Masterarbeit

Erweiterung von Curry um Typklassen und Typkonstruktorklassen

B.Sc. Finn Teegen

September 2016

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

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

Erklärung der Urheberschaft

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum	Unterschrift

Zusammenfassung

Die deklarative Programmiersprache Curry vereint mehrere Programmierparadigmen in sich, darunter die funktionale, logische und nebenläufige Programmierung. Ihre Syntax und Semantik ist in weiten Teilen derjenigen der funktionalen Programmiersprache Haskell ähnlich, Curry fehlen allerdings einige weitergehende Sprachmerkmale. Darunter sind auch Typ- und Typkonstruktorklassen, welche in Haskell das systematische Überladen von Funktionen ermöglichen.

In dieser Arbeit wird ein bestehendes Curry-System um die Unterstützung eben jener Typ- und Typkonstruktorklassen erweitert. Dabei werden sowohl im Allgemeinen die notwendigen Erweiterungen des Typsystems als auch im Speziellen die am Compiler vorgenommenen Änderungen beschrieben. Die Implementierung nutzt dabei den bewährten Wörterbuchansatz, wobei auf eine Besonderheit bei dessen Verwendung in funktionallogischen Sprachen wie Curry eingegangen wird.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Eine Funktion, die unterschiedliche Typen für ihre Argumente akzeptiert, wird als *polymorph* bezeichnet. Im Allgemeinen werden mit *parametrischem* und *Ad-Hoc-Polymorphismus* zwei Arten von Polymorphismus unterschieden [Str67].

Parametrischer Polymorphismus tritt auf, wenn eine Funktion unabhängig vom Typ ihrer Argumente definiert ist. Üblicherweise operieren solche Funktionen auf einer Behälterstruktur, also beispielsweise einer Liste. Weil prinzipiell jeder Typ als Elementtyp des Behälters infrage kommt, bleibt er der Funktion unbekannt. Ohne diese Information kann daher nur indirekt auf die Elemente des Behälters zugegriffen werden, zum Beispiel durch Funktionen höherer Ordnung. Das Hindley-Milner-Typsystem [Hin69; Mil78], das die Grundlage vieler funktionaler Programmiersprachen darstellt, unterstützt diese Art des Polymorphismus. Beispiele für solche Sprachen sind die reinfunktionale Sprache *Haskell*[1] und die Multiparadigmensprache *Curry*[2].

Beim Ad-Hoc-Polymorphismus dagegen, der auch als *Überladen* bezeichnet wird, ist einer Funktion der genaue Argumenttyp bekannt und sie hat direkten Zugriff auf die Werte ihrer Argumente. Insbesondere kann das Verhalten einer Funktion in Abhängigkeit von dieser Kenntnis grundlegend variieren. Typische Beispiele hierfür sind die Gleichheit sowie arithmetische Operationen. So ist der Algorithmus zur Multiplikation zweier Ganzzahlen zum Beispiel ein anderer als der zur Multiplikation zweier Gleitkommazahlen, obwohl für beide Operationen oft das gleiche Symbol genutzt wird. Für jeden Typ muss eine eigene Implementierung angegeben werden, daher ist das Verhalten einer überladenen Funktion in der Regel nur auf einer Teilmenge aller Typen definiert.

Lange gab es kein einheitliches Konzept, um Funktionen in funktionalen Programmiersprachen überladen zu können. Sprachen wie *Miranda*[3] oder *Standard ML*[4] verfolgten verschiedene und teils sogar mehrere Ansätze zugleich [Hal+96]. Mit *Typklassen* wurde schließlich ein neuartiger Ansatz zur Erweiterung des Hindley-Milner-Typsystems um Ad-Hoc-Polymorphismus vorgestellt [WB89], der direkt in die Entwicklung von Haskell einfloss und so Teil der Sprache wurde [HPW92]. Bald darauf wurde eine Verallgemeinerung von Typklassen zu *Typkonstruktorklassen* vorgeschlagen, mit deren Hilfe etwa beliebige Monaden implementiert werden können [Jon93]. Auch diese Entwicklung wurde in Haskell übernommen [Jon03] und bis heute zählen Typ- und Typkonstruktorklassen zu den herausragendsten Merkmalen von Haskell. Aus diesem Grund ist es wünschenswert, diesen Ansatz für Ad-Hoc-Polymorphismus auch in Curry bereitzustellen.

[1] `https://www.haskell.org/`
[2] `http://www.curry-language.org/`
[3] `http://miranda.org.uk/`
[4] `http://www.standardml.org/`

1.1. Zielsetzung

Das Ziel dieser Arbeit ist die Erweiterung eines bestehenden Curry-Systems um die Unterstützung für Typ- und Typkonstruktorklassen. Bei dem System, das wir anpassen werden, handelt es sich um das *Portland Aachen Kiel Curry System (PAKCS)*[5], welches Curry- in Prolog-Programme übersetzt. Von dieser Arbeit wird allerdings auch das *Kiel Curry System Version 2 (KiCS2)*[6] profitieren, das nach Haskell kompiliert, da beide Systeme auf einem gemeinsamen Front-End aufbauen.

Aufgrund der Tatsache, dass die Anpassung eines gesamten Systems auch die sämtlicher Bibliotheken und Werkzeuge einschließen würde, beschränken wir uns in dieser Arbeit ausschließlich auf den Compiler, genauer gesagt auf das Front-End. Einzig und allein die Standardbibliothek wird zwangsläufig mit berücksichtigt.

1.2. Verwandte Arbeiten

Es hat bereits vorangegangene Arbeiten zum Thema gegeben, Curry um Typklassen und sogar Typkonstruktorklassen zu erweitern, die nachfolgend kurz vorgestellt werden.

Im Rahmen seiner Masterarbeit hat Matthias Böhm 2013 das KiCS2 um die Unterstützung für einfache Typklassen erweitert [Boh13]. Seine Arbeit wurde allerdings wegen einiger konzeptioneller Schwierigkeiten – unter anderem nahm seine Implementierung keine klare Trennung von Prüfungs- und Transformationsphasen vor – nie für die Hauptversion adaptiert. Aus denselben Gründen wäre es auch schwierig gewesen, seine Implementierung um die Unterstützung für Typkonstruktorklassen zu erweitern. Nichtsdestotrotz finden sich viele Ansätze seiner Arbeit in dieser wieder.

Mit dem *Münster Curry Compiler (MCC)*[7] von Wolfgang Lux gibt es eine weitere Curry-Distribution, welche Typklassen inklusive Typkonstruktorklassen unterstützt [Lux08]. Zwar ist diese Unterstützung zur Zeit nur für einen experimentellen Entwicklungszweig gegeben, dennoch war auch diese Arbeit eine starke Inspiration – nicht zuletzt aufgrund der Tatsache, dass das gemeinsame Front-End des KiCS2 und des PAKCS ursprünglich auf dem MCC basiert.

1.3. Gliederung

Diese Arbeit besteht im Wesentlichen aus drei Teilen. Der erste Teil stellt einige Grundlagen bereit. Er umfasst eine kurze Einführung in die Programmiersprache Curry (Kapitel 2) sowie einen informellen Überblick über Typ- und Typkonstruktorklassen zusammen mit einer formalen Beschreibung der Typklassenelemente (Kapitel 3). Es wird außerdem die Umsetzung von Typklassen mittels Wörterbüchern beschrieben (Kapitel 4).

[5] `https://www.informatik.uni-kiel.de/~pakcs/`
[6] `https://www-ps.informatik.uni-kiel.de/kics2/`
[7] `http://danae.uni-muenster.de/curry/`

Im zweiten Teil wird das Typsystem von Curry behandelt. Zunächst werden die theoretischen Grundlagen des Hindley-Milner-Typsystems, auf dem Curry basiert, wiederholt und die wichtigsten Erkenntnisse für dieses zusammengefasst (Kapitel 5), bevor im direkten Anschluss auf die Erweiterung des Typsystems zur Unterstützung von Typ- und Typkonstruktorklassen eingegangen wird (Kapitel 6).

Der dritte Teil bildet den Hauptteil dieser Ausarbeitung und befasst sich mit der eigentlichen Implementierung. Nachdem ein Überblick über den ursprünglichen Zustand des Front-Ends des Curry-Systems gegeben wurde (Kapitel 7), werden alle an ihm vorgenommenen Änderungen im Detail beschrieben (Kapitel 8).

Am Ende werden die Ergebnisse dieser Arbeit noch einmal zusammengefasst sowie ein Ausblick auf mögliche weiterführende Arbeiten gegeben (Kapitel 9).

Teil I.

Grundlagen

2. Curry

Curry ist eine *funktionallogische Programmiersprache* und vereint als solche sowohl Elemente der funktionalen als auch der logischen Programmierung in sich. Dabei ist die funktionale Komponente von Curry stark an Haskell angelehnt, was sich vor allem in der Ähnlichkeit der Syntax und Semantik äußert. Im Folgenden gehen wir davon aus, dass der Leser mit Haskell und den darin umgesetzten Konzepten vertraut ist, und konzentrieren uns daher auf die Vorstellung der Konzepte, die der logischen Programmierung entstammen: *Nichtdeterminismus*, *freie Variablen* und *Constraints*.

2.1. Nichtdeterminismus

Nichtdeterministische Funktionen werden in Curry durch überlappende Regeln ausgedrückt. Ein einfaches Beispiel ist die folgende nullstellige Funktion `coin`, welche einen Münzwurf simuliert.

```
coin :: Int
coin = 0
coin = 1
```

Im Gegensatz zu Haskell, wo die zweite Regel von `coin` ignoriert werden würde, werden in Curry beide Regeln angewandt. Der Ausdruck `coin` ist also nichtdeterministisch und hat mit `0` und `1` mehrere mögliche Ergebnisse.

Beispiel 2.1.:
Wir definieren mit `insert` *zunächst eine Funktion, die ein Element nichtdeterministisch in eine Liste einfügt.*

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

Unter Verwendung dieser lässt sich dann die Operation `perm` *definieren, die (ebenfalls nichtdeterministisch) jede Permutation einer Liste berechnet.*

```
perm :: [a] -> [a]
perm []     = []
perm (x:xs) = insert x (perm xs)
```

Neben der Möglichkeit, Nichtdeterminismus durch überlappende Regeln auszudrücken, kann dies auch mithilfe des vordefinierten `?`-Operators geschehen.

```
coin = 0 ? 1
```

Die Tatsache, dass überlappende Regeln in Curry automatisch in Nichtdeterminismus resultieren, erfordert eine gewisse Vorsicht vonseiten des Programmierers, wie das folgende Beispiel verdeutlicht.

```
not :: Bool -> Bool
not True = False
not _    = True
```

Während der Ausdruck `not False` noch wie erwartet ausschließlich zu `True` ausgewertet wird, hat der Ausdruck `not True` zwei mögliche Ergebnisse: `False` und `True`. Dieses ungewollte Verhalten kommt durch die zweite, immer zutreffende Regel zustande und lässt sich beheben, indem diese durch die folgende ersetzt wird.

```
not False = True
```

Einschub 2.2. (Call-Time-Choice vs. Run-Time-Choice)**:**
Im Kontext von Nichtdeterminismus ist es wichtig, bei dessen Auflösung durch die Auswahl einer der möglichen Alternativen zwischen der Call-Time-Choice- *und der* Run-Time-Choice-Semantik *[HA77; Hus92] zu unterscheiden. Bei ersterer wird der Wert eines nichtdeterministischen Arguments schon beim Aufruf einer Funktion festgelegt, bei letzterer dagegen erst bei deren Auswertung – und zwar an jeder Stelle, an der das Argument auftritt. Insbesondere folgt daraus, dass im ersten Fall eventuell vorhandener Nichtdeterminismus* geteilt *wird, was auch als* Sharing *bezeichnet wird.*

Wir betrachten beispielhaft den Ausdruck `pair coin`*, wobei die Funktion* `pair` *folgendermaßen definiert ist.*

```
pair :: a -> (a, a)
pair x = (x, x)
```

Abhängig von der implementierten Semantik sind nun verschiedene Ergebnisse für diesen Ausdruck möglich: Im Falle der Call-Time-Choice-Semantik (s. Abbildung 2.1) lauten die Ergebnisse `(0, 0)` *und* `(1, 1)`*, im Falle der Run-Time-Choice-Semantik (s. Abbildung 2.2) kommen ergänzend die beiden Ergebnisse* `(0, 1)` *und* `(1, 0)` *hinzu.*

Curry implementiert die Call-Time-Choice-Semantik. Diese entspricht zumeist der Intuition des Programmierers und wird üblicherweise für funktionallogische Sprachen gewählt, so beispielsweise auch für TOY [FH99].

2.2. Freie Variablen und Constraints

Curry erlaubt mithilfe des Schlüsselworts `free` die lokale Deklaration freier Variablen. Diese können in Verbindung mit Constraints dazu genutzt werden, unbekannte Werte zu berechnen. Zum Beispiel hat die folgende, nullstellige Funktion als Ergebnis den für die freie Variable `x` ermittelten Wert `1`.

```
f | x =:= 1 = x
  where x free
```

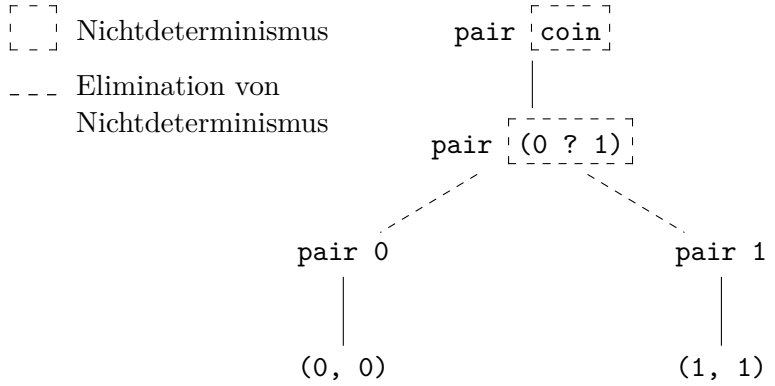

Abbildung 2.1.: Schematische Auswertung mit Call-Time-Choice-Semantik

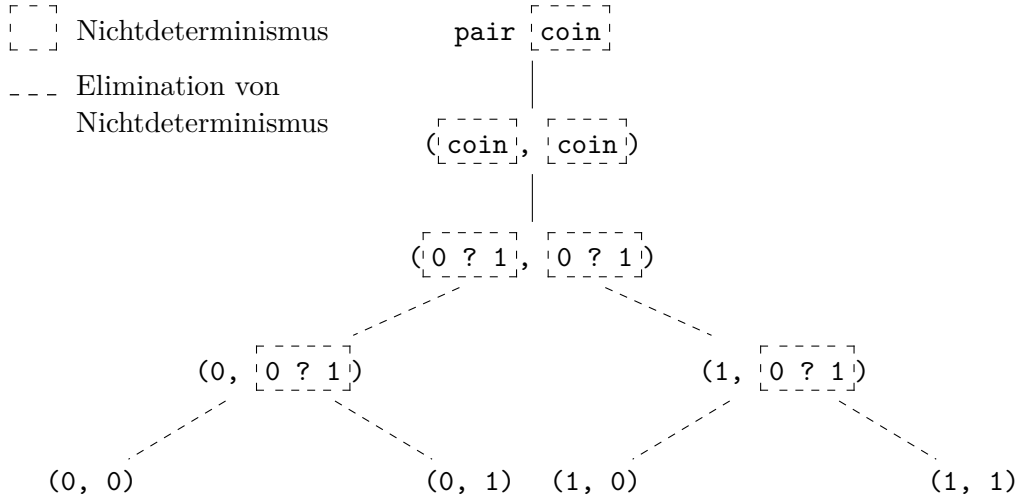

Abbildung 2.2.: Schematische Auswertung mit Run-Time-Choice-Semantik

Das Constraint für die Gleichheit `(=:=)` hat dabei den Typ `a -> a -> Bool` und wertet nur dann zu `True` aus, wenn beide Seiten zu identischen *Grundtermen*, d.h. Terme ohne Variablen oder Funktionen, reduzierbar sind. Ist das Constraint nicht erfüllbar, schlägt die Berechnung fehl und es kommt gar kein Ergebnis heraus.

Mithilfe von freien Variablen ist es beispielsweise möglich, die Parameter einer Funktion zu berechnen, sodass ein bestimmtes Constraint erfüllt ist.

Beispiel 2.3.:
Die Funktion `last` *berechnet das letzte Element einer Liste und gibt dieses zurück.*

```
last :: [a] -> a
last xs | ys ++ [e] =:= xs = e
  where ys, e free
```

3. Typ- und Typkonstruktorklassen

Typklassen wurden von Philip Wadler und Stephen Blott entwickelt [WB89], um einen systematischen Ansatz zur Unterstützung von Ad-Hoc-Polymorphismus in funktionalen Programmiersprachen wie Haskell bereitzustellen. Mark P. Jones verallgemeinerte Typklassen schließlich zu *Typkonstruktorklassen* [Jon93].

3.1. Informelle Einführung

Zunächst werden Typklassen anhand mehrerer Beispiele eingeführt und dabei ihre verschiedenen Nutzungsaspekte diskutiert. Die behandelten Beispiele beziehen sich zwar allesamt auf Haskell, werden aber in jedem Fall auch auf Curry übertragbar sein. Es muss unter Umständen lediglich auf überlappende Regeln Acht gegeben werden (vgl. Abschnitt 2.1).

3.1.1. Überladung

Funktionen können überladen werden, indem eine *Typklasse* für sie erstellt wird. Diese enthält die durch sie überladenen Funktionen, die sogenannten *Klassenmethoden*.

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

Obige Deklaration führt beispielsweise eine Typklasse namens `Eq` ein, durch die der Gleichheitsoperator `(==)` des Typs `a -> a -> Bool` überladen wird. Die Typvariable `a` fungiert dabei als Platzhalter für den konkreten Typ, für den die Gleichheit überladen werden soll.

Die Überladung für bestimmte Typen erfolgt nun, indem man für ihn eine *Instanz* definiert, in der Implementierungen für die Klassenmethoden angegeben werden.

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

Durch die obige Instanzdeklaration wird die Gleichheit für den Typen `Bool` überladen, sodass man Ausdrücke wie `True == False` schreiben kann. Die Implementierung der Funktion `(==)` hat dabei den Typ `Bool -> Bool -> Bool`. Die Typvariable `a` in der Klassendeklaration wurde also in den Typen der Implementierungen durch den konkreten Typ der Instanz ersetzt.

Auf Basis von Klassenmethoden können auch andere Funktionen überladen werden. Die folgende Funktion `elem` nutzt beispielsweise die zuvor überladene Funktion `(==)`.

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

Der Typ von `elem` lautet `Eq a => a -> [a] -> Bool` und drückt aus, dass die Gleichheit für den Elementtyp überladen sein muss. Der Teil `Eq a` wird Kontext genannt und beschränkt die erlaubten Typen für die Typvariable `a` auf jene, die eine Instanz der Klasse `Eq` sind. Daher spricht man auch von beschränkten Typen.

Es ist auch möglich, dass Instanzen selbst einen Kontext besitzen. Die nachfolgende Implementierung der Gleichheit für Tupel setzt zum Beispiel voraus, dass die Gleichheit auch für die beiden Teiltypen überladen ist.

```
instance (Eq a, Eq b) => Eq (a, b) where
  (a, b) == (c, d) = a == c && b == d
```

Funktionen können nicht nur für Typen, sondern auch für Typkonstruktoren überladen werden. Dies geschieht über Typkonstruktorklassen, die sich im Grunde nur dadurch von Typklassen unterscheiden, dass die Typvariable in der Klassendefinition für einen partiell und nicht vollständig angewandten Typkonstruktor steht. Ein Beispiel dafür ist die Typkonstruktorklasse `Functor`, die wie folgt definiert ist.

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

In dem Typ der Klassenmethode `fmap` wird die Typvariable `f` auf andere Typen appliziert, in diesem Fall auf weitere Typvariablen. Für Typkonstruktorklassen ist also eine Typvariablenapplikation notwendig.

Instanzen werden analog zu normalen Typklassen angegeben, nur dass der konkrete Instanztyp nun ein partiell applizierter Typkonstruktor ist.

```
instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

3.1.2. Standardimplementierungen

Betrachtet man die vorgestellte Typklasse `Eq`, erscheint es unter Umständen naheliegend, zusätzlich zu dem Operator für die Gleichheit einen für die Ungleichheit bereitzustellen. Damit aber nicht alle Instanzen für die `Eq`-Klasse danach auch die Ungleichheitsfunktion implementieren müssen, kann man eine Standardimplementierung angeben, die ausnutzt, dass die Ungleichheit unter Verwendung der Negation auf die Gleichheit zurückgeführt werden kann.

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

  x /= y = not (x == y)
```

Wann immer nun eine Instanz die Ungleichheitsfunktion nicht implementiert, wird die Standardimplementierung verwendet. So behält beispielsweise auch die `Eq-Bool`-Instanz aus Unterabschnitt 3.1.1 weiterhin ihre Gültigkeit.

3.1.3. Superklassen

Bei der Definition einer Typklasse, können optional Superklassen angegeben werden. Als ein Beispiel betrachten wir die folgende, vereinfachte Version der `Ord`-Typklasse.

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

  x < y = x <= y && not (x == y)
```

Der Kontext `Eq a` ist hier notwendig, weil die Standardimplementierung von `(<)` die überladene Funktion `(==)` verwendet.

3.2. Formale Beschreibung

In diesem Abschnitt geben wir jeweils eine kurze formale Beschreibung für alle zuvor vorgestellten Typklassenelemente an.

3.2.1. Klassendeklarationen

Definition 3.1. (Klassendeklaration)**:**
Eine Klassendeklaration hat die allgemeine Form

`class (`S_1 u`, ..., `S_k u`) => `C u `where`
  $method_1$ `::` K_1 `=>` t_1
  ...
  $method_n$ `::` K_n `=>` t_n

  Implementierung von $method_l$
  ...
  Implementierung von $method_m$

mit $n \geq 0$, $k \geq 0$ *und* $\{l, \ldots, m\} \subseteq [1, n]$. *Dabei sind* $S_1, \ldots, S_k$ *sowie* C *Klassennamen,* u *eine Typvariable,* $method_1, \ldots, method_n$ *Funktionsnamen,* $K_1, \ldots, K_n$ *Kontexte und* $t_1, \ldots, t_n$ *Typen.*

Eine solche Deklaration führt eine neue Typklasse C mit den assoziierten Klassenmethoden $method_1, \ldots, method_n$ ein. Für $i \in [1, n]$ hat die Methode $method_i$ den Typ

$(\{C\ u\} \cup K_i)$ `=>` t_i, wobei $\{C\ u\}$ in diesem Zusammenhang *impliziter Klassenkontext* genannt wird. Die Typvariable u wird auch als Klassenvariable bezeichnet und muss in allen Typen $t_1, \ldots, t_n$ vorkommen. Die Kontexte $K_1, \ldots, K_n$ dürfen die Klassenvariable nicht zusätzlich beschränken. Den Kontext $\{S_1\ u, \ldots, S_k\ u\}$ nennt man auch Klassenkontext. Bei $S_1, \ldots, S_k$ handelt es sich um die Superklassen von C. Die Superklassenrelation darf nicht zyklisch definiert sein.

3.2.2. Instanzdeklarationen

Definition 3.2. (Instanzdeklaration)**:**
Für eine Klassendeklaration der Form

`class (`$S_1\ u$`, ..., `$S_k\ u$`) => `$C\ u$` where`
  $method_1$ `::` K_1 `=>` t_1
  ...
  $method_n$ `::` K_n `=>` t_n

  Implementierung von $method_l$
  ...
  Implementierung von $method_m$

mit $n \geq 0$, $k \geq 0$ *und* $\{l, \ldots, m\} \subseteq [1, n]$ *hat eine Instanzdeklaration die allgemeine Form*

`instance (`$C_1\ v_x$`, ..., `$C_g\ v_s$`) => `C` (`$T\ v_1 \ldots v_h$`) where`
  Implementierung von $method_o$
  ...
  Implementierung von $method_t$

mit $g \geq 0$, $h \geq 0$, $\{v_x, \ldots, v_s\} \subseteq \{v_1, \ldots, v_h\}$ *und* $\{o, \ldots, t\} \in [1, n]$. *Dabei sind* $C_1, \ldots, C_g$ *sowie* C *Klassennamen,* T *ein Typkonstruktor,* $v_1, \ldots, v_h$ *Typvariablen und* $method_1, \ldots, method_n$ *Funktionsnamen.*

Eine solche Deklaration definiert eine C-T-Instanz. Die Typvariablen $v_1, \ldots, v_h$ müssen paarweise verschieden sein und bei dem Typkonstruktor T darf es sich um kein Typsynonym handeln. Den Kontext $\{C_1\ v_x, \ldots, C_g\ v_s\}$ nennt man auch Instanzkontext. Für alle S_i mit $i \in [1, k]$ muss eine S_i-T-Instanz existieren.

3.2.3. Constraints

Definition 3.3. (Constraint)**:**
Ein Constraint (nicht zu verwechseln mit dem gleichnamigen Begriff aus Abschnitt 2.2) hat die folgende allgemeine Form.

$C\ t$

Dabei ist C *ein Klassenname und* t *ein Typ.*

Ein solches Constraint repräsentiert die Annahme, dass der Typ t eine Instanz der Typklasse C ist. Handelt es sich bei dem Typ t um eine Typvariable, bezeichnet man das Constraint auch als *einfach*.

3.2.4. Kontexte

Definition 3.4. (Kontext)**:**
Ein Kontext hat die allgemeine Form

$$(C_1\,,\ \ldots,\ C_n)$$

mit $n \geq 0$. *Dabei sind* $C_1, \ldots, C_n$ *Constraints.*

Sind alle in einem Kontext enthaltenen Constraints einfach, so spricht man auch von einem *einfachen Kontext*. Genau solche Kontexte sind es auch, die in Klassen- und Instanzdeklarationen angegeben werden können.

3.2.5. Kontextreduktion

Kontexte können auf zweierlei Weisen umgeformt werden: einerseits durch die Vereinfachung einzelner Constraints, andererseits durch das Entfernen selbiger. Beide Maßnahmen zusammengenommen bezeichnet man als Kontextreduktion [Jon99]. Für die folgenden Betrachtungen wird ein Kontext als eine Multimenge und nicht als eine Liste von Constraints aufgefasst.

Zunächst wird die Vereinfachung eines Constraints betrachtet. Hierfür wird auf die im Programm gegebenen Instanzdefinitionen zurückgegriffen, was sich folgendermaßen formalisieren lässt.

Transformation 3.5. (Vereinfachung eines Constraints)**:**
Ein Constraint $C\ (T\ t_1 \ldots t_n)$ *mit* $n \geq 0$ *kann zu dem Kontext* $\{C_1\ t_{k_1}, \ldots, C_m\ t_{k_m}\}$ *mit* $m \geq 0$ *und* $\{t_{k_1}, \ldots, t_{k_m}\} \subseteq \{t_1, \ldots, t_n\}$ vereinfacht *werden, wenn eine Instanz* `instance` $(C_1\ u_{k_1}\texttt{ , }\ldots\texttt{, }C_m\ u_{k_m})$ `=>` $(C\ T\ u_1\ \ldots\ u_n)$ *existiert.*

Beispiel 3.6.:
Das Constraint `Eq [a]` *kann zu* {`Eq a`} *vereinfacht werden, wenn die Instanz* `instance Eq a => Eq [a]` *existiert.*

Die Vereinfachung eines Kontexts besteht dann in der wiederholten Vereinfachung der Constraints.

Transformation 3.7. (Vereinfachung eines Kontexts)**:**
Ein Kontext $K \dot\cup \{C\ t\}$ *kann zu dem Kontext* $K \cup \{C_1\ t_1, \ldots, C_n\ t_n\}$ *mit* $n \geq 0$ vereinfacht *werden, wenn das Constraint* $C\ t$ *zu dem Kontext* $\{C_1\ t_1, \ldots, C_n\ t_n\}$ *vereinfacht werden kann. Für die* vollständige Vereinfachung *eines Kontexts wird wiederholt jedes Element des Kontexts vereinfacht, bis der Kontext nicht weiter vereinfacht werden kann.*

Zu beachten ist hierbei, dass der Kontext durch die Vereinfachung eines Constraints sowohl größer als auch kleiner werden kann.

Beispiel 3.8.:
Der Kontext {`Eq (a, b)`} *kann mithilfe der Instanz* `instance (Eq a, Eq b) => Eq (a, b)` *zu* {`Eq a`, `Eq b`} *vereinfacht werden und vergrößert sich somit. Der Kontext* {`Eq Int`} *kann dagegen zu dem leeren Kontext vereinfacht werden, wenn eine* `Eq-Int`*-Instanz existiert.*

Wir gehen im Folgenden davon aus, dass alle spezifizierten Kontexte *gültig* sind, wobei der Begriff der Gültigkeit eines Kontexts wie folgt definiert ist.

Definition 3.9. (Gültiger Kontext)**:**
Ein Kontext heißt gültig*, wenn er so vereinfacht werden kann, dass jedes Constraint einfach ist.*

Nach der Vereinfachung aller Constraints eines Kontexts können ggf. einige nicht notwendige Elemente entfernt werden. Die Nichtnotwendigkeit eines Constraints ist gegeben, wenn dieses vom restlichen Kontext *impliziert* wird. Dies wird durch eine Folgerungsrelation $\Vdash$ beschrieben [JJM97], für die zunächst die Regeln in Abbildung 3.1 gelten.

Die Regeln (*trans*) und (*mono*) sagen aus, dass die Kontextimplikation *transitiv* und *monoton* ist. Die Regel (*union*) besagt, dass ein Kontext, wenn er zwei andere Kontexte impliziert, auch die Vereinigung beider impliziert. Die Regel (*extend*) drückt aus, dass ein Kontext, wenn er einen anderen impliziert, beliebig erweitert werden kann, ohne die Implikationseigenschaft zu verletzen. Weiterhin folgt aus der Regel (*mono*) direkt, dass die Folgerungsrelation $\Vdash$ *reflexiv* ist; es gilt also $P \Vdash P$ für einen beliebigen Kontext P.

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

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

$$\frac{P \Vdash Q \qquad P \Vdash Q'}{P \Vdash Q \cup Q'} \quad (union)$$

$$\frac{P \Vdash Q}{P \cup P' \Vdash Q} \quad (extend)$$

Abbildung 3.1.: Regeln der Folgerungsrelation $\Vdash$ für Kontexte (Teil 1)

Weitere Regeln für die Kontextimplikation ergeben sich aus den gegebenen Instanz- und Klassendeklarationen (s. Abbildung 3.2). Dabei bezeichne der Ausdruck $\mathrm{TV}(P)$ die Menge aller im Kontext P auftretenden Typvariablen, θ eine Substitution, die Typvariablen auf Typausdrücke abbildet, und $\mathrm{dom}(\theta)$ ihren Wertebereich. Die Prämisse in den Regeln bedeutet somit, dass die Substition θ für alle Typvariablen, die in dem Kontext P vorkommen, definiert ist.

$$\frac{\mathrm{TV}(C) \subseteq \mathrm{dom}(\theta)}{\theta(P) \Vdash \theta(C)} \qquad \text{Instanz } \texttt{instance } P \texttt{ => } C \text{ existiert} \qquad (\textit{inst})$$

$$\frac{\mathrm{TV}(C) \subseteq \mathrm{dom}(\theta)}{\theta(C) \Vdash \theta(P)} \qquad \text{Klassendefinition } \texttt{class } P \texttt{ => } C \text{ existiert} \qquad (\textit{super})$$

Abbildung 3.2.: Regeln der Folgerungsrelation $\Vdash$ für Kontexte (Teil 2)

Beispiel 3.10.:
Der Kontext {`Ord Int`} *impliziert nach der Regel (inst) den Kontext* {`Ord [Int]`}*, wenn die Instanz* `instance Ord a => Ord [a]` *existiert. Dieser Kontext impliziert wiederum aufgrund der Regel (super) den Kontext* {`Eq [c]`}*, wenn es die Klassendefinition* `class Eq a => Ord a` *gibt.*

Mithilfe dieser Regeln lässt sich nun die *Reduzierbarkeit* eines Kontexts unter Verwendung der *Kontextimplikation* formalisieren.

Definition 3.11. (Kontextimplikation)**:**
Ein Kontext P impliziert *einen Kontext Q, wenn sich $P \Vdash Q$ mithilfe der in den Abbildungen 3.1 und 3.2 dargestellten Regeln ableiten lässt.*

Definition 3.12. (Reduzierbarkeit eines Kontexts)**:**
Ein Kontext P kann zu einem Kontext Q reduziert *werden, wenn Q von P impliziert wird.*

Üblicherweise wird bei der Reduktion eines Kontexts Q nach dem kleinstmöglichen Kontext P gesucht, für den $P \Vdash Q$ gilt.

Definition 3.13. (Vollständig reduzierter Kontext)**:**
Sei Q ein reduzierter Kontext von einem Kontext P. Q ist genau dann vollständig reduziert*, wenn kein anderer Kontext $Q' \subseteq Q$ mit $P \Vdash Q'$ existiert.*

Es ist möglich, einen Algorithmus anzugeben, der einen Kontext vollständig reduziert (s. Abbildung 3.3).

Eingabe: Kontext Q
Ausgabe: Reduzierter Kontext P
Setze $P :=$ Vereinfachung von Q
Solange $\exists C \in P$ mit $P \setminus \{C\} \Vdash \{C\}$
  Setze $P := P \setminus \{C\}$

Abbildung 3.3.: Algorithmus zur Kontextreduktion

Satz 3.14. (Algorithmus zur Kontextreduktion, [Boh13])**:**
Nach Anwendung des Algorithmus in Abbildung 3.3 für einen Eingabekontext Q gilt $P \Vdash Q$ für den Ausgabekontext P und P ist ein vollständig reduzierter Kontext von Q.

Für alle weiteren Kapitel wird die Existenz einer kanonischen Darstellung für reduzierte Kontexte angenommen. Insbesondere ist der vollständig reduzierte Kontext dadurch eindeutig bestimmt. Außerdem muss die Reihenfolge der Kontextelemente nicht gesondert berücksichtigt werden, da diese durch die kanonische Darstellung gegeben ist. Wie wir diese Darstellung in unserer Implementierung erreichen, erläutern wir in Abschnitt 8.4.

3.2.6. Beschränkte Typen

Definition 3.15. (Beschränkter Typ)**:**
Ein beschränkter Typ hat die folgende allgemeine Form.

K `=>` t

Dabei ist K ein Kontext und t ein Typ.

In dem Kontext K dürfen nur Typvariablen auftreten, die auch im Typ t vorkommen. Ansonsten spricht man von einem mehrdeutigen Kontext bzw. mehrdeutigen Typvariablen. Wenn der Kontext K nur ein Constraint enthält, können die Klammern um den Kontext entfallen. Ist der Kontext K sogar leer, darf er zusammen mit dem Kontexttrennelement `=>` ausgespart werden, sodass nur der Typ t übrig bleibt.

3.2.7. Default-Deklarationen

Definition 3.16. (Default-Deklaration)**:**
Eine Default-Deklaration hat die allgemeine Form

`default (`t_1`, ..., `t_n`)`

mit $n \geq 0$. Dabei sind $t_1, \ldots, t_n$ Typen.

Jeder Typ t_i für $i \in [1, n]$ muss eine Instanz der `Num`-Klasse bilden. Pro Modul darf höchstens eine Default-Deklaration existieren und ihr Effekt beschränkt sich auf das jeweilige Modul. Wenn keine Default-Deklaration angegeben wurde, wird

```
default (Int, Float)
```

angenommen. Ein Default-Deklaration der Form

```
default ()
```

unterbindet das Defaulting im Modul.

4. Umsetzung mittels Wörterbüchern

In diesem Kapitel beschreiben wir die Implementierung von Typklassen mittels *Wörterbüchern*. Dieses Verfahren wurde bereits in der Originalarbeit zu Typklassen [WB89] vorgeschlagen und eignet sich gleichermaßen für die Implementierung von Typkonstruktorklassen [Jon93]. So folgt beispielsweise auch der *Glasgow Haskell Compiler (GHC)*[1] diesem Ansatz [GHC15]. Die Idee dabei ist es, ein Programm mit Typklassenelementen in ein äquivalentes Programm ohne eben diese zu transformieren; es handelt sich also um eine *Quellcode-zu-Quellcode-Transformation*.

Es sei an dieser Stelle darauf hingewiesen, dass es auch andere Ansätze für die Implementierung von Typklassen gibt. Einer davon, der ausdrücklich für funktionallogische Sprachen wie Curry vorgeschlagen wird, stammt von Enrique Martin-Martin und verwendet *typindizierte Funktionen*, die mithilfe von *Typzeugen* die passende Implementierung einer Klassenmethode auswählen [Mar11]. Neben der Notwendigkeit eines neuartigen Typsystems besteht das Hauptproblem dieses Ansatzes aber vor allem darin, dass nicht ersichtlich ist, ob er sich mit Typkonstruktorklassen verträgt.

4.1. Behandlung von Klassendeklarationen

Wir beginnen mit der Erläuterung, wie Klassendeklarationen behandelt werden. Für jede dieser Deklarationen wird ein Wörterbuchtyp angelegt. Dabei nutzen wir einen algebraischen Datentyp mit genau einem Datenkonstruktor, der für jede Klassenmethode ein Feld vom Typ der Klassenmethode erhält. Die Reihenfolge der Felder wird durch die Reihenfolge der Methoden innerhalb der Klassendeklaration bestimmt. Im Falle der Typklasse `Eq` lautet die erzeugte Datentypdeklaration für den Wörterbuchtyp beispielsweise folgendermaßen.

```
data Dict_Eq a = Dict_Eq (a -> a -> Bool)
                         (a -> a -> Bool)
```

Statt algebraischer Datentypen könnten theoretisch auch Typsynonyme für Tupel verwendet werden, wie es zum Beispiel in [Boh13] oder auch in [HB90] gemacht wurde. Wir haben uns allerdings dagegen entschieden, unter anderem weil keine null- oder einstelligen Tupel existieren und man daher für Typklassen mit weniger als zwei Klassenmethoden auf andere Darstellungen ausweichen müsste.

Gleichzeitig wird für jede Klassenmethode ein Selektor eingeführt, der die passende Methode aus einem gegebenen Wörterbuch extrahiert. Für die Typklasse `Eq` lauten diese Selektoren wie folgt.

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

```
(==) :: Dict_Eq a -> a -> a -> Bool
(==) (Dict_Eq m _) = m
(/=) :: Dict_Eq a -> a -> a -> Bool
(/=) (Dict_Eq _ m) = m
```

Wir behalten die Namen der Klassenmethoden für die Selektoren bei, da deren Typen unter Berücksichtigung des impliziten Klassenkontexts ohnehin den transformierten Typen (vgl. Abschnitt 4.3) der Klassenmethoden entsprechen.

Sollte eine Klasse Superklassen besitzen, werden diese berücksichtigt, indem zusätzliche Felder für deren Wörterbücher hinzugefügt werden. Dies kann man am Beispiel der Typklasse `Ord` sehen.

```
data Dict_Ord a = Dict_Ord (Dict_Eq a)
                           (a -> a -> Ordering)
                           (a -> a -> Bool)
                           (a -> a -> Bool)
                           (a -> a -> Bool)
                           (a -> a -> Bool)
                           (a -> a -> a)
                           (a -> a -> a)
```

Folglich müssen auch Selektoren für die Superklassenwörterbücher generiert werden, die der Extraktion eines solchen Wörterbuchs dienen.

```
super_Ord,Eq :: Dict_Ord a -> Dict_Eq a
super_Ord,Eq (Dict_Ord d _ _ _ _ _ _ _) = d
```

Eventuell vorhandene Standardimplementierungen werden auf oberste Ebene gehoben (*Lambda Lifting*, [Joh85]) und dabei umbenannt, da sie unter Umständen in der Transformation für Instanzdeklarationen (s. nachfolgenden Abschnitt) benötigt werden.

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

Zu beachten ist hierbei, dass weder die angegebene Typsignatur noch der Rumpf der Methode transformiert wird. Die Umformung von Signatur und Rumpf erfolgt durch andere Transformationen (s. Abschnitt 4.3 und 4.4).

Sollte für eine Klassenmethode keine Standardimplementierung spezifiziert worden sein, wird stattdessen eine erzeugt, die zur Laufzeit eine Fehlermeldung ausgibt. Für die Methode `abs` der Typklasse `Num` sähe dies folgendermaßen aus.

```
default_Num,abs :: Num a => a -> a
default_Num,abs =
  error "No default implementation for class method 'abs'"
```

Eine vollständige formale Beschreibung der Behandlung von Klassendeklarationen ist mit der folgenden Transformation gegeben.

Transformation 4.1. (Klassendeklaration)**:**
Für eine Klassendeklaration der allgemeinen Form

```
class (S_1 u, ..., S_k u) => C u where
  method_1 :: K_1 => t_1
  ...
  method_n :: K_n => t_n

  Implementierung von method_l
  ...
  Implementierung von method_m
```

mit $n \geq 0$, $k \geq 0$ und $\{l, \ldots, m\} \subseteq [1, n]$ wird die Datentypdeklaration

```
data Dict_C u = Dict_C (Dict_T1 u) ... (Dict_Tp u) t'_1 ... t'_n
```

erzeugt, wobei $\{T_1\ u, \ldots, T_p\ u\}$ mit $p \geq 0$ dem vollständig reduzierten Kontext von $\{S_1\ u, \ldots, S_k\ u\}$ entspricht und $t'_1, \ldots, t'_n$ die transformierten Typausdrücke von K_1 `=>` $t_1, \ldots, K_n$ `=>` t_n sind. Für die Klassenmethoden werden die Selektoren

```
method_1 :: Dict_C u -> t'_1
method_1 (Dict_C d_1 ... d_p m_1 ... m_n) = m_1
...
method_n :: Dict_C u -> t'_n
method_n (Dict_C d_1 ... d_p m_1 ... m_n) = m_n
```

und für die Superklassenwörterbücher die Selektoren

```
super_{C,T_1} :: Dict_C u -> Dict_T1 u
super_{C,T_1} (Dict_C d_1 ... d_p m_1 ... m_n) = d_1
...
super_{C,T_p} :: Dict_C u -> Dict_Tp u
super_{C,T_p} (Dict_C d_1 ... d_p m_1 ... m_n) = d_p
```

generiert. Außerdem werden noch die Deklarationen

```
default_{C,method_l} :: ({C u} ∪ K_l) => t_l
Implementierung von method_l umbenannt in default_{C,method_l}
...
default_{C,method_m} :: ({C u} ∪ K_m) => t_m
Implementierung von method_m umbenannt in default_{C,method_m}
```

für Standardimplementierungen erzeugt. Für alle $i \in [1, n] \setminus \{l, \ldots, m\}$ wird stattdessen eine Deklaration

```
default_{C,method_i} :: ({C u} ∪ K_i) => t_i
default_{C,method_i} = error
  "No default implementation for class method 'method_i'"
```

generiert.

4.2. Behandlung von Instanzdeklarationen

Bei Instanzdeklarationen werden zunächst – ähnlich den Standardimplementierungen in Klassendeklarationen – alle Implementierungen auf oberste Ebene angehoben und umbenannt. Bei der zuvor behandelten `Eq-Bool`-Instanz aus Unterabschnitt 3.1.1 ist davon zum Beispiel die Implementierung des Gleichheitsoperators betroffen.

```
impl_Eq,Bool,== :: Bool -> Bool -> Bool
impl_Eq,Bool,== False False = True
impl_Eq,Bool,== False True  = False
impl_Eq,Bool,== True  False = False
impl_Eq,Bool,== True  True  = True
```

Wenn für eine Klassenmethode keine Implementierung angegeben wurde, wie es für Operator `(/=)` der gleichen Instanz der Fall ist, wird auf die Standardimplementierung zurückgegriffen.

```
impl_Eq,Bool,/= :: Bool -> Bool -> Bool
impl_Eq,Bool,/= = default_Eq,/= inst_Eq,Bool
```

Schließlich wird unter Nutzung des Wörterbuchtyps der zugehörigen Klasse ein konkretes Wörterbuch für die Instanz angelegt, wobei dem Datenkonstruktor die angehobenen Implementierungen als Argumente übergeben werden.

```
inst_Eq,Bool :: Dict_Eq Bool
inst_Eq,Bool = Dict_Eq impl_Eq,Bool,== impl_Eq,Bool,/=
```

Im Falle einer Klasse mit Superklassen muss ergänzend das passende konkrete Superklassenwörterbuch angegeben werden.

```
inst_Ord,Bool :: Dict_Ord Bool
inst_Ord,Bool = Dict_Ord inst_Eq,Bool
                         impl_Ord,Bool,compare
                         impl_Ord,Bool,<
                         impl_Ord,Bool,<=
                         impl_Ord,Bool,>
                         impl_Ord,Bool,>=
                         impl_Ord,Bool,max
                         impl_Ord,Bool,min
```

Wie in Unterabschnitt 3.1.1 bei der Definition einer `Eq-(,)`-Instanz gesehen, können auch Instanzdeklarationen einen Kontext besitzen. Diesem Umstand wird Rechnung getragen, indem der Instanzkontext den Typen der angehobenen Implementierungen und des konkreten Wörterbuchs hinzugefügt wird.

```
impl_Eq,(,),== :: (Eq a, Eq b) => (a, b) -> (a, b) -> Bool
impl_Eq,(,),== (a, b) (c, d) = a == c && b == d
impl_Eq,(,),/= :: (Eq a, Eq b) => (a, b) -> (a, b) -> Bool
impl_Eq,(,),/= = default_Eq,/= inst_Eq,(,)
```

```
inst_Eq,(,)  :: (Eq a, Eq b) => Dict_Eq (a, b)
inst_Eq,(,)  = Dict_Eq impl_Eq,(,),==  impl_Eq,(,),/=
```

Die folgende Transformation umfasst eine formale Beschreibung aller beschriebenen Umformungen für Instanzdeklarationen.

Transformation 4.2. (Instanzdeklaration)**:**
Für eine Klassendeklaration der allgemeinen Form

```
class (S_1 u, ..., S_k u) => C u where
  method_1 :: K_1 => t_1
  ...
  method_n :: K_n => t_n

  Implementierung von method_l
  ...
  Implementierung von method_m
```

mit $n \geq 0$, $k \geq 0$ *und* $\{l, \ldots, m\} \subseteq [1, n]$ *und eine Instanzdeklaration der allgemeinen Form*

```
instance (C_1 v_x, ..., C_g v_s) => C (T v_1 ... v_h) where
  Implementierung von method_o
  ...
  Implementierung von method_t
```

mit $g \geq 0$, $h \geq 0$, $\{v_x, \ldots, v_s\} \subseteq \{v_1, \ldots, v_h\}$ *und* $\{o, \ldots, t\} \in [1, n]$ *wird das konkrete Wörterbuch*

```
inst_C,T  :: (D_1 v_y, ..., D_q v_r) => Dict_C (T v_1 ... v_h)
inst_C,T  = Dict_C inst_T_1,T ... inst_T_p,T impl_C,T,method_1 ... impl_C,T,method_n
```

erzeugt, wobei $\{T_1\ u, \ldots T_p\ u\}$ *mit* $p \geq 0$ *dem vollständig reduzierten Kontext von* $\{S_1\ u, \ldots, S_k\ u\}$ *und* $\{D_1\ v_y, \ldots D_q\ v_r\}$ *mit* $q \geq 0$ *und* $\{v_y, \ldots, v_r\} \subseteq \{v_x, \ldots, v_s\}$ *dem vollständig reduzierten Kontext von* $\{C_1\ v_x, \ldots, C_g\ v_s\}$ *entspricht. Zudem werden noch die Deklarationen*

```
impl_C,T,method_1  :: (D_1 v_y, ..., D_q v_r) => t'_1
Impl(1)
...
impl_C,T,method_n  :: (D_1 v_y, ..., D_q v_r) => t'_n
Impl(n)
```

mit

$$\text{Impl}(i) = \begin{cases} \text{Impl. von } method_i \text{ umbenannt in } \texttt{impl}_{C,T,method_i} & \text{falls } i \in \{o, \ldots, t\} \\ \texttt{impl}_{C,T,method_i}\ \texttt{= default}_{C,method_i}\ \texttt{inst}_{C,T} & \text{sonst} \end{cases}$$

erzeugt, wobei $t'_1, \ldots, t'_n$ *die transformierten Typausdrücke von* K_1 `=>` $t_1, \ldots, K_n$ `=>` t_n *sind.*

4.3. Behandlung von beschränkten Typen

Alle im Code vorkommenden beschränkten Typen, insbesondere auch jene, die durch die vorherigen Transformationen hinzugefügt wurden, werden für jedes Constraint ihres Kontexts um einen Wörterbuchtyp als zusätzliches Funktionsargument ergänzt. Für die Funktion `elem` aus Unterabschnitt 3.1.1, deren beschränkter Typ

```
Eq a => a -> [a] -> Bool
```

lautete, erhalten wir zum Beispiel

```
Dict_Eq a -> a -> [a] -> Bool
```

als Ergebnis. Die formale Beschreibung dieser Transformation ist wie folgt definiert.

Transformation 4.3. (Beschränkte Typen)**:**
Ein Typausdruck der allgemeinen Form

`(`C_1 u_1`, ...,` C_n u_m`) =>` t

mit $m \geq 0$ *und* $n \geq 0$ *wird zu*

`Dict_`D_1 u_k `-> ... -> Dict_`D_l u_h `->` t

umgewandelt, wobei $\{D_1\ u_k, \ldots, D_l\ u_h\}$ *mit* $l \geq 0$ *und* $\{u_k, \ldots, u_h\} \subseteq \{u_1, \ldots, u_m\}$ *dem vollständig reduzierten Kontext von* $\{C_1\ u_1, \ldots, C_n\ u_m\}$ *entspricht.*

4.4. Einfügen von Wörterbüchern

Nach Durchführung aller vorangegangenen Transformationen erfolgt das Einfügen von Wörterbüchern. Dafür müssen die Typen sämtlicher Funktionen im Quellcode bekannt sein, womit sowohl die allgemeinen als auch die speziellen Typen der Funktionen gemeint sind. Letztere gleichen den allgemeinen Typen, nachdem alle Typvariablen durch diejenigen Typen ersetzt wurden, auf die die Funktionen appliziert werden. Das bedeutet insbesondere, dass sich der spezielle vom allgemeinen Typ unterscheiden kann, wie das folgende Beispiel verdeutlicht.

Beispiel 4.4.:
In dem Ausdruck `x == 3` *besitzt die Funktionsanwendung von* `(==)` *den speziellen Typ* `Eq Int => Int -> Int -> Bool`*, während der allgemeine Typ von* `(==)` `Eq a => a -> a -> Bool` *lautet. Der spezielle Typ ergibt sich durch die Ersetzung der Typvariablen* `a` *durch den Typen* `Int`*.*

Im Folgenden nehmen wir die allgemeinen und speziellen Typen von Funktionen als gegeben an. Unsere spätere Implementierung wird dazu in der Lage sein, diese automatisch zu ermitteln und zur Verfügung zu stellen (s. Abschnitt 8.3 bzw. Unterabschnitt 8.5.9).

Der erste Schritt ist das Hinzufügen zusätzlicher Parameter auf der linken Seite von Funktionsgleichungen auf der Basis des allgemeinen Typs der jeweiligen Funktion. Dabei

wird für jedes Constraint des Kontexts ein *Wörterbuchparameter* hinzugefügt. Dieses Vorgehen korrespondiert konsequenterweise zu der bereits vorgestellten Transformation der beschränkten Typen (s. vorherigen Abschnitt), in der für jedes Constraint ein Wörterbuchtyp als neues Funktionsargument eingefügt wurde. Für die Funktion `elem` wird beispielsweise ein Wörterbuchparameter für das Constraint `Eq a` ergänzt.

```
elem dict_Eq,a _ []     = ...
elem dict_Eq,a x (y:ys) = ...
```

Der zweite Schritt besteht in der Wörterbucherstellung in den Rümpfen der Funktionsgleichungen. Da jede überladene Funktion zuvor um Wörterbuchparameter für ihre Constraints erweitert wurde, müssen nun alle Aufrufe dieser Funktionen entsprechend angepasst werden. Dies geschieht, indem diejenigen konkreten Wörterbücher als Argumente übergeben werden, die sich aus dem speziellen Typ der aufgerufenen Funktion ergeben und – ggf. unter Verwendung der ergänzten Wörterbuchparameter – erstellt werden müssen. Dabei können die folgenden drei Fälle eintreten.

- Es wird eines der als Parameter ergänzten konkrekten Wörterbücher benötigt. In diesem Fall kann dieses direkt als Argument übergeben werden.
- Es bedarf eines Superklassenwörterbuchs aus einem konkreten Wörterbuch. Dann muss der passende Selektor aufgerufen werden.
- Das konkrete Wörterbuch für eine bestimmte Instanz wird benötigt. Dieses kann erzeugt werden, wobei ein eventuell vorhandener Instanzkontext berücksichtigt werden muss.

Im Rumpf der Funktion `elem` wird zum einen der überladene Gleichheitsoperator verwendet, zum anderen ruft sich die Funktion selbst rekursiv auf. Für beide Funktionsaufrufe gilt, dass ihre speziellen Typen identisch mit den allgemeinen sind. Der spezielle Typ für `(==)` ist also `Eq a => a -> a -> Bool` und der für `elem` lautet `Eq a => a -> [a] -> Bool`. Das bedeutet, dass in beiden Fällen ein konkretes Wörterbuchargument für das Constraint `Eq a` ergänzt werden muss. Damit ist der erste der drei infrage kommenden Fälle eingetreten, nach dem der auf der linken Seite ergänzte Wörterbuchparameter verwendet werden kann.

```
elem dict_Eq,a _ []     = False)
elem dict_Eq,a x (y:ys) | (==) dict_Eq,a x y = True
                        | otherwise  = elem dict_Eq,a x ys
```

Nachfolgend wird das beschriebene Verfahren zum Einfügen von Wörterbüchern formalisiert. Die Generierung des Codes für die Wörterbucherstellung kann durch ein Regelsystem beschrieben werden (s. Abbildung 4.1) [Jon92b]. Die dort aufgeführten Regeln stimmen exakt mit den oben erwähnten Fällen überein. Die Syntax der Prämissen und Konklusionen lautet wie folgt.

$$P \Vdash d : C\ t$$

Dabei bezeichnet P einen Kontext, d den generierten Code und $C\ t$ ein Constraint. P steht für die Menge der verfügbaren Wörterbücher und ist durch den allgemeinen Kontext einer Funktion gegeben. Somit entsprechen die verfügbaren Wörterbücher genau den ergänzten Wörterbuchparametern. $C\ t$ ist das Constraint, für das der Code zum Erstellen des konkreten Wörterbuchs generiert werden soll.

$$P \Vdash \texttt{dict}_{C,t} : C\ t \qquad C\ a \in P \qquad (\textit{avail})$$

$$\frac{P \Vdash d : C\ t}{P \Vdash \texttt{super}_{C,C'}\ d : C'\ t} \qquad C' \text{ Superklasse von } C \qquad (\textit{super})$$

$$\frac{P \Vdash d_1 : C_1\ t_{k_1} \quad \ldots \quad P \Vdash d_n : C_n\ t_{k_n}}{P \Vdash \texttt{inst}_{C,T}\ d_1 \ldots d_n : C\ (T\ t_1 \ldots t_m)}$$

Instanz `instance` K `=>` C `(`$T\ u_1 \ldots u_m$`)` mit $m \geq 0$ existiert und $\{C_1\ u_{k_1}, \ldots, C_n\ u_{k_n}\}$ entspricht dem vollständig reduzierten Kontext von K mit $n \geq 0$ und $\{u_{k_1}, \ldots, u_{k_n}\} \subseteq \{u_1, \ldots, u_m\}$ (*inst*)

Abbildung 4.1.: Regeln zur Wörterbucherstellung

Mithilfe der angegebenen Regeln kann nun eine Funktion definiert werden, welche den Code für die Wörterbucherstellung erzeugt. Anschließend kann mit dieser Funktion die Gesamttransformation beschrieben werden.

Definition 4.5. (Wörterbucherstellung)**:**
Die Wörterbucherstellung *generiert für einen Kontext P und ein Constraint C den Code, um aus den durch P gegebenen Wörterbüchern ein Wörterbuch für C zu erstellen.*

$$\text{Dict}(P, C) = d \quad \Leftrightarrow \quad P \Vdash d : C$$

Transformation 4.6. (Funktionen)**:**
Eine Funktion der allgemeinen Form

$f\ \ x_1\ \ \ldots\ \ x_k$ `=` Rumpf

mit $k \geq 0$ und vollständig reduziertem Kontext $P = \{C_1\ t_1, \ldots, C_n\ t_n\}$, wobei $n \geq 0$ gilt, wird für jedes Constraint um ein Wörterbuchparameter ergänzt.

$f\ \ \texttt{dict}_{C_1,t_1}\ \ \ldots\ \ \texttt{dict}_{C_n,t_n}\ \ x_1\ \ \ldots\ \ x_k$ `=` Rumpf

Weiterhin wird für jede im Rumpf von f verwendete (überladene) Funktion g, die den speziellen Kontext $\{D_1, \ldots, D_m\}$ mit $m \geq 0$ besitzt, der Ausdruck für g wie folgt ersetzt.

$g\ \ \text{Dict}(P, D_1)\ \ \ldots\ \ \text{Dict}(P, D_m)$

Abschließend wird noch ein umfangreicheres Beispiel diskutiert, in dem auch die anderen Regeln zur Wörterbucherstellung Anwendung finden.

Beispiel 4.7.:
Es seien die folgenden Klassen und Instanzen im Programm vorhanden.

```
class Eq a
class Eq a => Ord a

instance Eq Int
instance Eq Bool
instance Eq a => Eq [a]
instance (Eq a, Eq b) => Eq (a, b)

instance Ord Bool
instance Ord a => Ord [a]
instance (Ord a, Ord b) => Ord (a, b)
```

Wir betrachten die Funktion `f`*, welche wie folgt definiert ist.*

```
f :: (Ord a, Ord b) => a -> b -> Bool
f x y = (1, [y]) == (1, [y]) && ([True], x) <= ([True], x)
```

Der allgemeine Typ ist durch die Typsignatur gegeben und die speziellen Typen der im Rumpf verwendeten Funktionen lauten

```
Eq (Int, [b]) => (Int, [b]) -> (Int, [b]) -> Bool
```

für `(==)` *respektive*

```
Ord ([Bool], a) => ([Bool], a) -> ([Bool], a) -> Bool
```

für `(<=)`*. Für die Constraints der Kontexte der speziellen Typen müssen konkrete Wörterbücher erstellt werden. Dies geschieht mithilfe der in Abbildung 4.1 angegebenen Regeln, wobei die Ableitungen in Abbildung 4.2 zu sehen sind. Nach Hinzufügen der Wörterbuchparameter und dem Einfügen der erstellten Wörterbücher als Argumente für die Selektoren erhalten wir die folgende, transformierte Funktion.*

```
f :: Dict_Ord a -> Dict_Ord b -> a -> b -> Bool
f dict_Ord,a dict_Ord,b x y =
  (==) (inst_Eq,(,)
         inst_Eq,Int
         (inst_Eq,[] (super_Ord,Eq dict_Ord,b)))
       (1, [y])
       (1, [y])
  &&
  (<=) (inst_Ord,(,)
         (inst_Ord,[] inst_Ord,Bool)
         dict_Ord,a)
       ([True], x)
       ([True], x)
```

$$\dfrac{\dfrac{}{P \Vdash \texttt{inst}_{\texttt{Eq,Int}} : \texttt{Eq Int}}\,(inst) \quad \dfrac{\dfrac{\dfrac{}{P \Vdash \texttt{dict}_{\texttt{Ord,b}} : \texttt{Ord b}}\,(avail)}{P \Vdash \texttt{super}_{\texttt{Ord,Eq}}\ \texttt{dict}_{\texttt{Ord,b}} : \texttt{Eq b}}\,(super)}{P \Vdash \texttt{inst}_{\texttt{Eq,[]}}\ (\texttt{super}_{\texttt{Ord,Eq}}\ \texttt{dict}_{\texttt{Ord,b}}) : \texttt{Eq [b]}}\,(inst)}{P \Vdash \texttt{inst}_{\texttt{Eq,(,)}}\ \texttt{inst}_{\texttt{Eq,Int}}\ (\texttt{inst}_{\texttt{Eq,[]}}\ (\texttt{super}_{\texttt{Ord,Eq}}\ \texttt{dict}_{\texttt{Ord,b}})) : \texttt{Eq (Int, [b])}}\,(inst)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P \Vdash \texttt{inst}_{\texttt{Ord,Bool}} : \texttt{Ord Bool}}\,(inst)}{P \Vdash \texttt{inst}_{\texttt{Ord,[]}}\ \texttt{inst}_{\texttt{Ord,Bool}} : \texttt{Ord [Bool]}}\,(inst) \quad \dfrac{}{P \Vdash \texttt{dict}_{\texttt{Ord,a}} : \texttt{Ord a}}\,(avail)}{P \Vdash \texttt{inst}_{\texttt{Ord,(,)}}\ (\texttt{inst}_{\texttt{Ord,[]}}\ \texttt{inst}_{\texttt{Ord,Bool}})\ \texttt{dict}_{\texttt{Ord,a}} : \texttt{Ord ([Bool], a)}}\,(inst)$$

Abbildung 4.2.: Beispielableitungen zur Wörterbucherstellung

4.5. Besonderheit in funktionallogischen Sprachen

Es gibt in Bezug auf funktionallogische Programmiersprachen eine Besonderheit bei der Implementierung von Typklassen mittels Wörterbüchern zu beachten, auf die wir kurz eingehen möchten.

Aufgrund der in Curry implementierten Call-Time-Choice-Semantik (s. Einschub 2.2 in Abschnitt 2.1) können bei der Verwendung von nullstelligen Klassenmethoden Werte verloren gehen [Lux09]. Wir betrachten dazu das folgende Beispiel.

```
class Arb a where
  arb :: a

instance Arb Bool where
  arb = True ? False

arbs :: Arb a => [a]
arbs = [arb, arb]

arbBools :: [Bool]
arbBools = arbs
```

Die Typklasse `Arb` führt eine nullstellige Klassenmethode `arb` ein, die beliebige Werte eines Typs zurückgibt. Wir geben eine Instanz dieser Klasse für den Typ `Bool` an. Die Funktion `arbs` konstruiert eine zweielementige Liste beliebiger Werte eines Typs und die Funktion `arbBools` tut dies schließlich für den Typ `Bool`.

Die erwarteten Ergebnisse für die Funktion `arbBools` sind die Listen `[False, False]`, `[False, True]`, `[True, False]` und `[True, True]`. Tatsächlich werden aber nur die Listen `[False, False]` und `[True, True]` berechnet. Das Problem hierbei wird ersichtlich, wenn man das Ergebnis der Wörterbuchtransformation für das obige Programm einer genaueren Betrachtung unterzieht.

```
data Dict_Arb a = Dict_Arb a

arb :: Dict_Arb a -> a
arb (Dict_Arb m) = m

default_Arb,arb :: a
default_Arb,arb =
  error "No default implementation for class method 'arb'"

inst_Arb,Bool :: Dict_Arb Bool
inst_Arb,Bool = Dict_Arb impl_Arb,Bool,arb

impl_Arb,Bool,arb :: Bool
impl_Arb,Bool,arb = True ? False

arbs :: Dict_Arb a -> [a]
arbs dict_Arb = [arb dict_Arb, arb dict_Arb]

arbBools :: [Bool]
arbBools = arbs inst_Arb,Bool
```

In der transformierten Funktion `arbBools` wird der Funktion `arbs` das konkrete Wörterbuch $\texttt{inst}_{\texttt{Arb,Bool}}$ als Argument übergeben und für beide Aufrufe von `arb` genutzt. Da $\texttt{impl}_{\texttt{Arb,Bool,arb}}$ aber eine Funktion ohne Argumente ist, führt Currys Call-Time-Choice-Semantik dazu, dass das Argument des Konstruktors $\texttt{Dict}_{\texttt{Arb}}$ nur einmal ausgewertet und dadurch unbeabsichtigterweise in beiden Aufrufen geteilt wird.

Das Problem lässt sich umgehen, indem alle nullstelligen Klassenmethoden um ein zusätzliches Argument, beispielsweise `()`, erweitert werden. Mit dieser Korrektur lautet der transformierte Code dann folgendermaßen.

```
data Dict_Arb a = Dict_Arb (() -> a)

arb :: Dict_Arb a -> () -> a
arb (Dict_Arb m) = m

default_Arb,arb :: () -> a
default_Arb,arb =
  error "No default implementation for class method 'arb'"

inst_Arb,Bool :: Dict_Arb Bool
inst_Arb,Bool = Dict_Arb impl_Arb,Bool,arb

impl_Arb,Bool,arb :: () -> Bool
impl_Arb,Bool,arb () = True ? False
```

```
arbs :: Dict_Arb a -> [a]
arbs dict_Arb = [arb dict_Arb (), arb dict_Arb ()]

arbBools :: [Bool]
arbBools = arbs inst_Arb,Bool
```

Zwar wird das konkrete Wörterbuch erneut für beide Aufrufe von `arb` genutzt, das Ergebnis des Ausdrucks `arb dict_Arb` ist diesmal aber keine Konstante sondern eine einstellige Funktion. Diese wird an beiden Stellen auf das Argument `()` angewandt. Die Auswertung dieser Aufrufe erfolgt an beiden Stellen unabhängig voneinander, sodass die Ergebnisse von `arbBools` nun wie erwartet sind.

Teil II.

Typsystem

5. Hindley-Milner-Typsystem

Das Typsystem von Curry, welches in diesem Kapitel vorgestellt wird, ist im Wesentlichen ein *Hindley-Milner-Typsystem*. Dieses wurde erstmals von J. Roger Hindley beschrieben [Hin69] und später unabhängig von Robin Milner wiederentdeckt [Mil78]. Es ist auch als *Damas-Milner-Typsystem* bekannt, da Luis Damas eine formale Analyse sowie einen Beweis der Methode beitrug [DM82].

5.1. Grundlegende Definitionen

Die zu typenden *Ausdrücke* sind genau jene des Lambda-Kalküls, erweitert um ein zusätzliches `let`-Konstrukt, welches die Definition und Nutzung von polymorphen Ausdrücken erlaubt.

Definition 5.1. (Ausdruck)**:**
Ein Ausdruck *ist durch die folgende Backus-Naur-Form (BNF) gegeben.*

$$\begin{array}{lll} e ::= x & & (\textit{Variable}) \\ \quad \mid \; e_1\ e_2 & & (\textit{Applikation}) \\ \quad \mid \; \lambda x.e & & (\textit{Abstraktion}) \\ \quad \mid \; \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & & (\textit{Let-Ausdruck}) \end{array}$$

Die Applikation $e_1\ e_2$ repräsentiert die Anwendung einer Funktion e_1 auf das Argument e_2 und die Abstraktion $\lambda x.e$ eine anonyme Funktion, die die Eingabe x auf die Ausgabe e abbildet. $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ stellt das Ergebnis der Ersetzung jedes Vorkommens von x in e_2 mit e_1 dar. Im Prinzip kann jedes Sprachkonstrukt in Curry auf die hier definierten Konstrukte zurückgeführt werden.

Auf Typebene wird zwischen *Typen* und *Typschemata* unterschieden. Letztere ermöglichen den parametrischen Polymorphismus wie er eingangs in Kapitel 1 beschrieben wurde.

Definition 5.2. (Typ und Typschema)**:**
Ein Typ *ist durch die folgende BNF gegeben.*

$$\begin{array}{lll} \tau ::= \alpha & & (\textit{Typvariable}) \\ \quad \mid \; \chi\ \tau_1 \ldots \tau_n & & (\textit{Typkonstruktion}) \end{array}$$

Ein Typschema *hat die folgende Form.*

$$\sigma ::= \forall \alpha_1 \ldots \alpha_n.\tau$$

Für einen n-stelligen Typkonstruktor χ repräsentiert die Typkonstruktion $\chi\ \tau_1 \ldots \tau_n$ die vollständige Applikation auf n Typargumente $\tau_1, \ldots, \tau_n$.

Beispiel 5.3.:
Der Typ `Maybe a` *in Curry entspricht der Anwendung des einstelligen Typkonstruktors* `Maybe` *auf die Typvariable* `a`*.*

Die quantifizierten Typvariablen $\alpha_1, \ldots, \alpha_n$ eines Typschemas geben an, dass an deren Stelle beliebige Typen verwendet werden können.

Beispiel 5.4.:
Das Typschema der Funktion `id`*, welches* $\forall$`a.a` $\rightarrow$ `a` *lautet, lässt sich so interpretieren, dass ihr Ergebnis unabhängig von dem für die Typvariable* `a` *eingesetzten Typ immer die Identität ist.*

Die nachfolgend eingeführten Abkürzungen für Typen und Typschemata dienen der besseren Lesbarkeit.

$$\begin{aligned} \tau &\hat{=} \forall\emptyset.\tau \\ \forall\alpha.\sigma &\hat{=} \forall\alpha_1 \ldots \alpha_n\ \alpha.\tau \qquad \text{für } \sigma = \forall\alpha_1 \ldots \alpha_n.\tau \end{aligned}$$

Ein Ausdruck der Form $\mathrm{TV}(X)$ bezeichnet die Menge aller in X vorkommenden Typvariablen, wobei X sowohl ein Typ, ein Typschema als auch eine Typannahme (s. Abschnitt 5.2) sein kann.
Zu guter Letzt wird noch der Begriff der *generischen Instanz* definiert, welcher in den nachfolgenden Abschnitten benötigt wird.

Definition 5.5. (Generische Instanz)**:**
Ein Typschema $\sigma' = \forall\beta_1 \ldots \beta_m.\tau'$ *ist genau dann eine* generische Instanz *eines Typschemas* $\sigma = \forall\alpha_1 \ldots \alpha_n.\tau$*, wenn* $\tau' = [\alpha_i/\tau_i]\tau$ *für beliebige Typen* $\tau_1, \ldots, \tau_n$ *gilt und die Typvariablen* $\beta_1, \ldots, \beta_m$ *nicht frei in* σ *vorkommen.*

Wir schreiben auch $\sigma' \leq \sigma$ für den Fall, dass σ' eine generische Instanz von σ ist.

5.2. Typungsregeln

Wenn e ein Ausdruck ist, dann steht $A \vdash e : \sigma$ für die Aussage, dass e bezüglich der Typannahme A das Typschema σ besitzt. Hierbei ist eine *Typannahme* wie folgt definiert.

Definition 5.6. (Typannahme)**:**
Eine Typannahme *ist eine endliche Menge von Zuordnungen von Variablen zu Typschemata der Form* $x : \sigma$*, in der keine Variable* x *mehrfach vorkommt.*

A_x bezeichne die Typannahme, die man aus A erhält, wenn man die Zuordnung für die Variable x entfernt.

Es gilt genau dann $A \vdash e : \sigma$, wenn sich dies mithilfe der in Abbildung 5.1 gegebenen Inferenzregeln ableiten lässt. Dabei behandeln die Regeln $(\forall E)$ und $(\forall I)$ den Ab- bzw. Aufbau von Typschemata. Die restlichen Regeln (*var*), $(\to E)$, $(\to I)$ sowie (*let*) orientieren sich an der Syntax und dienen der Herleitung der Typen von Variablen, Applikationen, Abstraktionen bzw. Let-Ausdrücken. Bei der Regel (*let*) ist zu beachten, dass der Typ des Ausdrucks e_1 in ein Typschema umgewandelt wird, um die erwähnte polymorphe Verwendung von Ausdrücken zu ermöglichen.

$$\frac{(x : \sigma) \in A}{A \vdash x : \sigma} \quad (var)$$

$$\frac{A \vdash e_1 : \tau_2 \to \tau_1 \qquad A \vdash e_2 : \tau_2}{A \vdash (e_1\ e_2) : \tau_1} \quad (\to E)$$

$$\frac{A_x \cup \{x : \tau_2\} \vdash e : \tau_1}{A \vdash (\lambda x.e) : \tau_2 \to \tau_1} \quad (\to I)$$

$$\frac{A \vdash e_1 : \sigma \qquad A_x \cup \{x : \sigma\} \vdash e_2 : \tau}{A \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \tau} \quad (let)$$

$$\frac{A \vdash e : \sigma_2 \qquad \sigma_2 \leq \sigma_1}{A \vdash e : \sigma_1} \quad (\forall E)$$

$$\frac{A \vdash e : \sigma \qquad \alpha \notin \mathrm{TV}(A)}{A \vdash e : \forall \alpha.\sigma} \quad (\forall I)$$

Abbildung 5.1.: Typungsregeln für das Hindley-Milner-Typsystem

In Curry dagegen würde die Regel (*let*) zu einem Problem führen, da die Typen freier Variablen immer *monomorph* sein müssen, d.h., freie Variablen dürfen nur einen festen Typ haben. Wäre dem nicht so, bestünde die Möglichkeit, ungültige Funktionen wie

```
bug = x =:= 1 & x =:= 'a' where x free
```

anzugeben. Freie Variablen in Curry machen also eine *Monomorphierestriktion* erforderlich, die es garantiert, dass die Typen von lokalen Variablen in Let-Ausdrücken nicht in Typschemata umgewandelt werden. Für Curry wird daher eine alternative Regel für Let-Ausdrücke bereitgestellt (s. Abbildung 5.2).

$$\frac{A \vdash e_1 : \tau_1 \qquad A_x \cup \{x : \tau_1\} \vdash e_2 : \tau_2}{A \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \tau_2} \quad (let')$$

Abbildung 5.2.: Typungsregel für Let-Ausdrücke in Curry

In der Praxis wird der Typ von nicht-nullstelligen Funktionen dennoch in ein Typschema umgewandelt, um die polymorphe Verwendung dieser zu erlauben. Dies ist in den obigen Regeln nicht berücksichtigt.

5.3. Typinferenzalgorithmus

Die Regeln in Abbildung 5.1 sind zwar dazu geeignet, um den Typ eines Ausdrucks gegen eine gegebene Typannahme zu validieren, stellen allerdings keine geeignete Basis für einen Typinferenzalgorithmus zur automatischen Typermittlung dar. Es gibt unter Umständen mehr als nur eine anwendbare Regel und es ist nicht immer klar, welche dieser Regeln das beste Ergebnis liefert.

Bevor eine alternative Menge von Regeln angegeben werden kann, die eben diese Probleme vermeidet, muss das Konzept der *Unifikation* eingeführt werden.

Definition 5.7. (Unifikator)**:**
Eine Substitution S ist genau dann ein Unifikator *zweier Typen τ und τ', wenn $S\tau = S\tau'$ gilt. Die Typen τ and τ' werden auch als* unifizierbar *bezeichnet, wenn ein Unifikator für diese existiert.*

Definition 5.8. (Allgemeinster Unifikator)**:**
Ein Unifikator S ist genau dann ein allgemeinster Unifikator *zweier Typen, wenn für jeden Unifikator R eine Substitution S' existiert, sodass $R = S'S$ gilt.*

Wir schreiben auch $\tau \overset{U}{\sim} \tau'$, wenn zwei Typen τ und τ' mit dem allgemeinsten Unifikator U unifizierbar sind. Entscheidend ist, dass ein solcher allgemeinster Unifikator stets gefunden werden kann, wenn zwei Typen miteinander unifizierbar sind.

Satz 5.9. (Existenz und Berechenbarkeit eines allgemeinsten Unifikators, [Rob65])**:**
Falls zwei Typen miteinander unifizierbar sind, dann existiert ein allgemeinster Unifikator für diese Typen, welcher effektiv berechenbar ist.

Weiterhin wird der Begriff der *Generalisierung* eines Typen benötigt.

Definition 5.10. (Generalisierung)**:**
Die Generalisierung *eines Typen τ in Bezug auf eine Typannahme A quantifiziert alle Typvariablen, die frei in τ sind, aber nicht in A vorkommen.*

$$\mathrm{Gen}(A, \tau) = \forall \alpha_1 \ldots \alpha_n . \tau \qquad \{\alpha_1, \ldots, \alpha_n\} = \mathrm{TV}(\tau) \setminus \mathrm{TV}(A)$$

Mit diesen Mitteln ist es uns nun möglich, einen entsprechenden Typinferenzalgorithmus zur automatischen Typermittlung zu definieren. Dieser ist durch die Regeln in Abbildung 5.3 gegeben und als *Algorithmus $\mathcal{W}$* bekannt.

Der Algorithmus $\mathcal{W}$ ist *korrekt* und *vollständig*, was die beiden folgenden Sätze aussagen.

$$\frac{(x : \forall\alpha_1 \ldots \alpha_n.\tau) \in A}{A \overset{\mathcal{W}}{\vdash} x : [\alpha_i/\beta_i]\tau} \quad \beta_i \text{ neu} \qquad (var)^W$$

$$\frac{S_1 A \overset{\mathcal{W}}{\vdash} e_1 : \tau_1 \qquad S_2 S_1 A \overset{\mathcal{W}}{\vdash} e_2 : \tau_2 \qquad S_2\tau_1 \overset{U}{\sim} \tau_2 \to \alpha}{U S_2 S_1 A \overset{\mathcal{W}}{\vdash} (e_1\ e_2) : U\alpha} \quad \alpha \text{ neu} \qquad (\to E)^W$$

$$\frac{S(A_x \cup \{x : \alpha\}) \overset{\mathcal{W}}{\vdash} e : \tau}{SA \overset{\mathcal{W}}{\vdash} (\lambda x.e) : S\alpha \to \tau} \quad \alpha \text{ neu} \qquad (\to I)^W$$

$$\frac{S_1 A \overset{\mathcal{W}}{\vdash} e_1 : \tau_1 \qquad S_2(S_1 A_x \cup \{x : \mathrm{Gen}(S_1 A, \tau_1)\}) \overset{\mathcal{W}}{\vdash} e_2 : \tau_2}{S_2 S_1 A \overset{\mathcal{W}}{\vdash} (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \tau_2} \qquad (let)^W$$

Abbildung 5.3.: Algorithmus $\mathcal{W}$ für das Hindley-Milner-Typsystem

Satz 5.11. (Korrektheit des Algorithmus $\mathcal{W}$, [Dam84])**:**
Wenn $SA \overset{\mathcal{W}}{\vdash} e : \tau$ gilt, dann gilt auch $SA \vdash e : \tau$.

Satz 5.12. (Vollständigkeit des Algorithmus $\mathcal{W}$, [Dam84])**:**
Es gelte $SA \vdash e : \sigma$. Dann ist $TA \overset{\mathcal{W}}{\vdash} e : \tau$ ableitbar und es gibt eine Substitution R, sodass $SA = RTA$ und $\sigma \leq R\,\mathrm{Gen}(TA, \tau)$ gilt.

Es wäre wünschenswert, dass der Algorithmus $\mathcal{W}$ nicht nur irgendein gültiges Typschema zu einem Ausdruck berechnet, sondern stets das *allgemeinste*, das folgendermaßen definiert ist.

Definition 5.13. (Allgemeinstes Typschema)**:**
Seien eine Typannahme A sowie ein Ausdruck e gegeben. Ein Typschema σ ist genau dann ein allgemeinstes Typschema *von e unter A, wenn $A \vdash e : \sigma$ ableitbar ist und $\sigma' \leq \sigma$ für jede Substitution σ' mit $A \vdash e : \sigma'$ gilt.*

Glücklicherweise ist dies der Fall. Der nachfolgende Satz ist eine direkte Folgerung aus Satz 5.12 und sagt aus, dass der Algorithmus $\mathcal{W}$ immer das allgemeinste Typschema berechnet, sofern dies möglich ist.

Satz 5.14. (Berechnung eines allgemeinsten Typschemas, [Dam84])**:**
Ist $A \vdash e : \sigma$ für ein Typschema σ ableitbar, dann berechnet der Algorithmus $\mathcal{W}$ ein allgemeinstes Typschema für den Ausdruck e unter der Typannahme A.

6. Erweitertes Typsystem

Das bestehende Typsystem von Curry kann auf natürliche Art und Weise um die Unterstützung von Typklassen erweitert werden. Zentrale Begriffe hierbei sind *Prädikate* und *prädizierte Typen*. Die hinter diesen Begriffen stehende Theorie, welche in diesem Kapitel zusammenfassend erläutert wird, wurde erstmals ausführlich von Mark P. Jones beschrieben [Jon92a; Jon92b]. Typklassen stellen letztendlich nur einen Spezialfall dieser Theorie dar. Für die Unterstützung von Typkonstruktorklassen ist darüber hinaus die Einführung von *Sorten* für Typen erforderlich. Auch dieser Zusammenhang wurde von Mark P. Jones herausgestellt [Jon93].

6.1. Sorten und Konstruktoren

Wenn man sich die Typkonstruktorklasse `Functor` aus Abschnitt 3.1 ins Gedächtnis ruft, dann kann der Typkonstruktor `Int` keine Instanz dieser Typklasse sein, da der Typ `(a -> b) -> Int a -> Int b` der assoziierten Methode `fmap` offensichtlich nicht wohlgeformt ist. Um Fälle wie diesen zu vermeiden, muss sichergestellt werden, dass alle Elemente einer gegebenen Typklasse von derselben *Sorte* sind.

Definition 6.1. (Sorte)**:**
Eine Sorte *ist durch die folgende BNF gegeben.*

$$\begin{aligned} \kappa ::= & * \\ \mid & \ \kappa_1 \to \kappa_1 \end{aligned}$$

Gewöhnliche Typen, wie sie in Abschnitt 5.1 vorgestellt wurden, sind von der Sorte $*$, während Typkonstruktoren von der Sorte $\kappa_1 \to \kappa_2$ sind, wobei κ_1 und κ_2 wiederum Sorten sind.

Beispiel 6.2.:
Die Typen `Bool` *und* `Int` *in Curry sind von der Sorte* $*$*. Der Typkonstruktor* `Maybe` *dagegen ist von der Sorte* $* \to *$*.*

Um das Beispiel der Typkonstruktorklasse `Functor` aufzugreifen: Alle Elemente dieser Typklasse müssen folglich von der Sorte $* \to *$ sein, um eine gültige Instanz darzustellen, womit zum Beispiel `Functor Int` ausgeschlossen wird.

Dieser Ansatz kann formalisiert werden, indem eine Sprache für *Konstruktoren* eingeführt wird, in der jedes Vorkommen eines Konstruktors mit seiner jeweiligen Sorte annotiert ist.

Definition 6.3. (Konstruktor)**:**
Die Menge der Konstruktoren *einer Sorte κ ist durch die folgende BNF gegeben.*

$$\begin{aligned} C^\kappa ::= {} & \alpha^\kappa && (\textit{Konstruktorvariable}) \\ \mid {} & \chi^\kappa && (\textit{Konstruktorkonstante}) \\ \mid {} & C^{\kappa' \to \kappa}\ C^{\kappa'} && (\textit{Konstruktorapplikation}) \end{aligned}$$

Im Folgenden werden Substitutionen nur noch auf *Sorten-erhaltende Substitutionen* beschränkt. Dies sind solche, die Konstruktorvariablen ausschließlich auf Konstruktoren gleicher Sorte abbilden.

Die Typen von Ausdrücken entsprechen unter Verwendung der obigen Definition genau den Konstruktoren der Sorte $*$.

Definition 6.4. (Typ)**:**
Ein Typ *hat die folgende Form.*

$$\tau ::= C^* \qquad (\textit{Typ})$$

Analog zu Abschnitt 5.1 steht ein Ausdruck wie $\mathrm{CV}(X)$ für die Menge aller in X vorkommenden Konstruktorvariablen. Dabei kann X ein Konstruktor, ein (prädizierter) Typ, ein (beschränktes) Typschema oder auch eine Typannahme sein.

6.2. Prädikate und prädizierte Typen

Ein *Prädikat* ist folgendermaßen definiert.

Definition 6.5. (Prädikat)**:**
Ein Prädikat *ist ein Ausdruck der Form $\pi = p\ \tau_1 \ldots \tau_n$, wobei p ein Prädikatensymbol ist, das zu einer n-stelligen Relation korrespondiert, und $\tau_1, \ldots, \tau_n$ Typen sind. Das Prädikat π repräsentiert die Annahme, dass die Typen $\tau_1, \ldots, \tau_n$ in der durch p beschriebenen Relation enthalten sind.*

Die Eigenschaften von Prädikaten werden durch eine Folgerungsrelation $\Vdash$ zwischen endlichen Mengen von Prädikaten erfasst. Diese muss die folgenden drei Gesetze erfüllen.

- *Monotonie*: Wenn $Q \subseteq P$ gilt, dann gilt auch $P \Vdash Q$.
- *Transitivität*: Wenn $P \Vdash Q$ und $Q \Vdash R$ gelten, dann gilt auch $P \Vdash R$.
- *Abgeschlossenheit*: Wenn S eine Substitution ist und $P \Vdash Q$ gilt, dann gilt auch $SP \Vdash SQ$.

Der Ausdruck $P \Vdash Q$ bedeutet, dass die Prädikate in Q gelten, wann immer die Prädikate in P erfüllt sind.

Im speziellen Fall der Typklassen werden die Klassennamen als Prädikatensymbole genutzt und die dazugehörigen Relationen sind immer einstellig. Als Folgerungsrelation

wird jene gewählt, die in Unterabschnitt 3.2.5 präsentiert wurde. Somit entspricht ein Prädikat genau den bereits im selben Abschnitt vorgestellten Constraints.

Auf Basis der Prädikate können nun *prädizierte Typen* eingeführt sowie die Definition der *Typschemata* neu formuliert werden.

Definition 6.6. (Prädizierter Typ und Typschema)**:**
Ein prädizierter Typ *und ein* Typschema *haben jeweils die folgende Form.*

$$\rho ::= \{\pi_1, \ldots, \pi_n\} \Rightarrow \tau \qquad (\textit{prädizierter Typ})$$
$$\sigma ::= \forall \alpha_1^{\kappa_1} \ldots \alpha_n^{\kappa_n}.\rho \qquad (\textit{Typschema})$$

So wie die Prädikate den bereits bekannten Constraints entsprechen, korrespondieren die prädizierten Typen zu den beschränkten Typen.

Erneut werden einige Abkürzungen für prädizierte Typen und Typschemata verwendet, um die Lesbarkeit zu erhöhen.

$$\tau \mathrel{\hat{=}} \emptyset \Rightarrow \tau$$
$$\pi \Rightarrow \rho \mathrel{\hat{=}} P \cup \{\pi\} \Rightarrow \tau \qquad \text{für } \rho = P \Rightarrow \tau$$
$$\rho \mathrel{\hat{=}} \forall \emptyset.\rho$$
$$\forall \alpha^{\kappa}.\sigma \mathrel{\hat{=}} \forall \alpha_1^{\kappa_1} \ldots \alpha_n^{\kappa_n} \alpha^{\kappa}.\rho \qquad \text{für } \sigma = \forall \alpha_1^{\kappa_1} \ldots \alpha_n^{\kappa_n}.\rho$$

6.3. Beschränkte Typschemata

Im Kontext von prädizierten Typen ist es sinnvoll, *beschränkte Typschemata* einzuführen.

Definition 6.7. (Beschränktes Typschema)**:**
Ein beschränktes Typschema *ist ein Paar der Form* $(P \mid \sigma)$*, wobei* P *eine Menge von Prädikaten und* σ *ein Typschema ist.*

Dem Aufbau des Hindley-Milner-Typsystems aus Kapitel 5 folgend redefinieren wir auf Basis von beschränkten Typschemata den Begriff der *generischen Instanz*.

Definition 6.8. (Generische Instanz)**:**
Ein Typschema $\sigma' = \forall \beta_1^{\iota_1} \ldots \beta_m^{\iota_m}.P' \Rightarrow \tau'$ *ist genau dann eine* generische Instanz *eines beschränkten Typschemas* $(Q \mid \forall \alpha_1^{\kappa_1} \ldots \alpha_n^{\kappa_n}.P \Rightarrow \tau)$*, wenn* $\tau' = [\alpha_i^{\kappa_i}/C_i]\tau$ *und* $P' \Vdash Q \cup [\alpha_i^{\kappa_i}/C_i]P$ *für beliebige Konstruktoren* $C_1 \in C^{\kappa_1}, \ldots, C_n \in C^{\kappa_n}$ *gilt und die Konstruktorvariablen* $\beta_1^{\iota_1}, \ldots, \beta_m^{\iota_m}$ *nicht frei in* σ *vorkommen.*

Wie schon zuvor wird die Tatsache, dass σ' eine generische Instanz von σ ist, mit $\sigma' \leq \sigma$ notiert.

Als Nächstes wird eine *Ordnung auf beschränkten Typschemata* definiert, die es uns erlaubt, Aussagen darüber zu treffen, wann ein beschränktes Typschema *allgemeiner* als ein anderes ist.

Definition 6.9. (Ordnung auf beschränkten Typschemata)**:**
Ein beschränktes Typschema $(P \mid \sigma)$ *ist genau dann* allgemeiner *als ein beschränktes Typschema* $(P' \mid \sigma')$*, geschrieben* $(P' \mid \sigma') \leq (P \mid \sigma)$*, wenn jede generische Instanz von* $(P \mid \sigma)$ *auch eine generische Instanz von* $(P' \mid \sigma')$ *ist.*

Da ein Typschema σ äquivalent zu dem beschränkten Typschema $(\emptyset \mid \sigma)$ ist, können wir auch Typschemata mit der Relation $\leq$ vergleichen. Insbesondere gilt $(P \mid \rho) \leq P \Rightarrow \rho$ und $P \Rightarrow \rho \leq (P \mid \rho)$. Das beschränkte Typschema $(P \mid \rho)$ und das Typschema $P \Rightarrow \rho$ sind somit bzgl. der Ordnung $\leq$ äquivalent und beliebig austauschbar.

6.4. Typungsregeln

Auch für das erweiterte Typsystem lassen sich entsprechende Typungsregeln angeben. Zu diesem Zweck müssen die Inferenzregeln um Prädikate erweitert werden: Der Ausdruck $P \mid A \vdash e : \sigma$ steht nun dafür, dass e bezüglich der Typannahme A das Typschema σ besitzt, wenn die Prädikate in P erfüllt sind.

Die Inferenzregeln für das erweiterte Typsystem werden in Abbildung 6.1 dargestellt. Die Regeln (var), $(\rightarrow E)$, $(\rightarrow I)$ und (let) entsprechen stark denjenigen des ursprünglichen Typsystems und auch die Regeln $(\forall E)$ und $(\forall I)$ wurden lediglich in Hinblick auf die neue Typdarstellung angepasst. Neu sind dagegen die beiden Regeln $(\Rightarrow E)$ und $(\Rightarrow I)$, in denen prädizierte Typen ab- bzw. aufgebaut werden. Zu diesem Zweck kann entweder ein Prädikat π entfernt werden, wenn dieses von der Prädikatenmenge P impliziert wird, oder hinzugefügt werden, wenn es zur korrekten Typung benötigt wird.

6.5. Typinferenzalgorithmus

Der Typinferenzalgorithmus für das erweiterte Typsystem basiert ebenso wie derjenige für das Hindley-Milner-Typsystem aus Abschnitt 5.3 auf der Unifikation. Allerdings handelt es sich nun um die Unifikation von Konstruktoren anstelle von einfachen Typen. Diese ist ein wenig komplizierter, da die Sorten der involvierten Konstruktoren berücksichtigt werden müssen.

Zu Beginn erfolgt erneut die Definition eines *Unifikators* sowie eines *allgemeinsten Unifikators*, welche dieses Mal auf Basis von Sorten-erhaltenden Substitutionen definiert sind.

Definition 6.10. (Unifikator)**:**
Eine Sorten-erhaltende Substitution S *ist genau dann ein* Unifikator *zweier Konstruktoren* $C, C' \in C^\kappa$*, wenn* $SC = SC'$ *gilt. Die Konstruktoren* C *und* C' *werden auch als* unifizierbar *bezeichnet, wenn ein Unifikator für diese existiert.*

Definition 6.11. (Allgemeinster Unifikator)**:**
Ein Unifikator S *ist genau dann ein* allgemeinster Unifikator *zweier Konstruktoren, wenn für jeden Unifikator* R *eine Substitution* S' *existiert, sodass* $R = S'S$ *gilt.*

$$\frac{(x:\sigma)\in A}{P \mid A \vdash x:\sigma} \quad (var)$$

$$\frac{P \mid A \vdash e_1 : \tau_2 \to \tau_1 \qquad P \mid A \vdash e_2 : \tau_2}{P \mid A \vdash (e_1\ e_2) : \tau_1} \quad (\to E)$$

$$\frac{P \mid A_x \cup \{x:\tau_2\} \vdash e : \tau_1}{P \mid A \vdash (\lambda x.e) : \tau_2 \to \tau_1} \quad (\to I)$$

$$\frac{P \mid A \vdash e_1 : \tau_1 \qquad Q \mid A_x \cup \{x:\tau_1\} \vdash e_2 : \tau_2}{P \cup Q \mid A \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \tau_2} \quad (let)$$

$$\frac{P \mid A \vdash e : \pi \Rightarrow \rho \qquad P \Vdash \{\pi\}}{P \mid A \vdash e : \rho} \quad (\Rightarrow E)$$

$$\frac{P \cup \{\pi\} \mid A \vdash e : \rho}{P \mid A \vdash e : \pi \Rightarrow \rho} \quad (\Rightarrow I)$$

$$\frac{P \mid A \vdash e : \forall \alpha^\kappa.\sigma \qquad C \in C^\kappa}{P \mid A \vdash e : [\alpha^\kappa / C]\sigma} \quad (\forall E)$$

$$\frac{P \mid A \vdash e : \sigma \qquad \alpha^\kappa \notin \mathrm{CV}(A) \cup \mathrm{CV}(P)}{P \mid A \vdash e : \forall \alpha^\kappa.\sigma} \quad (\forall I)$$

Abbildung 6.1.: Typungsregeln für das erweiterte Typsystem

Wir schreiben auch $C \overset{U}{\sim}_\kappa C'$, wenn zwei Konstruktoren $C, C' \in C^\kappa$ mit dem allgemeinsten Unifikator U unifizierbar sind. Die Regeln in Abbildung 6.2 definieren einen *Unifikationsalgorithmus*, der für zwei Konstruktoren einen allgemeinsten Unifikator berechnet [Jon93].

$$\alpha \overset{id}{\sim}_\kappa \alpha \quad (idVar)$$

$$\chi \overset{id}{\sim}_\kappa \chi \quad (idConst)$$

$$\alpha \overset{[\alpha/C]}{\sim}_\kappa C \qquad \alpha \notin \mathrm{CV}(C) \quad (bindVar)$$

$$C \overset{[\alpha/C]}{\sim}_\kappa \alpha \qquad \alpha \notin \mathrm{CV}(C) \quad (bindVar')$$

$$\frac{C_1 \overset{S_1}{\sim}_{\kappa_2 \to \kappa_1} D_1 \qquad S_1 C_2 \overset{S_2}{\sim}_{\kappa_2} S_1 D_2}{C_1 C_2 \overset{S_2 S_1}{\sim}_{\kappa_1} D_1 D_2} \quad (apply)$$

Abbildung 6.2.: Unifikationsalgorithmus für Konstruktoren

Die Definition der *Generalisierung* lässt sich direkt auf prädizierte Typen übertragen.

Definition 6.12. (Generalisierung)**:**
Die Generalisierung *eines prädizierten Typen ρ in Bezug auf eine Typannahme A quantifiziert alle Konstruktorvariablen, die frei in ρ sind, aber nicht in A vorkommen.*

$$\mathrm{Gen}(A, \rho) = \forall \alpha_1^{\kappa_1} \ldots \alpha_n^{\kappa_n}.\rho \qquad \{\alpha_1^{\kappa_1}, \ldots, \alpha_n^{\kappa_n}\} = \mathrm{CV}(\rho) \setminus \mathrm{CV}(A)$$

Der resultierende Algorithmus $\mathcal{W}$ für das erweiterte Typsystem ist in Abbildung 6.3 zu sehen.

$$\frac{(x : \forall \alpha_1^{\kappa_1} \ldots \alpha_n^{\kappa_n}.P \Rightarrow \tau) \in A}{P[\alpha_i^{\kappa_i}/\beta_i^{\kappa_i}] \mid A \overset{\mathcal{W}}{\vdash} x : [\alpha_i^{\kappa_i}/\beta_i^{\kappa_i}]\tau} \quad \beta_i^{\kappa_i} \text{ neu} \quad (var)^W$$

$$\frac{P \mid S_1 A \overset{\mathcal{W}}{\vdash} e_1 : \tau_1 \qquad Q \mid S_2 S_1 A \overset{\mathcal{W}}{\vdash} e_2 : \tau_2 \qquad S_2\tau_1 \overset{U}{\sim}_* \tau_2 \to \alpha^*}{U(S_2 P \cup Q) \mid U S_2 S_1 A \overset{\mathcal{W}}{\vdash} (e_1\ e_2) : U\alpha^*} \quad \alpha^* \text{ neu} \quad (\to E)^W$$

$$\frac{P \mid S(A_x \cup \{x : \alpha^*\}) \overset{\mathcal{W}}{\vdash} e : \tau}{P \mid SA \overset{\mathcal{W}}{\vdash} (\lambda x.e) : S\alpha^* \to \tau} \quad \alpha^* \text{ neu} \quad (\to I)^W$$

$$\frac{P_1 \mid S_1 A \overset{\mathcal{W}}{\vdash} e_1 : \tau_1 \qquad P_2 \mid S_2(S_1 A_x \cup \{x : \mathrm{Gen}(S_1 A, P_1 \Rightarrow \tau_1)\}) \overset{\mathcal{W}}{\vdash} e_2 : \tau_2}{P_2 \mid S_2 S_1 A \overset{\mathcal{W}}{\vdash} (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \tau_2} \quad (let)^W$$

Abbildung 6.3.: Algorithmus $\mathcal{W}$ für das erweiterte Typsystem

Die Eigenschaften der Korrektheit und Vollständigkeit sind auch für den Algorithmus $\mathcal{W}$ für das erweiterte Typsystem gegeben.

Satz 6.13. (Korrektheit des Algorithmus $\mathcal{W}$, [Jon92b])**:**
Wenn $P \mid SA \overset{\mathcal{W}}{\vdash} e : \tau$ gilt, gilt auch $P \mid SA \vdash e : \tau$.

Satz 6.14. (Vollständigkeit des Algorithmus $\mathcal{W}$, [Jon92b])**:**
Es gelte $P \mid SA \vdash e : \sigma$. Dann ist $Q \mid TA \overset{\mathcal{W}}{\vdash} e : \tau$ ableitbar und es gibt eine Substitution R, sodass $SA = RTA$ und $(P \mid \sigma) \leq R\,\mathrm{Gen}(TA, Q \Rightarrow \tau)$ gilt.

Gleichermaßen trifft für den Algorithmus $\mathcal{W}$ für das erweiterte Typsystem zu, dass dieser stets das allgemeinste Typschema berechnet, dessen Definition wie folgt auf beschränkte Typschemata angehoben werden kann.

Definition 6.15. (Allgemeinstes Typschema)**:**
Seien eine Typannahme A sowie ein Ausdruck e gegeben. Ein beschränktes Typschema $(P \mid \sigma)$ ist genau dann ein allgemeinstes Typschema von e unter A, wenn $P \mid A \vdash e : \sigma$ ableitbar ist und $(P' \mid \sigma') \leq (P \mid \sigma)$ für jede Substitution σ' mit $P' \mid A \vdash e : \sigma'$ gilt.

Satz 6.16. (Berechnung eines allgemeinsten Typschemas, [Jon92b])**:**
Ist $P \mid A \vdash e : \sigma$ für ein Typschema σ ableitbar, dann berechnet der Algorithmus $\mathcal{W}$ für das erweiterte Typsystem ein allgemeinstes Typschema für den Ausdruck e unter der Typannahme A.

Teil III.

Implementierung

7. Aufbau des Front-Ends

Bevor wir auf die notwendigen Anpassungen des Front-Ends eingehen, die die Unterstützung für Typklassen ermöglichen, gehen wir in diesem Kapitel zunächst auf dessen allgemeine Struktur ein.

7.1. Ausgabeformate

Das Front-End ist dazu in der Lage abhängig vom spezifizierten Zielformat, verschiedene Ausgaben zu erzeugen, welche wir nachfolgend mitsamt ihres jeweiligen Zwecks erläutern.

7.1.1. FlatCurry

Die Generierung von *FlatCurry* bildet die Hauptaufgabe des Front-Ends und ermöglicht die Anbindung mehrerer Back-Ends, welche den FlatCurry-Code in andere Sprachen übersetzen. So kompiliert das KiCS2 Curry-Programme nach Haskell, während das PAKCS Prolog-Programme erzeugt. Die Aufteilung des Compilers in Front- und Back-End ist in Abbildung 7.1 illustriert.

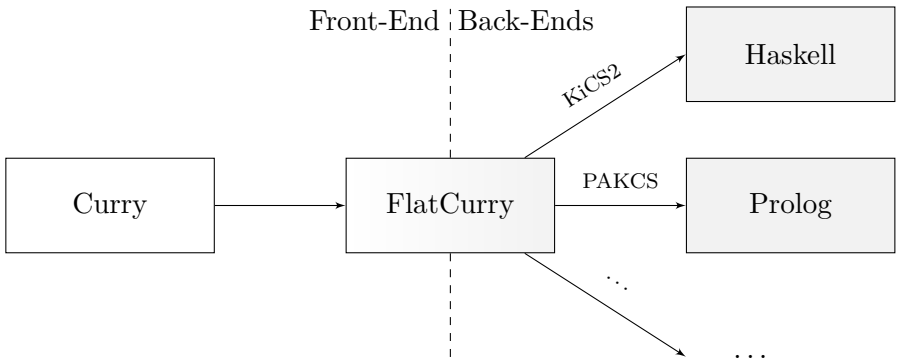

Abbildung 7.1.: Aufteilung des Compilers in ein Front-End und mehrere Back-Ends

Bei FlatCurry handelt es sich um eine stark reduzierte Variante von Curry, wobei diese Reduzierung, bei der vor allem funktionale Merkmale entfernt werden, der einfacheren Anbindung der verschiedenen Back-Ends an das Front-End dient. Ein FlatCurry-Modul besteht im Wesentlichen nur noch aus Datentyp- und Funktionsdeklarationen. Während die Datentypdeklarationen dieselbe Form wie in Curry haben, sind Funktionsdeklarationen auf die folgende Art und Weise vereinfacht.

- Funktionen führen kein *Pattern Matching* mehr durch. Stattdessen wird dieses durch Case-Ausdrücke implementiert.
- Jede Funktionsdefinition besteht nur noch aus einer einzigen Regel. Dies wird durch das Zusammenfassen mehrerer Regeln in Case-Ausdrücke sowie das explizite Ausdrücken von Nichtdetermismus ermöglicht.
- Es gibt auf Ausdrucksebene nur noch (freie) Variablen, Literale, Funktionsapplikationen, Case-Ausdrücke und Let-Ausdrücke, welche nur noch einfache Variablen- und keine Funktionsdefinitionen mehr enthalten.
- Muster bestehen nur noch aus Datenkonstruktoren, angewandt auf Variablen, oder aus Literalen.

7.1.2. AbstractCurry

Bei *AbstractCurry* handelt es sich um die abstrakte Repräsentation eines Curry-Moduls. Das Curry-System stellt verschiedene Bibliotheken bereitstellt, um diese abstrakte Darstellung einzulesen, Transformationen auf ihr durchzuführen und sie wieder auszugeben.

Generell wird zwischen getyptem und ungetyptem AbstractCurry unterschieden. Während beim getypten AbstractCurry sämtliche Funktionsdeklarationen mit ihrem jeweiligen Typ annotiert werden, wird bei ungetyptem AbstractCurry darauf verzichtet.

7.1.3. HTML

Die HTML-Ausgabe bietet neben Syntaxhighlighting auch Querverweise für Funktionen und Typkonstruktoren – sowohl innerhalb eines Moduls als auch modulübergreifend.

7.1.4. Token-Stream

Der Token-Stream entspricht der aus der lexikographischen Analyse hervorgegangenen Tokenfolge (s. Unterabschnitt 7.3.1). Er wird unter anderem von einem Werkzeug von Katharina Rahf zum Überprüfen von Stilrichtlinien für Curry-Code genutzt [Rah16].

7.2. Compiler-Umgebung

Während des Kompilierprozesses hält der Compiler eine Umgebung mit wichtigen, für den Kompiliervorgang benötigten Informationen vor. Neben dem Modul- und Dateinamen des aktuellen Moduls sowie einer Liste aller für das Modul aktivierten Spracherweiterungen enthält die Compiler-Umgebung vor allem weitere Unterumgebungen.

Einschub 7.1. (Entitätenumgebung)**:**
Mit Ausnahme der Schnittstellenumgebung handelt es sich bei allen nachfolgenden Unterumgebungen um sogenannte Entitätenumgebungen*. Diese implementieren die Mehrdeutigkeitssemantik in Curry, die sich grundlegend von derjenigen in Haskell unterscheidet. Eine Entitätenumgebung ist eine Zuordnung von Namen zu* Entitäten *[DJH02], wobei letztere Funktionen, Datenkonstruktoren, Typkonstruktoren etc. sind.*

Dabei besitzt jede Entität einen Originalnamen und es wird darüber hinaus die Information gespeichert, ob eine Entität lokal definiert oder importiert wurde. Der eindeutige Originalname gibt an, aus welchem Modul eine Entität ursprünglich stammt.

Unter einem Namen – qualifiziert oder unqualifiziert – können unter Umständen mehrere Entitäten eingetragen sein, wenngleich die Originalnamen paarweise verschieden sein müssen. Wird ein Name, der im Quelltext verwendet wird, in einer Entitätenumgebung nachgeschlagen und existieren mehrere Entitäten unter diesem Namen, wird zunächst geprüft, ob eine dieser Entitäten lokal definiert ist. Wenn das der Fall ist, wird die lokale Entität zurückgegeben. Andernfalls wird ein Fehler ausgegeben, dass der verwendete Name mehrdeutig ist.

7.2.1. Typkonstruktorenumgebung

In der Typkonstruktorenumgebung werden sämtliche im Programm vorkommenden Typkonstruktoren mit ihrer Arität eingetragen. Für Datentyp- oder Newtype-Konstruktoren werden dabei zusätzlich die sichtbaren Datenkonstruktoren, für Typsynonyme dagegen der Typausdruck, für den das Typsynonym steht, gespeichert.

7.2.2. Werteumgebung

In der Werteumgebung werden alle lokalen und importierten Funktionen sowie Datenkonstruktoren zusammen mit ihrer Stelligkeit und ihrem Typschema abgelegt. Die Typen sind dabei vollständig expandiert, d.h., sämtliche Typsynonyme wurden aufgelöst.

7.2.3. Präzedenzenumgebung

Die Präzedenzenumgebung enthält für jeden Infix-Operator dessen Präzedenz und Assoziativität, wobei nur Einträge für Operatoren existieren, für die auch eine Infix-Deklaration angegeben wurde. Für alle anderen Operatoren wird eine Standardpräzedenz und -assoziativität angenommen.

7.2.4. Schnittstellenumgebung

In der Schnittstellenumgebung wird zu jedem importierten Modul unter dessen Namen die zugehörige Schnittstelle (s. Abschnitt 7.4) gespeichert.

Beispiel 7.2.:
Gegeben seien die folgenden Module.

```
module M1 where
  f :: a
  f = ...

  g :: a -> a
  g x = ...
```

```
module M2 where
  f :: a -> a
  f x = ...

  g :: a -> a -> a
  g = ...

  h :: a -> a
  h x = ...
```

```
module M3 where
  import M1 as N1
  import M2

  f :: a -> a -> a
  f x y = ...

  i :: a
  i = ...
```

Dann ergibt sich beim Kompilieren des Moduls `M3` *die in Abbildung 7.2 dargestellte Werteumgebung, an der sich die zuvor beschriebene Mehrdeutigkeitssemantik nachvollziehen lässt. Dazu betrachten wir die folgenden beiden Beispiele.*

- *Wird in* `M3` *der Name* `f` *verwendet, so verweist er auf die Funktion aus demselben Modul, da lokal definierte Entitäten eine höhere Priorität als importierte besitzen.*
- *Die Verwendung des Namens* `g` *ist mehrdeutig, da unter ihm zwei importierte, aber keine lokal definierten Entitäten abgelegt sind.*

Alle anderen Namen sind hingegen eindeutig, da unter ihnen jeweils nur eine Entität eingetragen ist.

Name	Originalname	Stelligkeit	Typschema	Quelle
f	M3.f	2	∀a.a → a → a	Lokal
	M1.f	0	∀a.a	Importiert
	M2.f	1	∀a.a → a	Importiert
g	M1.g	1	∀a.a → a	Importiert
	M2.g	0	∀a.a → a → a	Importiert
h	M2.h	1	∀a.a → a	Importiert
i	M3.i	0	∀a.a	Lokal
N1.f	M1.f	0	∀a.a	Importiert
M2.f	M2.f	1	∀a.a → a	Importiert
M3.f	M3.f	2	∀a.a → a → a	Lokal
N1.g	M1.g	1	∀a.a → a	Importiert
M2.g	M2.g	0	∀a.a → a → a	Importiert
M2.h	M2.h	1	∀a.a → a	Importiert
M3.i	M3.i	0	∀a.a	Lokal

Abbildung 7.2.: Beispiel für eine Werteumgebung

7.3. Kompilierphasen

Der Kompiliervorgang des Front-Ends ist in mehrere, aufeinander aufbauende Phasen aufgeteilt, die sich wiederum in drei Abschnitte gliedern lassen: die Erkennung, die Prüfungen und die Transformationen.

Die Erkennung resultiert in einem abstrakten Syntaxbaum, der die Arbeitsgrundlage für alle nachfolgenden Phasen bildet. Im Anschluss daran stellen die Prüfungen die syntaktische Korrektheit und Wohlgetyptheit des Quellprogramms sicher. Dabei wird versucht, das zu kompilierende Modul möglichst wenigen Umformungen zu unterziehen, sodass sein Originalzustand weitestgehend erhalten bleibt. Das ist vor allem für die Erzeugung anderer Ausgaben wie der AbstractCurry-Darstellung wichtig, die erst nach den Prüfungen erfolgt. Die abschließenden Transformationen formen das Quellprogramm dagegen auf verschiedene Arten und Weisen um und haben die Generierung von FlatCurry-Code zum Ziel.

Der gesamte Kompilierverlauf ist in Abbildung 7.3 zu sehen. Sowohl in der genannten Abbildung als auch im restlichen Teil der Arbeit nutzen wir die englischen Originalbezeichnungen der Phasen. Wir werden die Bezeichnungen der Phasen im weiteren Verlauf als Eigennamen verwenden und als solche zur Kennzeichnung *kursiv* schreiben.

7.3.1. Lexer

Die erste Phase des Kompiliervorgangs ist die lexikographische Analyse, welche durch den *Lexer* durchgeführt wird und den Quellcode in logisch zusammengehörige Einheiten, sogenannte *Tokens*, zerlegt.

7.3.2. Parser

Der *Parser* konstruiert auf Basis einer Grammatik aus der vom *Lexer* generierten Tokenfolge den abstrakten Syntaxbaum.

7.3.3. Kind Check

Der *Kind Check* prüft, ob alle Typkonstruktoren im Quelltext vollständig angewandt wurden. Da das ursprüngliche Typsystem von Curry (s. Kapitel 5) noch keine Typen höherer Ordnung, also solche von einer anderen Sorte als $*$, unterstützt, muss nur sichergestellt werden, dass die Anzahl der Argumente bei einer Typkonstruktion genau der Arität entspricht, die für den beteiligten Typkonstruktor in der Typkonstruktorenumgebung eingetragen ist.

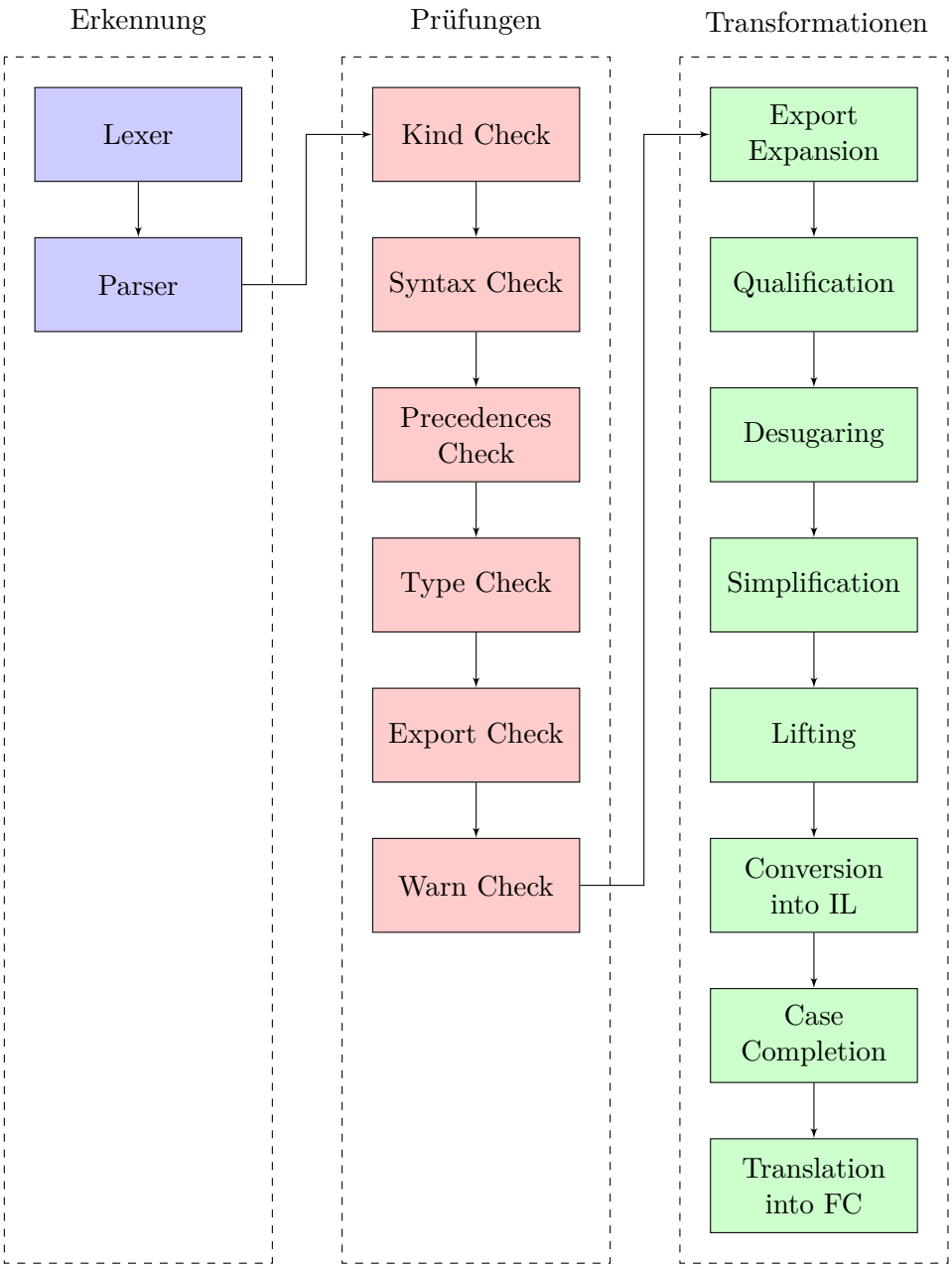

Abbildung 7.3.: Ursprünglicher Kompilierverlauf

Beispiel 7.3.:
Es sei der folgende Datentyp gegeben.

```
data T a b = ...
```

Dann sind die Signaturen

```
... -> T a -> ...
... -> T a b c -> ...
```

nicht erlaubt. Im ersten Fall werden dem Typkonstruktor `T` *zu wenige, im zweiten Fall zu viele Argumente übergeben. Korrekt ist hier nur ein Typausdruck der Form* `T a b`*, da* `T` *zweistellig ist.*

Weiterhin nimmt der *Kind Check* eine Unterscheidung zwischen nullstelligen Typkonstruktoren und -variablen vor. In Haskell ist diese Unterscheidung auf syntaktischer Ebene möglich, da Typkonstruktoren immer mit einem Großbuchstaben beginnen müssen, während für Typvariablen genau das Gegenteil der Fall ist. Da in Curry keine solchen Beschränkungen existieren (s. Unterabschnitt A.2.2 des Anhangs), muss für jedes Vorkommen eines nicht weiter applizierten Bezeichners in einem Typausdruck geprüft werden, ob es sich um einen Typkonstruktor oder eine Typvariable handelt. Zu diesem Zweck wird der jeweilige Bezeichner einfach in der Typkonstruktorenumgebung nachgeschlagen. Gleichzeitig wird auch sichergestellt, dass keiner der Bezeichner, die in den Typausdrücken verwendetet werden, mehrdeutig ist.

Beispiel 7.4.:
Es sei der folgende Code gegeben.

```
data T = ...

f :: S -> T -> Bool
f x y = True
```

Dann verweist der Bezeichner `T` *in der Typsignatur von* `f` *auf den entsprechenden Typkonstruktor, während* `S` *eine Typvariable ist, da kein gleichnamiger Typkonstruktor existiert.*

Ebenfalls Teil des *Kind Checks* ist die Untersuchung auf mehrfache Definitionen von Typkonstruktoren sowie der Test, ob die auf der rechten Seite einer Typdeklaration verwendeten Typkonstruktoren und -variablen definiert sind.

7.3.4. Syntax Check

Der *Syntax Check* überprüft, ob alle Funktionen und Variablen, welche im Quelltext benutzt werden, definiert sind, und stellt sicher, dass diese nicht mehrdeutig sind.

Da auch für Datenkonstruktoren und Variablen gilt, dass diese sowohl groß- als auch kleingeschrieben werden dürfen, nimmt der *Syntax Check* – ähnlich wie der *Kind Check* für Typkonstruktoren und -variablen – eine Unterscheidung zwischen nullstelligen Datenkonstruktoren und Variablen vor. Das Vorgehen hierbei ist analog zum *Kind Check*:

Der Bezeichner wird in der Werteumgebung nachgeschlagen, um festzustellen, ob es sich um einen Datenkonstruktor oder eine Variable handelt.

Beispiel 7.5.:
Es sei der folgende Code gegeben.

```
data T = S

f S U = ...
```

In der Funktion `f` *ist* `S` *dann ein Muster für den nullstelligen Typkonstruktor desselben Namens und* `U` *eine Variable.*

Darüber hinaus nimmt der *Syntax Check* eine Umbenennung vor, bei der alle Funktionen und Variablen, welche nicht auf oberster Ebene definiert sind, mit einer natürlichen Zahl als Schlüssel versehen werden, der gleichzeitig auch den Sichtbarkeitsbereich identifiziert. Top-Level-Deklarationen werden nicht umbenannt bzw. erhalten den Schlüssel 0. Nach der Umbenennung teilen sich alle Variablen, die im selben Geltungsbereich definiert wurden, einen Schlüssel, wodurch Konflikte in den Definitionen einfach erkannt werden können. Die Umbenennung erlaubt es dem Compiler zudem, eine flache Werteumgebung vorzuhalten, und ermöglicht das Anheben von lokalen Definitionen in späteren Phasen (s. beispielsweise Unterabschnitt 7.3.13).

Beispiel 7.6.:
Der Quellcode

```
f x = x
  where g x = x
        h y = x
```

wird in

```
f0 x1 = x1
  where g2 x3 = x3
        h2 y5 = x1
```

umbenannt. Die hinzugefügten Schlüssel, welche im eigentlichen Quellcode nicht sichtbar sind, sind hier durch tiefgestellte Zahlen dargestellt.

Der *Syntax Check* prüft außerdem, ob alle Gleichungen einer Funktion dieselbe Stelligkeit, d.h. die gleiche Anzahl an Parametern, besitzen. Im Anschluss daran werden alle Gleichungen einer Funktion zusammengeführt, denn, anders als in Haskell, dürfen diese in Curry nämlich über den Quelltext verteilt sein.

Zu guter Letzt ist es auch Teil des *Syntax Checks*, im Modulkopf spezifizierte Spracherweiterungen zu überprüfen und ggf. zu aktivieren, indem sie der Compiler-Umgebung hinzugefügt werden.

7.3.5. Precedences Check

Da dem Parser die Präzedenzen von Infix-Operatoren nicht bekannt sind, wird der Quellcode zunächst so geparst, als ob alle Operatoren rechtsassoziativ binden und die gleiche Präzedenz haben würden. Der abstrakte Syntaxbaum muss dann unter Berücksichtigung der im Quellcode angegebenen Assoziativitäten und Präzedenzen umgeordnet werden. Zu diesem Zweck sucht der *Precedences Check* alle Infix-Deklarationen des Moduls und trägt die darin angegebenen Präzedenzen unter dem betroffenen Operator in die Präzedenzenumgebung ein.

Beispiel 7.7.:
Der Ausdruck `3 * 2 + 1` *wird durch den Parser zunächst als* `3 * (2 + 1)` *geparst. Während des Precedences Checks findet dann eine Umordnung unter Berücksichtigung der Präzedenzen der beiden beteiligten Operatoren* `*` *und* `+` *statt und der Ausdruck wird in* `(3 * 2) + 1` *umgewandelt.*

7.3.6. Type Check

Der *Type Check* baut die Typkonstruktorenumgebung auf, indem er ihr alle im Modul definierten Typen hinzufügt. Dabei werden alle Typsynonyme expandiert, weswegen diese nicht rekursiv definiert sein dürfen, was ebenfalls durch diese Phase sichergestellt wird.

Schließlich wird die Typprüfung des Quellcodes durchgeführt, deren Ablauf wir nachfolgend umreißen. Die hierbei verwendete, interne Typrepräsentation ist identisch zu der in Abschnitt 5.1 vorgestellten.

1. Als erstes werden die Typen aller Datenkonstruktoren und Feldselektoren für Records in die Werteumgebung eingetragen, damit diese für die Typermittlung zur Verfügung stehen.

2. Der nächste Schritt besteht in der Aufteilung des Codes in Deklarationsgruppen, in der informell gesprochen alle Funktionen voneinander abhängig sind. Auf das genaue Verfahren hierfür wird in Einschub 7.8 eingegangen. Die Deklarationsgruppen werden außerdem topologisch sortiert.

3. Schließlich wird auf Basis des in Abschnitt 5.3 präsentierten Algorithmus $\mathcal{W}$ in topologischer Reihenfolge die Typermittlung für alle Deklarationsgruppen durchgeführt. Dabei fungiert die Werteumgebung als Typannahme. Der Algorithmus $\mathcal{W}$ kann leicht auf andere Konstrukte des abstrakten Syntaxbaums erweitert werden. So lassen sich beispielsweise Where-Klauseln auf Let-Ausdrücke und Sections sowie Infix-Anwendungen von Operatoren auf die Applikation zurückführen.

4. Zum Schluss erfolgt das Eintragen der durch den Algorithmus $\mathcal{W}$ ermittelten Typschemata in die Werteumgebung, sodass diese für die Typermittlung in anderen Deklarationsgruppen verfügbar sind.

Im Quelltext eventuell vorhandene Typsignaturen in einer Deklarationsgruppe werden berücksichtigt, indem sie der Typannahme im Vorfeld hinzugefügt werden. Sollte der inferierte Typ am Ende nicht mit dem angegebenen übereinstimmen, ist das gleichbedeutend damit, dass die Typsignatur zu allgemein gewesen ist, da der inferierte Typ nur spezieller als der angegebene Typ sein kann.

Einschub 7.8. (Ermittlung der Deklarationsgruppen)**:**
Für die Ermittlung der Deklarationsgruppen wird ein gerichteter Graph betrachtet, dessen Knoten einzelne Entitäten sind. Dabei enthält der Graph eine gerichtete Kante von einem Knoten k zu einem Knoten j, wenn j von k genutzt wird. Die Deklarationsgruppen entsprechen dann genau den starken Zusammenhangskomponenten *des Graphen. Eine starke Zusammenhangskomponente ist dabei eine maximale Menge von Knoten K, sodass für alle Knoten $k, j \in K$ gilt, dass ein gerichteter Pfad von k nach j existiert. Ein Beispiel für die Ermittlung von Deklarationsgruppen mit Funktionen als Entitäten ist in Abbildung 7.4 dargestellt.*

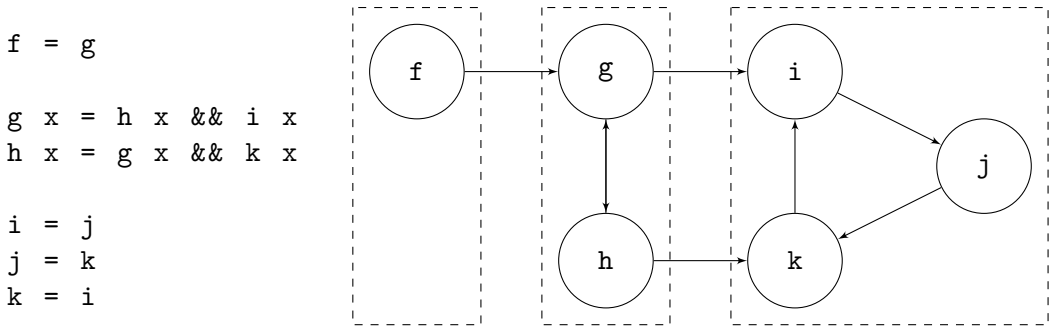

Abbildung 7.4.: Beispiel für die Ermittlung von Deklarationsgruppen

7.3.7. Export Check

Im *Export Check* wird die Exportspezifikation des Moduls geprüft. Das umfasst zum einen die Prüfung, ob die dort angegebenen Entitäten definiert bzw. im Falle eines Reexports sichtbar sind, zum anderen, ob keine Mehrdeutigkeiten bei den exportierten Entitäten vorliegen. Letzteres ist der Fall, wenn zwei voneinander verschiedene, exportierte Entitäten denselben unqualifizierten Namen besitzen. Beides kann durch Nachschlagen in den Entitätenumgebungen ermittelt werden.

Beispiel 7.9.:
Die Exportspezifikation des Moduls

```
module M1 (Bool, Prelude.Bool) where
  data Bool = False | True
```

ist ungültig, da der unqualifizierte Bezeichner `Bool` *sowohl auf* `Prelude.Bool` *als auch auf* `M1.Bool` *verweisen kann. Wäre diese Exportspezifikation erlaubt, könnte die Importspezifikation des Moduls*

```
module M2 where
  import M1 (Bool)
```

nicht eindeutig aufgelöst werden.

7.3.8. Warn Check

Der *Warn Check* untersucht das Modul auf potentiell inkorrekten Code und gibt entsprechende Warnmeldungen aus. Dabei werden die folgenden Punkte berücksichtigt:

- Mehrfachimporte von Modulen oder Entitäten,
- überlappende Modulaliase,
- unreferenzierte Variablen,
- überdeckte Variablen (*Shadowing*),
- überlappende Case-Alternativen,
- unvollständiges *Pattern Matching*,
- nicht-adjazente Funktionsgleichungen sowie
- fehlende Typsignaturen für Top-Level-Funktionen.

Die Warnungen können für jeden der obigen Punkte über Optionen für das Front-End einzeln an- oder ausgeschaltet werden.

7.3.9. Export Expansion

In der *Export Expansion* wird die Exportspezifikation durch die folgende Transformation vereinheitlicht.

Transformation 7.10. (Exportspezifikation)**:**
Exporte der Form T`(..)` *werden in* T`(`C_1`, ..., `C_n`, `l_1`, ..., `l_m`)` *mit* $n \geq$ *und* $m \geq 0$ *transformiert, wobei* T *ein Typkonstruktor,* $C_1, \ldots, C_n$ *die Datenkonstruktoren des Typs* T *und* $l_1, \ldots, l_m$ *die zugehörigen Felder sind. Eine Angabe der Form* `module` M *wird durch eine Liste aller vom Modul* M *exportierten und gleichzeitig im aktuellen Modul unqualifiziert importierten Entitäten ersetzt.*

Um Typkonstruktoren von Funktionen unterscheiden zu können, werden erstere darüber hinaus in die äquivalente Darstellung T`()` übersetzt.

7.3.10. Qualification

In der *Qualification*-Phase werden sämtliche Bezeichner, die auf Funktionen, Daten- oder Typkonstruktoren verweisen, durch ihre Originalnamen ersetzt. Nur lokale Funktionen und Variablen sowie Funktionsargumente bleiben unverändert. Dies vereinfacht die nachfolgenden Transformationen erheblich, da in diesen so weder Modulaliase noch Mehrdeutigkeiten berücksichtigt werden müssen.

Beispiel 7.11.:
Es seien die folgenden beiden Module gegeben.

```
module M1 where    module M2 where
  f x = x            import M1 as N1

                     f = N1.f
                     g = f
```

Durch die Qualifizierung werden die Funktionsdeklarationen des Moduls `M2` *in*

```
f = M1.f
g = M2.f
```

geändert.

Auch die Compiler-Umgebung wird vollständig qualifiziert, sodass alle Entitäten, die aus anderen Modulen importiert wurden, nur noch unter ihrem Originalnamen zu finden sind. Konkret sind davon die Typkonstruktoren-, die Werte- und die Präzedenzenumgebung, also sämtliche Entitätenumgebungen, betroffen.

7.3.11. Desugaring

Im *Desugaring* wird die Komplexität des abstrakten Syntaxbaums reduziert, indem sämtlicher sogenannter „syntaktischer Zucker“ entfernt und durch äquivalente Ausdrücke ersetzt wird. Der so entstehende Code ist zwar schwieriger zu lesen, die Struktur selbst aber ist einfacher, wodurch spätere Transformationsphasen simpler gehalten werden können. Eine Auflistung aller vorgenommenen Umformungen ist in den Abbildungen 7.5, 7.6 und 7.7 zu sehen.

Darüber hinaus wird auch die explizite Klammerung von Mustern und Ausdrücken entfernt, da diese bereits im abstrakten Syntaxbaum berücksichtigt ist. Weiterhin werden sämtliche Typsynonyme aufgelöst, wodurch deren Typdeklarationen wegfallen können.

7.3.12. Simplification

Die *Simplification* nimmt weitere Vereinfachungen des Quellcodes vor. Unter anderem werden lokale Musterdeklarationen durch Variablendeklarationen unter Verwendung von neu eingeführten Selektoren ersetzt.

Konstrukt	**Ausdruck/Muster**	**Umformung**
Left-Section	`(`e $\circ$`)`	`(`$\circ$`)` e
Right-Section	`(`$\circ$ e`)`	`flip (`$\circ$`)` e
Infix-Applikation Infix-Muster	e $\circ$ f	`(`$\circ$`)` e f

Arithmetische Sequenz	`[`a `..]`
	`[`a`,` b `..]`
	`[`a `..` b`]`
	`[`a`,` b `..` c`]`
		`enumFrom` a
		`enumFromThen` a b
		`enumFromTo` a b
		`enumFromThenTo` a b c

Tupel
Tupelmuster
	`(`a`,` b`)`
	`(`a`,` b`,` c`)`
	...
		`(,)` a b
		`(,,)` a b c
		...

Liste
Listenmuster
	`[`a`,` b`, ...,` c`]`	`(:)` a
		`((:)` b
		...
		`((:)` c `[])`
		...
		`)`

List-Comprehension	`[`e `|` `]`
	
	`[`e `|` t `<-` l`,` qs`]`
	
	`[`e `|` b`,` qs`]`
	`[`e `|` `let` ds`,` qs`]`
		e
		`foldr f [] `l
		`  where`
		`  f x xs = case x of`
		`  `t `-> [`e `|` qs`] ++ xs`
		`  _ -> xs`
		`if` b `then [`e `|` qs`] else []`
		`let` ds `in [`e `|` qs`]`

Konditional	`if` e `then` f `else` g	`case` e `of` `  True ->` f `  False ->` g

Where-Klausel	e `where`
	`  `f `=`
	`  `g `=`
	`  ...`
		`let` f `=`
		`  `g `=`
		`  ...`
		`in` e

Record-Konstruktion	C `{` l `=` e`, ... }`	C e ...
Record-Update	e `{` l `=` f`, ... }`	`fcase` e `of` `  `C `... _ ... ->` C ... f ...

Abbildung 7.5.: Umformungen für Ausdrücke und Muster (Teil 1)

Konstrukt	Ausdruck/Muster	Umformung
Do-Notation	`do` e	e

	`do`
	  e
	  Anweisungen
	  f
		e `>> (do`
		  Anweisungen
		  f`)`

	`do`
	  t `<-` e
	  Anweisungen
	  f
		e `>>= \`t `-> (do`
		  Anweisungen
		  f`)`

	`do`
	  `let` ds
	  Anweisungen
	  e
		`let` ds `in (do`
		  Anweisungen
		  e`)`

Abbildung 7.6.: Umformungen für Ausdrücke und Muster (Teil 2)

Konstrukt	Deklaration	Umformung

Datentypdeklaration	`data` T `=` C `{` l `::` t
	  `,` ... `}`
	  `|` ...
		`data` T `=` C t `|` ...
		
		l `::` T `->` t
		l `(`C `a) = a`
		...

Newtype-Deklaration	`newtype` T `=` C `{` l `::` t `}`	`newtype` T `=` C τ l `::` T `->` t l `(`C `a) = a`

Abbildung 7.7.: Umformungen für Typdeklarationen

Beispiel 7.12.:
Der Code

```
f x = let (y, z) = x
      in y || z
```

wird in

```
f x = let sel_y (y, _) = y
          y = sel_y x
          sel_z (_, z) = z
          z = sel_z x
      in y || z
```

transformiert.

Zudem nimmt diese Phase auch einige rudimentäre Optimierungen vor. So werden ungenutzte Deklarationen aus dem Quellcode entfernt und unter bestimmten Voraussetzungen werden Funktionsrümpfe von verwendeten Funktionen direkt eingesetzt. Letzteres wird auch als *Inlining* bezeichnet und wird nur dann durchgeführt, wenn die verwendete Funktion ausschließlich aus einer Variablen, einem Literal oder einer nullstelligen Funktion besteht.

Beispiel 7.13.:
Der Code

```
f x = x + y
  where y = 2

g = True
```

wird in

```
f x = x + 2
```

transformiert.

7.3.13. Lifting

Durch das *Lifting* werden alle lokalen Funktionsdeklarationen auf Top-Level angehoben. Für alle freien Variablen in den Funktionsdeklarationen werden neue Parameter ergänzt und die Aufrufe der jeweiligen Funktionen werden so angepasst, dass die zuvor frei vorkommenden Variablen als zusätzliche Argumente übergeben werden [Joh85].

Beispiel 7.14.:
Der Code

```
f x = g x
  where g y = (x, y)
```

wird in

```
f x = g_lifted x 1
g_lifted x y = (x, y)
```

transformiert.

7.3.14. Conversion into Intermediate Language

In der *Conversion into Intermediate Language* wird der abstrakte Syntaxbaum in eine vereinfachte Zwischensprache (*intermediate language, IL)* konvertiert. Diese ist der ohnehin bereits erheblich vereinfachten Struktur des abstrakten Syntaxbaums sehr ähnlich, sodass die meisten Konstrukte ohne Änderungen direkt übertragen werden können. Jedoch wird das *Pattern Matching* der Funktionen in Case-Ausdrücke überführt und letztere darüber hinaus vereinfacht. Die Funktionen werden dabei transformiert, sodass sie nur noch Variablen als Parameter besitzen und aus einer einzigen Gleichung bestehen. Die Muster in den Alternativen der Case-Ausdrücke sind nach deren Vereinfachung nur noch aus Konstruktoren-, Literal- und Variablenmustern zusammengesetzt.

Beispiel 7.15.:
Der Code

```
f (Just [x]) = x
f Nothing    = False
```

wird in

```
f y = case y of
        Just z  -> case z of
                     x:xs -> case xs of
                               [] -> x
        Nothing -> False
```

transformiert.

Außerdem werden überlappende Muster (s. Abschnitt 2.1) in expliziten Nichtdeterminismus übersetzt. Für diesen gibt es in der Darstellung der Zwischensprache ein separates Konstrukt. Als Folge dessen besteht jede Funktionsdefinition – ob deterministisch oder nichtdeterministisch – in der Zwischensprache nur noch aus einer einzigen Gleichung.

Beispiel 7.16.:
Die aus zwei Gleichungen bestehende Funktionsdefinition

```
coin = 0
coin = 1
```

wird in der Zwischensprache als

```
coin = 0 | 1
```

dargestellt, wobei `|` *die nichtdeterministische Auswahl repräsentiert.*

7.3.15. Case Completion

Da FlatCurry nur Konstruktoren- und Literal-, aber keine Variablenmuster unterstützt, müssen letztere geeignet umgeformt werden. Dies geschieht in der *Case Completion*. Dabei wird jede Case-Alternative mit einem Variablenmuster durch neue Zweige mit Konstruktorenmustern ersetzt, wobei die neuen Case-Zweige genau jene Datenkonstruktoren abdecken, auf die das Variablenmuster zuvor zugetroffen hat. Die fehlenden Datenkonstruktoren werden mithilfe der Schnittstellenumgebung ermittelt, da dort stets alle zu einem Typ gehörigen Datenkonstruktoren zu finden sind.

Beispiel 7.17.:
Sei der folgende Code gegeben.

```
data T = C | D | E

f x = case x of
  C -> True
  _ -> False
```

Durch die Vervollständigung von Case-Ausdrücken wird die Funktionsdeklaration wie folgt abgeändert.

```
f x = let y = False
      in case x of
           C -> True
           D -> y
           E -> y
```

7.3.16. Translation into FlatCurry

Als letzter Schritt wird das Zwischensprachenmodul nach FlatCurry übersetzt. Aufgrund der starken Ähnlichkeit zwischen der Zwischensprache und FlatCurry erfolgen in der *Translation into FlatCurry* keine relevanten Umformungen mehr. Es werden lediglich einige zuvor bereits entfernte Elemente wie Typsynonyme oder Infix-Deklarationen wieder hinzugefügt, um den auf FlatCurry aufbauenden Tools zusätzliche Informationen über die ursprüngliche Struktur des Quellcodes zuteil werden zu lassen.

7.4. Modulsystem

Curry besitzt ein Haskell sehr ähnliches Modulsystem, das die Kapselung von Funktionalitäten und die einfache Wiederverwendung von Code auf Basis von Schnittstellen erlaubt.

7.4.1. Import

Der Import wird vor Beginn der Prüfungsphasen durchgeführt (s. Abbildung 7.8). Für jede Import-Deklaration im Modul wird zunächst die zugehörige Schnittstelle geladen und unter ihrem Modulnamen in die Schnittstellenumgebung eingetragen. Anschließend werden die in der Schnittstelle enthaltenen Entitäten gemäß der Importspezifikation den unterschiedlichen Unterumgebungen der Compiler-Umgebung hinzugefügt, sodass sie für alle folgenden Kompilierphasen verfügbar sind.

Sofern nicht durch das Schlüsselwort `qualified` ein auschließlich qualifizierter Import angezeigt wird, erfolgt immer sowohl ein unqualifizierter als auch ein qualifizierter Import. Beim unqualifizierten Import werden die Entitäten unter ihrem unqualifizierten Namen in die jeweiligen Entitätenumgebungen eingetragen, bei dem qualifizierten Import dagegen unter ihrem qualifizierten Namen.

7.4.2. Export

Beim Export, der nach der *Qualification*-Phase stattfindet (s. Abbildung 7.8), wird die Schnittstelle für das zu kompilierende Modul erstellt, indem ihr jede in der Exportspezifikation angegebene Entität hinzugefügt wird. Wenn für eine der exportierten Entitäten eine Präzedenz angegeben wurde, wird diese automatisch mit exportiert.

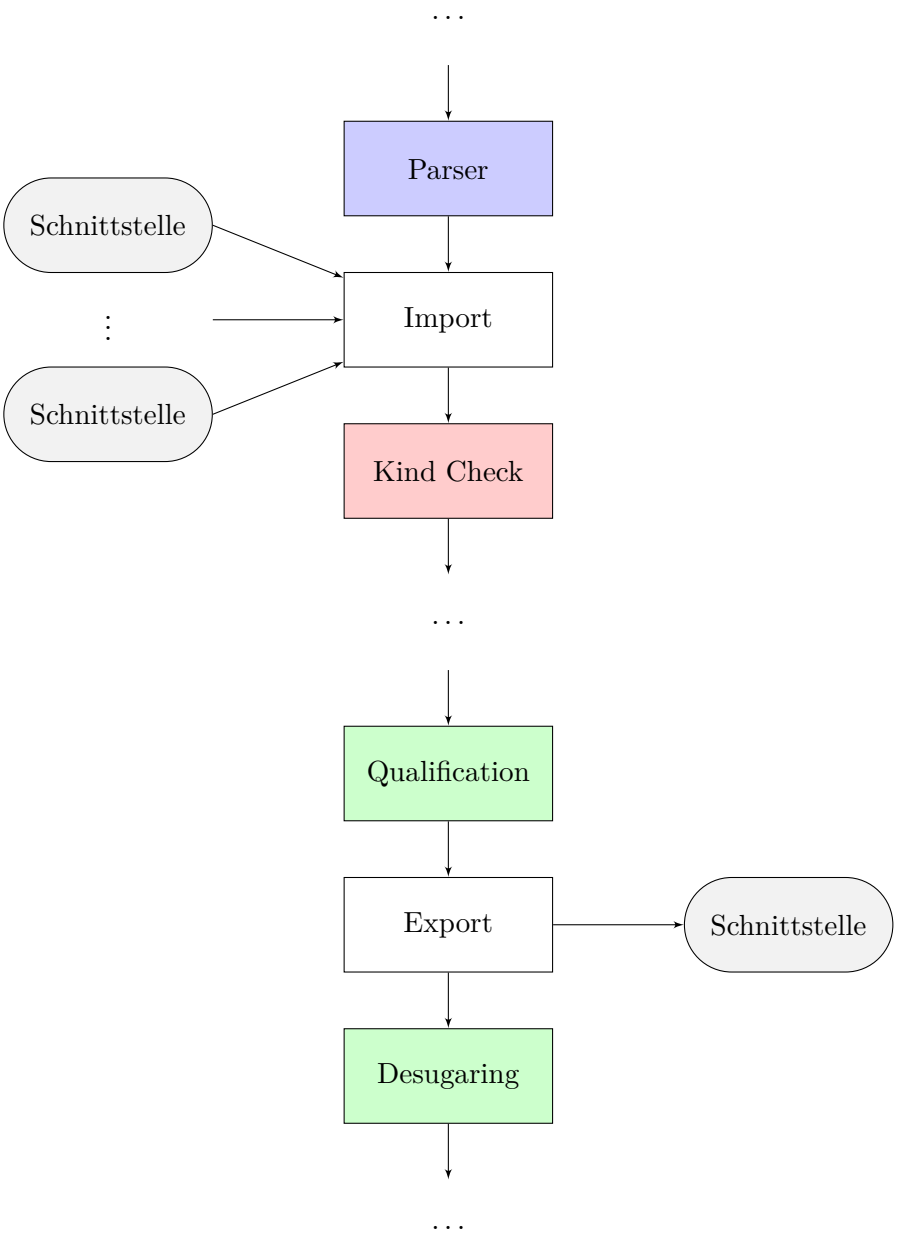

Abbildung 7.8.: Im- und Export im Kompilierverlauf

8. Anpassungen des Front-Ends

In diesem Kapitel werden die am Front-End vorgenommenen Änderungen erläutert. Deren Schilderung beschränkt sich zumeist auf eine allgemeine Beschreibung der Anpassungen, weswegen für weitergehende Details in Hinblick auf die Implementierung auf den Quelltext[1] sowie die darin enthaltenen Kommentare verwiesen sei.

Die Idee der Umsetzung ist es, die Typklassenelemente über die Prüfungsphasen hinweg beizubehalten und sie erst in den Transformationsphasen unter Verwendung des in Kapitel 4 beschriebenen Wörterbuchansatzes zu entfernen. Dieses Vorgehen wahrt auch die Trennung von Prüfungs- und Transformationsphasen und deckt sich somit mit der Leitlinie, das Quellprogramm bis zu Beginn der Transformationen möglichst unverändert zu lassen. So wird es letztendlich auch ermöglicht, die Typklassenelemente in die AbstractCurry- und HTML-Darstellung einfließen zu lassen.

8.1. Syntax

Da die Syntax von Curry bereits an die Haskells angelehnt ist, wurde diese auch für die neuen Typklassenelemente in Curry adaptiert. Neu hinzugekommen sind Default-, Klassen- und Instanzdeklarationen sowie Kontexte und damit einhergehend beschränkte Typen. Außerdem wurde die Typsyntax dahingehend verallgemeinert, dass die für die Unterstützung von Typkonstruktorklassen erforderliche Typvariablenapplikation ausgedrückt werden kann. Eine vollständige Beschreibung der modifizierten Syntax findet sich in Abschnitt A.4 des Anhangs.

8.2. Compiler-Umgebung

Die Anpassung der Compiler-Umgebung schließt sowohl Änderungen der bestehenden als auch das Hinzufügen neuer Unterumgebungen ein.

8.2.1. Typkonstruktorenumgebung

Typklassen wurden der Typkonstruktorenumgebung als neue Entität hinzugefügt. Die Einordnung von Typklassen auf Ebene der Typkonstruktoren ist insofern sinnvoll, als dass sich Typkonstruktoren und Typklassen denselben Namensraum teilen. Durch die Verwendung derselben Umgebung für beide Arten von Entitäten können Namenskonflikte auf einfache Art und Weise festgestellt werden.

[1] Verfügbar unter `https://git.ps.informatik.uni-kiel.de/fte/curry-base` und `https://git.ps.informatik.uni-kiel.de/fte/curry-frontend`

Für jede Typklasse wird ihre Sorte sowie eine Liste aller sichtbaren Klassenmethoden gespeichert. Bei Typkonstruktoren wird anstelle der Arität nur noch die Sorte gespeichert, da die Arität sich ohnehin aus letzterer ergibt. Einzig und allein für Typsynonyme trifft dies nicht zu, wie das folgende Beispiel zeigt, weswegen für diese beide Informationen vorgehalten werden.

Beispiel 8.1.:
Es sei das folgende Typsynonym gegeben.

```
type T = []
```

Dann ist der Typkonstruktor `T` *von derselben Sorte wie* `[]`*, nämlich* $* \to *$*, aber null- und nicht einstellig.*

8.2.2. Werteumgebung

Wie bei Typklassen in der Typkonstruktorenumgebung ist es auch für Klassenmethoden sinnvoll, diese in der gleichen Umgebung wie herkömmliche Funktionen abzulegen, um Mehrfachdefinitionen schnell bemerken zu können.

Zusätzlich zu den bereits vorhandenen Informationen wird nun noch gespeichert, ob es sich bei einer Funktion um eine Klassenmethode handelt. Dies ist unter anderem für die korrekte Umsetzung der Wörterbuchtransformation (s. Unterabschnitt 8.5.13) sowie den Export (s. Unterabschnitt 8.7.2) relevant.

8.2.3. Klassenumgebung

Die neu hinzugekommene Klassenumgebung speichert zu einem Klassennamen die Liste der direkten Superklassen sowie alle assoziierten Klassenmethoden. Da sowohl für die Klassen selbst als auch für die Superklassen deren Originalnamen verwendet werden und somit keine Mehrdeutigkeiten entstehen können, wird keine Entitätenumgebung benötigt und die Umsetzung der Klassenumgebung als einfache Zuordnung reicht aus.

Die Listen der Superklassen und der assoziierten Methoden einer Klasse enthalten immer alle Klassen bzw. Methoden und beschränken sich nicht nur auf die jeweils sichtbaren. Das ist wichtig, um für die spätere Wörterbuchtransformation (s. Unterabschnitt 8.5.13) alle benötigten Informationen zur Verfügung zu haben.

8.2.4. Instanzenumgebung

Da Instanzen immer eindeutig über den involvierten Klassennamen und Typkonstruktor identifiziert werden können, sofern für beides der Originalname verwendet wird, ist die neue Instanzenumgebung, die Informationen über alle bekannten Instanzdefinitionen vorhält, ebenfalls als einfache Zuordnung realisiert.

Eben jene Paare von Klassennamen und Typkonstruktor werden auf ein Tripel abgebildet, das aus dem Modulnamen, in dem die Instanz ursprünglich definiert wurde, dem vollständig reduzierten Instanzkontext sowie einer Liste aller für die Instanz implementierten Klassenmethoden inklusive ihrer Stelligkeit besteht.

8.3. Abstrakter Syntaxbaum

Abgesehen davon, dass die Änderungen der Syntax (s. Abschnitt 8.1) im abstrakten Syntaxbaum abgebildet wurden, ist dieser um Typannotationen erweitert worden. Wie bereits in Abschnitt 4.4 festgehalten wurde, ist die Kenntnis der allgemeinen und speziellen Typen aller Funktionen für die Wörterbuchtransformation vonnöten. Durch das Ablegen dieser Informationen im abstrakten Syntaxbaum sind die benötigten Typinformationen nun direkt an Ort und Stelle verfügbar.

Die Elemente der linke Seite einer Funktion werden mit ihren allgemeinen Typen, die der rechten Seite dagegen mit ihren speziellen Typen annotiert. Zusätzlich wird die Funktion selbst mit ihrem allgemeinen Typen versehen, damit ihr Kontext bekannt ist. Abbildung 8.1 zeigt eine vereinfachte Darstellung des abstrakten Syntaxbaums für die Funktion `elem` aus Unterabschnitt 3.1.1.

8.4. Interne Typdarstellung

Die interne Typdarstellung, die vor allem für den *Type Check* relevant ist, wurde analog zu den in Kapitel 6 beschriebenen Erweiterungen des Typsystems modifiziert und um Konstruktoren sowie Prädikate erweitert.

Die in Unterabschnitt 3.2.5 angenommene, kanonische Darstellung für vollständig reduzierte Kontexte wird in der Implementierung erreicht, indem für die Repräsentation von Prädikatenmengen geordnete Mengen gewählt werden. Darüber hinaus werden sämtliche Typklassen und Typkonstruktoren ausschließlich mit ihrem Originalnamen referenziert. Die Verwendung der Originalnamen in der internen Darstellung garantiert zusammen mit der Mengensemantik, dass alle vorkommenden Constraints eindeutig sind, während die Ordnung auf den Constraints sicherstellt, dass deren Reihenfolge einheitlich ist und somit vernachlässigt werden kann.

8.5. Kompilierphasen

Während nahezu sämtliche Prüfungsphasen angepasst und sogar neue Phasen zu den Prüfungen hingezufügt werden mussten, ist der Ansatz bei den Transformationen gewesen, mit der Wörterbuchtransformation nur eine neue Phase einzufügen und diese möglichst früh in den Kompilierverlauf einzubauen. Auf diese Weise waren auch nur geringfügige Änderungen an den nachfolgenden Phasen notwendig.

Der vollständige, angepasste Kompilierverlauf ist in Abbildung 8.2 zu sehen, wobei neue Phasen stärker umrandet und modifizierte Phasen kursiv hervorgehoben sind.

8.5.1. Lexer

Der *Lexer* muss um die zusätzlichen Schlüsselwörter für die Typklassenelemente, also `class`, `default` und `instance`, sowie um den reservierten Operator `=>`, der als Trennelement zwischen Kontext und Typ in einem beschränkten Typ fungiert, erweitert werden.

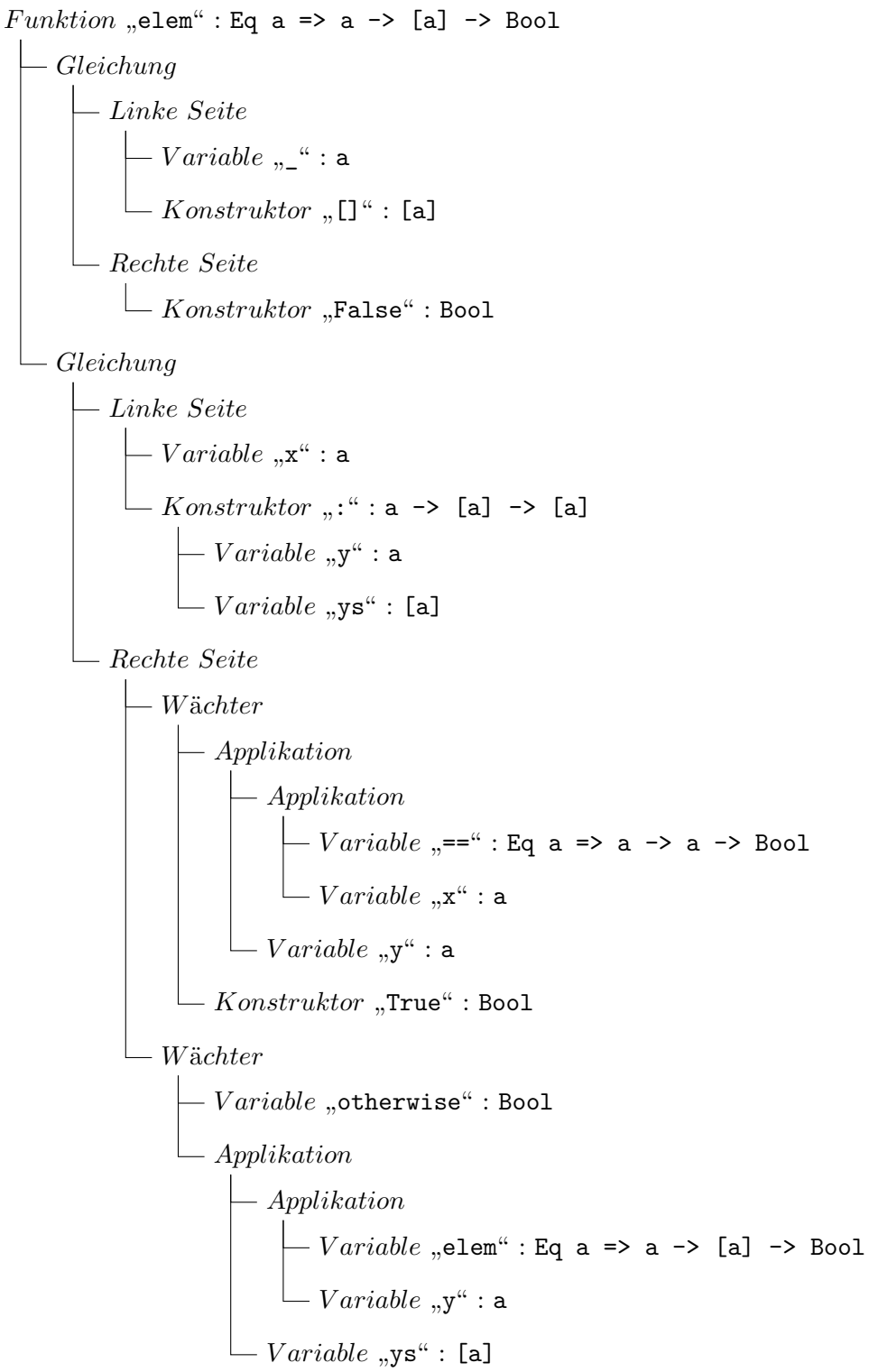

Abbildung 8.1.: Beispiel für einen annotierten abstrakten Syntaxbaum

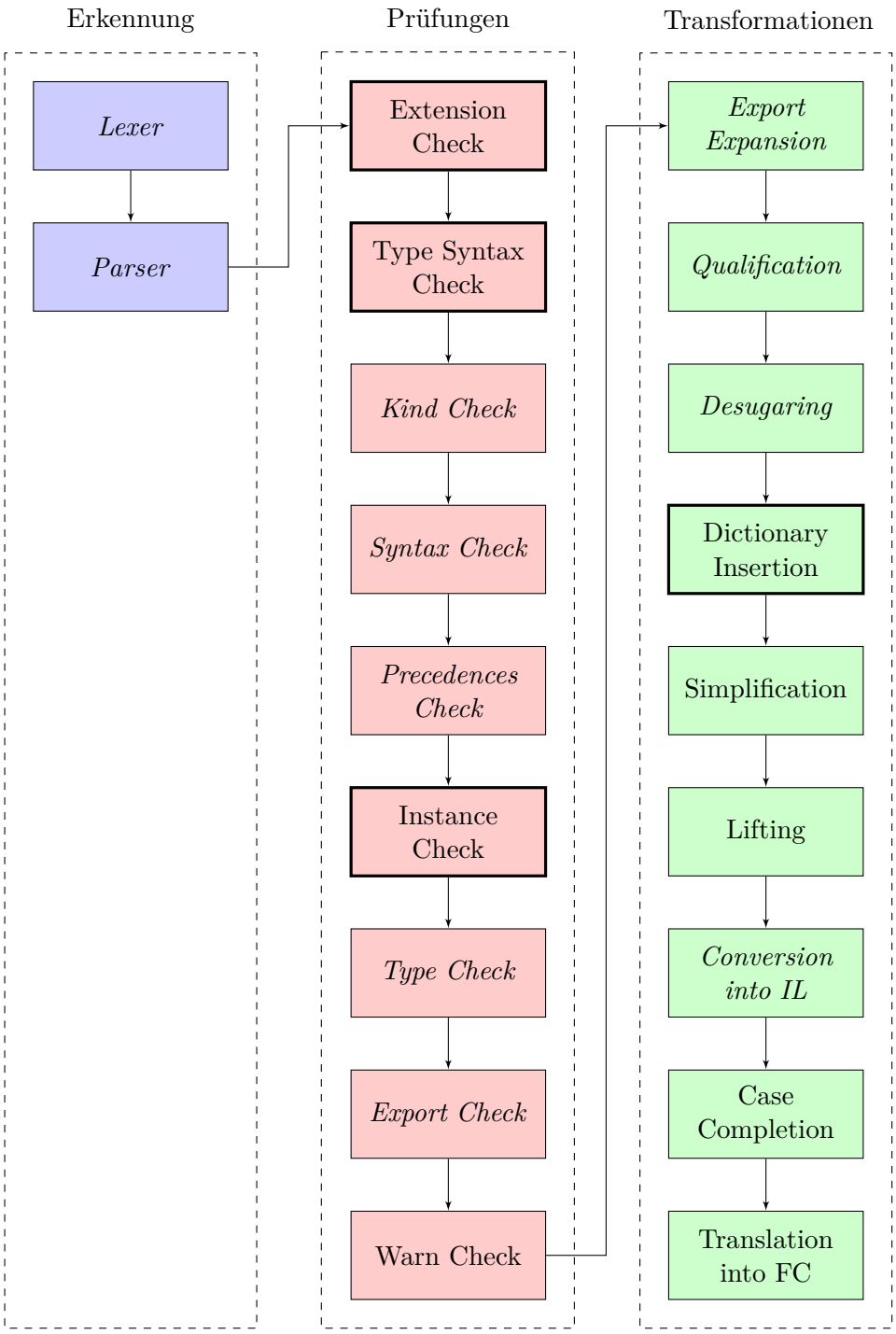

Abbildung 8.2.: Angepasster Kompilierverlauf mit neuen Kompilierphasen

8.5.2. Parser

Der *Parser* setzt die Änderungen an der Modulsyntax um. Die vollständige Grammatik hierfür findet sich in Abschnitt A.4 des Anhangs. Der generierte abstrakte Syntaxbaum bleibt zunächst unannotiert, da die Typen der Funktionen und Ausdrücke noch unbekannt sind.

8.5.3. Extension Check

Für den Fall, dass Erweiterungen der Typsyntax vorgesehen sind, müssen Spracherweiterungen bereits im *Type Syntax Check* (s. nächsten Unterabschnitt) bekannt sein. Es ist daher nicht länger ausreichend, sie erst im *Syntax Check* zu berücksichtigen. Aus diesem Grund wurde mit dem *Extension Check* eine neue Phase eingefügt, welche die Prüfung und Aktivierung von im Modulkopf spezifizierten Spracherweiterungen vornimmt.

8.5.4. Type Syntax Check

Einige Teile des *Kind Checks* wurden in eine separate Phase, den *Type Syntax Check*, ausgegliedert. So erfolgt die Differenzierung zwischen nullstelligen Typkonstruktoren und Typvariablen, die Voraussetzung für den nachfolgenden, modifizierten *Kind Check* ist, nun in dieser Phase. Auch die Prüfung, ob alle auf der rechten Seite einer Typdeklaration verwendeten Typkonstruktoren und Typvariablen definiert sind, ist jetzt Teil des neuen *Type Syntax Checks*. Zusätzlich wird die Einhaltung aller syntaktischen Einschränkungen für Typklassenelemente (s. Abschnitt 3.2) überprüft.

8.5.5. Kind Check

Der Aufbau der Typkonstruktorenumgebung, der vorher im *Type Check* stattfand, wurde in den *Kind Check* verlagert, da dem neuen *Instance Check* (s. Unterabschnitt 8.5.8) die Typkonstruktoren und Typklassen bereits bekannt sein müssen. Dementsprechend wird nun auch im *Kind Check* sichergestellt, dass Typsynonyme nicht rekursiv definiert sind. Des Weiteren wird in dieser Phase auch getestet, ob die Superklassenbeziehung azyklisch ist (vgl. Unterabschnitt 3.2.1).

In dem in Kapitel 6 vorgestellten, erweiterten Typsystem ist die Kenntnis der Sorten für alle Konstruktoren und Konstruktorvariablen erforderlich gewesen. Diese können glücklicherweise mittels bekannter Techniken automatisch ermittelt werden, sodass keine explizite Angabe der Sorten notwendig ist. Im Prinzip können die Typausdrücke als einfach getyptes Lambda-Kalkül aufgefasst werden, dessen einziger Basistyp die Sorte $*$ ist. Die Sorteninferenz entspricht dann genau der Typinferenz für den einfach getypten Lambda-Kalkül [Chu40].

Unter Umständen kann es vorkommen, dass für eine Typvariable keine eindeutige Sorte ermittelt werden kann. Dies passiert zum Beispiel bei dem folgenden Datentyp:

```
data T a = L | F (T a) (T a)
```

Für den Typkonstruktor `T` wird zunächst die allgemeine Sorte $\kappa \to *$ für eine beliebige Sorte κ ermittelt. Da wir aber keine polymorphen Sorten unterstützen, werden alle unbestimmten Sorten in solchen Fällen mit $*$ instanziiert. In dem obigen Beispiel wird für den Typkonstruktor `T` letztendlich also die Sorte $* \to *$ ermittelt.

Nach der Sorteninferenz für alle vorkommenden Konstruktoren innerhalb des Moduls besteht die Aufgabe der Prüfung der Wohlgeformtheit von Typausdrücken nur noch in dem Test, ob jeder Typ von der Sorte $*$ ist. In ähnlicher Weise muss für jede Instanzdeklaration geprüft werden, ob der dort angegebene Instanztyp dieselbe Sorte wie die zugehörige Typklasse hat.

8.5.6. Syntax Check

Zunächst wurde die Prüfung auf mehrfache Definitionen im *Syntax Check* so erweitert, dass nun auch Klassenmethoden miteinbezogen werden. Darüber hinaus wurden sämtliche Überprüfungen des *Syntax Checks* ebenso wie die Unterscheidung von nullstelligen Typkonstruktoren und Typvariablen auf Klassen- und Instanzdeklarationen erweitert. Gleiches gilt für die in dieser Phase vorgenommene Umbenennung (s. Unterabschnitt 7.3.4).

8.5.7. Precedences Check

Der *Precedences Check* wurde erweitert, sodass er auch die Infix-Deklarationen in Klassendeklarationen berücksichtigt. Weiterhin wurde die Umordnung des abstrakten Syntaxbaums auf Klassen- und Instanzdeklarationen ausgeweitet.

8.5.8. Instance Check

Der neue *Instance Check* baut die Instanzenumgebung auf, indem er ihr alle im Modul definierten Instanzen hinzufügt und dabei sicherstellt, dass es keine doppelten Instanzen für dieselbe Typklasse und denselben Typkonstruktor gibt. Es wird außerdem für jede C-T-Instanz geprüft, ob der Typ T eine Instanz jeder Superklasse der Klasse C ist. Weiterhin muss ein angegebener Instanzkontext alle anderen Instanzkontexte der zugehörigen Superklasseninstanzen erfüllen, sprich implizieren.

Beispiel 8.2.:
Gegeben sei das folgende Modul.

```
class Eq a
class Eq a => Ord a

instance (Eq a, Eq b) => Eq (a, b)
instance (Ord a, Ord b) => Ord (a, b)
```

Dann muss für die `Ord-(,)`*-Instanz auch eine* `Eq-(,)`*-Instanz existieren, weil* `Eq` *eine Superklasse von* `Ord` *ist. Aus demselben Grund muss der Instanzkontext der* `Ord-(,)`*-Instanz,* {`Ord a`, `Ord b`}*, den der* `Eq-(,)`*-Instanz,* {`Eq a`, `Eq b`}*, erfüllen. Beide Bedingungen sind hier erfüllt.*

Zuletzt prüft der *Instance Check*, ob alle in der Default-Deklaration spezifizierten Typen, sofern eine solche Deklaration angegeben wurde, Instanzen der `Num`-Klasse sind (vgl. Unterabschnitt 3.2.7).

8.5.9. Type Check

Da die Typkonstruktorenumgebung bereits im *Kind Check* aufgebaut wurde, entfällt dieser Schritt nun im *Type Check*. Ansonsten ist der allgemeine Ablauf dieser Phase weitestgehend identisch geblieben, weswegen nachfolgend nur auf die wichtigsten Änderungen und Ergänzungen eingegangen wird.

Eintragen von Klassenmethoden Zu Beginn der Typprüfung werden nun zusätzlich auch die Typen aller Klassenmethoden in die Werteumgebung eingetragen, da diese, ähnlich wie Datenkonstruktoren und Feldbezeichner, für die Typermittlung bekannt sein müssen.

Typermittlung Die Typermittlung verwendet nun den in Abschnitt 6.5 vorgestellten Algorithmus $\mathcal{W}$ für das erweiterte Typsystem, der analog zu dem ursprünglichen Algorithmus $\mathcal{W}$ auf die übrigen Konstrukte des abstrakten Syntaxbaums erweitert wurde (s. Unterabschnitt 7.3.6). Die jeweils ermittelten Typen werden nicht länger nur in die Werteumgebung, sondern wie in Abschnitt 8.3 beschrieben auch als Annotation in den abstrakten Syntaxbaum eingetragen. Im Unterschied zu vorher wird für numerische Literale in Ausdrücken nun ein `Num`-Kontext angenommen. Ebenso werden für arithmetische Sequenzen und Do-Notationen `Enum`- bzw. `Monad`-Kontexte eingeführt. Durch diese Maßnahmen wird die Überladung der jeweiligen syntaktischen Konstrukte ermöglicht.

Defaulting Unter Umständen können bei der Typermittlung mehrdeutige Typvariablen (s. Unterabschnitt 3.2.6) auftreten. Tritt eine solche Mehrdeutigkeit mit einem numerischen Kontext auf, wird versucht, sie mithilfe der gegebenen Default-Deklaration (s. Unterabschnitt 3.2.7) aufzulösen, indem der erste Typ aus der Liste gewählt wird, der alle für die mehrdeutige Typvariable geltenden Constraints erfüllt.

Berücksichtigung von Klassen- und Instanzdeklarationen Die Deklarationen in Klassen- und Instanzdeklarationen werden wie explizit getypte Funktionen behandelt, wobei für Instanzimplementierung die Klassenvariable im Typ der Methode durch den entsprechenden Instanztyp ersetzt werden muss.

Berücksichtigung von Typsignaturen Wie zuvor werden angegebene Typsignaturen bei der Typermittlung berücksichtigt, indem sie der Typannahme im Vorfeld hinzugefügt werden. Die Prüfung, ob der inferierte Typ mit dem in der Typsignatur spezifizierten übereinstimmt unterliegt aber einer Änderung, da nun der Kontext berücksichtigt werden muss. Für den Kontext des inferierten Typs muss gelten, dass dieser durch den in der Typsignatur angegebenen Kontext impliziert wird.

Beispiel 8.3.:
Es sei die folgende Funktion gegeben.

```
f :: Ord a => a -> a -> Bool
f = (==)
```

Dann ist obige Typsignatur gültig, da der angegebene Kontext {`Ord a`} *den aufgrund der Verwendung von* `(==)` *inferierten Kontext* {`Eq a`} *impliziert. Der Typ von* `f` *wurde also gewissermaßen künstlich beschränkt.*

Implizit und explizit getypte Deklarationen Innerhalb von Deklarationsgruppen wird eine getrennte Typermittlung für implizit und explizit getypte Deklarationen durchgeführt. Explizit getypte Deklarationen sind dabei solche, für die vom Programmierer eine Typsignatur angegeben wurde. Die Idee, explizit getypte vor implizit getypten Deklarationen zu prüfen, stammt aus einer Arbeit von Mark P. Jones [Jon99]. Der Vorteil dieses Verfahrens liegt darin, dass in vielen Fällen allgemeinere Typen für Funktionen inferiert werden können, wie das folgende Beispiel verdeutlicht.

Beispiel 8.4.:
Es sei der folgende Code gegeben.

```
f :: Eq a => a -> Bool
f x = (x == x) || g True

g y = (y <= y) || f True
```

Dann wird für `g` *der Typ* `Ord a => a -> Bool` *ermittelt, da* `f` *als explizit getypte Funktion erst nach* `g` *betrachtet wird. Würden beide Funktionen dagegen zusammen geprüft werden, würde für* `g` *nur der Typ* `Bool -> Bool` *bestimmt werden.*

Liberalisierung der Monomorphierestriktion Zwar ist das Typsystem von Curry sowohl in seiner ursprünglichen als auch erweiterten Form korrekt (s. Sätze 5.11 und 6.13), es ist unter Umständen aber unnötig restriktiv, was sich anhand des folgenden Ausdrucks nachvollziehen lässt.

```
let nil = [] in (1 : nil, 'a' : nil)
```

Dieser Ausdruck würde vom Compiler zurückgewiesen werden, da `nil` im Rumpf des Let-Ausdrucks polymorph verwendet wird und dies nach der in Curry geltenden Monomorphierestriktion (s. Abschnitt 5.2) verboten ist.

Im Allgemeinen wäre es jedoch sicher, die Generalisierung für Variablen zuzulassen, die an Ausdrücke ohne freie Variablen gebunden sind. Das Identifizieren solcher Grundterme (s. Abschnitt 2.2) würde allerdings eine komplexe semantische Analyse des Programms erfordern. Deshalb wird folgende Annäherung genutzt: Die Generalisierung wird für Variablen zugelassen, die an Ausdrücke gebunden sind, für die gezeigt werden kann, dass sie keine freien Variablen enthalten. Dies sind genau die *nicht-expansiven* Ausdrücke [Lux07].

Definition 8.5. (Nicht-expansive Ausdrücke)**:**
Ein Ausdruck ist nicht-expansiv*, wenn es sich um eins der folgenden Dinge handelt:*

- *ein Literal,*
- *eine Variable,*
- *die Applikation eines n-stelligen Konstruktors auf höchstens n nicht-expansive Ausdrücke,*
- *die Applikation einer n-stelligen Funktion auf höchstens $n-1$ nicht-expansive Ausdrücke,*
- *ein Let-Ausdruck, dessen innerer Ausdruck nicht-expansiv ist und dessen Deklarationen nur Funktionen oder nicht-expansive Ausdrücke definieren,*
- *ein Ausdruck, dessen vereinfachte Form (s. Unterabschnitt 7.3.11) einer der obigen Fälle entspricht.*

Ein Ausdruck, der nicht nicht-expansiv ist, wird auch als expansiv *bezeichnet.*

Wir haben die Typermittlung in unserer Implementierung der vorgestellten Idee folgend modifiziert, sodass das oben genannte Beispiel als wohlgetypt erkannt wird. Der Ausdruck `[]` ist nach gegebener Definition nicht-expansiv, da es sich um einen nullstelligen Konstruktor handelt, der auf keine weiteren Argumente angewendet wird.

8.5.10. Export Check

Da Typklassen Teil der Typkonstruktoren- und Klassenmethoden Teil der Werteumgebung sind, sind die Anpassungen des *Export Checks* nur von geringem Ausmaß und betreffen im Prinzip ausschließlich mögliche Fehlermeldungen.

8.5.11. Export Expansion

Die in Unterabschnitt 7.3.9 beschriebene Transformation zur Vereinheitlichung der Exportspezifikation wurde wie folgt modifiziert.

Transformation 8.6. (Exportspezifikation)**:**
Ein Export der Form `T(..)` *wird, wenn* T *eine Typklasse ist, in* `T(`m_1`, ..., `m_n`)` *mit* $n \geq 0$ *transformiert, wobei* $m_1, \ldots, m_n$ *die Klassenmethoden der Typklasse* T *sind.*

8.5.12. Desugaring

Die *Desugaring*-Phase wurde um neue Umformungen für Ganz- und Gleitkommazahlen erweitert (s. Abbildung 8.3). Auf diese Weise wird die Überladung numerischer Literale in Ausdrücken realisiert. Alle anderen Umformungen (s. Unterabschnitt 7.3.11) sind bestehen geblieben, wobei anzumerken ist, dass es sich bei einigen der verwendeten Funktionen nun um Klassenmethoden handelt.

Konstrukt	**Ausdruck**	**Umformung**
Ganzzahl	n	`fromInteger` n
Gleitkommazahl	n	`fromRational` n

Abbildung 8.3.: Zusätzliche Umformungen für Ausdrücke

Bei der Anwendung sämtlicher Umformungen muss beachtet werden, dass die Typannotationen im abstrakten Syntaxbaum entsprechend angepasst bzw. für hinzugefügte Ausdrücke überhaupt erst erzeugt werden müssen. Außerdem werden jetzt auch Klassen- und Instanzdeklarationen berücksichtigt, indem alle Umformungen auf die darin enthaltenen Deklarationen ausgedehnt werden.

8.5.13. Dictionary Insertion

In der *Dictionary Insertion* setzen wir die in Kapitel 4 vorgestellte Wörterbuchtransformation um. Da in dieser Phase noch einige andere Schritte durchgeführt werden, skizzieren wir ihren Ablauf nachfolgend.

1. Der erste Schritt ist die Ergänzung aller nullstelligen Klassenmethoden um ein zusätzliches Unit-Argument. Damit wird das Problem des ungewollten Teilens von Nichtdeterminismus behoben (s. Abschnitt 4.5).

   Diese *Augmentation* umfasst wiederum mehrere Schritte. Innerhalb von Klassendeklarationen müssen zunächst alle Typsignaturen und Funktionsgleichungen entsprechend angepasst werden. In Instanzdeklarationen sind nur letztere betroffen, da dort keine Typsignaturen erlaubt sind. Im Anschluss daran muss jeder Aufruf einer nullstelligen Klassenmethode im Code ebenfalls um ein Unit-Argument ergänzt werden, damit das Modul insgesamt konsistent bleibt.

Neben der Anpassung des Moduls an sich muss auch die gesamte Compiler-Umgebung modifiziert werden, um zu reflektieren, dass auch Methoden in importierten Modulen umgeformt wurden.

- In der Werteumgebung müssen Arität und Typschema von nullstelligen Klassenmethoden geändert werden. Hier ist nun die Information relevant, ob eine eingetragene Funktion eine Klassenmethode ist, da normale nullstellige Funktionen nicht von dem Problem betroffen sind und daher unverändert bleiben.
- In der Klassen- und Instanzenumgebung müssen lediglich die Aritäten entsprechend angepasst werden.
- Auch die Schnittstellenumgebung ist von der Anpassung betroffen, sodass der Typ jeder nullstelligen Klassenmethode erweitert und ihre Arität inkrementiert werden muss.

2. Nach der Erweiterung aller nullstelligen Klassenmethoden um ein Unit-Argument erfolgt die eigentliche Umformung des Moduls, bei der die in Kapitel 4 formal beschriebenen Transformationen 4.1, 4.2, 4.3 und 4.6 nacheinander angewendet werden. An dieser Stelle ist es wichtig, dass die Klassenumgebung alle definierten Superklassen und Methoden und nicht nur die sichtbaren beinhaltet, da ansonsten nicht der korrekte Wörterbuchtyp erzeugt werden könnte.

   Ähnlich wie bei der *Augmentation* muss auch die Compiler-Umgebung passend zur Wörterbuchtransformation umgeformt werden. Konkret müssen neu erzeugte Typ- und Funktionsdeklarationen in die Typkonstruktoren- respektive Werteumgebung eingetragen und alle bereits in die Werteumgebung eingetragenen Typen transformiert werden. Die Schnittstellenumgebung ist erneut betroffen und muss analog angepasst werden.

3. Im Anschluss werden einige Bereinigungen durchgeführt. Es werden sämtliche Typklassen aus der Typkonstruktorenumgebung entfernt. Klassenmethoden müssen dagegen nicht aus der Werteumgebung gelöscht werden, da sie nun den neu eingeführten Selektoren entsprechen (s. Abschnitt 4.1). Außerdem müssen möglicherweise einige Infix-Deklarationen und die zugehörigen Einträge in der Präzedenzenumgebung entfernt werden, da durch die Wörterbuchtransformation zusätzliche Parameter hinzugekommen sein könnten (s. Beispiel 8.7).

4. Als letzter Schritt wird eine Optimierung des erzeugten Codes durchgeführt. Wann immer ein Selektor für eine Klassenmethode auf ein bekanntes konkretes Wörterbuch angewandt wird, kann dieser Aufruf direkt durch die angehobene Implementierung der jeweiligen Instanz ersetzt werden [PJ93].

Nach den beschriebenen Schritten ist sowohl das Modul als auch die Compiler-Umgebung von jeglichen Typklassenelementen befreit.

Beispiel 8.7.:
Die Infix-Deklaration

```
infix 4 `elem`
```

verliert ihre Gültigkeit, da die Funktion `elem` *nach der Wörterbuchtransformation den Typ*

```
Dict_Eq a -> a -> [a] -> Bool
```

besitzt und somit nicht mehr zweistellig ist.

8.5.14. Conversion into Intermediate Language

Die Umwandlung in die Zwischensprache ist weitestgehend unverändert geblieben. Die Typdarstellung der Zwischensprache unterstützt allerdings keine Typausdrücke höherer Ordnung, sodass solche entsprechend konvertiert werden müssen. Zu diesem Zweck wird die Existenz eines abstrakten Typkonstruktors `(@)` angenommen, der die Typvariablenapplikation repräsentiert und als Typsynonym der Form `type (@) a b = a b` aufgefasst werden kann [YW14].

Beispiel 8.8.:
Der Typ der Klassenmethode `return`

```
Monad m => a -> m a
```

wird durch die Wörterbuchtransformation in

```
Dict_Monad m -> a -> m a
```

umgewandelt. Dabei bleibt mit `m a` *die Typvariablenapplikation im Ergebnistyp erhalten. Diese wird bei der Übersetzung in die Zwischensprache in*

```
Dict_Monad m -> a -> (@) m a
```

umgewandelt.

8.6. Repräsentation der Wörterbücher

Bei der erfolgten Umsetzung von Typklassen mittels Wörterbüchern ergibt sich ein Problem bei der internen Darstellung der Wörterbuchtypen, auf welches nachfolgend näher eingegangen wird. Von dem beschriebenen Problem sind generell alle Klassen betroffen, die polymorphe Methoden enthalten, also solche Klassenmethoden, deren Typen außer der Klassenvariable noch andere Typvariablen enthalten.

Als Beispiel betrachten wir in diesem Abschnitt die aus Unterabschnitt 3.1.1 bekannte Typkonstruktorklasse `Functor`. Für diese wird durch Transformation 4.1 der folgende Wörterbuchtyp erzeugt.

```
data Dict_Functor f =
  Dict_Functor ((a -> b) -> f a -> f b)
```

Die Problematik liegt nun in der Tatsache, dass die Typvariablen `a` und `b` in der erzeugten Deklaration unzulässigerweise ungebunden sind. Für eine korrekte Transformation müssten die Typvariablen allerdings wie folgt allquantifiziert sein.

```
data Dict_Functor f =
  Dict_Functor (forall a b . (a -> b) -> f a -> f b)
```

Diese Art der geschachtelten Allquantifizierung wird gemeinhin als *Polymorphismus höheren Ranges* bezeichnet, wobei der tatsächliche Rang vom Grad der Verschachtelung abhängt [OL96]. Im vorliegenden Fall handelt es sich beispielsweise um *Polymorphismus zweiten Ranges*. Da eine Unterstützung dieser Art von Polymorphismus allerdings weitreichendere Änderungen nach sich ziehen würde (s. Abschnitt 9.2), wird vorerst die undokumentierte Unterstützung des Front-Ends für existentielle Typen genutzt, sodass die Typvariablen `a` und `b` folgendermaßen existentiell quantifiziert werden.

```
data Dict_Functor f =
  forall a b . Dict_Functor ((a -> b) -> f a -> f b)
```

Dabei ist zu beachten, dass es sich um keine korrekte Repräsentation des tatsächlichen Wörterbuchtyps handelt. Dieser Umstand führt in der vorliegenden Implementierung aber zu keinen Fehlern, da der erzeugte FlatCurry-Code an sich korrekt ist.

8.7. Modulsystem

Die Anpassungen des Modulsystems betreffen neben dem Im- und Export an sich auch die Erweiterung der Schnittstellen um Typklassen und Instanzen.

8.7.1. Import

Typklassen werden beim Import mit den jeweils passenden Informationen sowohl in die Typkonstruktorenumgebung als auch in die neue Klassenumgebung eingetragen. Klassenmethoden werden dabei der Werteumgebung hinzugefügt. Die Instanzen dagegen werden in die Instanzenumgebung eingetragen.

8.7.2. Export

Beim Export wurden der Schnittstelle die Typklassen und Instanzen als Entitäten hinzugefügt. Während die exportierten Klassen von der angegebenen Exportspezifikation abhängen, werden immer alle Instanzen exportiert [Jon03]. Statt ganzer Typklassen können auch einzelne Klassenmethoden exportiert werden. Diese tauchen dann als Funktionen in der Schnittstelle auf, wobei sie als Methode markiert werden, damit sie bei der Wörterbuchtransformation (s. Unterabschnitt 8.5.13) entsprechend berücksichtigt werden.

8.8. Sonstiges

Alle bisherigen Anpassungen bezogen sich auf die Generierung des FlatCurry-Codes. Es wurden aber auch einige Änderungen hinsichtlich anderer Ausgaben des Front-Ends vorgenommen.

8.8.1. AbstractCurry

Wir haben die AbstractCurry-Darstellung um neue Konstrukte für die Typklassenelemente und die Typvariablenapplikation erweitert. Klassenmethoden werden unabhängig davon, ob getyptes oder ungetyptes AbstractCurry erzeugt werden soll, immer mit ihrem Typ annotiert, da sie überhaupt erst über ihren Typ definiert werden und ein Fehlen dieser Information sonst zu einer unvollständigen Repräsentation des Programms führen würde.

8.8.2. HTML

Die Generierung der HTML-Ausgabe wurde ebenfalls dahingehend angepasst, dass die Typklassenelemente bei der Syntaxhervorhebung sowie dem Erstellen der Querverweise berücksichtigt werden.

9. Abschlussbetrachtungen

In diesem Kapitel werden die erzielten Ergebnisse noch einmal zusammengefasst und ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

9.1. Ergebnisse

Im Rahmen dieser Arbeit wurde das Front-End des PAKCS um Typ- und Typkonstruktorklassen erweitert und somit erfolgreich der Grundstein für die zukünftige Unterstützung von Typ- und Typkonstruktorklassen in der Programmiersprache Curry gelegt.

Die vorliegende Implementierung stellt mit Ausnahme des automatischen Ableitens von Instanzen für bestimmte vordefinierte Klassen sämtliche aus Haskell bekannten Nutzungsmöglichkeiten zur Verfügung, inklusive des Defaultings für mehrdeutige Typvariablen mit numerischem Kontext. Darüber hinaus ist als Konsequenz der Erweiterung um Typ- und Typkonstruktorklassen jetzt auch die Überladung numerischer Literale, arithmetischer Sequenzen und der Do-Notation möglich.

Die Integration unserer Anpassungen in die bestehende Struktur des Compilers ist in einer Art und Weise erfolgt, die künftige Wartungen und Erweiterungen (s. nachfolgenden Abschnitt) einfach gestattet. Im Zuge der vorgenommenen Implementierung wurde die Compiler-Umgebung um zusätzliche Unterumgebungen für Klassen und Instanzen erweitert und der Kompilierverlauf unter Wahrung der bestehenden Trennung von Prüfungs- und Transformationsphasen um neue Phasen ergänzt. Die neue Typprüfung basiert auf einem um Konstruktoren und Prädikate erweiterten Typsystem, für das sowohl Typungsregeln als auch ein Typinferenzalgorithmus angegeben wurden. Die Unterstützung der Typ- und Typkonstruktorklassen wurde mithilfe von Wörterbüchern umgesetzt, wobei die zugrundeliegenden Transformationen allesamt formal beschrieben wurden. Weiterhin wurde auf eine Besonderheit bei der Verwendung dieses Ansatz in funktionallogischen Sprachen eingegangen, die das ungewollte Teilen von Nichtdeterminismus betrifft, sowie eine entsprechende Lösung präsentiert und umgesetzt.

9.2. Ausblick

Wie bereits in der Einleitung dieser Arbeit erwähnt, schließt die Anpassung eines ganzen Curry-Systems auch immer die der Bibliotheken und Werkzeuge mit ein, woraus sich direkt der nächste Schritt ergibt. Abseits davon bietet die vorliegende Implementierung aber auch so einige Ansatzpunkte für Ergänzungen und Verbesserungen, die wir nachfolgend auflisten.

- Wie bereits erwähnt, fehlt aktuell die Möglichkeit der automatischen Ableitung von Instanzen für bestimmte vordefinierte Typklassen. Dank der Struktur der Implementierung lässt sich diese Funktion aber leicht nachrüsten.

  Zunächst müsste dafür die Modulsyntax um Deriving-Klauseln für Datentyp- und Newtype-Deklarationen erweitert werden. Im *Instance Check* muss dann geprüft werden, ob für die in einer Deriving-Klausel angegebenen Klassen überhaupt Instanzen erzeugt werden können. In diesem werden alle abgeleiteten Instanzen in die Instanzenumgebung eingetragen, damit sie im *Type Check* verfügbar sind. Auf diese Weise wären auch keinerlei Änderungen der Schnittstellensyntax oder des Im- und Exports von Modulen erforderlich. Die eigentliche Code-Erzeugung, welche bereits ausführlich von Matthias Böhm beschrieben wurde [Boh13], würde als zusätzliche Phase bei den Transformationen eingefügt werden, genauer gesagt vor der *Desugaring*-Phase. Dies hat zum einen den Grund, dass das Quellprogramm während der Prüfungsphasen weitestgehend unverändert bleiben soll (vgl. Abschnitt 7.3), zum anderen, dass die Code-Erzeugung sich dadurch vergleichsweise simpel gestaltet, da noch sämtliche Konstrukte verwendet werden können, die ansonsten durch das *Desugaring* wegfallen würden.

  Alle anderen Kompilierphasen könnten unverändert bleiben. Allein eine Anpassung der AbstractCurry-Darstellung wäre zusätzlich vonnöten, um die Deriving-Klauseln widerzuspiegeln.

- Momentan werden noch nicht alle Standardklassen von Haskell vollständig unterstützt [Jon03]. Insbesondere fehlen die Typklassen `RealFrac`, `Floating` sowie `RealFloat` und die Typkonstruktorklasse `MonadPlus`. Diese lassen sich aber samt der passenden Instanzen einfach in der Standardbibliothek ergänzen.

  Im gleichen Zuge könnte auch das *Functor-Applicative-Monad Proposal* umgesetzt werden [HW2], in dem eine Ergänzung bzw. Änderung der Standard-Superklassenhierarchie vorgeschlagen wird. Da neuere Haskell-Compiler wie der GHC 7.10 diesen Vorschlag ohnehin bereits adaptiert haben [GHC15], wäre eine Umsetzung für Curry nur logisch, um eine möglichst große Ähnlichkeit zu aktuelleren Versionen von Haskell zu wahren.

- Der *Warn Check* ist in der Implementierung bisher unverändert geblieben. Insbesondere werden Klassen- und Instanzdeklarationen aktuell nicht berücksichtigt und keinerlei Typklassen-bezogenen Warnungen ausgegeben. Es wäre daher sinnvoll, den *Warn Check* entsprechend zu erweitern.

  – Die bereits vorhandenen Prüfungen müssten auf Klassen- und Instanzdeklarationen ausgeweitet werden. Dabei ist allerdings zu beachten, dass einige Warnungen für die inneren Deklarationen außer Acht gelassen werden müssen. So ist es beispielsweise nicht zulässig, Typsignaturen für Implementierungen in Instanzen anzugeben (s. Unterabschnitt 3.2.2), sodass eine Warnung für eine fehlende Typsignatur keinen Sinn macht.

  - Es könnten Warnungen für nicht notwendige Constraints in Kontexten (vgl. Unterabschnitt 3.2.5) ausgegeben werden. Diese ließen sich durch einen Vergleich des angegebenen mit dem vollständig reduzierten Kontext ermitteln.
  - Meldungen über fehlende Implementierungen in Instanzdeklarationen, sofern die zugehörige Typklasse selbst keine Standardimplementierung bereitstellt, wären denkbar. Hierfür muss in der Klassenumgebung für jede Klassenmethode noch zusätzlich gespeichert werden, ob es eine Standardimplementierung gibt, um falsche Warnungen zu diesem Punkt zu vermeiden.
  - Es könnte vor verwaisten Instanzen gewarnt werden. Dabei handelt es sich um Instanzen, bei denen weder die involvierte Klasse noch der involvierte Typkonstruktor im selben Modul wie die Instanz deklariert wurde. Da immer alle Instanzen im- und exportiert werden (s. Unterabschnitt 8.7.2), können importierte verwaiste Instanzen unter Umständen zu Konflikten mit lokal definierten oder anderen importierten Instanzen führen [HW1].

  Für alle vorgeschlagenen Warnungen müssten darüber hinaus Optionen für das Front-End bereitgestellt werden, die deren gezieltes An- und Abschalten erlauben.

- Im Laufe der Zeit sind einige Erweiterungen für Typklassen entstanden, um deren Unterstützung Curry ebenfalls ergänzt werden könnte. Ein Beispiel für eine solche Erweiterung sind Multiparametertypklassen [CHO92], die es erlauben, Typklassen mit mehr als einer Klassenvariable zu definieren. Im direkten Zusammenhang mit Multiparametertypklassen steht die Erweiterung um *Functional Dependencies* [Jon00]. Sie ermöglicht es, funktionale Abhängigkeiten zwischen den verschiedenen Parametern einer Multiparametertypklasse zu definieren.

Neben den oben genannten kommen noch weitere Punkte infrage, die zwar keinen unmittelbaren Teil der Arbeit darstellen, aber dennoch mit ihr im Zusammenhang stehen.

- Aktuell besteht in der Implementierung das Problem, dass bei der internen Repräsentation der Wörterbuchtypen mitunter auf existentielle Typen ausgewichen werden muss (s. Abschnitt 8.6). Um die Wörterbuchtransformation als wirkliche Quellcode-zu-Quellcode-Transformation realisieren zu können, müsste eine Möglichkeit geschaffen werden, intern Polymorphismus zweiten Ranges darstellen zu können. Eine solche Anpassung würde aber automatisch auch entsprechende Änderungen an der Zwischensprache sowie FlatCurry und damit auch am Back-End nach sich ziehen. Ein Vorteil läge allerdings darin, dass die Umsetzung der KiCS2-Unterstützung dadurch einfacher werden würde (s. nächsten Punkt).

- In dieser Arbeit wurde zunächst einmal nur das PAKCS berücksichtigt (vgl. Abschnitt 1.1) und das KiCS2 außen vor gelassen. Da beide Systeme aber auf einem gemeinsamen Front-End aufbauen, liegt es nahe, auch letzterem die Unterstützung von Typ- und Typkonstruktorklassen zuteil werden zu lassen. Während es für das PAKCS noch ausreichend war, ausschließlich das Front-End zu modifizieren,

sind für das KiCS2 hingegen weitere Anpassungen am Back-End erforderlich. Es müsste ein Weg gefunden werden, Wörterbücher mit polymorphen Datenkomponenten (s. Abschnitt 8.6) geeignet nach Haskell zu übertragen. Zwar bietet der GHC, auf dem das KiCS2 aufbaut, hierfür eine Spracherweiterung namens `PolymorphicComponents` an [GHC15], es bleibt aber zunächst offen, inwieweit sich deren Verwendung mit der bestehenden Übersetzung von FlatCurry nach Haskell vertragen würde. Ein weiteres Problem besteht in der Tatsache, dass im Back-End des KiCS2 noch eine Typinferenz auf dem generierten FlatCurry-Code durchgeführt wird [Obe12]. Diese müsste wegen des Polymorphismus zweiten Ranges ebenfalls angepasst werden.

- Im aktuellen Zustand werden die Typannotationen im abstrakten Syntaxbaum (s. Abschnitt 8.3) nach der *Dictionary Insertion* de facto nicht länger benötigt und gehen spätestens mit der Übersetzung in die Zwischensprache verloren. Es wäre jedoch eine Überlegung wert, sowohl die Zwischensprache als auch FlatCurry so zu erweitern, dass alle Typannotationen erhalten blieben. Eine solche annotierte FlatCurry-Variante könnte für viele Anwendungsbereiche interessant sein, wäre aber insbesondere für die Gewährleistung der Typklassenunterstützung im KiCS2 nützlich, weil dadurch die zuvor angesprochene FlatCurry-Typinferenz im Back-End des KiCS2 gänzlich entfallen könnte (s. vorherigen Punkt).

- Zwar wurde im Rahmen dieser Arbeit bereits eine Optimierung der Wörterbuchtransformation durch Spezialisierung der Methodenaufrufe umgesetzt (s. Unterabschnitt 8.5.13), Mark P. Jones hat allerdings ein Verfahren vorgestellt, das diesen Ansatz weiter fortführt [Jon95]. Ausgehend von dem Programm, das aus der Wörterbuchtransformation hervorgegangen ist, werden sämtliche Wörterbücher entfernt, indem spezialisierte Varianten aller überladenen Funktionen generiert werden. Dabei wird partielle Auswertung als Technik eingesetzt, um die Anzahl der neu erzeugten Funktionen so gering wie möglich zu halten. In der Tat hat sich am Ende sogar gezeigt, dass es nicht zu der häufig erwarteten Code-Explosion durch die vollständige Spezialisierung kommt. Leider ist ein entscheidender Nachteil dieses Verfahrens, dass es in der vorgestellten Form nicht mit dem Konzept der getrennten Kompilierung von Modulen, wie es auch in Curry zu finden ist, vereinbar ist. Dennoch erscheint eine Weiterverfolgung dieses Ansatzes vielversprechend. Abgesehen vom zu erwartenden Geschwindigkeitsgewinn würde sich durch den vollständigen Wegfall aller Wörterbücher vor allem das Problem mit dem Polymorphismus zweiten Ranges erübrigen, sodass alle zuvor erwähnten Vorschläge im Prinzip hinfällig wären.

Literatur

[Boh13] Matthias Böhm. „Erweiterung von Curry um Typklassen“. Masterarbeit. Christian-Albrechts-Universität zu Kiel, Okt. 2013.

[CHO92] Kung Chen, Paul Hudak und Martin Odersky. „Parametric Type Classes“. In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. LFP '92. San Francisco, California, USA: ACM, 1992, S. 170–181.

[Chu40] Alonzo Church. „A Formulation of the Simple Theory of Types“. In: *Journal of Symbolic Logic* 5.2 (Juni 1940), S. 56–68.

[Dam84] Luis Damas. „Type Assignment in Programming Languages“. Dissertation. University of Edinburgh, 1984.

[DJH02] Iavor S. Diatchki, Mark P. Jones und Thomas Hallgren. „A Formal Specification of the Haskell 98 Module System“. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania: ACM, 2002, S. 17–28.

[DM82] Luis Damas und Robin Milner. „Principal Type-schemes for Functional Programs“. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: ACM, 1982, S. 207–212.

[FH99] Francisco Javier López Fraguas und Jaime Sánchez Hernández. „TOY: A multiparadigm declarative system“. In: *International Conference on Rewriting Techniques and Applications*. Springer. 1999, S. 244–247.

[GHC15] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.10.3*. Dez. 2015. URL: `https://www.haskell.org/ghc/docs/latest/users_guide.pdf` (abgerufen am 10. 04. 2016).

[HA77] M. C.B. Hennessy und E. A. Ashcroft. „Parameter-passing Mechanisms and Nondeterminism“. In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: ACM, 1977, S. 306–311.

[Hal+96] Cordelia V. Hall u. a. „Type Classes in Haskell“. In: *ACM Trans. Program. Lang. Syst.* 18.2 (März 1996), S. 109–138.

[Han16] Michael Hanus (Hrsg.) *Curry: An Integrated Functional Logic Language (Version 0.9.0)*. Verfügbar unter: `http://www.curry-language.org`. 2016.

[HB90] Kevin Hammond und Stephen Blott. „Implementing Haskell Type Classes“. In: *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. London, UK, UK: Springer-Verlag, 1990, S. 266–286.

[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.

[HPW92] Paul Hudak, Simon Peyton Jones und Philip Wadler (Hrsg.) „Report on the Programming Language Haskell: A Non-strict, Purely Functional Language Version 1.2“. In: *SIGPLAN Not.* 27.5 (Mai 1992), S. 1–164.

[Hus92] Heinrich Hussmann. „Nondeterministic algebraic specifications and nonconfluent term rewriting“. In: *The Journal of Logic Programming* 12.3 (1992), S. 237–255.

[HW1] HaskellWiki. *Orphan instance*. URL: `https://wiki.haskell.org/Orphan_instance` (abgerufen am 30. 08. 2016).

[HW2] HaskellWiki. *Functor-Applicative-Monad Proposal*. URL: `https://wiki.haskell.org/Functor-Applicative-Monad_Proposal` (abgerufen am 06. 09. 2016).

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

[Joh85] Thomas Johnsson. „Lambda lifting: Transforming programs to recursive equations“. In: *FPCA*. Springer. 1985, S. 190–203.

[Jon00] Mark P. Jones. „Type Classes with Functional Dependencies“. In: *Proceedings of the 9th European Symposium on Programming, ESOP 2000*. Berlin, Heidelberg: Springer-Verlag, 2000, S. 230–244.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[Jon92a] Mark P. Jones. „A Theory of Qualified Types“. In: *Proceedings of the 4th European Symposium on Programming*. ESOP '92. London, UK, UK: Springer-Verlag, 1992, S. 287–306.

[Jon92b] Mark P. Jones. „Qualified Types: Theory and Practice“. Diss. Programming Research Group, Oxford University Computing Laboratory, 1992.

[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*. FPCA '93. Copenhagen, Denmark: ACM, 1993, S. 52–61.

[Jon95] Mark P. Jones. „Dictionary-free Overloading by Partial Evaluation“. In: *Lisp Symb. Comput.* 8.3 (Sep. 1995), S. 229–248.

[Jon99] Mark P. Jones. „Typing Haskell in Haskell“. In: *Haskell Workshop*. 1999.

[Lux07] Wolfgang Lux. „Lifting Curry's Monomorphism Restriction“. In: *24. Workshop der GI-Fachgruppe „Programmiersprachen und Rechenkonzepte“*. 2007.

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

[Lux09] Wolfgang Lux. *Type-classes and call-time choice vs. run-time choice*. Post to the Curry mailing list. 27. Aug. 2009. URL: `https://www.informatik.uni-kiel.de/~curry/listarchive/0790.html` (abgerufen am 06. 06. 2016).

[Mar11] Enrique Martin-Martin. „Type Classes in Functional Logic Programming“. In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '11. Austin, Texas, USA: ACM, 2011, S. 121–130.

[Mil78] Robin Milner. „A theory of type polymorphism in programming“. In: *Journal of computer and system sciences* 17.3 (1978), S. 348–375.

[Obe12] Jonas Oberschweiber. „Type Inference for a Declarative Intermediate Language“. Bachelorarbeit. Christian-Albrechts-Universität zu Kiel, Sep. 2012.

[OL96] Martin Odersky und Konstantin Läufer. „Putting Type Annotations to Work“. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: ACM, 1996, S. 54–67.

[Pee16] Björn Peemöller. „Normalization and Partial Evaluation of Functional Logic Programs“. Dissertation. Christian-Albrechts-Universität zu Kiel, 2016.

[PJ93] John Peterson und Mark Jones. „Implementing Type Classes“. In: *SIGPLAN Not.* 28.6 (Juni 1993), S. 227–236.

[Rah16] Katharina Rahf. „Überprüfung von Stilrichtlinien für deklarative Programme“. Masterarbeit. Christian-Albrechts-Universität zu Kiel, Juli 2016.

[Rob65] J. A. Robinson. „A Machine-Oriented Logic Based on the Resolution Principle“. In: *J. ACM* 12.1 (Jan. 1965), S. 23–41.

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

[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*. POPL '89. Austin, Texas, USA: ACM, 1989, S. 60–76.

[YW14] Jeremy Yallop und Leo White. „Lightweight Higher-Kinded Polymorphism“. In: *Functional and Logic Programming: 12th International Symposium, Flops 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*. Hrsg. von Michael Codish und Eijiro Sumii. Cham: Springer International Publishing, 2014, S. 119–135.

Teil IV.

Anhang

A. Syntaxbeschreibung

Da sich die vom Front-End erkannte Syntax durch die Erweiterung um Typklassen geändert hat, geben wir im Folgenden eine vollständige Beschreibung der Syntax in Form einer kontextfreien Grammatik an. Alle mit unserer Arbeit im Zusammenhang stehenden Änderungen und Zusätze sind farbig hervorgehoben.

Dieses Kapitel ist mit Ausnahme des letzten Abschnitts, der die Schnittstellensyntax betrifft, mit einigen Anpassungen Björn Peemöllers Dissertation [Pee16] entliehen.

A.1. Notationen

Die Syntax ist in erweiterter Backus-Naur-Form (EBNF) angegeben, wobei die folgenden Notationen verwendet werden.

$\mathit{NonTerm} ::= \alpha$	Produktion
$\mathit{NonTerm}$	Nichtterminalsymbol
`Term`	Terminalsymbol
$[\alpha]$	optionales Vorkommen
$\{\alpha\}$	optionale Wiederholung
(α)	Gruppierung
$\alpha \mid \beta$	Alternative
$\alpha_{\langle\beta\rangle}$	Differenz – Elemente, welche von α, jedoch nicht von β generiert werden

A.2. Wortschatz

A.2.1. Kommentare

Kommentare beginnen entweder mit „`--`“ und enden am Ende der Zeile, oder mit „`{-`“ und enden mit einem zugehörigen „`-}`“. Die Begrenzer „`{-`“ und „`-}`“ verhalten sich also wie Klammern und können geschachtelt werden.

A.2.2. Bezeichner und Schlüsselwörter

Die Groß- und Kleinschreibung von Bezeichnern spielt eine Rolle, d.h., der Bezeichner „abc“ unterscheidet sich von „ABC“. Auch wenn der Curry Report [Han16] vier verschiedene Modi für die Groß- und Kleinschreibung (Prolog, Gödel, Haskell, frei) spezifiziert, unterstützt das Front-End nur den *freien* Modus, der keinerlei Einschränkungen in Bezug auf die Groß- und Kleinschreibung von Bezeichnern vornimmt.

$$
\begin{aligned}
\mathit{Letter} &::= \textit{any ASCII letter} \\
\mathit{Dashes} &::= \texttt{--}\ \{\texttt{-}\} \\
\\
\mathit{Ident} &::= (\mathit{Letter}\ \{\mathit{Letter} \mid \mathit{Digit} \mid \texttt{\_} \mid \texttt{'}\})_{\langle \mathit{ReservedID} \rangle} \\
\mathit{Symbol} &::= \texttt{~} \mid \texttt{!} \mid \texttt{@} \mid \texttt{\#} \mid \texttt{\$} \mid \texttt{\%} \mid \texttt{\^{}} \mid \texttt{\&} \mid \texttt{*} \mid \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{?} \mid \texttt{.} \mid \texttt{/} \mid \texttt{|} \mid \texttt{\textbackslash} \mid \texttt{:} \\
\\
\mathit{ModuleID} &::= \{\mathit{Ident}\ \texttt{.}\}\ \mathit{Ident} \\
\mathit{TypeConstrID} &::= \mathit{Ident} \\
\mathit{TypeVarID} &::= \mathit{Ident} \mid \texttt{\_} \\
\mathit{ClassVarID} &::= \mathit{Ident} \\
\mathit{ExistVarID} &::= \mathit{Ident} \\
\mathit{DataConstrID} &::= \mathit{Ident} \\
\mathit{InfixOpID} &::= (\mathit{Symbol}\ \{\mathit{Symbol}\})_{\langle \mathit{Dashes} \mid \mathit{ReservedSym} \rangle} \\
\mathit{FunctionID} &::= \mathit{Ident} \\
\mathit{VariableID} &::= \mathit{Ident} \\
\mathit{LabelID} &::= \mathit{Ident} \\
\mathit{ClassID} &::= \mathit{Ident} \\
\\
\mathit{QTypeConstrID} &::= [\mathit{ModuleID}\ \texttt{.}]\ \mathit{TypeConstrID} \\
\mathit{QDataConstrID} &::= [\mathit{ModuleID}\ \texttt{.}]\ \mathit{DataConstrID} \\
\mathit{QInfixOpID} &::= [\mathit{ModuleID}\ \texttt{.}]\ \mathit{InfixOpID} \\
\mathit{QFunctionID} &::= [\mathit{ModuleID}\ \texttt{.}]\ \mathit{FunctionID} \\
\mathit{QLabelID} &::= [\mathit{ModuleID}\ \texttt{.}]\ \mathit{LabelID} \\
\mathit{QClassID} &::= [\mathit{ModuleID}\ \texttt{.}]\ \mathit{ClassID}
\end{aligned}
$$

Die folgenden Bezeichner werden als Schlüsselwörter erkannt und können daher nicht als reguläre Bezeichner verwendet werden.

$$
\begin{aligned}
\mathit{ReservedID} ::= {} & \texttt{case} \mid \texttt{class} \mid \texttt{data} \mid \texttt{default} \mid \texttt{do} \mid \texttt{else} \mid \texttt{external} \mid \texttt{fcase} \\
\mid {} & \texttt{foreign} \mid \texttt{free} \mid \texttt{if} \mid \texttt{import} \mid \texttt{in} \mid \texttt{infix} \mid \texttt{infixl} \mid \texttt{infixr} \\
\mid {} & \texttt{instance} \mid \texttt{let} \mid \texttt{module} \mid \texttt{newtype} \mid \texttt{of} \mid \texttt{then} \mid \texttt{type} \mid \texttt{where}
\end{aligned}
$$

Es ist zu beachten, dass die Bezeichner `as`, `forall`, `hiding` und `qualified` keine Schlüsselwörter sind. Sie haben nur im Kopf eines Moduls eine besondere Bedeutung und können daher anderswo als herkömmliche Bezeichner verwendet werden.

Die folgenden Symbole haben ebenfalls eine besondere Bedeutung und können nicht als Infix-Operatoren verwendet werden.

$$
\mathit{ReservedSym} ::= \texttt{..} \mid \texttt{:} \mid \texttt{::} \mid \texttt{=} \mid \texttt{\textbackslash} \mid \texttt{|} \mid \texttt{<-} \mid \texttt{->} \mid \texttt{@} \mid \texttt{~} \mid \texttt{=>}
$$

A.2.3. Zahlen- und Buchstabenliterale

Im Kontrast zum Curry Report adaptiert das Front-End sowohl für numerische als auch für Buchstaben- und Zeichenkettenliterale Haskells Schreibweise – um die Möglichkeit erweitert, binäre Zahlenliterale anzugeben.

$$
\begin{aligned}
\mathit{Int} ::= {} & \mathit{Decimal} \\
\mid {} & \texttt{0b}\ \mathit{Binary} \mid \texttt{0B}\ \mathit{Binary}
\end{aligned}
$$

$$
\begin{array}{rcl}
 & | & \texttt{0o}\ \mathit{Octal} \mid \texttt{0O}\ \mathit{Octal} \\
 & | & \texttt{0x}\ \mathit{Hexadecimal} \mid \texttt{0X}\ \mathit{Hexadecimal} \\
\\
\mathit{Float} & ::= & \mathit{Decimal}\ \texttt{.}\ \mathit{Decimal}\ [\mathit{Exponent}] \\
 & | & \mathit{Decimal}\ \mathit{Exponent} \\
\mathit{Exponent} & ::= & (\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \mathit{Decimal} \\
\\
\mathit{Decimal} & ::= & \mathit{Digit}\ \{\mathit{Digit}\} \\
\mathit{Binary} & ::= & \mathit{Binit}\ \{\mathit{Binit}\} \\
\mathit{Octal} & ::= & \mathit{Octit}\ \{\mathit{Octit}\} \\
\mathit{Hexadecimal} & ::= & \mathit{Hexit}\ \{\mathit{Hexit}\} \\
\\
\mathit{Digit} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\mathit{Binit} & ::= & \texttt{0} \mid \texttt{1} \\
\mathit{Octit} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \\
\mathit{Hexit} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
 & | & \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f}
\end{array}
$$

Die Syntax für Buchstaben- und Zeichenkettenliterale lautet wie folgt.

$$
\begin{array}{rcl}
\mathit{Char} & ::= & \texttt{'}\ (\ \mathit{Graphic}_{\langle \texttt{\textbackslash} \rangle} \mid \mathit{Space} \mid \mathit{Escape}_{\langle \texttt{\textbackslash\&} \rangle}\ )\ \texttt{'} \\
\mathit{String} & ::= & \texttt{"}\ \{\ \mathit{Graphic}_{\langle \texttt{"} \mid \texttt{\textbackslash} \rangle} \mid \mathit{Space} \mid \mathit{Escape} \mid \mathit{Gap}\ \}\ \texttt{"} \\
\mathit{Escape} & ::= & \texttt{\textbackslash}\ (\ \mathit{CharEsc} \mid \mathit{AsciiEsc} \mid \mathit{Decimal} \mid \texttt{o}\ \mathit{Octal} \mid \texttt{x}\ \mathit{Hexadecimal}\ ) \\
\mathit{CharEsc} & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{f} \mid \texttt{n} \mid \texttt{r} \mid \texttt{t} \mid \texttt{v} \mid \texttt{\textbackslash} \mid \texttt{"} \mid \texttt{'} \mid \texttt{\&} \\
\mathit{AsciiAsc} & ::= & \texttt{\^{}}\ \mathit{Cntrl} \mid \texttt{NUL} \mid \texttt{SOH} \mid \texttt{STX} \mid \texttt{ETX} \mid \texttt{EOT} \mid \texttt{ENQ} \mid \texttt{ACK} \\
 & | & \texttt{BEL} \mid \texttt{BS} \mid \texttt{HT} \mid \texttt{LF} \mid \texttt{VT} \mid \texttt{FF} \mid \texttt{CR} \mid \texttt{SO} \mid \texttt{SI} \mid \texttt{DLE} \\
 & | & \texttt{DC1} \mid \texttt{DC2} \mid \texttt{DC3} \mid \texttt{DC4} \mid \texttt{NAK} \mid \texttt{SYN} \mid \texttt{ETB} \mid \texttt{CAN} \\
 & | & \texttt{EM} \mid \texttt{SUB} \mid \texttt{ESC} \mid \texttt{FS} \mid \texttt{GS} \mid \texttt{RS} \mid \texttt{US} \mid \texttt{SP} \mid \texttt{DEL} \\
\mathit{Cntrl} & ::= & \texttt{A} \mid \ldots \mid \texttt{Z} \mid \texttt{@} \mid \texttt{[} \mid \texttt{\textbackslash} \mid \texttt{]} \mid \texttt{\^{}} \mid \texttt{\_} \\
\mathit{Gap} & ::= & \texttt{\textbackslash}\ \mathit{WhiteChar}\ \{\mathit{WhiteChar}\}\ \texttt{\textbackslash} \\
\mathit{Graphic} & ::= & \textit{any graphical character} \\
\mathit{WhiteChar} & ::= & \textit{any whitespace character}
\end{array}
$$

A.3. Layout

Ähnlich zu Haskell können Layout-Informationen dazu genutzt werden, um die Struktur von Blöcken zu definieren. Zu diesem Zweck definieren wir die Einrückung eines Symbols als diejenige Spalte, die den Beginn des Symbols anzeigt. Die Einrückung einer Zeile ist die Einrückung ihres ersten Symbols.[1]

Die Layout-Regel (oder „Off-Side-Regel“) gilt für Listen syntaktischer Entitäten nach den Schlüsselwörtern `let`, `where`, `do` oder `of`. In der nachfolgenden kontextfreien Modulsyntax sind diese Listen von geschweiften Klammern umschlossen (`{ }`) und die einzelnen Entitäten durch Semikola (`;`) getrennt. Anstelle der geschweiften Klammern und Semikola kann auch Einrückung verwendet werden, um diese Listen anzugeben: Die Einrückung

[1] Um die exakte Spaltennummer zu bestimmen, nehmen wir eine Schrift fester Breite mit Tabstopps in jeder achten Spalte an.

einer Liste syntaktischer Entitäten nach einem `let`, `where`, `do` oder `of` ist die Einrückung des nächsten Symbols, welches dem `let`, `where`, `do` bzw. `of` folgt. Jedes Element dieser Liste beginnt mit derselben Einrückung wie die Liste. Zeilen, welche ausschließlich aus Leerräumen bestehen oder eine größere Einrückung als die der Liste besitzen, setzen das Element in der vorherigen Zeile fort. Zeilen mit einer geringen Einrückung als die der Liste terminieren diese gänzlich. Ferner werden Listen, die mit `let` beginnen, durch das Schlüsselwort `in` beendet. Somit kann der in der kontextfreien Grammatik gültige Satz

```
f x = h x where { g y = y + 1 ; h z = (g z) * 2 }
```

unter Anwendung der Layout-Regeln als

```
f x = h x
  where g y = y + 1
        h z = (g z) * 2
```

oder auch als

```
f x = h x  where
  g y = y + 1
  h z = (g z)
          * 2
```

geschrieben werden. Um die Einrückung von Top-Level-Deklarationen zu vermeiden, wird angenommen, dass das Schlüsselwort `module` sowie das End-Of-File-Token in der Spalte 0 beginnen.

A.4. Modulsyntax

$$
\begin{aligned}
\mathit{Module} &::= \texttt{module}\ \mathit{ModuleID}\ [\mathit{Exports}]\ \texttt{where}\ \mathit{Block} \\
&\mid \mathit{Block} \\
\mathit{Block} &::= \texttt{\{}\ [\mathit{ImportDecls}\ \texttt{;}]\ \mathit{BlockDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{BlockDecl}_n\ \texttt{\}} && (n \geq 0) \\
\mathit{Exports} &::= \texttt{(}\ \mathit{Export}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Export}_n\ \texttt{)} && (n \geq 0) \\
\mathit{Export} &::= \mathit{QFunction} \\
&\mid \mathit{QTypeConstrID}\ [\texttt{(}\ \mathit{ConsLabel}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{ConsLabel}_n\ \texttt{)}] && (n \geq 0) \\
&\mid \mathit{QTypeConstrID}\ \texttt{(..)} \\
&\mid \mathit{QClassID}\ [\texttt{(}\ \mathit{Function}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Function}_n\ \texttt{)}] && (n \geq 0) \\
&\mid \mathit{QClassID}\ \texttt{(..)} \\
&\mid \texttt{module}\ \mathit{ModuleID} \\
\mathit{ConsLabel} &::= \mathit{DataConstr} \mid \mathit{Label} \\
\mathit{ImportDecls} &::= \mathit{ImportDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{ImportDecl}_n && (n \geq 1) \\
\mathit{ImportDecl} &::= \texttt{import}\ [\texttt{qualified}]\ \mathit{ModuleID}\ [\texttt{as}\ \mathit{ModuleID}]\ [\mathit{ImportSpec}] \\
\mathit{ImportSpec} &::= \texttt{(}\ \mathit{Import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Import}_n\ \texttt{)} && (n \geq 0) \\
&\mid \texttt{hiding}\ \texttt{(}\ \mathit{Import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Import}_n\ \texttt{)} && (n \geq 0)
\end{aligned}
$$

$$
\begin{array}{rcll}
\mathit{Import} & ::= & \mathit{Function} & \\
& | & \mathit{TypeConstrID}\ [\texttt{(}\ \mathit{ConsLabel}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{ConsLabel}_n\ \texttt{)}] & (n \geq 0) \\
& | & \mathit{TypeConstrID}\ \texttt{(..)} & \\
& | & \mathit{ClassID}\ [\texttt{(}\ \mathit{Function}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Function}_n\ \texttt{)}] & (n \geq 0) \\
& | & \mathit{ClassID}\ \texttt{(..)} & \\
\\
\mathit{BlockDecl} & ::= & \mathit{TypeSynDecl} & \\
& | & \mathit{DataDecl} & \\
& | & \mathit{NewtypeDecl} & \\
& | & \mathit{FixityDecl} & \\
& | & \mathit{FunctionDecl} & \\
& | & \mathit{DefaultDecl} & \\
& | & \mathit{ClassDecl} & \\
& | & \mathit{InstanceDecl} & \\
\\
\mathit{TypeSynDecl} & ::= & \texttt{type}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{TypeExpr} & \\
\mathit{SimpleType} & ::= & \mathit{TypeConstrID}\ \mathit{TypeVarID}_1\ \ldots\ \mathit{TypeVarID}_n & (n \geq 0) \\
\\
\mathit{DataDecl} & ::= & \texttt{data}\ \mathit{SimpleType} & (\textit{externer Datentyp}) \\
& | & \texttt{data}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{ConstrDecls} & \\
\mathit{ConstrDecls} & ::= & \mathit{ConstrDecl}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \mathit{ConstrDecl}_n & (n \geq 1) \\
\mathit{ConstrDecl} & ::= & [\mathit{ExistVars}]\ [\mathit{Context}\ \texttt{=>}]\ \mathit{ConDecl} & \\
\mathit{ExistVars} & ::= & \texttt{forall}\ \mathit{ExistVarID}_1\ \ldots\ \mathit{ExistVarID}_n\ \texttt{.} & (n \geq 1) \\
\mathit{ConDecl} & ::= & \mathit{DataConstr}\ \mathit{SimpleTypeExpr}_1\ \ldots\ \mathit{SimpleTypeExpr}_n & (n \geq 0) \\
& | & \mathit{TypeAppExpr}\ \mathit{ConOp}\ \mathit{TypeAppExpr} & (\textit{Infix-Datenkonstruktor}) \\
& | & \mathit{DataConstr}\ \texttt{\{}\ \mathit{FieldDecl}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FieldDecl}_n\ \texttt{\}} & (n \geq 0) \\
\mathit{FieldDecl} & ::= & \mathit{Label}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Label}_n\ \texttt{::}\ \mathit{TypeExpr} & (n \geq 1) \\
\\
\mathit{NewtypeDecl} & ::= & \texttt{newtype}\ \mathit{SimpleType}\ \texttt{=}\ \mathit{NewConstrDecl} & \\
\mathit{NewConstrDecl} & ::= & \mathit{DataConstr}\ \mathit{SimpleTypeExpr} & \\
& | & \mathit{DataConstr}\ \texttt{\{}\ \mathit{Label}\ \texttt{::}\ \mathit{TypeExpr}\ \texttt{\}} & \\
\\
\mathit{QualTypeExpr} & ::= & [\mathit{Context}\ \texttt{=>}]\ \mathit{TypeExpr} & \\
\mathit{Context} & ::= & \mathit{Constraint} & \\
& | & \texttt{(}\ \mathit{Constraint}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Constraint}_n\ \texttt{)} & (n \geq 0) \\
\mathit{Constraint} & ::= & \mathit{QClassID}\ \mathit{ClassVarID} & \\
& | & \mathit{QClassID}\ \texttt{(}\ \mathit{ClassVarID}\ \mathit{SimpleTypeExprs}\ \texttt{)} & \\
\mathit{SimpleTypeExprs} & ::= & \mathit{SimpleTypeExpr}_1\ \ldots\ \mathit{SimpleTypeExpr}_n & (n \geq 1) \\
\mathit{TypeExpr} & ::= & \mathit{TypeAppExpr}\ [\texttt{->}\ \mathit{TypeExpr}] & \\
\mathit{TypeAppExpr} & ::= & [\mathit{TypeAppExpr}]\ \mathit{SimpleTypeExpr} & (\textit{Typapplikation}) \\
\mathit{SimpleTypeExpr} & ::= & \mathit{TypeVarID} & \\
& | & \mathit{GTypeConstr} & \\
& | & \texttt{(}\ \mathit{TypeExpr}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{TypeExpr}_n\ \texttt{)} & (\textit{Tupeltyp},\ n \geq 2) \\
& | & \texttt{[}\ \mathit{TypeExpr}\ \texttt{]} & (\textit{Listentyp}) \\
& | & \texttt{(}\ \mathit{TypeExpr}\ \texttt{)} & (\textit{geklammerter Typ})
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{GTypeConstr} & ::= & \texttt{()} & (\mathit{Unit\text{-}Typ}) \\
& | & \texttt{[]} & (\mathit{Listentypkonstruktor}) \\
& | & \texttt{(->)} & (\mathit{Funktionstypkonstruktor}) \\
& | & \texttt{(,\{,\})} & (\mathit{Tupeltypkonstruktor}) \\
& | & \mathit{QTypeConstrID} & \\
\\
\mathit{FixityDecl} & ::= & \mathit{Fixity}\ [\mathit{Int}]\ \mathit{Op}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Op}_n & (n \geq 1) \\
\mathit{Fixity} & ::= & \texttt{infixl} \mid \texttt{infixr} \mid \texttt{infix} & \\
\\
\mathit{DefaultDecl} & ::= & \texttt{default (}\ \mathit{TypeExpr}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{TypeExpr}_n\ \texttt{)} & (n \geq 0) \\
\\
\mathit{ClassDecl} & ::= & \texttt{class}\ [\mathit{SimpleContext}\ \texttt{=>}]\ \mathit{ClassID}\ \mathit{ClassVarID}\ [\texttt{where}\ \mathit{ClsDecls}] & \\
\mathit{ClsDecls} & ::= & \texttt{\{}\ \mathit{ClsDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{ClsDecl}_n\ \texttt{\}} & (n \geq 0) \\
\mathit{ClsDecl} & ::= & \mathit{Signature} & \\
& | & \mathit{Equat} & \\
\mathit{SimpleContext} & ::= & \mathit{SimpleConstraint} & \\
& | & \texttt{(}\ \mathit{SimpleConstraint}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{SimpleConstraint}_n\ \texttt{)} & (n \geq 0) \\
\mathit{SimpleConstraint} & ::= & \mathit{QClassID}\ \mathit{ClassVarID} & \\
\\
\mathit{InstanceDecl} & ::= & \texttt{instance}\ [\mathit{SimpleContext}\ \texttt{=>}]\ \mathit{QClassID}\ \mathit{InstType}\ [\texttt{where}\ \mathit{InstDecls}] & \\
\mathit{InstDecls} & ::= & \texttt{\{}\ \mathit{InstDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{InstDecl}_n\ \texttt{\}} & (n \geq 0) \\
\mathit{InstDecl} & ::= & \mathit{Equat} & \\
\mathit{InstType} & ::= & \mathit{GTypeConstr} & \\
& | & \texttt{(}\ \mathit{GTypeConstr}\ \mathit{ClassVarID}_1\ \ldots\ \mathit{ClassVarID}_n\ \texttt{)} & (n \geq 0) \\
& | & \texttt{(}\ \mathit{ClassVarID}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{ClassVarID}_n\ \texttt{)} & (n \geq 2) \\
& | & \texttt{[}\ \mathit{ClassVarID}\ \texttt{]} & \\
& | & \texttt{(}\ \mathit{ClassVarID}_1\ \texttt{->}\ \mathit{ClassVarID}_2\ \texttt{)} & \\
\\
\mathit{FunctionDecl} & ::= & \mathit{Signature} \mid \mathit{ExternalDecl} \mid \mathit{Equation} & \\
\mathit{Signature} & ::= & \mathit{Functions}\ \texttt{::}\ \mathit{QualTypeExpr} & \\
\mathit{ExternalDecl} & ::= & \mathit{Functions}\ \texttt{external} & (\textit{extern definierte Funktion}) \\
\mathit{Functions} & ::= & \mathit{Function}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Function}_n & (n \geq 1) \\
\mathit{Equation} & ::= & \mathit{FunLhs}\ \mathit{Rhs} & \\
\mathit{FunLhs} & ::= & \mathit{Function}\ \mathit{SimplePat}_1\ \ldots\ \mathit{SimplePat}_n & (n \geq 0) \\
& | & \mathit{ConsPattern}\ \mathit{FunOp}\ \mathit{ConsPattern} & \\
& | & \texttt{(}\ \mathit{FunLhs}\ \texttt{)}\ \mathit{SimplePats} & \\
\\
\mathit{Rhs} & ::= & \texttt{=}\ \mathit{Expr}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
& | & \mathit{CondExprs}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
\mathit{CondExprs} & ::= & \texttt{|}\ \mathit{InfixExpr}\ \texttt{=}\ \mathit{Expr}\ [\mathit{CondExprs}] & \\
\\
\mathit{LocalDecls} & ::= & \texttt{\{}\ \mathit{LocalDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{LocalDecl}_n\ \texttt{\}} & (n \geq 0) \\
\mathit{LocalDecl} & ::= & \mathit{FunctionDecl} & \\
& | & \mathit{PatternDecl} & \\
& | & \mathit{Variable}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Variable}_n\ \texttt{free} & (n \geq 1) \\
& | & \mathit{FixityDecl} & \\
\mathit{PatternDecl} & ::= & \mathit{Pattern}\ \mathit{Rhs} &
\end{array}
$$

Pattern	::=	*ConsPattern* [*QConOp Pattern*]	(*Infix-Konstruktorenmuster*)
ConsPattern	::=	*GDataConstr SimplePats*	(*Konstruktorenmuster*)
	|	- (*Int* | *Float*)	(*negatives Muster*)
	|	*SimplePat*	
SimplePats	::=	$SimplePat_1$ ... $SimplePat_n$	($n \geq 1$)
SimplePat	::=	*Variable*	
	|	_	(*Platzhalter*)
	|	*GDataConstr*	(*Konstruktor*)
	|	*Literal*	(*Literal*)
	|	(*Pattern*)	(*geklammertes Muster*)
	|	($Pattern_1$, ... , $Pattern_n$)	(*Tupelmuster*, $n \geq 2$)
	|	[$Pattern_1$, ... , $Pattern_n$]	(*Listenmuster*, $n \geq 1$)
	|	*Variable* @ *SimplePat*	(*As-Muster*)
	|	~ *SimplePat*	(*unwiderlegbares Muster*)
	|	(*QFunction SimplePats*)	(*funktionales Muster*)
	|	(*ConsPattern QFunOp Pattern*)	(*funktionales Infix-Muster*)
	|	*QDataConstr* { *FieldPats* }	(*Feldmuster*)
FieldPats	::=	$FieldPat_1$, ... , $FieldPat_n$	($n \geq 0$)
FieldPat	::=	*QLabel* = *Pattern*	
Expr	::=	*InfixExpr* :: *QualTypeExpr*	(*Ausdruck mit Typsignatur*)
	|	*InfixExpr*	
InfixExpr	::=	*NoOpExpr QOp InfixExpr*	(*Infix-Operator-Applikation*)
	|	- *InfixExpr*	(*einstelliges Minus*)
	|	*NoOpExpr*	
NoOpExpr	::=	\ *SimplePats* -> *Expr*	(*Lambda-Ausdruck*)
	|	let *LocalDecls* in *Expr*	(*Let-Ausdruck*)
	|	if *Expr* then *Expr* else *Expr*	(*Konditional*)
	|	case *Expr* of { Alt_1 ; ... ; Alt_n }	(*Case-Ausdruck*, $n \geq 1$)
	|	fcase *Expr* of { Alt_1 ; ... ; Alt_n }	(*Fcase-Ausdruck*, $n \geq 1$)
	|	do { $Stmt_1$; ... ; $Stmt_n$; *Expr* }	(*Do-Ausdruck*, $n \geq 0$)
	|	*FuncExpr*	
FuncExpr	::=	[*FuncExpr*] *BasicExpr*	(*Applikation*)
BasicExpr	::=	*Variable*	(*Variable*)
	|	_	(*anonyme freie Variable*)
	|	*QFunction*	(*qualifizierte Funktion*)
	|	*GDataConstr*	(*allgemeiner Konstruktor*)
	|	*Literal*	(*Literal*)
	|	(*Expr*)	(*geklammerter Ausdruck*)
	|	($Expr_1$, ... , $Expr_n$)	(*Tupel*, $n \geq 2$)
	|	[$Expr_1$, ... , $Expr_n$]	(*endliche Liste*, $n \geq 1$)
	|	[*Expr* [, *Expr*] .. [*Expr*]]	(*arithmetische Sequenz*)
	|	[*Expr* | $Qual_1$, ... , $Qual_n$]	(*List-Comprehension*, $n \geq 1$)
	|	(*InfixExpr QOp*)	(*Left-Section*)

$$
\begin{array}{rcll}
 & | & \texttt{(}\ \mathit{QOp}_{\langle \texttt{-} \rangle}\ \mathit{InfixExpr}\ \texttt{)} & (\mathit{Right\text{-}Section}) \\
 & | & \mathit{QDataConstr}\ \texttt{\{}\ \mathit{FBinds0}\ \texttt{\}} & (\mathit{Record\text{-}Konstruktion}) \\
 & | & \mathit{BasicExpr}_{\langle \mathit{QDataConstr} \rangle}\ \texttt{\{}\ \mathit{FBinds1}\ \texttt{\}} & (\mathit{Record\text{-}Update}) \\
\\
\mathit{Alt} & ::= & \mathit{Pattern}\ \texttt{->}\ \mathit{Expr}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
 & | & \mathit{Pattern}\ \mathit{GdAlts}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
\mathit{GdAlts} & ::= & \texttt{|}\ \mathit{InfixExpr}\ \texttt{->}\ \mathit{Expr}\ [\mathit{GdAlts}] & \\
\\
\mathit{FBinds0} & ::= & \mathit{FBind}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FBind}_n & (n \geq 0) \\
\mathit{FBinds1} & ::= & \mathit{FBind}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{FBind}_n & (n \geq 1) \\
\mathit{FBind} & ::= & \mathit{QLabel}\ \texttt{=}\ \mathit{Expr} & \\
\\
\mathit{Qual} & ::= & \mathit{Pattern}\ \texttt{<-}\ \mathit{Expr} & (\mathit{Generator}) \\
 & | & \texttt{let}\ \mathit{LocalDecls} & (\mathit{lokale\ Deklarationen}) \\
 & | & \mathit{Expr} & (\mathit{Wächter}) \\
\\
\mathit{Stmt} & ::= & \mathit{Pattern}\ \texttt{<-}\ \mathit{Expr} & \\
 & | & \texttt{let}\ \mathit{LocalDecls} & \\
 & | & \mathit{Expr} & \\
\\
\mathit{Literal} & ::= & \mathit{Int} \mid \mathit{Float} \mid \mathit{Char} \mid \mathit{String} & \\
\\
\mathit{GDataConstr} & ::= & \texttt{()} & (\mathit{Unit}) \\
 & | & \texttt{[]} & (\mathit{leere\ Liste}) \\
 & | & \texttt{(,\{,\})} & (\mathit{Tupel}) \\
 & | & \mathit{QDataConstr} & \\
\\
\mathit{Variable} & ::= & \mathit{VariableID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\mathit{Variable}) \\
\mathit{Function} & ::= & \mathit{FunctionID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\mathit{Funktion}) \\
\mathit{QFunction} & ::= & \mathit{QFunctionID} \mid \texttt{(}\ \mathit{QInfixOpID}\ \texttt{)} & (\mathit{qualifizierte\ Funktion}) \\
\mathit{DataConstr} & ::= & \mathit{DataConstrID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\mathit{Konstruktor}) \\
\mathit{QDataConstr} & ::= & \mathit{QDataConstrID} \mid \texttt{(}\ \mathit{QInfixOpID}\ \texttt{)} & (\mathit{qualifizierter\ Konstruktor}) \\
\mathit{Label} & ::= & \mathit{LabelID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\mathit{Feldbezeichner}) \\
\mathit{QLabel} & ::= & \mathit{QLabelID} \mid \texttt{(}\ \mathit{QInfixOpID}\ \texttt{)} & (\mathit{qualifizierter\ Feldbezeichner}) \\
\\
\mathit{VarOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{VariableID}\ \texttt{`} & (\mathit{Variablenoperator}) \\
\mathit{FunOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{FunctionID}\ \texttt{`} & (\mathit{Funktionsoperator}) \\
\mathit{QFunOp} & ::= & \mathit{QInfixOpID} \mid \texttt{`}\ \mathit{QFunctionID}\ \texttt{`} & (\mathit{qualifizierter\ Funktionsoperator}) \\
\mathit{ConOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{DataConstrID}\ \texttt{`} & (\mathit{Konstruktor\text{-}Operator}) \\
\mathit{QConOp} & ::= & \mathit{GConSym} \mid \texttt{`}\ \mathit{QDataConstrID}\ \texttt{`} & (\mathit{qual.\ Konstruktor\text{-}Operator}) \\
\mathit{LabelOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{LabelID}\ \texttt{`} & (\mathit{Feldbezeichner\text{-}Operator}) \\
\mathit{QLabelOp} & ::= & \mathit{QInfixOpID} \mid \texttt{`}\ \mathit{QLabelID}\ \texttt{`} & (\mathit{qual.\ Feldbezeichner\text{-}Operator}) \\
\\
\mathit{Op} & ::= & \mathit{FunOp} \mid \mathit{ConOp} \mid \mathit{LabelOp} & (\mathit{Operator}) \\
\mathit{QOp} & ::= & \mathit{VarOp} \mid \mathit{QFunOp} \mid \mathit{QConOp} \mid \mathit{QLabelOp} & (\mathit{qualifizierter\ Operator}) \\
\mathit{GConSym} & ::= & \texttt{:} \mid \mathit{QInfixOpID} & (\mathit{allgemeines\ Konstruktor\text{-}Symbol})
\end{array}
$$

A.5. Schnittstellensyntax

$$
\begin{aligned}
\mathit{Interface} &::= \texttt{interface}\ \mathit{ModuleID}\ \texttt{where}\ \mathit{IBlock} \\
\mathit{IBlock} &::= \texttt{\{}\ [\mathit{IImportDecls}\ \texttt{;}]\ \mathit{IBlockDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{IBlockDecl}_n\ \texttt{\}} && (n \geq 0) \\
\mathit{IImportDecls} &::= \mathit{IImportDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{IImportDecl}_n && (n \geq 1) \\
\mathit{IImportDecl} &::= \texttt{import}\ \mathit{ModuleID} \\
\mathit{IBlockDecl} &::= \mathit{ITypeSynDecl} \\
&\mid \mathit{HidingDataDecl} \\
&\mid \mathit{IDataDecl} \\
&\mid \mathit{INewtypeDecl} \\
&\mid \mathit{IFixityDecl} \\
&\mid \mathit{IFunctionDecl} \\
&\mid \mathit{HidingClassDecl} \\
&\mid \mathit{IClassDecl} \\
&\mid \mathit{IInstanceDecl} \\
\mathit{ITypeSynDecl} &::= \texttt{type}\ \mathit{QSimpleType}\ \texttt{=}\ \mathit{TypeExpr} \\
\mathit{QSimpleType} &::= \mathit{QTypeConstr}\ \mathit{TypeVarID}_1\ \ldots\ \mathit{TypeVarID}_n && (n \geq 0) \\
\mathit{QTypeConstr} &::= \mathit{QTypeConstrID} \\
&\mid \texttt{(}\ \mathit{QTypeConstrID}\ \texttt{::}\ \mathit{KindExpr}\ \texttt{)} \\
\mathit{KindExpr} &::= \mathit{SimpleKindExpr}\ [\texttt{->}\ \mathit{KindExpr}] \\
\mathit{SimpleKindExpr} &::= \texttt{*} \\
&\mid \texttt{(}\ \mathit{KindExpr}\ \texttt{)} && (\textit{geklammerte Sorte}) \\
\mathit{HidingDataDecl} &::= \texttt{hiding data}\ \mathit{QSimpleType} \\
\mathit{IDataDecl} &::= \texttt{data}\ \mathit{QSimpleType} \\
&\mid \texttt{data}\ \mathit{QSimpleType}\ \texttt{=}\ \mathit{ConstrDecls}\ [\mathit{HidingPragma}] \\
\mathit{HidingPragma} &::= \texttt{\{-\# HIDING}\ \mathit{ConsLabel}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{ConsLabel}_n\ \texttt{\#-\}} && (n \geq 0) \\
\mathit{INewtypeDecl} &::= \texttt{newtype}\ \mathit{QSimpleType}\ \texttt{=}\ \mathit{NewConstrDecl}\ [\mathit{HidingPragma}] \\
\mathit{IFixityDecl} &::= \mathit{Fixity}\ \mathit{Int}\ \mathit{IOp} \\
\mathit{IOp} &::= \mathit{QFunOp} \mid \mathit{QConOp} \mid \mathit{QLabelOp} \\
\mathit{IFunctionDecl} &::= \mathit{QFunction}\ [\mathit{MethodPragma}]\ [\mathit{Int}]\ \texttt{::}\ \mathit{QualTypeExpr} \\
\mathit{MethodPragma} &::= \texttt{\{-\# METHOD}\ \mathit{ClassVarID}\ \texttt{\#-\}} \\
\mathit{HidingClassDecl} &::= \texttt{hiding class}\ \mathit{ClassHead} \\
\mathit{ClassHead} &::= [\mathit{SimpleContext}\ \texttt{=>}]\ \mathit{QClass}\ \mathit{ClassVarID} \\
\mathit{QClass} &::= \mathit{QClassID} \\
&\mid \texttt{(}\ \mathit{QClassID}\ \texttt{::}\ \mathit{KindExpr}\ \texttt{)} \\
\mathit{IClassDecl} &::= \texttt{class}\ \mathit{ClassHead}\ \texttt{\{}\ \mathit{ClassMethods}\ \texttt{\}}\ [\mathit{ClassHidingPragma}] \\
\mathit{ClassMethods} &::= \texttt{\{}\ \mathit{ClassMethod}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{ClassMethod}_n\ \texttt{\}} && (n \geq 0)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{ClassMethod} &::= \mathit{Function}\ [\mathit{Int}]\ \texttt{::}\ \mathit{QualTypeExpr} \\
\mathit{ClassHidingPragma} &::= \texttt{\{-\#}\ \texttt{HIDING}\ \mathit{Function}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Function}_n\ \texttt{\#-\}} && (n \geq 0) \\
\\
\mathit{IInstanceDecl} &::= \texttt{instance}\ \mathit{InstanceHead}\ \texttt{\{}\ \mathit{InstanceMethods}\ \texttt{\}}\ [\mathit{ModulePragma}] \\
\mathit{InstanceHead} &::= [\mathit{SimpleContext}\ \texttt{=>}]\ \mathit{QClassID}\ \mathit{InstanceType} \\
\mathit{InstanceMethods} &::= \texttt{\{}\ \mathit{InstanceMethod}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{InstanceMethod}_n\ \texttt{\}} && (n \geq 0) \\
\mathit{InstanceMethod} &::= \mathit{Function}\ [\mathit{Int}] \\
\mathit{ModulePragma} &::= \texttt{\{-\#}\ \texttt{MODULE}\ \mathit{ModuleID}\ \texttt{\#-\}}
\end{aligned}
$$

B. Standardbibliothek

Nachfolgend geben wir die modifizierte Standardbibliothek an, wobei wir uns aufgrund ihres Umfangs nur auf die vorgenommenen Änderungen beschränken.

```
module Prelude
  ( Eq(..)
  , elem, notElem, lookup
  , Ord(..)
  , Show(..), print, shows, showChar, showString, showParen
  , Read (..)
  , Bounded (..), Enum (..), boundedEnumFrom, boundedEnumFromThen
  , asTypeOf
  , Num(..), Fractional(..), Real(..), Integral(..)
  , Monad(..)
  , Functor(..)
  , Bool (..) , Char (..) , Int (..) , Float (..), String , Ordering (..)
  , Success, Maybe (..), Either (..), IO (..), IOError (..)
  , (.), id, const, curry, uncurry, flip, until, seq, ensureNotFree
  , ensureSpine, ($), ($!), ($!!), ($#), ($##), error
  , failed, (&&), (||), not, otherwise, if_then_else
  , fst, snd, head, tail, null, (++), length, (!!), map, foldl, foldl1
  , foldr, foldr1, filter, zip, zip3, zipWith, zipWith3, unzip, unzip3
  , concat, concatMap, iterate, repeat, replicate, take, drop, splitAt
  , takeWhile, dropWhile, span, break, lines, unlines, words, unwords
  , reverse, and, or, any, all
  , ord, chr, (=:=), success, (&), (&>), maybe
  , either, (>>=), return, (>>), done, putChar, getChar, readFile
  , writeFile, appendFile
  , putStr, putStrLn, getLine, userError, ioError, showError
  , catch, doSolve, sequenceIO, sequenceIO_, mapIO
  , mapIO_, (?), unknown
  , when, unless, forIO, forIO_, liftIO, foldIO
  , normalForm, groundNormalForm, apply, cond, (=:<=), (=:<<=)
  , enumFrom_, enumFromTo_, enumFromThen_, enumFromThenTo_, negate_
  , negateFloat, letrec
  ) where
```

```
eqChar :: Char -> Char -> Bool
eqChar x y = (prim_eqChar $# y) $# x

prim_eqChar :: Char -> Char -> Bool
prim_eqChar external

eqInt :: Int -> Int -> Bool
eqInt x y = (prim_eqInt $# y) $# x

prim_eqInt :: Int -> Int -> Bool
prim_eqInt external

eqFloat :: Float -> Float -> Bool
eqFloat x y = (prim_eqFloat $# y) $# x

prim_eqFloat :: Float -> Float -> Bool
prim_eqFloat external

ltEqChar :: Char -> Char -> Bool
ltEqChar x y = (prim_ltEqChar $# y) $# x

prim_ltEqChar :: Char -> Char -> Bool
prim_ltEqChar external

ltEqInt :: Int -> Int -> Bool
ltEqInt x y = (prim_ltEqInt $# y) $# x

prim_ltEqInt :: Int -> Int -> Bool
prim_ltEqInt external

ltEqFloat :: Float -> Float -> Bool
ltEqFloat x y = (prim_ltEqFloat $# y) $# x

prim_ltEqFloat :: Float -> Float -> Bool
prim_ltEqFloat external

elem :: Eq a => a -> [a] -> Bool
elem x = any (x ==)

notElem :: Eq a => a -> [a] -> Bool
notElem x = all (x /=)

lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

```
lookup _ []       = Nothing
lookup k ((x,y):xys)
      | k==x      = Just y
      | otherwise = lookup k xys

enumFrom_ :: Int -> [Int]
enumFrom_ n = n : enumFrom_ (n+1)

enumFromThen_ :: Int -> Int -> [Int]
enumFromThen_ n1 n2 = iterate ((n2-n1)+) n1

enumFromTo_ :: Int -> Int -> [Int]
enumFromTo_ n m = if n>m then [] else n : enumFromTo_ (n+1) m

enumFromThenTo_ :: Int -> Int -> Int -> [Int]
enumFromThenTo_ n1 n2 m = takeWhile p (enumFromThen_ n1 n2)
 where
  p x | n2 >= n1  = (x <= m)
      | otherwise = (x >= m)

(+$)   :: Int -> Int -> Int
x +$ y = (prim_Int_plus $# y) $# x

prim_Int_plus :: Int -> Int -> Int
prim_Int_plus external

(-$)   :: Int -> Int -> Int
x -$ y = (prim_Int_minus $# y) $# x

prim_Int_minus :: Int -> Int -> Int
prim_Int_minus external

(*$)   :: Int -> Int -> Int
x *$ y = (prim_Int_times $# y) $# x

prim_Int_times :: Int -> Int -> Int
prim_Int_times external

div_   :: Int -> Int -> Int
x `div_` y = (prim_Int_div $# y) $# x

prim_Int_div :: Int -> Int -> Int
prim_Int_div external
```

```
mod_   :: Int -> Int -> Int
x `mod_` y = (prim_Int_mod $# y) $# x

prim_Int_mod :: Int -> Int -> Int
prim_Int_mod external

divMod_ :: Int -> Int -> (Int, Int)
divMod_ x y = (x `div` y, x `mod` y)

quot_ :: Int -> Int -> Int
x `quot_` y = (prim_Int_quot $# y) $# x

prim_Int_quot :: Int -> Int -> Int
prim_Int_quot external

rem_ :: Int -> Int -> Int
x `rem_` y = (prim_Int_rem $# y) $# x

prim_Int_rem :: Int -> Int -> Int
prim_Int_rem external

quotRem_ :: Int -> Int -> (Int, Int)
quotRem_ x y = (x `quot` y, x `rem` y)

negate_ :: Int -> Int
negate_ x = 0 - x

negateFloat :: Float -> Float
negateFloat x = prim_negateFloat $# x

prim_negateFloat :: Float -> Float
prim_negateFloat external

(>>=$)             :: IO a -> (a -> IO b) -> IO b
(>>=$) external

returnIO            :: a -> IO a
returnIO external

(>>$) :: IO _ -> IO b -> IO b
a >>$ b = a >>=$ (\_ -> b)
```

```
show_    :: _ -> String
show_ x = prim_show $## x

prim_show    :: _ -> String
prim_show external

print :: Show a => a -> IO ()
print t = putStrLn (show t)

class Eq a where
  (==), (/=) :: a -> a -> Bool

  x == y = not (x /= y)
  x /= y = not (x == y)

instance Eq Ordering where
  LT == LT = True
  LT == EQ = False
  LT == GT = False
  EQ == LT = False
  EQ == EQ = True
  EQ == GT = False
  GT == LT = False
  GT == EQ = False
  GT == GT = True

instance Eq Bool where
  False == False  = True
  False == True   = False
  True  == False  = False
  True  == True   = True

instance Eq Char where
  c == c' = c `eqChar` c'

instance Eq Int where
  i == i' = i `eqInt` i'

instance Eq Float where
  f == f' = f `eqFloat` f'

instance Eq a => Eq [a] where
  []    == []    = True
```

```
    []    == (_:_) = False
    (_:_) == []    = False
    (x:xs) == (y:ys) = x == y && xs == ys

  instance Eq () where
    () == () = True

  instance (Eq a, Eq b) => Eq (a, b) where
    (a, b) == (a', b') = a == a' && b == b'

  instance (Eq a, Eq b, Eq c) => Eq (a, b, c) where
    (a, b, c) == (a', b', c') = a == a' && b == b' && c == c'

  instance (Eq a, Eq b, Eq c, Eq d) => Eq (a, b, c, d) where
    (a, b, c, d) == (a', b', c', d') =
      a == a' && b == b' && c == c' && d == d'

  instance (Eq a, Eq b, Eq c, Eq d, Eq e)
    => Eq (a, b, c, d, e) where
    (a, b, c, d, e) == (a', b', c', d', e') =
      a == a' && b == b' && c == c' && d == d' && e == e'

  instance (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f)
    => Eq (a, b, c, d, e, f) where
    (a, b, c, d, e, f) == (a', b', c', d', e', f') =
      a == a' && b == b' && c == c' && d == d' && e == e' &&
      f == f'

  instance (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g)
    => Eq (a, b, c, d, e, f, g) where
    (a, b, c, d, e, f, g) == (a', b', c', d', e', f', g') =
      a == a' && b == b' && c == c' && d == d' && e == e' &&
      f == f' && g == g'

  instance Eq a => Eq (Maybe a) where
    Nothing == Nothing = True
    Just _  == Nothing = False
    Nothing == Just _  = False
    Just x  == Just y  = x == y

  instance (Eq a, Eq b) => Eq (Either a b) where
    Left x  == Left y  = x == y
    Left _  == Right _ = False
```

```
  Right _ == Left _  = False
  Right x == Right y = x == y

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<=) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (<)  :: a -> a -> Bool
  (>)  :: a -> a -> Bool

  min :: a -> a -> a
  max :: a -> a -> a

  x < y = x <= y && x /= y
  x > y = not (x <= y)
  x >= y = y <= x
  x <= y = compare x y == EQ || compare x y == LT

  compare x y | x == y = EQ
              | x <= y = LT
              | otherwise = GT

  min x y | x <= y    = x
          | otherwise = y

  max x y | x >= y    = x
          | otherwise = y

instance Ord Ordering where
  compare LT LT = EQ
  compare LT EQ = LT
  compare LT GT = LT
  compare EQ LT = GT
  compare EQ EQ = EQ
  compare EQ GT = LT
  compare GT LT = GT
  compare GT EQ = GT
  compare GT GT = EQ

instance Ord Bool where
  False <= False = True
  False <= True  = True
  True  <= False = False
```

```
    True  <= True  = True

  instance Ord Char where
    c1 <= c2 = c1 `ltEqChar` c2

  instance Ord Int where
    i1 <= i2 = i1 `ltEqInt` i2

  instance Ord Float where
    f1 <= f2 = f1 `ltEqFloat` f2

  instance Ord a => Ord (Maybe a) where
    Nothing <= Nothing = True
    Nothing <= Just _  = True
    Just _  <= Nothing = False
    Just x  <= Just y  = x <= y

  instance (Ord a, Ord b) => Ord (Either a b) where
    Left x  <= Left y  = x <= y
    Left _  <= Right _ = True
    Right _ <= Left _  = False
    Right x <= Right y = x <= y

  instance Ord a => Ord [a] where
    []    <= []    = True
    (_:_) <= []    = False
    []    <= (_:_) = True
    (x:xs) <= (y:ys) | x == y    = xs <= ys
                     | otherwise = x < y

  instance Ord () where
    () <= () = True

  instance (Ord a, Ord b) => Ord (a, b) where
    (a, b) <= (a', b') = a < a' || (a == a' && b <= b')

  instance (Ord a, Ord b, Ord c) => Ord (a, b, c) where
    (a, b, c) <= (a', b', c') = a < a'
       || (a == a' && b < b')
       || (a == a' && b == b' && c <= c')

  instance (Ord a, Ord b, Ord c, Ord d) => Ord (a, b, c, d) where
    (a, b, c, d) <= (a', b', c', d') = a < a'
```

```
     || (a == a' && b < b')
     || (a == a' && b == b' && c < c')
     || (a == a' && b == b' && c == c' && d <= d')

instance (Ord a, Ord b, Ord c, Ord d, Ord e) => Ord (a, b, c, d, e) where
  (a, b, c, d, e) <= (a', b', c', d', e') = a < a'
     || (a == a' && b < b')
     || (a == a' && b == b' && c < c')
     || (a == a' && b == b' && c == c' && d < d')
     || (a == a' && b == b' && c == c' && d == d' && e <= e')

type ShowS = String -> String

class Show a where
  show :: a -> String

  showsPrec :: Int -> a -> ShowS

  showList :: [a] -> ShowS

  showsPrec _ x s = show x ++ s
  show x = shows x ""
  showList ls s = showList' shows ls s

showList' :: (a -> ShowS) ->  [a] -> ShowS
showList' _     []     s = "[]" ++ s
showList' showx (x:xs) s = '[' : showx x (showl xs)
  where
    showl []     = ']' : s
    showl (y:ys) = ',' : showx y (showl ys)

shows :: Show a => a -> ShowS
shows = showsPrec 0

showChar :: Char -> ShowS
showChar c s = c:s

showString :: String -> ShowS
showString str s = foldr showChar s str

showParen :: Bool -> ShowS -> ShowS
showParen b s = if b then showChar '(' . s . showChar ')' else s
```

```
instance Show () where
  showsPrec _ () = showString "()"

instance (Show a, Show b) => Show (a, b) where
  showsPrec _ (a, b) = showTuple [shows a, shows b]

instance (Show a, Show b, Show c) => Show (a, b, c) where
  showsPrec _ (a, b, c) = showTuple [shows a, shows b, shows c]

instance (Show a, Show b, Show c, Show d)
  => Show (a, b, c, d) where
  showsPrec _ (a, b, c, d) =
    showTuple [shows a, shows b, shows c, shows d]

instance (Show a, Show b, Show c, Show d, Show e)
  => Show (a, b, c, d, e) where
  showsPrec _ (a, b, c, d, e) =
    showTuple [shows a, shows b, shows c, shows d, shows e]

instance Show a => Show [a] where
  showsPrec _ = showList

instance Show Bool where
  showsPrec _ True = showString "True"
  showsPrec _ False = showString "False"

instance Show Ordering where
  showsPrec _ LT = showString "LT"
  showsPrec _ EQ = showString "EQ"
  showsPrec _ GT = showString "GT"

instance Show Char where
  showsPrec _ c = showString (show_ c)

  showList cs = showString (show_ cs)

instance Show Int where
  showsPrec _ i = showString $ show_ i

instance Show Float where
  showsPrec _ f = showString $ show_ f

instance Show a => Show (Maybe a) where
```

```
  showsPrec _ Nothing = showString "Nothing"
  showsPrec p (Just x) = showParen (p > appPrec)
    (showString "Just " . showsPrec appPrec1 x)


instance (Show a, Show b) => Show (Either a b) where
  showsPrec p (Left x) = showParen (p > appPrec)
    (showString "Left " . showsPrec appPrec1 x)
  showsPrec p (Right y) = showParen (p > appPrec)
    (showString "Right " . showsPrec appPrec1 y)

showTuple :: [ShowS] -> ShowS
showTuple ss = showChar '('
             . foldr1 (\s r -> s . showChar ',' . r) ss
             . showChar ')'

appPrec :: Int
appPrec = 10

appPrec1 :: Int
appPrec1 = 11

type ReadS a = String -> [(a, String)]

class Read a where
  readsPrec :: Int -> ReadS a

  readList :: ReadS [a]

reads :: Read a => ReadS a
reads = readsPrec minPrec

minPrec :: Int
minPrec = 0

class Bounded a where
  minBound, maxBound :: a

class Enum a where
  succ :: a -> a
  pred :: a -> a

  toEnum   :: Int -> a
```

```
    fromEnum :: a -> Int

    enumFrom       :: a -> [a]
    enumFromThen   :: a -> a -> [a]
    enumFromTo     :: a -> a -> [a]
    enumFromThenTo :: a -> a -> a -> [a]

    succ = toEnum . (+ 1) . fromEnum
    pred = toEnum . (\x -> x -1) . fromEnum
    enumFrom x = map toEnum [fromEnum x ..]
    enumFromThen x y = map toEnum [fromEnum x, fromEnum y ..]
    enumFromTo x y = map toEnum [fromEnum x .. fromEnum y]
    enumFromThenTo x1 x2 y =
      map toEnum [fromEnum x1, fromEnum x2 .. fromEnum y]

  instance Bounded () where
    minBound = ()
    maxBound = ()

  instance Enum () where
    succ _      = error "Prelude.Enum.().succ: bad argument"
    pred _      = error "Prelude.Enum.().pred: bad argument"

    toEnum x | x == 0    = ()
             | otherwise = error "Prelude.Enum.().toEnum: bad argument"

    fromEnum () = 0
    enumFrom ()         = [()]
    enumFromThen () ()  = let many = ():many in many
    enumFromTo () ()    = [()]
    enumFromThenTo () () () = let many = ():many in many

  instance Bounded Bool where
    minBound = False
    maxBound = True

  instance Enum Bool where
    succ False = True
    succ True  = error "Prelude.Enum.Bool.succ: bad argument"

    pred False = error "Prelude.Enum.Bool.pred: bad argument"
    pred True  = False
```

```
  toEnum n | n == 0 = False
           | n == 1 = True
           | otherwise =
             error "Prelude.Enum.Bool.toEnum: bad argument"

  fromEnum False = 0
  fromEnum True  = 1

  enumFrom = boundedEnumFrom
  enumFromThen = boundedEnumFromThen

instance (Bounded a, Bounded b) => Bounded (a, b) where
  minBound = (minBound, minBound)
  maxBound = (maxBound, maxBound)

instance (Bounded a, Bounded b, Bounded c) => Bounded (a, b, c) where
  minBound = (minBound, minBound, minBound)
  maxBound = (maxBound, maxBound, maxBound)

instance (Bounded a, Bounded b, Bounded c, Bounded d)
  => Bounded (a, b, c, d) where
  minBound = (minBound, minBound, minBound, minBound)
  maxBound = (maxBound, maxBound, maxBound, maxBound)

instance (Bounded a, Bounded b, Bounded c, Bounded d, Bounded e)
  => Bounded (a, b, c, d, e) where
  minBound = (minBound, minBound, minBound, minBound, minBound)
  maxBound = (maxBound, maxBound, maxBound, maxBound, maxBound)

instance Bounded Ordering where
  minBound = LT
  maxBound = GT

instance Enum Ordering where
  succ LT = EQ
  succ EQ = GT
  succ GT = error "Prelude.Enum.Ordering.succ: bad argument"

  pred LT = error "Prelude.Enum.Ordering.pred: bad argument"
  pred EQ = LT
  pred GT = EQ

  toEnum n | n == 0 = LT
```

```
            | n == 1 = EQ
            | n == 2 = GT
            | otherwise =
              error "Prelude.Enum.Ordering.toEnum: bad argument"

  fromEnum LT = 0
  fromEnum EQ = 1
  fromEnum GT = 2

  enumFrom = boundedEnumFrom
  enumFromThen = boundedEnumFromThen

uppermostCharacter :: Int
uppermostCharacter = 0x10FFFF

instance Bounded Char where
   minBound = chr 0
   maxBound = chr uppermostCharacter

instance Enum Char where
  succ c | ord c < uppermostCharacter = chr $ ord c + 1
         | otherwise = error "Prelude.Enum.Char.succ: no successor"

  pred c | ord c > 0 = chr $ ord c - 1
         | otherwise = error "Prelude.Enum.Char.succ: no predecessor"

  toEnum = chr
  fromEnum = ord

  enumFrom = boundedEnumFrom
  enumFromThen = boundedEnumFromThen

instance Enum Int where
  succ x = x + 1
  pred x = x - 1

  toEnum n = n
  fromEnum n = n

  enumFrom = enumFrom_
  enumFromTo = enumFromTo_
  enumFromThen = enumFromThen_
  enumFromThenTo = enumFromThenTo_
```

```
boundedEnumFrom :: (Enum a, Bounded a) => a -> [a]
boundedEnumFrom n =
  map toEnum [fromEnum n .. fromEnum (maxBound `asTypeOf` n)]

boundedEnumFromThen :: (Enum a, Bounded a) => a -> a -> [a]
boundedEnumFromThen n1 n2
  | i_n2 >= i_n1  =
    map toEnum [i_n1, i_n2 .. fromEnum (maxBound `asTypeOf` n1)]
  | otherwise     =
    map toEnum [i_n1, i_n2 .. fromEnum (minBound `asTypeOf` n1)]
  where
    i_n1 = fromEnum n1
    i_n2 = fromEnum n2

class Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a

  fromInteger :: Int -> a

  x - y = x + negate y
  negate x = 0 - x

instance Num Int where
  x + y = x +$ y
  x - y = x -$ y
  x * y = x *$ y

  negate x = 0 - x

  abs x | x >= 0 = x
        | otherwise = negate x

  signum x | x > 0     = 1
           | x == 0    = 0
           | otherwise = -1

  fromInteger x = x

instance Num Float where
```

```
    x + y = x +. y
    x - y = x -. y
    x * y = x *. y

    negate x = negateFloat x

    abs x | x >= 0 = x
          | otherwise = negate x


    signum x | x > 0     = 1
             | x == 0    = 0
             | otherwise = -1

    fromInteger x = i2f x

  class Num a => Fractional a where
    (/) :: a -> a -> a
    recip :: a -> a

    recip x = 1/x
    x / y = x * recip y

    fromRational :: Float -> a

  instance Fractional Float where
    x / y = x /. y
    recip x = 1.0/x

    fromRational x = x

  class (Num a, Ord a) => Real a where
    -- toRational :: a -> Rational

  class Real a => Integral a where
    div  :: a -> a -> a
    mod  :: a -> a -> a
    quot :: a -> a -> a
    rem  :: a -> a -> a

    divMod  :: a -> a -> (a, a)
    quotRem :: a -> a -> (a,a)
```

```
  n `div`  d = q where (q, _) = divMod n d
  n `mod`  d = r where (_, r) = divMod n d
  n `quot` d = q where (q, _) = n `quotRem` d
  n `rem`  d = r where (_, r) = n `quotRem` d

instance Real Int where
  -- no class methods to implement

instance Integral Int where
  divMod n d = (n `div_` d, n `mod_` d)
  quotRem n d = (n `quot_` d, n `rem_` d)

asTypeOf :: a -> a -> a
asTypeOf = const

(+.)   :: Float -> Float -> Float
x +. y = (prim_Float_plus $# y) $# x

prim_Float_plus :: Float -> Float -> Float
prim_Float_plus external

(-.)   :: Float -> Float -> Float
x -. y = (prim_Float_minus $# y) $# x

prim_Float_minus :: Float -> Float -> Float
prim_Float_minus external

--- Multiplication on floats.
(*.)   :: Float -> Float -> Float
x *. y = (prim_Float_times $# y) $# x

prim_Float_times :: Float -> Float -> Float
prim_Float_times external

(/.)   :: Float -> Float -> Float
x /. y = (prim_Float_div $# y) $# x

prim_Float_div :: Float -> Float -> Float
prim_Float_div external

i2f    :: Int -> Float
i2f x = prim_i2f $# x
```

```
prim_i2f :: Int -> Float
prim_i2f external

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

instance Functor [] where
  fmap = map

class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  m >> k = m >>= \_ -> k
  return :: a -> m a
  fail :: String -> m a
  fail = error

instance Monad IO where
  a1 >>= a2 = a1 >>=$ a2
  a1 >>  a2 = a1 >>$  a2
  return = returnIO

instance Monad Maybe where
  Nothing  >>= _ = Nothing
  (Just x) >>= f = f x
  return = Just
```

C. Bekannte Einschränkungen

- Curry stellt mit `Int` und `Float` aktuell nur jeweils einen Datentyp für Ganz- bzw. Gleitkommazahlen bereit, während in Haskell beispielsweise noch `Integer` oder `Double` vorgesehen sind. Als Folge dessen haben einige Klassenmethoden der vordefinierten Standardklassen einen anderen Typ als üblich (s. Anhang B). Zum Beispiel lautet der Typ der `fromInteger`-Methode der `Num`-Klasse `Num a => Int -> a` anstelle von `Num a => Integer -> a`.

- Numerische Literale in Mustern werden anders als in Haskell nicht für beliebige Typen der `Num`-Klasse, sondern nur für die vordefinierten Typen `Int` und `Float` überladen. Die Verwendung der Umformung für numerischer Literale in Ausdrücken (s. Unterabschnitt 8.5.12) für Muster würde unter Umständen zu einer ungewollten Unterbrechung der Berechnung (*Suspension*) führen [Lux08].

- Bei der in Unterabschnitt 8.5.9 eingeführten Unterscheidung von expansiven und nicht-expansiven Ausdrücken werden numerische Literale als nicht-expansiv angesehen. Streng genommen ist das jedoch falsch (vgl. Definition 8.5), da numerische Literale für die vollständigen Applikationen der Klassenmethoden `fromInteger` bzw. `fromRational` stehen (s. Unterabschnitt 8.5.12).

  Die Annahme, dass numerische Literale immer nicht-expansiv sind, wäre nur zulässig, wenn sichergestellt wäre, dass sämtliche Implementierungen der genannten Klassenmethoden nichtdeterministisch sind. Das ist zwar in der Regel der Fall, weswegen diese Annahme überhaupt erst getroffen wurde, muss im Allgemeinen aber nicht gelten. Eine mögliche Alternative wäre es, nur numerische Literale der vordefinierten Typen `Int` und `Float` als nicht-expansiv anzunehmen.

- Das Front-End stellt eine Spracherweiterung namens `NoImplicitPrelude` bereit, die den automatischen Import der Standardbibliothek unterbindet. Die Verwendung dieser Erweiterung kann aktuell zu internen Fehlern führen, da einige Kompilierphasen Entitäten aus der Standardbibliothek voraussetzen. So wird zum Beispiel die `Num`-Klasse zum Überladen von numerischen Literalen genutzt. Dieses Problem ist allerdings nicht durch diese Arbeit entstanden, sondern bestand schon vorher. Dies wird auch ersichtlich, wenn man die Umformung für Right-Sections im *Desugaring* betrachtet (s. Abbildung 7.5), für die die in der Standardbibliothek definierte Funktion `flip` benötigt wird.

  Eine allgemeine Lösung bestünde darin, die erlaubte Syntax bei nicht importierter Standardbibliothek einzuschränken. Die Einhaltung dieser Beschränkungen müsste dann ergänzend im *Syntax Check* überprüft werden.

- Das in Unterabschnitt 8.5.9 vorgestellte Verfahren zur Liberalisierung der in Curry erforderlichen Monomorphierestriktion (s. Abschnitt 5.2) stellt wie dort beschrieben lediglich eine Annäherung zur Identifikation von Grundtermen dar. Daher kann es vorkommen, dass der Typ einer lokalen Definition innerhalb eines Let-Ausdrucks nicht generalisiert wird, obwohl dies eigentlich zulässig wäre. So wird der Typ von `nil` in dem Ausdruck

  ```
  let nil = id [] in (1 : nil, 'a' : nil)
  ```

  zum Beispiel nicht generalisiert, weil der Ausdruck `id []` nach der Definition 8.5 expansiv ist.

- Im Gegensatz zu Haskell sind innerhalb von Klassendeklarationen keine Infix-Deklarationen vorgesehen. Stattdessen kann die Präzedenz und Assoziativität einer Klassenmethode auf oberste Ebene, also gemeinsam mit anderen Infix-Deklarationen, angegeben werden. Anstelle von

  ```
  class C a where
    op :: a -> a -> Bool

    infix 4 `op`
  ```

  muss man also

  ```
  class C a where
    op :: a -> a -> Bool

  infix 4 `op`
  ```

  schreiben. Es handelt sich hierbei lediglich um eine Designentscheidung und die Alternative, Infix-Deklarationen doch in Klassendeklarationen zuzulassen, ließe sich bei Bedarf einfach im *Precedences Check* ergänzen. Es müsste einzig darauf geachtet werden, dass eine Infix-Deklaration in einer Klassendeklaration sich nur auf eine in der jeweiligen Klasse deklarierte Methode bezieht und dass nicht gleichzeitig eine weitere Top-Level-Infix-Deklaration für dieselbe Methode angegeben ist.