Erweiterung von Curry um Multiparametertypklassen

Leif-Erik Krüger

Masterarbeit
Oktober 2021

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

Betreut durch
Prof. Dr. Michael Hanus
M.Sc. Finn Teegen
M.Sc. Kai Prott

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Typklassen bieten ein elegantes Konzept zur Umsetzung von Ad-hoc-Polymorphismus in statisch getypten Programmiersprachen und haben sich zu einem herausstechenden Merkmal der funktionalen Programmiersprache Haskell entwickelt. Nach ihrer Adaption in die Haskell-ähnliche deklarative Programmiersprache Curry, welche die Konzepte der funktionalen Programmierung mit denen der logischen Programmierung vereint, konnten Typklassen auch dort breite Anwendung finden. Die zahlreichen Erweiterungen zu Typklassen, die in Haskell neue Anwendungsmöglichkeiten für Typklassen eröffnen, haben bislang aber noch nicht ihren Weg nach Curry gefunden.

Diese Arbeit erweitert ein Curry-System um Multiparametertypklassen, also um die Unterstützung von Typklassen mit beliebig vielen Klassenparametern. Außerdem werden mit Erweiterungen zu flexiblen Instanzen und flexiblen Kontexten bisher bei der Verwendung von Typklassen geltende Einschränkungen aufgehoben. Eine ebenfalls angedachte Erweiterung um funktionale Abhängigkeiten zum Ausdrücken von Beziehungen zwischen Klassenparametern konnte leider nicht umgesetzt werden. Die behandelten Erweiterungen werden motiviert, formal definiert und ihre Implementierung im Curry-System beschrieben. Auch die bei der Umsetzung funktionaler Abhängigkeiten aufgetretenen Probleme werden geschildert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Wenn Funktionen nur für feste Typen definiert werden können, sorgt dies für umfangreiche Codeduplikation. Für eine Funktion wie `length`, welche die Länge einer Liste berechnet, müsste für jede Art von Liste, für welche die Länge benötigt wird, eine eigene Definition angegeben werden. Es bräuchte also z. B. jeweils eine `length`-Variante für Listen von Ganzzahlen, Listen von Fließkommazahlen, Zeichenketten, Listen von Zeichenketten, etc. Diese Varianten würden aber allesamt identisch oder ähnlich definiert sein, da die Länge einer Liste nicht vom Typ der Listenelemente abhängt.

Diese Eigenschaft einer `length`-Funktion kann mit *parametrischem Polymorphismus* angegeben werden: Er erlaubt die Angabe von Typparametern in den Argument- und Rückgabetypen einer Funktion und drückt aus, dass die Funktion für jede Belegung dieser Parameter mit Typen anwendbar ist [Str67]. Der parametrische Polymorphismus stößt aber an seine Grenzen, wenn z. B. eine `elem`-Funktion implementiert werden soll, die prüft, ob ein übergebenes Argument Element einer Liste ist. Hierzu müssen die Elemente der Liste mit dem gesuchten Argument verglichen werden. Die dafür verwendete Vergleichsfunktion kann aber nicht nur parametrisch-polymorph sein, da die Art, auf die zwei Werte miteinander verglichen werden, in der Regel von den Typen dieser Werte abhängig ist. Diese Vergleichsfunktion muss deswegen stattdessen *Ad-hoc-Polymorphismus* nutzen: Bei dieser Form des Polymorphismus kann eine Funktion ebenfalls auf verschiedene Typen angewandt werden, kann sich aber abhängig von diesen Typen unterschiedlich verhalten [Str67].

Die von Wadler und Blott [WB89] erstmals vorgestellten *Typklassen* bieten eine elegante Möglichkeit, den gewünschten Ad-hoc-Polymorphismus mit einem statischen Typsystem zu kombinieren. Eine Typklasse steht für eine Menge von Typen, für die bestimmte *Klassenmethoden* implementiert sind, beispielsweise ein Gleichheitsoperator für eine Klasse von Typen mit vergleichbaren Werten. Mit *Instanzen*, die eine Implementierung dieser Methoden angeben, können neue Typen zu dieser Menge hinzugefügt werden und mit *Constraints* kann im Typ einer Funktion angegeben werden, dass die Funktion nur für die Belegung eines Typparameters mit einem Element einer bestimmten Typklasse anwendbar ist.

Typklassen wurden für die rein funktionale Programmiersprache *Haskell*[1] entwickelt [WB89] und in dieser Sprache mehrfach um zusätzliche Funktionalitäten erweitert. So hat Jones [Jon93] Typklassen zu *Typkonstruktorklassen* verallgemeinert, die es unter anderem ermöglichen, Klassen von Behältertypen wie Listen zu definieren. Eine weitere wichtige Erweiterung von Typklassen, die bereits bei der Vorstellung von Typklassen erwähnt [WB89] und zuerst in der funktionalen Programmiersprache *Gofer* implementiert wurde [Jon94], sind *Multiparametertypklassen*. Typklassen mit mehreren Typparametern, die durch diese Erweiterung erlaubt werden, stellen Relationen auf Typen dar und können z. B. verwendet werden, um eine allgemeine Funktion zur Transformation von Werten zwischen Typen zu implementieren. Als weitere Ergänzung zu Multiparametertypklassen wurden später *funktionale Abhängigkeiten* entwickelt, mit denen ein Zusammenhang zwischen Typklassenparametern ausgedrückt werden kann [Jon00]. Diese Zusammenhänge ermöglichen es, aus der Typbelegung einiger Typklassenparameter die Belegung der davon abhängigen Parameter zu bestimmen, und verbessern

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

so die Anwendbarkeit von Multiparametertypklassen.

Mehrere Implementierungen der funktionallogischen Programmiersprache *Curry*[2] unterstützen bereits Typ- und Typkonstruktorklassen, aber noch keine der weitergehenden Funktionen von Typklassen, die im *Glasgow Haskell Compiler (GHC)*[3], der am weitesten verbreiteten Haskell-Implementierung, mithilfe von *Spracherweiterungen* aktiviert werden können. Um diese Funktionen auch in Curry nutzen zu können und damit auch eine höhere Kompatibilität zwischen Curry und Haskell zu erreichen, ist eine Umsetzung dieser Spracherweiterungen in Curry wünschenswert.

1.1. Zielsetzung

Das Ziel dieser Arbeit ist die Erweiterung des *Curry-Frontends*[4] um Spracherweiterungen für Multiparametertypklassen und funktionale Abhängigkeiten. Als wünschenswerte optionale Ziele werden zudem Spracherweiterungen für flexible Instanzen und flexible Kontexte angestrebt, welche die Regeln lockern, die für die Form von Instanzen bzw. Constraints gelten. Beim Curry-Frontend handelt es sich um ein bestehendes Curry-System, das als Überbau für zwei bedeutende Curry-Implementierungen dient, das *Portland Aachen Kiel Curry System (PAKCS)*[5] und das *Kiel Curry System 2 (KiCS2)*[6].

1.2. Vorangegangene Arbeiten

In diesem Abschnitt betrachten wir kurz einige der vorangegangenen Arbeiten, die Beiträge zur Unterstützung von Typklassen mit Erweiterungen in Curry geleistet haben:

Da in dieser Arbeit das Curry-Frontend um neue Typklassenfunktionalitäten erweitert wird, baut sie vor allem auf der dort bereits umgesetzten Unterstützung für einfache Typklassen und Typkonstruktorklassen auf. Diese Typklassenunterstützung hat Finn Teegen 2016 im Rahmen seiner Masterarbeit entwickelt [Tee16] und später um die automatische Ableitung von Instanzen zu einigen vordefinierten Klassen erweitert.

Der *Münster Curry Compiler (MCC)*[7] ist eine weitere Curry-Implementierung, die Typklassen und Typkonstruktorklassen unterstützt, wenn auch nur in einem experimentellen Entwicklungszweig. Insbesondere, da das Curry-Frontend ursprünglich auf dem MCC basiert, war diese von Wolfgang Lux umgesetzte Erweiterung [Lux08] eine Inspiration zur Umsetzung der Typklassen-Unterstützung im Curry-Frontend [Tee16].

Einen im Vergleich zu den bereits genannten Curry-Implementierungen anderen Ansatz zur Kompilation von Curry-Programmen bietet das *Curry-Plugin* für den GHC, welches Kai-Oliver Prott 2020 im Rahmen seiner Masterarbeit entwickelt hat [Pro20]. Da dieses Plugin den GHC verwendet, um Curry-Programme zu kompilieren, können viele der vom GHC unterstützten Spracherweiterungen ohne großen zusätzlichen Aufwand auch mit dem Plugin verwendet werden, darunter auch Multiparametertypklassen, funktionale Abhängigkeiten, flexible Instanzen und flexible Kontexte. Das Curry-Plugin ist aufgrund seines Ansatzes aber auch stark vom GHC abhängig und unterstützt zum aktuellen Zeitpunkt noch nicht den gesamten Sprachumfang von Curry, weswegen eine Implementierung dieser Spracherweiterungen in einer eigenständigen Curry-Implementierung weiterhin sinnvoll ist.

[2] `http://curry-lang.org/`
[3] `https://www.haskell.org/ghc/`
[4] `https://git.ps.informatik.uni-kiel.de/curry/curry-frontend`
[5] `https://www.informatik.uni-kiel.de/~pakcs/`
[6] `https://www-ps.informatik.uni-kiel.de/kics2/`
[7] `http://danae.uni-muenster.de/curry/`

1.3. Gliederung

Nach der in diesem Kapitel erfolgten Einleitung gliedert sich der der Rest dieser Arbeit in die folgenden Kapitel:

Zunächst gibt Kapitel 2 eine Einführung in die Programmiersprache Curry, bei der insbesondere die Unterschiede zu Haskell behandelt werden. Daraufhin werden in Kapitel 3 reguläre Typklassen und die mit dieser Arbeit angestrebten Erweiterungen motiviert und informell beschrieben.

In Kapitel 4 folgt dann eine formale Beschreibung von Typklassen mit allen angestrebten Erweiterungen, bei der auch die Behandlung von Typklassenelementen bei der Prüfung eines Curry-Programms ausgeführt wird. Als nächstes geht es in Kapitel 5 um die Implementierung der behandelten Typklassenerweiterungen, wobei zuerst das Curry-Frontend vorgestellt wird und dann die durchgeführten Überarbeitungen sowie die bei der Umsetzung funktionaler Abhängigkeiten aufgetretenen Probleme beschrieben werden.

Abschließend fasst Kapitel 6 die Ergebnisse dieser Arbeit zusammen und gibt einen kurzen Ausblick auf mögliche weiterführende Arbeiten und alternative Ansätze.

Kapitel 2

Die Programmiersprache Curry

Curry ist eine deklarative Programmiersprache, die Konzepte aus der funktionalen und der logischen Programmierung zu einer *funktionallogischen Programmiersprache* vereint [Han+16]. Curry weist große Ähnlichkeiten zu der rein funktionalen Programmiersprache Haskell [Mar+10] auf: So ist die Curry-Syntax stark an jene von Haskell angelehnt und auch bei der Semantik gibt es große Gemeinsamkeiten, darunter eine nicht-strikte Auswertungsstrategie. Aufgrund dessen benötigen zahlreiche Haskell-Programme keine oder nur kleine Anpassungen, um als Curry-Programm akzeptiert zu werden und dort das gleiche Verhalten zu zeigen.

Wir gehen im Folgenden davon aus, dass der Leser mit der Programmiersprache Haskell vertraut ist, und werden wegen der Ähnlichkeit zu Curry in Bezug auf funktionale Programmierung in diesem Kapitel vor allem die logische Komponente von Curry betrachten, zu der die Konzepte von *Nichtdeterminismus* und *freien Variablen* gehören, die wichtige Unterschiede zwischen Curry und Haskell darstellen.

2.1. Nichtdeterminismus

Im Gegensatz zu funktionalen Sprachen, bei denen es in der Regel das Ziel einer Berechnung ist, einen Ausdruck zu höchstens einem Wert auszuwerten, können in der Logikprogrammierung nichtdeterministisch mehrere Lösungen für eine Anfrage gefunden werden. In der Programmiersprache Curry kann durch überlappende Funktionsregeln dafür gesorgt werden, dass ein Ausdruck zu verschiedenen Werten ausgewertet werden kann. Ein Beispiel dafür ist die folgende `insert`-Funktion, die nichtdeterministisch ein Element an eine beliebige Stelle in einer Liste einfügt:

```
insert :: a -> [a] -> [a]
insert x ys       = x : ys
insert x (y : ys) = y : insert x ys
```

Die Semantik von Haskell sieht vor, die oberste Regel einer Funktion anzuwenden, die zu den übergebenen Argumenten passt. Da die obere `insert`-Regel unabhängig von den Argumenten passt, würde auch nur diese Regel tatsächlich verwendet und das Element somit immer an die erste Stelle der Liste eingefügt werden. In Curry können hingegen unabhängig von der Reihenfolge alle passenden Funktionsregeln angewandt werden. Wird `insert` mit einer nichtleeren Liste als Argument aufgerufen, wird das Element somit nichtdeterministisch entweder an den Anfang der Liste gesetzt oder in die Liste hinter dem Listenkopf eingefügt. Für den Ausdruck `insert 2 [1, 3]` werden deswegen die drei verschiedenen Ergebnisse `[1, 3, 2]`, `[1, 2, 3]` und `[2, 1, 3]` gefunden.

Aufgrund der Möglichkeit für mehrere Lösungen kann es auch sinnvoll sein, eine Funktion nur partiell zu definieren, um die Lösungen einzuschränken. Ein Beispiel dafür ist die Funktion `sorted`, die ihr Argument unverändert zurückgibt, aber nur für aufsteigend sortierte Listen definiert ist:

```
sorted :: Ord a => [a] -> [a]
sorted xs             | length xs <= 1 = xs
sorted (x1 : x2 : xs) | x1 <= x2       = x1 : sorted (x2 : xs)
```

`sorted` kann nun mit der Funktion `insert` kombiniert werden, um ein Element an die korrekte Stelle in einer aufsteigend sortierten Liste einzufügen. Beispielsweise liefert der Ausdruck `sorted (insert 2 [1, 3])` nur noch `[1, 2, 3]` als einziges Ergebnis.

Neben überlappenden Funktionsregeln können auch vordefinierte Operatoren und Funktionen wie `?` und `anyOf` verwendet werden, um Nichtdeterminismus auszudrücken. Beispiele dafür sind die folgenden äquivalenten Funktionen `dice1` und `dice2` zum nichtdeterministischen Würfeln:

```
dice1, dice2 :: Num a => a
dice1 = 1 ? 2 ? 3 ? 4 ? 5 ? 6
dice2 = anyOf [1, 2, 3, 4, 5, 6]
```

Die Semantik von Curry bei überlappenden Funktionsregeln kann für Haskell-Programmierer aber auch unerwünschte Folgen haben, wie die folgende Implementierung einer `zip`-Funktion zeigt:

```
zip :: [a] -> [b] -> [(a, b)]
zip (x : xs) (y : ys) = (x, y) : zip xs ys
zip _        _        = []
```

Dadurch, dass in Curry die untere Regel von `zip` auch dann angewandt werden kann, wenn die obere passt, ist diese `zip`-Implementierung nichtdeterministisch und gibt eine beliebig lange Anfangsliste der gewünschten Lösung zurück. Um dieses Problem zu beheben, müssen die Regeln so angepasst werden, dass sie sich nicht überlappen, was unter Umständen die Ergänzung weiterer Regeln erfordert. Alternativ kann die Fallunterscheidung in einen **`case`**-Ausdruck verschoben werden, da dort wie in Haskell nur die erste passende Alternative gewählt wird. Die folgende `zip'`-Funktion zeigt, wie eine korrigierte Version von `zip` ohne überlappende Funktionsregeln aussehen kann:

```
zip' :: [a] -> [b] -> [(a, b)]
zip' (x : xs) (y : ys) = (x, y) : zip' xs ys
zip' (_ : _)  []       = []
zip' []       _        = []
```

2.2. Freie Variablen

Mithilfe des **`free`**-Schlüsselworts werden in Curry freie Variablen deklariert, die dann in einem Wächter oder auf der rechten Seite einer Funktionsgleichung verwendet werden können, ohne auf der linken Seite vorkommen zu müssen. Die Verwendung freier Variablen drückt aus, dass eine Belegung dieser Variablen gefunden werden soll, für die eine gültige Lösung berechnet werden kann. Ein Beispiel für die Verwendung freier Variablen ist in der folgenden `last`-Funktion zu sehen, die das letzte Element einer Liste zurückgibt:

```
last :: Data a => [a] -> a
last xs | _ ++ [x] =:= xs = x
  where x free
```

Hier sind `x` und die anonyme Variable `_` freie Variablen, wobei dies bei letzterer nicht explizit angegeben werden muss. Der Operator `=:=` hat den Typ `Data a => a -> a -> Bool` und drückt ein *Gleichheitsconstraint* aus, welches nicht mit den *Typconstraints* zu verwechseln ist, die in Abschnitt 3.1

erklärt werden. `=:=` ist so definiert, dass `True` zurückgegeben wird, wenn beide Argumente zu unifizierbaren Termen ohne Funktionssymbole ausgewertet werden können. Für negative Ergebnisse, also ungleiche Argumente, ist `=:=` nicht definiert, sodass z. B. `False =:= True` nicht zu `False` ausgewertet wird, sondern kein Ergebnis liefert. Das Typconstraint `Data a`, welches der Operator `=:=` und somit auch die Funktion `last` hat, stellt sicher, dass Werte des Typs `a` miteinander verglichen werden können.

Für die Auswertung eines Ausdrucks wie `last [1, 2, 3]` wird also nach einer Belegung für eine anonyme Variable und `x` gesucht, sodass `_ ++ [x] =:= [1, 2, 3]` erfüllt ist. Da dies nur für die Belegung von `x` mit `3` und der anonymen Variable mit `[1, 2]` gilt, ist `3` die einzige Lösung für den auszuwertenden Ausdruck.

Bei der interaktiven Verwendung von Curry können freie Variablen auch im initialen Ausdruck verwendet werden. Dann wird zusätzlich zu den berechneten Lösungen auch die jeweils zugehörige Variablenbelegung ausgegeben. Dies lässt sich nutzen, um einfach eine bereits definierte Funktion umzukehren, also die Argumente zu berechnen, mit denen die Funktion ein angegebenes Ergebnis zurückliefert. Folgendes Beispiel zeigt, wie dies für die in Abschnitt 2.1 definierte `insert`-Funktion aussehen kann:

```
> insert x xs =:= [1, 2, 3] where x, xs free
{x=3, xs=[1,2]} True
{x=2, xs=[1,3]} True
{x=1, xs=[2,3]} True
```

Kapitel 3

Einführung in Typklassen mit Erweiterungen

Nachdem einfache Typklassen bereits in der Einleitung motiviert wurden, werden in diesem Kapitel zunächst die einzelnen Elemente der bereits im Curry-Frontend unterstützten Typklassen an Beispielen genauer erklärt (Abschnitt 3.1). Dann werden die in dieser Arbeit angestrebten Typklassenerweiterungen anhand einer Klasse, die eine Schnittstelle für listenähnliche Behältertypen anbieten soll, motiviert und informell beschrieben (Abschnitte 3.2 bis 3.5).

3.1. Reguläre Typklassen

In diesem Abschnitt wollen wir informell beschreiben, wie Typklassen mit genau einem Parameter, die wir *reguläre Typklassen* nennen, in Curry funktionieren. Dazu werden wir als erstes die grundlegenden Komponenten von Typklassen behandeln und danach in den Unterabschnitten 3.1.1 bis 3.1.3 weitergehende Funktionen von Typklassen vorstellen.

Eine Typklasse wird mit einer *Klassendeklaration* angelegt, die den Namen der Klasse, den *Klassenparameter* und die *Klassenmethoden* enthält. Ein Beispiel dafür ist die folgende Deklaration einer `Eq`-Klasse:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Diese `Eq`-Klasse verwendet als Klassenparameter die Typvariable `a` und besitzt mit dem Gleichheitsoperator `(==)` genau eine Klassenmethode. Mit dem angegebenen Typ `a -> a -> Bool` erfüllt `(==)` die Einschränkung, dass alle Klassenmethoden eine explizite Typsignatur benötigen, welche den Klassenparameter enthalten muss. Zu beachten ist, dass der vollständige Typ vom `(==)`-Operator `Eq a => a -> a -> Bool` ist. Das hinzugefügte *Typconstraint* `Eq a` drückt aus, dass mit `(==)` nur Werte verglichen werden können, für deren Typ eine `Eq`-Instanz existiert. Insgesamt wird der Teil eines Typs, der vor dem Doppelpfeil `=>` steht, als *Kontext* bezeichnet und kann aus beliebig vielen Constraints bestehen.

Um mit dem `(==)`-Operator zwei Werte vergleichen zu können, benötigen wir also eine Instanz der `Eq`-Klasse, die `(==)` für den Typ dieser Werte implementiert. Eine solche Typklasseninstanz wird mit einer *Instanzdeklaration* angelegt, die den Namen der *Instanzklasse*, einen *Instanztyp* und Implementierungen von Klassenmethoden der Instanzklasse enthält. Im Folgenden sehen wir eine Instanz der Klasse `Eq` für den Instanztyp `Bool`:

```
instance Eq Bool where
  True  == y = y
  False == y = not y
```

Die Implementierung einer Klassenmethode in einer Instanz muss den in der Klassendeklaration angegebenen Typ haben, wobei der Klassenparameter durch den Instanztyp ersetzt wird. So hat die obige Implementierung des `(==)`-Operators für den Instanztyp `Bool` korrekterweise den Typ `Bool -> Bool -> Bool`.

Bei der Verwendung einer Klassenmethode muss sichergestellt werden, dass eine passende Instanz mit der benötigten Implementierung existiert. Dies kann über das Typconstraint der Klassenmethode geprüft werden, wie sich anhand des Ausdrucks `False == False` zeigen lässt: Hier wird `(==)` auf zwei Werte vom Typ `Bool` angewandt, sodass `Eq a => a -> a -> Bool`, der allgemeine Typ von `(==)`, hier zu `Eq Bool => Bool -> Bool -> Bool` spezialisiert wird. Durch diese Verwendung von `(==)` entsteht also das Constraint `Eq Bool`. Da mit der `Eq`-Instanz für `Bool` eine passende Instanz und somit eine passende Implementierung der Klassenmethode existiert, kann das Constraint aufgelöst werden.

Wenn eine Klassenmethode in einer polymorphen Funktion verwendet wird, kann auf diese Weise auch ein Typconstraint entstehen, das eine Typvariable beinhaltet. Als Beispiel hierfür betrachten wir die folgende `elem`-Funktion, die zurückgibt, ob ihr erstes Argument ein Element der als zweites Argument übergebenen Liste ist:

```
elem :: Eq e => e -> [e] -> Bool
elem _ []       = False
elem x (y : ys) = x == y || elem x ys
```

Hier werden `x` und `y` miteinander verglichen, die laut der Typsignatur beide den Typ `e` haben. Somit hat `(==)` in dieser Verwendung den Typ `Eq e => e -> e -> Bool`. Da keine Instanz zum Constraint `Eq e` passt, muss es zum Kontext des Typs von `elem` hinzugefügt werden, um auszudrücken, dass die Funktion nur angewandt werden kann, wenn für den Typ, mit dem `e` belegt wird, eine `Eq`-Instanz existiert. Dieses Constraint muss nun auch beim rekursiven Aufruf von `elem` beachtet werden: Die Constraints einer gewöhnlichen Funktion müssen genauso behandelt werden wie die einer Klassenmethode, also muss auch das durch die Verwendung von `elem` entstehende Constraint aufgelöst oder dem Kontext hinzugefügt werden. Da `x` vom Typ `e` und `ys` vom Typ `[e]` ist, entsteht erneut das Constraint `Eq e`, welches bereits im Kontext enthalten und somit abgedeckt ist.

Eine Instanzdeklaration kann ebenfalls einen Kontext haben. Ein Beispiel hierfür ist diese `Eq`-Instanz für Listen:

```
instance Eq a => Eq [a] where
  []       == []       = True
  (x : xs) == (y : ys) = x == y && xs == ys
  []       == (_ : _)  = False
  (_ : _)  == []       = False
```

Da zum Vergleich von Listen die einzelnen Listenelemente miteinander verglichen werden, benötigt diese Instanz ein `Eq`-Constraint für den Typ dieser Listenelemente, das im Kontext der Instanz enthalten sein muss. Somit kann diese Instanz als formale Beschreibung angesehen werden, wie eine Implementierung zum Vergleich einzelner Werte zum Vergleich von Listen dieser Werte erweitert werden kann. Der nicht-leere Kontext hat auch einen Einfluss darauf, wie die Instanz zur Auflösung von Constraints eingesetzt wird: Wenn `(==)` auf zwei Listen angewandt wird und dadurch ein Constraint der Form `Eq [a]` für einen Typ `a` entsteht, kann dieses Constraint mit der `Eq`-Instanz für Listen nicht direkt aufgelöst werden. Stattdessen wird es durch den Kontext dieser Instanz ersetzt, in diesem Fall also zu `Eq a` vereinfacht. Im Allgemeinen werden diese Vereinfachungen sowie die Auflösung von Constraints als *Kontextreduktion* bezeichnet.

3.1.1. Standardimplementierungen

In einer Klassendeklaration kann zu einer Klassenmethode auch eine Implementierung angegeben werden. Ein Beispiel dafür sehen wir in der anschließenden Variante der `Eq`-Klasse, die um einen `(/=)`-Operator erweitert wurde:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Hier ist die Ungleichheit als Negation der Gleichheit implementiert und umgekehrt. Auf diese *Standardimplementierungen* von `(==)` bzw. `(/=)` wird dann zurückgegriffen, wenn die jeweils passende Instanz die entsprechende Klassenmethode nicht implementiert. Aufgrund dessen ist es ausreichend, wenn eine `Eq`-Instanz eine Implementierung für einen dieser Operatoren beinhaltet. Somit implementieren die zuvor definierten `Eq`-Instanzen indirekt auch `(/=)` und müssen nicht an die Erweiterung der `Eq`-Klasse angepasst werden.

3.1.2. Oberklassen

Bei der Deklaration einer Klasse können *Oberklassen* im Kontext angegeben werden, um auszudrücken, dass für jede Instanz der deklarierten Klasse auch eine Instanz der angegebenen Oberklasse für den gleichen Instanztyp existieren muss. Ein Beispiel dafür ist die folgende Variante einer `Ord`-Klasse, deren Klassenmethoden weitere Vergleichsoperatoren sind, die zum Sortieren von Werten verwendet werden können:

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
```

Da `Eq` eine Oberklasse von `Ord` ist, kann eine Funktion mit dem Typ `Ord a => a -> a -> Bool` sowohl `(<)` als auch `(==)` auf Werte des Typs `a` anwenden und benötigt nicht zusätzlich ein `Eq`-Constraint. `Ord`-Instanzen mit einem Instanztyp, für den keine `Eq`-Instanz existiert, dürfen jedoch nicht zugelassen werden, damit die Existenz einer passenden Implementierung von `(==)` gewährleistet ist.

3.1.3. Typkonstruktorklassen

Manche Typkonstruktoren müssen auf Typargumente angewandt werden, um einen Typ zu bilden. So ist z. B. `Maybe a -> Bool` ein gültiger Typ für eine Funktion, aber `Maybe -> Bool` nicht, da `Maybe` mit genau einem Typargument einen Typ bildet. Als Typkonstruktorklassen werden Typklassen bezeichnet, deren Klassenparameter für einen nicht vollständig angewandten Typkonstruktor steht. Die folgende Klasse `ListLike` ist ein Beispiel für eine Typkonstruktorklasse, was sich daran erkennen lässt, dass der Klassenparameter in den Typen der Klassenmethoden auf ein Typargument angewandt wird.

```
class ListLike l where
  empty :: l e
  insert :: e -> l e -> l e
```

Die Klasse `ListLike` ist dazu gedacht, Varianten der grundlegenden Funktionen von Listen anzubieten. Damit sollen auf Listen arbeitende Funktionen wie `map` allgemeiner definiert und mit verschiedenen Behältertypen verwendet werden können. Zur Vereinfachung beinhaltet die obige `ListLike`-Klasse aber zunächst nur Funktionen zum Aufbau von Listen.

Instanzdeklarationen zu Typkonstruktorklassen unterscheiden sich von anderen Instanzdeklarationen hauptsächlich darin, dass als Instanztyp ein nicht vollständig angewandter Typkonstruktor angegeben wird. Eine `ListLike`-Instanz für den Listenkonstruktor selbst sieht beispielsweise folgendermaßen aus:

```
instance ListLike [] where
  empty = []
  insert x ys = x : ys
```

3.2. Multiparametertypklassen

Multiparametertypklassen sind eine Erweiterung regulärer Typklassen, durch die auch Klassendeklarationen erlaubt sind, die keinen oder mehr als einen Klassenparameter haben. Um diese Erweiterung zu motivieren, betrachten wir in Abschnitt 3.2.1 ein Problem mit der in Abschnitt 3.1.3 deklarierten `ListLike`-Klasse. Danach beleuchten wir in Abschnitt 3.2.2 kurz die Hintergründe dieser Erweiterung und beschreiben in Abschnitt 3.2.3 informell ihre Funktionsweise, indem wir das zuvor betrachtete Problem mit Multiparametertypklassen lösen.

3.2.1. Motivation

Die `ListLike`-Klasse soll für verschiedene Behältertypen verwendet werden können, in Abschnitt 3.1.3 wurde bislang aber nur eine Instanz für den Listenkonstruktor definiert. Um dies zu ändern, wollen wir nun eine `ListLike`-Instanz für einen Mengenkonstruktor ergänzen. Der Gedanke dahinter ist, dass Mengen sich in ihrem Verhalten zwar deutlich von Listen unterscheiden, es aber zahlreiche Funktionen wie `map` und `filter` gibt, die auf beiden Datenstrukturen sinnvoll angewandt werden können. Implementierungen einer Mengen-Datenstruktur stehen sowohl für Haskell[1] als auch für Curry[2] zur Verfügung. Wir arbeiten im Folgenden mit der Haskell-Variante dieser Datenstruktur, da `Set` dort ein Datentyp ist, während `Set` in der Curry-Variante ein Typsynonym ist und deswegen nicht ohne Weiteres als Instanztyp verwendet werden kann.

Wir können nun die in der `Set`-Schnittstelle angebotenen Funktionen `empty` und `insert` verwenden, um eine einfache `ListLike`-Instanz für den Mengenkonstruktor `Set` anzugeben:

```
instance ListLike Set where
  empty = Set.empty
  insert = Set.insert
```

Bei dem Versuch, diese Instanzdeklaration zu kompilieren, gibt das Curry-Frontend jedoch die folgende Fehlermeldung aus:

```
Method type too specific
Method: insert
Inferred type: Ord a => a -> Set a -> Set a
Expected type: a -> Set a -> Set a
```

Das Problem liegt am Typ von `Set.insert`, der dem in der Fehlermeldung ausgegebenen „Inferred type“ entspricht. Das `Ord`-Constraint, welches somit durch die Verwendung der Funktion entsteht, kann nicht zum Kontext der Implementierung der Klassenmethode `insert` hinzugefügt werden, da ihr Typ durch den in der Klassendeklaration angegebenen Typ und den Instanztyp vorgegeben ist.

Eine Lösungsmöglichkeit für dieses Problem besteht daraus, den Typ der Klassenmethode `insert` in der Klassendeklaration um ein `Ord`-Constraint zu erweitern. Damit wäre die `insert`-Funktion aber für alle `ListLike`-Instanzen eingeschränkt und man könnte z. B. mit `insert` keine Elemente zu einer

[1] `https://hackage.haskell.org/package/containers/docs/Data-Set.html`
[2] `https://www-ps.informatik.uni-kiel.de/~cpm/DOC/containers-3.0.0/Data.Set.html`

Liste von Funktionen hinzufügen, da für Funktionen in der Regel keine `Ord`-Instanzen angegeben werden können.

Eine bessere Lösungsmöglichkeit ist, die Typen der Sammlungselemente nicht allgemein, sondern abhängig vom Behältertyp einzuschränken. Dies ist mit einer regulären Typklasse aber nicht möglich, da der Typ der Elemente nicht in einem Instanztyp erwähnt werden kann und somit vollständig unabhängig vom Behältertyp sein muss.

3.2.2. Hintergrund

Multiparametertypklassen wurden bereits bei der Vorstellung von Typklassen als mögliche Erweiterung erkannt [WB89]. Als Beispiel wurde dort folgende Klasse mit zwei Parametern gezeigt:

```
class Coerce a b where
  coerce :: a -> b
```

Für diese Klasse sollten Instanzen wie `instance Coerce Int Float` angelegt werden können, die beschreiben, wie Werte eines Typs zu einem Wert eines anderen Typs konvertiert werden können. Chen, Hudak und Odersky [CHO92] haben mit diesem Ansatz jedoch mehrere Probleme gesehen: Eines davon ist, dass bereits einfache Ausdrücke wie `(coerce . coerce)` nicht typkorrekt sind, da sie *mehrdeutige Typvariablen* erzeugen. In diesem Fall ist bei der Hintereinanderausführung von `coerce` nicht klar, welcher Typ als Zwischentyp verwendet werden soll (mit diesem Problem werden wir uns in Zusammenhang mit funktionalen Abhängigkeiten in Abschnitt 3.3 genauer befassen). Aufgrund dieser Probleme haben sie mit *parametrischen Typklassen* einen anderen Ansatz zur Erweiterung regulärer Typklassen detailliert vorgestellt [CHO92]. Parametrische Typklassen können mehrere Typparameter haben, benötigen aber auch eine sogenannte *Platzhaltervariable*, die in verschiedenen Instanzen zur selben Klasse nicht mit dem gleichen Typ belegt werden darf. Mit einer solchen Einschränkung lässt sich eine Klasse wie `Coerce` nicht sinnvoll umsetzen, da eine Instanz für `Coerce Int Integer` einen Konflikt mit der bereits erwähnten Instanz für `Coerce Int Float` erzeugen würde, wenn wir den ersten Parameter von `Coerce` als Platzhaltervariable verwenden.

In der von Mark P. Jones entwickelten Haskell-ähnlichen Programmiersprache Gofer wurde der Ansatz parametrischer Typklassen allerdings nicht weiterverfolgt, sondern die ursprüngliche Variante von Multiparametertypklassen unterstützt [Jon94]. Diese Implementierung war eine Grundlage für eine Arbeit von Simon Peyton Jones, Mark P. Jones und Erik Meijer, in der verschiedene Designentscheidungen zur Umsetzung von Multiparametertypklassen und anderen Typklassenerweiterungen in Haskell diskutiert wurden [JJM97]. Diese Arbeit wiederum beeinflusste die ursprüngliche Unterstützung von Multiparametertypklassen im GHC[3].

Die Implementierung von Multiparameterklassen in Curry, die wir im nächsten Unterabschnitt informell beschreiben, orientiert sich an der aktuellen GHC-Implementierung, die im GHC-Nutzerhandbuch [GHC21] beschrieben wird[4].

3.2.3. Informelle Beschreibung der Erweiterung

Mithilfe von Multiparametertypklassen können wir einen zusätzlichen Klassenparameter für den Typ der Sammlungselemente zu der in Abschnitt 3.1.3 definierten `ListLike`-Klasse hinzufügen. In der folgenden erweiterten Variante der Klasse haben wir zudem weitere Klassenmethoden hinzugefügt und zur klaren Unterscheidung zwischen `ListLike`-Varianten ein `M` an die Namen der Klasse und Klassenmethoden angehängt:

[3] https://downloads.haskell.org/~ghc/4.06/docs/users_guide/multi-param-type-classes.html

[4] https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/multi_param_type_classes.html

```
class ListLikeM l e where
  emptyM :: l e
  insertM :: e -> l e -> l e
  isEmptyM :: l e -> Bool
  headM :: l e -> e
  tailM :: l e -> l e
```

Nun kann bei der Angabe einer Instanz zur neuen `ListLikeM`-Klasse auch der Typ der Sammlungselemente eingeschränkt werden, um das in Abschnitt 3.2.1 aufgetretene Problem eines fehlenden `Ord`-Constraints bei der `ListLike`-Instanz zum Typkonstruktor `Set` zu vermeiden. Da z. B. für den Typ `Int` eine `Ord`-Instanz vordefiniert ist, können wir nun eine Instanz angeben, die keine Typprobleme verursacht:

```
instance ListLikeM Set Int where
  emptyM = Set.empty
  insertM = Set.insert
  isEmptyM = Set.null
  headM = Set.findMin
  tailM = Set.deleteMin
```

Der Nachteil, dass diese Instanz nur für `Int`-Mengen verwendet werden kann und damit sehr spezifisch ist, kann mit flexiblen Instanzen aufgehoben werden, wie in Abschnitt 3.4 gezeigt wird.

Mithilfe der zusätzlichen Klassenmethoden aus der `ListLikeM`-Klasse lässt sich nun eine verallgemeinerte `map`-Funktion angeben:

```
map' :: (ListLikeM l e1, ListLikeM l e2) => (e1 -> e2) -> l e1 -> l e2
map' f l | isEmptyM l = emptyM
         | otherwise  = insertM (f (headM l)) (map' f (tailM l))
```

Diese Funktion benötigt zwei `ListLikeM`-Constraints, da sie einerseits z. B. mit `headM` auf die Eingabesammlung zugreift und andererseits z. B. mit `insertM` die Ausgabesammlung aufbaut. Diese Sammlungen können verschiedene Elementtypen haben und die Existenz einer `ListLikeM`-Instanz muss für beide sichergestellt werden.

Nachdem wir hiermit erfolgreich die `ListLikeM`-Klasse um eine Instanz für den Mengenkonstruktor erweitert und eine Anwendungsmöglichkeit der Klasse gezeigt haben, sollen als Nächstes noch einige weitere Möglichkeiten erwähnt werden, die durch Multiparametertypklassen entstehen:

Nullstellige Typklassen
Ein Sonderfall unter den Multiparametertypklassen sind nullstellige Klassen, die keinen Klassenparameter haben. Für solche Klassen kann höchstens eine Instanz angegeben werden, da es ohne Klassenparameter auch keine Instanztypen gibt, die sich zwischen Instanzen unterscheiden könnten. Die Funktionalität dieser Typklassen ist somit stark eingeschränkt, aber sie können z. B. zur Dokumentation eingesetzt werden. Beispielsweise kann eine Klasse angelegt werden, welche globale Einstellungen beschreibt. Dann können Funktionen, die auf diese Einstellungen zugreifen, mit einem Constraint der Klasse gekennzeichnet und die Werte der Einstellungen in einer Instanz festgelegt werden.

Oberklassen
Mit Multiparametertypklassen werden die Möglichkeiten bei der Angabe von Oberklassen wesentlich vielfältiger. Während bei regulären Typklassen zu jeder Oberklasse der einzige Klassenparameter der

deklarierten Klasse angegeben werden musste, kann bei Multiparametertypklassen eine beliebige Auswahl an Klassenparametern angegeben werden. Dabei darf auch der gleiche Klassenparameter mehrfach bei derselben Oberklasse vorkommen und die gleiche Klasse mehrfach mit unterschiedlichen Parametern als Oberklasse angegeben werden. Beispielsweise ist die folgende Klassendeklaration gültig, wenn `C` eine zweistellige und `D` eine nullstellige Typklasse ist:

```
class (C b a, C b b, D) => E a b
```

Überlappende Instanzen
In Instanzdeklarationen zu regulären Typklassen muss der Instanztyp eine bestimmte Form haben, in der keine mehrfach vorkommenden Typvariablen erlaubt sind. Diese Einschränkung sorgt dafür, dass ohne Aktivierung einer Erweiterung zu flexiblen Instanzen (siehe Abschnitt 3.4) keine Überlappung zwischen Instanzen möglich ist. Dies bedeutet, dass kein Constraint entstehen kann, zu welchem zwei verschiedene Instanzen passen. Für Instanzen zu Multiparametertypklassen gelten diese Einschränkungen zwar auch, allerdings wird jeder Instanztyp einzeln betrachtet. Dadurch ist es möglich, die gleiche Typvariable in verschiedenen Instanztypen vorkommen zu lassen und damit überlappende Instanzdeklarationen anzugeben. Ein Beispiel hierfür sind diese beiden Instanzen zu einer dreistelligen Typklasse `C`:

```
instance C [a] [a] [b]
instance C [a] [b] [b]
```

Diese Instanzen bezeichnen wir als *potenziell überlappend*, da es für beide Instanzen Constraints gibt, zu denen nur diese Instanz passt, aber auch Constraints wie `C [Int] [Int] [Int]`, zu denen beide Instanzen passen. Eine solche Instanzenüberlappung ist erlaubt und wird bei Deklaration der Instanzen auch nicht geprüft. Falls aber ein Constraint entsteht, zu dem mehrere Instanzen passen, wird eine Fehlermeldung ausgegeben.

Ein alternativer Ansatz, der während der Arbeit diskutiert wurde, ist es, mehrfach vorkommende Typvariablen auch Instanztypen-übergreifend zu verbieten. Dies würde dazu führen, dass auch mit Multiparametertypklassen keine überlappenden Instanzen definiert können, solange nicht auch die Erweiterung zu flexiblen Instanzen aktiviert wurde. Um dem Verhalten des GHC zu entsprechen und damit die Kompatibilität zu Haskell so hoch wie möglich zu halten, wurde sich jedoch dazu entschieden, diesen alternativen Ansatz vorerst nicht umzusetzen.

3.3. Funktionale Abhängigkeiten

Funktionale Abhängigkeiten sind eine Erweiterung zu Multiparametertypklassen, mit der Abhängigkeiten zwischen Klassenparametern angegeben werden können. Wir werden diese Erweiterung in Abschnitt 3.3.1 motivieren, indem wir Probleme mit einer Optimierung der `ListLikeM`-Klasse aus Abschnitt 3.2.3 vorstellen. Daraufhin geht Abschnitt 3.3.2 kurz auf die Ursprünge dieser Erweiterung ein und gibt an, wie funktionale Abhängigkeiten die vorgestellten Probleme lösen.

3.3.1. Motivation

In Abschnitt 3.2.3 konnte die `ListLikeM`-Klasse um eine Instanz für den allgemeinen Mengenkonstruktor `Set` und den Typ `Int` erweitert werden. Speziell für `Int`-Mengen wird in Haskell mit dem Datentyp `IntSet` aber auch eine effizientere Implementierung angeboten[5]. Wir wollen nun versuchen, auch diese Mengenimplementierung mit der `ListLikeM`-Klasse verwenden zu können.

[5] `https://hackage.haskell.org/package/containers/docs/Data-IntSet.html`

Das erste Problem, auf das wir dabei stoßen, ist dass der Datentyp `IntSet` im Gegensatz zum Typkonstruktor `Set` keinen Typparameter hat. Da der erste Klassenparameter der `ListLikeM`-Klasse aber für einen Typkonstruktor steht, der genau ein Typargument entgegennehmen muss, kann somit keine `ListLikeM`-Instanz für `IntSet` angelegt werden. Es ist zwar möglich, `IntSet` in einen Platzhalterdatentyp zu verpacken, der sein Typargument ignoriert, eine bessere Lösung ist es aber, die `ListLikeM`-Klasse so anzupassen, dass ein solcher Schritt nicht nötig ist.

Hierzu passen wir den ersten Klassenparameter der `ListLikeM`-Klasse so an, dass er für den Typ der gesamten Sammlung steht. Dadurch erhalten wir die folgende Typklasse `ListLikeA`, bei der wir zur Vereinfachung der Beispiele zwei Klassenmethoden aus der `ListLikeM`-Klasse entfernt haben:

```
class ListLikeA le e where
  emptyA :: le
  insertA :: e -> le -> le
  headA :: le -> e
```

Diese Klassendeklaration wird jedoch nicht akzeptiert, da der Typ der Klassenmethode `emptyA` nicht die Typvariable `e` und somit nicht alle Klassenparameter enthält. Warum solche als *mehrdeutig* bezeichneten Typvariablen wie `e` ein Problem sind, sehen wir an folgendem Beispiel:

```
instance ListLikeA Bool Int  where emptyA = True
instance ListLikeA Bool Char where emptyA = False
someBool = emptyA :: Bool
```

Hier hängt der Wert, zu dem `someBool` ausgewertet wird, von der Instanz ab, die zur Implementierung von `emptyA` verwendet wird. Da der zweite Klassenparameter von `ListLikeA` aber nicht im Typ von `emptyA` vorkommt, kann auch mit einer Typsignatur nicht bestimmt werden, welche Instanz verwendet werden soll. Aus diesem Grund müssen Klassenmethoden und andere Funktionen mit mehrdeutigen Typvariablen abgelehnt werden.

Ein weiteres Problem, das wir beobachten können, wenn wir die `emptyA`-Klassenmethode entfernen und damit die Gültigkeit der Klassendeklaration herstellen, sind unerwartete inferierte Typen. Als Beispiel ist anschließend eine Funktion `insHead` zusammen mit dem von Curry inferierten Typ angegeben:

```
insHead :: (ListLikeA b a, ListLikeA b c) => a -> b -> c
insHead e l = headA (insertA e l)
```

Da diese Funktion ihr erstes Argument in eine Sammlung einfügt und dann ein Element aus ebendieser Sammlung zurückgibt, sollte ihr Rückgabetyp identisch zu dem des ersten Arguments sein. Stattdessen sind es verschiedene Typen, was impliziert, dass `insHead` dazu in der Lage ist, ihr erstes Argument in einen anderen Typ umzuwandeln. Der Grund dafür, dass dieser unerwartete Typ inferiert wurde, ist die Unabhängigkeit zwischen dem Typ der Sammlung und dem Typ der Sammlungselemente. In der `ListLikeM`-Variante bestand hier ein klarer Zusammenhang, da der Elementtyp als Typargument im Sammlungstyp vorkam.

3.3.2. Beschreibung der Erweiterung

Funktionale Abhängigkeiten wurden von Mark P. Jones als Lösung für zahlreiche der Probleme vorgestellt, die bei der Anwendung von Multiparametertypklassen auftraten [Jon00]. In gewisser Weise wurden mit funktionalen Abhängigkeiten die Vorteile der in Abschnitt 3.2.2 kurz behandelten parametrischen Typklassen [CHO92] auf Multiparametertypklassen übertragen und dabei erweitert: Während es bei parametrischen Typklassen einen speziellen Klassenparameter gibt, von dem die

anderen Parameter abhängen, können mit funktionalen Abhängigkeiten beliebige Abhängigkeiten zwischen Klassenparametern angegeben werden.

Eine funktionale Abhängigkeit wird bei der Klassendeklaration hinter einem Trennstrich nach den Klassenparametern angegeben. Als Beispiel hierfür betrachten wir die Klasse `ListLikeF`, die sich von `ListLikeA` in der hinzugefügten funktionalen Abhängigkeit unterscheidet.

```
class ListLikeF le e | le -> e where
  emptyF :: le
  insertF :: e -> le -> le
  headF :: le -> e
```

Die funktionale Abhängigkeit `le -> e` steht dafür, dass `e`, der Typ der Sammlungselemente, eindeutig durch `le`, den Typ der Sammlung, bestimmt ist. Im Allgemeinen kann eine Klasse beliebig viele funktionale Abhängigkeiten haben und jede davon kann eine beliebige Auswahl von Klassenparametern auf der linken und rechten Seite haben. Wenn mehrere Parameter auf der linken Seite einer funktionalen Abhängigkeit stehen, bestimmen sie nur zusammen die Parameter auf der rechten Seite.

Aufgrund der funktionalen Abhängigkeit wird die obige Klassendeklaration nicht mehr abgelehnt. Die nicht im Typ von `emptyF` vorkommende Typvariable `e` zählt nun nicht mehr als mehrdeutig, da `le` im Typ vorkommt und damit laut der funktionalen Abhängigkeit `e` eindeutig bestimmt ist. Eine weitere Auswirkung dieser funktionalen Abhängigkeit sehen wir, wenn wir erneut den inferierten Typ von `insHead` betrachten, wobei wir die Constraints zu `ListLikeF` umbenannt haben:

```
insHead :: (ListLikeF b a, ListLikeF b c) => a -> b -> c
```

Da mit der funktionalen Abhängigkeit nun der erste Klassenparameter von `ListLikeF` eindeutig den zweiten bestimmt, können wir aus diesen Constraints ziehen, dass `b` eindeutig `a` und `b` eindeutig `c` bestimmt. Wenn `a` und `c` für verschiedene Typen stehen würden, wäre somit die funktionale Abhängigkeit von `ListLikeF` verletzt. Deswegen müssen `a` und `c` für denselben Typ stehen, was dazu führt, dass der folgende Typ inferiert wird:

```
insHead' :: ListLikeF b a => a -> b -> a
insHead' e l = headF (insertF e l)
```

Außerdem können wir nun ohne Schwierigkeiten eine `ListLikeF`-Instanz für die Instanztypen `IntSet` und `Int` angeben:

```
instance ListLikeF IntSet Int where
  emptyF = IntSet.empty
  insertF = IntSet.insert
  headF = IntSet.findMin
```

Viele weitere Instanzen für `ListLikeF` sind jedoch nur mit flexiblen Instanzen möglich und werden deswegen in Abschnitt 3.4 erwähnt.

Damit keine Typfehler dadurch entstehen, dass funktionale Abhängigkeiten so wie hier beschrieben angewandt werden, muss der Compiler prüfen, ob alle Instanzdeklarationen die funktionalen Abhängigkeiten einhalten. Z. B. muss die Instanz `instance ListLikeF IntSet [a]` gleich aus zwei Gründen abgelehnt werden: Einerseits gibt es einen Konflikt mit der `ListLikeF`-Instanz für `IntSet` und `Int`, da die auf der linken Seite der funktionalen Abhängigkeit stehenden Instanztypen übereinstimmen, die auf der rechten Seite stehenden aber nicht. Deswegen gibt es bei dem Constraint `ListLikeF IntSet b` zwei mögliche Typen für die Variable `b`, obwohl sie eindeutig bestimmt sein müsste. An diesem Constraint lässt sich auch der zweite Grund erkennen, aus dem die Instanz abgelehnt werden muss: Auch wenn die Instanz für `IntSet` und `Int` entfernt würde, blieben z. B. mit `[Bool]` und `[Int]`

verschiedene Möglichkeiten für `b`. Aus diesem Grund müssen auch Instanzen abgelehnt werden, bei denen die Instanztypen auf der rechten Seite einer funktionalen Abhängigkeit Typvariablen enthalten, die nicht in den Instanztypen auf der linken Seite vorkommen.

3.4. Flexible Instanzen

Die Spracherweiterung zu flexiblen Instanzen hebt die Regeln auf, die ansonsten für die Form von Instanztypen gelten. Für gewöhnlich muss jeder Instanztyp ein zu einem Datentyp gehörender Typkonstruktor sein, der auf mindestens null unterschiedliche Typvariablen angewandt wird. Beispielsweise sind die nächsten beiden Instanzen unter der Voraussetzung, dass `C` eine einstellige Typklasse ist und keine Datentypen mit den Namen `a` oder `b` existieren, ohne Erweiterung erlaubt:

```
instance C (a, b)       -- Alternative Schreibweise: instance C ((,) a b)
instance C Int          -- Anwendung auf null Typvariablen ist ebenfalls erlaubt
```

Die folgenden Instanzen sind hingegen nur dann erlaubt, wenn die Spracherweiterung zu flexiblen Instanzen aktiviert ist. Der Grund ist jeweils im Kommentar angegeben:

```
instance C a            -- 'a' ist eine Typvariable und kein Typkonstruktor
instance C String       -- 'String' ist ein Typsynonym und kein Datentyp
instance C (Maybe Int)  -- Typkonstruktor wurde auf weiteren Typkonstruktor angewandt
instance C (a, a)       -- Typkonstruktor wurde mehrfach auf Typvariable 'a' angewandt
```

Diese Einschränkungen bei der Form der Instanztypen sorgen bei regulären Typklassen dafür, dass keine überlappenden Instanzen deklariert werden können (für ein Beispiel zu Überlappung siehe Abschnitt 3.2.3). Sie vereinfachen auch den Umgang mit Instanzen bei der Typinferenz. Bei einigen Multiparametertypklassen sorgen diese Einschränkungen aber dafür, dass viele nützliche Instanzen gar nicht oder nur mithilfe von Platzhaltertypen angegeben werden können. Zu diesen Multiparametertypklassen gehören auch `ListLikeM` aus Abschnitt 3.2.3 und `ListLikeF` aus Abschnitt 3.3.2. Anschließend sind deshalb einige Instanzen dieser Klassen aufgeführt, die nur mit einer Erweiterung zu flexiblen Instanzen deklariert werden können:

```
instance ListLikeM [] e
instance Ord e => ListLikeM Set e
instance ListLikeF [e] e
instance Ord e => ListLikeF (Set e) e
```

Die Implementierungen der Klassenmethoden zu diesen Instanzen sind hier nicht angegeben, da sie sich nicht oder nur geringfügig von den angegebenen Implementierungen aus den Instanzdeklarationen der vorherigen Abschnitte unterscheiden.

3.5. Flexible Kontexte

Als Gegenstück zu flexiblen Instanzen hebt die Spracherweiterung zu flexiblen Kontexten die Regeln auf, die für die Form von Typconstraints in Kontexten gelten. Nach diesen Regeln muss ein Constraint eine Typklasse angewandt auf mindestens null Typvariablen sein. Wenn keine Spracherweiterungen aktiviert sind, sorgt diese eingeschränkte Form in Kombination mit den Einschränkungen für Instanztypen dafür, dass nur Constraints angegeben werden können, die nicht mit Instanzen reduziert werden können. Dies vereinfacht die Typinferenz, da nicht darauf geachtet werden muss, dass entstandene Constraints nicht weiter reduziert werden dürfen als die in der Typsignatur angegebenen.

Mit der Erweiterung können Klassen in Constraints auf beliebige Typen angewandt werden, was insbesondere bei Constraints zu Multiparametertypklassen nützlich ist. Ein Beispiel dafür ist die folgende verallgemeinerte `and`-Funktion, welche ein nur mit flexiblen Kontexten erlaubtes Constraint zu der in Abschnitt 3.2.3 definierten `ListLikeM`-Klasse hat:

```
and :: ListLikeM l Bool => l Bool -> Bool
and l = isEmptyM l || headM l && and (tailM l)
```

Neben Kontexten in Typsignaturen sind von der Erweiterung auch die Kontexte von Instanz- und Klassendeklarationen betroffen. So ist es mit der Erweiterung z. B. erlaubt, diese Oberklasse zur Deklaration der `ListLikeM`-Klasse hinzuzufügen:

```
class ListLikeF (l e) e => ListLikeM l e where ...
```

Das hinzugefügte Oberklassenconstraint drückt einen Zusammenhang zwischen `ListLikeM` und der in Abschnitt 3.3.2 definierten Klasse `ListLikeF` aus: Instanzen der Klasse `ListLikeM` können auch als Instanzen der Klasse `ListLikeF` angegeben werden. Dabei muss der erste Klassenparameter angepasst werden, da dieser in `ListLikeM` für einen Typkonstruktor steht, der angewandt auf den Elementtyp den Typ der Sammlung bildet, während er in `ListLikeF` alleine für den Sammlungstyp steht. Mit dieser Oberklassenbeziehung wird für Funktionen wie `map` dann nur eine `ListLikeF`-Variante benötigt, die sich dann auch mit `ListLikeM`-Constraints verwenden lässt.

Kapitel 4

Formale Beschreibung von Typklassen

In diesem Kapitel wollen wir die Form von Typklassen und der dazugehörigen Komponenten wie Constraints und Instanzen sowie die Regeln zur Behandlung dieser Elemente bei der Prüfung eines Curry-Programms formal beschreiben. In dieser Beschreibung werden alle in Kapitel 3 informell behandelten Erweiterungen miteinbezogen. Es wird zudem erwähnt, welche syntaktischen Einschränkungen gelten, wenn bestimmte Spracherweiterungen nicht aktiviert sind.

Dieses Kapitel ist in die folgenden Abschnitte gegliedert: Zunächst werden in Abschnitt 4.1 grundlegende Begriffe wie *Typausdruck* und *Constraint* formal eingeführt. Darauf aufbauend beschreibt Abschnitt 4.2 den Aufbau von Klassen- und Instanzdeklarationen. In Abschnitt 4.3 wird dann beschrieben, wie die *Kontextvereinfachung*, ein wichtiger Teil der Typprüfung, mit erweiterten Typklassen funktioniert. Anhand von Problemen, die bei dieser Vereinfachung auftreten können, werden in Abschnitt 4.4 Einschränkungen für Typklassenelemente, deren Einhaltung geprüft werden muss, motiviert und daraufhin beschrieben. Als Nächstes stellt Abschnitt 4.5 kurz vor, wie das Typsystem von Curry grundsätzlich funktioniert und wie funktionale Abhängigkeiten die Typinferenz erweitern. Zuletzt behandeln wir in Abschnitt 4.6 das *Defaulting*, einen speziellen Mechanismus zur Vermeidung von Typfehlern bei der Verwendung numerischer Literale.

4.1. Grundlagen

Typen sind ein wichtiger Bestandteil der Typklassenelemente, die wir in diesem Kapitel behandeln wollen. Um eine klare Definition von Typen und verwandten Sprachelementen aufstellen zu können, definieren wir zuerst *Typausdrücke*, mit denen Typen gebildet werden können:

Definition 4.1 (Typausdruck)**.**
Ein Typausdruck ist durch folgende Backus-Naur-Form (BNF) gegeben:

$$
\begin{aligned}
t &::= u && \text{(\textit{Typvariable})}\\
&\mid TC && \text{(\textit{Typkonstruktor})}\\
&\mid t_1\ t_2 && \text{(\textit{Typanwendung})}
\end{aligned}
$$

Für einen Typausdruck der Form $(\ldots((t_1\ t_2)\ t_3)\ldots)\ t_n$ *für ein* $n \in \mathbb{N}_{\geqslant 3}$ *und Typausdrücke* $t_1, t_2, t_3, \ldots, t_n$ *wird auch die Notation* $t_1\ t_2\ t_3\ \ldots\ t_n$ *verwendet.*

Wichtig ist, dass wir mit Typkonstruktoren in diesem Kapitel immer einen Bezeichner meinen, der durch eine Datentyp-, Typalias- oder Klassendeklaration eingeführt wurde oder wie z. B. der Listenkonstruktor vordefiniert ist.

Typausdrücke sind beliebig kombinierbar, aber nicht all diese Kombinationen lassen sich auf die gleiche Weise verwenden und einige müssen wir als nicht sinnvoll zurückweisen. Dazu definieren wir *Sorten*, mit denen Typausdrücke kategorisiert werden können:

Definition 4.2 (Sorte)**.**
Eine Sorte ist durch folgende BNF gegeben:

$$\begin{aligned} s ::= {} & * \\ \mid {} & s_1 \to s_2 \\ \mid {} & \mathsf{Constraint} \end{aligned}$$

Für eine Sorte der Form $s_1 \to (s_2 \to (\ldots (s_{n-1} \to s_n) \ldots))$ *für ein* $n \in \mathbb{N}_{\geqslant 3}$ *und Sorten* $s_1, s_2, \ldots, s_{n-1}, s_n$ *wird auch die Notation* $s_1 \to s_2 \to \cdots \to s_{n-1} \to s_n$ *verwendet.*

Einigen Typausdrücken lässt sich eine dieser Sorten zuordnen. Die Sorten von Typvariablen und Typkonstruktoren können mittels *Sorteninferenz* ermittelt werden, mit der wir uns in dieser Arbeit nicht weiter befassen wollen. Abgesehen von den Sorten der zu Klassen gehörenden Typkonstruktoren, die wir in Abschnitt 4.2 behandeln, sehen wir diese Sorten deswegen als gegeben an und definieren nur die Sorten von Typanwendungen:

Definition 4.3 (Sorte einer Typanwendung)**.**
Für alle Typausdrücke t_1, t_2 *und Sorten* s_1, s_2, *mit denen* t_1 *von der Sorte* $s_1 \to s_2$ *und* t_2 *von der Sorte* s_1 *ist, hat der Typausdruck* $t_1\ \ t_2$ *die Sorte* s_2.

Dadurch, dass hiermit nur bestimmten Typanwendungen eine Sorte zugewiesen wird, lässt sich auch nicht allen Typausdrücken eine Sorte zuordnen. Solche nicht sortenkorrekten Typausdrücke müssen als ungültig zurückgewiesen werden.

Mit unseren Definitionen von Typausdrücken und Sorten lässt sich nun eine einfache Definition von Typen angeben:

Definition 4.4 (Typ)**.**
Ein Typ *ist ein Typausdruck der Sorte* $*$.

Ähnlich wie Typen lassen sich auch Typconstraints definieren. Da ohne die Erweiterung um flexible Kontexte nicht alle Formen von Constraints erlaubt sind, führen wir zudem besondere Bezeichnungen für Constraints ein, die eine bestimmte Form haben:

Definition 4.5 (Constraint)**.**
Ein Constraint *ist ein Typausdruck der Sorte* Constraint.
Wir bezeichnen ein Constraint der Form

$$C\ \ (u_1\ \ t_{(1,1)}\ \ \ldots\ \ t_{(1,m_1)})\ \ \ldots\ \ (u_n\ \ t_{(n,1)}\ \ \ldots\ \ t_{(1,m_n)}),$$

bei dem C *ein Typkonstruktor,* $n \in \mathbb{N}_0$ *und für alle* $i \in \{1, \ldots, n\}$ u_i *eine Typvariable,* $m_i \in \mathbb{N}_0$ *sowie* $t_{(i,1)}, \ldots, t_{(i,m_i)}$ *Typausdrücke sind, als* regulär.
Ist $m_i = 0$ *für alle* $i \in \{1, \ldots, n\}$, *bezeichnen wir ein Constraint dieser Form zudem als* einfach.

Um eine Anwendung der bisherigen Definitionen zu zeigen, bestimmen wir beispielhaft die Sorten einiger Typausdrücke und prüfen, ob es sich um einen Typ oder ein Constraint handelt:

Beispiel 4.6 (Sorten von Typausdrücken)**.**
Wir gehen in diesem Beispiel davon aus, dass der Typkonstruktor `Maybe` *von der Sorte* $* \to *$, `Int` *von der Sorte* $*$, `Functor` *von der Sorte* $(* \to *) \to$ Constraint, `Eq` *von der Sorte* $* \to$ Constraint *und zudem die Typvariable* `a` *von der Sorte* $* \to *$ *ist. Nun können wir die Sorten der folgenden Typausdrücke bestimmen:*

- `Maybe Int` *ist von der Sorte* $*$ *und somit ein Typ.*
- `Maybe a` *kann keine Sorte zugeordnet werden und ist somit kein sortenkorrekter Typausdruck.*

- `Functor Maybe` *ist von der Sorte* Constraint *und somit ein Constraint, aber weder regulär noch einfach.*
- `Functor a` *ist von der Sorte* Constraint *und ein reguläres sowie einfaches Constraint.*
- `Eq (a Int)` *ist von der Sorte* Constraint *und ein reguläres aber nicht einfaches Constraint.*

Für Mengen von Constraints führen wir den Begriff *Kontext* formal ein, bei dem wir nur aus regulären bzw. einfachen Constraints bestehende Kontexte besonders kennzeichnen:

Definition 4.7 (Kontext)**.**
Ein Kontext *hat die allgemeine Form*

$(C_1, \ldots, C_n)$

Dabei sind $n \in \mathbb{N}_0$ und $C_1, \ldots, C_n$ Constraints. Wir bezeichnen einen Kontext als regulär*, wenn C_i für alle $i \in \{1, \ldots, n\}$ regulär ist, und als* einfach*, wenn C_i für alle $i \in \{1, \ldots, n\}$ einfach ist.*

Für einen Kontext wird in Curry die Tupelschreibweise verwendet, im Verhalten entspricht ein Kontext aber einer Menge von Constraints. Deswegen werden wir stellenweise die Mengennotation für Kontexte verwenden und auch Mengenoperationen auf Kontexten anwenden.

Mit Kontexten und Typen lassen sich *beschränkte Typen* wie folgt definieren:

Definition 4.8 (Beschränkter Typ)**.**
Ein beschränkter Typ *hat die allgemeine Form*

K `=>` T

Dabei ist K ein Kontext und T ein Typ. Ist die Erweiterung zu flexiblen Kontexten nicht aktiviert, muss K regulär sein. Für beschränkte Typen der Form `()` `=>` *T wird alternativ auch die Notation T verwendet.*

Ein beschränkter Typ wird in einer Typsignatur oder Typannotation verwendet, um den Typ einer Funktion oder eines Ausdrucks anzugeben. Zu beachten ist, dass die Einschränkung auf reguläre Kontexte auch in inferierten, also nicht explizit angegebenen Funktionstypen besteht, wenn die Erweiterung zu flexiblen Kontexten nicht aktiviert ist.

Als nächstes definieren wir eine Funktion zur Bestimmung aller Typvariablen der bereits definierten Sprachelemente:

Definition 4.9 (Typvariablen von Typausdrücken, Kontexten und beschränkten Typen)**.**
Für eine Typvariable u, einen Typkonstruktor TC, Typausdrücke t_1, t_2, $n \in \mathbb{N}_0$, Constraints $C_1, \ldots, C_n$, einen Kontext K und einen Typ T sind die Typvariablen eines Typausdrucks, Kontexts und beschränkten Typs über die Funktion TV *wie folgt definiert:*

$$\begin{aligned}
\mathrm{TV}(u) &= \{u\} \\
\mathrm{TV}(TC) &= \varnothing \\
\mathrm{TV}(t_1\ t_2) &= \mathrm{TV}(t_1) \cup \mathrm{TV}(t_2) \\
\mathrm{TV}((C_1, \ldots, C_n)) &= \mathrm{TV}(C_1) \cup \cdots \cup \mathrm{TV}(C_n) \\
\mathrm{TV}(K \texttt{ => } T) &= \mathrm{TV}(K) \cup \mathrm{TV}(T)
\end{aligned}$$

Die Funktion TV_M *ist eine Multimengen-Variante von* TV *und analog definiert, wobei als Vereinigungsoperation $\uplus$, also die Summe der Elementanzahlen der vereinigten Mengen verwendet wird.*

Nachdem wir beschränkte Typen eingeführt und die Typvariablen zu einigen Sprachelementen bestimmt haben, können wir *Typschemata* definieren:

Definition 4.10 (Typschema)**.**
Ein Typschema *hat die allgemeine Form*

$$\forall\ u_1\ \ldots\ u_n\ .\ CT$$

Dabei sind CT ein beschränkter Typ, $n \in \mathbb{N}_0$ *und* $u_1, \ldots, u_n \in \mathrm{TV}(CT)$.

Beschränkte Typen können durch die Allquantifizierung aller vorkommenden Typvariablen zu Typschemata umgewandelt werden. Der Grund, warum mit Typschemata trotzdem ein eigener Begriff eingeführt wird, liegt darin, dass inferierte beschränkte Typen *freie Typvariablen* enthalten können. Damit sind Typvariablen gemeint, die in einer lokalen Deklaration genauso belegt werden müssen wie in einer übergeordneten Deklaration. In der lokalen Deklaration sind diese Typvariablen frei und dürfen nicht allquantifiziert werden, was durch ein Typschema ausgedrückt werden kann.

Zuletzt wollen wir *Substitutionen* definieren, mit denen wir Typvariablen in Typausdrücken durch andere Typausdrücke ersetzen können:

Definition 4.11 (Substitution)**.**
Eine Substitution *ist eine Abbildung von Typvariablen auf Typausdrücke der Form*

$$\{u_1 \mapsto t_1, \ldots, u_n \mapsto t_n\}$$

Dabei sind $n \in \mathbb{N}_0$, $u_1, \ldots, u_n$ *unterschiedliche Typvariablen und* $t_1, \ldots, t_n$ *Typausdrücke.*
Eine Substitution σ *kann wie folgt auf Typausdrücke, Kontexte und beschränkte Typen angewandt werden:*

$$\begin{aligned}
\sigma(u) &= \begin{cases} t & \text{wenn } (u \mapsto t) \in \sigma \\ u & \text{sonst} \end{cases} \\
\sigma(TC) &= TC \\
\sigma(t_1\ t_2) &= \sigma(t_1)\ \sigma(t_2) \\
\sigma((C_1,\ \ldots,\ C_n)) &= (\sigma(C_1),\ \ldots,\ \sigma(C_n)) \\
\sigma(K \texttt{ => } T) &= \sigma(K) \texttt{ => } \sigma(T)
\end{aligned}$$

Dabei sind u eine Typvariable, TC ein Typkonstruktor, t, t_1 *und* t_2 *Typausdrücke,* $n \in \mathbb{N}_0$, $C_1, \ldots, C_n$ *Constraints, K ein Kontext und T ein Typ.*
Wir bezeichnen mit $\mathrm{dom}(\sigma)$ *die Domäne von* σ*, die folgendermaßen definiert ist:*

$$\mathrm{dom}(\sigma) = \{v \mid v \text{ ist Typvariable} \wedge \exists \text{ Typausdruck } \tau : (v \mapsto \tau) \in \sigma\}$$

4.2. Allgemeiner Aufbau von Klassen- und Instanzdeklarationen

In diesem Abschnitt werden wir die allgemeine Form von Klassen- und Instanzdeklarationen beschreiben. Wir beginnen mit dem Aufbau von Klassendeklarationen:

Definition 4.12 (Klassendeklaration)**.**
Eine Klassendeklaration hat die allgemeine Form

class K **=>** $C\ u_1\ \ldots\ u_m$ **|** $F_1,\ \ldots,\ F_n$ **where**
  $method_1$ :: K_1 **=>** T_1
  ...
  $method_p$ :: K_p **=>** T_p
  Standardimplementierung von $method_{j_1}$
  ...
  Standardimplementierung von $method_{j_q}$

Dabei sind

- $m, n, p, q \in \mathbb{N}_0$ *und* $j_1, \ldots, j_q \in \{1, \ldots, p\}$;
- C *ein Klassenname und* $u_1, \ldots, u_m$ *(die Klassenparameter) unterschiedliche Typvariablen;*
- K *ein Kontext mit* $\mathrm{TV}(K) \subseteq \{u_1, \ldots, u_m\}$;
- $F_1, \ldots, F_n$ *funktionale Abhängigkeiten, deren Form später beschrieben wird;*
- $method_1, \ldots, method_p$ *Funktionsnamen,* $K_1, \ldots, K_p$ *Kontexte und* $T_1, \ldots, T_p$ *Typen.*

Weiterhin ist zu beachten, dass

- K *einfach und* $K_1, \ldots, K_p$ *regulär sein müssen, wenn die Erweiterung zu flexiblen Kontexten nicht aktiviert ist;*
- $m = 1$ *sein muss, wenn die Erweiterung zu Multiparametertypklassen nicht aktiviert ist;*
- $n = 0$ *sein muss, wenn die Erweiterung zu funktionalen Abhängigkeiten nicht aktiviert ist;*
- *der senkrechte Strich vor den funktionalen Abhängigkeiten entfällt, wenn* $n = 0$ *ist.*

Für alle $i \in \{1, \ldots, p\}$ *hat die funktionale Abhängigkeit* F_i *die Form*

$l_{(i,1)} \ \ldots \ l_{(i,lmax_i)}$ `->` $r_{(i,1)} \ \ldots \ r_{(i,rmax_i)}$

Dabei sind $lmax_i, rmax_i \in \mathbb{N}_0$, $l_{(i,li)} \in \{u_1, \ldots, u_n\}$ *für alle* $li \in \{1, \ldots, lmax_i\}$ *und* $r_{(i,ri)} \in \{u_1, \ldots, u_n\}$ *für alle* $ri \in \{1, \ldots, rmax_i\}$.

Diese Klassendeklaration führt eine Klasse C mit den Klassenparametern $u_1, \ldots, u_m$, den durch K gegebenen Oberklassen und den funktionalen Abhängigkeiten $F_1, \ldots, F_p$ ein. C ist dabei auch ein Typkonstruktor von der Sorte $s_1 \to \cdots \to s_n \to \mathsf{Constraint}$, wobei s_k die durch Sorteninferenz bestimmte Sorte von u_k für alle $k \in \{1, \ldots, m\}$ ist. Die Deklaration führt zudem für alle $h \in \{1, \ldots, p\}$ die Funktion $method_h$ ein, deren vollständiger beschränkter Typ noch um das *implizite Klassenconstraint* erweitert werden muss. Dadurch hat $method_h$ den Typ $(\{C \ u_1 \ \ldots \ u_m\} \cup K_h)$ `=>` T_h. Diesen Typ muss auch die Standardimplementierung von $method_h$ haben, falls sie in der Klassendeklaration angegeben ist.

Die Instanzdeklarationen, die zu einer solchen Klasse angegeben werden können, haben den folgenden allgemeinen Aufbau:

Definition 4.13 (Instanzdeklaration)**.**
Zu einer Klasse C *von der Sorte* $s_1 \to \cdots \to s_m \to \mathsf{Constraint}$ *mit Klassenparametern* $u_1, \ldots u_m$ *und Klassenmethoden* $method_1, \ldots method_p$*, wobei* $m, p \in \mathbb{N}_0$ *und* $s_1, \ldots, s_m$ *Sorten sind, hat eine Instanzdeklaration die allgemeine Form*

`instance` K **`=>`** $C \ t_1 \ \ldots \ t_m$ **`where`**
Implementierung von $method_{j_1}$
...
Implementierung von $method_{j_q}$

Dabei sind K *ein Kontext,* $t_1, \ldots, t_m$ *Typausdrücke,* $q \in \mathbb{N}_0$ *und* $j_1, \ldots, j_q \in \{1, \ldots, p\}$*. Außerdem muss* t_i *für alle* $i \in \{1, \ldots, m\}$ *von der Sorte* s_i *sein und die Implementierung von* $method_k$ *für alle* $k \in \{j_1, \ldots, j_q\}$ *den beschränkten Typ* $(K \cup \sigma(K_k))$ `=>` $\sigma(T_k)$ *haben, wobei die Substitution* $\sigma = \{u_1 \mapsto t_1, \ldots, u_m \mapsto t_m\}$ *und* K_k `=>` T_k *der beschränkte Typ von* $method_k$ *exklusive implizitem Klassenconstraint ist.*

Weiterhin ist zu beachten, dass

- K *einfach sein muss, wenn die Erweiterung zu flexiblen Kontexten nicht aktiviert ist;*
- $m = 1$ *sein muss, wenn die Erweiterung zu Multiparametertypklassen nicht aktiviert ist;*

- *t_i für alle $i \in \{1, \ldots, m\}$ die folgende Form haben muss, wenn die Erweiterung zu flexiblen Instanzen nicht aktiviert ist:*

  $TC_i \;\; v_{(i,1)} \;\; \ldots \;\; v_{(i,n_i)}$

  Dabei sind TC_i ein zu einem Datentyp gehörender Typkonstruktor, $n_i \in \mathbb{N}_0$ und $v_{(i,1)}, \ldots, v_{(i,n_i)}$ unterschiedliche Typvariablen.

Diese Instanzdeklaration führt eine Instanz der Klasse C für die *Instanztypen* $t_1, \ldots, t_m$ ein. In der Deklaration bezeichnen wir K als *Instanzkontext* und das Constraint $C \;\; t_1 \;\; \ldots \;\; t_m$ als *Instanzkopf*.

4.3. Kontextvereinfachung

Die *Kontextvereinfachung*, die wir in diesem Abschnitt behandeln, ist ein Prozess, bei dem eine Menge benötigter Constraints zu einer weitestgehend gleichwertigen Constraintmenge vereinfacht wird. Die Kontextvereinfachung wird einerseits dazu verwendet, besser lesbare Typen für Funktionen zu inferieren, dient andererseits aber auch der Prüfung, ob alle benötigten Constraints durch gegebene Instanzen und Constraints erfüllt werden können. Im Folgenden werden wir die zwei wesentlichen Schritte der Kontextvereinfachung beschreiben, die wir *Kontextreduktion* und *Kontextimplikation* nennen. Um diese Schritte möglichst unkompliziert definieren zu können, stellen wir zunächst jedoch Grundannahmen für diesen Abschnitt auf:

- In den Typausdrücken, die wir in diesem Abschnitt behandeln, müssen alle vorkommenden Typsynonyme durch die Typausdrücke, über die sie definiert sind, ersetzt worden sein. Dies bedeutet, dass z. B. alle Vorkommen von `String` durch `[Char]` ersetzt werden müssen. Diese Annahme ermöglicht es uns, die einfache strukturelle Gleichheit für Typausdrücke zu verwenden.

- Wir gehen davon aus, dass in Klassen- und Instanzdeklarationen nur frische Typvariablen vorkommen, es also keine Schnittmenge zwischen den Typvariablen aus der Klasse bzw. Instanz und den ansonsten in der jeweiligen Situation verwendeten Typvariablen gibt. Diese Eigenschaft wird in der Praxis durch Variablenumbenennung erreicht, durch die Grundannahme frischer Variablen ersparen wir uns im weiteren Verlauf aber die explizite Angabe einer solchen Umbenennung.

Wir betrachten zunächst die Kontextreduktion, bei der Constraints in einem Kontext, für die eine passende Instanz existiert, durch den entsprechenden Instanzkontext ersetzt werden. Außerdem wird bei der Kontextreduktion eine Menge gegebener Constraints beachtet, die ebenfalls zur Reduktion von Constraints eingesetzt wird. Die Kontextreduktion ist formal über die folgende Funktion definiert:

Definition 4.14 (Kontextreduktion)**.**
Für $n \in \mathbb{N}_0$, Constraints $C, C_1, \ldots, C_n$ und eine Menge gegebener Constraints M ist die Funktion RED *zur Reduktion von Constraints und Kontexten folgendermaßen definiert:*

$$\mathrm{RED}(\{C_1, \ldots, C_n\}, M) = \mathrm{RED}(C_1, M) \cup \cdots \cup \mathrm{RED}(C_n, M)$$

$$\mathrm{RED}(C, M) = \begin{cases} \varnothing & \text{wenn } C \in M \\ \mathrm{RED}(\sigma(K), M) & \text{sonst, wenn eine Substitution } \sigma \text{ und ein Kontext } K \text{ existieren, für die gelten:} \\ & (1)\ \text{Es gibt eine Instanz mit Kontext } K \text{ und Kopf } H, \text{ sodass } \sigma(H) = C \\ & (2)\ |\{H' \mid H' \text{ ist Kopf einer Instanz} \wedge \exists \text{ Substitution } \theta : \theta(H') = \theta(C)\}| = 1 \\ \{C\} & \text{sonst} \end{cases}$$

Zudem definieren wir die Funktion RED_S*, die analog zu* RED *ist, mit der einzigen Ausnahme, dass die in obiger Definition angegebene Bedingung* (2) *durch folgende Bedingung* $(2')$ *ersetzt wird:*

$(2')$ $|\{H' \mid H' \text{ ist Kopf einer Instanz} \wedge \exists \text{ Substitution } \theta : \theta(H') = C\}| = 1$

In dieser Definition sollte die Bedingung (2) besonders beachtet werden, welche dafür sorgt, dass Constraints nicht reduziert werden, wenn mehrere Instanzen potenziell zu ihnen passen. Zur Veranschaulichung dieser Bedingung betrachten wir die einstellige Typklasse `C` und zwei Instanzen mit den Köpfen `C a` und `C Bool`. Wenn nun das Constraint `C b` reduziert werden soll, erfüllt nur die erste Instanz mit der Substitution $\{\texttt{a} \mapsto \texttt{b}\}$ Bedingung (1) und ist somit *passend*. In Bedingung (2) wird die Substitution aber auch auf das Constraint angewandt, sodass hier mit der Substitution $\{\texttt{b} \mapsto \texttt{Bool}\}$ auch der zweite Instanzkopf Teil der definierten Menge ist. Diese Instanz ist somit abhängig von der Belegung von `b` *potenziell passend*, weswegen das Constraint nicht durch RED reduziert wird.

Mit der abgewandelten Bedingung $(2')$ aus der Funktion RED_S werden hingegen nur tatsächlich passende Instanzen berücksichtigt. In unserem Beispiel würde das Constraint `C b` also durch RED_S mit der passenden Instanz reduziert werden. Wir führen mit RED und RED_S zwei verschiedene Funktionen zur Reduktion von Kontexten ein, da erstere Funktion bei der Typinferenz und -prüfung von Funktionsdeklarationen angewandt wird, während die letztere Funktion bei der Prüfung von Instanzdeklarationen zum Einsatz kommt, wie in Einschränkung 4.30 beschrieben wird. Diese unterschiedliche Behandlung entspricht dem Verhalten des GHC.

Außerdem wollen wir kurz beleuchten, warum die Menge gegebener Constraints bei der Kontextreduktion berücksichtigt werden muss: Hierzu betrachten wir eine einstellige Klasse `D`, die Instanz `instance D a => D [a]` und eine Funktion mit dem explizit angegebenen Kontext `D [Bool]`. Wenn genau dieses Constraint im Körper der Funktion benötigt wird, würde es ohne Berücksichtigung der gegebenen Constraints zu `D Bool` reduziert werden. Dieses Constraint ist allerdings nicht im angegebenen Kontext enthalten und falls keine Instanz zu dem Constraint passt, würde ein Typfehler gemeldet werden. Ein weiteres mögliches Problem, das wir an diesem Beispiel erkennen können, ist das Entstehen eines Constraints `D [b]` im Körper der angesprochenen Funktion. Dieses Constraint würde durch RED zu `D b` reduziert werden. Im Verlauf der Typinferenz könnte `b` allerdings zu `Bool` instanziiert werden, wodurch sich wieder das angesprochene Problem ergibt. Aus diesem Grund wird die Kontextreduktion erst dann auf einen Kontext angewandt, wenn die Typinferenz für die enthaltenen Typvariablen bereits abgeschlossen ist.

Als zweiten Schritt der Kontextvereinfachung betrachten wir die Kontextimplikation, über die bestimmt wird, welche Kontexte durch angegebene Oberklassen von einem Kontext impliziert werden. Die Kontextimplikation [JJM97] ist folgendermaßen definiert:

Definition 4.15 (Kontextimplikation)**.**
Die Folgerungsrelation $\Vdash$ *beschreibt die Implikation von Kontexten und ist über folgende Inferenzregeln definiert:*

$$\frac{Q \subseteq P}{P \Vdash Q} \quad (\textit{mono})$$

$$\frac{P \Vdash Q \quad Q \Vdash R}{P \Vdash R} \quad (\textit{trans})$$

$$\frac{P \Vdash Q \quad P \Vdash R}{P \Vdash Q \cup R} \quad (\textit{union})$$

$$\frac{\{u_1, \ldots, u_m\} \subseteq \text{dom}(\sigma) \quad \text{Existenz von } \texttt{class } K \texttt{ => } C\ u_1\ \ldots\ u_m\ \ldots}{\{\sigma(C\ u_1\ \ldots\ u_m)\} \Vdash \sigma(K)} \quad (\textit{super})$$

Dabei sind P*,* Q*,* R *und* K *Kontexte,* C *ein Klassenname,* $m \in \mathbb{N}_0$*,* $u_1, \ldots, u_m$ *Typvariablen und* σ *eine Substitution.*

Im Gegensatz zu der Kontextimplikation, die von Simon Peyton Jones, Mark P. Jones und Erik Meijer vorgestellt wurde [JJM97], verwenden wir keine Instanzdeklarationen zur Implikation weiterer Constraints, sondern behandeln Instanzen getrennt in der Kontextreduktion. Grund hierfür ist, dass durch die Kontextimplikation Constraints eingeführt werden können, zu denen mehrere Instanzen passen. Wenn auf dieser Basis weitere Constraints durch Instanzen impliziert werden könnten, wäre nicht mehr sichergestellt, dass die implizierten Constraints eindeutig auf den gegebenen Kontext zurückgeführt werden können.

Mit der Kontextimplikation können wir Notationen für den so weit wie möglich erweiterten bzw. vereinfachten Kontext einführen:

Definition 4.16 (Maximaler und minimaler Kontext)**.**
Zu einem Kontext K bezeichnen wir mit K_+ den maximalen von K implizierten Kontext, also dass $K \Vdash K_+$ und $K' \subseteq K_+$ für alle Kontexte K' mit $K \Vdash K'$ ist.
Zu einem Kontext K bezeichnen wir mit K_- den minimalen Kontext, der K impliziert, also dass $K_- \Vdash K$ und $K_- \subseteq K$, aber $K \nsubseteq K'_+$ für alle Kontexte K' mit $K' \subset K_-$ ist.

Die bislang eingeführten Prozesse und Notationen können wir nun kombinieren, um die vollständige Kontextvereinfachung und ihre Ergebnisse zu beschreiben:

Definition 4.17 (Ergebnis der Kontextvereinfachung)**.**
Die Kontextvereinfachung wird durchgeführt, wenn einer Deklaration oder einem explizit getypten Ausdruck ein Typschema zugewiesen werden soll. Hierzu führen wir zunächst die folgenden Mengen ein:

- *P ist die durch Typinferenz bestimmte Menge aller Constraints, die in der Deklaration oder dem Ausdruck benötigt werden.*
- *F ist die Menge freier Typvariablen, also die Menge der in übergeordneten Deklarationen und Ausdrücken gebundenen Typvariablen.*
- *$P_L := \{C \in P \mid \mathrm{TV}(C) \cap F = \varnothing\}$ ist die Menge aller Constraints aus P, die keine freien Typvariablen beinhalten.*
- *$P_G := \{C \in P \mid \mathrm{TV}(C) \neq \varnothing \wedge \mathrm{TV}(C) \subseteq F\}$ ist die Menge aller Constraints aus P, die mindestens eine und ausschließlich freie Typvariablen beinhalten.*
- *$P_R := P \setminus (P_L \cup P_G)$ ist die Menge aller Constraints aus P, die freie und nicht-freie Typvariablen beinhalten.*
- *E ist die Menge aller Constraints, die in übergeordneten Deklarationen und Ausdrücken in Kontexten explizit angegeben wurden. Wir gehen davon aus, dass die darin vorkommenden Typvariablen so umbenannt wurden, dass Variablen nur dann den gleichen Namen haben, wenn sie für identische Typen stehen.*

Für eine Deklaration oder einen Ausdruck mit explizit angegebenem beschränkten Typ K `=>` T ist die Kontextvereinfachung erfolgreich, wenn $\mathrm{RED}(P_L, (E \cup K)_+) = \varnothing$ ist. Der Deklaration bzw. dem Ausdruck kann das Typschema $\forall\ u_1\ \ldots\ u_n\ .\ K$ `=>` T zugewiesen werden, wobei $\{u_1, \ldots, u_n\} = \mathrm{TV}(K$ `=>` $T)$ für ein $n \in \mathbb{N}_0$ und Typvariablen $u_1, \ldots, u_n$ ist.

Für eine Deklaration ohne Typsignatur ist die Kontextvereinfachung erfolgreich, wenn $\{C \in K \mid \mathrm{TV}(C) = \varnothing\} = \varnothing$ für den Kontext $K := \mathrm{RED}(P_L, E_+)_- \cup P_R$ gilt. Außerdem muss K regulär oder die Spracherweiterung zu flexiblen Kontexten aktiviert sein. Wenn für die Deklaration der Typ T inferiert wurde, kann ihr das Typschema $\forall\ u_1\ \ldots\ u_n\ .\ K$ `=>` T zugewiesen werden, wobei $\{u_1, \ldots, u_n\} = \mathrm{TV}(K$ `=>` $T) \setminus F$ für ein $n \in \mathbb{N}_0$ und Typvariablen $u_1, \ldots, u_n$ ist.

In beiden Fällen wird P_G und im ersten Fall auch P_R zu der Menge benötigter Constraints für die übergeordnete Deklaration bzw. den übergeordneten explizit getypten Ausdruck hinzugefügt.

Ist die Kontextreduktion für eine Deklaration oder einen explizit getypten Ausdruck in einem Programm nicht erfolgreich, wird das Programm zurückgewiesen.

Zur abschließenden Veranschaulichung der in diesem Abschnitt eingeführten Definitionen wenden wir die Kontextvereinfachung in einem Beispiel an:

Beispiel 4.18 (Kontextvereinfachung)**.**
Es seien die anschließenden Klassen- und Instanzdeklarationen gegeben:

(1) `class C a`
(2) `class C a => D a b`
(3) `instance C Bool`
(4) `instance (C a, C b) => C (a, b)`
(5) `instance D a b => D [a] b`

Wir betrachten nun die Kontextvereinfachung für eine Deklaration ohne Typsignatur, bei der die in Definition 4.17 beschriebenen Mengen wie folgt belegt sind:

- $P = P_L = \{\texttt{C (c, Bool)}, \texttt{D [c] Bool}\}$
- $F = P_G = P_R = E = \varnothing$

Nun können wir durch Kontextreduktion die Menge K berechnen (die zu einem Constraint passenden Instanzen sind mit der entsprechenden Substitution jeweils angegeben):

$$\begin{aligned}
K = \mathrm{RED}(P_L, E_+) &= \mathrm{RED}(\{\texttt{C (c, Bool)}, \texttt{D [c] Bool}\}, \varnothing) \\
&= \overbrace{\mathrm{RED}(\texttt{C (c, Bool)}, \varnothing)}^{(4)\text{ mit }\{\texttt{a} \mapsto \texttt{c}, \texttt{b} \mapsto \texttt{Bool}\}} \cup \overbrace{\mathrm{RED}(\texttt{D [c] Bool}, \varnothing)}^{(5)\text{ mit }\{\texttt{a} \mapsto \texttt{c}, \texttt{b} \mapsto \texttt{Bool}\}} \\
&= \mathrm{RED}(\{\texttt{C c}, \texttt{C Bool}\}, \varnothing) \cup \mathrm{RED}(\{\texttt{D c Bool}\}, \varnothing) \\
&= \mathrm{RED}(\texttt{C c}, \varnothing) \cup \overbrace{\mathrm{RED}(\texttt{C Bool}, \varnothing)}^{(3)\text{ mit }\{\}} \cup \mathrm{RED}(\texttt{D c Bool}, \varnothing) \\
&= \{\texttt{C c}\} \cup \mathrm{RED}(\varnothing, \varnothing) \cup \{\texttt{D c Bool}\} \\
&= \{\texttt{C c}\} \cup \varnothing \cup \{\texttt{D c Bool}\} \\
&= \{\texttt{C c}, \texttt{D c Bool}\}
\end{aligned}$$

Zur Minimierung des Kontextes K wenden wir als Nächstes die Kontextimplikation an:

$$\dfrac{\dfrac{\{\texttt{D c Bool}\} \subseteq \{\texttt{D c Bool}\}}{\{\texttt{D c Bool}\} \Vdash \{\texttt{D c Bool}\}}(\textit{mono}) \quad \dfrac{\{\texttt{a}, \texttt{b}\} \subseteq \mathrm{dom}(\{\texttt{a} \mapsto \texttt{c}, \texttt{b} \mapsto \texttt{Bool}\}) \quad \text{Existenz von (2)}}{\{\texttt{D c Bool}\} \Vdash \{\texttt{C c}\}}(\textit{super})}{\{\texttt{D c Bool}\} \Vdash \{\texttt{C c}, \texttt{D c Bool}\}}(\textit{union})$$

Da $\{\} \not\Vdash \{\texttt{C c}, \texttt{D c Bool}\}$*, ist* $\{\texttt{D c Bool}\}$ *der minimale Kontext, der K impliziert, also ist* $K_\_ = K' = \{\texttt{D c Bool}\}$*. Dieser Kontext ist nach Definition 4.7 nicht regulär, also ist die Kontextvereinfachung nur dann mit K′ als Ergebnis erfolgreich, wenn die Erweiterung zu flexiblen Kontexten aktiviert ist.*

4.4. Einschränkungen für Typklassenelemente

In diesem Abschnitt führen wir Einschränkungen für die in den Abschnitten 4.1 und 4.2 eingeführten Typklassenelemente ein, deren Einhaltung während der Kompilation eines Curry-Programms geprüft werden muss. Diese Einschränkungen dienen einerseits dazu, Voraussetzungen zu schaffen, unter denen korrektes Verhalten und die sichergestellte Terminierung der Kompilation auch mit einer weniger komplexen Implementierung erreicht werden kann. Andererseits soll der Programmierer durch die Zurückweisung von Programmen, welche diese Einschränkungen nicht einhalten, möglichst frühzeitig auf Typklassenelemente hingewiesen werden, die bei ihrer Anwendung im späteren Programmverlauf

Probleme verursachen.

Wir betrachten zuerst in Abschnitt 4.4.1 die Einschränkungen, die für Klassenmethoden und andere Funktionen gelten. Abschnitt 4.4.2 zeigt dann Situationen auf, in denen die Kontextvereinfachung nicht terminiert und stellt Bedingungen vor, unter denen diese Probleme nicht auftreten können. Zuletzt formulieren wir in Abschnitt 4.4.3 weitere Einschränkungen für Instanzdeklarationen. Zudem gehen wir wie schon in Abschnitt 4.3 in diesem Abschnitt davon aus, dass Typausdrücke keine Typsynonyme enthalten.

4.4.1. Klassenmethoden und andere Funktionen

Dieser Abschnitt behandelt die Einschränkungen, die an die Typen von Klassenmethoden und anderen Funktionen gestellt werden.

Zuerst betrachten wir hierzu die für Klassenmethoden von regulären Typklassen (mit einem Klassenparameter) geltende Einschränkung, dass der Kontext einer Klassenmethode den Klassenparameter nicht weiter einschränken darf. Diese Einschränkung wird durch die Möglichkeit beliebig vieler Klassenparameter und beliebig vieler Typvariablen in Constraints wie folgt verallgemeinert:

Einschränkung 4.19 (Beschränkte Klassenparameter)**.**
Für jede Klassenmethode mit Funktionsnamen method, für die eine Klassendeklaration der Form

```
class ... => C u_1 ... u_m ... where
  ...
  method :: K => ...
  ...
```

mit einem Klassennamen C, $m \in \mathbb{N}_0$ *und Typvariablen* $u_1, \ldots, u_m$ *existiert, muss für alle Constraints* $c \in K$ *gelten:* $\mathrm{TV}(c) = \varnothing$ *oder* $\mathrm{TV}(c) \setminus \{u_1, \ldots, u_m\} \neq \varnothing$

Ein Constraint einer Klassenmethode muss also entweder gar keine Typvariablen oder mindestens eine abseits der Klassenparameter enthalten.

Eine weitere Einschränkung für Klassenmethoden, die in Abschnitt 3.3.1 motiviert wurde, ist das Verbot mehrdeutiger Typvariablen. Diese Variablen können allerdings nicht nur bei Klassenmethoden, sondern auch bei anderen Funktionen auftreten: In der Regel sind Typvariablen mehrdeutig, wenn sie im Kontext eines beschränkten Typs, aber nicht im eigentlichen Typ vorkommen. Der Spezialfall von Klassenmethoden, bei denen nicht im Typ der Methode vorkommende Klassenparameter mehrdeutig sind, wird hierbei durch Betrachtung des vollständigen beschränkten Typs mit implizitem Klassenconstraint abgedeckt.

Zudem muss beachtet werden, dass die sonst mehrdeutigen Typvariablen, die nur im Kontext eines beschränkten Typs vorkommen, durch funktionale Abhängigkeiten eindeutig bestimmt sein können. Um zu ermitteln, welche Typvariablen auf diese Art abgedeckt sind, kann der Algorithmus 4.20 verwendet werden.

Dieser Algorithmus durchläuft alle Constraints im gegebenen Kontext K und alle funktionalen Abhängigkeiten der Klasse des Constraints. Wenn alle im Constraint auf der linken Seite der funktionalen Abhängigkeit vorkommenden Typvariablen bereits durch die Typvariablenmenge V abgedeckt sind, werden die entsprechend auf der rechten Seite vorkommenden Typvariablen zu V' hinzugefügt. Weil durch eine solche Ergänzung möglicherweise weitere funktionale Abhängigkeiten auf diese Art angewandt werden können, ruft der Algorithmus sich rekursiv selbst auf, wenn sich die Menge der abgedeckten Typvariablen verändert hat.

Algorithmus 4.20 (Abgedeckte Typvariablen)**.**
Die Funktion COV *ist über den folgenden Algorithmus definiert:*

Eingabe: Kontext K, Menge bereits abgedeckter Typvariablen V
Ausgabe: Menge von V durch funktionale Abhängigkeiten in K abgedeckte Typvariablen V'
$V' \leftarrow V$
`for each` $(C\ t_1\ \ldots\ t_m) \in K$
  Betrachte Klassendeklaration **`class`** $\ldots$ `=>` $C\ u_1\ \ldots\ u_m$ `|` F_1`, ...,` $F_n\ \ldots$
  `for each` $(l_1\ \ldots\ l_{lmax}$ `->` $r_1\ \ldots\ r_{rmax}) \in \{F_1, \ldots, F_n\}$
    `if` $\{v \in \mathrm{TV}(t_i) \mid i \in \{1, \ldots, m\} \wedge u_i \in \{l_1, \ldots, l_{lmax}\}\} \subseteq V$
      `then` $V' \leftarrow V' \cup \{v \in \mathrm{TV}(t_i) \mid i \in \{1, \ldots, m\} \wedge u_i \in \{r_1, \ldots, r_{rmax}\}\}$
`if` $V' \neq V$
  `then` $V' \leftarrow \mathrm{COV}(K, V')$

Den vorgestellten Algorithmus können wir nutzen, um mehrdeutige Typvariablen genau zu definieren:

Einschränkung 4.21 (Mehrdeutige Typvariablen in Typschemata)**.**
Für ein Typschema der Form $\forall\ u_1\ \ldots\ u_n\ .\ K$ `=>` T*, bei dem* $n \in \mathbb{N}_0$*,* $u_1, \ldots, u_n$ *Typvariablen,* K *ein Kontext und* T *ein Typ sind, muss für die Menge der mehrdeutigen Typvariablen*

$M := \{u_1, \ldots, u_n\} \setminus \mathrm{COV}(K, \mathrm{TV}(T))$

$M = \varnothing$ *gelten.*

Auf beschränkte Typen außerhalb eines Typschemas kann diese Einschränkung analog angewandt werden, wenn statt $\{u_1, \ldots, u_n\}$ die Menge $\mathrm{TV}(K)$ verwendet wird.

Welche beschränkten Typen mit den bislang eingeführten Einschränkungen gültig und welche ungültig sind, können wir anhand von Klassenmethoden an einem Beispiel veranschaulichen:

Beispiel 4.22 (Gültige und ungültige Klassenmethoden)**.**
Unter der Voraussetzung, dass die Klassendeklarationen

```
class C
class D a b | a -> b
```

existieren und die Erweiterung zu flexiblen Kontexten aktiviert ist, sind die anschließenden Klassenmethoden einer Klasse `E` *aus den im Kommentar angegebenen Gründen entweder gültig oder ungültig:*

```
class E a b c | a -> b where
  m1 :: a -> c               -- Gültig, da implizites Klassenconstraint 'b' abdeckt
  m2 :: b -> c               -- Ungültig, da 'a' nicht abgedeckt ist
  m3 :: Eq a => a -> c       -- Ungültig, da Constraint nur Klassenparameter beschränkt
  m4 :: C => a -> b -> c     -- Gültig, da Constraint keine Typvariablen beschränkt
  m5 :: D a (c, d) => a -> b -- Gültig, da Constraint nicht nur Klassenparameter
                             -- beschränkt und 'c' sowie 'd' abdeckt
```

4.4.2. Terminierungsbedingungen für die Kontextvereinfachung

Da die in Abschnitt 4.3 definierte Kontextvereinfachung Teil der Kompilation von Curry-Programmen ist, muss ihre Terminierung gewährleistet sein. Mit den bislang eingeführten Einschränkungen gibt es

allerdings für beide Schritte der Kontextvereinfachung, also sowohl für die Kontextimplikation als auch für die Kontextreduktion Fälle, in denen es zu einer nichtterminierenden Berechnung kommt.

Als erstes betrachten wir den in anschließendem Beispiel dargestellten Fall, bei dem unendlich viele Inferenzschritte mit der Kontextimplikation möglich sind und so ein unendlich großer Kontext impliziert wird:

Beispiel 4.23 (Unendlicher maximaler Kontext)**.**
Mit der Klassendeklaration `class C [a] => C a` *ist für den Kontext* $K := \{\texttt{C b}\}$ *der maximale Kontext* K_+ *eine unendliche Menge, wie folgende Kontextimplikation zeigt:*

$$\dfrac{\dfrac{}{\{\texttt{C b}\} \Vdash \{\texttt{C b}\}}\,(mono) \qquad \dfrac{\dfrac{\{\texttt{a} \mapsto \texttt{b}\}\ (super)}{\{\texttt{C b}\} \Vdash \{\texttt{C [b]}\}} \qquad \dfrac{\dfrac{\{\texttt{a} \mapsto \texttt{b}\}\ (super)}{\{\texttt{C b}\} \Vdash \{\texttt{C [b]}\}} \qquad \dfrac{\{\texttt{a} \mapsto \texttt{[b]}\}\ (super)}{\{\texttt{C [b]}\} \Vdash \{\texttt{C [[b]]}\}}\,(trans)}{\{\texttt{C b}\} \Vdash \{\texttt{C [[b]]}\}} \qquad \dfrac{\ldots}{\{\texttt{C b}\} \Vdash \ldots}}{\{\texttt{C b}\} \Vdash \{\texttt{C [[b]]}, \ldots\}}}{\{\texttt{C b}\} \Vdash \{\texttt{C [b]}, \texttt{C [[b]]}, \ldots\}}}{\{\texttt{C b}\} \Vdash \{\texttt{C b}, \texttt{C [b]}, \texttt{C [[b]]}, \ldots\}}$$

Hinweis: Aus Platzgründen ist nicht angegeben, wenn (*union*) *als Inferenzregel angewandt wurde, und die Angabe von* (*mono*) *und* (*super*) *erfolgt mit gekürzten Informationen über statt neben dem Querstrich.*

Wenn ein maximaler Kontext unendlich groß ist, kann es zu einer nichtterminierenden Berechnung kommen, wenn geprüft werden soll, ob ein Constraint Element des Kontextes ist. Um dies zu verhindern, müssen Oberklassen nach folgender Einschränkung azyklisch sein:

Einschränkung 4.24 (Azyklizität von Oberklassen)**.**
Für alle $n \in \mathbb{N}_+$ *und für jede Folge von Klassennamen* $C_1 C_2 \ldots C_n C_{n+1}$*, bei der für alle* $i \in \{1, \ldots, n\}$ *eine Klassendeklaration der Form*

`class (...,` C_{i+1} `..., ...) =>` C_i `...`

existiert, muss $C_1 \neq C_{n+1}$ *gelten.*

Jede Anwendung der Inferenzregel (*super*) stellt mit der Einschränkung einen Schritt nach vorne in einer solchen Folge von Klassennamen dar. Weil es nur endlich viele Typklassen und endlich viele Oberklassenconstraints pro Klasse geben kann, impliziert jedes Constraint und damit auch jeder Kontext nur einen endlichen Kontext.

Als nächstes kommen wir zur Terminierung der Kontextreduktion, wo wir ein ähnliches Problem wie bei der Kontextimplikation beobachten können:

Beispiel 4.25 (Nichtterminierende Kontextreduktion 1)**.**
Wir betrachten die folgenden Instanzdeklaration zur einstelligen Klasse C:

```
instance C [a] => C a
```

Das Constraint `C b` *wird ohne gegebene Constraints nun folgendermaßen reduziert:*

$$\overbrace{\mathrm{RED}(\texttt{C b}, \varnothing)}^{\{\texttt{a} \mapsto \texttt{b}\}} = \mathrm{RED}(\{\texttt{C [b]}\}, \varnothing) = \overbrace{\mathrm{RED}(\texttt{C [b]}, \varnothing)}^{\{\texttt{a} \mapsto \texttt{[b]}\}} = \mathrm{RED}(\{\texttt{C [[b]]}\}, \varnothing) = \overbrace{\mathrm{RED}(\texttt{C [[b]]}, \varnothing)}^{\{\texttt{a} \mapsto \texttt{[[b]]}\}} = \ldots$$

Um die Möglichkeit einer solchen nichtterminierenden Reduktion zu verhindern, wäre es anders als bei der Kontextimplikation aber nicht sinnvoll, Instanzen zu verbieten, bei denen der gleiche Klassenname in Kontext und Kopf vorkommt. Das Problem bei der im Beispiel angegebenen Instanz ist vielmehr, dass ein Constraint im Instanzkontext größer als der Instanzkopf ist. Diese Größe können wir über eine Definition genau bestimmen:

Definition 4.26 (Größe eines Typausdrucks)**.**
Für eine Typvariable u, einen Typkonstruktor TC und Typausdrücke t_1, t_2 ist die Größe eines Typausdrucks über die Funktion size *wie folgt definiert:*

$$\begin{aligned} \text{size}(u) &= 1 \\ \text{size}(TC) &= 1 \\ \text{size}(t_1\ \ t_2) &= \text{size}(t_1) + \text{size}(t_2) \end{aligned}$$

Eine Beschränkung der Größe von Constraints in Instanzkontexten ist allerdings noch nicht ausreichend, um die Terminierung der Kontextreduktion zu gewährleisten. Dies zeigt anschließendes Beispiel, bei dem in beiden betrachteten Instanzen die Constraints im Instanzkontext kleiner sind als der jeweilige Instanzkopf:

Beispiel 4.27 (Nichtterminierende Kontextreduktion 2)**.**
Wir betrachten die folgenden Instanzdeklarationen zur dreistelligen Klasse C und zweistelligen Klasse D:

(1) `instance D a a => C a b c`
(2) `instance C [a] b b => D a [[[b]]]`

Das Constraint `C [[[d]]] e e` *wird ohne gegebene Constraints nun folgendermaßen reduziert:*

$$\begin{aligned} &\overbrace{\text{RED}(\texttt{C [[[d]]] e e}, \varnothing)}^{(1)\text{ mit }\{\texttt{a}\mapsto\texttt{[[[d]]]}, \texttt{b}\mapsto\texttt{e}, \texttt{c}\mapsto\texttt{e}\}} && = \overbrace{\text{RED}(\texttt{D [[[d]]] [[[d]]]}, \varnothing)}^{(2)\text{ mit }\{\texttt{a}\mapsto\texttt{[[[d]]]}, \texttt{b}\mapsto\texttt{d}\}} \\ = &\overbrace{\text{RED}(\texttt{C [[[[d]]]] d d}, \varnothing)}^{(1)\text{ mit }\{\texttt{a}\mapsto\texttt{[[[[d]]]]}, \texttt{b}\mapsto\texttt{d}, \texttt{c}\mapsto\texttt{d}\}} && = \overbrace{\text{RED}(\texttt{D [[[[d]]]] [[[[d]]]]}, \varnothing)}^{(2)\text{ mit }\{\texttt{a}\mapsto\texttt{[[[[d]]]]}, \texttt{b}\mapsto\texttt{[d]}\}} \\ = &\overbrace{\text{RED}(\texttt{C [[[[[d]]]]] [d] [d]}, \varnothing)}^{(1)\text{ mit }\{\texttt{a}\mapsto\texttt{[[[[[d]]]]]}, \texttt{b}\mapsto\texttt{[d]}, \texttt{c}\mapsto\texttt{[d]}\}} && = \ldots \end{aligned}$$

Hinweis: Zur Abkürzung der Reduktion sind die Schritte, in denen RED *auf eine Menge von Constraints angewandt wird, nicht dargestellt.*

In diesem Beispiel ist das Problem, dass es Substitutionen gibt, unter denen die Constraints der Instanzkontexte größer als die die jeweiligen Instanzköpfe sind. So ist z. B. mit $\sigma = \{\texttt{a} \mapsto \texttt{[[d]]}\}$:

$$\begin{gathered} \text{size}(\texttt{D a a}) = 3 < 4 = \text{size}(\texttt{C a b c}) \\ \text{size}(\sigma(\texttt{D a a})) = \text{size}(\texttt{D [[d]] [[d]]}) = 7 > 6 = \text{size}(\texttt{C [[d]] b c}) = \text{size}(\sigma(\texttt{C a b c})) \end{gathered}$$

Diese Eigenschaft führt im Beispiel dazu, dass immer größere Constraints und eine nichtterminierende Berechnung bei der Kontextreduktion entstehen.

Als Lösung für dieses Problem kann die Beschränkung festgelegt werden, dass keine Typvariable häufiger in einem Constraint im Kontext einer Instanz als im Kopf dieser Instanz vorkommen darf. Zusammen mit der Einschränkung zur Größe von Constraints in Instanzkontexten ist diese Einschränkung Teil der *Paterson-Bedingungen* [SDPJ+07]. Die anschließend angegebene Variante dieser Bedingungen entspricht dem für Curry relevanten Teil der im GHC-Nutzerhandbuch angegebenen Regeln[1]:

[1] `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts/instances.html#instance-termination-rules`

Einschränkung 4.28 (Paterson-Bedingungen)**.**
In einer Instanzdeklaration der Form

```
instance (C_1, ..., C_n) => H where ...
```

mit einem Instanzkopf H, $n \in \mathbb{N}_0$ und Constraints $C_1, \ldots, C_n$, müssen für alle $i \in \{1, \ldots, n\}$ die folgenden Paterson-Bedingungen *gelten:*

$$\forall u \in \mathrm{TV}(C_i) : (\mathrm{TV}_\mathrm{M}(C_i))(u) \leqslant (\mathrm{TV}_\mathrm{M}(H))(u) \qquad \text{(1. Paterson-Bedingung)}$$

$$\mathrm{size}(C_i) < \mathrm{size}(H) \qquad \text{(2. Paterson-Bedingung)}$$

Bei TV_M handelt es sich um die Multimengen-Variante von TV (siehe Definition 4.9). Zudem muss beachtet werden, dass die Paterson-Bedingungen nur auf Typausdrücke ohne Typsynonyme angewandt werden dürfen, da sowohl die Größe eines Typausdrucks als auch die Anzahl der Auftritte einer Typvariable darin sich durch die Expansion der Typsynonyme verändern kann.

Mit den Paterson-Bedingungen ist die Terminierung der Kontextreduktion gewährleistet:

Satz 4.29 (Terminierung der Kontextreduktion)**.**
Sind die in Einschränkung 4.28 angegebenen Paterson-Bedingungen für ein Programm erfüllt, terminiert die Kontextreduktion[2] [SDPJ+07].

4.4.3. Weitere Einschränkungen für Instanzdeklarationen

Als letzten Teil der Einschränkungen für Typklassenelemente behandeln wir in diesem Abschnitt noch die Einschränkungen für Instanzdeklarationen, die nicht für die Terminierung der Kontextvereinfachung notwendig sind.

Die erste dieser Einschränkungen, die bereits in Abschnitt 3.1.2 kurz erwähnt wurde, ist die notwendige Existenz von Instanzen zu jedem Oberklassenconstraint der Klasse, zu der eine Instanz angelegt wird. Wichtig ist hierbei, dass nicht nur eine Oberklasseninstanz mit dem passenden Kopf existieren muss, sondern der Kontext dieser Oberklasseninstanz auch durch den Kontext der angelegten Instanz impliziert werden muss. Diese Bedingung wird durch die folgende Einschränkung ausgedrückt:

Einschränkung 4.30 (Existenz von Oberklasseninstanzen)**.**
Für alle Kontexte CK und IK, Klassennamen C, $m \in \mathbb{N}_0$, Typvariablen $u_1, \ldots, u_m$, Typausdrücke $t_1, \ldots, t_m$, jede Klassendeklaration der Form

```
class CK => C u_1 ... u_m | ... where ...
```

und jede Instanzdeklaration der Form

```
instance IK => C t_1 ... t_m where ...
```

muss $\mathrm{RED}_\mathrm{S}(\sigma(CK), IK_+) = \varnothing$ *mit* $\sigma = \{u_1 \mapsto t_1, \ldots, u_m \mapsto t_m\}$ *gelten.*

Bei RED_S handelt es sich um eine Variante von RED mit leicht abgewandelten Reduktionsregeln (siehe Definition 4.14).

Als letzte Einschränkung dieses Abschnitts definieren wir ein Verbot verschiedener Instanzen mit Instanzköpfen, die nach der Expansion von Typsynonymen bis auf Variablenumbenennung identisch sind. Solche Instanzen sorgen bei ihrer Anwendung zwangsläufig für einen Fehler bei der Kontextreduktion wegen überlappender Instanzen.

Einschränkung 4.31 (Doppelte Instanzköpfe)**.**
Für jedes Paar verschiedener Instanzen mit Instanzköpfen H_1, H_2 und alle Substitutionen σ, θ muss $\sigma(H_1) \neq H_2$ oder $H_1 \neq \theta(H_2)$ gelten.

[2] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts/instances.html#instance-termination-rules

4.5. Typinferenz mit funktionalen Abhängigkeiten

Funktionale Abhängigkeiten bieten, wie in Abschnitt 3.3.2 informell beschrieben, in bestimmten Fällen die Möglichkeit, speziellere Typen zu inferieren. Dieser Abschnitt stellt deswegen kurz vor, wie das Typsystem inklusive der Typinferenz in Curry grundsätzlich funktioniert, bevor beschrieben wird, wie die Typinferenz durch funktionale Abhängigkeiten erweitert wird. Zudem definieren wir in diesem Abschnitt, welche weiteren Einschränkungen Instanzdeklarationen erfüllen müssen, um alle funktionalen Abhängigkeiten einzuhalten. Hierzu gehen wir von den gleichen Grundannahmen aus, wie sie zu Beginn von Abschnitt 4.3 beschrieben wurden, dass es also keine Typsynonyme in Typausdrücken und nur frische Typvariablen in Instanzdeklarationen gibt.

Das Typsystem von Curry baut auf dem *Hindley-Milner-Typsystem* auf, welches unabhängig voneinander von Roger Hindley [Hin69] und Robin Milner [Mil78] vorgestellt wurde. Dieses Typsystem wurde später von Mark P. Jones um *Prädikate* und *qualifizierte Typen* erweitert [Jon92], womit eine theoretische Grundlage für ein Typsystem mit Unterstützung für Typklassen geschaffen wurde. Für unsere Zwecke entsprechen die Prädikate den in Definition 4.5 definierten Constraints und die qualifizierten Typen den in Definition 4.8 definierten beschränkten Typen.

Die Typinferenz im Typsystem von Curry besteht im Wesentlichen daraus, Aussagen der Form $P \mid A \vdash e : \sigma$ abzuleiten, wobei P eine Prädikatmenge, A eine Typannahme, welche Variablen Typschemata zuordnet, e ein Curry-Ausdruck und σ ein Typschema ist. Diese Aussage ist dann so zu lesen, dass e unter der Annahme A das Typschema σ hat, wenn die Prädikate in P erfüllt sind. Die Relation $\vdash$ kann um folgende Inferenzregel erweitert werden, mit der *verbessernde Substitutionen* zu jedem Zeitpunkt der Typinferenz angewandt werden können [Jon95]:

$$\frac{P \mid S(A) \vdash e : \sigma \quad S' \text{ verbessert } P}{S'(P) \mid S'(S(A)) \vdash e : S'(\sigma)} \qquad (imp)$$

Um diese Inferenzregel, in der S und S' Substitutionen sind, anwenden zu können, definieren wir als Nächstes, welche verbessernden Substitutionen sich durch funktionale Abhängigkeiten ergeben [Jon00]:

Definition 4.32 (Verbessernde Substitutionen durch funktionale Abhängigkeiten)**.**
Für eine Klassendeklaration der Form

class ... **=>** $C\ u_1\ \ldots\ u_m$ | ..., $l_1\ \ldots\ l_{lmax}$ -> $r_1\ \ldots\ r_{rmax}$, ... **where** ...

mit $m, lmax, rmax \in \mathbb{N}_0$, einem Klassennamen C und Typvariablen $u_1, \ldots, u_m, l_1, \ldots, l_{lmax}, r_1, \ldots, r_{rmax}$ setzen wir zunächst $L := \{i \in \{1, \ldots, m\} \mid u_i \in \{l_1, \ldots, l_{lmax}\}\}$ und $R := \{i \in \{1, \ldots, m\} \mid u_i \in \{r_1, \ldots, r_{rmax}\}\}$.

Für ein Constraint $C\ t_1\ \ldots\ t_m$ aus einer Menge von Constraints P gibt es nun folgende verbessernde Substitutionen für P:

- *Für jedes Constraint $(C\ \tau_1\ \ldots\ \tau_m) \in P$ mit Typausdrücken $\tau_1, \ldots, \tau_m$, für das $t_i = \tau_i$ für alle $i \in L$ gilt, ist der allgemeinste Unifikator U mit $U(t_j) = U(\tau_j)$ für alle $j \in R$ eine verbessernde Substitution für P.*
- *Für jede Instanzdeklaration der Form*

  instance ... **=>** $C\ \tau_1\ \ldots\ \tau_m$ **where** ...

  mit Typausdrücken $\tau_1, \ldots, \tau_m$ und eine Substitution S mit $S(\tau_i) = t_i$ für alle $i \in L$ ist der allgemeinste Unifikator U mit $U(t_j) = U(S(\tau_j))$ für alle $j \in R$ eine verbessernde Substitution für P.

Der *allgemeinste Unifikator*, der in dieser Definition verwendet wird, ist eine Substitution, die mittels *Unifikation* bestimmt wird, auf die wir in dieser Arbeit nicht weiter eingehen werden. Wenn die sonstigen Bedingungen erfüllt sind, aber die Unifikation fehlschlägt, also keine Substitution U mit

den angegebenen Eigenschaften existiert, schlägt auch die Typinferenz fehl und das Programm muss zurückgewiesen werden.

Verbessernde Substitutionen können an jeder Stelle der Typinferenz inklusive der Kontextreduktion und der Kontextimplikation und beliebig oft angewandt werden. Insbesondere kann es direkt durch die Anwendung einer verbessernden Substitution dazu kommen, dass weitere Verbesserungen möglich sind.

Die beschriebenen verbessernden Substitutionen sind jedoch nur dann gültig, wenn die funktionalen Abhängigkeiten eingehalten werden. Damit dies gewährleistet ist, müssen zwei weitere Einschränkungen an Instanzdeklarationen gestellt werden. Die erste dieser Einschränkungen bezieht sich auf Paare von Instanzen und besagt, dass wenn die auf der linken Seite einer funktionalen Abhängigkeit stehenden Instanztypen zwischen zwei Instanzen übereinstimmen, dann auch die entsprechend auf der rechten Seite stehenden Instanztypen übereinstimmen müssen [Jon00]:

Einschränkung 4.33 (Konflikt zwischen Instanzköpfen)**.**
Für jede Klasse C und jede funktionale Abhängigkeit, für die eine Klassendeklaration der Form

`class` ... `=>` $C\ u_1\ \ldots\ u_m$ | ..., $l_1\ \ldots\ l_{lmax}$ -> $r_1\ \ldots\ r_{rmax}$, ... `where` ...

existiert, wobei $m, lmax, rmax \in \mathbb{N}_0$ *und* $u_1, \ldots, u_m, l_1, \ldots, l_{lmax}, r_1, \ldots, r_{rmax}$ *Typvariablen sind, und jedes Paar von Instanzen, für die Instanzdeklarationen der Form*

`instance` ... `=>` $C\ t_1\ \ldots\ t_m$ `where` ...
`instance` ... `=>` $C\ \tau_1\ \ldots\ \tau_m$ `where` ...

mit Typausdrücken $t_1, \ldots, t_m, \tau_1, \ldots, \tau_m$ *bestehen, muss für jede Substitution* σ *gelten: Wenn* $\sigma(t_i) = \sigma(\tau_i)$ *für alle* $i \in \{1, \ldots, m\}$ *mit* $u_i \in \{l_1, \ldots, l_{lmax}\}$ *ist, muss auch* $\sigma(t_j) = \sigma(\tau_j)$ *für alle* $j \in \{1, \ldots, m\}$ *mit* $u_j \in \{r_1, \ldots, r_{rmax}\}$ *sein.*

Die zweite Einschränkung, welche von Instanzdeklarationen eingehalten werden muss, betrifft einzelne Instanzdeklarationen. Hier müssen Typvariablen aus Instanztypen, die auf der rechten Seite einer funktionalen Abhängigkeit stehen, durch die entsprechend auf der linken Seite stehenden Instanztypen abgedeckt werden, also auch in diesen vorkommen [Jon00]:

Einschränkung 4.34 (Typvariablenabdeckung in Instanzköpfen)**.**
Für jede Klasse C und jede funktionale Abhängigkeit, für die eine Klassendeklaration der Form

`class` ... `=>` $C\ u_1\ \ldots\ u_m$ | ..., $l_1\ \ldots\ l_{lmax}$ -> $r_1\ \ldots\ r_{rmax}$, ... `where` ...

existiert, wobei $m, lmax, rmax \in \mathbb{N}_0$ *und* $u_1, \ldots, u_m, l_1, \ldots, l_{lmax}, r_1, \ldots, r_{rmax}$ *Typvariablen sind, und jede Instanzdeklaration der Form*

`instance` ... `=>` $C\ t_1\ \ldots\ t_m$ `where` ...

mit Typausdrücken $t_1, \ldots, t_m$ *muss für alle* $i \in \{1, \ldots, m\}$ *mit* $u_i \in \{r_1, \ldots, r_{rmax}\}$ *gelten:*

$$\mathrm{TV}(t_i) \subseteq \{v \in \mathrm{TV}(t_j) \mid j \in \{1, \ldots, m\} \wedge u_j \in \{l_1, \ldots, l_{lmax}\}\}$$

Zuletzt soll noch der Einfluss funktionaler Abhängigkeiten auf die Menge der *freien Typvariablen* erwähnt werden, der insbesondere für die in Definition 4.17 beschriebene Kontextvereinfachung relevant ist. Bei den freien Typvariablen handelt es sich für gewöhnlich um die Typvariablen, die durch eine Typannahme A gebunden sind, die wir als $\mathrm{TV}(A)$ bezeichnen. Aufgrund funktionaler Abhängigkeiten können weitere Typvariablen eindeutig durch diese gebundenen Variablen bestimmt sein. Die so bestimmten Typvariablen dürfen deswegen nicht in Typschemata zu lokalen Deklarationen allquantifiziert werden. Um dies zu gewährleisten, erweitern wir die Menge der freien Typvariablen F um die durch eine Prädikatmenge P auf diese Art bestimmten Variablen, setzen also $F = \mathrm{COV}(P, \mathrm{TV}(A))$ [Jon00].

4.6. Defaulting

Als *Defaulting* bezeichnet man einen speziellen Mechanismus in Programmiersprachen wie Haskell und Curry, mit dem Typvariablen, die nach Einschränkung 4.21 mehrdeutig sind und somit einen Typfehler hervorrufen würden, unter bestimmten Umständen ein Standardtyp zugewiesen werden kann. In diesem Abschnitt wird dieses Defaulting kurz motiviert und dann formal beschrieben.

Numerische Literale haben in Curry einen überladenen Typ, können also für verschiedene Typen stehen. So hat z. B. das Literal `5` den beschränkten Typ `Num a => a` und lässt sich somit je nach Situation z. B. als `Int` oder `Float` interpretieren, da `Num`-Instanzen für beide Typen vordefiniert sind. Wenn solche Literale jedoch so verwendet werden, dass nicht inferiert werden kann, als welcher Typ sie interpretiert werden sollen, ist die Typvariable, die diesen Typ repräsentiert, mehrdeutig. Dies tritt bereits bei einfachen Ausdrücken wie `show 5` auf, der ohne Defaulting den ungültigen beschränkten Typ `(Num a, Show a) => String` hat. In solchen Situationen kann `a` über Defaulting ein Standardtyp wie z. B. `Int` zugewiesen werden.

Welcher Standardtyp in einem solchen Fall gewählt werden soll, wird über eine *Default-Deklaration* entschieden, welche aus Typen mit `Num`-Instanz besteht und die folgende allgemeine Form hat:

Definition 4.35 (Default-Deklaration)**.**
Eine Default-Deklaration *hat die allgemeine Form*

`default` `(`T_1`, ..., `T_n`)`

Dabei sind $n \in \mathbb{N}_0$ *und* $T_1, \ldots, T_n$ *Typen. Für alle* $i \in \{1, \ldots, n\}$ *muss gelten, dass* $\mathrm{TV}(T_i) = \varnothing$ *und* $\mathrm{RED}(\{\texttt{Num}\ T_i\}, \varnothing) = \varnothing$ *ist.*

Eine solche Default-Deklaration gilt für genau das Modul, in dem sie angegeben ist. Falls ein Modul keine Default-Deklaration enthält, wird in Curry **`default`** `(Int, Float)` als Default-Deklaration angenommen. Als Standardtyp für eine mehrdeutige Typvariable wird vereinfacht gesagt erste Typ aus der jeweils geltenden Default-Deklaration gewählt, der alle Constraints erfüllt, die im betrachteten Kontext an diese Typvariable gestellt werden. Gibt es keinen solchen Typen, wird Defaulting nicht angewandt und es kommt zu einem Typfehler wegen der mehrdeutigen Typvariable. Somit kann mit der Default-Deklaration **`default`** `()` das Defaulting in einem Modul unterbunden werden.

Ohne weitere Einschränkungen kann es mit diesen informellen Defaulting-Regeln aber zu komplizierten Fällen kommen, insbesondere mit Multiparametertypklassen. Als Beispiel hierfür betrachten wir diese beiden Instanzen zu einer zweistelligen Klasse `C`:

```
instance C Int Float
instance C Float Int
```

Wenn es nun zwei mehrdeutige Typvariablen `a` und `b` gibt, die ausschließlich in einem Constraint `C a b` vorkommen, könnte `a` und `b` nicht getrennt voneinander ein Standardtyp zugewiesen werden. Stattdessen würde ein komplizierterer Defaulting-Algorithmus benötigt, dessen Ergebnis für den Programmierer möglicherweise nicht sofort ersichtlich ist.

Um solche Situationen zu vermeiden, wird Defaulting ausschließlich bei einfachen Constraints zu Klassen aus dem vordefinierten Modul `Prelude` angewandt. Diese Einschränkung entspricht dem im Haskell-Report[3] beschriebenen Vorgehen und wird in folgender Definition zum Defaulting für Curry formal beschrieben:

[3] https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-790004.3.4

Definition 4.36 (Defaulting von Typvariablen)**.**
Sei **`default`** `(`T_1`, ..., `T_n`)` *mit* $n \in \mathbb{N}_0$ *und Typen* $T_1, \ldots, T_n$ *die für das aktuelle Modul geltende Default-Deklaration, M die Menge mehrdeutiger Typvariablen, E die Menge gegebener Constraints und K der betrachtete Kontext. Einer Typvariable* $u \in M$ *wird ein Standardtyp zugewiesen, wenn folgende Bedingungen erfüllt sind:*

$$\exists \text{ Klasse } C : (C\ \ u) \in K \wedge (\texttt{Num}\ \ u) \in \{C\ \ u\}_+ \tag{1}$$

$$\forall c \in K : u \notin \mathrm{TV}(c) \vee (\exists \text{ Klasse } C : (C\ \ u) = c \wedge \text{C ist im Modul } \texttt{Prelude} \text{ definiert}) \tag{2}$$

$$D \neq \varnothing \text{ mit } D := \{T \in \{T_1, \ldots, T_n\} \mid \forall (C\ \ u) \in K : \mathrm{RED}(\{C\ \ T\}, E_+) = \varnothing\} \tag{3}$$

u wird unter diesen Voraussetzungen der Typ T_k *zugewiesen, wobei* $k := \min(\{i \in \{1, \ldots, n\} \mid T_i \in D\})$ *ist.*

Kapitel 5

Implementierung im Curry-Frontend

Dieses Kapitel behandelt die Implementierung der in Kapitel 3 informell eingeführten Spracherweiterungen zu Multiparametertypklassen, flexiblen Instanzen und flexiblen Kontexten im Curry-Frontend. Die Erweiterung zu funktionalen Abhängigkeiten konnte nicht umgesetzt werden, wird in diesem Kapitel allerdings in Form der unvollständigen Implementierung dennoch zusammen mit den anderen Erweiterungen betrachtet.

Zuerst wird in Abschnitt 5.1 ein umfangreicher Überblick über den Stand des Curry-Frontends vor der Implementierung von Typklassenerweiterungen gegeben. Danach werden die durchgeführten Anpassungen an der Kompilierumgebung sowie weiteren intern verwendeten Datenstrukturen (Abschnitt 5.2), an den Kompilierphasen (Abschnitt 5.3) und am Import und Export von Modulen (Abschnitt 5.4) beschrieben. Als letztes gehen wir in Abschnitt 5.5 auf das Hauptproblem ein, welche die vollständige Umsetzung funktionaler Abhängigkeiten verhindert hat.

5.1. Überblick über das Curry-Frontend

Das Curry-Frontend ist ein Programm, welches Curry-Code einliest, prüft, transformiert und in verschiedenen Formaten ausgibt. Zu diesen Formaten zählt *FlatCurry*, das als gemeinsame Zwischensprache der Curry-Compiler PAKCS und KiCS2 dient und von deren Backends zu Prolog- bzw. Haskell-Code übersetzt wird. Abbildung 5.1 gibt eine Übersicht darüber, wie die Kompilation von Curry-Code mit verschiedenen Backends abläuft und aus welchen Kompilierphasen das Frontend aufgebaut ist.

In diesem Abschnitt wird zunächst kurz auf die verschiedenen Ausgabeformate des Curry-Frontends eingegangen (Abschnitt 5.1.1). Danach wird eine Übersicht über die in Abbildung 5.1 gezeigten Elemente des Frontends, also die intern verwendeten Datenstrukturen wie die Kompilierumgebung (Abschnitt 5.1.2) und die Kompilierphasen (Abschnitt 5.1.3) gegeben. Zuletzt betrachten wir kurz die Funktionsweise des Imports und Exports von Modulen (Abschnitt 5.1.4).

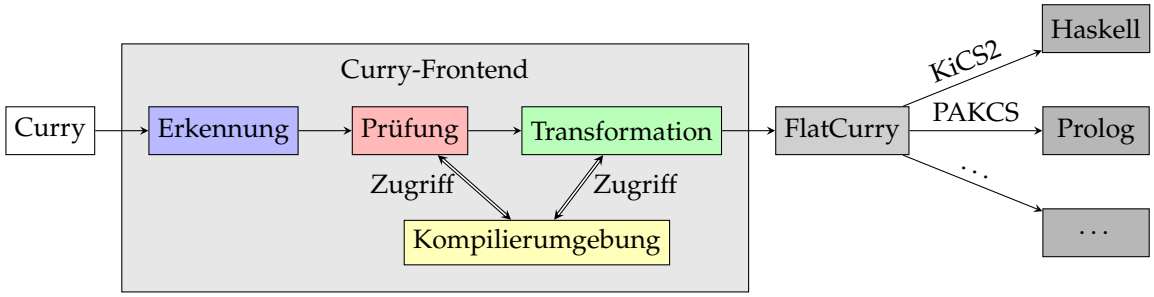

Abbildung 5.1. Kompilation von Curry-Code mit dem Curry-Frontend und verschiedenen Backends

5.1.1. Ausgabeformate

Das Curry-Frontend unterstützt die in den folgenden Unterabschnitten beschriebenen Ausgabeformate. Mit Ausnahme von FlatCurry mussten sie alle mit der Implementierung der Typklassenerweiterungen leicht überarbeitet werden. Da diese Anpassungen aber ähnlich zu anderen beschriebenen Änderungen sind (s. Abschnitt 5.2.1 und Abschnitt 5.3.1), werden sie in diesem Kapitel nicht genauer behandelt.

FlatCurry

FlatCurry ist eine stark vereinfachte Variante von Curry. Unter anderem bestehen alle Funktionen aus genau einer Funktionsregel und haben nur Variablen als Parameter. *Pattern Matching* findet stattdessen nur in **case**-Ausdrücken statt. In FlatCurry kommen keine Typklassenelemente oder Sprachkonstrukte, die syntaktischen Zucker darstellen, mehr vor.

Als Varianten von FlatCurry stehen zudem *getyptes FlatCurry* und *typannotiertes FlatCurry* zur Verfügung. Beide enthalten zusätzliche Typinformationen zu Ausdrücken und Mustern in Funktionsdeklarationen, wobei die typannotierte Variante etwas mehr Informationen enthält und parametrisch im Typ der Annotationen ist.

AbstractCurry

AbstractCurry ist eine abstrakte Repräsentation von Curry-Code, die Ähnlichkeiten mit dem intern im Curry-Frontend verwendeten *abstrakten Syntaxbaum* (AST) zu Curry aufweist, der in Abschnitt 5.1.2 behandelt wird. Es wird zwischen getyptem und ungetyptem AbstractCurry unterschieden, wobei nur die erstgenannte Variante die Typen von Funktionsdeklarationen beinhaltet.

HTML

Das HTML-Ausgabeformat kann zur Generierung von HTML-Dokumentationsseiten zu Curry-Quellcode genutzt werden. Es bietet eine Syntaxhervorhebung und Querverweise von Bezeichnern zu deren Deklaration.

Token-Stream

Mit diesem Format wird die *Tokenfolge* ausgegeben, die vom Lexer (s. Abschnitt 5.1.3) zurückgegeben wurde. Bei einem *Token* handelt es sich um zusammengehörige Zeichen im Quellcode, z. B. um ein Schlüsselwort oder ein Literal.

5.1.2. Kompilierumgebung und Datenstrukturen

Im Curry-Frontend werden zahlreiche verschiedene Datentypen verwendet, um ganze Curry-Programme oder spezielle Teile von ihnen zu repräsentieren. Die folgenden Datenstrukturen sind für diese Arbeit besonders relevant:

Abstrakter Syntaxbaum zu Curry

Ein abstrakter Syntaxbaum stellt ein Curry-Programm in Gänze in Form einer Baumstruktur dar, in welcher logische Einheiten des Programms zusammengefasst werden. So wird z. B. eine Funktionsdeklaration im AST durch einen Knoten dargestellt, der Unterknoten zum Bezeichner der Funktion

und den Funktionsregeln enthält. Die meisten Knoten enthalten zudem einen *Source-Span*, der auf die Quellcodestelle der entsprechenden Spracheinheit verweist.

Interne Typdarstellung

Die Datentypen des ASTs zu Typausdrücken umfassen Konstruktoren wie den Konstruktor für Tupeltypen, die syntaktischen Zucker darstellen, und beinhalten zudem Source-Spans. Die Arbeit mit diesen Typen würde daher viele Funktionen im Frontend, in denen diese Zusatzinformationen nicht relevant sind, unnötig vergrößern. Aus diesem Grund wird an vielen Stellen des Frontends mit einer einfacheren Typdarstellung gearbeitet. Neben Datentypen für Typausdrücke bietet diese Darstellung auch Typen für Constraints, Kontexte, beschränkte Typen und Typschemata an.

Kompilierumgebung

Die Kompilierumgebung bildet einen Zustand, der zwischen den Kompilierschritten im Curry-Frontend zusätzlich zum kompilierten Modul weitergegeben wird. Sie besteht aus zahlreichen Unterumgebungen, die in bestimmten Kompilierschritten um neue Einträge ergänzt werden. Auf diese Einträge kann ab dem jeweiligen Schritt dann zur Prüfung oder Transformation von Programmteilen zugegriffen werden. Abbildung 5.2 zeigt die Unterumgebungen der Kompilierumgebung in der Reihenfolge, in der sie beim Kompiliervorgang verwendet werden. Die dort aufgeführte Typbezeichnerumgebung wird nicht zwischen Kompilierschritten weitergegeben und ist damit eigentlich kein Teil der Kompilierumgebung, weist aber Ähnlichkeiten zu anderen Unterumgebungen auf. Zusätzlich zu den Unterumgebungen umfasst die Kompilierumgebung auch den Namen sowie den Dateipfad des aktuellen Moduls, eine Liste der aktivierten Spracherweiterungen und eine Tokenfolge mit Source-Spans.

5.1.3. Kompilierphasen

Wie in Abbildung 5.1 zu erkennen ist, ist der Kompiliervorgang im Curry-Frontend in drei Kompilierphasen unterteilt: Die *Erkennungsphase*, in der das Programm eingelesen und in einen AST umgewandelt wird, die *Prüfungsphase*, in der die semantische Korrektheit des Programms überprüft wird, und die *Transformationsphase*, in der das Programm schrittweise zu FlatCurry-Code vereinfacht wird. Diese Kompilierphasen gliedern sich wiederum in mehrere Kompilierschritte, die in den Abbildungen 5.3, 5.4, 5.5 und 5.6 dargestellt sind. Für diese Schritte werden in dieser Arbeit zur einfacheren Zuordnung zum Quellcode des Curry-Frontends die englischen Bezeichnungen verwendet.

5.1.4. Import und Export von Modulen

Ein zusätzlicher Teil der Kompilation von Curry-Programmen, der aufgrund der schwierigen Darstellung in Abbildung 5.1 zur Kompilation von Curry-Code und in Abschnitt 5.1.3 zu den Kompilierphasen nicht behandelt wurde, ist das Modulsystem von Curry, welches mit Schnittstellendateien arbeitet, welche beim Export von Modulen geschrieben und beim Import gelesen werden. Die folgenden Unterabschnitte stellen dieses System genauer vor.

Schnittstellenumgebung – Die Schnittstellenumgebung bildet Modulnamen auf die Schnittstellen dieser Module ab. Sie wird verwendet, um beim Import von Modulen auf die entsprechende Schnittstelle zurückgreifen zu können.

Modulaliasumgebung – Diese Umgebung bildet jeden Modulnamen auf den Namen ab, mit dem Bezeichner aus diesem Modul im aktuellen Modul qualifiziert werden können. Dies ist entweder der Modulname selbst oder der in der entsprechenden Importdeklaration angegebene Modulalias.

(Typbezeichnerumgebung) – In dieser Umgebung werden Bezeichnern die zu ihnen passenden Typkonstruktoren zugeordnet. Zu Datentypen werden zudem die Konstruktornamen und zu Typklassen die Namen der Klassenmethoden gespeichert. Die Typbezeichnerumgebung wird zur Prüfung von Typausdrücken im Type Syntax Check sowie im Interface Syntax Check verwendet.

Typkonstruktorenumgebung – Die Typkonstruktorenumgebung kann als erweiterte Typbezeichnerumgebung aufgefasst werden. Sie beinhaltet auch Typvariablen und enthält die Sorte von Typkonstruktoren und -variablen. Außerdem werden zu Datentypen die vollständigen Datenkonstruktoren, zu Typsynonymen die Stelligkeit sowie der Typausdruck und zu Typklassen die Klassenmethoden mit Namen, Stelligkeit der Standardimplementierung sowie beschränktem Typ gespeichert.

Klassenumgebung – Ordnet jedem Typklassennamen die Informationen der dazugehörigen Typklasse zu. Diese Klasseninformationen umfassen die Namen der direkten Oberklassen und der Klassenmethoden. Zu letzteren ist auch eingetragen, ob eine Standardimplementierung vorliegt.

Präzedenzenumgebung – Diese Umgebung bildet Infix-Operatoren auf ihre *Präzedenz* ab. Mit diesen Präzedenzen kann bestimmt werden, wie Ausdrücke mit mehreren Infix-Operatoren implizit geklammert sind.

Instanzenumgebung – Die Instanzenumgebung beinhaltet alle Instanzen, die im aktuellen Modul sichtbar sind. Instanzen werden über den Namen der Klasse und des Instanztyps identifiziert und bestehen in dieser Umgebung aus dem Namen des Moduls, in dem die Instanz deklariert ist, dem Instanzkontext und den Namen der implementierten Klassenmethoden mitsamt der Stelligkeit der Implementierung.

Werteumgebung – Diese Umgebung ordnet Datenkonstruktoren und Variablen inklusive Funktionen ihrem Typschema zu. Zu Datenkonstruktoren werden zudem die Stelligkeit und bei Record-Konstruktoren auch die Feldselektoren gespeichert. Zu Variablen ist die Stelligkeit und zu Klassenmethoden zusätzlich der zugehörige Klassenname eingetragen. Einträge zu Feldselektoren enthalten die Namen der Datenkonstruktoren, welche dieses Feld besitzen.

Abbildung 5.2. Unterumgebungen der Kompilierumgebung im Curry-Frontend

Abbildung 5.3. Kompilierphasen im Curry-Frontend - Teil 1

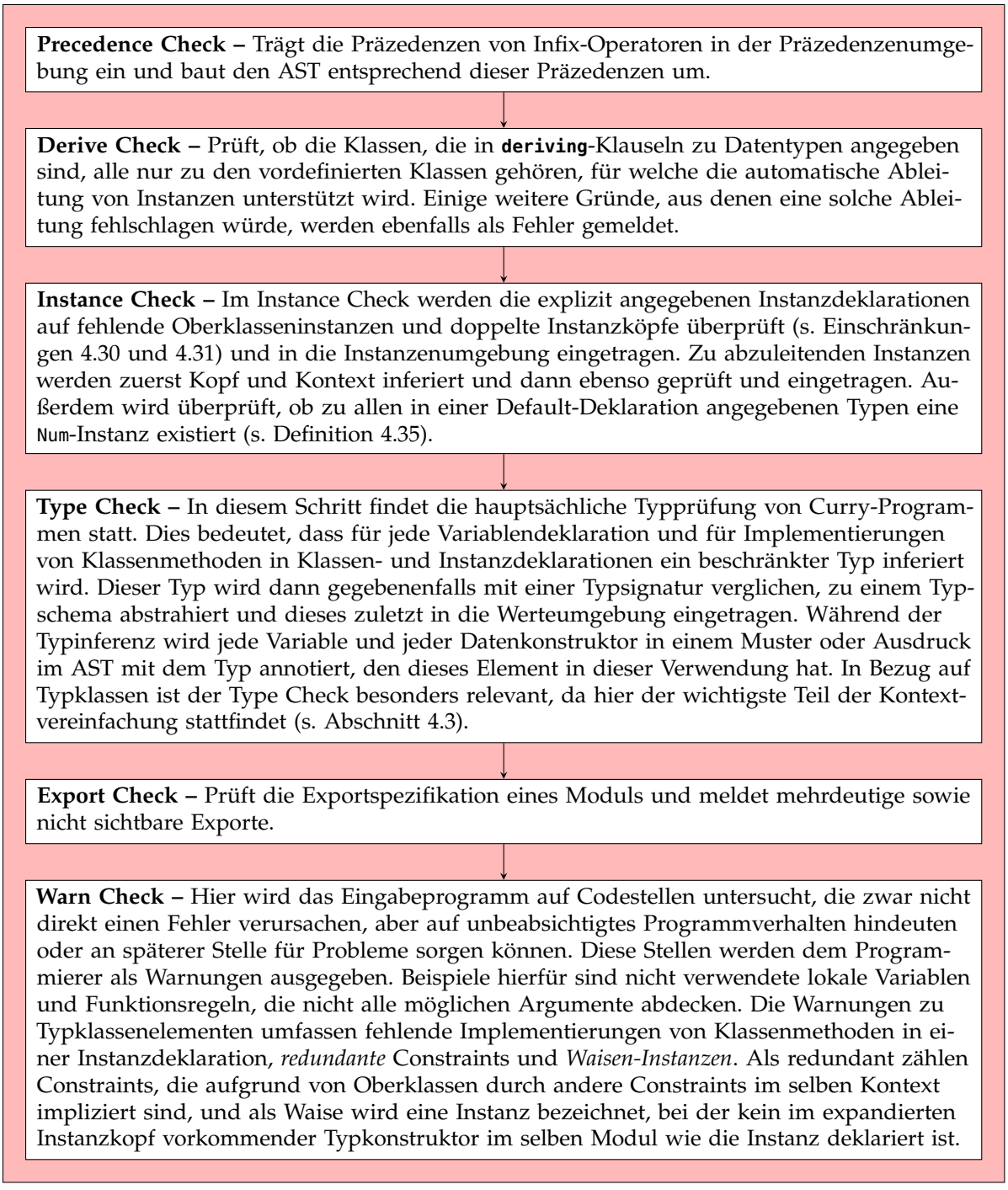

Abbildung 5.4. Kompilierphasen im Curry-Frontend - Teil 2

Abbildung 5.5. Kompilierphasen im Curry-Frontend - Teil 3

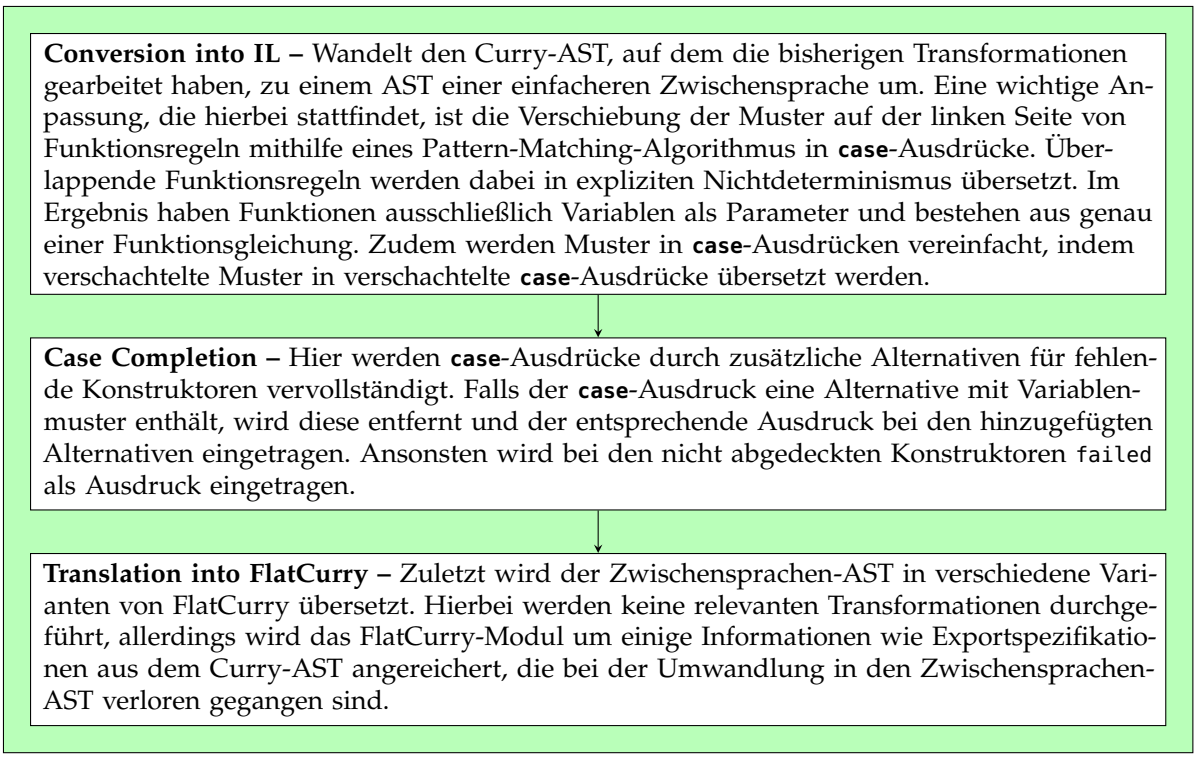

Abbildung 5.6. Kompilierphasen im Curry-Frontend - Teil 4

Import

Der Import von Modulen findet zwischen der Erkennungs- und der Prüfungsphase statt. An dieser Stelle kann im AST auf die Importdeklarationen zugegriffen und damit die Schnittstellendateien der importierten Module geladen werden. Diese Schnittstellen durchlaufen nun drei eigene Prüfungsschritte. Zuerst werden im **Interface Syntax Check** die Typausdrücke der Schnittstelle überprüft. Dieser Schritt weist eine große Ähnlichkeit zum Type Syntax Check auf, der in Abbildung 5.3 dargestellt ist. Im **Interface Check** wird dann geprüft, ob die Schnittstellen der importierten Module konsistent zu ihren Abhängigkeiten sind oder ob sie rekompiliert werden müssen. Hierzu werden die von einer Schnittstelle exportierten Deklarationen, die ursprünglich aus anderen Modulen stammen, mit den Schnittstellen ebendieser Module abgeglichen. Als letzten Prüfungsschritt werden im **Import Syntax Check** die Importdeklaration des betrachteten Moduls auf Fehler überprüft und ähnlich wie in der Export Expansion (s. Abbildung 5.5) erweitert.

Nach diesen Schritten werden die importierten Deklarationen mithilfe der geladenen Schnittstellen in die jeweiligen Unterumgebungen der Kompilierumgebung eingetragen. Die Namen, unter denen sie eingetragen werden, hängen dabei von der Importdeklaration ab: In jedem Fall werden die Deklarationen unter dem mit dem Modulnamen bzw. dem Modulalias (falls angegeben) qualifizierten Bezeichner eingetragen. Falls nicht durch ein **qualified** gekennzeichnet ist, dass ausschließlich qualifizierte Bezeichner importiert werden sollen, werden die Deklarationen zudem mit unqualifizierten Bezeichnern eingetragen.

Export

Die Schnittstelle zu einem Modul mit allen exportierten Deklarationen wird zwischen den Schritten Qualification und Instance Deriving aus der Transformationsphase (s. Abbildung 5.5) erstellt. Wichtig ist, dass diese Schnittstelle neben den explizit exportierten Deklarationen auch Instanzdeklarationen, die nicht explizit exportiert werden können, und sogenannte *versteckte Deklarationen* enthält. Zu ersteren zählen alle sichtbaren Instanzen, die bei einem Import des aktuellen Moduls benötigt werden könnten. Die versteckten Deklarationen umfassen Typkonstruktoren, die in der Schnittstelle vorkommen, aber nicht explizit exportiert werden. Mit ihnen wird die Abgeschlossenheit der Schnittstelle sichergestellt und sie werden benötigt, um in der Schnittstelle Typkonstruktoren eindeutig von Typvariablen unterscheiden zu können, die sich in Curry ansonsten die gleichen Namen teilen könnten.

5.2. Kompilierumgebung und Datenstrukturen

Dieser Abschnitt beschreibt die Änderungen, die zur Implementierung von Typklassenerweiterungen im Curry-Frontend am abstrakten Syntaxbaum, der internen Typdarstellung und der Klassen- sowie Instanzenumgebung aus Kompilierumgebung durchgeführt wurden. Ein Überblick über diese Datenstrukturen kann in Abschnitt 5.1.2 gefunden werden.

5.2.1. Abstrakter Syntaxbaum

Am Curry-AST und den darauf aufbauenden Hilfsfunktionen mussten insbesondere für die Erweiterung um Multiparametertypklassen zahlreiche kleine Anpassungen vorgenommen werden, die im Folgenden vorgestellt werden:

- Zu der Liste unterstützter Spracherweiterungen wurden die Namen hinzugefügt, mit denen die implementierten Typklassenerweiterungen aktiviert werden können. Die gewählten Namen `MultiParamTypeClasses`, `FlexibleInstances` und `FlexibleContexts` sowie `FunctionalDependencies` entsprechen denen, die im GHC verwendet werden.
- Die Datenkonstruktoren im AST, die Klassendeklarationen, Instanzdeklarationen und Constraints repräsentieren, enthielten vorher je ein Feld für den Klassenparameter, den Instanztyp und das Typargument eines Constraints. Diese Felder wurden durch solche für Listen dieser Elemente ersetzt.
- Ein Konstruktor für die Sorte Constraint aus Definition 4.2 wurde zum AST-Datentyp für Sorten hinzugefügt. Diese Sorte wurde vorher nicht benötigt, da zu Typklassen die Sorte ihres Klassenparameters eingetragen und die Sortenprüfung mit Typklassen als Spezialfall implementiert war. Eine Multiparametertypklasse bräuchte mit diesem Ansatz jedoch eine Liste von Sorten, was mit der Constraint-Sorte vermieden werden konnte.
- Für die Erweiterung zu funktionalen Abhängigkeiten wurde ein Datentyp zur Repräsentation funktionaler Abhängigkeiten ergänzt, der für die linke und rechte Seite einer Abhängigkeit je eine Liste von Typvariablen speichert. Der Konstruktor für Klassendeklarationen wurde um ein Feld für Listen funktionaler Abhängigkeiten erweitert.
- Zahlreiche auf dem AST aufbauende Hilfsfunktionen haben Überarbeitungen erfahren. Zum Beispiel wurde die Funktion, welche den Source-Span einer Klassendeklaration aus den Source-Spans der einzelnen Elemente berechnet, so überarbeitet, dass sie auch im Falle einer nullstelligen Typklasse oder einer leeren **`where`**-Klausel das korrekte Ergebnis liefert.

5.2.2. Interne Typdarstellung

In der internen Typdarstellung wurde insbesondere die Repräsentation von Constraints bzw. Prädikaten, wie sie in diesem Teil des Curry-Frontends genannt werden, überarbeitet. Die wichtigsten der durchgeführten Änderungen sind in folgender Liste beschrieben:

- Genau wie im AST beinhalten Constraints bzw. Prädikate nun eine Liste von Typausdrücken für ihre Typargumente statt eines einzelnen Typausdrucks.
- Ein Datentyp zur Repräsentation einer Instanz in Form ihres Kontexts und ihrer Instanztypen wurde eingeführt. Bislang waren zu diesem Zweck gewöhnliche beschränkte Typen verwendet worden.
- In bestimmten Fällen ist es notwendig, den beschränkten Typ einer Klassenmethode ohne das implizite Klassenconstraint zu betrachten, z. B. um den Typ der Implementierung einer Klassenmethode in einer Instanz zu ermitteln (siehe Definition 4.13). Ohne Typklassenerweiterungen konnte hierzu das implizite Klassenconstraint recht einfach aus dem vollständigen beschränkten Typen entfernt werden: Bei der Übertragung des Typs einer Klassenmethode in die interne Typdarstellung, in der Typvariablen durch Indizes dargestellt werden, wird darauf geachtet, dass der Klassenparameter den niedrigsten Index erhält. Aufgrund dessen, dass Klassenparameter in Klassenmethoden nicht weiter eingeschränkt werden dürfen (s. Einschränkung 4.19), war das implizite Klassenconstraint nun immer das einzige Constraint mit dem niedrigsten Typvariablenindex. Da Kontexte in der internen Typdarstellung sortierte Mengen von Prädikaten sind, war dieses Constraint dank einer entsprechenden `Ord`-Instanz für Prädikate immer das geringste Element des Kontexts und konnte problemlos entfernt werden.

  Mit Multiparametertypklassen kann es jedoch zu dem Fall kommen, dass eine Klassenmethode einer nullstelligen Typklasse ein Constraint zu einer anderen nullstelligen Typklasse im Kontext enthält. In diesem Fall funktioniert der beschriebene Ansatz nicht umsetzen und es besteht ohne zusätzliche Informationen auch anderweitig keine Möglichkeit, das implizite Klassenconstraint eindeutig zu bestimmen. Deswegen wurde ein Flag für Prädikate eingeführt, welches kennzeichnet, ob es sich um ein implizites Klassenconstraint oder ein anderes Prädikat handelt. Bei der Verwendung dieses Flags musste genau darauf geachtet werden, dass es in jeder Situation richtig gesetzt und bei Operationen auf Prädikaten auch korrekt weitergegeben oder aufgehoben wird.
- Es wurde eine zusätzliche Art von Prädikaten eingeführt, die neben dem eigentlichen Prädikat auch Daten enthält, die bei der Ausgabe des Prädikats in Fehlermeldungen verwendet werden können. Diese Prädikate und Mengen davon werden im Type Check verwendet, deshalb sind die Gründe für die Ergänzung in Abschnitt 5.3.6 näher beschrieben. Es wurde zudem eine Typklasse für beide Arten von Prädikaten angelegt und viele Operationen auf Prädikaten wurden so überarbeitet, dass sie parametrisch in der Art der Prädikate sind.

5.2.3. Klassenumgebung

Die Klassenumgebung, die bislang recht einfach aufgebaut war (s. Abbildung 5.2), musste aufgrund der Erweiterungen um Multiparametertypklassen, flexible Kontexte und funktionale Abhängigkeiten grundlegend überarbeitet werden. Zunächst wurden die zu einer Typklasse gespeicherten Informationen um die Stelligkeit der Klasse erweitert. Als weiterer Schritt musste die Repräsentation der Oberklassen angepasst werden, da die Speicherung der Oberklassennamen mit Multiparametertypklassen nicht mehr ausreichend ist (s. Abschnitt 3.2.3). Um auch flexible Kontexte korrekt zu unterstützen,

werden Oberklassenkontexte nun als Prädikatmenge gespeichert. Die Typausdrücke in den Oberklassenconstraints werden beim Eintrag in die Klassenumgebung nun expandiert, um die enthaltenen Typsynonyme aufzulösen und einen eindeutigen Kontext zu erhalten. Die Typausdrücke müssen außerdem bei der Berechnung indirekter Oberklassen weitergegeben werden (s. die Kontextimplikation aus Definition 4.15).

Für die Erweiterung um funktionale Anhängigkeiten wurden die Klasseninformationen um ein weiteres Feld für diese Abhängigkeiten erweitert. Zur Repräsentation der Typvariablen auf der linken und rechten Seite einer funktionalen Abhängigkeit wurde ein Paar von Variablenindexmengen gewählt. Mit dieser Darstellung konnten der in Algorithmus 4.20 vorgestellte COV-Algorithmus zur Bestimmung abgedeckter Typvariablen sowie einige Hilfsfunktionen einfach und effizient umgesetzt werden.

5.2.4. Instanzenumgebung

Ebenso wie die Klassenumgebung hat die Instanzenumgebung umfangreiche Änderungen erfahren, insbesondere wegen der Erweiterungen um Multiparametertypklassen, flexible Instanzen und funktionale Abhängigkeiten. Der Hauptgrund für diese Änderungen war, dass die bisherige Identifikation von Instanzen über den Klassennamen und den Typkonstruktornamen des bzw. der Instanztypen mit der Möglichkeit mehrfach in Instanzköpfen vorkommender Typvariablen (s. Abschnitt 3.2.3) und spätestens mit flexiblen Instanzen nicht mehr eindeutig ist. Deswegen werden neben dem Klassennamen nun die vollständigen Instanztypen zur Instanzidentifikation genutzt.

Damit dieser Identifikator zwischen Instanzen einheitlich ist, werden die Instanztypen einerseits expandiert, Typsynonyme also aufgelöst, und andererseits wird die Typvariablenbenennung *normalisiert*. Bei dieser Normalisierung werden den Typvariablen abhängig davon, wann sie beim Lesen der Instanztypen von links nach rechts zuerst vorkommen, aufsteigende Indizes zugewiesen. Nach diesen beiden Schritten können die nach Einschränkung 4.31 nicht erlaubten doppelten Instanzköpfe einfach erkannt werden, da sie den gleichen Identifikator erhalten.

Mit dieser neuen Instanzidentifikation geht die Implementierung eines neuen Verfahrens zum Finden von Instanzen einher. Die Instanzenumgebung stellt dafür eine Funktion bereit, welche zu einem Constraint die passenden und potenziell passenden Instanzen liefert (s. Bedingungen (1) und (2) der Kontextreduktion aus Definition 4.14). Die dazugehörigen Substitutionen, durch deren Anwendung die Gleichheit zwischen gesuchtem Constraint und Instanzkopf hergestellt wird, werden mit *Typmatching* für die passenden Instanzen und *Typunifikation* für die potenziell passenden Instanzen bestimmt. Diese Verfahren existieren bereits an anderen Stellen im Curry-Frontend und werden in dieser Arbeit nicht näher behandelt.

Um sowohl die Suche (potenziell) passender Instanzen, bei der das gegebene Constraint mit allen Instanzen der passenden Klasse verglichen werden muss, als auch die Zugriffe, bei denen der genaue Instanzidentifikator bekannt ist, möglichst effizient zu implementieren zu können, besteht die Instanzenumgebung aus verschachtelten Abbildungen. Die erste hiervon bildet Klassennamen auf die Instanzen der Klasse ab und die Instanzen selbst sind durch eine weitere Abbildung von Instanztypen auf Informationen zu dieser Instanz dargestellt. Diese Instanzinformationen wurden durch die Implementierung der Typklassenerweiterungen nicht verändert, bestehen also weiterhin aus dem Modul, in dem die Instanz deklariert ist, dem Instanzkontext und den Namen der implementierten Klassenmethoden mit der Stelligkeit der Implementierung.

Das anschließende Beispiel zeigt, wie Instanzen in der Instanzenumgebung im Vergleich zu ihren Deklarationen im Quellcode aussehen:

Beispiel 5.1 (Ausschnitt einer Instanzenumgebung)**.**
Wir betrachten die folgenden Klassen- und Instanzdeklarationen aus einem Modul `M`*:*

```
class C a b where methodC :: a -> b -> a

instance           C String [a] where methodC "Hello" _ = "World"
instance Eq [a] => C (b, a) [a] where methodC           = const
```

Ein Ausschnitt aus der Instanzenumgebung, der die obigen Instanzdeklarationen und eine Instanz der `Eq`*-Klasse enthält, sieht nun folgendermaßen aus:*

Schlüssel	**Wert**			
	Schlüssel	**Wert**		
Klassenname	Instanztypen	Modul	Instanzkontext	Implementierungen
`M.C`	`[Prelude.Char,[a]]`	`M`	$\varnothing$	$[$`(methodC,2)`$]$
	`[(a, b),[b]]`	`M`	$\{$`Prelude.Eq [b]`$\}$	$[$`(methodC,0)`$]$
`Prelude.Eq`	`[a]`	`Prelude`	$\{$`Prelude.Eq a`$\}$	$[$`((==),2)`$]$

In der Tabelle aus diesem Beispiel zeigen die Überschriften **Schlüssel** und **Wert** die Struktur der Abbildungen. Außerdem ist die Verwendung eindeutiger Originalnamen bei den Typkonstruktoren, die Expansion von Typausdrücken beim Instanztyp `String` und die Normalisierung von Typvariablen bei der zweiten Instanz zu erkennen. Die in der Instanzenumgebung intern für Typvariablen verwendeten Zahlen wurden in der Tabelle durch die üblicherweise verwendeten Buchstaben ersetzt.

Für die Erweiterung um funktionale Abhängigkeiten wurde der Instanzenumgebung außerdem eine Funktion hinzugefügt, welche zu einem Constraint und den Instanzen aus der Instanzenumgebung alle Paare von Typen berechnet, die aufgrund funktionaler Abhängigkeiten identisch sein müssen, nach Definition 4.32 zur Bestimmung einer verbessernden Substitution also unifiziert werden können.

5.3. Kompilierphasen

Für die Implementierung der Typklassenerweiterungen wurden keine neuen Kompilierschritte zu den Kompilierphasen des Curry-Frontends hinzugefügt, aber viele der bestehenden Schritte überarbeitet. Diese Überarbeitungen werden in den anschließenden Unterabschnitten vorgestellt. Besonderes Augenmerk sei dabei auf Abschnitt 5.3.8 gelegt, der nicht nur die durchgeführten Änderungen, sondern kurz auch die allgemeine Funktionsweise der Wörterbuchübersetzung von Typklassenelementen beschreibt. Darüber hinaus ist ein Überblick, der den Stand der Kompilierschritte von vor dieser Arbeit beschreibt, in Abschnitt 5.1.3 zu finden.

5.3.1. Lexer

Die in dieser Arbeit behandelten Typklassenerweiterungen verwenden keine neuen Sonderzeichen oder Schlüsselwörter, sodass der Lexer für das Einlesen von Curry-Programmen nicht überarbeitet werden musste. Bei den Schnittstellendateien, in der die Sorten von Typkonstruktoren eingetragen sind, hat sich durch die Einführung der Sorte Constraint jedoch eine Änderung ergeben, die vom Lexer berücksichtigt werden musste. **`Constraint`** wurde hierzu als *spezieller Bezeichner* zu den Token hinzugefügt. Ein solcher spezieller Bezeichner unterscheidet sich von Schlüsselwörtern dadurch, dass er in der Regel als normaler Bezeichner verwendet werden kann und nur in bestimmten Codestellen eine besondere Bedeutung hat. Andere Beispiele für solche speziellen Bezeichner sind **`qualified`** und **`hiding`**, die nur in Importdeklarationen eine besondere Bedeutung haben.

5.3.2. Parser

Der Parser wurde für die Erweiterung um Multiparametertypklassen so überarbeitet, dass Klassendeklarationen, Instanzdeklarationen und Constraints mit einer beliebigen Anzahl an Klassenparametern, Instanztypen bzw. Typargumenten korrekt geparst werden, und damit an die Veränderungen im AST angepasst. Mit der Erweiterung um flexible Kontexte wurden zudem die beim Parsen der Typargumente von Constraints verwendeten Funktionen, die auf reguläre Constraints (s. Definition 4.5) beschränkt waren, entfernt und durch die bestehenden Funktionen zum Parsen von Typausdrücken ersetzt.

Für die Erweiterung um funktionale Abhängigkeiten wurde der Parser zudem um Funktionen zum Parsen dieser Abhängigkeiten erweitert.

5.3.3. Type Syntax Check

Im Type Syntax Check werden zahlreiche Prüfungen in Bezug auf Typklassenelemente durchgeführt, die im Rahmen dieser Arbeit überarbeitet werden mussten. Insbesondere erfolgt in diesem Kompilierschritt nun die Prüfung, ob alle für das jeweilige Modul benötigten Typklassenerweiterungen auch aktiviert sind. Etwas detaillierter sind die durchgeführten Änderungen in folgender Liste beschrieben:

- Wenn eine Typklasse ohne oder mit mehr als einem Klassenparameter oder eine Instanz zu einer solchen Klasse deklariert wird, ohne dass die Erweiterung zu Multiparametertypklassen aktiviert ist, gibt das Frontend nun eine Fehlermeldung aus, die auf den Fehler und die Erweiterung hinweist. Constraints zu Multiparametertypklassen können hingegen ohne die Erweiterung verwendet werden.
- Bereits vor dieser Arbeit wurden im Type Syntax Check die Einschränkungen geprüft, die ohne Typklassenerweiterungen an die Form von Instanztypen und Typargumenten in Constraints gestellt werden (s. Definition 4.13 und Definition 4.5). Diese Prüfungen wurden einerseits an Multiparametertypklassen angepasst, werden andererseits nun aber nicht mehr durchgeführt, wenn die Erweiterung zu flexiblen Instanzen bzw. flexiblen Kontexten aktiviert ist. Zudem weisen die dazugehörigen Fehlermeldungen auf die entsprechende Spracherweiterung hin.
- Die Prüfung auf Klassenmethoden, welche den bzw. die Klassenparameter weiter einschränken wurde wie in Einschränkung 4.19 beschrieben auf Multiparametertypklassen erweitert.
- Für die Erweiterung um funktionale Abhängigkeiten wurde eine Fehlermeldung hinzugefügt, die bei der Deklaration einer Klasse mit funktionalen Abhängigkeiten ohne Aktivierung der Erweiterung ausgegeben wird. Außerdem wird die durch Definition 4.12 vorgegebene Form funktionaler Abhängigkeiten überprüft.
- Ebenfalls für die Erweiterung um funktionale Abhängigkeiten wurden die im Type Syntax Check durchgeführten Prüfungen auf Typvariablen, die nur im Kontext eines beschränkten Typs vorkommen (mehrdeutige Typvariablen, siehe Einschränkung 4.21), aus diesem Kompilierschritt entfernt und in den Kind Check verschoben, da erst dort die Klassenumgebung und damit die funktionalen Abhängigkeiten zur Verfügung stehen.

5.3.4. Kind Check

Im Kind Check, welcher die Sortenkorrektheit eines Curry-Programms überprüft, wurden die Funktionen zur Sorteninferenz und -prüfung von Klassendeklarationen, Instanzdeklarationen und Constraints

überarbeitet. Einerseits waren einige kleinere Anpassungen notwendig, damit Typklassen mit ihrer Stelligkeit und dem möglicherweise flexiblen Kontext in die veränderte Klassenumgebung eingetragen werden und die Sorteninferenz in Klassendeklarationen auch mit mehreren Klassenparametern funktioniert. Andererseits wurde eine neue Funktion zur Prüfung der Typklassenstelligkeit für Instanzköpfe und Constraints umgesetzt. Obwohl wir in Definition 4.5 Constraints und somit auch Instanzköpfe als spezielle Typausdrücke definiert haben, können hierzu die bereits bestehenden Funktionen zur Sortenprüfung von Typausdrücken nicht direkt verwendet werden, da Instanzköpfe und Constraints als Klassenname mit einer Liste von Typargumenten statt direkt als Typausdrücke gespeichert sind.

Für die Erweiterung um funktionale Abhängigkeiten wurde, wie im letzten Unterabschnitt zum Type Syntax Check erwähnt, die Prüfung auf mehrdeutige Typvariablen in den Kind Check verschoben. Hierzu wurde die Funktion zur Sortenprüfung von Typsignaturen so erweitert, dass vor der eigentlichen Sortenprüfung die nach Einschränkung 4.21 mehrdeutigen Typvariablen bestimmt werden. Die hierbei benötigten funktionalen Abhängigkeiten zu den im Kontext vorkommenden Typklassen werden der zu diesem Zeitpunkt möglicherweise noch unvollständigen Klassenumgebung entnommen. Bei Klassenmethoden wird das implizite Klassenconstraint bei dieser Prüfung explizit zum Kontext hinzugefügt.

5.3.5. Instance Check

Der Instance Check trägt Instanzdeklarationen in die Instanzenumgebung ein, inferiert Köpfe und Kontexte abzuleitender Instanzen und meldet alle noch nicht in den vorangegangenen Kompilierschritten festgestellten Fehler in Instanzköpfen und -Kontexten. An all diesen Teilen des Instance Checks haben sich Änderungen ergeben, die im Folgenden beschrieben werden:

- Wie in Abschnitt 5.2.4 zu den Veränderungen der Instanzenumgebung beschrieben, müssen Instanzköpfe sowie die dazugehörigen Kontexte nun in expandierter und normalisierter Form in die Umgebung eingetragen werden. Die dafür zuständigen Funktionen im Instance Check wurden dementsprechend angepasst.

- Der Instance Check wendet Kontextreduktion und -Implikation einerseits an, um die Existenz von Oberklasseninstanzen zu prüfen (s. Einschränkung 4.30), und andererseits, um die inferierten Kontexte abzuleitender Instanzen zu vereinfachen. Insbesondere die Kontextreduktion hat sich mit der Einführung von Typklassenerweiterungen deutlich verändert, sodass die entsprechenden Teile des Instance Checks überarbeitet wurden. Näheres zu diesen Änderungen wird im anschließenden Unterabschnitt zum Type Check beschrieben, in dem die Kontextvereinfachung hauptsächlich stattfindet.

- Die Paterson-Bedingungen aus Einschränkung 4.28, welche die Terminierung der Kontextreduktion mit Typklassenerweiterungen gewährleisten, wurden dem Instance Check hinzugefügt. Neben den explizit angegebenen Instanzen werden hierbei auch die im Instance Check inferierten Köpfe und Kontexte automatisch abzuleitender Instanzen überprüft, welche diese Bedingungen ebenfalls verletzen können. Dabei wurde darauf geachtet, dass diese Bedingungen nicht erst mit anderen Instanzprüfungen am Ende des Instance Checks, sondern direkt beim Eintrag in die Instanzenumgebung geprüft werden. Ansonsten würde die während dieses Kompilierschritts durchgeführte Kontextreduktion nicht zwangsläufig terminieren.

- Für die Erweiterung um funktionale Abhängigkeiten wurden dem Instance Check Funktionen zur Prüfung der Einhaltung funktionaler Abhängigkeiten durch Instanzdeklarationen hinzugefügt. Zum einen wird der in Einschränkung 4.33 beschriebene Instanzenkonflikt überprüft, bei dem

eine Instanz einen Instanztyp, der durch eine funktionale Abhängigkeit und eine andere Instanz eindeutig bestimmt sein sollte, mit einem anderen Typausdruck belegt. Zur Erkennung dieser Konflikte wird Typunifikation zwischen Instanzen angewandt. Wichtig ist, dass dabei nicht nur neu angelegte Instanzdeklarationen mit anderen Instanzen verglichen werden müssen, sondern auch importierte Instanzen untereinander solche Konflikte haben können. Nur Instanzpaare, bei denen beide Instanzen aus demselben importierten Modul stammen, werden zur Optimierung nicht überprüft.

Zum anderen wurde eine Prüfung der Typvariablenabdeckung in Instanzköpfen umgesetzt, die in Einschränkung 4.34 beschrieben ist. Diese Typvariablenabdeckung lässt sich wesentlich einfacher als Instanzkonflikte prüfen, da sie sich nur auf einzelne Instanzen bezieht.

5.3.6. Type Check

Der Type Check stellt das Kernstück der Implementierung der Typklassenerweiterungen im Curry-Frontend dar, auch wenn die Änderungen in anderen Teilen des Curry-Frontends keineswegs vernachlässigt werden sollten. Insbesondere bei der Kontextvereinfachung im Type Check wurden während der Entwicklung mehrere verschiedene Ansätze getestet, bis letztendlich das in Definition 4.17 beschriebene Verfahren implementiert wurde. Dieser Abschnitt beschreibt einige Aspekte der Umsetzung dieses Verfahrens und die Unterschiede zur bisherigen Kontextvereinfachung. Außerdem wird kurz auf die Änderungen beim Defaulting von Typvariablen eingegangen, das ebenfalls im Type Check stattfindet:

- Bislang wurde die Kontextreduktion während der Typinferenz bei allen relevanten Veränderungen an der Menge benötigter Constraints aufgerufen. Dadurch konnte zu Fehlern wegen nicht vorhandener Instanzen, die bei dieser Kontextreduktion aufgefallen sind, die problematische Quellcodestelle recht genau gemeldet werden. Aufgrund der mit Typklassenerweiterungen möglichen Instanzüberlappungen und der notwendigen Berücksichtigung explizit gegebener Constraints (siehe nächsten Punkt) musste die Kontextreduktion jedoch von diesen Stellen an den Punkt verschoben werden, an dem die Typinferenz für alle Typvariablen im zu reduzierenden Kontext bereits abgeschlossen ist. Um trotzdem genaue Fehlermeldungen ausgeben zu können, wurde der Type Check auf eine bereits in Abschnitt 5.2.2 vorgestellte Art von Constraints bzw. Prädikaten umgestellt, welche zusätzlich zum eigentlichen Prädikat Daten zur Quellcodestelle enthalten, an der das Prädikat entstanden ist.

- Wie in Definition 4.14 und dem darauffolgenden Text beschrieben, müssen die durch explizite Typsignaturen gegebenen Constraints bei der Kontextreduktion berücksichtigt werden. Ohne Typklassenerweiterungen war dies nicht notwendig, da sich explizit nur nicht reduzierbare Constraints angeben ließen, die am Ende der Typinferenz zu einer Deklaration oder einem explizit getypten Ausdruck mit den inferierten Constraints verglichen werden konnten. Nun werden diese Constraints bereits zu Beginn der Typinferenz der zu prüfenden Spracheinheit in eine dynamische Instanzenumgebung eingetragen. Diese dynamische Umgebung wird im Type Check zusammen mit der statischen Instanzenumgebung aus der Kompilierumgebung mitgeführt und bei der Kontextreduktion verwendet. Sie bestand bereits vor dieser Arbeit im Curry-Frontend und wurde ursprünglich verwendet, um Constraints zu existentiell quantifizierten Typvariablen in Datenkonstruktoren mitzuführen. Dieses Feature ist zum aktuellen Zeitpunkt jedoch deaktiviert. Da die dynamische Instanzenumgebung nun auch Constraints ohne Typvariablen beinhalten kann, z. B. zu nullstelligen Typklassen, bei denen nicht durch die Typvariablenbenennung sichergestellt werden

kann, dass die Constraints nicht außerhalb ihres Sichtbarkeitsbereichs genutzt werden können, musste zudem die Verwendung lokaler Instanzenumgebungen implementiert werden.

- Im Type Check wird in einer hinzugefügten Funktion geprüft, ob für eine Deklaration ohne explizite Typsignatur ein flexibler Kontext inferiert wurde. Falls die Erweiterung zu flexiblen Kontexten nicht aktiviert ist, wird in diesen Fällen eine Fehlermeldung ausgegeben.
- Beim Defaulting von ansonsten mehrdeutigen Typvariablen mit `Num`-Constraint wurden im Type Check wurde bislang nicht die Bedingung (2) aus Definition 4.36 zum Defaulting geprüft, es wurde also Defaulting auch mit selbstdefinierten Typklassen zugelassen. Mit dieser Arbeit wurde das Defaulting dieser Definition entsprechend eingeschränkt, da sich ansonsten die in Abschnitt 4.6 beschriebenen Probleme beim Defaulting ergeben hätten.
- Funktionale Abhängigkeiten wurden letztendlich nie vollständig im Type Check umgesetzt, da das in Abschnitt 5.5 beschriebene Problem mit der Wörterbuchübersetzung während der Überarbeitung des Type Checks zu funktionalen Abhängigkeiten aufgefallen ist. Der Teil, der weitgehend umgesetzt wurde, ist die am Ende von Abschnitt 4.5 beschriebene Erweiterung der freien Variablen um die aufgrund funktionaler Abhängigkeiten eindeutig von ihnen bestimmten Variablen. Noch nicht umgesetzt wurden jedoch die in Definition 4.32 beschriebenen verbessernden Substitutionen, also die durch funktionale Abhängigkeiten gegebenen Möglichkeiten zur Typinferenz. Es ist zu vermuten, dass die Umsetzung dieser Substitutionen weitere Überarbeitungen an der Kontextreduktion erfordert hätte, da sich z. B. durch die Reduktion eines Constraints neue verbessernde Substitutionen ergeben können, die sich auf die Reduktion eines anderen Constraints auswirken. Bei der aktuell implementierten Kontextreduktion wird jedoch davon ausgegangen, dass solche Wechselwirkungen nicht existieren und Constraints somit unabhängig voneinander reduziert werden können.

5.3.7. Warn Check

Der Warn Check, welcher Warnungen zu einem Curry-Programm ausgibt, wurde im Rahmen dieser Arbeit nur geringfügig überarbeitet. Es wurden keine neuen Warnungen hinzugefügt, aber die bestehenden Prüfungen zu redundanten Constraints und Waisen-Instanzen (siehe Abbildung 5.4) wurden an die Kontextimplikation mit potenziell flexiblen Oberklassenconstraints und die potenziell aus mehreren und flexiblen Instanztypen bestehenden Instanzköpfe angepasst.

Für die Erweiterung um funktionale Abhängigkeiten war geplant, die Prüfung auf Waisen-Instanzen für Instanzen zu Klassen mit Abhängigkeiten weiter zu überarbeiten. So muss eine Instanz auch dann als Waise angesehen werden, wenn es eine funktionale Abhängigkeit zu der Klasse der Instanz gibt, für die in den Instanztypen auf der linken Seite der Abhängigkeit kein lokal definierter Typkonstruktor zu einem Datentyp vorkommt. Zu dieser Überarbeitung ist es wegen dem in Abschnitt 5.5 beschriebenen Problem jedoch nicht mehr gekommen.

5.3.8. Dictionary Insertion

Die Wörterbuchübersetzung ist eine bereits in der ursprünglichen Arbeit zu Typklassen [WB89] vorgeschlagene und später verfeinerte [HHPJ+96] Möglichkeit zur Implementierung von Typklassen. Mit der Wörterbuchübersetzung lassen sich die einzelnen Typklassenkomponenten so in Datentypen, Funktionen und zusätzliche Funktionsparameter bzw. -Argumente übersetzen, dass ein Typklassen nutzendes Curry-Programm in ein äquivalentes Curry-Programm ohne Typklassen übersetzt werden kann.

Die im Kompilierschritt Dictionary Insertion des Curry-Frontends implementierte Wörterbuchübersetzung ist in der Arbeit zu Typklassen in Curry ausführlich beschrieben [Tee16]. Die dort beschriebene Übersetzung behandelt Typklassen ohne Erweiterungen und wurde deswegen im Rahmen dieser Arbeit so überarbeitet, dass auch Multiparametertypklassen, flexible Instanzen und flexible Kontexte übersetzt werden können. Dabei war kein grundlegend anderer Übersetzungsansatz, aber zahlreiche Anpassungen an verschiedenen Teilen der Wörterbuchübersetzung notwendig.

In den nächsten Unterabschnitten stellen wir die einzelnen Schritte der Wörterbuchübersetzung anhand einer Variante der in Kapitel 3 verwendeten `ListLike`-Klassen vor. Außerdem gehen wir auf einen wichtigen Aspekt der Überarbeitung, die veränderte Benennung von Wörterbuchkomponenten ein.

Übersetzung von Klassendeklarationen

Eine Klassendeklaration wird bei der Wörterbuchübersetzung in eine Datentypdeklaration, den *Wörterbuchtyp* übersetzt. Dieser Wörterbuchtyp hat die gleichen Parameter wie die Klasse und besteht aus genau einem Datenkonstruktor mit je einem Feld für jedes Oberklassenconstraint und jede Klassenmethode der Klasse. Die Oberklassenconstraints sind dabei in Form des entsprechenden Wörterbuchtypen und die Klassenmethoden in Form ihres beschränkten Typs ohne implizites Klassenconstraint eingetragen.

Für jedes Oberklassenconstraint und jede Klassenmethode wird zudem eine Funktionsdeklaration angelegt, welche die jeweilige Komponente aus dem Wörterbuch extrahiert. Die Extraktionsfunktionen zu Klassenmethoden übernehmen in dieser Übersetzung den Methodennamen, während für die *Oberklassenfunktionen* ein eindeutiger Name generiert wird.

In folgendem Beispiel übersetzen wir eine Variante der `ListLikeM`-Klasse, der in Abschnitt 3.2.3 definierten Multiparameter-Version von `ListLike`. Diese Variante enthält zur Vereinfachung nur die Klassenmethode `insertM` und hat, wie zuerst in Abschnitt 3.5 diskutiert, ein Oberklassenconstraint zu `ListLikeF`, der ListLike-Version zu funktionalen Abhängigkeiten.

Beispiel 5.2 (Übersetzung von Klassendeklarationen)**.**
Die Klassendeklaration

```
class ListLikeF (l e) e => ListLikeM l e where
  insertM :: e -> l e -> l e
```

wird in die folgenden Datentyp- und Funktionsdeklarationen übersetzt:

```
data Dict_ListLikeM l e = Dict_ListLikeM (Dict_ListLikeF (l e) e) (e -> l e -> l e)

super_ListLikeM l e, ListLikeF (l e) e :: Dict_ListLikeM l e -> Dict_ListLikeF (l e) e
super_ListLikeM l e, ListLikeF (l e) e (Dict_ListLikeM d m) = d

insertM :: Dict_ListLikeM l e -> e -> l e -> l e
insertM (Dict_ListLikeM d m) = m
```

Hierbei bezeichnet $\texttt{Dict}_{\texttt{ListLikeF}}$ *den Wörterbuchtypen zu* `ListLikeF`*.*

Übersetzung von Instanzdeklarationen

Eine Instanzdeklaration wird bei der Wörterbuchübersetzung in eine *Wörterbuchfunktion* übersetzt, die ein Wörterbuch passend zu der Klasse und den Instanztypen der Instanz zurückgibt. Dieses Wörterbuch baut diese Funktion aus den zu den Oberklassenconstraints passenden Wörterbuchfunktionen

und den Methodenimplementierungen aus der Instanz zusammen. Letztere werden unter neuem und eindeutigem Namen zu Top-Level-Deklarationen angehoben.

Das nächste Beispiel zeigt die Wörterbuchübersetzung einer `ListLikeM`-Instanz zu beliebigen geordneten Mengen, wobei wir wie in Abschnitt 3.2.1 auf der aus Haskell bekannten Schnittstelle zum `Set`-Typkonstruktor[1] aufbauen.

Beispiel 5.3 (Übersetzung von Instanzdeklarationen)**.**
Die Instanzdeklaration

```
instance Ord e => ListLikeM Set e where
  insertM = Set.insert
```

wird in die folgenden Funktionsdeklarationen übersetzt:

```
inst_ListLikeM Set e :: Ord e => Dict_ListLikeM Set e
inst_ListLikeM Set e = Dict_ListLikeM inst_ListLikeF (Set e) e impl_insertM, ListLikeM Set e

impl_insertM, ListLikeM Set e :: Ord e => e -> Set e -> Set e
impl_insertM, ListLikeM Set e = Set.insert
```

Hierbei bezeichnet $\texttt{inst}_{\texttt{ListLikeF (Set e) e}}$ *die Wörterbuchfunktion zu einer Instanz mit dem Kopf* `ListLikeF (Set e) e`*.*

An diesem Beispiel lässt sich erkennen, dass die Kontexte von Wörterbuchfunktionen und Methodenimplementierungen in diesem Schritt noch nicht übersetzt werden. Dies geschieht zusammen mit der Übersetzung anderer Variablendeklarationen beim Einsetzen von Wörterbüchern im nächsten Schritt.

Einsetzen von Wörterbüchern

In diesem Schritt werden Wörterbücher auf zwei Arten eingesetzt. Auf der linken Seite von Funktionsregeln wird ein zusätzlicher *Wörterbuchparameter* für jedes Constraint im Kontext des Funktionstyps ergänzt. Dementsprechend werden im Funktionstyp die Wörterbücher zu den Constraints als zusätzliche Argumenttypen ergänzt und der Kontext entfernt. Auf der rechten Seite von Funktionsregeln werden zu jeder Variable *Wörterbuchargumente* für die Constraints aus dem Kontext des Variablentyps hinzugefügt. Die hierfür benötigten Wörterbücher werden mithilfe der Oberklassenfunktionen und der Wörterbuchfunktionen zu Instanzen aus den gegebenen Wörterbuchparametern konstruiert.

Folgendes Beispiel zeigt die Übersetzung der Methodenimplementierung aus dem letzten Beispiel sowie die Übersetzung einer weiteren Funktion, an der die Konstruktion von Wörterbuchargumenten mit Wörterbuchfunktionen zu erkennen ist.

Beispiel 5.4 (Einsetzen von Wörterbüchern)**.**
Die Funktionsdeklarationen

```
impl_insertM, ListLikeM Set e :: Ord e => e -> Set e -> Set e
impl_insertM, ListLikeM Set e = Set.insert

insertTrue :: Set Bool -> Set Bool
insertTrue = insertM True
```

werden in die folgenden Funktionsdeklarationen übersetzt:

[1] https://hackage.haskell.org/package/containers/docs/Data-Set.html

```
impl_{insertM, ListLikeM Set e} :: Dict_{Ord} e -> e -> Set e -> Set e
impl_{insertM, ListLikeM Set e} d = Set.insert d

insertTrue :: Set Bool -> Set Bool
insertTrue = insertM (inst_{ListLikeM Set e} inst_{Ord Bool}) True
```

Hierbei hat `Set.insert` *den Typ* `Ord a => a -> Set a -> Set a`, $\texttt{Dict}_{\texttt{Ord}}$ *bezeichnet den Wörterbuchtyp zur Klasse* `Ord` *und* $\texttt{inst}_{\texttt{Ord Bool}}$ *bezeichnet die Wörterbuchfunktion zur* `Ord`*-Instanz mit dem Instanztyp* `Bool`*.*

Benennung von Wörterbuchkomponenten

Für Wörterbuchtypen, Oberklassenfunktionen, Wörterbuchfunktionen und Methodenimplementierungen wurden in den vorherigen Unterabschnitten konzeptionelle Bezeichner wie $\texttt{Dict}_{\texttt{ListLikeM}}$ verwendet. Die Namen, die im Curry-Frontend tatsächlich für diese Wörterbuchkomponenten verwendet werden, beinhalten jedoch keine tiefgestellten Zeichen und müssen teilweise Originalnamen (s. Abbildung 5.5) für Typkonstruktoren verwenden, um garantiert eindeutig zu sein.

Im bisherigen Benennungsschema für diese Wörterbuchkomponenten wurden Instanztypen in Form ihres Typkonstruktornamens und Oberklassenconstraints in Form des Oberklassennamens dargestellt. Für Typklassenerweiterungen ist dieses Benennungsschema unzureichend und der Instanzkopf bzw. das Oberklassenconstraint wird stattdessen vollständig und in expandierter Form dargestellt. Folgendes Beispiel zeigt die im Frontend verwendeten Namen nach dem bisherigen und dem neuen Benennungsschema:

Beispiel 5.5 (Benennung von Wörterbuchkomponenten)**.**
Wir gehen davon aus, dass die Klassen `ListLikeM`*,* `ListLikeF` *sowie die folgenden Klassen und Instanzen im Modul* `M` *deklariert sind.*

```
class C a b

instance C (Maybe a) [a]
instance C (Maybe a) [b]
```

Die anschließende Tabelle zeigt nun, welche Namen anstelle der konzeptionellen Bezeichner vom Curry-Frontend für Wörterbuchkomponenten generiert werden. In der rechten Spalte wird dabei zuerst der Name nach dem bisherigen Benennungsschema und dann, falls er sich unterscheidet, der Name nach dem neuen Schema genannt.

Konzeptioneller Bezeichner	Bisherige / Neue Benennung
$\texttt{Dict}_{\texttt{ListLikeM}}$	`_Dict#ListLikeM`
$\texttt{super}_{\texttt{ListLikeM l e, ListLikeF (l e) e}}$	`_super#M.ListLikeM#M.ListLikeF` `_super#M.ListLikeM#M.ListLikeF(0(1))(1)`
$\texttt{inst}_{\texttt{ListLikeM Set e}}$	`_inst#M.ListLikeM#Data.Set.Set#` `_inst#M.ListLikeM(Data.Set.Set)(0)`
$\texttt{impl}_{\texttt{insertM, ListLikeM Set e}}$	`_impl#insertM#M.ListLikeM#Data.Set.Set#` `_impl#insertM#M.ListLikeM(Data.Set.Set)(0)`
$\texttt{inst}_{\texttt{C (Maybe a) [a]}}$	`_inst#M.C#Prelude.Maybe#[]` `_inst#M.C(Prelude.Maybe(0))([](0))`
$\texttt{inst}_{\texttt{C (Maybe a) [b]}}$	`_inst#M.C#Prelude.Maybe#[]` `_inst#M.C(Prelude.Maybe(0))([](1))`

Die Verwendung von Sonderzeichen in diesen Namen, die in Curry nicht für Bezeichner erlaubt sind, ist an dieser Stelle unproblematisch, da die Bezeichner später im Backend von PAKCS bzw. KiCS2 in Bezeichner umgewandelt werden, die in der jeweiligen Zielsprache gültig sind. Typvariablen werden in der neuen Benennung durch Zahlen repräsentiert, um Konflikte mit Typkonstruktoren auszuschließen. Klammern werden in Typausdrücken einheitlich als Trennzeichen verwendet, da die Struktur der Ausdrücke so eindeutig erhalten bleibt.

Die letzten beiden konzeptionellen Bezeichner aus dem Beispiel zeigen Situationen, in denen das bisherige Benennungsschema offensichtlich nicht zur eindeutigen Benennung ausreicht, da verschiedenen Instanzen die gleichen Bezeichner zugewiesen werden. Ein ähnlicher Fall kann auch bei Oberklassenconstraints auftreten.

5.4. Import und Export von Modulen

Curry verfügt über ein Modulsystem, welches den Import und Export von Deklarationen über Schnittstellen ermöglicht. Ein Überblick über den Ablauf von Import und Export wird in Abschnitt 5.1.4 gegeben. In diesem Abschnitt werden wir die wesentlichen Änderungen behandeln, die im Rahmen der Implementierung von Typklassenerweiterungen an den Schritten dieses Ablaufs durchgeführt wurden.

5.4.1. Interface Syntax Check

Der Interface Syntax Check prüft Schnittstellen auf grundlegende Fehler in den Typausdrücken und trennt Typvariablen von Typkonstruktoren. Aufgrund der damit gegebenen großen Ähnlichkeit zum Type Syntax Check ist auch ein Großteil der durchgeführten Anpassungen am Interface Syntax Check analog zu den in Abschnitt 5.3.3 beschriebenen Änderungen am Type Syntax Check. Ein wichtiger Unterschied zwischen den beiden Prüfungsschritten ist jedoch, dass Schnittstellen keine Informationen zu aktivierten Spracherweiterungen enthalten, auf die der Interface Syntax Check zugreifen könnte. Deshalb wird in diesem Prüfungsschritt davon ausgegangen, dass alle unterstützten Typklassenerweiterungen aktiviert sind. Die Prüfungen zur Form von Instanzköpfen und Constraints fielen dadurch letztendlich weg.

5.4.2. Berechnung der Export-Schnittstelle

Bei der Berechnung der exportierten Schnittstelle wurden im Rahmen dieser Arbeit insbesondere die Kriterien überarbeitet, nach denen die exportierten Instanzdeklarationen ausgewählt werden. In diesem Aspekt unterscheidet sich die tatsächliche Implementierung klar von der Sprachdokumentation: Sowohl in der Arbeit zu Typklassen in Curry [Tee16] als auch im Haskell-Report[2] ist explizit angegeben, dass ein Modul alle Instanzen exportiert, die in diesem Modul zur Verfügung stehen bzw. sichtbar sind.

Praktisch würde diese Vorgehensweise aber für unnötig große Schnittstellen sorgen. So sind z. B. in jedem Modul, welches den Import des `Prelude`-Moduls nicht durch eine Spracherweiterung deaktiviert, die aktuell über 100 in `Prelude` deklarierten Instanzen sichtbar. Nehmen wir für ein solches Modul namens `M` an, dass keine der in `Prelude` deklarierten Klassen direkt von `M` exportiert wird oder im Typ einer von `M` exportierten Deklaration (abseits der `Prelude`-Instanzen) vorkommt. Ein Modul `N`, welches ausschließlich `M` und nicht `Prelude` importiert, könnte nun keine der von `M` potenziell exportierten

[2] `https://www.haskell.org/onlinereport/haskell2010/haskellch5.html#x11-1060005.4`

Instanzen aus `Prelude` zur Kontextreduktion nutzen, da in `N` auf keine Weise ein Constraint zu einer `Prelude`-Klasse entstehen kann. Die Instanzen könnten nur dann in `N` verwendet werden, wenn eine solche Klasse durch ein anderes Modul importiert würde.

Dieses Beispiel weist darauf hin, dass der Export von Instanzen an den Export von Typklassen geknüpft werden kann: Dazu gehen wir davon aus, dass jede `Prelude`-Instanz immer zusammen mit der jeweiligen Klasse exportiert wird. Wenn nun in `N` zur Reduktion eines Constraints eine `Prelude`-Instanz benötigt wird, muss durch den Import, durch den die Klasse des Constraints in `N` sichtbar wurde, auch die benötigte Instanz importiert worden sein. Somit können die Exporte von `Prelude`-Instanzen bei einem Modul wie `M` vermieden werden, ohne dass es dazu kommen kann, dass eine benötigte Instanz aufgrund unzureichender Exporte nicht verfügbar ist.

Auf ähnliche Weise wie mit Typklassen können Instanzen unter Umständen mit Typkonstruktoren aus den Instanztypen verknüpft werden. Die vollständigen Kriterien, die zur Minimierung unnötiger Instanzenexporte entwickelt und im Curry-Frontend umgesetzt wurden, sind in folgender Definition beschrieben:

Definition 5.6 (Kriterien für den Export von Instanzdeklarationen)**.**
Eine Instanz wird genau dann exportiert, wenn sie

- *im aktuellen Modul deklariert ist und alle im Instanzkopf vorkommenden Typkonstruktoren, die ebenfalls im aktuellen Modul deklariert sind, exportiert werden;*
- *aus einem anderen als dem aktuellen Modul stammt und der beim Lesen von links erste Typkonstruktor aus dem Instanzkopf, der im selben Modul wie die Instanz deklariert ist, exportiert wird;*
- *eine Waisen-Instanz (s. Abbildung 5.4) ist*
- *oder zu einer Klasse mit funktionalen Abhängigkeiten gehört und die Klasse exportiert wird oder in einem anderen Modul als die Instanz deklariert ist.*

Wir gehen dabei von expandierten Instanzköpfen aus und zählen Typklassen zu den Typkonstruktoren.

5.5. Problem bei der Umsetzung funktionaler Abhängigkeiten

Funktionale Abhängigkeiten konnten in der Erkennungs- und in den meisten Schritten der Prüfungsphase des Curry-Frontends erfolgreich umgesetzt werden, wobei die Überarbeitung des Type Checks noch nicht abgeschlossen wurde, wie in Abschnitt 5.3.6 beschrieben. Zu diesem Zeitpunkt ist bei der testweisen Kompilation einfacher Programme mit funktionalen Abhängigkeiten ein Problem in der Wörterbuchübersetzung bzw. der Dictionary Insertion (s. Abschnitt 5.3.8) in Zusammenhang mit funktionalen Abhängigkeiten aufgefallen. Nach genauerer Analyse des Problems wurde es als zu schwerwiegend eingestuft, um in der fortgeschrittenen Phase der Entwicklung noch behoben zu werden. In diesem Abschnitt werden wir zuerst in Abschnitt 5.5.1 das aufgetretene Problem vorstellen und dann in Abschnitt 5.5.2 Lösungsansätze und die mit ihnen verbundenen Schwierigkeiten vorstellen.

5.5.1. Vorstellung des Problems

Das Problem, das bei der Wörterbuchübersetzung in Zusammenhang mit funktionalen Abhängigkeiten auftritt, hängt mit den Typen zusammen, die während des Type Checks im AST eingetragen werden. Funktionsdeklarationen werden beim Type Check im AST mit dem beschränkten Typ aus dem in die Werteumgebung eingetragenen Typschema annotiert. Dieser beschränkte Typ ist in der Praxis gleichwertig zum Typschema, da die allquantifizierten Typvariablen über ihre Indizes von den ungebundenen Typvariablen unterschieden werden können.

Zu Datenkonstruktoren und Variablen in Mustern und Ausdrücken wird hingegen der Typ, den diese Variable in der jeweiligen Verwendung hat, ohne Kontext eingetragen. So hat z. B. der Operator `(==)` das Typschema `∀ a . Eq a => a -> a -> Bool`, im Ausdruck `True == True` wird `==` aber mit dem Typ `Bool -> Bool -> Bool` annotiert. Als Typvariablen kommen in diesen als *monomorph* bezeichneten Typen die während der Typinferenz verwendeten Typvariablen vor. Diese Typvariablen sind zwischen den Variablen und Datenkonstruktoren im Ausdruck einheitlich, unterscheiden sich aber von den allquantifizierten Typvariablen aus dem beschränkten Typ der Funktionsdeklaration.

Als Beispiel für die im AST eingetragenen Typen betrachten wir die Funktion `f` aus folgendem Programmcode:

```
class C a b | a -> b where
methodC :: a -> b

f :: (C a c, C c b) => a -> b
f = methodC . methodC
```

Im beschränkten Typ der Funktion `f` kommt die Typvariable `c` nur im Kontext vor, sie ist laut Einschränkung 4.21 aber nicht mehrdeutig, da sie über das Constraint `C a c` und die funktionale Abhängigkeit von `C` eindeutig durch `a` bestimmt ist. Die folgende Tabelle listet zu `f` und jeder Variable im Körper von `f` den im AST annotierten Typ und das in der Werteumgebung eingetragene Typschema der zugehörigen Variablendeklaration auf:

Variable	Annotierter (beschränkter) Typ	Typschema
`f`	`(C a c, C c b) => a -> b`	`∀ a b c . (C a c, C c b) => a -> b`
`methodC` (links)	`z -> y`	`∀ a b . C a b => a -> b`
`.`	`(z -> y) -> (x -> z) -> x -> y`	`∀ a b c . (a -> b) -> (c -> a) -> c -> b`
`methodC` (rechts)	`x -> z`	`∀ a b . C a b => a -> b`

In dieser Tabelle bezeichnen `x`, `y` und `z` die während der Typinferenz verwendeten monomorphen Typvariablen.

Wir gehen nun die Schritte der in Abschnitt 5.3.8 in vereinfachter Form behandelten Einsetzung von Wörterbüchern am Beispiel von `f` im Detail durch:

1. Um beim Einsetzen der Wörterbuchargumente auf der rechten Seite der Funktionsregel von `f` die benötigten und die gegebenen Wörterbücher einander korrekt zuordnen zu können, werden zuerst die monomorphen Varianten der Constraints im Kontext von `f` bestimmt. Hierzu wird der Typ aus dem Typschema von `f` ohne Kontext auf den monomorphen Typ von `f` gematcht. Dieser monomorphe Typ wird über die im AST annotierten Typen zu den Mustern auf der linken Regelseite und den Variablen und Datenkonstruktoren auf der rechten Regelseite bestimmt und ist in diesem Fall `x -> y`. Das Typmatching berechnet nun die Substitution σ mit $\sigma(\texttt{a -> b}) = \texttt{x -> y}$. Durch Anwendung der ermittelten Substitution $\sigma := \{\texttt{a} \mapsto \texttt{x}, \texttt{b} \mapsto \texttt{y}\}$ auf den Kontext des Typschemas von `f` erhalten wir die Constraints `C x c` und `C c y`. Wichtig ist, dass diese Constraints nicht neu sortiert, sondern in der Reihenfolge des Typschema-Kontexts belassen werden.

2. Für die Constraints aus dem letzten Schritt wird je ein Wörterbuchparameter auf der linken Regelseite ergänzt. `f = ...` wird dadurch zu `f` $\texttt{dict}_0$ $\texttt{dict}_1$ `= ...` erweitert. Zum späteren Zugriff auf die Wörterbücher werden die hinzugefügten Parameter unter dem zugehörigen Constraint in eine *Wörterbuchumgebung* eingetragen. Diese Umgebung sieht in unserem Beispiel folgendermaßen aus: $\{\texttt{C x c} \mapsto \texttt{dict}_0, \texttt{C c y} \mapsto \texttt{dict}_1\}$

3. Nun folgt das Einsetzen von Wörterbuchargumenten auf der rechten Regelseite. Hierzu werden zu jeder Variable die monomorphen Constraints bestimmt, genau wie zu `f` im ersten Schritt. Bei der zuerst transformierten äußersten Variable, der Funktionskomposition, ist der Kontext leer, weswegen keine Argumente ergänzt werden müssen. Für das danach übersetzte linke Vorkommen von `methodC` wird durch das Verfahren aus dem ersten Schritt das Constraint `C z y` ermittelt.

4. Zu den Constraints aus dem letzten Schritt werden nun die passenden Wörterbuchargumente konstruiert. Dabei wird zunächst die Wörterbuchumgebung nach dem benötigten Constraint durchsucht. `C z y` ist jedoch ungleich zu `C x c` und `C c y` und somit nicht in der Wörterbuchumgebung enthalten, weshalb als nächstes die Instanzenumgebung nach einer passenden Instanz durchsucht wird. In unserem Beispiel existiert eine solche Instanz nicht, weswegen die Konstruktion des passenden Wörterbucharguments gescheitert ist und ein Fehler zurückgegeben wird.

Das im obigen Verfahren aufgetretene Problem lässt sich darauf zurückführen, dass im ersten Schritt die Typvariable `c` nicht im Typ der rechten Regelseite enthalten war und somit nicht auf `z` gematcht wurde. Dieser Zusammenhang zwischen `c` und `z` kann bis zur Transformation des linken `methodC` auch unter Berücksichtigung funktionaler Abhängigkeiten nicht hergestellt werden:

Die Constraints aus der Wörterbuchumgebung zusammen mit dem benötigten Constraint sind zu dem Zeitpunkt `C x c`, `C c y` und `C z y`. Laut Definition 4.32 kann mit diesen Constraints keine verbessernde Substitution abgeleitet werden, da die Typen auf der linken Seite der funktionalen Abhängigkeit von `C`, also `x`, `c` und `y`, zwischen keinem Paar von Constraints übereinstimmen. Erst mit dem Constraint `C x z`, welches für das rechte Vorkommen von `methodC` benötigt wird, kann zusammen mit dem Constraint `C x c` die Gleichheit zwischen `c` und `z` festgestellt werden.

5.5.2. Problemlösungsansätze

Das im letzten Abschnitt beschriebene Problem entsteht durch die mit funktionalen Abhängigkeiten gegebene Möglichkeit, dass Typvariablen nur im Kontext eines beschränkten Typs vorkommen. Der bisher bei der Wörterbuchübersetzung verfolgte Ansatz, die gegebenen bzw. benötigten Wörterbücher nur auf Basis des unbeschränkten monomorphen Typs zu bestimmen, ist damit nicht mehr ausreichend, da die gesuchte Benennung dieser Typvariablen nicht mehr eindeutig aus diesen Daten abgeleitet werden kann. Der Umstieg auf die Verwendung beschränkter monomorpher Typen birgt allerdings weitere Schwierigkeiten, die in diesem Abschnitt vorgestellt werden. Wir betrachten dabei zuerst die die linke Regelseite, auf der die beschränkten Typen zur Bestimmung der gegebenen Wörterbücher benötigt werden, und dann die rechte Regelseite, auf der Variablen mit beschränkten Typen annotiert werden müssen, um die benötigten Wörterbücher zu bestimmen.

Linke Regelseite

Das im konkreten Beispiel aus Abschnitt 5.5.1 auftretende Problem könnte damit gelöst werden, zur Funktionsdeklaration `f` im AST den inferierten monomorphen beschränkten Typ `(C x z, C z y) => x -> y` einzutragen und diesen zum Einsetzen der Wörterbuchparameter zu verwenden.

Neben dem direkten Problem, dass im AST aktuell kein freies Feld existiert, das für diesen monomorphen beschränkten Typ verwendet werden kann, gibt es das schwerwiegendere Problem, dass der inferierte Kontext nicht unbedingt direkt äquivalent zum Kontext des zugehörigen Typschemas aus der Werteumgebung ist. Ein einfaches Beispiel dafür sehen wir, wenn wir die Benennung der Typvariablen `x` und `z` aus dem monomorphen beschränkten Typ von `f` tauschen. Dadurch erhalten wir den beschränkten Typ `(C z x, C x y) => z -> y`, wobei der Kontext aufgrund der internen

Darstellung als sortierte Menge die Reihenfolge `(C x y, C z x)` hätte. Würden Wörterbuchparameter basierend auf diesem Kontext ergänzt, wären die Wörterbuchparameter im Vergleich zum Typ, welcher der übersetzten Funktion zugewiesen wird, in der verkehrten Reihenfolge.

Dieser Fehler kann im Falle einer Variablendeklaration mit expliziter Typsignatur, wo sich der Kontext aus dem Typschema aufgrund von Oberklassen und der Kontextimplikation vom inferierten Kontext unterscheiden kann, noch komplizierter ausfallen. Um diesen Fehler zu verhindern, muss entweder beim Type Check der monomorphe Kontext dem Kontext aus dem Typschema angeglichen werden und z. B. in Form einer Liste statt einer Menge eingetragen werden, oder die Wörterbuchübersetzung um gewisse Komponenten zur Typinferenz erweitert werden. Bei letzterem Ansatz könnte zunächst wie bisher Typmatching durchgeführt werden, um wie im Beispiel die Constraints `C x c` und `C c y` zu erhalten. Basierend auf diesen und den Constraints aus dem monomorphen Kontext könnten funktionale Abhängigkeiten angewandt werden, um Typgleichungen zu inferieren. In unserem Beispiel würde zusammen mit dem monomorphen Kontext `(C x z, C z y)` inferiert werden, dass `c` und `z` für den gleichen Typ stehen müssen. Die Substitution, die sich durch die Lösung dieser Gleichungen ergibt, müsste dann während der Einsetzung von Wörterbüchern mitgeführt werden.

Rechte Regelseite

Im betrachteten Beispiel aus Abschnitt 5.5.1 treten auf der rechten Regelseite von `f` keine Variablen auf, für welche die benötigten Wörterbücher nicht mit dem aktuell verwendeten Verfahren korrekt bestimmt werden können. Ein solcher Fall könnte sich aber z. B. ergeben, wenn die Funktion `f` selbst auf der rechten Seite einer Funktionsregel vorkäme.

Wenn auf der rechten Regelseite zu jeder Variable der beschränkte monomorphe Typ annotiert und dieser zur Bestimmung der benötigten Wörterbücher verwendet wird, ergeben sich einerseits ähnliche Probleme wie auf der linken Regelseite: Aufgrund der möglicherweise verkehrten Constraintreihenfolge müsste entweder der monomorphe Kontext im Type Check z. B. in Form einer Liste eingetragen werden oder bei der Wörterbuchübersetzung das Typschema herangezogen und Typinferenz mit funktionalen Abhängigkeiten durchgeführt werden.

Ein eigenes Problem auf der rechten Regelseite entsteht jedoch, wenn eine rekursive Funktion ohne explizite Typsignatur einen nichtleeren Kontext hat. Als Beispiel hierfür betrachten wir die folgende Variante der `elem`-Funktion ohne explizite Typsignatur:

```
elem _ []       = False
elem x (y : ys) = x == y || elem x ys
```

In diesem Beispiel steht zu dem Zeitpunkt, zu dem die Variable `elem` im Körper der Funktion in der Typinferenz behandelt wird, das vollständige Typschema von `elem` nicht in der Werteumgebung zur Verfügung. Deswegen kann zu diesem Zeitpunkt der monomorphe Typ der Variable nur ohne Kontext im AST eingetragen werden. Da Substitutionen Typvariablen nur auf unbeschränkte Typen abbilden, kann der Kontext später auch nicht auf diese Weise einfach hinzugefügt werden.

Stattdessen kann man sich die Eigenschaft zunutze machen, dass eine Funktion in einer Rekursion dieser Art nicht polymorph verwendet werden kann. Die Variable `elem` auf der rechten Seite der zweiten Funktionsregel muss also den gleichen monomorphen Typ wie `elem` auf den linken Seiten der Funktionsregeln haben. Damit können wir bei der Wörterbuchübersetzung davon ausgehen, dass die von `elem` auf der rechten Seite benötigten Constraints bereits mit der richtigen Typvariablenbenennung in der Wörterbuchumgebung stehen. Deswegen sollte es auch ohne im AST annotierten monomorphen Kontext möglich sein, mithilfe der mit dem Typschema und Typmatching bestimmten benötigten Constraints, den Constraints aus der Wörterbuchumgebung und den dadurch gegebenen funktionalen Abhängigkeiten die korrekten monomorphen Constraints zu inferieren.

Kapitel 6

Fazit

Dieses abschließende Kapitel fasst die Ergebnisse dieser Arbeit zusammen (Abschnitt 6.1) und behandelt dann kurz mögliche weiterführende Arbeiten und alternative Ansätze (Abschnitt 6.2).

6.1. Ergebnisse

Im Rahmen dieser Arbeit wurden verschiedene Spracherweiterungen zu Typklassen im Curry-Frontend umgesetzt. Mit der Erweiterung zu Multiparametertypklassen können Typklassen mit einer beliebigen Zahl an Klassenparametern deklariert werden, die Erweiterung zu flexiblen Instanzen lockert die Einschränkungen, die bei der Angabe von Typklasseninstanzen eingehalten werden müssen, und die Erweiterung zu flexiblen Kontexten erlaubt beliebige sortenkorrekte Typausdrücke in Constraints. Die Erweiterung zu funktionalen Abhängigkeiten, mit welcher Beziehungen zwischen Klassenparametern ausgedrückt werden können, wäre als Ergänzung zu diesen Erweiterungen wünschenswert gewesen, konnte allerdings aufgrund eines bei der Implementierung aufgefallenen Problems nicht vollständig umgesetzt werden.

Auch die umgesetzten Typklassenerweiterungen bieten jedoch zahlreiche neue Möglichkeiten zur Anwendung von Typklassen, wie in dieser Arbeit anhand motivierender Beispiele gezeigt wurde. Mit den Erweiterungen können zu einem höheren Grad polymorphe Funktionen angegeben und viele Programmierprobleme gelöst werden, für die bislang Codeduplikation oder restriktivere bzw. kompliziertere Alternativlösungen notwendig waren. Außerdem können Haskell-Programme, welche die vom GHC ebenfalls unterstützten Typklassenerweiterungen verwenden, einfacher nach Curry übertragen werden. Hierzu trägt bei, dass auf eine große Ähnlichkeit im Verhalten der Erweiterungen zwischen dem Curry-Frontend und dem GHC geachtet wurde.

Die Ergänzung dieser Typklassenerweiterungen zu Curry und dem Curry-Frontend bestand einerseits aus der Erweiterung der syntaktischen Möglichkeiten für Klassendeklarationen, Instanzdeklarationen und Constraints. Der wesentlich wichtigere Aspekt der Überarbeitung waren jedoch die neuartigen Situationen und Probleme, die sich bei der Behandlung dieser Typklassenelemente ergeben können. Diese Fälle, die bei einstelligen Typklassen ohne flexible Instanzen oder Kontexte durch die automatisch gegebenen Einschränkungen nicht auftreten konnten, mussten in erweiterten Prozeduren behandelt oder durch neue Einschränkungen abgefangen werden. In dieser Arbeit wurden sowohl formale Definitionen zu diesen Prozeduren und Einschränkungen gegeben als auch ihre Implementierung im Frontend beschrieben.

Die Überarbeitungen dieser Arbeit wurden auf einem eigenen Zweig des Git-Repositoriums zum Curry-Frontend durchgeführt[1]. Die unvollständige Umsetzung funktionaler Abhängigkeiten ist darin nicht enthalten und wurde stattdessen in einem anderen Zweig festgehalten[2]. Von den umgesetzten Typklassenerweiterungen wurden insbesondere Multiparametertypklassen zusammen mit dem Backend

[1] https://git.ps.informatik.uni-kiel.de/curry/curry-frontend/-/tree/lkrueger-ma

[2] https://git.ps.informatik.uni-kiel.de/curry/curry-frontend/-/tree/lkrueger-ma-fundeps

von Version 3.3.0 des PAKCS an verschiedenen Curry-Programmen getestet, durch die zahlreiche Fehlermöglichkeiten abgedeckt wurden. Zudem wurden zahlreiche Testfälle zu Multiparametertypklassen zur internen Testsuite des Curry-Frontends hinzugefügt.

6.2. Weiterführende Arbeiten

Die folgende Liste behandelt verschiedene Ergänzungen zu Typklassen, die auf dieser Arbeit aufbauend im Curry-Frontend umgesetzt werden können, und stellt mit *Typfamilien* eine Alternative zu funktionalen Abhängigkeiten kurz vor:

- Funktionale Abhängigkeiten konnten zwar in dieser Arbeit nicht vollständig umgesetzt werden, scheinen aber nicht grundsätzlich inkompatibel zum Curry-Frontend zu sein. Durch die erfolgreich umgesetzten Multiparametertypklassen und die erfolgte Vorarbeit bieten sich funktionale Abhängigkeiten nun mehr als zuvor als Erweiterung für das Curry-Frontend an. Neben den in Abschnitt 5.5.2 beschriebenen Lösungsansätzen zum bei der Wörterbuchübersetzung aufgetretenen Problem sollten bei einer erneuten Umsetzung auch die in Abschnitt 5.3.6 erwähnten erforderlichen Überarbeitungen am Type Check beachtet werden.

- Im Rahmen dieser Arbeit wurden keine neuen Warnungen zum Warn Check des Curry-Frontends hinzugefügt, allerdings bietet sich die Ergänzung weiterer Warnungen mit Bezug zu Typklassen an:

  - Durch die umgesetzten Typklassenerweiterungen lassen sich Instanzen wie `instance Eq String` angeben, die niemals angewandt werden können, da sie sich zwangsläufig mit anderen Instanzen überlappen, wie in diesem Fall mit der Instanz zu `Eq [a]`. Da der in solchen Instanzen angegebene Code somit unerreichbar ist, könnte bei der Deklaration der Instanz eine Warnung ausgegeben werden.
  - In expliziten Typsignaturen lassen sich Constraints angeben, die direkt mit einer Instanz reduziert werden können. Solche Constraints machen die in Abschnitt 4.3 beschriebene Berücksichtigung der gegebenen Constraints bei der Kontextreduktion erst notwendig und stellen aktuell keinen Mehrwert gegenüber der Angabe der reduzierten Constraints dar. In Kombination mit funktionalen Abhängigkeiten könnte sich durch solche Constraints auch ein Konfluenzproblem bei der Typinferenz ergeben: Ein Constraint könnte mit einer Instanz reduziert werden, obwohl sich bei der Reduktion anderer Constraints später eine verbessernde Substitution ergibt, mit der dieses Constraint identisch zu einem explizit angegebenen Constraint ist und deswegen nicht mit der Instanz hätte reduziert werden dürfen. Aus diesem Grund könnte zu solchen Constraints eine Warnung ausgegeben werden.
  - Es gibt aktuell bereits eine Warnung zu redundanten Constraints, allerdings werden hierbei nur Constraints gemeldet, die durch andere Constraints im selben Kontext bereits impliziert werden. Diese Warnung könnte erweitert werden, sodass auch Constraints in expliziten Typsignaturen gemeldet werden, die im Körper der dazugehörigen Funktionsdeklaration nicht benötigt werden.

- Wenn bei der Kontextreduktion mehrere Instanzen zu einem Constraint passen, wird dieses Constraint nicht reduziert und, falls die Reduktion des Constraints erforderlich ist, stattdessen ein Fehler wegen überlappender Instanzen gemeldet. Alternativ könnte in solchen Fällen aber auch die spezifischste oder eine beliebige der passenden Instanzen zur Reduktion verwendet werden. Für den Fall, dass ein solches Verhalten gewünscht ist, bietet der GHC Möglichkeiten an, dies über eine

Spracherweiterung für alle Instanzen eines Moduls oder über Pragmas zu einzelnen Instanzdeklarationen zu spezifizieren[3]. Diese Optionen könnten auch im Curry-Frontend umgesetzt werden. Dabei sollten Programmierer jedoch darauf aufmerksam gemacht werden, dass diese Optionen bei unvorsichtiger Anwendung möglicherweise unerwünschtes Programmverhalten bewirken können. Zum Beispiel kann die gewählte Instanz und damit die verwendeten Methodenimplementierungen so von den in einem Modul sichtbaren Instanzen abhängen, dass sich allein durch die Ergänzung oder Entfernung von Importdeklarationen das Programmverhalten deutlich verändert.

- In einigen Fällen kann die Einschränkung 4.19, welche die Beschränkung von Klassenparametern durch Klassenmethoden verbietet, ein Programmierhindernis sein. Beispielsweise könnte ohne diese Einschränkung eine `elem`-Methode mit `Eq`-Constraint zu den `ListLike`-Klassen aus Kapitel 3 oder anderen Klassen von Sammlungstypen hinzugefügt werden. Der GHC bietet hierzu eine Spracherweiterung an, welche diese Einschränkung aufhebt und bei Verwendung der Erweiterung zu Multiparametertypklassen sogar automatisch aktiviert wird[4]. Die Umsetzung einer solchen Erweiterung im Curry-Frontend sollte sich einfach gestalten, da es nach den Überarbeitungen der internen Sortierung von Kontexten (s. Abschnitt 5.2.2) wie in Haskell auch [JJM97] keine technischen Gründe für diese Einschränkung mehr zu geben scheint.
- Im Vergleich zu funktionalen Abhängigkeiten stellen Typfamilien [SPJC+08] einen alternativen Ansatz zur Angabe von Beziehungen zwischen Typen dar. Typfamilien sind als Haskell-Spracherweiterung im GHC implementiert und bestehen aus Typfamilien- und Instanzdeklarationen, ähnlich wie Typklassen. Eine solche Instanz weist einer Typfamilie abhängig von der Form der Argumenttypen einen Rückgabetyp zu. Damit können Typfamilien auch als Funktionen auf Typen angesehen werden, denen durch Instanzdeklarationen neue Funktionsregeln hinzugefügt werden. Funktionale Abhängigkeiten sind auf eine ähnliche Weise anwendbar, da einzelne Argumenttypen eines Constraints über die zur Form der gegebenen Argumenttypen passende Typklasseninstanz inferiert werden können (siehe Definition 4.32). Trotz dieser Gemeinsamkeiten gibt es zahlreiche Unterschiede zwischen Typfamilien und funktionalen Abhängigkeiten[5]: So lassen sich z. B. gegenseitige Abhängigkeiten direkt nur mit funktionalen Abhängigkeiten ausdrücken, während Typfamilien in Kombination mit existentiell quantifizierten Typvariablen, die aktuell nicht vom Curry-Frontend unterstützt werden, besser funktionieren.

[3] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts/instances.html#overlapping-instances
[4] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts/constrained_class_methods.html
[5] https://gitlab.haskell.org/ghc/ghc/-/wikis/tf-vs-fd

Literatur

[CHO92] Kung Chen, Paul Hudak und Martin Odersky. „Parametric Type Classes“. In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. 1992, S. 170–181.

[GHC21] GHC Team. *GHC User's Guide Documentation (Release 9.0.1)*. Verfügbar unter `https://downloads.haskell.org/ghc/latest/docs/users_guide.pdf`. 2021.

[Han+16] Michael Hanus (Hrsg.) u. a. *Curry: An Integrated Functional Logic Language (Version 0.9.0)*. Verfügbar unter `http://www.curry-lang.org`. 2016.

[HHPJ+96] Cordelia V. Hall, Kevin Hammond, Simon Peyton Jones und Philip Wadler. „Type Classes in Haskell“. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.2 (1996), S. 109–138.

[Hin69] Roger Hindley. „The Principal Type-Scheme of an Object in Combinatory Logic“. In: *Transactions of the American Mathematical Society* 146 (1969), S. 29–60.

[JJM97] Simon Peyton Jones, Mark P. Jones und Erik Meijer. „Type Classes: An Exploration of the Design Space“. In: *Haskell Workshop*. Jan. 1997, S. 1–16.

[Jon00] Mark P. Jones. „Type Classes with Functional Dependencies“. In: *European Symposium on Programming*. Springer. 2000, S. 230–244.

[Jon92] Mark P. Jones. „A Theory of Qualified Types“. In: *European Symposium on Programming*. Springer. 1992, S. 287–306.

[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*. 1993, S. 52–61.

[Jon94] Mark P. Jones. *The Implementation of the Gofer Functional Programming System*. Forschungsbericht YALEU/DCS/RR-1030. New Haven, Connecticut, USA: Yale University, 1994.

[Jon95] Mark P. Jones. „Simplifying and Improving Qualified Types“. In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. 1995, S. 160–169.

[Lux08] Wolfgang Lux. „Adding Haskell-style overloading to Curry“. In: *25. Workshop der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“*. 2008, S. 67–76.

[Mar+10] Simon Marlow (Hrsg.) u. a. *Haskell 2010 Language Report*. Verfügbar unter `https://www.haskell.org/onlinereport/haskell2010`. 2010.

[Mil78] Robin Milner. „A Theory of Type Polymorphism in Programming“. In: *Journal of Computer and System Sciences* 17.3 (1978), S. 348–375.

[Pro20] Kai-Oliver Prott. „Extending the Glasgow Haskell Compiler for functional-logic Programs with *Curry-Plugin*“. Masterarbeit. Christian-Albrechts-Universität zu Kiel, Okt. 2020.

[SDPJ+07] Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones und Peter J. Stuckey. „Understanding Functional Dependencies via Constraint Handling Rules“. In: *Journal of Functional Programming* 17.1 (2007), S. 83–129.

[SPJC+08] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty und Martin Sulzmann. „Type Checking with Open Type Functions“. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. 2008, S. 51–62.

[Str67] Christopher Strachey. *Fundamental Concepts in Programming Languages*. Lecture notes for International Summer School in Computer Programming. Copenhagen, Aug. 1967.

[Tee16] Finn Teegen. „Erweiterung von Curry um Typklassen und Typkonstruktorklassen“. Masterarbeit. Christian-Albrechts-Universität zu Kiel, Sep. 2016.

[WB89] Philip Wadler und 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*. 1989, S. 60–76.