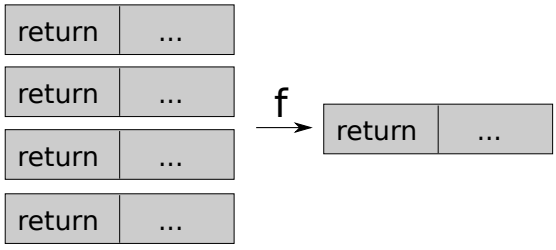

Abbildung 2: Erhaltung von reinen Ausdrücken

In einer Formel ausgedrückt heißt das, dass das folgende Theorem gilt.

Theorem 2. *Sei $f :: Monad\ \mu \Rightarrow [\mu\ Int] \to \mu\ Int$, sei κ eine Instanz von Monad, für die Monadengesetz (1) gilt. Sei $l :: [\kappa\ Int]$. Wenn jedes Element in l ein Bild von $return_\kappa$ ist, dann ist auch $f_\kappa\ l$ ein Bild von $return_\kappa$.*

Um zu zeigen, dass dieses Theorem gilt, leiten wir zunächst das freie Theorem zur Funktion f her. Da an dieser Stelle klar sein sollte, wie das Prinzip des schrittweisen Abrollens funktioniert, wird hier direkt das resultierende Theorem angegeben.

$$\forall k_1, k_2 \in (* \to *), \mathcal{M} : k_1 \Leftrightarrow k_2, \mathcal{M} \text{ beachtet Monad}$$
$$\forall (x, y) \in [\mathcal{M}\ id_{Int}]$$
$$(f_{k_1}\ x, f_{k_2}\ y) \in \mathcal{M}\ id_{Int}$$

Wir möchten nun zeigen, dass das Ergebnis von f_κ für eine beliebige Liste von $return_\kappa$-Bildern selbst ein $return_\kappa$-Bild ist. Hierzu betrachten wir eine beliebige Liste $l' :: [Int]$. Wenden wir auf jedes Listenelement von l' die Funktion $return_\kappa$ an, erhalten wir folglich eine solche Liste von $return_\kappa$-Bildern. Anders schreiben lässt sich das mithilfe der *map*-Funktion als $map\ return_\kappa\ l'$.

Nun müssen wir beweisen, dass man durch Anwenden von f_κ auf diese Liste einen Ausdruck erhält, der einem $return_\kappa$, angewandt auf einen Ausdruck, entspricht. Um dies zu tun, führen wir zunächst einen neuen Typ ein und deklarieren für diesen eine Monadeninstanz.

```
newtype Id a = Id { unId :: a }

instance Monad Id where
    return a = Id a
    Id a >>= k = k a
```

Diese Monade reicht also einfach nur den internen Wert weiter, `return` setzt diesen Wert. Sonstige Effekte sind nicht vorhanden. Es ist nun zu zeigen, dass für jede Liste $l' :: [Int]$ Folgendes gilt.

$$f_k(map\ return_\kappa\ l') = return_\kappa(unId(f_{Id}(map\ Id\ l')))$$

Dass dies die Behauptung ist, die wir beweisen wollen, ist nicht unbedingt auf den ersten Blick klar. Auf der rechten Seite der Gleichung sieht man den Ausdruck $map\ Id\ l'$. Es wird also der Konstrktor Id auf jedes Element der Liste l' angewandt. Auf den resultierenden Ausdruck wird die Funktion f_{Id} angewandt, also f, instanziiert auf den Typkonstruktor Id. Das ist möglich, da Id eine Instanz der Klasse `Monad` ist. Auf den daraus entstehenden Ausdruck wird `unId` aufgerufen, was uns wieder den internen Wert der `Id`-Monade gibt. Schließlich wird darauf die Funktion $return_\kappa$ angewandt.

Das Problem ist ja, dass wir für eine beliebige Monade κ keine Möglichkeit haben, den reinen Wert zurückzuholen. Wir können aber eine eigene Monade definieren und das freie Theorem geschickt nutzen, um eine Beziehung zwischen der κ-Monade und der Id-Monade zu erhalten.

Wir definieren eine Funktion auf Relationen $\mathcal{F} : \kappa \Leftrightarrow Id$ wie folgt.

$$\mathcal{F}\mathcal{R} = return_\kappa^{-1}; \mathcal{R}; Id$$

Jetzt zeigen wir, dass diese Funktion die Klasse `Monad` beachtet[5]. Es ist also zunächst Folgendes für die *return*-Funktion zu zeigen.

$$(return_\kappa, return_{Id}) \in \forall\mathcal{R}.\mathcal{R} \to \mathcal{F}\mathcal{R}$$

Rollt man diese Gleichung ab, erhält man die folgende Aussage, die zu zeigen ist.

$$\begin{aligned}&\forall T_1, T_2 \in Typen, \mathcal{R} : T_1 \Leftrightarrow T_2\\ &\quad\forall (a, b) \in \mathcal{R}\\ &\qquad (return_\kappa\ a, return_{Id}\ b) \in \mathcal{F}\ \mathcal{R}\end{aligned}$$

Setzt man das oben definierte $\mathcal{F}$ ein, sieht man, dass diese Aussage stimmt, da man wie folgt argumentieren kann.

Die Verkettung $return_\kappa^{-1}; \mathcal{R}; Id$ kann man ansehen als Einkapselung der Relation in die Monaden-Konstruktoren für κ und Id: Zwei Werte sind bezüglich dieser Relation verwandt, wenn es eine Möglichkeit gibt, den ersten Wert mit der inversen Relation zur Funktion $return_\kappa$ in einen puren Wert umzuwandeln und es einen zu diesem puren Wert verwandten Wert in $\mathcal{R}$ gibt, der sich mithilfe von Id wieder in einen Wert des Id-Datentyps umwandeln lässt.

Es ist $return_\kappa^{-1}\ return_\kappa\ a = a$ und $a \in \mathcal{R}$. Da Id der Konstruktor zum Datentyp Id ist, der für beliebige Typen definiert ist, ist natürlich auch $(a, Id\ a) \in Id$, wenn man den Konstruktor Id als Relation auffasst. Und da $return_{Id}$ implementiert ist mit `return a = Id a`, gilt die folgende Aussage.

$$(return_\kappa\ a, return_{Id}\ b) = (return_\kappa\ a, Id\ b) \in return_\kappa^{-1}; \mathcal{R}; Id$$

Ein wenig aufwendiger wird es, die Parametrizitätsaussage für die zweite Funktion zu zeigen.

[5] Voigtländer nennt eine Funktion, die die Klasse `Monad` beachtet, auch eine Monaden-Aktion [Voi09]

$$
\begin{aligned}
&((>>=_\kappa), (>>=_{Id})) \in \forall\mathcal{R}.\forall\mathcal{S}.\mathcal{F}\mathcal{R} \to ((\mathcal{R} \to \mathcal{F}\mathcal{S}) \to \mathcal{F}\mathcal{S}) \\
\Leftrightarrow\ &\forall T_1, T_2 \in Typen, \mathcal{R} : T_1 \Leftrightarrow T_2 \\
&\quad \forall T_3, T_4 \in Typen, \mathcal{S} : T_3 \Leftrightarrow T_4 \\
&\quad\quad \forall (a, b) \in \mathcal{F}\ \mathcal{R} \\
&\quad\quad\quad \forall (k_1, k_2) \in \mathcal{R} \to \mathcal{F}\ \mathcal{S} \\
&\quad\quad\quad\quad ((>>=)_{\kappa T_1 T_3}\ a\ k_1, (>>=)_{IdT_2T_4}\ b\ k_2) \in \mathcal{F}\mathcal{S} \\
\Leftrightarrow\ &\forall T_1, T_2 \in Typen, \mathcal{R} : T_1 \Leftrightarrow T_2 \\
&\quad \forall T_3, T_4 \in Typen, \mathcal{S} : T_3 \Leftrightarrow T_4 \\
&\quad\quad \forall (a, b) \in return_\kappa^{-1}\ ;\ \mathcal{R}\ ;\ Id \\
&\quad\quad\quad \forall (k_1, k_2) \in \mathcal{R} \to return_\kappa^{-1}\ ;\ \mathcal{S}\ ;\ Id \\
&\quad\quad\quad\quad (a\ >>=_{\kappa T_1 T_3}\ k_1, b\ >>=_{IdT_2T_4}\ k_2) \in return_\kappa^{-1}\ ;\ \mathcal{S}\ ;\ Id
\end{aligned}
$$

Wir betrachten also beliebige $\mathcal{R}$ und $(return_\kappa\ a, Id\ b) \in return_\kappa^{-1}\ ;\ \mathcal{R}\ ;\ Id$ (durch die Komposition mit $return_\kappa^{-1}$ und Id können die Elemente von $\mathcal{F}$ nur von dieser Form sein). Zudem seien $(k_1, k_2) \in \mathcal{R} \to \mathcal{F}\ \mathcal{S}$, was bedeutet, dass für alle $(x, y) \in \mathcal{R}$ gilt, dass $(k_1\ x, k_2\ y) \in return_\kappa^{-1}\ ;\ \mathcal{S}\ ;\ Id$.

Durch das erste Monadengesetz, `return a >>= k = k a`, gilt die folgende Aussage.

$$(return_\kappa\ a >>=_\kappa k_1, Id\ b >>=_{Id}\ k_2) = (k_1\ a, k_2\ b)$$

Da $(k_1\ a, k_2\ b) \in return_\kappa^{-1}\ ;\ \mathcal{S}\ ;\ Id$, ist auch die Parametrizitätsaussage für die zweite Funktion gezeigt. Somit sind beide Aussagen gezeigt und es gilt, dass $\mathcal{F}$ die Klasse `Monad` beachtet. Das heißt, dass das freie Theorem zu f für $\mathcal{F}$ gilt, sprich: $(f_\kappa, f_{Id}) \in [\mathcal{F}\ id_{Int}] \to \mathcal{F}\ id_{Int}$. Es gilt nun Folgendes.

$$
\begin{aligned}
&\mathcal{F}\ id_{Int} \\
=\ &return_\kappa^{-1}\ ;\ id_{Int}\ ;\ Id \\
=\ &return_\kappa^{-1}\ ;\ Id \\
=\ &return_\kappa^{-1}\ ;\ unId^{-1} \\
=\ &(return_\kappa \circ unId)^{-1}
\end{aligned}
$$

Das freie Theorem zu f impliziert also die folgende Aussage, da $\mathcal{F}$ die Klasse `Monad` beachtet und man die Relationen wieder zu Funktionen spezialisieren kann, und somit beweist die folgende Aussage die Behauptung.

$$
\begin{aligned}
&\forall (x, y) \in [(return_\kappa \circ unId)^{-1}] \\
&\quad (f_\kappa\ x, f_{Id}\ y) \in (return_\kappa \circ unId)^{-1} \\
\Rightarrow\ &\forall l' :: [Int] \\
&\quad return_\kappa(unId(f_{Id}\ (map\ Id\ l'))) = f_\kappa\ (map\ return_\kappa\ l')
\end{aligned}
$$

4 Die Bibliothek free-theorems

Die Generierung freier Theoreme ist geradlinig und unkompliziert. Gerade deshalb ist es jedoch mühselig, diese Arbeit jedes Mal per Hand durchzuführen, da immer die gleichen Schritte durchgeführt werden. Von daher ist es wünschenswert, diesen Prozess zu automatisieren, und genau das macht die Bibliothek *free-theorems* [Böh07]. Diese Bibliothek beinhaltet die Werkzeuge, die nötig sind, um freie Theoreme aus Haskell-Programmcode zu generieren.

In diesem Kapitel wird näher darauf eingegangen, welche Werkzeuge dies sind und wie sie zusammenspielen. Es wird erläutert, welche Schritte notwendig sind, um aus Quellcode eine Formel herzuleiten, die das entsprechende freie Theorem darstellt. Abgeschlossen wird diese Übersicht dann mit einem kleinen Beispiel, anhand dessen das Zusammenspiel der einzelnen Datentypen zwischen den verschiedenen Funktionen veranschaulicht wird.

4.1 Aufbau

Die Bibliothek lässt sich grundsätzlich in drei Teile untergliedern: Frontend, Core und Pretty Printer [Böh07], wobei die Bibliothek auch in dieser Reihenfolge durchlaufen wird, zu sehen in Abbildung 3.

Das Frontend ist der Teil, der den Haskell-Code entgegennimmt. Hier wird zunächst der Programmcode geparst und in einen abstrakten Syntaxbaum übersetzt. Außerdem ist das Frontend dafür zuständig, Fehlerprüfungen durchzuführen und ungültige Definitionen herauszufiltern, bevor der Syntaxbaum dann weiter an den Core gegeben wird.

Im Core findet die eigentliche Arbeit statt: Hier wird der Syntaxbaum in die entsprechende Relationaldarstellung überführt und in eine Zwischendarstellung abgelegt, `Intermediate` genannt. Im nächsten Schritt wird diese Relationaldarstellung dann Schritt für Schritt abgerollt und in eine Formel überführt. Diese Formel wird schließlich zurückgegeben.

Um diese Formel darzustellen, werden dann im Pretty Printer Teil entsprechende Funktionen bereitgestellt, die in der Lage sind, den Formel-Datentypen in Zeichenketten umzuwandeln.

4.2 Parser

Um zu einer in Haskell geschriebenen Funktionssignatur irgendetwas zu generieren, muss zunächst eins getan werden: Aus dem Programmcode muss die notwendige Information herausgezogen werden, der Haskell-Code muss geparst werden. *free-theorems* erfindet hier das Rad nicht neu sondern setzt auf einen Haskell-Parser, der mit `haskell-src-exts` bereits als Paket verfügbar ist [has] (bzw. `haskell-src` als Version, die keine Spracherweiterungen unterstützt).

Dieses Paket bietet einen Parser für den kompletten Sprachumfang von Haskell inklusive aller Spracherweiterungen, die GHC unterstüzt. Der Parser liefert eine eigene Datenstruktur für einen abstrakten Syntaxbaum, aber der enorme Sprachumfang hat

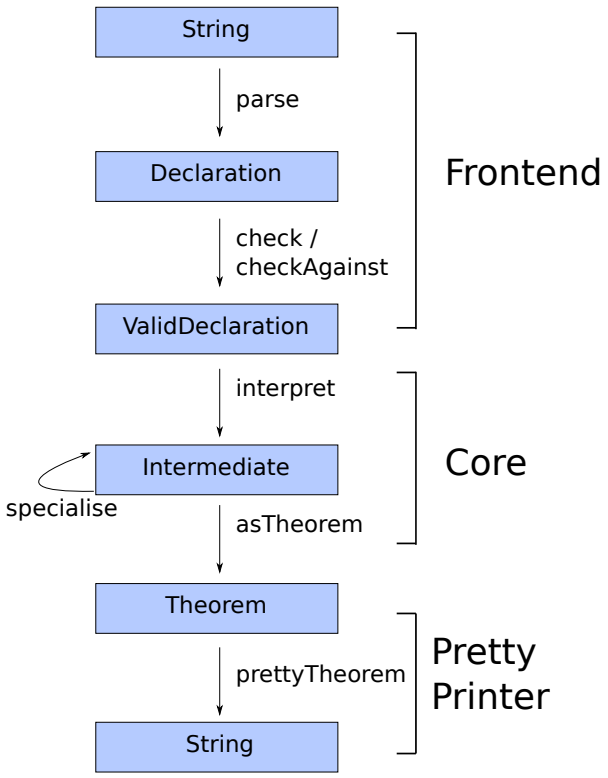

Abbildung 3: Überblick über die Bibliothek free-theorems

natürlich zur Folge, dass dieser Syntaxbaum sehr komplex werden kann. Nun werden aber gar nicht sämtliche syntaktischen Möglichkeiten der Sprache benötigt, ganz im Gegenteil: Für die Generierung freier Theoreme spielen hauptsächlich Typsignaturen eine Rolle. Die konkreten Implementierungen der Funktionen, die in einem typischen Programm den Großteil des Programms ausmachen, können getrost vernachlässigt werden.

Aus diesem Grund führt *free-theorems* eine eigene Syntaxbaum-Datenstruktur ein und überführt den Syntaxbaum, der von `haskell-src-exts` generiert wird, in eine vereinfachte Darstellung, genannt `BasicSyntax`.

Ein paar kleine Beispiele sollen ein Gefühl dafür vermitteln, wie der `BasicSyntax`-Datentyp aussieht. Dabei ist jeweils eine Funktionssignatur gegeben und dahinter die Haskell-Darstellung des resultierenden BasicSyntax-Ausdrucks.

Listing 2 zeigt die Datenstruktur am Beispiel `id :: a → a`. Die `parse`-Funktion des Parser-Moduls von *free-theorems* liefert eine Liste sämtlicher relevanter Definitionen als `BasicSyntax` zurück. In diesem Beispiel ist nur eine Typsignatur gegeben, `parse` gibt also eine einelementige Liste zurück, bestehend aus genau einem Konstruktor `TypeSig`, der eine Typsignatur repräsentiert. Der Datentyp erwartet einen `Signature`-Konstruktor, der aus dem Namen der Signatur sowie einem Typausdruck besteht.

In diesem Fall ist der Typausdruck eine Funktion zwischen zwei Typvariablen, der

```
id :: a -> a

[TypeSig
    (Signature {
       signatureName = Ident "id" ,
       signatureType =
          TypeFun
             (TypeVar (TV (Ident "a")))
             (TypeVar (TV (Ident "a")))
    )
]
```

Listing 2: Beispiel

Konstruktor `TypeFun` wird also auf zwei `TypeVar`-Konstruktoren angewandt, die jeweils die Variablennamen beinhalten.

Da man freie Theoreme aus den Typsignaturen von Funktionen herleitet, könnte man sich überlegen, dass man auf alles andere als Typsignaturen verzichten kann und der Rest des Programms irrelevant ist. Während Funktionsrümpfe, also die Implementierungen der Funktionen, tatsächlich keinerlei Relevanz für freie Theoreme haben, gibt es dennoch Deklarationen neben Typsignaturen, an denen man ebenfalls interessiert ist.

Problematisch wird es nämlich sonst, wenn der Anwender der Bibliothek in seinem Programm eigene Datentypen deklariert und diese in der zu Funktionssignatur verwendet. Neben Funktionssignaturen werden also zusätzlich auch Datentypdeklarationen benötigt. Hierzu gehören sowohl die `data`- als auch `type`- und `newtype`-Deklarationen. Listing 3 zeigt ein Beispiel für eine `data`-Deklaration und deren entsprechende Darstellung als `BasicSyntax`.

`DataDecl` ist dabei ein weiterer Konstruktor des `Declaration`-Datentyps. Er erwartet den `Data`-Konstruktor, der für einen Datentyp den Namen, die übergebenen Typvariablen und die unterschiedlichen Konstruktoren übergeben bekommt, letztere jeweils mit Namen und Typausdruck. Eine Besonderheit ist dabei, dass die Typausdrücke eingekapselt sind in die `BangTypeExpression`-Struktur. Diese bietet die Konstruktoren `Unbanged` und `Banged` und erwarten ansonsten lediglich den entsprechenden Typausdruck. Der Grund für diese Unterscheidung ist das in Haskell erlaubte Striktheitsflag.

Es ist möglich, in Datentypdeklarationen das besondere Symbol `!` vor Datentypen eines Konstruktors zu setzen. Das hat zur Folge, dass dieses Argument des Datentyps stets strikt ausgewertet wird [Jon03]. Folgender Ausdruck sei als Beispiel gegeben.

```
data CompInt = String !Int
```

Tritt nun ein Ausdruck auf, in dem `CompInt` auf Argumente angewandt wird, beispielsweise `(CompInt "Result" (7 `div` 0))`, so wird das zweite Argument, das

```
data Annotated a = Annotated String a

[DataDecl
   (Data {
      dataName =
         Ident "Annotated",
      dataVars = [
         TV (Ident "a")
      ],
      dataCons = [
         DataCon {
            dataConName = Ident "Annotated",
            dataConTypes = [
               Unbanged {
                  withoutBang =
                     TypeCon (Con (Ident "String")) []},
               Unbanged {
                  withoutBang =
                     TypeVar (TV (Ident "a"))}]}]})]
```

Listing 3: Beispiel

in der `data`-Deklaration mit einem Striktheitsflag versehen wurde, strikt ausgewertet – was in diesem Fall zu einem Fehler führen würde, der bei einer Lazy-Auswertung nicht zwingend auftreten müsste.

Für die bisher eingeführte Generierung freier Theoreme würde ein solches Striktheitsflag natürlich keine Rolle spielen, da immer nur die eigentlichen Typen von Bedeutung sind. In Kapitel 6 wird jedoch kurz darauf eingegangen, warum Striktheit dennoch eine Rolle bei freien Theoremen spielen kann.

Klassendeklarationen werden insbesondere in Kapitel 5 benötigt, wenn es darum geht, Typkonstruktorklassen einzuführen. Sie werden aber auch schon von `free-theorems` beachtet und in die `BasicSyntax` mit aufgenommen. Der Grund dafür ist, dass ja auch Typvariablen in Typsignaturen auftreten können, die auf eine bestimmte Klasse eingeschränkt sind. Die aus der Allquantifizierung resultierenden Relationen dürfen also nicht über alle Typen quantifiziert sein, sondern nur über solche, die zur entsprechenden Klasse passen.

4.3 Fehlerprüfung

An dieser Stelle liegt das Eingabeprogramm also in einem vereinfachten abstrakten Syntaxbaum vor, sofern der Parser nicht mit einem Fehler abgebrochen hat. Parserfehler können ab jetzt ausgeschlossen werden. Nun reicht es leider nicht, syntaktische Korrektheit sicherzustellen, es können auch Fehler im Eingabecode enthalten sein, die über

bloße Syntax hinausgehen und dafür sorgen, dass Deklarationen ungültig sind. Diese semantischen Fehler werden in der `check`-Funktion gesucht.

Fehlerhafte Deklarationen werden aus der Liste der Deklarationen gefiltert, und es werden entsprechende Fehlermeldungen generiert. Fehler müssen hier nicht automatisch zum Abbruch führen, es ist ja durchaus möglich, dass die Fehler in Deklarationen auftreten, die für die betrachtete Funktionssignatur keine Rolle spielen.

Es wird zwischen lokaler und globaler Fehlerprüfung unterschieden. Lokale Prüfungen werden pro Deklaration durchgeführt, ohne den globalen Kontext zu betrachten. Hier werden lediglich solche Fehler betrachtet, die sich anhand der Deklaration selbst erkennen lassen. Ein Beispiel sind `data`-Deklarationen: Hier müssen auf der rechten Seite auftretende Typvariablen auch auf der linken Seite vorkommen; einen Fehler würde beispielsweise folgender Code erzeugen, der vom Parser erst einmal fehlerfrei erkannt wird.

```
data Test = Test a
```

Die Typvariable a, die auf der rechten Seite der Gleichung verwendet wird, hat auf der linken Seite der Gleichung keine Entsprechung – es liegt also an der `check`-Funktion, diesen Fehler zu erkennen. Weitere Beispiele für lokale Prüfungen sind, dass alle Variablennamen auf der linken Seite paarweise verschieden sein müssen, primitive Datentypen dürfen nicht neu deklariert werden, usw.

Nach den lokalen Überprüfungen finden die globalen Fehlerprüfungen statt. Hier geht es um solche Prüfungen, bei denen stets das komplette Programm betrachtet werden muss, zum Beispiel um sicherzustellen, dass alle Funktionsnamen im Programm deklariert sind, pro Funktionsname nur eine Deklaration existiert, Typkonstruktoren die korrekte Anzahl an Parametern haben etc. Als Beispiel seien hier die folgenden Deklarationen gegeben.

```
calcValues :: Int -> Int -> [Int]
calcValues :: Int -> [Int]
```

Dieser Code, syntaktisch absolut korrekt, würde von `check` als fehlerhaft erkannt werden, da es zwei Signaturen mit demselben Namen `calcValues` gibt.

Lokale Überprüfungen reichen hier nicht aus, da man innerhalb einer Deklaration nicht erkennen kann, dass es an einer anderen Stelle noch eine Funktion mit diesem Namen gibt.

Eine besondere Aufgabe der globalen Überprüfung ist es auch, sämtliche Typausdrücke zu *schließen*; es ist in Haskell erlaubt, freie Typvariablen in Funktionssignaturen zu verwenden - diese werden implizit als allquantifiziert angesehen, wie bereits in der Einleitung erläutert wurde. Das Schließen eines Typausdrucks besteht nun darin, sämtliche freien Typvariablen zu beseitigen, indem eine entsprechende explizite Allquantifizierung davorgesetzt wird.

Das ist nicht unbedingt notwendig, vereinfacht aber die spätere Verarbeitung, da so im weiteren Programmverlauf davon ausgegangen werden kann, dass Typvariablen nicht mehr frei vorkommen, sondern jedes Vorkommen einer Typvariablen von einer entsprechenden Allquantifizierung umschlossen wird.

Tabelle 1 zeigt eine Übersicht aller globalen Überprüfungen. Am Anfang werden alle Deklarationen in einer Liste an die erste Prüfung übergeben, jede Prüfung reicht dann die Liste der übrig bleibenden Deklarationen an die nachfolgenden Fehlerprüfungen weiter.

Name	Prüfung
checkUnique	Alle Namen kommen nur einmal vor
checkArities	Typkonstruktoren haben korrekte Anzahl an Parametern
checkAcyclicTypeSynonyms	Typsynonyme sind azyklisch
checkAcyclicTypeClasses	Typklassen sind azyklisch
checkAllConsAndClassesDeclared	Alle Typkonstruktoren und Typklassen sind deklariert

Tabelle 1: Globale Überprüfungen

Die Prüfungen arbeiten dabei in der `Writer`-Monade, in der auftretende Fehlermeldungen protokolliert werden.

Außerdem wird jede `Declaration` in eine `ValidDeclaration` eingekapselt, die als zusätzliche Information noch ein Flag enthält, in dem festgehalten wird, ob in der Deklaration Striktheitsannotationen vorkommen. Sind alle globalen Überprüfungen abgeschlossen, werden die übrig bleibenden Deklarationen an den Core-Teil weitergereicht.

4.4 Interpretieren der Typen als Relationen

Die erste Funktion des Cores ist `interpret`. Diese Funktion *interpretiert* eine Funktionssignatur als Relation. Sie erwartet eine (geprüfte) Deklaration als Eingabe und liefert eine Relation, verpackt in eine sogenannte `Intermediate`-Struktur, die neben der eigentlichen Relation noch Zusatzinformationen beinhaltet, beispielsweise den Namen der interpretierten Funktion sowie freie Variablennamen für neu einzuführende Variablen. Diese Funktion setzt also, bezogen auf die Generierung von freien Theoremen, den Schritt um, die Funktionssignatur in die Relationsschreibweise zu überführen.

`interpret` durchläuft rekursiv die `BasicSyntax`-Struktur und erzeugt entsprechende Relationen. Tabelle 5 zeigt, welche Konstruktoren der `Relation`-Datentyp hat und aus welchen `BasicSyntax`-Ausdrücken diese generiert werden. Dabei ist zu beachten, dass `FunVar` und auch `FunAbs` nur verwendet werden, wenn Relationen zu Funktionen spezialisiert werden, was im nächsten Abschnitt von Bedeutung sein wird.

Der `BasicSyntax`-Ausdruck `TypeExp` wiederum tritt in geparsten Ausdrücken erst einmal nicht auf, er wird innerhalb von `interpret` verwendet, um die neu eingeführten Typvariablen bei Relationsabstraktionen auszudrücken. `RelFunLab` spielt nur für Ungleichungs-Theoreme eine Rolle und wird in dieser Arbeit nicht beachtet.

Wie vorangehend erwähnt, werden sämtliche freie Variablen in Typsignaturen geschlossen, was dazu führt, dass `interpret` stets zuerst auf einen `forall`-Ausdruck stoßen wird, bevor es die Typvariable selbst antrifft. Das wird ausgenutzt, indem beim Antreffen von `forall` ein Eintrag in einer Map angelegt wird, auf den dann bei Verarbeitung jedes Vorkommens der Typvariable über den Variablennamen wieder zugegriffen

BasicSyntax	Relation	Relationale Entsprechung
TypeVar	RelVar	$\mathcal{R}$
FunVar	-	f
TypeCon	RelBasic	$id_{Int}, id_{Char}, id_{[Char]} \cdots$
	RelLift	$[\mathcal{R}]$, $Maybe\ \mathcal{R}$, ...
TypeFun	RelFun	$S \to T$
	RelFunLab	$S \to T$
TypeAbs	RelAbs	$\forall R.FR$
	FunAbs	$\forall f.Ff$
TypeExp	-	

Tabelle 2: Konstruktoren des Datentyps Relation

wird.

Ansonsten wird systematisch jeder Ausdruck der `BasicSyntax` durch den entsprechenden `Relation`-Ausdruck ersetzt. Typkonstruktorausdrücke werden entweder zu einer `RelBasic`, also einer einfachen Relation auf Typmengen, oder zu `RelLift`-Ausdrücken, also *gelifteten* Relationen, je nachdem ob es sich um nullstellige oder mehrstellige Typkonstruktoren handelt.

Zusätzlich ist noch zu erwähnen, dass jeder `Relation`-Ausdruck eine `RelationInfo`-Struktur enthält. Diese enthält neben der verwendeten Teilsprache (die dementsprechend bei jedem `Relation`-Ausdruck gleich ist) die Typausdrücke der linken und der rechten Seite der Relation. Das Besondere hieran ist, dass diese Typausdrücke bei Typabstraktionen zunächst auch den entsprechenden `forall`-Ausdruck beinhalten sowie die jeweilige allquantifizierte Typvariable.

Beim Typausdruck, der in die Map geschrieben wird (und demzufolge auch bei jedem weiteren Vorkommen der allquantifizierten Relationsvariable) werden sämtliche Vorkommen dieser allquantifizierten Typvariable jedoch durch die entsprechende neu eingeführte Typvariable der Relationsabstraktion ersetzt.

Aus der `interpret`-Funktion resultiert also letztendlich ein `Relation`-Ausdruck, der im nächsten Schritt weiterverarbeitet wird.

4.5 Spezialisieren von Relationsvariablen zu Funktionen

Der Spezialisierungsschritt ist optional. Wie in den Grundlagen erläutert, kann man jede Funktion auch als Relation auffassen, Funktionen sind also spezielle Relationen. Hat man eine allgemeine Aussage über beliebige Relationen, dann trifft dieselbe Aussage auch auf alle solche Relationen zu, bei denen es sich um Funktionen handelt - man spezialisiert die Aussage einfach auf Funktionen.

Es bietet sich an, das zu tun, weil die Funktionsdarstellung übersichtlicher und intuitiver verständlich ist. Aus diesem Grund bietet *free-theorems* die Funktion `specialise` an. Diese Funktion erwartet eine Intermediate-Struktur und eine Relationsvariable und transformiert die Intermediate-Struktur, indem sie sämtliche vorkommen der Relations-

variablen durch Funktionsvariablen ersetzt.

Im zweiten Schritt führt sie die `reduceLifts`-Funktion aus, die nach Möglichkeit geliftete Datentypen vereinfacht. Wenn es sich hierbei um eine Funktion handelt, dann wird überprüft, ob es sich beim Typ um Listen oder um den spezifischen Typ `Maybe` handelt. In diesen Fällen wird der Ausdruck ersetzt durch die Funktion *map* bzw. $fmap$, angewandt auf die entsprechende Funktion. Man kann dies als das *Lifting* der Funktion in den entsprechenden Kontext auffassen.

Am einfachsten lässt sich das an einem Beispiel zeigen.

```
test :: a -> [a]
```

Man kann leicht nachvollziehen, dass diese Signatur zu folgendem freien Theorem führt.

$$\begin{aligned}&\forall\ T_1, T_2 \in Types, \mathcal{R} : T_1 \Leftrightarrow T_2\\ &\quad \forall (x, y) \in \mathcal{R}\\ &\qquad (test_{T1}\ x, test_{T2}\ y) \in [\mathcal{R}]\end{aligned}$$

Ersetzt man die Relationsvariablen jetzt durch Funktionensvariablen, ergibt sich die folgende Aussage.

$$\begin{aligned}&\forall f : T_1 \to T_2\\ &\quad \forall x \in T_1\\ &\quad lift_{[]}(f)(test_{T_1}\ x) = test_{T2}(f\ x)\end{aligned}$$

Zu beachten ist hier, dass der letzte Schritt nicht zwingend durchgeführt werden kann. Ein solches Lifting ist nur in besonderen Fällen möglich, in diesem Fall entspricht das Lifting der Funktion f in den Listenkontext ganz einfach der Haskell-Funktion `map`. Das Theorem lässt sich also auch wie folgt ausdrücken.

$$\begin{aligned}&\forall f : T_1 \to T_2\\ &\forall x \in T_1\\ &map\ f\ (test_{T_1}\ x) = test_{T2}(f\ x)\end{aligned}$$

Handelt es sich um den Typ `Maybe`, lässt sich stattdessen $fmap\ f$ verwenden.

Die `asTheorem`-Funktion, die im nächsten Abschnitt erläutert wird, arbeitet sowohl mit Relationsvariablen als auch mit Funktionsvariablen.

4.6 Generierung des Theorems

Der Schritt, der zu Beginn als "Abrollen" der Parametrizitätsaussage eingeführt wurde, wird in der Funktion `asTheorem` durchgeführt. Wie in Abschnitt 3.2 erläutert, gilt zu jeder aus einer Typsignatur hergeleiteten Relation das Parametrizitäts-Theorem der Form $(e, e) \in \mathcal{R}$ (e ist der Ausdruck, $\mathcal{R}$ ist die hergeleitete Relation).

Als Eingabe erwartet `asTheorem` sinnigerweise die Intermediate-Struktur aus den vorangegangenen Schritten. Diese wird nun rekursiv in eine Formel umgewandelt. Als Ergebnis liefert die Funktion eine `Formula`, wobei es sich um einen Datentyp handelt, der die Darstellung von Formeln ermöglicht. Man kann diesen Datentyp als eine Vorstufe zur reinen Zeichenkette ansehen, denn er stellt in keinster Weise sicher, dass die Formeln sinnvoll sind. Lediglich syntaktische Korrektheit ist durch Verwenden der Datenstruktur gegeben.

Intern ruft die Funktion eine Funktion namens `unfoldFormula` auf, die die folgende Signatur hat.

```
unfoldFormula :: Term -> Term -> Relation -> Unfolded Formula
```

Die Parameter entsprechen dabei dem Ausgangsausdruck $(x, y) \in \mathcal{R}$, wobei $\mathcal{R}$ eine (eventuell aus Relationen konstruierte) Relation ist, deren Definition angewandt werden soll. Handelt es sich bei $\mathcal{R}$ beispielsweise um eine einfache Relationsvariable, gibt `unfoldFormula` die Formel $(x, y) \in \mathcal{R}$ zurück. Handelt es sich um eine `RelBasic`, d.h. sie ist aus einem nullstelligen Typkonstruktor entstanden (z.B. `Int`), dann wird hingegen die Formel $x = y$ zurückgegeben, da in der Basistypen durch Identitätsrelationen dargestellt werden und x und y folglich nur verwandt sein können, wenn sie gleich sind.

Hinter dem Rückgabewert `Unfolded Formula` verbirgt sich noch eine Monade, mit der eventuell auftretende Fehlermeldungen durchgereicht werden können. Das interessante Ergebnis ist die resultierende Formel, die als `Formula` zurückgegeben wird.

Konkret bedeutet das also: Wird die Funktion `asTheorem` mit einer Intermediate-Struktur aufgerufen, die den Funktionsnamen `n` und die Relation `r` beinhaltet, wertet sie den folgenden Ausdruck aus.

```
unfoldFormula n n r
```

4.7 Vereinfachung der Formel

An dieser Stelle liegt die Formel im `Formula`-Datentyp vor. Bevor sie schließlich mithilfe der Pretty Printer Funktionen in eine Zeichenkette umgewandelt wird, kann optional noch die `simplify`-Funktion aufgerufen werden, die einige Vereinfachungen auf die Formel anwendet. Diese Funktion hat keine Kenntnis von der ursprünglichen Relationaldarstellung, sie vereinfacht einfach nur typische Muster.

Ein Beispiel wäre zum Beispiel das Entfernen aller unbenutztzen allquantifizierten Variablen, wie in der folgenden Formel zu sehen ist.

$$\forall v \forall x.\ x = x$$
$$\Leftrightarrow \forall x.x = x$$

Auch möglich ist zum Beispiel die folgende Vereinfachung, die im Zusammenhang mit freien Theoremen häufig angewandt werden kann.

$$\forall v.f\ v = g\ v$$
$$\Leftrightarrow f = g$$

Die Funktion `simplify` erwartet also eine `Formula` und transformiert diese. Die resultierende Formel kann dann per Pretty Printer in eine Zeichenkette umgewandelt werden.

4.8 Abrollen von Lifts und Typklassen

Schließlich bietet *free-theorems* noch die Möglichkeit, die verwendeten Datentypen und Typklassen aus einer Signatur zu extrahieren und jeweils eine Relationaldarstellung zu generieren. Wie in Abschnitt 3.2 bereits beschrieben, stellt man verwendete Datentypen, die eine freie Typvariable als Parameter erwarten, als Relationen dar. Das Vorgehen ist auch hier sehr generisch, sodass dies von einem Programm leicht durchgeführt werden kann. Betrachten wir das folgende Beispiel.

```
data MyType a = MyCons String Int a | MyOtherCons String

test :: MyType a -> a
```

In der Funktionssignatur zu `test` findet der selbstdefinierte Datentyp `MyType` Verwendung. Ihm wird die Typvariable `a` als Parameter übergeben. Im freien Theorem wird die relationale Repräsentation eines gelifteten Datentyps immer durch eine Funktion *lift* dargestellt (auch für Relationen – das unterscheidet die Bibliothek vom bisher eingeführten Verfahren). *free-theorems* bietet die Funktion `unfoldLifts`, die sämtliche Lifts aus dem Theorem extrahiert und deren Definition als Formel generiert.

Auch bei benutzerdefinierten Datentypen geht es letztlich darum, welche Ausdrücke *verwandt* sind bezüglich der Relation. Ein Datentyp hat stets verschiedene Konstruktoren. In der Relation entspricht das der Vereinigung verschiedener Teilmengen, eine Teilmenge pro Konstruktor. Die folgende Formel ergibt sich aus dem obigen Beispiel.

```
"lift{MyType}(R)
  = {(MyCons x1 x2 x3, MyCons y1 y2 y3) |
       ((x1 = y1) && (x2 = y2)) && ((x3, y3) in R)}
  u {(MyOtherCons x1, MyOtherCons y1) | x1 = y1}"
```

Es wird einfach das bekannte Prinzip weitergeführt: Ausdrücke von Basistypen sind verwandt, wenn sie gleich sind. Ausdrücke des polymorphen Typs, der als Typvariable übergeben wurde, sind genau dann verwandt, wenn sie in der zugrundeliegenden Relation verwandt sind.

Eine weitere Funktion, die *free-theorems* anbietet, ist `unfoldClasses`. Diese Funktion extrahiert die Klassen, die im Theorem eine Rolle spielen, und erstellt zu jeder Klasse eine Formel, die beschreibt, wann Relationen diese Klasse *respektieren*. Dabei wird die entsprechende Formel automatisch abgerollt. Im Folgenden wird dies an einem Beispiel verdeutlicht.

```
class TestClass t where
   testfun :: t -> t

test :: TestClass t => t -> t
```

Für dieses Beispiel erzeugt die Funktion `unfoldClasses` die folgende Formel.

```
"R respects TestClass if
  forall (x, y) in R. (testfun_{t1} x, testfun_{t2} y) in R"
```

4.9 Beispiel

In den vorangegangenen Abschnitten wurde ein kurzer Umriss der Bibliothek free-theorems gegeben, die im Folgenden um Typkonstruktorklassen erweitert werden soll. Bevor die benötigten Änderungen erläutert werden, soll hier zunächst ein kompletter Durchlauf als Beispiel gegeben werden. Es werden sämtliche Daten betrachtetet, die auf dem Weg von einer Eingabezeichnekette zur resultierenden Ausgabezeichenkette entstehen.

Wir bemühen das Beispiel aus dem vorangegangenen Kapitel. Die Typsignatur ist simpel genug, um die grundlegende Funktionsweise zu erläutern, ohne die auftretenden Datenstrukturen unnötig kompliziert zu machen.

```
f :: [a] -> [a]
```

Im ersten Schritt setzt man einen bereitgestellten Parser ein, um diesen Haskell-Code in den bibliotheksinternen, vereinfachten Syntaxbaum `BasicSyntax` zu parsen. Die `parse`-Funktion gibt genaugenommen eine Liste von `Declarations`, also Deklarationen. Dabei handelt es sich um alle Toplevel-Deklarationen, die eine Rolle spielen, also Funktionssignaturen, Datentypdeklarationen und Klassendeklarationen.

```
(TypeSig (Signature {
   signatureName = (Ident "test"),
   signatureType =
      (TypeAbs (TV (Ident "a"))
         []
```

```
            (TypeFun
                (TypeCon ...
                    [(TypeVar (TV (Ident "a")))]
                )
                (TypeCon ...
                    [(TypeVar (TV (Ident "a")))]
                )
            )
        )
    })
)
```

Hier wurde die Darstellung ein wenig vereinfacht, um das Verständnis zu erleichtern. Zu beachten ist aber, dass sämtliche Funktionsimplementierungen wegfallen. Sie spielen für die Generierung freier Theoreme keine Rolle, das bedeutet aber auch, dass *free-theorems* keine Fehlerprüfungen im Implementierungsteil durchführt. Die Bibliothek kann ohne Auftreten von Fehlern durchlaufen, selbst wenn der zu einer Funktionssignatur gehörige Code Fehler enthält.

Jetzt, da die `BasicSyntax` der Beispielfunktion vorhanden ist, muss `check` aufgerufen werden, um Fehler zu entdecken. Ein Aufruf macht aus der Liste von `Declarations` eine Liste von `ValidDeclarations`. Tatsächlich enthält diese Datenstruktur lediglich ein zusätzliches Feld `isStrictDeclaration`, das anzeigt, ob die entsprechende Deklaration strikte Elemente enthält oder von diesen abhängt.

Um genau zu sein, ist der Ergebnistyp von `check` monadisch, es werden per Writer-Monade Fehlermeldungen generiert, falls Fehler auftreten. Es ist noch anzumerken, dass `check` auch beim Auftreten von Fehlern eine Liste von Deklarationen zurückgibt. Lediglich die Deklarationen, die Fehler enthalten, werden weggelassen; das bedeutet, dass unter Umständen Theoreme generiert werden können, wenn nur Fehler auftreten, die keine Auswirkung auf das zu generierende Theorem haben.

```
[(ValidDeclaration (TypeSig ...) False)]
```

An dieser Stelle haben wir eine Liste gültiger Deklarationen, und wir haben eine Liste eventell aufgetretener Fehler. Um nun freie Theoreme zu einer Typsignatur zu generieren, wird zunächst einmal eine Typsignatur benötigt. free-theorems bietet die Funktion `filterTypeSignatures`, um Typsignaturen aus einer Liste von `ValidSignatures` herauszufiltern, was in unserem Beispiel keinen Unterschied macht, weil es lediglich aus einer Typsignatur besteht.

Jetzt können wir `interpret` verwenden, um zu unserer `ValidSignature` unter Verwendung der übrigen `Declarations` die Relationaldarstellung unsereres Datentyps zu generieren. Das Ergebnis sieht wie folgt aus.

```
(Intermediate "test" BasicSubset
    (RelAbs "R" ("t1", "t2")
```

```
        (RelFun
            (RelLift ("[t1]", "[t2]") ConList (RelVar "R"))
            (RelLift ("[t1]", "[t2]") ConList (RelVar "R"))
        )
    )
    ...
)
```

Auch dieses Beispiel ist stark vereinfacht dargestellt, beinhaltet aber die wichtigsten Elemente. Zu beachten ist vor allem, dass es sich bei den Tupeln `("[t1]", "[t2]")` nicht wirklich um Tupel von Zeichenketten handelt, tatsächlich handelt es sich um Ausdrücke des Typs `TypeExpression` der `BasicSyntax`-Struktur. Der Einfachheit halber werden diese hier jedoch als Zeichenketten dargestellt, da ihre tatsächliche Form im Folgenden auch keine wirkliche Rolle mehr spielt.

Wie man sehen kann, wird alles in den `Intermediate`-Datentyp eingerahmt. Dieser enthält den Namen des Ausdrucks, d.h. den Funktionsnamen, für den die Typsignatur überhaupt angegeben wird. Zudem ist angegeben, welche *Teilsprache* verwendet wird. In diesem Fall ist dies `BasicSubset`; dieser Wert hängt davon ab, welcher Parameter der `interpret`-Funktion übergeben wurde.

Ansonsten spiegelt die Struktur ziemlich genau den Aufbau der `TypeSig`-Struktur wider. Zur Erinnerung: An dieser Stelle wird die Typsignatur auch lediglich als Konstruktion aus verschiedenen Relationaldarstellungen zusammengesetzt. Wie diese im Einzelnen definiert sind, spielt hier noch keine Rolle.

Nun soll der Ausdruck auf Funktionen spezialisiert werden. In den bisherigen Beispielen wurde die Aussage zunächst abgerollt, dann wurde ganz am Ende die Aussage spezialisiert. Es spielt für die Korrektheit der Aussage keine Rolle, ob sie zuerst abgerollt wird, oder ob zuerst Relationsvariablen auf Funktionsvariablen spezialisiert werden und das Abrollen danach stattfindet.

Es wird also die Funktion `specialise` aufgerufen, die den Ausdruck folgendermaßen transformiert.

```
(Intermediate "test" BasicSubset
    (FunAbs "f" ("t1", "t2")
        (RelFun
            (FunVar "map_{t1}_{t2} f")
            (FunVar "map_{t1}_{t2} f")
        )
    )
    ...
)
```

Die Struktur `FunVar` enthält nicht wirklich eine Zeichenkette, sondern vielmehr einen Ausdruck des Typs `Term`, mit dem der entsprechende Ausdruck dargestellt wird.

Der Punkt ist aber, dass durch `specialise` die Relationsvariable zu einer Funktionsvariable wurde, die dann durch Lift-Reduzierung zum Ausdruck `map f` vereinfacht wurde (inklusive Annotationen für Typinstanziierungen).

An dieser Stelle wird dann `asTheorem` aufgerufen, die Funktion, die die Aussage abrollt und die Formel des Theorems erzeugt. Folgende Struktur wird zurückgegeben.

```
(ForallFunctions (TVar "f") ("t1", "t2")
   (ForallVariables (TVar "x") "[t1]"
      (Predicate
         (IsEqual
            "map_{t1}_{t2} f test_{t1} x"
            "test_{t2} map_{t1}_{t2} f x"))))
```

Nun muss lediglich noch der PrettyPrinter verwendet werden, um diese `Formula`-Struktur in eine Zeichenkette umzuwandeln, und man erhält das zu erwartende Ergebnis.

```
"forall t1,t2 in TYPES, f :: t1 -> t2.
 forall x :: [t1].
  map_{t1}_{t2} f (test_{t1} x) = test_{t2} (map_{t1}_{t2} f x)"
```

Es lassen sich noch verwendete Datentypen extrahieren und deren relationale Darstellung ermitteln, sowie verwendete Typklassen relational zu beschreiben. Dazu bietet *free-theorems* die Funktionen `unfoldLifts` sowie `unfoldClasses`. Auch diese Funktionen liefern Formeln. Da der Listentypkonstruktor verwendet wird, liefert die Funktion `unfoldLifts` eine Formel, die das Lifting in den Listenkontext beschreibt. Hier liefert *free-theorems* eine spezielle Formel, die im Falle von Listen zurückgegeben wird. Bei benutzerdefinierten Datentypen würde eine Formel entsprechend dem Datentyp generiert werden.

Ebenfalls möglich ist die Verwendung der Funktion `simplify`, die allein auf der resultierenden Formel noch Vereinfachungen durchführt. Wendet man sie auf die obige Formel an und verwendet den PrettyPrinter, ergibt sich die folgende Zeichenkette.

```
"forall t1,t2 in TYPES, f :: t1 -> t2.
 map_{t1}_{t2} f . test_{t1} = test_{t2} . map_{t1}_{t2} f"
```

5 Erweiterung um Typkonstruktorklassen

In den bisherigen Abschnitten wurden die Grundlagen zu freien Theoremen erklärt, es wurde erläutert, wie freie Theoreme aussehen, wenn Typkonstruktorklassen eine Rolle spielen, und es wurde ein Einblick in die Bibliothek *free-theorems* gegeben. Es sind jetzt alle Werkzeuge vorhanden, um die Bibliothek so zu erweitern, dass auch Typkonstruktorklassen möglich sind, deren Typparameter die Sorte $* \to *$ haben. Und natürlich muss es möglich sein, Typsignaturen zu schreiben, in denen Typvariablen auf solche Klassen beschränkt und auf Typparameter angewandt werden.

Die Gliederung dieses Kapitels orientiert sich an Kapitel 4: Die Bibliothek wird in derselben Reihenfolge durchlaufen, und zu jeder Funktion wird erläutert, inwiefern sich Funktionsweisen ändern bzw. neue Funktionalitäten hinzugefügt werden. Dabei wird dort, wo es sinnvoll ist, ein Einblick in die veränderten Datenstrukturen gegeben, die hierbei eine Rolle spielen.

5.1 Erweiterungen der BasicSyntax

Alles beginnt wieder bei der Syntax. Der Sinn des eingeführten vereinfachten Syntaxbaums `BasicSyntax` war die Einsparung nicht benötigter Sprachkonstrukte Haskells. Unter anderem ist dieser Einsparung die Applikation von Typvariablen auf Typen zum Opfer gefallen, da diese ohne die Anwesenheit von Typkonstruktorklassen keine Rolle spielte.

Der erste Schritt in der Erweiterung von *free-theorems* ist also diese syntaktische Möglichkeit in der `BasicSyntax` vorzusehen. Da *free-theorems* intern Zugriff auf einen vollständigen Haskell-Parser hat, gestaltet sich diese Änderung relativ einfach, da die erzeugte Fehlermeldung an der entsprechenden Stelle in der `parse`-Funktion einfach durch eine Transformation des abstrakten Syntaxbaums in die erweiterte `BasicSyntax` ersetzt werden muss.

Rein syntaktisch kann man Typkonstruktorvariablen als Typvariablen sehen, die wie Typkonstruktoren verwendet werden. Das soll im folgenden Beispiel verdeutlicht werden.

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Das Beispiel zeigt die Signatur der Funktion >>=, in der die freie Typvariable `m` auf die Klasse `Monad` eingeschränkt wird. Diese Klasse ist so definiert, dass sie einen Typparameter erwartet, der von der Sorte $* \to *$ ist, folglich also selbst einen Parameter erwartet. Dementsprechend kommt in der Signatur beispielsweise der Ausdruck `m a` vor, d.h. die Typkonstruktorvariable `m` wird angewandt auf die Typvariable `a`.

Es ist noch zu beachten, dass eine solche Applikation bei den Standardtypkonstruktoren bereits erlaubt ist. So sind `[a]` und `Maybe a` auch ohne eine Anpassung möglich. Das liegt daran, dass bereits der Haskell-Parser zwischen Standardkonstruktoren und Variablennamen unterscheidet – Typvariablen werden durch `TyVar` dargestellt, während für Standardkonstruktoren ein `TyCon`-Ausdruck im Syntaxbaum auftaucht.

Um also dieser neuen syntaktischen Möglichkeit Rechnung zu tragen, wird einfach die `BasicSyntax` um den Konstruktor `TypeVarApp` erweitert. Am besten wird das deutlich, wenn man betrachtet, in welchen Ausdruck das obige Beispiel nach der Anpassung in `BasicSyntax` überführt wird – siehe hierzu Listing 4.

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b

TypeSig
(Signature {
   signatureName = (Ident "(>>=)"),
   signatureType =
      (TypeAbs (TV (Ident "a")) []
         (TypeAbs (TV (Ident "b")) []
            (TypeAbs (TV (Ident "m"))
               [(TypeClass (Ident "Monad"))]
               (TypeFun
                  (TypeVarApp (TV (Ident "m"))
                     [(TypeVar (TV (Ident "a")))])
                  (TypeFun
                     (TypeFun
                        (TypeVar (TV (Ident "a")))
                        (TypeVarApp (TV (Ident "m"))
                           [(TypeVar (TV (Ident "b")))]))
                     (TypeVarApp (TV (Ident "m"))
                        [(TypeVar (TV (Ident "b")))])))))))})
```

Listing 4: Beispielsignatur von >>= in BasicSyntax-Struktur

Damit ist bereits die einzige benötigte Syntaxänderung eingeführt – Klassendeklarationen sind in *free-theorems* ohnehin schon möglich. Natürlich sollte Typvariablenapplikation in jeder Typsignatur möglich sein, unabhängig davon, ob diese Signatur eine Toplevel-Signatur ist oder ob sie sich innerhalb einer Klassendeklaration befindet.

Die einzige Änderung, die dazu nötig ist, findet in der Funktion `mkAppTyEx` statt, die für die Überführung einer Typapplikation aus dem Originalsyntaxbaum in die einfachere `BasicSyntax` zuständig ist. Diese Funktion wird immer aufgerufen, egal um welche Art von Typsignatur es sich handelt.

5.2 Erweiterung des Relations-Datentyps

Der abstrakte Syntaxbaum in der `BasicSyntax`-Struktur wird von der *interpret*-Funktion in eine relationale Darstellung überführt. Das folgende Beispiel beinhaltet zur Verdeutlichung ein explizites `forall`.

```
test :: forall a. a -> a
```

Die Typsignatur dieses Beispiels lässt sich als Relation $\forall\mathcal{R}.\mathcal{R} \to \mathcal{R}$ auffassen. Dabei ist $\forall\mathcal{R}.\mathcal{A}(\mathcal{R})$ so definiert, dass über alle Relationen aller Typmengen allquantifiziert wird (vgl. Abschnitt 3.2). So weit ist das Vorgehen bereits bekannt und auch schon in *free-theorems* implementiert.

Interessant ist nun, wie es sich verhält, wenn eine solche Typvariable auf Typausdrücke angewandt wird, wie das folgende Beispiel zeigt.

```
test :: forall f a. Functor f => f a -> f a
```

Dieses Beispiel wird überführt in $\forall^{\{Funktor\}}\mathcal{F}\forall\mathcal{R}.\mathcal{FR} \to \mathcal{FR}$, wobei $\mathcal{F}$ *keine* Typrelation ist. Es handelt sich hierbei, im Gegensatz zu $\mathcal{R}$ aus dem ersten Beispiel, um eine Fuktion, die eine Relation auf eine andere Relation abbildet (natürlich wurde bereits festgestellt, dass jede Funktion auch als Relation aufgefasst werden kann, das ist an dieser Stelle aber nicht gemeint). Es wird hier also auch nicht über Relationen allquantifiziert, sondern über entsprechende Funktionen, was bedeutet, dass es sich um eine andere Konstruktion handelt als die Allquantifizierung über einfache Typvariablen.

Aus diesem Grund wird im Relations-Datentyp `Relation` ein zusätzlicher Konstruktor eingeführt für die Typkonstruktorabstraktion, der `RelTypeConsAbs` genannt wird. Außerdem wird der Konstruktor `RelTypeConsApp` eingeführt, der benutzt wird, wenn man eine Relationsfunktion auf eine Relation anwendet, im obigen Beispiel also $\mathcal{FR}$.

Tabellen 3 und 4 zeigen die Parameter für diese neuen Datentypkonstruktoren.

RelTypeConsAbs	
`RelationInfo`	Zusätzliche Informationen zur Relation, in jedem Konstruktor enthalten.
`RelationVariable`	Funktionsvariable, die allquantifiziert wird.
`(TypeExpression, TypeExpression)`	Typkonstruktorvariablen auf der linken und rechten Seite.
`[Restriction]`	Einschränkungen auf Klassen, aber auch auf *Striktheit*, *Stetigkeit*, etc.
`Relation`	Die Relation, die die quantifizierte Variable enthält.

Tabelle 3: Parameter des Konstruktors `RelConsAbs`

RelTypeConsApp	
`RelationInfo`	Zusätzliche Informationen zur Relation, in jedem Konstruktor enthalten.
`RelationVariable`	Relationsfunktionsvariable, die auf eine Relation angewandt wird.
`Relation`	Relation, auf die die Typkonstruktorfunktion angewandt wird.

Tabelle 4: Parameterliste des Konstruktors `RelConsApp`

Die meisten Parameter sollten selbsterklärend sein. Die Bedeutung der `RelationInfo`-Struktur wurde ja bereits in Abschnitt 4.4 erläutert. Wichtig ist hierbei, dass in der `RelTypeConsAbs`-Struktur das Tupel von `TypeExpressions` keine Typvariablen sind, sondern dass es sich hierbei um Variablen für Typkonstruktoren handelt.

Mit `RelConsFunVar` wird noch ein zusätzlicher Konstruktor für den `Relation`-Datentyp eingeführt. Dieser hat die gleichen Parameter wie der `RelVar`-Datentyp und wird auch nicht in die `Intermediate`-Struktur eingebaut. Er dient als Hilfsmittel für den Interpretationsschritt, der in Abschnitt 5.4 erklärt wird.

Mit diesen Erweiterungen ist es nun möglich, die hinzukommenden relationalen Strukturen auszudrücken.

5.3 Zusätzliche Fehlerprüfungen

Versucht man in der ursprünglichen Version von *free-theorems*, Typkonstruktorvariablen zu verwenden und auf andere Typen zu applizieren, wird dies noch bei der Überführung in die `BasicSyntax` mit einer Fehlermeldung abgebrochen. Dadurch fallen natürlich ein paar Fehlerfälle weg, die erst durch Typkonstruktorvariablen auftreten können. Gestattet man die Benutzung von Typkonstruktorvariablen und -klassen, muss man auch diese Probleme beachten.

Das folgende Beispiel zeigt ein solches Problem.

```
appl :: Functor f => f a -> f
```

Hierbei handelt es sich nicht um eine korrekte Haskell-Typsignatur: Die als `Functor` eingeführte Variable `f` wird in der Signatur einmal mit einem Parameter, einmal ohne Parameter aufgerufen. Selbst wenn man die Deklaration von `Functor` außer Acht lässt, kann man deutlich sehen, dass eines der beiden Vorkommen von `f` fehlerhaft sein muss. `f` erwartet entweder einen oder gar keinen Parameter, unterschiedliche Parameterzahlen sind nicht gestattet.

Da Functor bekannt ist, kann man einfach die Deklaration dieser Klasse betrachten.

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

Anhand der Signatur der Klassenfunktion kann man nun sehen, dass die Variable `f` stets auf einen einzelnen Parameter angewandt wird, das heißt, man kann bei allen Vorkommen der Klasse `Functor` davon ausgehen, dass diese auf genau einen Parameter appliziert werden müssen. Natürlich kommt hier eine weitere Überprüfung ins Spiel: Es muss sichergestellt werden, dass die Klassenvariable – in diesem Fall `f` – in sämtlichen Klassenfunktionen mit der gleichen Anzahl an Parametern verwendet wird.

Die Überprüfung, in der sichergestellt wird, dass die Klassenvariable einer Typklasse innerhalb dieser Klasse immer mit der gleichen Anzahl an Parametern verwendet wird, ist eine lokale Überprüfung – es wird stets nur die Klassendeklaration benötigt, ein globaler Kontext muss nicht bekannt sein. Möchte man überprüfen, ob Typkonstruktorvariablen

in beliebigen Deklarationen die korrekte Anzahl an Parametern haben, dann kommt man um den globalen Kontext nicht herum, da es aus jeder Signatur heraus möglich sein muss, die Klassendeklaration einzusehen. Letzteres fällt also in die Kategorie der globalen Überprüfungen.

Eine Besonderheit ist hierbei auch, dass eine Typvariable auf mehrere Klassen gleichzeitig eingeschränkt werden kann. So ist das folgende Beispiel legitimer Haksell-Code:

```
convert :: (SomeClass1 f, SomeClass2 f) => f a -> f b
```

Es muss also sichergestellt werden, dass alle Klassen, die pro Variable angegeben sind, die gleiche Anzahl an Parametern erwarten. Dieses Problem erledigt sich allerdings automatisch, wenn man die rechte Seite auf Fehler überprüft, da im Falle unterschiedlicher Aritäten mindestens einer der beiden Ausdrücke `f a` und `f b` einen Fehler erzeugen würde.

5.4 Erweiterung der Typinterpretation

Die `interpret`-Funktion ist dafür zuständig, den abstrakten Syntaxbaum von Typsignaturen in die enstprechende `Intermediate`-Struktur zu überführen. Natürlich sind hier Änderungen notwendig, wenn `interpret` auch Typkonsturktorvariablen und deren Applikation erkennen und korrekt behandeln soll.

Es wurde ja bereits in Abschnitt 4.4 erläutert, wie die Funktion mit dem `forall`-Konstrukt umgeht: Es wird in eine Allquantifizierung über Relationen umgewandelt, also in einen `RelAbs`-Ausdruck. Da es sich hierbei nicht mehr zwingend um Relationen handelt, wenn Typkonstruktorvariablen vorkommen können, sondern auch Funktionen auf Relationen afutreten können, wird jetzt an dieser Stelle eine Fallunterscheidung gemacht.

In Abschnitt 4.4 wurde auch beschrieben, dass die Implementierung von *free-theorems* hier eine `Map` als Umgebung nutzt, in der Typvariablen den dazugehörigen Variablen-Ausdrücken zugeordnet werden.

Dieser Vorgang wird nun aufgeteilt. Es wird zunächst überprüft, ob die Typvariable, über die allquantifiziert wird, in der restlichen Typsignatur auf Parameter angewandt wird. Wird ein solches Vorkommen nicht gefunden, wird wie üblich vorgegangen und es wird eine `RelAbs`-Struktur erzeugt; zudem wird in der angesprochenen *Map* dem Namen der Typvariablen die erzeugte Relationsvariablen-Struktur `RelVar` zugeordnet.

Wird stattdessen eine Applikation der Typvariablen gefunden, wird diese im Folgenden als Typkonstruktorvariable angesehen, und es wird nicht `RelAbs`, sondern stattdessen ein `RelTypeConsAbs`-Ausdruck erzeugt. Von den Parametern her unterscheiden sich die Konstruktoren nicht.

Der `RelTypeConsAbs`-Konstruktor wird nur eingeführt, damit die entsprechende Typkonstruktorvariable nicht wie normale Typvariablen verwendet wird. Zum Beispiel macht es keinen Sinn, die Relationsfunktionen zu Typkonstruktorvariablen auf Funktionen zu spezialisieren. Auch sieht natürlich die resultierende Formel für Typkonstruktorabstraktion anders aus als die für Typabstraktion.

Ein weiterer Unterschied besteht im Ausdruck, der in die Umgebungs-Map eingetragen wird. Während es sich bei Typvariablen um `RelVar`-Ausdrücke handelt, die dann später bei jedem Auftreten der Typvariable im Syntaxbaum in die Relationaldarstellung eingesetzt wird, wird bei Typkonstruktorvariablen eine neu eingeführte `RelConsFunVar`-Struktur eingetragen. Dies dient dazu, auch beim Auftreten der Typvariablen sicherzustellen, dass es sich nicht wirklich um eine Typkonstruktorvariable handelt.

Da in `check` bereits abgefangen wird, wenn Typkonstruktorvariablen mit der falschen Zahl an Parametern aufgerufen werden, kann in `interpret` davon ausgegangen werden, dass es sich bei einem Vorkommen einer Applikation von Typvariablen automatisch um eine Typkonstruktorvariable handeln muss.

5.5 Erweiterung der Spezialisierung auf Funktionen

Durch Anwenden der Funktion *specialise* wird die Formel des Theorems teilweise stark vereinfacht, da relationale Aussagen der Form $(x, y) \in \mathcal{R}$ in Gleichungen der Form $f\ x = y$ überführt werden. Es wäre wünschenswert, diese Vereinfachungen auch für Ausdrücke mit Typkonstruktorvariablen nutzbar zu machen.

In Abschnitt 4.5 wurde bereits beschrieben, wie *specialise* in *free-theorems* bisher arbeitete: Im ersten Schritt werden alle Vorkommen der zu spezialisierenden Relationsvariable durch eine Funktionsvariable ersetzt, im zweiten Schritt werden geliftete Relationen dort, wo es möglich ist, durch spezielle Funktionen wie *map* und $fmap$ ersetzt. Es liegt die Vorgehensweise nahe, die bisherige Methode auch auf beliebige Typkonstruktorvariablen auszuweiten. Hier tritt nur leider das Problem auf, das bereits in Abschnitt 3.4 angedeutet wurde: Man hat bei Allquantifizierung über Typkonstruktoren keine Informationen über die konkret verwendete Datenstruktur. Alles, was bekannt ist, ist die verwendete Typklasse, die die Existenz gewisser Funktionen voraussetzt.

Doch selbst wenn diese Klassenfunktionen gegeben sind, kann man keine Aussage darüber treffen, was diese Funktionen tun, da Typklassen Ad-Hoc-Polymorphismus nutzen: Über die tatsächliche Implementierung ist nichts bekannt. Das heißt, dass man selbst im Falle, dass die Relationsfunktionsvariable $\mathcal{F}$ auf Instanzen der Typklasse `Functor` eingeschränkt ist, nicht darauf schließen kann, dass ein Ausdruck der Art $(x, y) \in \mathcal{F}\ \mathcal{R}$ auf den Ausdruck $fmap\ f\ x = y$ spezialisiert werden kann.

Eine erste Implementierung sah die Verwendung unterschiedlicher spezieller Funktionen vor, abhängig von der Typklasse der Typkonstruktorvariable. Das folgende Beispiel zeigt das freie Theorem für die Typsignatur `Functor f => a -> f a`.

$$\begin{aligned}
&\forall k_1, k_2 \in (* \to *), \mathcal{K} : k_1 \Leftrightarrow k_2, \mathcal{K} \text{ beachtet Functor} \\
&\quad \forall T_1, T_2 \in Typen, \mathcal{R} : T_1 \Leftrightarrow T_2 \\
&\qquad \forall (x, y) \in \mathcal{R}.(test_{t_1\ k_1}\ x, test_{t_2\ k_2}\ y) \in \mathcal{K}\ \mathcal{R}
\end{aligned}$$

Spezialisiert man in diesem Beispiel die Relation $\mathcal{R}$ auf eine Funktion f, kann man auf die Idee kommen, dieses Theorem wie folgt zu transformieren.

$$\forall k_1, k_2 \in (* \to *), \mathcal{K} : k_1 \Leftrightarrow k_2, \mathcal{K} \text{ beachtet Functor}$$
$$\forall T_1, T_2 \in Typen, f : T_1 \to T_2$$
$$\forall x \in T_1.\text{fmap } f\ (test_{t_1 k_1}\ x) = test_{t_2 k_2}(f\ x)$$

Aus den oben genannten Gründen muss dies aber nicht immer gelten. Die jeweilige `fmap`-Funktion könnte beispielsweise die Reihenfolge der Daten ändern, die übergebene Funktion ignorieren, etc. Aus diesem Grund wird die Zeile transformiert wie folgt:

$$\forall x \in t_1.(test_{t_1 k_1}\ x, test_{t_2 k_2}\ (f\ x)) \in \mathcal{K}\ f$$

Da Funktionen spezielle Relationen sind, kann man umgekehrt aus der Funktion f wiederum eine Relation erzeugen, indem man die Relationsfunktion $\mathcal{K}$ auf f anwendet. Es ist natürlich unglücklich, dass dadurch die Spezialisierung der Relation prinzipiell wieder "umgekehrt" wird und wieder nur eine Aussage über Relationen getroffen wird, andererseits kann man nicht davon ausgehen, dass jede Funktion, die durch die Relationsfunktion transformiert wird, wieder eine Funktion ist; man muss also bei der Relationaldarstellung bleiben.

5.6 Erweiterung der Theoremgenerierung

Die Funktion `asTheorem` wandelt die `Intermediate`-Darstellung um in eine Formel. Natürlich sind auch hier Anpassungen vonnöten, um auf vorkommende Typkonstruktorvariablen zu reagieren. Der Datentyp, mit dem Formeln dargestellt werden, ist `Formula`. Auch dieser muss um einen Konstruktor für die Allquantifizierung über Typkonstruktorvariablen erweitert werden, `ForallTypeConstructors` genannt. Dieser Konstruktor ist definiert wie der Konstruktor `ForallRelations`, nur wird er verwendet für die Allquantifizierung über Relationsfunktionen.

Es gibt zwei neue Fälle zu beachten, was die Transformation von Relationen in Formeln angeht: Zum einen müssen die entsprechenden Relationsabstraktionen in die neue `ForallTypeConstructors`-Struktur überführt werden, zum anderen müssen die Vorkommen von `RelTypeConsApp` behandelt werden.

$$\forall^{\{Functor\}} \mathcal{F} \forall \mathcal{R}.\mathcal{R} \to \mathcal{F}\mathcal{R}$$

Der Ausdruck $\mathcal{F}\mathcal{R}$ aus diesem Beispiel wird als `Relation`-Struktur dargestellt durch den folgenden (hier leicht vereinfachten) Haskell-Ausdruck.

```
(RelTypeConsApp ("k1 t1", "k2 t2") "F" (RelVar "R"))
```

Ähnlich wie Datentypen, die auf eine Typvariable angewandt werden, wird auch die Typkonstruktorvariablenapplikation behandelt. Zur Erinnerung: Datentypkonstruktoren, angewandt auf allquantifizierte Typvariablen, werden durch eine *lift*-Funktion als

Relation ausgedrückt, deren Definition durch die `unfoldLifts`-Funktion erzeugt werden kann.

Bei Typkonstruktorvariablen wird eine solche explizite Lifting-Funktion nicht benötigt, da sie nicht als Relationen repräsentiert werden, sondern als Funktionen auf Relationen. Da keine weiteren Aussagen zu einem Ausdruck wie $\mathcal{F}\ \mathcal{R}$ gemacht werden können, wird dieser Ausdruck bei Anwendung von `asTheorem` beibehalten.

Da die `specialise`-Funktion in Bezug auf Typkonstruktorabstraktion und -applikation keine Auswirkungen hat, wie im vorangegangenen Abschnitt erläutert wurde, hat dies auch keine Auswirkungen auf die `asTheorem`-Funktion.

5.7 Beispieldurchlauf

Es wurden nun sämtliche Modifikationen an der Bibliothek erläutert, doch um das Zusammenspiel der einzelnen Änderungen einmal im Zusammenhang zu sehen, wird in diesem Abschnitt ein kompletter Durchlauf für eine Signatur dargestellt, die eine Typkonstruktorvariable beinhaltet.

Wir verwenden das Beispiel, das bereits in Abschnitt 3.3 betrachtet wurde.

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Dieser Ausdruck wird durch `parse` in die `BasicSyntax`-Struktur überführt und es ergibt sich, vereinfacht dargestellt, der Ausdruck in Listing 5.

Man kann den neu hinzgekommenen Konstruktor `TypeVarApp` sehen, der eine Liste von Ausdrücken erwartet, auf die die entsprechende Typvariable angewandt wird. Zu beachten ist auch, dass auf Syntax-Ebene sowohl Typvariablen als auch Typkonstruktorvariablen über denselben Ausdruck `TypeAbs` allquantifiziert werden. Unterschieden wird hier erst im nächsten Schritt.

Die Liste, die der `TypeAbs`-Konstruktor als zweiten Parameter erwartet, ist die Liste der Einschränkungen – auf Syntaxebene sind dies die im Kontext angegebenen Typklassen, in diesem Beispiel also die Klasse `Functor`.

Der nächste Schritt, der Aufruf der `check`-Funktion, soll nur am Rande erwähnt bleiben, da die entsprechenden Fehlerüberprüfungen in Abschnitt 5.3 bereits erläutert wurden und in unserem Beispiel kein Fehler zu erwarten ist – zudem ändert sich die Datenstruktur dadurch kaum.

Wichtig ist aber zu erwähnen, dass die `check`-Funktion (bzw. die `checkAgainst`) natürlich sicherstellt, dass die `Functor`-Klasse deklariert ist. In unserem Beispiel ist die eigentliche Deklaration der Klasse nicht enthalten. Natürlich muss die `check`- bzw später auch die `interpret`-Funktion wissen, welche vordefinierten Funktionen, Datentypen und Klassen gegeben sind. In der Implementierung des Webinterface [fre] werden zum Beispiel bereits geparste `BasicSyntax`-Ausdrücke vieler Deklarationen der Prelude aus einem separaten Modul importiert und den Funktionen `checkAgainst` und `interpret` zusätzlich zur Hauptsignatur übergeben, sodass sie dieser bekannt sind. Außerdem wird ein Eingabefeld angezeigt, in dem es möglich ist, zusätzliche Deklarationen anzugeben.

```
(TypeSig
   (Signature {
      signatureName = (Ident "fmap"),
      signatureType =
         (TypeAbs (TV (Ident "a"))
            []
            (TypeAbs (TV (Ident "b"))
               []
               (TypeAbs (TV (Ident "f"))
                  [(TypeClass (Ident "Functor"))]
                  (TypeFun
                     (TypeFun
                        (TypeVar (TV (Ident "a")))
                        (TypeVar (TV (Ident "b")))
                     )
                     (TypeFun
                        (TypeVarApp (TV (Ident "f"))
                         [(TypeVar (TV (Ident "a")))])
                        (TypeVarApp (TV (Ident "f"))
                        [(TypeVar (TV (Ident "b")))]))
                     ))))}))
```

Listing 5: BasicSyntax-Syntaxbaum für das fmap-Beispiel

Wir gehen im Folgenden davon aus, dass die typische Deklaration der `Functor`-Klasse gegeben ist, wie sie in dieser Arbeit auch schon mehrfach verwendet wurde. Der nächste Schritt besteht wieder darin, die Funktion *interpret* aufzurufen. Diese Funktion überführt die `BasicSyntax` nun also in die `Intermediate`-Darstellung, die vereinfacht in Listing 6 zu sehen ist.

Dieser Schritt ist wieder relativ selbsterklärend, da lediglich jede Typkonstruktion in die entsprechende Relationskonstruktion überführt wird. Die `RelTypeConsApp`-Ausdrücke entsprechen dabei der Applikation einer Relationsfunktion auf eine Relation. Die Typen sind dabei die Typkonstruktoren, angewandt auf die Typvariablen.

An dieser Stelle liegt also eine `Intermediate`-Struktur vor, die die Relationaldarstellung der Typsignatur enthält. Als nächstes soll diese Relationaldarstellung spezialisiert werden auf Funktionen, es wird als zunächst durch einen Aufruf der Funktion `relationVariables` die Liste aller Relationsvariablen ermittelt, für jede dieser Variablen wird dann `specialise` aufgerfen.

Die `Intermediate`-Struktur sieht daraufhin so aus wie in Listing 7.

Wie man sieht, werden sämtliche Vorkommen von Relationsvariablen durch entsprechende Funktionsvariablen ersetzt, auch die Variablen, die innerhalb der `RelTypeConsApp`-Struktur stehen. An den Applikationsausdrücken selbst hat sich nichts geändert. Zu be-

```
(Intermediate "fmap" BasicSubset
   (RelAbs "R" ("t1", "t2")
      (RelAbs "S" ("t3", "t4")
         (RelTypeConsAbs ("k1", "k2")
            (RelFun
               (RelFun
                  (RelVar "R")
                  (RelVar "S"))
                  (RelFun
                     (RelTypeConsApp ("k1 t1", "k2 t2") "R1" (RelVar "R"))
                     (RelTypeConsApp ("k1 t3", "k2 t4") "R1" (RelVar "S"))
                  ))))) ...)
```

Listing 6: Intermediate-Darstellung des Beispiels

achten ist noch, dass die als Tupel dargestellten Zeichenketten in `RelTypeConsApp` nur der Einfachheit in dieser Form zu sehen sind – tatsächlich handelt es sich jeweils um `RelationInfo`-Ausdrücke, die allgemein in jedem Ausdruck des `Relation`-Datentyps vorkommen und den jeweils dargestellten Typ beinhalten.

An dieser Stelle kann also die Umwandlung in die Formel des freine Theorems stattfinden, wozu die Funktion `asTheorem` bemüht wird. Diese Funktion führt nun das Abrollen der jeweiligen Relationsdefinitionen durch und erzeugt eine entsprechende `Formula`-Struktur. Auch diese Zwischendarstellung soll an dieser Stelle nicht noch einmal thematisiert werden, das Interessante ist die durch den Pretty Printer resultierende Zeichenkette, die wie folgt aussieht.

```
"forall t1,t2 in TYPES, f :: t1 -> t2.
 forall t3,t4 in TYPES, g :: t3 -> t4.
  forall k1,k2 in (* -> *), R1 : k1 <=> k2, R1 respects Functor.
   forall p :: t1 -> t3.
    forall q :: t2 -> t4.
     (forall x :: t1. g (p x) = q (f x))
     ==> (forall (y, z) in R1 f.
           (fmap_{t1}_{t3}_{k1} p y, fmap_{t2}_{t4}_{k2} q z) in R1 g)"
```

Und es zeigt sich, dass diese Zeichenkette dem in Abschnitt 3.4 per Hand hergeleiteten freien Theorem zu `fmap` entspricht.

```
(Intermediate "fmap" BasicSubset
   (FunAbs "f" ("t1", "t2")
      (FunAbs "g" ("t3", "t4")
         (RelTypeConsAbs ("k1", "k2")
            (RelFun
               (RelFun
                  (FunVar "f")
                  (FunVar "g"))
               (RelFun
                  (RelTypeConsApp ("k1 t1", "k2 t2") "R1"
                     (FunVar "f"))
                  (RelTypeConsApp ("k1 t3", "k2 t4") "R1"
                     (FunVar "g")))))))) ...)
```

Listing 7: Spezialisierte Intermediate-Darstellung des Beispiels

6 Striktheit und Fixpunktoperator

Bisher wurde eine Wahrheit größtenteils ignoriert: Die generierten freien Theoreme sind nur unter ganz bestimmten Bedingungen gültig. Beim Beispiel $f :: [a] \to [a]$ wurde behauptet, dass die Funktion `f` lediglich Ausdrücke aus der Eingabeliste in der Ausgabeliste weiderverwenden könne. Tatsächlich entspricht das nicht ganz der Wahrheit, denn in Haskell ist zum Beispiel auch denkbar, dass $\perp$ zurückgegeben wird, beispielsweise durch Nichtterminierung oder einen Laufzeitfehler.

free-theorems definiert verschiedene Untermengen der Programmiersprache Haskell, zu denen sich unterschiedlich komplexe Theoreme generieren lassen. Ein Faktor, der nämlich Probleme bereiten kann, ist der Fixpunktoperator, ein anderer der Striktheitsoperator. Die Bilbiothek definiert drei Untermengen der Sprache Haskell:

- BasicSubset
- SubsetWithFix
- SubsetWithSeq

Die erste Untermenge, BasicSubset, entspricht nach Böhme [Böh07] dem Girard-Reynoldschen Lambda-Kalkül, erweitert um Datentypen und Typklassen. Insbesondere sind aber keine undefinierten Werte erlaubt, die zum Beispiel durch Nichtterminierung oder Pattern-Matching-Fehler auftreten. Das ändert sich in `SubsetWithFix`. Hier ist zusätzlich der Fixpunktoperator *fix* verfügbar, und es müssen folglich undefinierte Werte erlaubt werden. Was noch fehlt, ist die `seq`-Funktion sowie Striktheits-Flags.

Die Untermenge SubsetWithSeq ist die Sprache, die auch `seq` und Striktheitsannotationen enthält. Es wird noch eine weitere Unterscheidung eingeführt: Zu `SubsetWithFix` und `SubsetWithSeq` kann noch unterschieden werden, ob die Theoreme als Gleichungen oder als Ungleichungen erwünscht sind.

Im Folgenden soll kurz auf die Problematik eingegangen werden, die durch die Existenz dieser Operatoren entsteht. Es wird angedeutet, inwiefern man die konstruierten Relationen einschränken muss, damit die generierten freien Theoreme Gültigkeit behalten, allerdings würde ein tieferes Einsteigen in diese Thematik an dieser Stelle zu weit führen.

6.1 Der Fixpunktoperator fix

Die Funktion `fix` hat die folgende Signatur.

```
fix :: (a -> a) -> a
```

Auch die Auswirkungen von `fix` auf freie Theoreme werden von Wadler [Wad89] behandelt. Der `fix`-Operator gibt den kleinsten Fixpunkt der übergebenen Funktion f zurück, sprich: Das kleinste (am wenigsten definierte) x, für das gilt $f(x) = x$. Bei x handelt es sich dabei typischerweise selbst um eine Funktion, d.h. f ist eine Funktion auf Funktionen.

Als Beispiel sei hier die Fakultätsfunktion `fak` gegeben, die wie folgt mithilfe des Fixpunktoperators definiert werden kann.

```
fak :: Int -> Int
fak = fix h
   where h :: (Int -> Int) -> (Int -> Int)
         h f = \x -> if x == 0 then 1
                                else x * f (x - 1)
```

`fak` ist also definiert als der kleinste Fixpunkt der Funktion h, die eine beliebige Funktion f abbildet auf eine neue Funktion, die für $x = 0$ einfach 1 zurückgibt und ansonsten die übergebene Funktion f aufruft mit dem Argument $x-1$ und deren Ergebnis mit x multipliziert. Der gesuchte Fixpunkt von h ist folglich eine solche Funktion, die durch erneute Anwendung von h nicht mehr verändert wird.

Letztlich ist h also eine Funktion, die als Parameter eine Funktion übergeben bekommt, die den nächsten Rekursionsschritt darstellt; der Fixpunktoperator ist somit die semantische Grundlage für Rekursion.

Voigtländer [Voi09] argumentiert nun folgendermaßen: Rekursion hat zur Folge, dass polymorphe Funktionen nun zusätzlich die Möglichkeit haben, eine Endlosrekursion und damit $\perp$ einzuführen. Wir kommen abermals zurück auf das Beispiel $f :: [a] \to [a]$. Auch dieses $\perp$-Element kann nicht zufällig entstehen: Die Funktion ist in ihren Entscheidungen nach wie vor auf die Listenlänge der Eingabeliste beschränkt, das heißt selbst wenn $\perp$ auftritt, wird es für zwei gleich lange Listen l und l' immer an den gleichen Stellen auftreten.

Für eine beliebige Funktion g auf dem konkreten Listentyp der Funktion f heißt das für eine beliebige Liste l, dass die folgende Aussage gilt, *wenn* g jedes $\perp$ wieder auf $\perp$ abbildet, sprich: strikt ist.

$$f\ (map\ g\ l) = map\ g\ (f\ l)$$

Dementsprechend müssen nach Wadler [Wad89] alle relationalen Interpretationen von Typen *zulässig* (sprich: *strikt* und *stetig*) sein, damit freie Theoreme ihre Gültigkeit behalten.

Nach Voigtländer [Voi09] heißt dies für Typkonstruktorvariablen, dass allquantifizierte Relationsfunktionen zusätzlich Striktheit bewahren müssen.

6.2 Der Striktheitsoperator seq

Deutlich einschränkender ist es leider, die Gültigkeit von freien Theorem zu erhalten, wenn der Striktheitsoperator `seq` vorhanden ist. Dieser hat die folgende Typsignatur.

```
seq :: a -> b -> b
```

Wie zu sehen ist, erwartet die Funktion zwei Parameter. Obwohl sie auf den ersten Blick ganz gewöhnlich scheint, hat sie eine bemerkenswerte Eigenschaft, die weitreichende Folgen hat. Der erste Parameter dieser Funktion wird strikt ausgewertet, anschließend wird der zweite Ausdruck zurückgegeben. Zur Folge hat das, dass der erste Parameter definitiv auf die Kopf-Normal-Form reduziert wird, unabhängig davon, ob er im zweiten Parameter verwendet wird oder nicht.

Handelt es sich beim ersten Parameter also um $\bot$, wird auch der `seq`-Aufruf zu $\bot$ ausgewertet, selbst wenn dieser Ausdruck im durchgereichten Parameter gar keine Verwendung findet.

In Abschnitt 4.2 wurde bereits erwähnt, dass es für Datentypdeklarationen eine spezielle Syntax gibt, um einzelne Parameter als *strikt* zu deklarieren. Das folgende Beispiel zeigt eine solche Striktheitsannotation.

```
data Annotated = String !SomeData
```

Die Annotation im Beispiel bewirkt, dass der Parameter vom Datentyp `SomeData` stets sofort ausgewertet wird und folglich auf Kopf-Normal-Form reduziert wird, statt wie üblich Lazy Evaluation anzuwenden.

Tritt `seq` auf, so müssen konstruierte Relationen zulässig und bottom-reflexiv sein, um die Gültigkeit der entsprechenden freien Theoreme zu erhalten.

Diese sehr strenge Einschränkung lässt sich aber zum Teil wieder auflösen, wenn man das resultierende Theorem als Ungleichung darstellt statt als Gleichung. Die Idee hinter der Darstellung als Ungleichungen, beschrieben von Johann und Voigtländer [JV04], ist die folgende: Durch Einführen von `seq` bricht die Gültigkeit der freien Theoreme auseinander, weil es passieren kann, dass eine Seite der resultierenden Gleichung $\bot$ wird. Man kann die Aussage jedoch als Ungleichung darstellen, indem man die Typrelationen asymmetrisch konstruiert; sie ordnen dann nicht mehr jeweils zwei Ausdrücke als "gleichwertig" einander zu, sondern legen eine Ordnung fest, die die *Definiertheit* widerspiegelt [JV04]. So reicht es, für Relationen zu fordern, dass sie zulässig, total und linksabgeschlossen sind.

Es soll an dieser Stelle nicht weiter darauf eingegangen werden, wie diese Ungleichungen hergeleitet werden, da dies zu weit über die eigentliche Thematik hinausgehen würde. Es sei aber erwähnt, dass die Bibliothek *free-theorems* sie bereits unterstützt, die Typkonstruktorvariablenerweiterung jedoch bisher nur für die `BasicSubset`-Sprache verfügbar ist.

6.3 Übersicht

Insgesamt ergeben sich insgesamt fünf Varianten, für die Böhme [Böh07] die folgenden Schreibweisen einführt, wobei M die Menge der verfügbaren Teilsprachen ist.

$$M = \{(basic, =), (fix, =), (fix, \sqsubseteq), (seq, =), (seq, \sqsubseteq)\}$$

Je nach gewählter Teilsprache kommen unterschiedliche Relationen bei Allquantifizierungen über Relationen infrage. Tabelle 5 listet die unterschiedlichen Relationen auf, wobei die jeweiligen Relationen stets auf den Typen T_1, T_2 definiert sind.

Bezeichnung	Beschreibung
$Rel^{(fix,=)}(T_1, T_2)$	Menge aller zulässigen Relationen
$Rel^{(fix,\sqsubseteq)}(T_1, T_2)$	Menge aller zulässigen und linksabgeschlossenen Relationen
$Rel^{(seq,=)}(T_1, T_2)$	Menge aller zulässigen und bottom-reflexiven Relationen
$Rel^{(seq,\sqsubseteq)}(T_1, T_2)$	Menge der zulässigen, totalen und linksabgeschlossenen Relationen

Tabelle 5: Relationen für die verschiedenen Teilsprachen

Natürlich müssen auch die relationalen Aktionen dementsprechend angepasst werden. So wird beispielsweise die relationale Aktion zum Funktionstypkonstruktor in der Teilsprache `SubsetWithSeq` folgendermaßen gebildet.

$$\mathcal{R} \rightarrow^{seq} \mathcal{S} = \{\ (f, g) \mid (f \neq \bot \Rightarrow g \neq \bot) \wedge \forall (x, y) \in \mathcal{R}.(f\ x, g\ y) \in \mathcal{S}\}$$

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine bereits existierende Haskell-Bibliothek zur automatischen Generierung freier Theoreme beschrieben und eine Erweiterung dieser Bibliothek zusammengefasst, mit der es möglich ist, freie Theoreme auch im Zusammenhang mit Typkonstruktorklassen zu generieren.

In Kapitel 3 wurde vorgestellt, wie nach Wadler [Wad89] die Interpretation von Typen als Typrelationen vonstatten geht und wie sich daraus wertvolle Theoreme ableiten lassen. Insbesondere wurde die Herleitung von Theoremen in der Anwesenheit von Typkonstruktorklassen und Typkonstruktorvariablen erläutert, wie sie von Voigtländer [Voi09] beschrieben wird.

Schließlich wurde die Bibliothek *free-theorems* [Böh07] beschrieben und anhand einiger Beispieldaten veranschaulicht. Die Technik zur Herleitung von Theoremen zu Typkonstruktorklassen wurde schließlich verwendet, um der beschriebenen Bibliothek die entsprechende Funktionalität hinzuzufügen.

Dazu war es notwendig, die internen Datenstrukturen so zu erweitern, dass sie die hinzukommenden syntaktischen und formalen Ausdrücke speichern können. Da die Bibliothek auf etwas älteren Modulen aufbaute, wurden hierbei auch Kompatibilitätsprobleme beseitigt, die ein Kompilieren der Bibliothek mit neueren Versionen des verwendeten GHC verhinderte.

Und natürlich wurde an einigen Stellen die Funktionalität der Bibliothek verändert, um die Theoremgenerierung auf Typkonstruktorvariablen auszuweiten. So wurden beispielsweise neue abzufangende Fehlerfälle eingeführt, die Überführung der Syntax in entsprechende relationale Strukturen wurde erweitert und das *Abrollen* der Parametrizitätsaussage musste an die erweiterte Vorgehensweise angepasst werden.

Schließlich wurde kurz darauf eingegangen, inwiefern der Fixpunktoperator und Striktheitsoperator dafür sorgen, dass freie Theorome in der Tat nicht ganz so simpel sind, wie sie im Großteil dieser Arbeit dargestellt werden. Kapitel 6 hat diese Probleme erläutert und hat Lösungen von Wadler, Voigtländer und Johann angedeutet, indem nämlich die Sprache Haskell in Teilsprachen aufgeteilt wird, wobei in jeder Teilsprache unterschiedlich starke Einschränkungen an die konstruierten Relationen gestellt werden.

Durch die Erweiterung der Bibliothek auf Typkonstruktorklassen ist es nun also möglich, freie Theoreme für einen breiteren Rahmen an Programmen zu generieren, da man auf dieses wichtige Sprachelement nicht länger verzichten muss.

7.1 Ausblick

Dass in der Erweiterung der Bibliothek bisher nur die Sprache `BasicSubset` unterstützt wird, in der weder der Fixpunktoperator noch Striktheit vorgesehen sind, ist eine starke Einschränkung, die die Anwendung für realitätsnahe Haskellprogramme deutlich erschwert. Da es sich bei diesen Sprachelementen um praktisch unverzichtbare Konstrukte in Haskell handelt, wäre es wünschenswert, die Implementierung dahingehend zu erweitern, dass auch die Teilsprachen `SubsetWithFix` und `SubsetWithSeq` unterstützt werden.

Darüber hinaus ist in der umgesetzten Implementierung nur die Typsorte $* \to *$ vorgesehen. Aktuelle Spracherweiterungen des GHC Compilers ermöglichen hingegen sogar benutzerdefinierte Typsorten [YWC+12]. Auch hier wäre noch Raum für Verbesserungen.

Die Theorie hinter der Erweiterung um Typkonstruktorklassen wendet dasselbe Vorgehen an wie Voigtländer [Voi09]. Es existieren weitere Ansätze, um freie Theoreme zu höheren Sorten zu generieren, so zum Beispiel von Aktey, der ein Modell aufstellt, in dem Sorten als reflexive Graphen dargestellt werden [Atk12].

Wirklich komfortabel wird der Umgang mit der Bibliothek *free-theorems* erst, wenn ein angenehmes Benutzerinterface verfügbar ist. Bei der Entstehung dieser Arbeit wurde beispielsweise ein kleines Tool geschrieben, das Zugriff auf die Funktionalitäten der Bibliothek bietet. Es verrichtet seine Arbeit, bietet aber keinen echten Komfort beim Erzeugen der Theoreme. Zudem läuft es lediglich in der Kommandozeile.

Angenehmer stellt sich die Verwendung dar, wenn beispielsweise ein Webinterface wie das Paket *free-theorems-webui* [fre] verwendet wird. Nicht nur wird die Eingabe dadurch intuitiver und einfacher, es ist auch eine übersichtlichere Darstellung des resultierenden Theorems möglich.

free-theorems-webui importiert dabei eine Datei vordefinierter Deklarationen, sodass bekannte Typen aus der Prelude verwendet werden können und man Funktionssignaturen zu bekannten Funktionen direkt über deren Namen angeben kann. Bei der Umsetzung der vorliegenden Arbeit wurden Deklarationen für `Functor` und `Monad` zu dieser Datei hinzugefügt, um diese für Tests zur Verfügung zu haben. Doch nicht nur gibt es weitere wichtige Typklassen, die jetzt ebenfalls verwendet werden können, es gibt auch viele Funktionen, die mit Typkonstruktorvariablen arbeiten und daher bisher nicht verwendbar waren.

Es bietet sich also an, die Liste der `KnownDeclarations` auszubauen und sich zu überlegen, welche Typkonstruktorklassen und Funktionen sinnvoll sind.

Literatur

[Atk12] Robert Atkey. Relational parametricity for higher kinds. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 16. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[Böh07] Sascha Böhme. Free theorems for sublanguages of haskell. *Diplomarbeit, Technische Universität Dresden*, 2007.

[Fla06] David Flanagan. *JavaScript: the definitive guide*. Ö'Reilly Media, Inc.", 2006.

[fma] The free theorem for fmap. `https://www.schoolofhaskell.com/user/edwardk/snippets/fmap`. Zugriff: 23.01.2017.

[fre] Automatic generation of free theorems. `http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi`. Zugriff: 17.01.2017.

[has] haskell-src-exts. `https://hackage.haskell.org/package/haskell-src-exts`. Zugriff: 04.01.2017.

[Hin69] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[Jon93] Mark P Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61. ACM, 1993.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[JV04] Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. *ACM SIGPLAN Notices*, 39(1):99–110, 2004.

[JV06] Patricia Johann and Janis Voigtländer. The impact of seq on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1-2):63–102, 2006.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[Rey83] John C Reynolds. Types, abstraction and parametric polymorphism. 1983.

[Str95] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 1995.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1-2):11–49, 2000.

[Voi09] Janis Voigtländer. Free theorems involving type constructor classes: Functional pearl. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 173–184, New York, NY, USA, 2009. ACM.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

[YWC+12] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.

A Anhang

Übersicht

Die in dieser Arbeit angesprochene Erweiterung der Bibliothek *free-theorems* liegt in der folgenden Ordnerstruktur vor. Dabei handelt es sich sowohl um das angepasste Paket *free-theorems* als auch um eine minimal angepasste Version von *free-theorems-webui* und ein Testwerkzeug in `test-free-theorems`.

```
free-theorems-0.3.2.0
└── src
    └── Language
        └── Haskell
            └── FreeTheorems
                ├── Frontend
                ├── Parser
                └── Theorems
free-theorems-webui-0.2.1.1
test-free-theorems
```

Installation

Um das Paket aus dem Verzeichnis zu installieren, kann das Kommandozeilentool `cabal` verwendet werden, da es sich bei *free-theorems* um ein cabal-Paket handelt. Dieses Paket wurde bei der Erweiterung um Typkonstruktorklassen angepasst, um mit neueren Versionen des GHC kompatibel zu sein, da die letzte Version von *free-theorems* etwas veraltet war. So wurden Versionsnummern der eingebundenen Pakete sowie verwendete Spracherweiterungen angepasst.

Um das Paket zu installieren, reicht der folgende Aufruf im Verzeichnis namens `free-theorems-0.3.2.0`, in dem sich auch die Datei `free-theorems.cabal` befindet.

```
cabal install
```

Das Paket wurde mit dem Glasgow Haskell Compiler Version 7.10.3 getestet und sollte auch mit neuren Versionen funktionieren. Die vom Paket benötigten Abhängigkeiten sollten vom cabal-Tool selbstständig aufgelöst werden.

Um eine modifizierte Version des Weboberflächen-Pakets zu installieren, die zusätzlich mit den Typkonstruktorklassen `Monad` und `Functor` ausgestattet ist, reicht ebenfalls der Befehl `cabal install` im Verzeichnis `free-theorems-webui-0.2.1.1`. Tatsächlich handelt es sich dabei um das Originalpaket, in dem lediglich die Datei `KnownDeclarations.hs` leicht modifiziert wurde. Die Bibliothek *free-theorems* ist auch mit dem unangepassten Original-Paket kompatibel, das sich über den folgenden Befehl installieren lässt.

```
cabal install free-theorems-webui
```

In diesem Fall unterstützt die Weboberfläche die Typklassen `Monad` und `Functor` nicht, die grundsätzliche Funktionalität von Typkonstruktorklassen ist aber enthalten (es lassen sich beispielsweise Typkonstruktorklassen im entsprechenden Feld selbst deklarieren).

Verwendung

Sobald die Bibliothek installiert wurde, lässt sich `Language.Haskell.FreeTheorems` in eigenen Haskell-Modulen importieren. Eine Beispielanwendung ist im Verzeichnis namens `test-free-theorems` zu finden in der Datei `test.hs`. Diese wurde während der Erweiterung der Bibliothek hauptsächlich zum Testen entwickelt, liefert jedoch die meisten Funktionalitäten, die die Bibliothek *free-theorems* bietet.

Die Datei sollte sich ganz normal über `ghc test.hs` kompilieren lassen. Das Programm fragt nach einer Funktionssignatur der Form `funktionsname :: funktionstyp`, zu der dann das entsprechende freie Theorem generiert wird, außerdem werden entsprechende *lift*-Definitionen angezeigt.

Darüber hinaus bietet `test` einige Befehle, die insbesondere hilfreich sein können, um die Interna der Bibliothek *free-theorems* nachzuvollziehen.

Befehl	Funktion
:v	Schaltet den *verbosen* Modus an bzw. aus. Dieser Modus zeigt zusätzlich zu den Ergebnisformeln die auftretenden Zwischendarstellungen an.
:basic	Setzt die Sprache auf `BasicSubset`
:fix	Setzt die Sprache auf `(SubsetWithFix EquationalTheorem)`
:fix!	Setzt die Sprache auf `(SubsetWithFix InquationalTheorem)`
:seq	Setzt die Sprache auf `(SubsetWithSeq EquationalTheorem)`
:seq!	Setzt die Sprache auf `(SubsetWithSeq InequationalTheorem)`
:q	Beendet das Programm

Tabelle 6: Befehle für das Testprogramm

Die Datei `KnownDeclarations.hs`, die von `test.hs` eingebunden wird, ist eine modifizierte Version derselben Datei, die sich im Original-Paket zum Paket namens `free-theorems-webui-0.2.1.1` finden lässt, wobei es sich um die Weboberflächen-Implementierung für *free-theorems* handelt. Der Ordner `free-theorems-webui-0.2.1.1` enthält diese Implementierung, wobei die `KnownDeclarations.hs` dort ebenfalls modifiziert wurde.