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

Masterarbeit

Erweiterung von Curry um Typklassen

Matthias Böhm

31. Oktober 2013

Selbstständigkeitserklärung

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

.....
Matthias Böhm

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Die Möglichkeit, Funktionen polymorph zu verwenden, ist ein wichtiges Merkmal vieler Programmiersprachen. Polymorphie ermöglicht es, Code wiederzuverwenden, zum Beispiel im Falle einer Funktion, die die Länge einer Liste zurückgibt (siehe Abb. 1.1). Anstatt für jeden Elementtyp eine eigene Funktion bereitstellen zu müssen, kann man eine einzige Definition angeben, die auf allen Elementtypen arbeiten kann. Dies wird in der Typsignatur durch die *Typvariable* `a` ausgedrückt. Diese Art der Überladung nennt man *parametrischen Polymorphismus*, da durch Parameter in der Typsignatur angegeben wird, an welchen Stellen beliebige Typen eingesetzt werden können. Ein Typsystem, das parametrischen Polymorphismus unterstützt, ist das *Damas-Milner-Typsystem*, das in vielen funktionalen Programmiersprachen als Grundlage dient.

Eine zweite Art der Überladung ist der sogenannte *Ad-hoc-Polymorphismus*: Hier geht es darum, unter einem Namen verschiedene Implementierungen bereitstellen zu können. Ein Beispiel dafür sind Klassenmethoden in objektorientierten Sprachen: Für jede Klasse kann für eine bestimmte Klassenmethode eine andere Implementierung angegeben werden.

Auch in funktionalen Programmiersprachen möchte man diese Art der Überladung verwenden. Zwei Beispiele dafür sind die Gleichheitsfunktion und eine Funktion, die Daten in eine Zeichenkette umwandelt. Diese Funktionen müssen natürlich für verschiedene Typen verschiedene Implementierungen besitzen.

Die Möglichkeit, für verschiedene Implementierungen denselben Namen verwenden zu können, ist sehr wichtig, wie folgendes Beispiel zeigt:

Angenommen, man möchte die Umwandlung von Daten in Zeichenketten implementieren. Dann müsste man für jeden Datentyp eine eigene Funktion bereitstellen:

```
showChar  :: Char  -> String
showInt   :: Int   -> String
showFloat :: Float -> String
...
```

Dieses Schema stößt schnell an seine Grenzen, wenn man zum Beispiel für Tupel die `show`-Funktion angeben will. Nun müssen für alle möglichen Kombinationen Funktionen bereitgestellt werden:

```
showTupleCharChar  :: (Char, Char)  -> String
showTupleCharInt   :: (Char, Int)   -> String
showTupleIntChar   :: (Int,  Char)  -> String
showTupleIntInt    :: (Int,  Int)   -> String
showTupleCharFloat :: (Char, Float) -> String
...
```

Ohne parametrischem Polymorphismus:

```
lengthBool :: [Bool] -> Int
lengthBool []     = 0
lengthBool (x:xs) = 1 + lengthBool xs

lengthInt  :: [Int]  -> Int
...

lengthChar :: [Char] -> Int
...
```

Mit parametrischem Polymorphismus:

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

Abbildung 1.1: Parametrischer Polymorphismus

Die Anzahl der benötigten Funktionsnamen wächst exponentiell mit der Anzahl der Elemente im Tupel.

Dieses Problem kann nicht mit parametrischem Polymorphismus gelöst werden, da für jeden Elementtyp eine unterschiedliche Implementierung angegeben werden muss.

1.1. Unterstützung von Ad-hoc-Polymorphismus

Ad-hoc-Polymorphismus kann auf verschiedene Arten behandelt werden. Im Laufe der Zeit haben sich in verschiedenen Programmiersprachen unterschiedliche Methoden entwickelt, Ad-hoc-Polymorphismus zu unterstützen. In objektorientierten Sprachen wie Java oder C++ wird zum Beispiel die Auflösung der auszuführenden Implementierung zur Laufzeit ausgeführt, indem in einer Tabelle für den konkreten Typen nachgeschlagen wird, welche Funktion aufgerufen werden soll. Ein anderer Ansatz ist, die Auflösung schon zur Compilezeit durchzuführen. Dies hat den Vorteil, dass der Code effizienter ist, und außerdem mehr Programmiersicherheit geboten wird, da schon zur Compilezeit Typfehler entdeckt werden.

In funktionalen Programmiersprachen wie *Miranda* und *Standard ML* wurden erste Ansätze verfolgt, Ad-hoc-Polymorphismus zu unterstützen, meist auf eine wenig strukturierte Art und Weise. In Standard ML beruht zum Beispiel die Überladung von arithmetischen Funktionen auf einem anderen Konzept als die Überladung des Gleichheitstests. Bei Miranda gibt es sogar drei verschiedene Konzepte, jeweils für die Gleichheit, für arithmetische Operationen, und für die Umwandlung von Daten in Zeichenketten [HHPJW96].

Die Entwickler der Programmiersprache *Haskell* haben nach einer Lösung gesucht,

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

instance Show Bool where
  show True  = "True"
  show False = "False"

instance Show Char where
  show c = "'" ++ [c] ++ "'"

instance (Show a, Show b) => Show (a, b) where
  show (x, y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

Abbildung 1.2: Überladung der `show`-Funktion mit Typklassen

Überladung auf eine einheitliche Weise zu unterstützen. Dies führte zu der Entwicklung von *Typklassen*, die zuerst unter anderem in [WB89] vorgeschlagen wurden. Typklassen haben sich mittlerweile zu einem mächtigen Sprachfeature von Haskell entwickelt. In sogenannten *Klassen* werden die Methoden angegeben, die überladen werden sollen; die eigentliche Überladung findet dann in *Instanzen* für jeweils einen bestimmten Typ statt. Ein Beispiel für eine solche Überladung ist in Abbildung 1.2 für die `show`-Funktion angegeben. Man sieht dort, dass sowohl für einfache Typen wie `Char` und `Bool` als auch für zusammengesetzte Typen wie Tupel leicht eine `show`-Implementierung angegeben werden kann. Auf alle Implementierungen kann über den einzigen Namen `show` zugegriffen werden; welche Implementierung ausgewählt wird, wird durch den Typ des Wertes bestimmt, auf den `show` angewendet wird[1]:

```
show 'c'           ⇝ "'" ++ ['c'] ++ "'"
                   ⇝ "'c'"
show True          ⇝ "True"
show ('c', True)   ⇝ "(" ++ show 'c' ++ "," ++ show True
                        ++ ")"
                   ⇝ "(" ++ "'c'" ++ "," ++ "True" ++ ")"
                   ⇝ "('c',True)"
```

1.2. Curry und funktional-logische Programmierung

In dieser Arbeit soll die Programmiersprache *Curry* [He12] um Typklassen erweitert werden. Curry ist eine *funktional-logische* Programmiersprache, das heißt, dass sie sowohl funktionale als auch logische Sprachelemente besitzt. Der funktionale Teil von Curry ist an die Programmiersprache Haskell angelehnt. Curry verwendet weitgehend dieselbe Syntax wie Haskell, und auch dasselbe Typsystem. Der logische Teil von Curry ist

[1] ⇝ bedeute „wird ausgewertet zu“

in die funktionale Syntax eingebettet. So können Funktionen in Curry im Gegensatz zu Funktionen in funktionalen Sprachen zu mehreren Werten ausgewertet werden, die Funktionen sind also *nichtdeterministisch*. Ein Beispiel hierfür ist die Simulation eines Münzwurfes:

```
coin = 0 ? 1
```

Die Funktion `coin` wird zu 0 *und* zu 1 ausgewertet. Dies wird durch den nichtdeterministischen Operator `?` angegeben.

Auch Funktionsdefinitionen mit überlappenden Regeln wie im folgenden Beispiel erzeugen Nichtdeterminismus:

```
choose x y = x
choose x y = y
```

Hier gibt die Funktion `choose` nichtdeterministisch das erste oder das zweite Argument zurück.

Eine weitere wichtige Eigenschaft von funktional-logischen Programmiersprachen ist, dass Funktionen als *Prädikate* verwendet werden können, wie es etwa aus der Programmiersprache *Prolog* bekannt ist:

```
[1, 2] ++ xs =:= [1, 2, 3, 4] where xs free
```

Wird dieser Ausdruck ausgewertet, so wird für `xs` der Wert `[3, 4]` ermittelt. In Prolog würde ein äquivalenter Ausdruck `append([1, 2], Xs, [1, 2, 3, 4])` lauten. Der Operator `=:=` drückt im Zusammenhang mit der Angabe „`xs free`“ aus, dass für `xs` Werte gesucht werden sollen, so dass die linke und die rechte Seite zu demselben Grundkonstruktorterm ausgerechnet werden können. Ein Grundkonstruktorterm ist ein Term, der keine Variablen und Funktionen enthält.

1.3. Implementierung von Typklassen

Curry basiert wie Haskell auf dem Damas-Milner Typsystem und kann daher ähnlich wie Haskell um Typklassen erweitert werden. Der von Haskell verfolgte Ansatz Typklassen zu implementieren, ist die Verwendung von *Wörterbüchern*: Die Implementierungen von überladenen Klassenmethoden für einen bestimmten Typen werden in einem Wörterbuch abgelegt, und die Wörterbücher werden als Parameter weitergereicht. Soll an einer Stelle eine gewisse Implementierung einer Klassenmethode verwendet werden, so wird diese aus dem Wörterbuch für den entsprechenden Typ bezogen. In der vorliegenden Arbeit werden Typklassen ebenfalls mit Wörterbüchern implementiert.

Es existieren auch andere Ansätze, Typklassen zu implementieren. Für funktional-logische Sprachen wird in [MM11] ein Ansatz vorgeschlagen, der auf einem neuartigen Typsystem basiert. In diesem Typsystem ist es möglich, Funktionen zu überladen, indem einfach die Funktionsgleichungen für die verschiedenen Typen hintereinander geschrieben werden. Typklassen werden dadurch implementiert, dass sogenannte *type witnesses (Typzeugen)* in den Funktionsgleichungen verwendet werden, aufgrund derer dann die Gleichungen mit der korrekten Implementierung ausgewählt werden.

1.4. Ziel dieser Arbeit

Ziel dieser Arbeit ist es, ein vorhandenes Curry-System um Ad-hoc-Polymorphismus auf Grundlage von Typklassen zu erweitern. Das dabei verwendete System ist KiCS2[2], das von Curry nach Haskell kompiliert. Auch der PAKCS-Compiler[3], der von Curry nach Prolog kompiliert, kann von dieser Arbeit profitieren, da KiCS2 und PAKCS dasselbe Frontend benutzten, und bis auf eine Ausnahme nur dieses bei der vorliegenden Arbeit modifiziert wurde.

1.5. Struktur der Ausarbeitung

Das zweite Kapitel dieser Ausarbeitung beschäftigt sich mit Typklassen und deren Implementierung mit Wörterbüchern. Zunächst werden in diesem Kapitel Typklassen informell eingeführt, worauf eine formale Beschreibung der Typklassen folgt. Anschließend folgt eine informelle Einführung darin, wie Typklassen mit Hilfe von Wörterbüchern implementiert werden können. Am Schluss des Kapitels wird die Implementierung mit Wörterbüchern formal beschrieben.

Im dritten Kapitel sind die Anpassungen des Typsystems beschrieben, die durchgeführt werden müssen, damit Programme mit Typklassen korrekt getypt werden können. Dabei wird zuerst auf das zugrunde liegende Damas-Milner-Typsystem eingegangen, und danach eine Erweiterung dieses Typsystems um *Prädikate* eingeführt.

Im vierten Kapitel wird die konkrete Implementierung von Typklassen für den Curry-Compiler KiCS2 beschrieben. Dabei wird zuerst das ursprüngliche System vorgestellt, und anschließend werden die Erweiterungen beschrieben, die notwendig sind, um Typklassen in diesem System praktisch zu implementieren.

Im letzten Kapitel sind die Zusammenfassung und ein Ausblick zu finden.

Im Anhang ist die formale Syntax von um Typklassen erweitertem Curry aufgeführt, und ein ausführliches Fallbeispiel angegeben. Außerdem ist dort die Typklassen-Prelude zu finden.

[2] `www-ps.informatik.uni-kiel.de/kics2/`
[3] `http://www.informatik.uni-kiel.de/~pakcs/`

2. Typklassen

2.1. Informelle Einführung

2.1.1. Überladung der Gleichheitsfunktion

Ein in funktionalen Programmiersprachen oft auf heterogene Weise gelöstes Problem ist die Bereitstellung einer Gleichheitsfunktion für verschiedene Daten. Mit Typklassen kann die Gleichheitsfunktion einfach überladen werden. Zuerst erstellt man eine *Klasse*, die die zu überladenden Funktionen enthält, hier also die Gleichheitsfunktion:

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

Für konkrete Typen kann die Gleichheitsfunktion nun überladen werden, indem für den Typen und die Klasse eine *Instanz* angegeben wird:

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

Wird nun die Gleichheitsfunktion auf boolesche Werte angewandt, so wird die angegebene Implementierung verwendet.

Oft will man auch für zusammengesetzte Typen eine Gleichheitsfunktion angeben, wie zum Beispiel für Tupel. Die Voraussetzung dafür, dass ein Tupel mit einem anderen Tupel verglichen werden kann, ist, dass auch die Elemente der Tupel miteinander verglichen werden können. Dies kann man durch folgende Instanz angeben:

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

Zu beachten ist hier, dass die drei Gleichheitsoperatoren jeweils auf anderen Typen arbeiten.

2.1.2. Kontexte

Auf Basis der überladenen Klassenfunktionen oder *Klassenmethoden* können nun andere Funktionen ebenfalls überladen werden. Zum Beispiel kann man jetzt eine Element-Funktion für Listen angeben:

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

Die Typsignatur von `elem` ist `a -> [a] -> Bool`. In dieser Typsignatur ist aber noch nicht berücksichtigt, dass die Elementfunktion nur auf Werte angewandt werden kann, für deren Typ die Gleichheitsfunktion überladen ist. Um dieses anzugeben, wird der Typsignatur ein *Kontext* hinzugefügt, sodass die vollständig Typsignatur nun `Eq a => a -> [a] -> Bool` lautet, wobei `Eq a` der Kontext genannt wird.

2.1.3. Default-Methoden

Es liegt nahe, zusätzlich zu der Gleichheitsfunktion auch die Ungleichheitsfunktion in der `Eq`-Klasse bereitzustellen. Dadurch müssten nun alle Instanzen mit der `Eq`-Klasse auch die Ungleichheitsfunktion explizit implementieren. Allerdings kann die Ungleichheitsfunktion leicht auf die Gleichheitsfunktion zurückgeführt werden: Zwei Werte sind ungleich, wenn sie nicht gleich sind. Damit reicht es aus, nur die Gleichheitsfunktion bereitzustellen. Damit nicht in jeder Instanz das Gesetz zwischen Gleichheit und Ungleichheit angegeben werden muss, kann in der Klassendefinition eine *Default-Methode* angegeben werden, die dieses Gesetz implementiert:

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

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

Wird nun die Ungleichheitsfunktion in einer Instanz nicht angegeben, so wird automatisch die Default-Implementierung eingefügt. Trotzdem hat der Programmierer immer noch die Möglichkeit, für die Ungleichheitsfunktion eine eigene Implementierung anzugeben.

2.1.4. Superklassen

Oft will man auch die Ordnungsoperatoren, wie `<`, `<=`, `>` und `>=` überladen. Dazu kann man, wie bei der `Eq`-Klasse, eine eigene Klasse bereitstellen, die diese Operatoren enthält. Diese Klasse wird oft `Ord` genannt. Wenn die Ordnungsoperatoren definiert sind, sollten aber auch die Gleichheitsoperatoren definiert sein, denn der `<`-Operator kann zum Beispiel auf den `<=`-Operator mit Hilfe der Gleichheit zurückgeführt werden:

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

Um anzugeben, dass in der `Ord`-Klasse auch die Gleichheitsfunktionen bereitgestellt werden sollen, kann man eine *Superklassenbeziehung* angeben:

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

Damit wird die `Ord`-Klasse als *Subklasse* der `Eq`-Klasse deklariert, womit in der `Ord`-Klasse auch die Gleichheitsoperatoren verfügbar sind. Eine Folge dessen ist, dass nun Implementatoren der `Ord`-Klasse auch immer eine Instanz für die `Eq`-Klasse angeben müssen, wie im folgenden Beispiel, in dem die Gleichheits- und Vergleichsoperatoren für den Datentyp `Char` überladen werden:

```
instance Eq Char where
  c == c' = ord c == ord c'
  c /= c' = ord c /= ord c'

instance Ord Char where
  c <= c' = ord c <= ord c'
  c <  c' = ord c <  ord c'
  c >  c' = ord c >  ord c'
  c >= c' = ord c >= ord c'
```

Auch hier kann durch Default-Methoden in der Klasse `Ord` wieder viel Schreibaufwand gespart werden, indem alle Ordnungsoperatoren auf einen einzigen Ordnungsoperator und die Gleichheitsfunktion zurückgeführt werden; hier zum Beispiel auf den Operator `<=`:

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

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

Damit können nun die obigen Instanzen folgendermaßen vereinfacht geschrieben werden:

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

instance Ord Char where
  c <= c' = ord c <= ord c'
```

2.1.5. Automatisches Ableiten von Typklasseninstanzen

Oft folgt die Implementierung bestimmter Typklassen immer demselben Schema. Wird zum Beispiel die Gleichheitsfunktion überladen, so werden zumeist die Datenkonstruktoren eines Datentyps miteinander verglichen, und wenn diese gleich sind, werden zusätzlich die Parameter der Datenkonstruktoren miteinander verglichen. Die Implementierung dieses Schemas per Hand ist fehleranfällig, und führt oft zu einer Code-Explosion, wenn der Datentyp viele Konstruktoren hat. Gerade in funktional-logischen Sprachen, bei denen überlappende Pattern automatisch Nichtdeterminismus bedeuten, ist dies der Fall (siehe Abb. 2.1). Für n Konstruktoren müssen n^2 Gleichungen aufgeschrieben werden.

Um dem Programmierer diese Arbeit abzunehmen, können für algebraische Datentypen *automatisch* Typklasseninstanzen *abgeleitet* werden. Dies wird durch eine *deriving*-Klausel in der Datentypdefinition angegeben. Für das aufgeführte Beispiel lautet die Syntax dafür:

```
data T a = T1 a | T2 a | T3 a

instance Eq a => Eq (T a) where
  T1 x == T1 x' = x == x'
  T2 x == T2 x' = x == x'
  T3 x == T3 x' = x == x'
  _    == _     = False
```

```
instance Eq a => Eq (T a) where
  T1 x == T1 x' = x == x'
  T1 _ == T2 _  = False
  T1 _ == T3 _  = False
  T2 _ == T1 _  = False
  T2 x == T2 x' = x == x'
  T2 _ == T3 _  = False
  T3 _ == T1 _  = False
  T3 _ == T2 _  = False
  T3 x == T3 x' = x == x'
```

Abbildung 2.1: Implementierung der `Eq`-Instanz in funktionalen Sprachen (links) und in funktional-logischen Sprachen (rechts). Um Nichtdeterminismus zu vermeiden, dürfen bei funktional-logischen Sprachen in der `Eq`-Instanz keine überlappende Regeln vorkommen.

```
data T a = T1 a | T2 a | T3 a
  deriving Eq
```

Damit wird automatisch die in Abb. 2.1 auf der rechten Seite angegebene `Eq`-Instanz generiert.

Auch für andere Klassen können automatisch Instanzen abgeleitet werden:

- Für die `Ord`-Klasse, die ja Vergleichsfunktionen überlädt: Die Konvention bei der `Ord`-Klasse ist, dass Konstruktoren, die in der Definition des algebraischen Datentyps weiter links stehen, immer kleiner sind als die, die weiter rechts stehen. Sind beim einem Vergleich zweier Datenkonstruktoren diese gleich, so werden *lexikografisch* alle Parameter der Datenkonstruktoren verglichen: Zuerst wird der erste Parameter herangezogen; ist die Beziehung der Parameter „kleiner“ oder „größer“, so lautet die Beziehung des Gesamtausdrucks ebenfalls „kleiner“ oder „größer“. Sind hingegen beide Parameter gleich, so wird der zweite Parameter herangezogen, und dasselbe Schema angewandt. Es werden also nach und nach alle Parameter miteinander verglichen, so lange, bis zwei Parameter ungleich zueinander sind. Ein Beispiel für die Ableitung einer `Ord`-Instanz ist Abb. 2.2 aufgeführt.

- Für die `Show`-Klasse: Die `Show`-Klasse enthält Funktionen, mit Hilfe derer Daten in eine Zeichenketten-Repräsentation umgewandelt werden können. Bei der automatischen Ableitung werden die Datenkonstruktornamen, wie sie im ursprünglichen Quelltext stehen, verwendet. Die `Show`-Klasse wird im nächsten Abschnitt detailliert betrachtet.

- Für die `Read`-Klasse: Die `Read`-Klasse ist das Gegenstück der `Show`-Klasse: sie enthält Funktionen mit Hilfe derer aus Zeichenketten wieder Daten (also zusammengesetzte algebraische Datentypen) gewonnen werden können.

```
data T a b c = T1 a b c | T2 a b
  deriving Ord
```

wird zu:

```
instance (Ord a, Ord b, Ord c) => Ord (T a b c) where
  T1 x y z <= T1 x' y' z' =
       x < x'
    || x == x' && y < y'
    || x == x' && y == y' && z <= z'
  T1 _ _ _ <= T2 _ _    = True
  T2 _ _   <= T1 _ _ _  = False
  T2 x y   <= T2 x' y' =
       x < x'
    || x == x' && y <= y'
```

Abbildung 2.2: Beispiel für eine abgeleitete `Ord`-Instanz

- Für die `Bounded`- und `Enum`-Klassen: Diese beide Klassen sind eng miteinander verbunden. Die `Bounded`-Klasse enthält Methoden, mit Hilfe derer die Grenzen eines *beschränkten* Datentyps ermittelt werden können. Bei einer *Enumeration (Auflistung)* von Datenkonstruktoren sind dies das erste und das letzte Element. Ein algebraischer Datentyp ist genau dann eine Enumeration, wenn alle Datenkonstruktoren nullstellig sind. Die `Enum`-Klasse enthält Methoden, mit Hilfe derer alle Datenkonstruktoren einer Enumeration aufgezählt werden können.

  Ein Beispiel für abgeleitete `Bounded`- und `Enum`-Instanzen ist in Abbildung 2.3 zu finden.

2.1.5.1. Ableitung der Show-Instanz

Die `Show`-Klasse bedarf einiger Erläuterungen. Zuerst die Klassendefinition der `Show`-Klasse:

```
class Show a where
  show :: a -> String
  showsPrec :: Int -> a -> ShowS
  showList :: [a] -> ShowS

  showsPrec _ x s = show x ++ s
  show x = shows x ""
```

Hier werden entgegen den Erwartungen *drei* Methoden definiert, anstatt nur die `show`-Methode. Die *`show`*-Methode ermöglicht es einem zwar leicht, eine Konversion von Daten in Zeichenketten anzugeben, hat aber bei zusammengesetzten Datentypen eine schlechte

```
data T = T1 | T2 | T3 | T4
  deriving (Bounded, Enum)
```

wird zu:

```
instance Bounded T where
  minBound = T1
  maxBound = T4

instance Enum T where
  -- fromEnum :: T -> Int
  fromEnum T1 = 0
  fromEnum T2 = 1
  fromEnum T3 = 2
  fromEnum T4 = 3

  -- toEnum :: Int -> T
  toEnum n | n == 0 = T1
           | n == 1 = T2
           | n == 2 = T3
           | n == 3 = T4
           | otherwise = error "toEnum: illegal index"

  -- succ :: T -> T
  succ T1 = T2
  succ T2 = T3
  succ T3 = T4
  succ T4 = error "no successor for T4"

  -- pred :: T -> T
  pred T1 = error "no predecessor for T1"
  pred T2 = T1
  pred T3 = T2
  pred T4 = T3
```

Abbildung 2.3: Beispiel abgeleiteter `Bounded`- und `Enum`-Instanzen

Performanz: Durch die Konkatenation von Zeichenketten ist die Laufzeit von `show` quadratisch in der Länge der Gesamtzeichenkette, wie folgende einfache Implementierung der Ausgabe von Tupeln zeigt:

```
instance (Show a, Show b) => Show (a, b) where
  show (a, b) = "(" ++ show a ++ "," ++ show b ++ ")"
```

Zuerst werden für die Tupelelemente Listen aufgebaut, und danach die Zeichenkette für das Tupel. Die Konkatenation geschieht also nicht „in einem Rutsch“.

Um die quadratische Laufzeit zu vermeiden, und eine lineare Laufzeit zu erhalten, werden *Differenzlisten* verwendet. Anstatt Listen zu konkatenieren, werden Funktionen verknüpft. Der Typ der Funktionen ist `String -> String` (alias `ShowS`), jede Funktion nimmt also einen String, und fügt einen String vor dem übergebenen String ein. Das kann dann zum Beispiel bei den folgenden Hilfsfunktionen so aussehen:

```
showChar :: Char -> ShowS
showChar c = (c :)

showString :: String -> ShowS
showString s = (s ++)

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

Am Schluss wird, um den eigentlichen String zu erhalten, die Funktion, die aus der Verkettung von Einzelfunktionen entstanden ist, auf den leeren String angewendet. Dadurch werden der Reihe nach alle Einzelfunktionen aufgerufen, die ihren String vor dem schon erhaltenen String einfügen. Dadurch wird die Laufzeit insgesamt linear.

Um die `Show`-Klasse mit Differenzlisten zu implementieren, muss man die `showsPrec`-Methode benutzen, welche auf Differenzlisten arbeitet. Die `showsPrec`-Methode hat außerdem noch einen `Int`-Parameter. Dieser hat den Zweck, unnötige Klammern zu vermeiden. Der Parameter enthält die Präzedenz des umgebenden Operators, anhand derer und der Präzedenz des aktuellen Operators ermittelt werden kann, ob Klammern eingefügt werden müssen oder nicht. Operatoren selber können durch algebraische Datentypen eingeführt werden, indem der Operator Infix geschrieben wird, wie zum Beispiel in „`data T a b = a :=: b`“. Außerdem können den Operatoren Präzedenzen *zugewiesen* werden, indem im Quelltext eine *Fixity*-Deklaration angegeben wird. Um die Präzedenz von `:=:` zum Beispiel auf 3 zu setzen, wird folgende Fixity-Deklaration angegeben: `infix 3 :=:` (diese Deklaration gibt außerdem an, dass der Operator nicht-assoziativ ist).

Die Verwendung des Präzedenz-Parameters kann anhand des folgenden Beispiels illustriert werden. Es sei der folgende Datentyp gegeben:

```
data Arith = Arith :+: Arith | Arith :*: Arith | Val Int
```

und die Präzedenz von `:+:` auf 6, und die von `:*:` auf 7 gesetzt. In Abbildung 2.4 sind zwei mögliche Werte des Typs `Arith` in Baumdarstellung abgebildet.

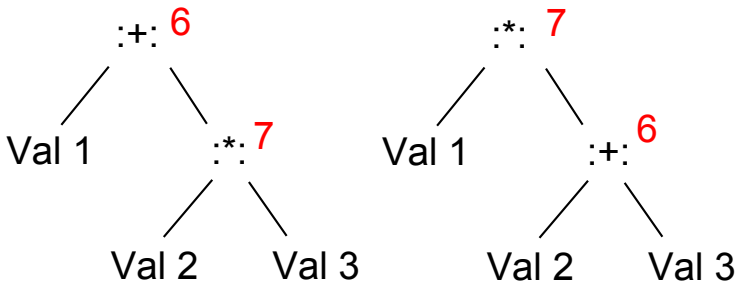

Abbildung 2.4: Zwei verschiedene Dateninstanzen des Datentyps `Arith`

Im ersten Fall wird, um den linken Unterbaum in einen String umzuwandeln, als umgebende Präzedenz die Präzedenz des Plus-Operators, also 6, der Funktion `showsPrec` übergeben. Da diese Präzedenz kleiner ist als die Präzedenz des Mal-Operators, müssen keine Klammern eingefügt werden. Das Ergebnis ist also „`Val 1 :+: Val 2 :*: Val 3`“. Im zweiten Fall wird der Funktion `showsPrec`, wenn diese auf den rechten Teilbaum angewandt wird, aufgrund des Mal-Operators die Präzedenz 7 übergeben. Da diese Präzedenz höher ist als die Präzedenz des Plus-Operators, müssen Klammern gesetzt werden. Das Ergebnis ist also „`Val 1 :*: (Val 2 :+: Val 3)`“. Wie eine abgeleitete Instanz für `Arith` aussieht, ist (vereinfacht) in Abbildung 2.5 angegeben.

Die Addition von eins auf die Präzedenzen ist nötig, um Ausdrücke, in denen ein assoziativer Operator mehrmals hintereinander angewendet wird, korrekt zu klammern. Somit wird `Var 1 :+: Var 2 :+: Var 3` zu `"(Var 1 :+: Var 2) :+: Var 3"` ausgewertet, und nicht zu `"Var 1 :+: Var 2 :+: Var 3"`.

Um eine Datenstruktur in einen String umzuwandeln, kann die Funktion „`shows`“ aufgerufen werden. Diese ruft die Methode „`showsPrec`“ mit einer Präzedenz von null auf:

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

Dadurch, dass die umgebene Präzedenz auf 0 gesetzt wird, wird erreicht, dass nie Klammern ganz außen gesetzt werden.

Die dritte Funktion, `showList`, ermöglicht es dem Programmierer, für Listen eines bestimmten Typs eine andere Formatierung zu wählen als die Standard-Formatierung von Listen in Haskell (also `[`a_1`, ..., `a_n`]`). Dies ist zum Beispiel für Zeichenketten nützlich. So möchte man, wenn man show auf die Zeichenkette `"abc"` anwendet, auch die Zeichenkette `"abc"` erhalten, und nicht `['a', 'b', 'c']`. Damit auch die `showList`-Funktion tatsächlich aufgerufen wird, ist die Instanzdefinition für Listen wie folgt gegeben:

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

Es werden insbesondere um die Liste auch keine Klammern gesetzt.

```
instance Show Arith where
  showsPrec d (Val n) =
    showParen (d > appPrec)
      (showString "Val " .
       showsPrec (appPrec + 1) n)
  showsPrec d (x :+: y) =
    showParen (d > plusPrec)
      (showsPrec (plusPrec + 1) x .
       showString " :+: " .
       showsPrec (plusPrec + 1) y)
  showsPrec d (x :*: y) =
    showParen (d > timesPrec)
      (showsPrec (timesPrec + 1) x .
       showString " :*: " .
       showsPrec (timesPrec + 1) y)

appPrec = 10 -- Präzedenz der Applikation
plusPrec = 6
timesPrec = 7
```

Abbildung 2.5: `Show`-Instanz für den Datentyp `Arith`

2.2. Formale Beschreibung der Typklassenelemente

2.2.1. Klassen

TypeClassDeclaration	::=	`class` [*SimpleContext* `=>`] *TypeClassID TypeVarID* [`where` *ClassDecls*]	
SimpleContext	::=	*Constraint* | `(` $Constraint_1$ `,` ... `,` $Constraint_n$ `)`	$(n \geq 0)$
Constraint	::=	*QTypeClassID TypeVarID*	
ClassDecls	::=	`{` $ClassDecl_1$ `;` ... `;` $ClassDecl_n$ `}`	$(n \geq 0)$
ClassDecl	::=	*Signature* | *Equat*	

Im allgemeinsten Fall haben Klassen die folgende Struktur:

```
class (Sc_1 a, ..., Sc_n a) => C a where
  {Typsignatur|Funktionsdefinition}
```

Hierbei sind $\texttt{Sc}_1$ bis $\texttt{Sc}_n$ die *Superklassen* von `C`. n kann auch 0 sein, in diesem Fall besitzt die angegebene Klasse keine Superklassen. Dann kann auch der Kontext samt dem `=>` weggelassen werden. Wenn n gleich 1 ist, können die Klammern um den Superklassenkontext weggelassen werden. Im Rumpf der Klassendefinition können sowohl Typsignaturen für die Klassenmethoden auftreten als auch Funktionsdefinitionen für Default-Implementierungen von Klassenmethoden.

`a` ist eine Typvariable; diese wird als die *Typvariable der Klasse* bezeichnet. Diese Typvariable ist ein Platzhalter für den konkreten Typ, der bei der Instanzdefinition angegeben wird. Diese Typvariable muss in allen Typsignaturen der Klassenmethoden vorkommen. Außerdem dürfen in den Typsignaturen der Klassenmethoden keine anderen Typvariablen als die Typvariable der Klasse auftreten. Eine gültige Typsignatur für die obige Klasse ist zum Beispiel `a -> a -> Bool`, während die Typsignaturen `Bool -> Bool` (die Klassenvariable kommt nicht vor) und `a -> b -> a` (eine andere Variable als die Klassenvariable wird verwendet) nicht gültig sind. Es gibt Erweiterungen, die einige dieser Einschränkungen aufheben (siehe Abschnitt 2.2.4).

2.2.2. Instanzen

InstanceDeclaration	::=	`instance` [*SimpleContext* `=>`] *QTypeClassID InstanceType* [`where` *InstanceDecls*]	
SimpleContext	::=	*Constraint* | `(` *Constraint*$_1$ `,` ... `,` *Constraint*$_n$ `)`	($n \geq 0$)
Constraint	::=	*QTypeClassID TypeVarID*	
InstanceType	::=	*GeneralTypeConstr*	
	|	`(` *GeneralTypeConstr TypeVarID*$_1$ ... *TypeVarID*$_n$ `)`	($n \geq 0$)
	|	`(` *TypeVarID*$_1$ `,` ... `,` *TypeVarID*$_n$ `)`	(*tuple type*, $n \geq 2$)
	|	`[` *TypeVarID* `]`	(*list type*)
	|	`(` *TypeVarID*$_1$ `->` *TypeVarID*$_2$ `)`	(*arrow type*)
GeneralTypeConstr	::=	*QTypeConstrID* | `()` | `[]` | `(->)` | `(,{,})`	
InstanceDecls	::=	`{` *InstanceDecl*$_1$ `;` ... `;` *InstanceDecl*$_n$ `}`	($n \geq 0$)
InstanceDecl	::=	*Equat*	

Die Struktur von Instanzdefinitionen ist wie folgt:

```
instance (C_1 a_{k_1}, ..., C_n a_{k_n}) => C (T a_1 ... a_m) where
  {Funktionsdefinition}
```

`C` ist die Klasse, für deren Klassenmethoden Implementierungen angegeben werden sollen; `T` ist der konkrete Typ, für den diese Implementierungen angegeben werden sollen. $\mathtt{a}_1$ bis $\mathtt{a}_m$ sind Typvariablen. Dabei muss die Anzahl der Typvariablen der Anzahl der Typvariablen in der Datentyp-Definition von `T` entsprechen. Die obige Instanzdefinition ist zum Beispiel nur dann korrekt, wenn der Datentyp `T` auf folgende Weise angegeben wurde: `data T` $\mathtt{b}_1$ `...` $\mathtt{b}_m$ `= ...`. Eine weitere Beschränkung ist, dass keine Typvariablen doppelt vorkommen dürfen.

Auch die $\mathtt{a}_{k_i}$, $\mathrm{i} \in \{1, \ldots, n\}$, $k_i \in \{1, \ldots, m\}$ sind Typvariablen. Wie schon durch die Indizierung angedeutet, müssen alle Typvariablen im Kontext der Instanzdefinition auch auf der rechten Seite nach dem `=>`-Symbol auftreten.

Es können sowohl n als auch m 0 sein; im ersten Fall ist der Kontext der Instanzdefinition leer; im zweiten Fall besitzt der Typ keine Typvariablen, wie es zum Beispiel für den Typ `Bool` der Fall ist.

Wenn n gleich 1 ist, können die Klammern um den Instanzkontext auch weggelassen werden. Wenn m gleich 0 ist, können auch die Klammern um den Typen `T` weggelassen werden.

Der Typ `T` kann sowohl ein Datentyp, der im Programm via `data` definiert wurde, als auch einer der folgenden speziellen Datentypen sein: `()` („Unit“), `[]` (Typkonstruktor für Listen), `(->)` (Typkonstruktor für Funktionen) und `(,)`, `(,,)`, `(,,,)` ... (Tupeltypkonstruktoren). `T` darf *nicht* ein Typsynonym sein.

Eine Instanz für die Klasse `C` und den Typen `T` wird `C-T`-Instanz genannt.

2.2.3. Kontexte

Context	::=	*SimpleContext*	(*für diese Implementierung*)
SimpleContext	::=	*Constraint* | `(` *Constraint*$_1$ `,` ... `,` *Constraint*$_n$ `)`	($n \geq 0$)
Constraint	::=	*QTypeClassID TypeVarID*	
Signature	::=	*FunctionNames* `::` [*Context* `=>`] *TypeExpr*	
TypedExpr	::=	*InfixExpr* `::` [*Context* `=>`] *TypeExpr*	(*expression type signature*)
	|	*InfixExpr*	

Kontexte, die in Typsignaturen im Quelltext angegeben werden dürfen, haben immer die folgende Form:

`(`$\mathtt{C}_1$ $\mathtt{a}_1$`, ...,` $\mathtt{C}_n$ $\mathtt{a}_n$`)`, $n \geq 0$

$\mathtt{C}_1$ bis $\mathtt{C}_n$ sind dabei Namen von Klassen, und $\mathtt{a}_1$ bis $\mathtt{a}_n$ sind Typvariablen. Die einzelnen Elemente des Kontexts werden *Constraints* („Einschränkungen“) genannt.

In Typsignaturen werden Kontexte der eigentlichen Typangabe vorangestellt, und durch ein `=>` von der Typangabe getrennt:

`(`$\mathtt{C}_1$ $\mathtt{a}_1$`, ...,` $\mathtt{C}_n$ $\mathtt{a}_n$`) =>` τ

τ steht für einen beliebigen Typen. Ist n gleich 1, so können die Klammern um den Kontext auch weggelassen werden. Es muss gelten, dass alle Typvariablen $\mathtt{a}_i$, $i \in \{1, \ldots, n\}$ auch auf der rechten Seite der Typsignatur, also in τ, auftreten. Ansonsten wird ein Syntaxfehler vom Compiler gemeldet, die Typsignatur wird in diesem Fall als *ambiguous* (mehrdeutig) bezeichnet.

Hier sei anzumerken, dass in dieser Implementierung die Kontexte, die in Typsignaturen verwendet werden dürfen, dieselbe Form haben wie die Kontexte in Klassen- und Instanzdefinitionen. Wird das Typklassensystem um Konstruktorklassen erweitert (siehe Abschnitt 2.2.4), so müssen für Typsignaturen erweiterte Kontexte zugelassen werden.

Kontexte, die bei der Übersetzung auftreten, können wesentlich komplexer sein als die Kontexte, die im Quellcode angegeben werden dürfen. Sie haben die folgende Form:

`(`$\mathtt{C}_1$ τ_1`, ...,` $\mathtt{C}_n$ τ_n`)`, $n \geq 0$

wobei $\tau_1, \ldots, \tau_n$ beliebige Typen sind. Typen bestehen aus *Typvariablen*, *Typkonstruktoren* angewandt auf weitere Typen, und dem *Arrow*-Typ, der die Funktionsapplikation angibt (siehe Abb. 2.6).

$$\begin{aligned}\tau &::= \alpha \\ \tau &::= \chi\ \tau_1\ \ldots\ \tau_n \\ \tau &::= \tau_1 \to \tau_2\end{aligned}$$

Abbildung 2.6: Syntax der Typen (α sind Typvariablen, χ Datenkonstruktoren)

2.2.3.1. Kontextreduktion

Die im Programmablauf auftretenden Kontexte können auf zweierlei Weisen vereinfacht werden: die einzelnen Elemente können vereinfacht werden, und es können Elemente aus dem Kontext entfernt werden. Den Gesamtvorgang nennt man *Kontextreduktion*. Für die folgenden Betrachtungen wird ein Kontext als *Menge* von Constraints, und nicht als Tupel, aufgefasst.

Vereinfachung einzelner Elemente Zu Beginn der Kontextreduktion werden die einzelnen Elemente im Kontext vereinfacht. Dabei wird versucht, die Elemente auf die Form „`C a`“ zu bringen, wobei `a` eine Typvariable ist. Diese Form wird *Kopfnormalform* oder *head normal form* genannt [Jon00, Seite 18]. Für die Vereinfachung werden die Instanzdefinitionen verwendet: So kann das Kontextelement „`Eq [a]`“ zu „`Eq a`“ vereinfacht werden, weil die Instanz `instance Eq a => Eq [a]` existiert. Dies lässt sich wie folgt formalisieren:

Transformation 2.1 (Vereinfachung einzelner Kontextelemente)**:**
Ein Kontextelement `C` ($\chi\ \tau_1 \ldots \tau_m$) kann, wenn eine Instanz `instance (`$\mathtt{C}_1\ \mathtt{a}_{k_1}$`, ..., `$\mathtt{C}_n\ \mathtt{a}_{k_n}$`) => C (`$\chi\ \mathtt{a}_1\ \ldots\ \mathtt{a}_m$`)` existiert, zu dem Kontext $\{\mathtt{C}_1\ \tau_{k_1}, \ldots, \mathtt{C}_n\ \tau_{k_n}\}$ *vereinfacht* werden.

Ein Kontext wird vereinfacht, indem solange immer wieder die einzelnen Kontextelemente vereinfacht werden, bis kein Element mehr vereinfacht werden kann:

Transformation 2.2 (Vereinfachung eines Kontexts)**:**
Ein Kontext $K \dot\cup \{C\ \tau\}$ kann zu dem Kontext $K \cup \{C_1\ \tau_1, \ldots, C_n\ \tau_n\}$ *vereinfacht* werden, wenn das Kontextelement $C\ \tau$ zum Kontext $\{C_1\ \tau_1, \ldots, C_n\ \tau_n\}$ vereinfacht werden kann (n kann jeden beliebigen Wert ≥ 0 annehmen!).

Für die vollständige Vereinfachung eines Kontexts wird dieser Schritt solange immer wieder durchgeführt, bis kein Element im Kontext mehr vereinfacht werden kann.

Anzumerken ist, dass unter Umständen durch diese Vereinfachung der Kontext größer werden kann, wie zum Beispiel bei der Vereinfachung von `{Eq (a, b)}` zu `{Eq a, Eq b}`. Bei der Vereinfachung kann aber auch ein Kontextelement vollständig entfernt werden, wie es zum Beispiel bei dem Kontextelement `Eq Int` der Fall ist, wenn eine `Eq-Int`-Instanz existiert.

Ein Kontext wird als *gültig* bezeichnet, wenn der Kontext so vereinfacht werden kann, dass nur noch Kontextelemente in Kopfnormalform in ihm enthalten sind.

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

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

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

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

Abbildung 2.7: Regeln für die Entailmentrelation, Teil 1

Entfernen von Elementen aus dem Kontext Nach der Vereinfachung der Kontextelemente werden alle Elemente aus dem Kontext *entfernt*, die nicht „notwendig“ sind. Was „notwendig“ genau bedeutet, lässt sich durch den Begriff der *Kontextimplikation* formalisieren, der im Folgenden eingeführt wird. Ein Kontextelement ist dann nicht notwendig, wenn es von allen anderen Kontextelementen *impliziert* wird.

Die Tatsache, dass ein Kontext P einen Kontext Q impliziert, wird geschrieben als $P \Vdash Q$. Die $\Vdash$-Relation wird auch *Entailment* genannt[JJM97].

Für die Entailmentrelation gelten zunächst die Regeln in Abb. 2.7. Diese lassen sich in Worten folgendermaßen beschreiben: Die Kontextimplikation ist *transitiv*; ein gegebener Kontext impliziert alle in ihm enthaltenen Kontextelemente (*Monotonie*); wenn ein Kontext zwei verschiedene Kontexte impliziert, so impliziert er auch die Vereinigung der beiden Kontexte; und wenn ein Kontext einen anderen impliziert, so kann der erstere beliebig erweitert werden, ohne die Implikationsbeziehung zu verletzen. Aus (mono) folgt direkt, dass außerdem $P \Vdash \emptyset$ gilt, und die Entailmentrelation *reflexiv* ist, es gilt also $P \Vdash P$.

Weitere Regeln für die Entailmentrelation ergeben sich aus Klassen- und Instanzdefinitionen.

Dazu ein Beispiel: Es seien die folgenden Klassen und Instanzen gegeben:

```
class Eq a => Ord a where ...

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

Der Kontext {`Ord a`} impliziert dann den Kontext {`Eq a`}, weil `Eq` eine Superklasse von `Ord` ist. Außerdem impliziert der Kontext {`Eq a`} den Kontext {`Eq [a]`} aufgrund der Instanzdefinition. Es gilt sogar für einen beliebigen Typen τ, dass {`Ord` τ} $\Vdash$ {`Eq` τ} und {`Eq` τ} $\Vdash$ {`Eq [`τ`]`} gilt.

Diese Beispiele lassen sich durch die Regeln in Abb. 2.8 verallgemeinern[JJM97]. „typevars“ sei dabei die Funktion, die alle Typvariablen in einem Kontext oder einem

(inst)	$\dfrac{\text{typevars}(P) \subseteq \text{dom}(\theta)}{\theta(C) \Vdash \theta(P)}$	wenn Instanz `instance` C `=>` P existiert
(super)	$\dfrac{\text{typevars}(P) \subseteq \text{dom}(\theta)}{\theta(P) \Vdash \theta(C)}$	wenn Klassendefinition `class` C `=>` P existiert

Abbildung 2.8: Regeln für die Entailmentrelation, Teil 2

einzelnen Constraint zurückgibt. θ ist eine Substitution von Typvariablen nach Typausdrücken. Mit $\text{dom}(\theta)$ wird der Wertebereich der Substitution θ angegeben. Die Prämisse der beiden Regeln bedeutet also, dass die Typsubstitution für alle Typvariablen von P definiert ist.

Die Anwendung der Substitution θ auf C und P in den Konklusionen ermöglicht, dass – wie im Beispiel – beliebige Typen an Stelle der Typvariablen verwendet werden können.

Damit lässt sich die Kontextimplikation wie folgt definieren:

Definition 2.1 (Kontextimplikation)**:**
Ein Kontext P *impliziert* einen Kontext Q, wenn $P \Vdash Q$ mit Hilfe der Regeln (trans), (mono), (union), (extend), (inst) und (super) hergeleitet werden kann.

Aufgrund von (mono) und (union) gilt insbesondere, dass ein Kontext P genau dann einen Kontext Q impliziert, wenn $P \Vdash \{\pi\}$ für alle $\pi \in Q$ gilt, wenn also P alle Elemente von Q impliziert[HB90, S. 279].

Nun ein komplettes Beispiel für die Kontextimplikation: Es soll gezeigt werden, dass der Kontext {`Ord a`, `Eq b`} den Kontext {`Eq ([Int], [a], b), Eq a`} impliziert. Die entsprechende Ableitung ist in Abbildung 2.9 zu finden. Der Einfachheit halber werden die Mengenklammern um einen Kontext mit nur einem Kontextelement weggelassen; „`Eq a`“ ist also dasselbe wie „{`Eq a`}“.

Nun lässt sich die Kontextreduktion mit dem Begriff der Kontextimplikation einführen:

Definition 2.2 (Kontextreduktion)**:**
Ein Kontext P kann zu einem Kontext P' *reduziert* werden, wenn $P' \Vdash P$ gilt.

Es wird üblicherweise bei der Kontextreduktion von P nach dem *kleinstmöglichen* Kontext P' gesucht, für den $P' \Vdash P$ gilt.

Im Konkreten bedeutet dies, dass nach der Vereinfachung des Kontexts drei verschiedene Kontexttransformationen angewandt werden können, um den Kontext schrittweise zu verkleinern:

1. Löschen von doppelten Constraints: So kann zum Beispiel {`Eq a`, `Eq a`} aufgrund von (mono) zu {`Eq a`} reduziert werden.

2. Anwendung von (super): So kann der Kontext {`Eq a`, `Ord a`} zu {`Ord a`} reduziert werden, da {`Ord a`} $\Vdash$ {`Eq a`} gilt.

Gegeben:

```
class Eq a where ...
class Eq a => Ord a where ...
instance Eq Int where ...
instance Eq a => Eq [a] where ...
instance (Eq a, Eq b, Eq c) => Eq (a, b, c) where ...
```

Abkürzungen:

(1) = (trans)
(2) = (mono)
(3) = (union)
(4) = (extend)
(5) = (inst)
(6) = (super)

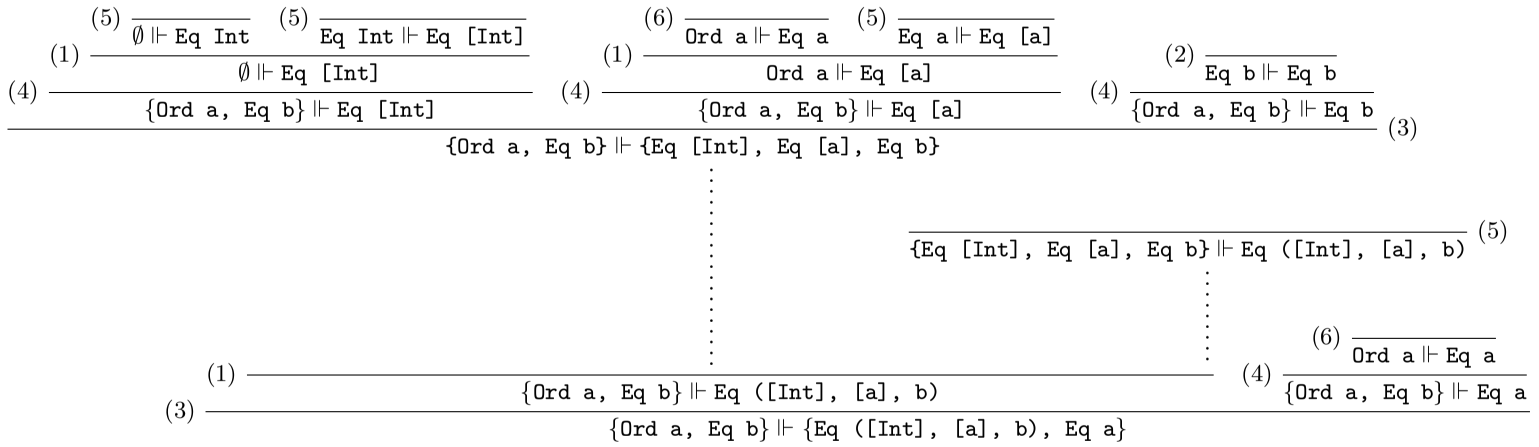

Abbildung 2.9: Ableitung von {Ord a, Eq b} ⊩ {Eq ([Int], [a], b), Eq a}

```
Gegeben: Kontext K

setze K' := vereinfache(K)

setze L := K'
solange ∃π in L mit L\{π} ⊩ {π}
  setze L := L\{π}

Ergebnis: L
```

Abbildung 2.10: Algorithmus für die Kontextreduktion

3. Anwendung von (inst): Der Kontext {`Eq a`, `Eq [a]`} kann aufgrund der obigen Instanzdefinition zu {`Eq a`} reduziert werden, da {`Eq a`} $\Vdash$ {`Eq [a]`} gilt.

   Ist außerdem die Instanzdefinition „**`instance Eq Int where`** ...“ gegeben, so kann der Kontext {`Eq Int`} zum leeren Kontext $\emptyset$ reduziert werden, da {} $\Vdash$ {`Eq Int`} gilt. Ein Constraint, der von einem leeren Kontext impliziert wird, heißt *tautologisch*.

Damit lässt sich der in Abbildung 2.10 angegebene Algorithmus zur Kontextreduktion angeben.

Satz 2.1:
Nach Anwendung des Algorithmus gilt $L \Vdash K$.

Beweis. Zuerst zeige ich, dass $K' \Vdash K$ gilt.

Dies ist aber klar, da bei jedem Vereinfachungsschritt $M \dot\cup \{\mathtt{C}\ (\chi\ \tau_1 \dots \tau_m)\} \to M \cup \{\mathtt{C}_1\ \tau_{k_1}, \dots, \mathtt{C}_m\ \tau_{k_m}\}$ für ein Kontextelement eine Instanzdefinition verwendet wird (siehe Transformation 2.1), und mit derselben Instanzdefinition wegen (inst) gilt, dass $\{\mathtt{C}_1\ \tau_{k_1} \dots \mathtt{C}_m\ \tau_{k_m}\} \Vdash \{\mathtt{C}\ (\chi\ \tau_1 \dots \tau_m)\}$. Somit gilt auch $M \cup \{\mathtt{C}_1\ \tau_{k_1} \dots \mathtt{C}_m\ \tau_{k_m}\} \Vdash M \cup \{\mathtt{C}\ (\chi\ \tau_1 \dots \tau_m)\}$, was durch folgende Ableitung gezeigt werden kann (mit $M_1 := \{\mathtt{C}_1\ \tau_{k_1} \dots \mathtt{C}_m\ \tau_{k_m}\}$ und $M_2 := \{\mathtt{C}\ (\chi\ \tau_1 \dots \tau_m)\}$):

$$\text{(union)}\ \frac{\text{(extend)}\ \dfrac{M_1 \Vdash M_2 \quad \text{(Voraussetzung)}}{M \cup M_1 \Vdash M_2} \qquad \text{(mono)}\ \dfrac{M \subseteq M \cup M_1}{M \cup M_1 \Vdash M}}{M \cup M_1 \Vdash M \cup M_2}$$

Aufgrund der Transitivität von $\Vdash$, und da bei jedem Vereinfachungsschritt $M \to M'$ $M' \Vdash M$ gilt, gilt also auch $K' \Vdash K$.

Nun zeige ich, dass $L \Vdash K'$ gilt.

Dazu betrachte ich zuerst *einen* Schritt des Algorithmus:

Es gilt $L \setminus \{\pi\} \Vdash \{\pi\}$. Zu zeigen ist, dass $L \setminus \{\pi\} \Vdash L$ gilt. Dies kann mit folgender Ableitung gezeigt werden:

$$\frac{L \setminus \{\pi\} \Vdash \{\pi\} \qquad \dfrac{L \setminus \{\pi\} \subseteq L \setminus \{\pi\}}{L \setminus \{\pi\} \Vdash L \setminus \{\pi\}}\ (\text{mono})}{L \setminus \{\pi\} \Vdash L}\ (\text{union})$$

Es gilt also $L_{i+1} \Vdash L_i$, wobei L_i die Menge L im i-ten Schritt des Algorithmus bezeichnen soll. Aufgrund der Transitivität der Entailmentrelation gilt damit, wenn die Schleife nach n Iterationen verlassen wurde, $L = L_n \Vdash L_{n-1} \Vdash \cdots \Vdash L_0 = K'$, also $L \Vdash K'$.

Es gilt also sowohl $L \Vdash K'$ als auch $K' \Vdash K$. Aus der Transitivität folgt, dass $L \Vdash K$ gilt. □

Somit ist L der Kontext, der durch die vollständige Reduktion von K entsteht und sich nicht weiter reduzieren lässt.

Der angegebene Algorithmus lässt noch offen, wie das Element π ausgewählt wird. Wie dies in der konkreten Implementierung geschieht, und aus welchen Gründen, wird in Abschnitt 4.3.2.7 erläutert. Hier sei schon einmal vorweggenommen, dass nicht immer das *erstmögliche* Kontextelement ausgewählt wird, wenn Kontexte als Listen betrachtet werden.

2.2.4. Erweiterungen

Zu der hier vorgestellten Struktur von Typklassen sind im Laufe der Zeit einige Erweiterungen hinzugekommen, die ich hier der Vollständigkeit halber vorstellen will.

In dem ursprünglichen Artikel über Typklassen [WB89] werden nur Typklassen in der hier vorgestellten Form zugelassen, es dürfen in den Typsignaturen von Klassenmethoden also nur die Klassenvariable und keine anderen Typvariablen verwendet werden.

Eine Erweiterung ist, dass diese Beschränkung aufgehoben wird, es dürfen also andere Typvariablen in den Typsignaturen auftreten. Diese Erweiterung erfordert aber ein Typsystem, in dem existentielle Variablen zugelassen sind.

In Haskell 1.3 [P+96] wurden *Konstruktorklassen* eingeführt, die in [Jon95b] vorgeschlagen wurden. In Konstruktorklassen kann die Klassenvariable auch für einen *partiell* angewandten Typkonstruktor stehen. Ein Beispiel dafür ist die `Functor`-Typklasse (siehe Abb. 2.11). Auch hier gilt wieder, dass existentielle Variablen für die Implementierung notwendig sind.

Mittlerweile wird nicht mehr zwischen Typklassen und Konstruktorklassen unterschieden, und es wird der Name Typklasse auch für Konstruktorklassen verwendet.

In [JJM97] wurden unter anderem die folgenden Erweiterungen vorgeschlagen:

- Klassen mit mehr als einer Klassenvariable: Solche Klassendefinitionen sind zum Beispiel nützlich für *Monaden*, im Speziellen für *State-Monaden*.
- Instanzen, in denen anstatt der Typvariablen auch Typkonstruktoren angegeben werden können: So könnte man auch folgende Instanz angegeben: `instance Eq (Maybe Int) where ...`. Ein Problem, das dabei auftritt, ist, dass Instanzen

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

Abbildung 2.11: Beispiel für eine Konstruktorklasse

eventuell *überlappen* können. Angenommen, zu der obigen Instanz wurde noch die Instanz `instance Eq (Maybe a) where ...` definiert. Dann ist unklar, welche Überladung des Gleichheitsoperators verwendet werden soll, wenn zwei Werte des Typs `Maybe Int` verglichen werden.

- Instanzen, in denen Typvariablen doppelt vorkommen können, zum Beispiel die Instanz `instance ... => Eq (a, a) where ....`

Die ersten beiden Erweiterungen sind mittlerweile neben anderen Spracherweiterungen in den Haskell-Compiler *GHC* eingebaut [The13].

2.3. Einführung in die Implementierung mit Wörterbüchern

Eine Möglichkeit Typklassen zu implementieren, ist die Verwendung von *Wörterbüchern*. Dieser Ansatz wird zum Beispiel in Haskell verfolgt, und Wörterbücher werden auch in der vorliegenden Implementierung von Typklassen verwendet.

In diesem Abschnitt möchte ich zunächst einen *informellen* Überblick darüber geben, wie Typklassen mit Hilfe von Wörterbüchern implementiert werden. Im nächsten Abschnitt wird dann die Implementierung formal beschrieben.

2.3.1. Behandlung von Typklassen

Für jede Typklasse wird ein *Wörterbuchtyp* angelegt, der ein Schema angibt, wie überladene Methoden im einem *Wörterbuch* gespeichert werden sollen. Für die `Eq`-Klasse (siehe Abschnitt 2.1.3) lautet der Wörterbuchtyp zum Beispiel[1]:

```
type Dict.Eq a = (a -> a -> Bool, a -> a -> Bool)
```

Dieses Wörterbuchschema gibt an, dass im Wörterbuch zwei Funktionen des Typs `a -> a -> Bool` gespeichert werden sollen; hier sind dies die `==` und die `/=` Funktionen.

Weiterhin werden für jede Klasse *Selektionsfunktionen* definiert, mit Hilfe derer bestimmte Funktionen aus dem Wörterbuch extrahiert werden können. Hier sind dies die folgenden Selektionsfunktionen:

```
sel.Eq.(==), sel.Eq.(/=) :: Dict.Eq a -> (a -> a -> Bool)
sel.Eq.(==) (impl.(==), impl.(/=)) = impl.(==)
sel.Eq.(/=) (impl.(==), impl.(/=)) = impl.(/=)
```

[1] Die Punkte in den Bezeichnernamen dienen der logischen Untergliederung der Namen.

Für Klassen mit Superklassen werden zusätzlich die Wörterbücher der direkten Superklassen im Wörterbuch abgelegt. So lautet der Wörterbuchtyp für die `Ord`-Klasse aus Abschnitt 2.1.4:

```
type Dict.Ord a =
  (Dict.Eq a, a -> a -> Bool,
              a -> a -> Bool,
              a -> a -> Bool,
              a -> a -> Bool)
```

Zusätzlich zu den Selektionsfunktionen für die Klassenmethoden wird nun außerdem eine Selektionsfunktion für das Superklassen-Wörterbuch angelegt:

```
sel.Ord.Eq :: Dict.Ord a -> Dict.Eq a
sel.Ord.Eq (dictEq,
  impl.(<=), impl.(<), impl.(>=), impl.(>)) = dictEq
```

Außerdem werden für alle Default-Methoden, die in der Klasse angegeben wurden, eigene Top-Level-Funktionen generiert, für die `Eq`-Klasse wird zum Beispiel die folgende Funktionsdefinition generiert:

```
def.Eq.(/=) :: Eq a => a -> a -> Bool
def.Eq.(/=) x y = not (x == y)
```

2.3.1.1. Wörterbuchtypen

Es sei hier erwähnt, dass Wörterbücher auf verschiedene Arten und Weisen dargestellt werden können. In der hier verwendeten Darstellung werden *Typsynonyme* für die Wörterbuchtypen verwendet, die zu *Tupeln* aufgelöst werden. Wörterbücher werden also als Tupel dargestellt, mit der Ausnahme, dass ein Wörterbuch, das nur ein Element enthält, genau den Typen dieses einen Elements besitzt (da es keine einelementigen Tupel gibt).

Eine weitere Möglichkeit, Wörterbücher darzustellen, ist die Verwendung von *algebraischen Datentypen*, wie es sie in Haskell und Curry gibt. Würden diese verwendet, so würden die Wörterbuchtypen für die `Eq`- und `Ord`-Klassen folgendermaßen lauten:

```
data Dict.Eq a = Dict.Eq (a -> a -> Bool)
                         (a -> a -> Bool)

data Dict.Ord a = Dict.Ord (Dict.Eq a)
                           (a -> a -> Bool)
                           (a -> a -> Bool)
                           (a -> a -> Bool)
                           (a -> a -> Bool)
```

Die beiden Darstellungsarten sind semantisch äquivalent, Unterschiede gibt es aber bei der Implementierung. Wie schon erwähnt, gibt es keine einstelligen Tupel, es gibt aber sehr wohl algebraische Datentypen, deren Konstruktoren nur einen Parameter haben. Außerdem sind algebraische Datentypen feste Entitäten, die im Verlauf der Kompilierung

bestehen bleiben, während Typsynonyme bei der Kompilierung aufgelöst und entfernt werden.

Die vorliegende Implementierung benutzt, wie schon weiter oben erwähnt, Typsynonyme und Tupel für die Implementierung von Wörterbüchern. Es gab zunächst keine besonderen Gründe, weshalb die Verwendung von Typsynonymen der Verwendung von algebraischen Datentypen vorgezogen wurde. Mittlerweile würde aber für die Implementierung von Wörterbüchern die Wahl auf algebraische Datentypen fallen, denn bei der Verwendung von Typsynonymen für die Wörterbuchtypen muss immer darauf geachtet werden, dass diese korrekt expandiert werden. Dies ist bei algebraischen Datentypen nicht notwendig.

2.3.2. Behandlung von Instanzdefinitionen

Für jede *Instanzdefinition* wird ein *konkretes* Wörterbuch erstellt. Zunächst werden für die angegebenen Implementierungen in der Instanzdefinition eigene Top-Level-Funktionen angelegt, für jede implementierte Klassenmethode eine eigene Funktion. Dann wird das Wörterbuch erstellt, indem die Top-Level-Funktionen im Wörterbuch abgelegt werden. Die Instanzdefinition

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

  False /= False = False
  False /= True  = True
  True  /= False = True
  True  /= True  = False
```

für die `Eq`-Klasse und den Typ `Bool` wird zum Beispiel in folgenden Code transformiert:

```
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

impl.Eq.Bool.(/=) :: Bool -> Bool -> Bool
impl.Eq.Bool.(/=) False False = False
impl.Eq.Bool.(/=) False True  = True
impl.Eq.Bool.(/=) True  False = True
impl.Eq.Bool.(/=) True  True  = False

dict.Eq.Bool :: Dict.Eq Bool
dict.Eq.Bool = (impl.Eq.Bool.(==), impl.Eq.Bool.(/=))
```

Wird die Implementierung von `/=` nicht angegeben, so wird im Wörterbuch die Default-Methode für `/=` abgelegt:

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

wird zu:

```
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

dict.Eq.Bool :: Dict.Eq Bool
dict.Eq.Bool = (impl.Eq.Bool.(==), def.Eq.(/=))
```

Wird schließlich überhaupt keine Implementierung für eine Klassenmethode angegeben, und es existiert keine Default-Methode, so wird ein *Stub* generiert, der eine Fehlermeldung ausgibt:

```
instance Eq Bool where
  -- no implementation
```

wird zu:

```
impl.Eq.Bool.(==) =
  error "Eq.(==) not implemented for type Bool"

dict.Eq.Bool :: Dict.Eq Bool
dict.Eq.Bool = (impl.Eq.Bool.(==), def.Eq.(/=))
```

Bei der Erstellung eines Wörterbuchs für eine `C`-`T`-Instanz müssen, falls vorhanden, auch die Superklassen von `C` berücksichtigt werden. So muss in dem Wörterbuch für die Klasse `Ord` und den Typen `T` auch das Wörterbuch für die Klasse `Eq` und den Typen `T` abgelegt werden. So werden die in Abschnitt 2.1.4 angegebenen Instanzen für `Ord`/`Eq` und `Char` folgendermaßen transformiert:

```
impl.Eq.Char.(==), impl.Eq.Char.(/=)
    :: Char -> Char -> Bool
impl.Eq.Char.(==) c c' = ord c == ord c'
impl.Eq.Char.(/=) c c' = ord c /= ord c'

dict.Eq.Char :: Dict.Eq Char
dict.Eq.Char = (impl.Eq.Char.(==), impl.Eq.Char.(/=))
```

```
impl.Ord.Char.(<=), impl.Ord.Char.(<),
impl.Ord.Char.(>=), impl.Ord.Char.(>)
    :: Char -> Char -> Bool

impl.Ord.Char.(<=) c c' = ord c <= ord c'
impl.Ord.Char.(<)  c c' = ord c <  ord c'
impl.Ord.Char.(>=) c c' = ord c >  ord c'
impl.Ord.Char.(>)  c c' = ord c >= ord c'

dict.Ord.Char :: Dict.Ord Char
dict.Ord.Char = (dict.Eq.Char,
  impl.Ord.Char.(<=), impl.Ord.Char.(<),
  impl.Ord.Char.(>=), impl.Ord.Char.(>))
```

Es folgt, dass für jede `C`-`T`-Instanz auch für alle Superklassen `C'` von `C` eine `C'`-`T`-Instanz existieren muss.

In Abschnitt 2.1.1 wurde auch eine Instanzdefinition für die `Eq`-Klasse und Paare (also für Tupel mit zwei Elementen) angegeben. Das Besondere an dieser Instanzdefinition ist, dass zusätzlich noch ein *Kontext* angegeben wird; dieser gibt an, dass auch für die Elemente des Paares die Gleichheitsfunktion überladen sein muss. Dieser Kontext muss bei der Transformation berücksichtigt werden:

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

wird zu

```
impl.Eq.(,).(==) :: (Eq a, Eq b) =>
   (a, b) -> (a, b) -> Bool
impl.Eq.(,).(==) (a, b) (a', b') = a == a' && b == b'

dict.Eq.(,) :: (Eq a, Eq b) => Dict.Eq (a, b)
dict.Eq.(,) = (impl.Eq.(,).(==), def.Eq.(/=))
```

Der Kontext der Instanzdefinition wird also in die Typsignaturen der Implementierungsfunktionen und in die Typsignatur des Wörterbuchs kopiert.

2.3.3. Einfügen von Wörterbüchern

Sind die Klassen- und Instanzdefinitionen transformiert, so wird auf dem entstandenen Code eine Transformation durchgeführt, bei der Wörterbücher bzw. Wörterbuchparameter eingefügt werden. Diese Transformation wird durch die Informationen, die bei einem *Typcheck* des Programmes gewonnen wurden, gesteuert.

Der Typcheck weist allen im Programm auftretenden Funktionen einen Typ zu; sowohl Funktionen, die im Programm definiert werden, als auch allen Verwendungen von Funktionen. Dabei wird das Damas-Milner-Typsystem [DM82] verwendet, erweitert um das Inferieren der Kontexte. Auf die Typinferenz wird später noch im Detail eingegangen.

Gegeben:

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

instance Show Int where
  ...

instance Show Char where
  ...

fun1   = show 1
fun2   = show 'c'
fun3 x = show x
```

Von der Typinferenz ermittelte Typen:

```
fun1 :: String
  = (show :: Show Int  => Int  -> String) 1
fun2 :: String
  = (show :: Show Char => Char -> String) 'c'
(fun3 :: Show a => a -> String) (x :: a)
  = (show :: Show a    => a    -> String) (x :: a)
```

Abbildung 2.12: Typinferenz

Hier wird die Typinferenz als „*black box*“ betrachtet, die uns bestimmte Typen liefert; warum diese Typen gerade die angegebene Form haben, wird in Kapitel 3 erläutert. Ein Beispiel für die Typinferenz ist in Abb. 2.12 aufgeführt.

Die inferierten Kontexte werden verwendet, um zu bestimmen, welche Wörterbücher bzw. Wörterbuchparameter eingefügt werden müssen. Für jedes Kontextelement wird dabei ein Wörterbuch oder ein Wörterbuchparameter eingefügt.

Beispiel 2.1 (Funktion `fun1`)**:**
Hier ist der Kontext der Typsignatur von `fun1` leer, und es wird kein Wörterbuchparameter eingefügt. Auf der rechten Seite der Funktionsdefinition wird die `show`-Methode auf einen `Int`-Wert angewendet, und im Kontext steht ein „`Show Int`“. Daraus folgt, dass hier ein konkretes Wörterbuch, nämlich das für die Klasse `Show` und den Typen `Int`, eingefügt werden muss. Außerdem wird die Funktion `show`, da sie eine Klassenmethode ist, umgewandelt in die Selektionsfunktion, die aus dem Wörterbuch die entsprechende `show`-Implementierung extrahiert.

Der entstandene Code ist also:

```
fun1 = sel.Show.show dict.Show.Int 1
```

Beispiel 2.2 (Funktion `fun2`)**:**
Hier ist die Situation ähnlich wie im ersten Beispiel, nur dass die Funktion `show` auf einen Character-Wert angewandt wird. Deshalb lautet der entstandene Code hier:

```
fun2 = sel.Show.show dict.Show.Char 'c'
```

Beispiel 2.3 (Funktion `fun3`)**:**
Hier steht sowohl auf der linken Seite als auch auf rechten Seite der Funktionsdefinition im Kontext ein „`Show a`“, wobei das `a` auf der linken Seite dem `a` auf der rechten Seite entspricht. Da die Typvariable `a` für einen beliebigen Typ steht, kann hier kein konkretes Wörterbuch eingefügt werden; stattdessen wird ein *Wörterbuchparameter* eingefügt, und zwar sowohl auf der linken als auch auf der rechten Seite:

```
fun3 dict.Show.a x = sel.Show.show dict.Show.a x
```

Wird `fun3` nun auf einen konkreten Typ angewendet, so wird der Funktion ein konkretes Wörterbuch übergeben, aus dem dann die konkrete Implementierung ausgewählt wird:

```
fun3 'c'
```

wird transformiert in

```
fun3 dict.Show.Char 'c'
```

und dies wird ausgewertet zu

```
sel.Show.show dict.Show.Char 'c'
```

Oft ist die Situation ein wenig komplexer als in den vorangegangenen Beispielen, wenn zum Beispiel Instanzdefinitionen mit Kontexten im Code auftreten. Dann nämlich müssen Wörterbücher aus anderen Wörterbüchern zusammengesetzt werden:

Beispiel 2.4:
Gegeben sei folgender Code:

```
instance (Show a, Show b) => Show (a, b) where
  show (u, v) = "(" ++ show u ++ "," ++ show v ++ ")"

fun4 x y = show (x, y)
```

Die Transformation der Instanzdefinition ergibt:

```
dict.Show.(,) :: (Show a, Show b) => (a, b) -> String
dict.Show.(,) = impl.Show.(,).show

impl.Show.(,).show :: (Show a, Show b)
                   => (a, b) -> String
```

```
impl.Show.(,).show (u, v) =
  "(" ++ show u ++ "," ++ show v ++ ")"

fun4 x y = show (x, y)
```

Die Typinferenz ermittelt folgende Typen:

```
(dict.Show.(,) :: (Show a, Show b) => (a, b) -> String)
  = (impl.Show(,).show
      :: (Show a, Show b) => (a, b) -> String)

(impl.show.(,).show
      :: (Show a, Show b) => (a, b) -> String)
    (u :: a, v :: b)
  = "(" ++ (show :: Show a => a -> String) (u :: a)
    ++ "," ++ (show :: Show b => b -> String) (v :: b)
    ++ ")"

(fun4 :: (Show a, Show b) => a -> b -> String)
    (x :: a) (y :: b)
  = (show :: Show (a, b) => (a, b) -> String)
    (x :: a, y :: b)
```

Im letzten Schritt werden die Wörterbuchparameter entsprechend der Kontexte eingefügt:

```
dict.Show.(,) dict.Show.a dict.Show.b
  = impl.Show.(,).show dict.Show.a dict.Show.b

impl.Show.(,).show dict.Show.a dict.Show.b (u, v)
  = "(" ++ sel.Show.show dict.Show.a u ++ ","
        ++ sel.Show.show dict.Show.b v ++ ")"

fun4 dict.Show.a dict.Show.b x y =
  sel.Show.show (dict.Show.(,) dict.Show.a dict.Show.b)
                (x, y)
```

Zwei Dinge sind hier hervorzuheben:

1. In der Implementierung von `impl.Show.(,).show` werden die Wörterbücher, die der Funktion übergeben werden, an die entsprechenden `show`-Funktionen im Rumpf der Funktion weitergereicht, und zwar das Wörterbuch in `dict.Show.a` an die `show`-Funktion, die auf `a` angewandt wird, und das Wörterbuch in `dict.Show.b` an die `show`-Funktion, die auf `b` angewandt wird.

2. Im Rumpf der Funktion `fun4` muss im Gegensatz zu den vorherigen Beispielen

ein Wörterbuch aus anderen Wörterbüchern *zusammengesetzt* werden, hier das Paar-Wörterbuch, das Wörterbücher für die Paarelemente benötigt.

Eine weitere Situation, die auftreten kann, ist, dass Wörterbücher für Superklassen aus einem Wörterbuch extrahiert werden müssen:

Beispiel 2.5:
Angenommen, man hat folgende Funktion, die sowohl Klassenmethoden der `Eq`-Klasse als auch der `Ord`-Klasse benutzt:

```
fun5 x y = x == y || x <= y
```

Die Typinferenz gibt uns folgende Typen an:

```
(fun5 :: Ord a => a -> a -> Bool) x y
  = ((==) :: Eq a  => a -> a -> Bool) x y ||
    ((<=) :: Ord a => a -> a -> Bool) x y
```

Nun wird auf der linken Seite von `fun5` nur das Wörterbuch für die `Ord`-Klasse eingefügt, nicht aber das Wörterbuch für die `Eq`-Klasse. Dieses kann aber erhalten werden, indem es aus dem Wörterbuch der `Ord`-Klasse mit Hilfe der entsprechenden Selektionsfunktion extrahiert wird. Es ergibt sich folgender Code:

```
fun5 dict.Ord.a x y = (==) (sel.Ord.Eq dict.Ord.a) x y
  || (<=) dict.Ord.a x y
```

Die Extraktion von Superklassenwörterbüchern und das Zusammensetzen von Wörterbüchern können auch gemeinsam nötig sein. Außerdem können Wörterbücher auch aus selber schon zusammengesetzten Wörterbüchern zusammengesetzt werden.

2.3.4. Wörterbücher und funktional-logische Programmierung

Bei der Implementierung von Typklassen mit Wörterbüchern muss an einer Stelle Acht gegeben werden. Um das Problem zu verstehen, müssen zunächst zwei Begriffe eingeführt werden; dies erfolgt im Folgenden anhand eines Beispiels:

Es sei wieder die `coin`-Funktion gegeben, und zusätzlich eine Funktion `twice`:

```
coin = 0 ? 1
twice x = (x, x)
```

Was ist nun das Ergebnis von „`twice coin`“? Es gibt zwei Möglichkeiten:

1. *Call-time-choice-Semantik*: Beim Aufruf der Funktion `twice` wird das Argument festgelegt. Mögliche Ergebnisse sind also (0, 0) und (1, 1).
2. *Run-time-choice-Semantik*: Ein Wert wird an der Stelle ausgewertet, an der er auftritt. Hier sind folgende Ergebnisse möglich: (0, 0), (0, 1), (1, 0) und (1,1).

Die üblicherweise in funktional-logischen Programmiersprachen gewünschte und implementierte Semantik ist die Call-time-choice-Semantik, da sie unter anderem häufiger der Intuition entspricht.

Bei der Implementierung von Typklassen mit Wörterbüchern kann nun der Fall eintreten, dass aufgrund der Call-time-choice-Semantik Werte *verloren gehen* [Lux]. Dies kann bei der Verwendung von *nullstelligen* Klassenmethoden geschehen. Dazu das folgende Beispiel:

```
class Arb a where
  arb :: a

instance Arb Bool where
  arb = False
  arb = True

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

arbLB = arbL :: [Bool]
```

Die Klasse `Arb` definiert eine nullstellige Methode, die beliebige Werte eines Typen zurückgeben soll. Die Funktion `arbL` konstruiert eine Liste, die aus zwei beliebigen Elementen eines Typs bestehen soll. `arbLB` schließlich erzeugt eine solche Liste für den Typ `Bool`.

Die erwarteten Ergebnisse der Funktion `arbLB` sind `[False, False]`, `[False, True]`, `[True, False]` und `[True, True]`. Tatsächlich werden aber nur die Werte `[False, False]` und `[True, True]` ermittelt. Der Grund dafür wird deutlich, wenn man das übersetzte Programm ohne Typklassenelemente betrachtet:

```
type Dict.Arb a = a

sel.Arb.arb :: Dict.Arb a -> a
sel.Arb.arb x = x

impl.Arb.Bool.arb :: Bool
impl.Arb.Bool.arb = False
impl.Arb.Bool.arb = True

dict.Arb.Bool :: Dict.Arb Bool -- also Bool
dict.Arb.Bool = impl.Arb.Bool.arb

arbL :: Dict.Arb a -> [a] -- also a -> [a]
arbL dict.Arb.a =
  [sel.Arb.arb dict.Arb.a, sel.Arb.arb dict.Arb.a]

arbLB = arbL dict.Arb.Bool
```

Wird also nun `arbLB` aufgerufen, so wird der Funktion `arbL` das Wörterbuch für `Arb` und `Bool` übergeben, *das aber im vorliegenden Fall nur aus einer Konstanten besteht,*

die die Werte True oder False annehmen kann. Aufgrund der Call-time-choice-Semantik wird der Parameter `dict.Arb.a` beim Aufruf der Funktion `arbL` an das übergebene Wörterbuch gebunden, also hier an `True` oder `False`. Somit bezeichnen die „`dict.Arb.a`“'s auf der rechten Seite von `arbL` immer denselben booleschen Wert, und es werden nur zwei der vier gewünschten Werte ausgegeben.

Eine Lösung für dieses Problem besteht darin, dass allen nullstelligen Klassenmethoden ein weiteres Unit-Argument hinzugefügt wird, sodass die Klassenmethoden zu Funktionen werden, und nicht mehr Konstanten sind. Mit dieser Korrektur lautet der übersetzte Code nun folgendermaßen:

```
type Dict.Arb a = () -> a

sel.Arb.arb :: Dict.Arb a -> () -> a
sel.Arb.arb x = x

impl.Arb.Bool.arb :: () -> Bool
impl.Arb.Bool.arb () = False
impl.Arb.Bool.arb () = True

dict.Arb.Bool = Dict.Arb Bool -- also () -> Bool
dict.Arb.Bool = impl.Arb.Bool.arb

arbL :: Dict.Arb a -> [a] -- also (() -> a) -> [a]
arbL dict.Arb.a =
  [sel.Arb.arb dict.Arb.a (), sel.Arb.arb dict.Arb.a ()]

arbLB = arbL dict.Arb.Bool
```

Zwar wird hier der Wörterbuchparameter beim Aufruf von `arbL` wieder an einen Wert gebunden, diesmal steht aber in `dict.Arb.a` eine Funktion, und keine Konstante. Diese Funktion wird dann in der Liste an beiden Stellen ausgewertet, indem sie auf das Unit-Argument angewandt wird. Da diese Auswertungen unabhängig voneinander geschehen, werden nun wieder alle gewünschten Werte berechnet.

2.4. Formale Beschreibung der Implementierung mit Wörterbüchern

Bei der Implementierung von Typklassen mit Hilfe von Wörterbüchern wird eine *Quellcode-Transformation* durchgeführt, die ein Programm mit Typklassenelementen in ein Programm ohne Typklassenelemente umwandelt. Dieses Programm ist dann gültig im *Damas-Milner-Typsystem.* Die Transformation erfolgt in drei Phasen (siehe Abb. 2.13): Zuerst werden für alle algebraischen Datentypen, für die eine Deriving-Klausel angegeben wurde, abgeleitete Instanzen erstellt. Anschließend werden Instanz- und Klassendefinitionen transformiert, sodass danach ein Programm ohne Instanz- und Klassendefinitionen

vorliegt; in den Typsignaturen treten aber noch Kontexte auf. Abschließend werden alle inferierten Kontexte herangezogen, um Wörterbücher und Wörterbuchparameter einzufügen. Das so entstandene Programm enthält keine Typklassenelemente mehr.

Ein größeres Fallbeispiel für die Phasen 2 und 3 ist in Anhang A.3 zu finden.

2.4.1. Phase 1: Erstellung der abgeleiteten Instanzen

2.4.1.1. `Eq`-Klasse

Ableitung 2.1 (`Eq`-Klasse)**:**
Gegeben sei folgende allgemeine Definition eines ADTs:

$$\texttt{data T a}_1 \ \ldots\ \texttt{a}_n \texttt{ = T}_1\ \tau_{k_{11}}\ \ldots\ \tau_{k_{1l_1}} \texttt{ | } \ldots \texttt{ | T}_m\ \tau_{k_{m1}}\ \ldots\ \tau_{k_{ml_m}}$$
$$\texttt{deriving Eq}$$

mit $n \geq 0$, $m \geq 1$ (für Datentypen ohne Konstruktoren werden keine Instanzen abgeleitet), $l_i \geq 0$ für $i \in [1, m]$[1] (Konstruktoren können nullstellig sein). Außerdem muss für alle $k_{ij}, i \in [1, m], j \in [1, l_i]$ gelten: $\text{vars}(\tau_{k_{ij}}) \subseteq \{a_i | i \in [1, n]\}$ (es dürfen auf der rechten Seite der Definition nur Typvariablen der linken Seite auftreten; vars sei die Funktion, die alle Typvariablen in einem Typausdruck zurückgibt).

Die erzeugte Instanz lautet:

$$\texttt{instance (Eq a}_1\texttt{, ..., Eq a}_n\texttt{) => Eq (T a}_1\ \ldots\ \texttt{a}_n\texttt{) where}$$
$$\texttt{equation}_{11}$$
$$\vdots$$
$$\texttt{equation}_{1m}$$
$$\vdots$$
$$\texttt{equation}_{m1}$$
$$\vdots$$
$$\texttt{equation}_{mm}$$

wobei die $\texttt{equation}_{ij}$, $(i, j) \in \{(i', j') | i' \in [1, m], j' \in [1, m]\}$, wie folgt gegeben sind:

$$\texttt{equation}_{ij} = \begin{cases} \texttt{T}_i\ \texttt{x}_1\ \ldots\ \texttt{x}_{l_i}\ \texttt{==}\ \texttt{T}_i\ \texttt{y}_1\ \ldots\ \texttt{y}_{l_i}\ \texttt{=} & \\ \quad \texttt{x}_1\ \texttt{==}\ \texttt{y}_1\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ \texttt{x}_{l_i}\ \texttt{==}\ \texttt{y}_{l_i} & \text{falls } i = j \text{ und } l_i \geq 1 \\ \texttt{T}_i\ \texttt{==}\ \texttt{T}_i\ \texttt{=}\ \texttt{True} & \text{falls } i = j \text{ und } l_i = 0 \\ \texttt{T}_i\ \underbrace{\texttt{\_ ... \_}}_{l_i}\ \texttt{==}\ \texttt{T}_j\ \underbrace{\texttt{\_ ... \_}}_{l_j}\ \texttt{=}\ \texttt{False} & \text{falls } i \neq j \end{cases}$$

Wie schon früher erwähnt, müssen in der funktional-logischen Programmierung für *alle* Paare $\{(\texttt{T}_i, \texttt{T}_j) \mid i \in [1, m], j \in [1, m]\}$ Gleichungen generiert werden. Der Grund dafür ist, dass überlappende Gleichungen ausgeschlossen werden müssen, da sonst die

[1] $[a, b]$ stehe für das Intervall $\{a, a + 1, \ldots, b - 1, b\}$

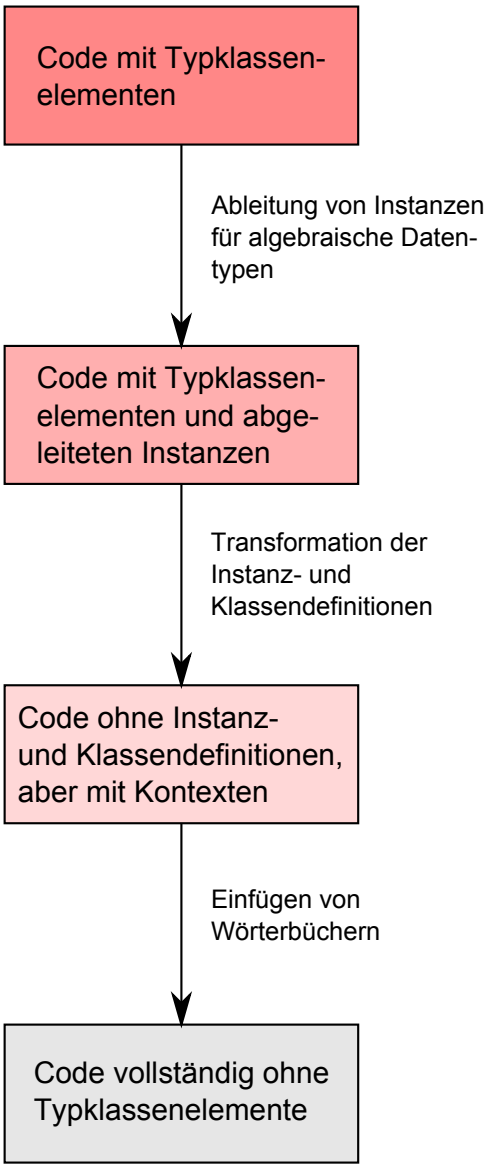

Abbildung 2.13: Transformationsschritte

```
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
```

Abbildung 2.14: `Ord`-Klasse

Gleichheitsoperation nicht-deterministisch würde, und in manchen Fällen sowohl `False` als auch `True` zurückgeben würde. Die „Abkürzung“, nur Paare ($\mathtt{T}_i$, $\mathtt{T}_j$) mit $i = j$ zu vergleichen, und für alle anderen Kombinationen eine einzige Gleichung der Form „`_ == _ = False`“ zu verwenden, ist hier nicht erlaubt.

2.4.1.2. `Ord`-Klasse

Um die `Ord`-Klasse (siehe Abbildung 2.14) zu implementieren, muss entweder die `<=`-Funktion oder die `compare`-Funktion angegeben werden. Alle anderen Operationen werden durch die Default-Methoden der Klasse `Ord` implementiert.

Hier gebe ich eine Implementierung der `<=`-Funktion an:

Ableitung 2.2 (`Ord`-Klasse)**:**
Gegeben sei wieder der algebraische Datentyp aus Ableitung 2.1. Für diesen wird die folgende `Ord`-Instanz generiert:

```
instance (Ord a_1, ..., Ord a_n) => Ord (T a_1 ... a_n) where
  equation_11
  ⋮
  equation_1m

  ⋮

  equation_m1
  ⋮
  equation_mm
```

wobei die Gleichungen $\texttt{equation}_{ij}$ wie folgt gegeben sind:

$$\texttt{equation}_{ij} = \begin{cases} \texttt{T}_i\ \texttt{\_}\ \ldots\ \texttt{\_}\ \texttt{<=}\ \texttt{T}_j\ \texttt{\_}\ \ldots\ \texttt{\_}\ \texttt{=}\ \texttt{True} & \text{falls } i < j \\ \texttt{T}_i\ \texttt{\_}\ \ldots\ \texttt{\_}\ \texttt{<=}\ \texttt{T}_j\ \texttt{\_}\ \ldots\ \texttt{\_}\ \texttt{=}\ \texttt{False} & \text{falls } i > j \end{cases}$$

wobei die Anzahl der Unterstriche für $\texttt{T}_i$ l_i beträgt und die Anzahl der Unterstriche für $\texttt{T}_j$ l_j.

Ansonsten ($i = j$) werden folgende Gleichungen generiert:

$$\texttt{equation}_{ii} = \begin{cases} \texttt{T}_i\ \texttt{<=}\ \texttt{T}_i\ \texttt{=}\ \texttt{True} & \text{wenn } l_i = 0 \\ \texttt{T}_i\ \texttt{x}_1\ \ldots\ \texttt{x}_{l_i}\ \texttt{<=}\ \texttt{T}_i\ \texttt{y}_1\ \ldots\ \texttt{y}_{l_i}\ \texttt{=} & \\ \quad \texttt{undTerm}_1\ \texttt{||}\ \ldots\ \texttt{||}\ \texttt{undTerm}_{l_i} & \text{ansonsten} \end{cases}$$

Die $\texttt{undTerm}_k$ sind wie folgt gegeben:

$$\texttt{undTerm}_k = \begin{cases} \texttt{x}_1\ \texttt{==}\ \texttt{y}_1\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ \texttt{x}_{k-1}\ \texttt{==}\ \texttt{y}_{k-1}\ \texttt{\&\&}\ \texttt{x}_k\ \texttt{<}\ \texttt{y}_k & \text{wenn } k \neq l_i \\ \texttt{x}_1\ \texttt{==}\ \texttt{y}_1\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ \texttt{x}_{k-1}\ \texttt{==}\ \texttt{y}_{k-1}\ \texttt{\&\&}\ \texttt{x}_k\ \texttt{<=}\ \texttt{y}_k & \text{wenn } k = l_i \end{cases}$$

Ein Beispiel für eine Ableitung der `Ord`-Instanz wurde bereits in Abb. 2.2 angegeben.

2.4.1.3. `Show`-Klasse

Ableitung 2.3 (`Show`-Klasse)**:**
Es sei der folgende algebraische Datentyp gegeben:

```
data T a_1 ... a_n = C_1 | ... | C_m
  deriving Show
```

wobei die $\texttt{C}_i$ eine der beiden folgenden Formen haben:

- $\texttt{T}_i\ \tau_{k_{i1}}\ \ldots\ \tau_{k_{il_i}}$, $l_i \geq 0$ ($\texttt{C}_i$ ist eine normale Datenkonstruktordeklaration)
- $\tau_{i1} \circ_i \tau_{i2}$ ($\texttt{C}_i$ definiert einen Datenkonstruktor $\circ_i$, der als binärer Operator verwendet werden kann)

Für die $\circ_i$ können im Quelltext Präzedenzen angegeben werden. So gibt eine *Fixity*-Deklaration der Form „`infix` p $\circ_i$“ an, dass der Operator $\circ_i$ die Präzedenz p haben soll. Wird keine Fixity-Deklaration für einen Operator angegeben, so wird eine Präzedenz von 9 angenommen.

Im Folgenden sei prec die Funktion, die einem Operator $\circ_i$ seine Präzedenz zuordnet. Es gilt also $\text{prec}(\circ_i) = p$, wenn eine Fixity-Deklaration der Form „`infix` p $\circ_i$“ existiert, ansonsten gilt $\text{prec}(\circ_i) = 9$.

Für den obigen algebraischen Datentyp wird die folgende Instanzdefinition abgeleitet:

```
instance (Show a1, ..., Show an)
    => Show (T a1 ... an) where
  equation1
  ⋮
  equationm
```

wobei für die `equation`$_i$, $i \in [1, m]$ drei Fälle möglich sind:

- `equation`$_i$ =
  ```
  showsPrec _ Ti = showString "Ti"
  ```
  falls `C`$_i$ eine normale Datenkonstruktordeklaration ist und $l_i = 0$ gilt, der Datenkonstruktor also nullstellig ist. Nullstellige Konstruktoren werden *immer* ohne einschließende Klammern angezeigt.

- `equation`$_i$ =
  ```
  showsPrec d (Ti x1 ... xli) =
    showParen (d > appPrec) (
      showString "Ti" .
      showString " " .
      showsPrec (appPrec + 1) x1 .
      showString " " .
      showsPrec (appPrec + 1) x2 .
      showString " " .
      ⋮
      showString " " .
      showsPrec (appPrec + 1) xli)
  ```
  wenn `C`$_i$ eine normale Datenkonstruktordeklaration ist, und $l_i > 0$. `appPrec` sei die Präzedenz der *Applikation*. Diese hat den Wert 10, liegt also um eine Stufe höher als die Präzedenz des am stärksten bindenden Operators.

  Um die Parameter des Datenkonstruktors in eine Zeichenkette umzuwandeln, wird der Funktion `showsPrec` in allen Aufrufen die Präzedenz `appPrec` + 1

übergeben. Dies bedeutet, dass *immer* Klammern um die Parameter des Datenkonstruktors gesetzt werden (außer um nullstellige Datenkonstruktoren, Listen und Tupel). Dies ist notwendig, da – wenn die Parameter wieder aus Datenkonstruktoren und Parametern zusammengesetzt sind – sonst diese Datenkonstruktoren und Parameter *alle* als Parameter von T_i aufgefasst würden.

- `equation`$_i$ =

```
showsPrec d (x ∘ᵢ y) =
  showParen (d > prec(∘ᵢ)) (
    showsPrec (prec(∘ᵢ) + 1) x .
    showString " " .
    showString "∘ᵢ" .
    showString " " .
    showsPrec (prec(∘ᵢ) + 1) y)
```

  wenn `C`$_i$ eine Datenkonstruktordeklaration ist, die „$\circ_i$“ definiert.

  Es wird also geprüft, ob der Operator „über“ dem gegebenen Operator eine höhere Präzedenz als die Präzedenz des gerade betrachteten Operators besitzt, und falls dies der Fall ist, werden Klammern gesetzt.

  Bei den Anwendungen der Funktion `showsPrec` auf die Parameter des Operators wird immer die Präzedenz des gerade betrachteten Operators *plus 1* übergeben. Dadurch wird erreicht, dass Folgen von Operatoranwendungen von $\circ_i$ explizit geklammert werden.

2.4.1.4. `Enum`-Klasse

Instanzen der Klasse `Enum` können nur dann automatisch für algebraische Datentypen abgeleitet werden, wenn deren Datenkonstruktoren nullstellig sind, also keine zusätzlichen Parameter haben. Solche Datentypen nennt man *Enumerationen* oder *Aufzählungen*. Für Enumerationen können folgendermaßen Instanzen abgeleitet werden:

Ableitung 2.4 (`Enum`-Klasse)**:**
Gegeben sei die folgende Enumeration:

```
data T = T₁ | ... | Tₙ
  deriving Enum
```

mit $n \geq 1$ (für leere Enumerationen kann keine `Enum`-Instanz abgeleitet werden).
Die erzeugte `Enum`-Instanz lautet wie folgt:

```
instance Enum T where
  succ T₁ = T₂
  ⋮
  succ Tₙ₋₁ = Tₙ
  succ Tₙ = error "no successor for Tₙ"
```

```
pred T1 = error "no predecessor for T1"
pred T2 = T1
⋮
pred Tn = Tn−1

fromEnum T1 = 0
fromEnum T2 = 1
⋮
fromEnum Tn = n − 1

toEnum k
  | k == 0 = T1
  | k == 1 = T2
  ⋮
  | k == n − 1 = Tn
  | otherwise = error "toEnum: index out of range"

-- enumFrom :: T -> [T]
enumFrom x =
  map toEnum (enumFromTo' (fromEnum x) (n − 1))

-- enumFromTo :: T -> T -> [T]
enumFromTo x y =
  map toEnum (enumFromTo' (fromEnum x) (fromEnum y))

-- enumFromThen :: T -> T -> [T]
enumFromThenTo x y =
  map toEnum
    (enumFromThenTo' (fromEnum x) (fromEnum y)
      (if fromEnum y < fromEnum x then 0 else n − 1))

-- enumFromThenTo :: T -> T -> T -> [T]
enumFromThenTo x y z =
  map toEnum
    (enumFromThenTo' (fromEnum x) (fromEnum y)
      (fromEnum z))
```

Die Hilfsfunktionen `enumFromTo'` und `enumFromThenTo'` arbeiten auf Integer-Werten, und erzeugen Aufzählungen von Integern gemäß ihrer Namen.

Zusätzlich zu den schon in Beispiel 2.3 angegebenen Klassenfunktionen `succ`, `pred`, `fromEnum`, `toEnum` sind hier auch noch die Implementierungen der Klassenfunktionen

`enumFrom`, `enumFromTo`, `enumFromThen` und `enumFromThenTo` angegeben. Die ersten beiden Funktionen `succ` und `pred` ermitteln den Nachfolger bzw. den Vorgänger eines Elements einer Enumeration. `fromEnum` und `toEnum` konvertieren zwischen der Position eines Elementes in der Enumeration und dem Element selber. `enumFrom`, `enumFromTo`, `enumFromThen` und `enumFromThenTo` generieren Aufzählungen der Elemente einer Enumeration, eventuell mit einer gewissen Schrittweite (bei den „`Then`"-Varianten); Beispiele dafür sind:

```
data T = T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8

enumFrom T5 = [T5, T6, T7, T8]
enumFromTo T2 T4 = [T2, T3, T4]
enumFromThen T1 T3 = [T1, T3, T5, T7]
enumFromThenTo T2 T4 T8 = [T2, T4, T6, T8]
enumFromThenTo T8 T6 T2 = [T8, T6, T4, T2]
```

Zu beachten ist hier, dass die Aufzählung der Enumerationselemente auch „rückwärts" geschehen kann, und dass die Aufzählung immer an der oberen bzw. an der unteren Grenze der Enumeration endet.

Um die Schrittweite korrekt handzuhaben, werden in den `enum...`-Funktionen zunächst die übergebenen Parameter in Integer umgewandelt, und danach die Hilfsfunktionen `enumFromTo'` bzw. `enumFromThenTo'` aufgerufen, die auf Integern arbeiten. Anschließend wird die so erhaltene Aufzählung von Integerwerten mittels `toEnum` wieder in eine Aufzählung der Elemente der Enumeration umgewandelt. Zu beachten ist bei der Implementierung von `enumFromThenTo`, dass die Schrittweite eventuell negativ sein kann, und deshalb als Grenze entweder die obere Grenze oder die untere Grenze der Enumeration ausgewählt werden muss.

Auch bei der `Enum`-Instanz muss wieder berücksichtigt werden, dass kein Nichtdeterminismus auftritt. Dies wäre zum Beispiel der Fall, wenn die `toEnum`-Funktion folgendermaßen implementiert würde:

```
toEnum 0 = T_1
toEnum 1 = T_2
⋮
toEnum n − 1 = T_n
toEnum _ = error "toEnum: index out of range"
```

Da die letzte Regel mit allen anderen überlappt, wird bei einem Aufruf von `toEnum` unter anderem immer die letzte Regel ausgeführt, sodass immer ein Fehler auftritt.

2.4.1.5. `Bounded`-Klasse

Die Bounded-Klasse ist wie folgt definiert:

```
class Bounded a
  minBound, maxBound :: a
```

`Bounded`-Instanzen können für zwei Varianten von algebraischen Datentypen abgeleitet werden:

- Für *Enumerationen*
- Für algebraische Datentypen, die *genau einen* Datenkonstruktor besitzen

Ableitung für eine Enumeration

Ableitung 2.5:
Es sei der algebraische Datentyp wie folgt gegeben:

```
data T = T_1 | ... | T_n
  deriving Bounded
```

und es gelte $n \geq 1$ (für leere Enumerationen kann keine `Bounded`-Instanz abgeleitet werden). Dann lautet die abgeleitete Instanz:

```
instance Bounded T where
  minBound = T_1
  maxBound = T_n
```

Ableitung für einen Datentyp mit einem Konstruktor

Ableitung 2.6:
Es sei der algebraische Datentyp wie folgt gegeben:

```
data T a_1 ... a_n = C τ_1 ... τ_k
  deriving Bounded
```

mit $n, k \geq 0$, wobei die τ_i beliebige Typen sind. Dann wird folgende `Bounded`-Instanz generiert:

```
instance (Bounded a_1, ..., Bounded a_n) =>
    Bounded (T a_1 ... a_n) where
  minBound = C minBound ... minBound
  maxBound = C maxBound ... maxBound
```

wobei auf den rechten Seiten der beiden Gleichungen `minBound` bzw. `maxBound` k Mal auftreten.

Es werden also dem Datenkonstruktor die oberen/unteren Grenzen der Parametertypen τ_i des Datenkonstruktors übergeben. Deshalb müssen auch für alle Parametertypen `Bounded`-Instanzen existieren.

2.4.2. Phase 2: Transformation der Typklassen- und Instanzdefinitionen

2.4.2.1. Typklassendefinitionen

Die Transformation von Typklassendefinitionen geschieht wie folgt:

Transformation 2.3 (Typklassendefinitionen, Teil 1)**:**
Es sei die folgende Typklassendefinition gegeben:

```
class (SC_1 a, ..., SC_k a) => C a where
  -- Typsignaturen der Klassenmethoden
  fun_1 :: τ_1
  ⋮
  fun_n :: τ_n

  -- Default methods
  implementation of fun_{m_1}
  ⋮
  implementation of fun_{m_l}
```

mit $n \geq 0$, $k \geq 0$, $m_i \in [1, n]$ für $i \in [1, l]$.

Zuerst wird für diese Klassendefinition der Wörterbuchtyp generiert:

```
type Dict.C a = (Dict.SC_1 a, ..., Dict.SC_k a, τ'_1, ..., τ'_n)
```

Für alle $i \in [1, n]$ werden außerdem die folgenden Selektionsfunktionen für Klassenmethoden generiert:

```
sel.C.fun_i :: Dict.C a -> τ'_i
sel.C.fun_i (sc_1, ..., sc_k, t_1, ..., t_n) = t_i
```

Für die τ'_i gelte:

$$\tau'_i = \begin{cases} \texttt{() -> } \tau_i & \text{wenn } \texttt{fun}_i \text{ nullstellig} \\ \tau_i & \text{sonst} \end{cases}$$

Bei nullstelligen Klassenmethoden muss ein zusätzliches Unit-Argument eingefügt werden; das Unit-Argument hat den Zweck, dass die nullstellige Klassenmethode von einer Konstante in eine Funktion umgewandelt wird. Somit können – wie in Abschnitt 2.3.4 beschrieben – bei der Auswertung mit der *Call-time-choice-Semantik* keine Ergebnisse mehr verloren gehen.

Weiterhin werden Selektionsfunktionen für direkte Superklassen erstellt; für alle $j \in [1, k]$ die folgenden Funktionen:

```
sel.C.SC_j :: Dict.C a -> Dict.SC_j a
sel.C.SC_j (sc_1, ..., sc_k, t_1, ..., t_n) = sc_j
```

Außerdem werden für die Implementierungen von $\texttt{fun}_{m_q}$, $q \in [1, l]$ Top-Level-Funktionen $\texttt{def.C.fun}_{m_q}$ mit den jeweiligen Code, der in der Klassendefinition angegeben wurde, generiert. Bei Implementierungen von nullstelligen Klassenmethoden wird zusätzlich ein Unit-Argument eingefügt.

Zusätzlich zu den Selektionsfunktionen für direkte Superklassen und Klassenmethoden werden auch Selektionsfunktionen für *nicht-direkte* Superklassen generiert. Ein Fall, in

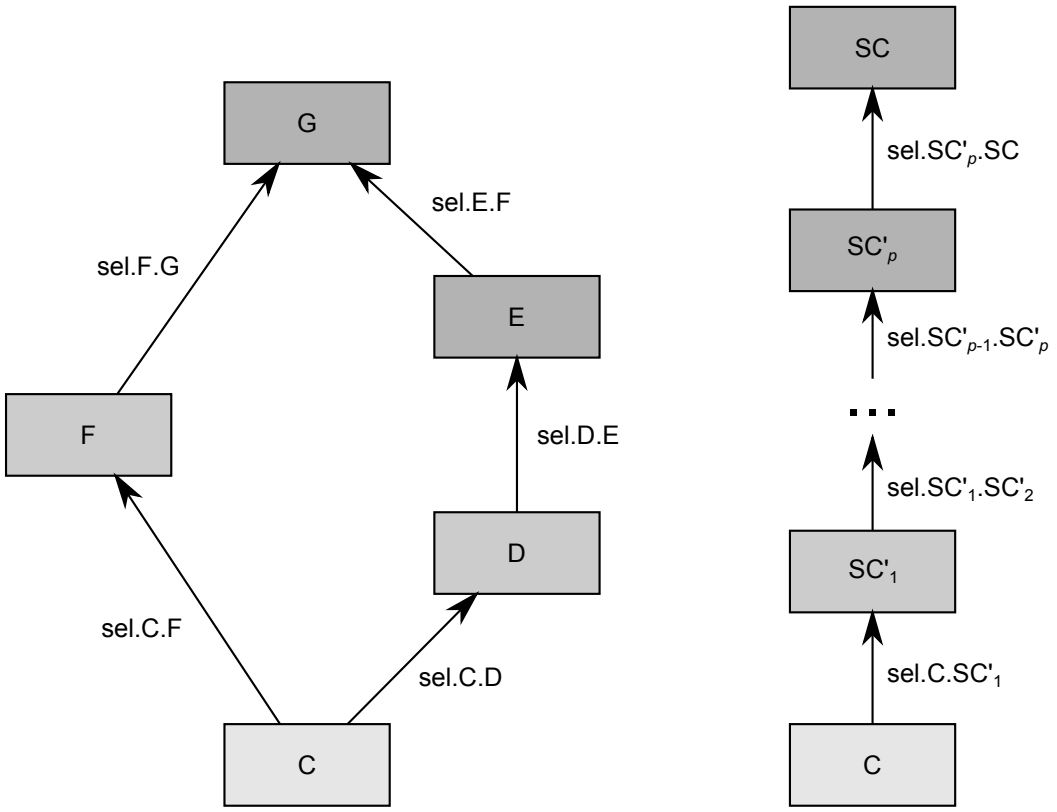

Abbildung 2.15: Superklassenbeziehungen (zu Beispiel/zu allgemeiner Typklassentransformation)

dem nicht-direkte Superklassen auftreten, ist auf der linken Seite von Abb. 2.15 dargestellt. In der dort gegebenen Klassenhierarchie ist zum Beispiel die Klasse `G` eine nicht-direkte Superklasse von `C`. Für die Klassen `C` und `G` wird also eine Selektionsfunktion erstellt, mit Hilfe derer aus dem Wörterbuch der Klasse `C` das Wörterbuch der Klasse `G` extrahiert werden kann. Die Selektionsfunktion wird implementiert, indem alle *direkten* Superklassenselektionsfunktionen verkettet werden. Im gegebenen Beispiel würde die Implementierung also folgendermaßen lauten:

```
sel.C.G :: Dict.C a -> Dict.G a
sel.C.G = sel.F.G . sel.C.F
```

Auch die folgende Implementierung wäre denkbar:

```
sel.C.G :: Dict.C a -> Dict.G a
sel.C.G = sel.E.G . sel.D.E . sel.C.D
```

Man wählt aber zweckmäßig immer den kürzesten Pfad (oder einen der kürzesten Pfade) in der Klassenhierarchie.

Der allgemeine Fall ist wie folgt:

Transformation 2.4 (Typklassendefinitionen, Teil 2)**:**
Sei `C` gegeben wie zuvor. Für alle nicht-direkten Superklassen wird dann ebenfalls Code generiert.

Sei `SC` eine solche nicht direkte Superklasse von `C`, es existiert also ein gerichteter Pfad `C`, $\texttt{SC}'_1$, ..., $\texttt{SC}'_p$, `SC` in dem Graphen der Superklassenbeziehungen mit $p \geq 1$ (siehe Abb. 2.15, rechte Seite). Dieser Pfad sei weiterhin der kürzeste Pfad von `C` nach `SC`. Dann wird für `C` und `SC` folgende Selektionsfunktion generiert:

```
-- Selektionsfunktionen für nicht-direkte Superklassen
sel.C.SC :: Dict.C a -> Dict.SC a
sel.C.SC' = sel.SC'_p.SC . sel.SC'_{p-1}.SC'_p . ··· .
  sel.SC'_1.SC'_2 . sel.C.SC'_1
```

2.4.2.2. Instanzdefinitionen

Eine Instanzdefinition wird folgendermaßen transformiert:

Transformation 2.5 (Instanzdefinitionen)**:**
Gegeben seien eine Klasse `C` und eine `C-T`-Instanzdefinition wie folgt:

```
class (SC_1 a, ..., SC_p a) => C a where
  fun_1 :: τ_1
  ⋮
  fun_q :: τ_q

  implementation of fun_{s_1}
  ⋮
  implementation of fun_{s_t}

instance (C_1 a_{k_1}, ..., C_n a_{k_n}) => C (T a_1 ... a_m) where
  implementation of fun_{l_1}
  ⋮
  implementation of fun_{l_r}
```

mit $k_i \in [1, m]$ und $l_j \in [1, q]$ für $j \in [1, r]$.

Die in der Instanzdefinition angegebenen Implementierungen – es müssen nicht notwendigerweise für alle Klassenmethoden Implementierungen angegeben sein – werden in Top-Level-Funktionen transformiert, indem der Code eins zu eins kopiert wird, und lediglich die Namen der Funktionen von $\texttt{fun}_{l_i}$ auf $\texttt{impl.C.T.fun}_{l_i}$ geändert werden. Es wird also für alle in der Instanzdefinition implementierten Klassenmethoden $\texttt{fun}_{l_i}$, $i \in [1, r]$ folgender Code generiert:

```
impl.C.T.fun_{l_i} :: (C_1 a_{k_1}, ..., C_n a_{k_n}) => τ'_{l_i}
<Top-Level Funktion impl.C.T.fun_{l_i} mit Implementierung
 von fun_{l_i}>
```

wobei

$$\tau'_{l_i} = \begin{cases} \texttt{() ->}\ \tau_{l_i}[\texttt{a}|\texttt{T a}_1\ \ldots\ \texttt{a}_m]^1 & \text{wenn } \texttt{fun}_{l_i} \text{ nullstellig} \\ \tau_{l_i}[\texttt{a}|\texttt{T a}_1\ \ldots\ \texttt{a}_m] & \text{sonst} \end{cases}$$

Weiterhin werden, wenn $\texttt{fun}_{l_i}$ nullstellig ist, nicht nur die Implementierungen aus der Instanzdefinition eins zu eins kopiert, sondern es wird auch auf allen linken Seiten der einzelnen Regeln ein Unit-Pattern eingefügt. Diese Änderung entspricht der äquivalenten Änderung der Typsignatur.

Im Folgenden seien folgende Mengen definiert:

$$\begin{aligned} \text{implemented} &:= \{l_i | i \in [1, r]\} \\ \text{defaultMs} &:= \{s_i | i \in [1, t]\} \end{aligned}$$

Wenn keine Implementierung für eine Methode $\texttt{fun}_i$, angegeben wurde (also $i \in [1, q] \backslash \text{implemented}$), und keine Default-Methode für $\texttt{fun}_i$ existiert ($i \notin \text{defaultMs}$), wird ein Stub generiert, der eine Fehlermeldung ausgibt:

```
impl.C.T.fun_i =
  error ("No implementation for class method fun_i "
    ++ "in C-T-instance")
```

Das Wörterbuch wird zusammengesetzt aus den Wörterbüchern für die Superklassen von `C` und den Typ `T` und aus den Implementierungsfunktionen und den Default-Methoden, je nachdem wie die Klassen- und Instanzdefinitionen beschaffen sind:

```
dict.C.T :: (C_1 a_{k_1}, ..., C_n a_{k_n}) => Dict.C (T a_1 ... a_m)
dict.C.T = (dict.SC_1.T, ..., dict.SC_p.T,
            method_1, ..., method_q)
```

wobei

$$\texttt{method}_i = \begin{cases} \texttt{impl.C.T.fun}_i & \text{wenn } i \in \text{implemented} \\ \texttt{def.C.fun}_i & \text{wenn } i \notin \text{implemented und } i \in \text{defaultMs} \\ \texttt{impl.C.T.fun}_i & \text{wenn } i \notin \text{implemented und } i \notin \text{defaultMs} \end{cases}$$

Hier ist es wichtig, dass für das Wörterbuch eine konkrete Typsignatur angegeben wird, denn dadurch wird erreicht, dass die eventuell eingesetzten Default-Methoden einen spezielleren Typ erhalten als sie ursprünglich besitzen. Damit wiederum wird erreicht, dass in der Wörterbuchtransformation das Wörterbuch `dict.C.T` an die Default-Methoden weitergereicht wird, so dass diese mit den Implementierungen der `C-T`-Instanz arbeiten (siehe nächsten Abschnitt und das Fallbeispiel in Anhang A.3).

[1] $\tau[\texttt{a}|\sigma]$ bezeichne die Substitution von `a` durch σ in τ

2.4.3. Phase 3: Einfügen von Wörterbüchern

Für das Einfügen von Wörterbüchern ist es notwendig, dass für alle Funktionen deren Typen ermittelt wurden. Es muss dabei sowohl der allgemeine Typ einer Funktion bekannt sein, als auch der konkrete Typ einer Funktion an einer Stelle bei der sie benutzt wird. Der konkrete Typ kann sich aufgrund des parametrischen Polymorphismus vom allgemeinen Typ der Funktion unterscheiden.

Für das Einfügen der Wörterbücher werden die ermittelten Kontexte herangezogen. Der Kontext einer Funktion wird ermittelt, indem zuerst alle Kontexte für die im Rumpf enthaltenen Funktionen ermittelt und diese dann vereinigt werden. Anschließend wird eine Kontextreduktion auf dem Kontext durchgeführt.

Im Folgenden werden die Kontexte in Mengenschreibweise geschrieben, wenn die Reihenfolge der Elemente keine Rolle spielt; ansonsten wird die Listenschreibweise $(\pi_1, \ldots, \pi_n)$ verwendet.

Beispiel 2.6 (Kontextermittlung durch Typcheck)**:**
Es sei die folgende Funktion gegeben:

```
f x y = (x, [y], [1]) == (x, [y], [1])
  && (x, y) <= (x, y)
```

Dann werden für die Funktionen im Rumpf folgende Kontexte ermittelt, die hier tiefgestellt angegeben werden:

```
f x y = (x, [y], [1]) ==Eq (a, [b], [Int]) (x, [y], [1])
  && (x, y) <=Ord (a, b) (x, y)
```

Die Vereinigung aller Kontexte aus dem Rumpf von `f` ergibt den Kontext {`Eq (a, [b], [Int])`, `Ord (a, b)`}. Nach der Kontextreduktion lautet dieser Kontext {`Ord a`, `Ord b`}, wie leicht mit den im Abschnitt 2.2.3.1 angegebenen Inferenzregeln geprüft werden kann.

Folglich hat die Funktion `f` den Kontext {`Ord a`, `Ord b`}.

Jedes Kontextelement steht nun für ein Wörterbuch, das an der entsprechenden Stelle benötigt wird. Bei einer Funktionsdeklaration werden entsprechend dem reduzierten Kontext Parameter für die Wörterbücher eingefügt:

Transformation 2.6:
Gegeben sei eine Funktion `f`, die den reduzierten Kontext (`C`$_1$ `a`$_1$`, ..., C`$_n$ `a`$_n$) hat.

Jede Gleichung

```
f x1 ... xn = <Rumpf>
```

wird in folgende Gleichung transformiert:

```
f dict.C1.a1 ... dict.Cn.an x1 ... xn =
  <transformierter Rumpf>
```

wobei auf die Transformation des Rumpfes später eingegangen wird.

(avail)	$\dfrac{}{P \Vdash \texttt{dict.C.a} : C\ a}$	wenn $C\ a \in P$
(super)	$\dfrac{P \Vdash d : C\ a}{P \Vdash \texttt{sel.C.C'}\ d : C'\ a}$	wenn C' Superklasse von C ist
(inst)	$\dfrac{P \Vdash d_1 : C_1\ \tau_{k_1} \quad \ldots \quad P \Vdash d_n : C_n\ \tau_{k_n}}{P \Vdash \texttt{dict.C.}\chi\ d_1 \ldots d_n : C\ (\chi\ \tau_1 \ldots \tau_m)}$	wenn $\{k_1, \ldots, k_n\} \subseteq [1, m]$, die Instanzdefinition `instance K => C (`χ `a`$_1$ ... `a`$_m$`)` existiert und `(C`$_1$ `a`$_{k_1}$`, ..., C`$_n$ `a`$_{k_n}$`)` der Kontext ist, der durch die Reduktion von `K` entsteht

Abbildung 2.16: Inferenzregeln zum Generieren von Wörterbuch-Code

Es wird also für jedes Kontextelement ein Wörterbuchparameter eingefügt, wobei die Wörterbuchparameter sowohl die Klasse als auch die Typvariable des Kontextelements im Namen tragen. Dies ist notwendig, weil die Wörterbücher später eindeutig identifiziert werden müssen.

Man hat nun also eine bestimmte Menge an Wörterbüchern zur Verfügung, die der Funktion übergeben werden. Aus diesen Wörterbüchern müssen nun alle im Funktionsrumpf benötigte Wörterbücher konstruiert werden. Es stehen folgende Operationen zur Verfügung:

- Wörterbücher können aus anderen *zusammengebaut* werden. Dies ist notwendig für Wörterbücher von Instanzen, die einen Kontext besitzen.
- Es können Wörterbücher für Superklassen aus einem gegebenen Wörterbuch ausgewählt werden.

Es muss also Code generiert werden, der diese Operationen durchführt. Diese Generierung kann durch ein Inferenzsystem beschrieben werden [Jon92b, Kapitel 4.5]. Die Syntax der Prämissen und der Konklusionen sei wie folgt:

$$P \Vdash d : C\ \tau$$

P ist ein Kontext, d ist der generierte Code, und $C\ \tau$ ein Kontextelement. P steht für die Menge der verfügbaren Wörterbücher (jedes Kontextelement in P entspricht ja einem Wörterbuch), $C\ \tau$ ist das Kontextelement, für das Code generiert werden soll, mit dem aus den gegebenen Wörterbüchern ein Wörterbuch für $C\ \tau$ gebildet werden kann.

Die Regeln des Inferenzsystems sind in Abbildung 2.16 aufgeführt.

Mit Hilfe der angegebenen Inferenzregeln kann nun folgende Funktion definiert werden:

Definition 2.3 (Wörterbuchcodegenerierungsfunktion)**:**
Die Funktion „dictCode“ ordne einem Kontext P und einem Kontextelement C den Code zu, der nötig ist, um aus den gegebenen Wörterbüchern ein Wörterbuch für C zu erstellen. Es gelte also $\text{dictCode}(P, C) = d$ genau dann wenn $P \Vdash d : C$ mit obigem Inferenzsystem ableitbar ist.

Damit kann man schließlich die Transformation der Funktionsrümpfe angeben:

Transformation 2.7 (Transformation in Funktionsrümpfen)**:**
Es sei eine Funktion `f` mit dem reduzierten Kontext P gegeben. Es sei weiterhin `g` eine (überladene) Funktion, die im Funktionsrumpf von `f` verwendet wird, und den Kontext $P' = (C_1, \ldots, C_n)$ und den Typ τ besitzt. `g` werde außerdem auf die Elemente $\texttt{p}_1$ bis $\texttt{p}_m$ angewendet. Der Ausschnitt aus dem Rumpf lautet also:

$\ldots$ `(g ::` P' `=>` τ`)` $\texttt{p}_1 \ \ldots \ \texttt{p}_m \ \ldots$

Für die Transformation dieses Ausschnitts sind drei Fälle zu unterscheiden:

- Fall 1: `g` ist *keine* Klassenmethode.

  Dann wird dieser Ausschnitt in folgenden Code transformiert:

  $\ldots$ `g` $\text{dictCode}(P, C_1) \ \ldots \ \text{dictCode}(P, C_n)$ $\texttt{p}_1 \ \ldots \ \texttt{p}_m \ \ldots$

- Fall 2: `g` ist eine Klassenmethode der Klasse `C`, und nicht nullstellig. Dann gilt $n = 1$. Der obige Ausschnitt wird in folgenden Code transformiert:

  $\ldots$ `sel.C.g` $\text{dictCode}(P, C_1)$ $\texttt{p}_1 \ \ldots \ \texttt{p}_m \ \ldots$

- Fall 3: `g` ist eine Klassenmethode der Klasse `C`, und nullstellig. Es gilt wieder $n = 1$, und zudem $m = 0$. Der obige Ausschnitt wird in folgenden Code transformiert:

  $\ldots$ `sel.C.g` $\text{dictCode}(P, C_1)$ `()` $\ldots$

Festzuhalten ist, dass im letzten Fall noch das zusätzliche Unit-Argument eingefügt wird, wie es in den anderen Transformationen bei nullstelligen Klassenmethoden ebenfalls der Fall ist.

Es ist noch anzumerken, dass die „$\text{dictCode}(P, C_i)$“ stets in der richtigen Reihenfolge eingefügt werden, da der Typcheck garantiert, dass für eine Funktion `f` die Reihenfolge und die Anzahl der Kontextelemente bei der Benutzung von `f` der Reihenfolge und der Anzahl der Kontextelemente bei der Deklaration von `f` entspricht.

Nun ein Beispiel für den gesamten Transformationsprozess:

Beispiel 2.7:
Es sei die Funktion `f` wie in Beispiel 2.6 gegeben. Die von der Typüberprüfung ermittelten Kontexte lauten:

```
f{Ord a,Ord b} x y =
  (x, [y], [1]) ==Eq (a, [b], [Int]) (x, [y], [1])
  && (x, y) <=Ord (a, b) (x, y)
```

Zuerst werden Wörterbuchparameter für die Elemente im reduzierten Kontext von `f` eingefügt:

```
f dict.Ord.a dict.Ord.b x y = ...
```

Danach werden die Kontexte der Klassenmethoden `==` und `<=` betrachtet. Der Kontext von `==` besteht aus einem Element. Der zu generierende Code für dieses Kontextelement ist dictCode({`Ord a`, `Ord b`}, `Eq (a, [b], [Int])`). Als Ergebnis erhält man den Code `dict.Eq.(,,) (sel.Ord.Eq dict.Ord.a) (dict.Eq.[] (sel.Ord.Eq dict.Ord.b)) (dict.Eq.[] dict.Eq.Int)`, wie durch die Ableitung in Abbildung 2.17 gezeigt werden kann. Der zu generierende Code für das Kontextelement aus dem Kontext von `<=` lautet dictCode({`Ord a`, `Ord b`}, `Ord (a, b)`) = `dict.Ord.(,) dict.Ord.b dict.Ord.b`. Die entsprechende Ableitung ist ebenfalls in Abbildung 2.17 gezeigt.

Werden nun die Wörterbücher im Rumpf eingefügt, so erhält man schließlich die folgende transformierte Version von `f`:

```
f dict.Ord.a dict.Ord.b x y =
  (sel.Eq.==) (dict.Eq.(,,)
                (sel.Ord.Eq dict.Ord.a)
                (dict.Eq.[] (sel.Ord.Eq dict.Ord.b))
                (dict.Eq.[] dict.Eq.Int))
              (x, [y], [1])
              (x, [y], [1])
  &&
  (sel.Ord.<=) (dict.Ord.(,) dict.Ord.b dict.Ord.b)
               (x, y)
               (x, y)
```

Gegeben:

```
P := {Ord a, Ord b}
class Eq a => Ord a where ...
instance Eq a => Eq [a] where ...
instance (Eq a, Eq b) => Eq (a, b) where ...
instance (Eq a, Eq b, Eq c) => Eq (a, b, c) where ...
instance Eq Int where ...
```

Abkürzungen:

(1) = (avail)
(2) = (super)
(3) = (inst)

Ableitung 1:

$$(3)\ \frac{(2)\ \dfrac{(1)\ \dfrac{}{P \Vdash \texttt{dict.Ord.a} : Ord\ a}}{P \Vdash \texttt{sel.Ord.Eq dict.Ord.a} : Eq\ a}\quad (3)\ \dfrac{(2)\ \dfrac{(1)\ \dfrac{}{P \Vdash \texttt{dict.Ord.b} : Ord\ b}}{P \Vdash \texttt{sel.Ord.Eq dict.Ord b} : Eq\ b}}{P \Vdash \texttt{dict.Eq.[] (sel.Ord.Eq dict.Ord.b)} : Eq\ [b]}\quad (3)\ \dfrac{(3)\ \dfrac{\emptyset}{P \Vdash \texttt{dict.Eq.Int} : Eq\ Int}}{P \Vdash \texttt{dict.Eq.[] dict.Eq.Int} : Eq\ Int}}{P \Vdash \texttt{dict.Eq.(,,) (sel.Ord.Eq dict.Ord.a) (dict.Eq.[] (sel.Ord.Eq dict.Ord.b)) (dict.Eq.[] dict.Eq.Int)} : Eq\ (a, [b], [Int])}$$

Ableitung 2:

$$(3)\ \frac{(1)\ \dfrac{}{P \Vdash \texttt{dict.Ord.a} : Ord\ a}\qquad (1)\ \dfrac{}{P \Vdash \texttt{dict.Ord.b} : Ord\ b}}{P \Vdash \texttt{dict.(,) dict.Ord.a dct.Ord.b} : Ord\ (a, b)}$$

Abbildung 2.17: Ableitung des Wörterbuchcodes für zwei Beispiele

3. Erweiterung des Typsystems

Um auch die Kontexte bei der Typüberprüfung zu ermitteln, muss das *Damas-Milner*-Typsystem, das von Haskell und Curry verwendet wird, *erweitert* werden. Diese Erweiterung wird in den folgenden zwei Abschnitten behandelt; im ersten Abschnitt wird das Damas-Milner-Typsystem erläutert und es werden wichtige Ergebnisse für dieses System zusammengefasst; im zweiten Abschnitt wird die eigentliche Erweiterung, nämlich die Erweiterung des Damas-Milner-Typsystems um Prädikate beschrieben (Typklassen sind ein Spezialfall dieser Erweiterung). Für weitergehende Informationen wie die Beweise der aufgestellten Sätze sei auf die entsprechenden Veröffentlichungen verwiesen ([DM82, Jon92a, Jon92b, NP95]).

3.1. Damas-Milner-Typsystem

Das Damas-Milner-Typsystem [DM82] ermöglicht parametrischen Polymorphismus, wie in der Einleitung bereits dargestellt wurde. Das Damas-Milner-Typsystem baut auf der Sprache des λ-Kalküls, erweitert durch `let`-Ausdrücke, auf. Alle Curry-Konstrukte können im Grunde genommen auf diese Sprache zurückgeführt werden. Die Sprache lautet also:

Definition 3.1 (Sprache der Ausdrücke im Damas-Milner-Typsystem)**:**
Ein *Ausdruck* ist durch folgende BNF gegeben:

$$\begin{array}{rcll} e & ::= & x & \text{(Variable)} \\ & | & e\ e' & \text{(Applikation)} \\ & | & \lambda x.e & \text{(Abstraktion)} \\ & | & \texttt{let}\ x = e\ \texttt{in}\ e' & \text{(Let)} \end{array}$$

Die Sprache des verwendeten Typsystems unterscheidet zwischen *Typen* und *Typschemata*:

Definition 3.2 (Typen und Typschemata)**:**
Die Syntax von *Typen* ist wie folgt:

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{(Typvariable)} \\ & | & \chi & \text{(Primitive Typen)} \\ & | & \tau \to \tau' & \text{(Funktionsapplikation)} \end{array}$$

Die Syntax der *Typschemata* ist wie folgt:

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

Die α_i werden *generische Variablen* genannt.

Die Typschemata ermöglichen im Damas-Milner-Typsystem den parametrischen Polymorphismus. Die Typvariablen nach dem $\forall$ in einem Typschema geben an, dass für die Typvariablen beliebige Typen eingesetzt werden können.

Weiterhin ist der Begriff der *generischen Instanz* wichtig:

Definition 3.3 (Generische Instanz)**:**
Es sei das Typschema $\sigma = \forall\alpha_1 \dots \alpha_n.\tau$ gegeben. σ' ist genau dann eine *generische Instanz* von σ, wenn $\sigma' = \forall\alpha_{k_1} \dots \alpha_{k_m}.S(\tau)$, wobei $\{k_1, \dots k_m\} \subset [1, n]$ gilt und S eine Substitution ist, die für alle $i \in [1, n] \setminus \{k_1, \dots k_m\}$ α_i auf ein τ_i abbildet.

Dass σ' eine generische Instanz von σ ist, wird auch als $\sigma' \leq \sigma$ geschrieben.

3.1.1. Typinferenz

Die Typinferenz ermittelt für eine gegebene *Typannahme (Assumption)* A, ob ein bestimmter Ausdruck e das Typschema σ besitzt. Wenn dies der Fall ist, wird dies als $A \vdash e : \sigma$ geschrieben. Eine Typannahme ist eine Menge, die Zuordnungen der Form $x \mapsto \sigma$ enthält.

Es gilt genau dann $A \vdash e : \sigma$, wenn dies mit Hilfe des Inferenzsystems aus Abb. 3.1 abgeleitet werden kann. A_x stehe dabei für die Menge A ohne Zuordnungen für x. Außerdem sei typevars(σ) die Menge aller Typvariablen, die in σ frei vorkommen, also nicht von einem $\forall$ gebunden werden. typevars(A) ist definiert durch typevars(A) = concat($\{$typevars(σ)$|x \mapsto \sigma \in A\}$) mit concat($\{U_1, \dots, U_n\}$) $= U_1 \cup \dots \cup U_n$.

Die Regeln (inst) und (gen) behandeln das Ab- und Aufbauen eines Typschemas, die Regel (appl) behandelt die *Applikation* eines λ-Ausdrucks auf einen anderen, die Regel (abs) behandelt die *Abstraktion* eines λ-Ausdrucks, und die Regel (let) behandelt `let`-Ausdrücke. Bei letzterer ist zu beachten, dass der Ausdruck e polymorph behandelt wird, also sein Typ zu einem Typschema generalisiert wird.

Curry verwendet bei einem `let`-Ausdruck eine leicht andere Semantik, und zwar wird der Typ von e nicht zu einem Typschema umgewandelt. Der Grund dafür ist, dass logische Variablen immer *genau einen* Typ haben müssen. Folgendes Beispiel wäre zum Beispiel erlaubt, wenn Variablen in `let`-Ausdrücken polymorph verwendet werden könnten:

```
bug =
  let x :: a
      x = fresh
  in x =:= 1 & x =:= 'a'
fresh :: a
fresh = y where y free
```

`x` verweist aber auf eine logische Variable, und darf daher nicht polymorph sein. Diese Einschränkung wird *Monomorphie-Restriktion* genannt.

Die Inferenzregel (let) lautet also für Curry:

$$\text{(taut)} \quad \frac{}{A \vdash x : \sigma} \qquad \text{wenn } x \mapsto \sigma \in A$$

$$\text{(inst)} \quad \frac{A \vdash e : \sigma}{A \vdash e : \sigma'} \qquad \text{wenn } \sigma' \leq \sigma$$

$$\text{(gen)} \quad \frac{A \vdash e : \sigma}{A \vdash e : \forall\alpha.\sigma} \qquad \text{wenn } \alpha \notin \text{typevars}(A)$$

$$\text{(appl)} \quad \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\ e') : \tau}$$

$$\text{(abs)} \quad \frac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash (\lambda x.e) : \tau' \to \tau}$$

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

Abbildung 3.1: Typinferenzregeln für das Damas-Milner-Typsystem

$$\text{(let')} \quad \frac{A \vdash e : \tau \qquad A_x \cup \{x : \tau\} \vdash e' : \tau'}{A \vdash (\texttt{let}\ x = e\ \texttt{in}\ e') : \tau'}$$

Die Typen von *nicht nullstelligen* in `let`-Ausdrücken definierten Funktionen, die in diesem Inferenzsystem nicht berücksichtigt sind, werden aber in Typschemata umgewandelt, womit die Funktionen polymorph verwendet werden können.

3.1.2. Algorithmus $\mathcal{W}$

Mit den oben angegebenen Inferenzregeln ist es zwar möglich, eine gegebene Typannahme zu *überprüfen*, sie stellen aber kein konstruktives Verfahren dar, um den Typ eines Ausdrucks bezüglich einer gegebenen Typannahme zu *ermitteln*.

Um die Typermittlung zu ermöglichen, muss diese anhand der *Struktur der Ausdrücke* durchgeführt werden. Eine solche Typermittlung wird durch den sogenannten *Algorithmus* $\mathcal{W}$ durchgeführt.

Der Algorithmus $\mathcal{W}$ ermittelt zu einer gegebenen Typannahme A und einem Ausdruck e eine Substitution S und einen Typ τ, so dass $SA \vdash e : \tau$ gilt. Um den Algorithmus $\mathcal{W}$ von der obigen Typinferenz zu unterscheiden, wird in den Regeln des Algorithmus $\mathcal{W}$ die Schreibweise $SA \vdash^{W} e : \tau$ verwendet.

Um den Algorithmus angeben zu können, muss zuerst der Begriff der *Unifikation* eingeführt werden:

(taut)	$\dfrac{x : \forall\alpha_1 \ldots \alpha_n.\tau \in A}{A \overset{W}{\vdash} x : \tau[\alpha_i \vert \beta_i]}$	β_i frische Typvariablen
(appl)	$\dfrac{S_1 A \overset{W}{\vdash} e_1 : \tau_1 \qquad S_2 S_1 A \overset{W}{\vdash} e_2 : \tau_2 \qquad S_2\tau_1 \overset{U}{\sim} \tau_2 \to \beta}{U S_2 S_1 A \overset{W}{\vdash} (e_1\ e_2) : U\beta}$	β frische Typvariable
(abstr)	$\dfrac{S_1(A_x \cup \{x : \beta\}) \overset{W}{\vdash} e_1 : \tau_1}{S_1 A \overset{W}{\vdash} \lambda x.e_1 : S_1\beta \to \tau_1}$	β frische Typvariable
(let)	$\dfrac{S_1 A \overset{W}{\vdash} e_1 : \tau_1 \qquad S_2(S_1 A_x \cup \{x : \mathrm{Gen}(S_1 A, \tau_1)\}) \overset{W}{\vdash} e_2 : \tau_2}{S_2 S_1 A \overset{W}{\vdash} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$	

Abbildung 3.2: Algorithmus $\mathcal{W}$ für das Damas-Milner-Typsystem

Definition 3.4:
Ein *Unifikator* für zwei Typen τ und τ' ist eine Substitution S, mit der $S\tau = S\tau'$ gilt.

Ein *allgemeinster Unifikator* (*mgu*) für zwei Typen τ und τ' ist ein Unifikator S, für den außerdem gilt: Wenn R ein Unifikator für τ und τ' ist, dann ist R als $S'S$ darstellbar, wobei S' eine weitere Substitution ist.

Sind zwei Typen τ und τ' mit dem allgemeinsten Unifikator U unifizierbar, so wird dies auch als $\tau \overset{U}{\sim} \tau'$ geschrieben.

Satz 3.1 ([Rob65])**:**
Es gibt einen Algorithmus, der den allgemeinsten Unifikator für zwei Typen τ und τ' berechnet.

Außerdem benötigen wir den Begriff der *Generalisierung*:

Definition 3.5 (Generalisierung)**:**
Ein Typ τ kann bezüglich einer gegebenen Typannahme A durch die Funktion Gen wie folgt in ein Typschema umgewandelt werden:

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

wobei die α_i die Typvariablen sind, die frei in τ, aber nicht frei in A vorkommen.

Diese Umwandlung wird als *Generalisierung* bezeichnet.

Damit lässt sich der Algorithmus $\mathcal{W}$ durch das Inferenzsystem in Abb. 3.2 darstellen.

Der Algorithmus $\mathcal{W}$ schlägt fehl, wenn für ein e und ein A keine Ableitung $SA \overset{W}{\vdash} e : \tau$ für ein S und ein τ existiert. Existiert eine solche Ableitung, so wird das Typschema $\mathrm{Gen}(SA, \tau)$ als das *vom Algorithmus $\mathcal{W}$ berechnete Typschema* bezeichnet.

Der Algorithmus $\mathcal{W}$ berechnet bei Erfolg immer ein korrektes Typschema, was durch folgenden Satz beschrieben wird:

Satz 3.2 (Korrektheit)**:**
Wenn $SA \overset{W}{\vdash} e : \tau$, dann gilt auch $SA \vdash e : \tau$.

Der Algorithmus $\mathcal{W}$ berechnet tatsächlich das *allgemeinste* Typschema für einen Ausdruck, wenn dieses existiert. Dieses Typschema wird *principal typescheme* genannt, und ist wie folgt definiert:

Definition 3.6 (Principal Typescheme)**:**
Zu einer Typannahme A und einem Ausdruck e ist σ das *principal typescheme*, wenn folgendes gilt:

- $A \vdash e : \sigma$
- Für alle σ' mit $A \vdash e : \sigma'$ gilt $\sigma' \leq \sigma$, σ' ist also immer eine generische Instanz von σ.

Damit lässt sich folgender Satz aufstellen:

Satz 3.3 (Berechnung des *principal typescheme*)**:**
Seien e und A gegeben, und es gelte $SA \overset{W}{\vdash} e : \tau$. Dann ist $\mathrm{Gen}(SA, \tau)$ das *principal typescheme* für e unter SA.

Der Algorithmus $\mathcal{W}$ berechnet also in diesem Sinne immer das *principal typescheme*.

Der Algorithmus $\mathcal{W}$ ist außerdem *vollständig*, berechnet also, wenn ein Typ bezüglich „$\vdash$“ existiert, auch einen korrekten Typ bezüglich „$\overset{W}{\vdash}$“:

Satz 3.4 (Vollständigkeit von $\mathcal{W}$)**:**
Seien A und e gegeben, und es gelte $QA \vdash e : \sigma$. Dann ist $SA \overset{W}{\vdash} e : \tau$ ableitbar für ein S und ein τ, und es gibt eine Substitution R mit $QA = RSA$ und $R\mathrm{Gen}(SA, \tau) \geq \sigma$.

Der Algorithmus $\mathcal{W}$ ist also korrekt und vollständig, und kann somit zur Typermittlung verwendet werden.

3.2. Erweiterung um Prädikate

Das im vorherigen Abschnitt beschriebene Typsystem kann auf einfache Weise um *Prädikate auf Typen* erweitert werden. Die Theorie der *predicated* oder *qualified types* wird in [Jon92a, Jon92b] detailliert beschrieben. Typklassen sind ein Spezialfall dieser Theorie; alle Ergebnisse, die für *qualified types* ermittelt wurden, gelten auch für Typklassen.

Zentral in dieser Theorie ist der Begriff des *Prädikats*. Ein Prädikat $\pi = \mathrm{p}\ \tau_1 \ldots \tau_n$ ist eine n-stellige Relation auf Typen, wobei auf den Prädikaten eine Relation $\Vdash$ definiert sein muss, die die folgenden Gesetze erfüllt:

- Monotonie: Es muss gelten $P \Vdash Q$ wenn $Q \subseteq P$.
- Transitivität: Wenn $P \Vdash Q$ und $Q \Vdash R$, dann auch $P \Vdash R$.

- Abgeschlossenheit bezüglich Substitutionen: Ist S eine Substitution, und gilt $P \Vdash Q$, so muss auch $SP \Vdash SQ$ gelten.

Bei Typklassen ist die Struktur der Prädikate „`C` τ“, wobei `C` eine Typklasse ist und τ ein Typ; die Prädikate sind also die schon bekannten *Constraints*. Als $\Vdash$-Relation wählen wird die Entailmentrelation aus Abschnitt 2.2.3.1. Diese erfüllt die obigen Eigenschaften. Somit sind Typklassen ein Spezialfall der *predicated types*.

3.2.1. Erweiterung der Typinferenzregeln

Wir betrachten nun eine um Prädikate bzw. um *predicated types* erweiterte Typsprache:

$$\begin{array}{lcll} \tau & ::= & \alpha \mid \chi \mid \tau \to \tau' & \text{(Typen)} \\ \rho & ::= & P \Rightarrow \tau & \text{(qualifizierte Typen)} \\ \sigma & ::= & \forall T.\rho & \text{(Typschemata)} \end{array}$$

wobei P eine Menge von Prädikaten sein soll, und T eine Menge von Typvariablen.

Weiterhin seien folgende Abkürzungen gegeben, die die Inferenzregeln lesbarer machen:

$$\begin{array}{ll} \tau & \hat{=}\ \emptyset \Rightarrow \tau \\ \pi \Rightarrow \rho & \hat{=}\ P \cup \{\pi\} \Rightarrow \tau \\ \rho & \hat{=}\ \forall\emptyset.\rho \\ \forall\alpha.\sigma & \hat{=}\ \forall(T \cup \{\alpha\}).\rho \end{array}$$

Damit lassen sich wie beim Damas-Milner-Typsystem Inferenzregeln angeben, mit denen abgeleitet werden kann, ob eine gegebene Typannahme korrekt ist. Die Sprache der Inferenzregeln wird um das Element der Prädikate erweitert und lautet nun:

$$P \mid A \vdash e : \sigma$$

Dies steht für: „Unter der Typannahme A ist der qualifizierte Typ von e σ, wenn die Prädikate in P erfüllt sind“.

Die Inferenzregeln sind in Abb. 3.3 zu finden; typevars(P) sei dabei die Menge aller in P frei vorkommenden Typvariablen.

Die Regeln (taut), (appl) und (abs) ähneln stark den entsprechenden Regeln des ursprünglichen Typinferenzsystems, es wird lediglich die Menge der Prädikate hinzugefügt. Auch die Regeln (inst), (gen) und (let) sind den entsprechenden Regeln des ursprünglichen Inferenzsystems ähnlich. Neu sind die zwei Regeln (pelim) und (pgen), in denen qualifizierte Typen auf- und abgebaut werden. Dabei kann entweder ein Prädikat aus der Menge P zum Typ hinzugefügt werden (pgen), oder es kann ein Prädikat eines qualifizierten Typs aus diesem gelöscht werden, wenn es von P impliziert wird (pelim).

(taut) $$\frac{}{P \mid A \vdash x : \sigma}$$ wenn $x \mapsto \sigma \in A$

(appl) $$\frac{P \mid A \vdash e : \tau' \to \tau \qquad P \mid A \vdash e' : \tau'}{P \mid A \vdash (e\ e') : \tau}$$

(abs) $$\frac{P \mid A_x \cup \{x : \tau'\} \vdash e : \tau}{P \mid A \vdash (\lambda x.e) : \tau' \to \tau}$$

(inst) $$\frac{P \mid A \vdash e : \forall\alpha.\sigma}{P \mid A \vdash e : \sigma[\alpha|\tau]}$$

(gen) $$\frac{P \mid A \vdash e : \sigma}{P \mid A \vdash e : \forall\alpha.\sigma}$$ wenn $\alpha \notin \text{typevars}(A) \cup \text{typevars}(P)$

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

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

(let) $$\frac{P \mid A \vdash e : \sigma \qquad Q \mid A_x \cup \{x : \sigma\} \vdash e' : \tau}{P \cup Q \mid A \vdash (\texttt{let}\ x = e\ \texttt{in}\ e') : \tau}$$

(let') (Curry) $$\frac{P \mid A \vdash e : \tau \qquad Q \mid A_x \cup \{x : \tau\} \vdash e' : \tau'}{P \cup Q \mid A \vdash (\texttt{let}\ x = e\ \texttt{in}\ e') : \tau'}$$

Abbildung 3.3: Typinferenzregeln für das erweiterte Typsystem

3.2.2. Constrained Types

Im Kontext von qualifizierten Typen ist es sinnvoll, den Begriff der *Constrained Types* einzuführen. Constrained Types sind Typen erweitert um eine Menge von Prädikaten, und haben die Form $(P \mid \sigma)$, wobei P die Menge von Prädikaten und σ das Typschema sind.

Damit lässt sich der Begriff der *generischen Instanz* für Constrained Types einführen (als Verallgemeinerung des Begriffs der generischen Instanz im Damas-Milner-Typsystem):

Definition 3.7 (Generische Instanzen)**:**
Ein qualifizierter Typ $R \Rightarrow \mu$ ist eine generische Instanz des Constrained Types $(P \mid \forall\alpha_i.Q \Rightarrow \tau)$, wenn es Typen τ_i gibt mit $R \Vdash P \cup Q[\alpha_i|\tau_i]$ und $\mu = \tau[\alpha_i|\tau_i]$.

(taut) $$\frac{x : \forall\alpha_1 \ldots \alpha_n.P \Rightarrow \tau \in A}{P[\alpha_i|\beta_i] \mid A \overset{W}{\vdash} x : \tau[\alpha_i|\beta_i]}$$ β_i frische Typvariablen

(appl) $$\frac{P \mid S_1 A \overset{W}{\vdash} e_1 : \tau_1 \qquad Q \mid S_2 S_1 A \overset{W}{\vdash} e_2 : \tau_2 \qquad S_2\tau_1 \overset{U}{\sim} \tau_2 \to \beta}{U(S_2 P \cup Q) \mid U S_2 S_1 A \overset{W}{\vdash} (e_1\ e_2) : U\beta}$$ β frische Typvariable

(abstr) $$\frac{P \mid S_1(A_x \cup \{x : \beta\}) \overset{W}{\vdash} e_1 : \tau_1}{P \mid S_1 A \overset{W}{\vdash} \lambda x.e_1 : S_1\beta \to \tau_1}$$ β frische Typvariable

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

Abbildung 3.4: Algorithmus $\mathcal{W}$ für das erweiterte Typsystem

Ähnlich wie beim Damas-Milner-Typsystem lässt sich auch eine Ordnung auf den Constrained Types festlegen:

Definition 3.8 (Ordnung auf Constrained Types)**:**
Es gelte $(P \mid \sigma) \leq (Q \mid \eta)$, in Worten „$(Q \mid \eta)$ ist allgemeiner als $(P \mid \sigma)$“, wenn jede generische Instanz von $(P \mid \sigma)$ auch eine generische Instanz von $(Q \mid \eta)$ ist.

Da ein Typschema σ äquivalent zu dem Constrained Type $(\emptyset \mid \sigma)$ ist, können mit der oben eingeführten $\leq$-Relation auch Typschemata verglichen werden. Außerdem gilt insbesondere $(P \mid \rho) \leq P \Rightarrow \rho$ und $P \Rightarrow \rho \leq (P \mid \rho)$, $(P \mid \rho)$ und $P \Rightarrow \rho$ sind also in der Ordnung $\leq$ äquivalent, und deshalb beliebig austauschbar.

Der Begriff der *Generalisierung* eines Typs bezüglich einer Typannahme lässt sich eins zu eins vom Damas-Milner-Typsystem übertragen:

Definition 3.9 (Generalisierung für qualifizierte Typen)**:**
Die Generalisierungsfunktion Gen sei für eine Typannahme A und einen qualifizierten Typ ρ wie folgt definiert: $\mathrm{Gen}(A, \rho) = \forall\alpha_i.\rho$, wobei die α_i frei in ρ, aber nicht frei in A vorkommen.

3.2.3. Algorithmus $\mathcal{W}$ auf *predicated types*

Der im Abschnitt 3.1.2 angegebene Algorithmus $\mathcal{W}$ kann nun um Prädikate erweitert werden. Die neuen Inferenzregeln des Algorithmus $\mathcal{W}$ sind in Abb. 3.4 angegeben.

Als das für einen Ausdruck e und eine Typannahme A vom Algorithmus $\mathcal{W}$ berechnete Typschema wird nun das Ergebnis von $\mathrm{Gen}(SA, P \Rightarrow \tau)$ bezeichnet, wenn $P \mid SA \overset{W}{\vdash} e : \tau$ mit dem Algorithmus $\mathcal{W}$ abgeleitet werden kann.

Der angegebene Algorithmus ist wie beim Damas-Milner-Typsystem *korrekt* und *vollständig*:

Satz 3.5 (Korrektheit)**:**
Wenn $P \mid SA \overset{W}{\vdash} e : \tau$, dann auch $P \mid SA \vdash e : \tau$.

Satz 3.6 (Vollständigkeit)**:**
Es seien A und e gegeben, und es gelte $P \mid QA \vdash e : \sigma$. Dann ist $P' \mid SA \overset{W}{\vdash} e : \tau$ ableitbar für ein P', ein S und ein τ, und es gibt eine Substitution R mit $QA = RSA$ und $R\text{Gen}(SA, P' \Rightarrow \tau) \geq (P \mid \sigma)$.

Außerdem berechnet der Algorithmus $\mathcal{W}$ immer das *principal typescheme*, das analog zum Damas-Milner-Typsystem definiert ist:

Definition 3.10 (Principal Typescheme)**:**
Zu einer Typannahme A und einem Ausdruck e ist $(P \mid \sigma)$ das *principal typescheme*, wenn folgendes gilt:

- $P \mid A \vdash e : \sigma$
- Für alle σ' mit $P' \mid A \vdash e : \sigma'$ gilt $(P' \mid \sigma') \leq (P \mid \sigma)$.

Satz 3.7:
Der Algorithmus $\mathcal{W}$ berechnet immer das *principal typescheme*: Seien A und e gegeben, und es gelte $P \mid SA \overset{W}{\vdash} e : \tau$ für ein P, S und τ. Dann ist $\text{Gen}(SA, P \Rightarrow \tau)$ das *principal typescheme* für A und e.

Die Erweiterung des Algorithmus $\mathcal{W}$ ist also wie der originale Algorithmus $\mathcal{W}$ zur Typermittlung nutzbar. Wenn als Prädikate die Constraints und als Prädikatrelation die Entailment-Relation verwendet werden, kann der Algorithmus $\mathcal{W}$ also dazu benutzt werden, für einen gegebenen Ausdruck zu bestimmen welchen Typ er hat, und welche Constraints für diesen Ausdruck gelten. Die im Compiler enthaltene Typ-Check-Komponente verwendet genau den oben beschriebenen erweiterten Algorithmus $\mathcal{W}$.

4. Implementierung

4.1. Übersicht über den KiCS2-Compiler

Der KiCS2-Compiler ist in zwei Komponenten aufgeteilt: das *Frontend* und das *Backend*. Das *Frontend* führt eine Prüfung des Curry-Codes durch, und kompiliert Curry in eine vereinfachte Zwischensprache namens *FlatCurry*. Die FlatCurry-Repräsentation wird anschließend vom *Backend* nach *Haskell* übersetzt. Diese Aufteilung hat den Vorteil, dass an das Frontend auch andere Backends angeschlossen werden können. Der PAKCS-Compiler teilt zum Beispiel mit dem KiCS2-Compiler das Frontend, kompiliert aber FlatCurry nach *Prolog*. Auch in andere Sprachen kann die FlatCurry-Repräsentation theoretisch übersetzt werden (siehe Abb. 4.1).

4.1.1. FlatCurry

Die FlatCurry-Sprache ist eine stark reduzierte Variante der Sprache Curry. Durch die starke Vereinfachung soll es auf einfache Weise ermöglicht werden, verschiedene Backends an das Frontend anzuschließen. Wie schon weiter oben erwähnt, ist es möglich, FlatCurry in so unterschiedliche Sprachen wie Prolog (PAKCS) und Haskell (KiCS2) zu kompilieren. Das kann aber nur erreicht werden, indem viele – vor allem funktionale – Merkmale aus der Sprache entfernt werden.

Ein FlatCurry-Modul besteht im Wesentlichen aus Funktions- und Datentypdeklarationen. Die Datentypdeklarationen haben dieselbe Form wie in Curry; die Funktionsdeklarationen sind aber in folgender Weise vereinfacht:

- Es wird kein Pattern Matching durchgeführt. Die Argumente der Funktionen sind alle einfache Variablen. Das Pattern Matching wird vollständig durch andere Sprachkonstrukte wie Case-Ausdrücke ersetzt.

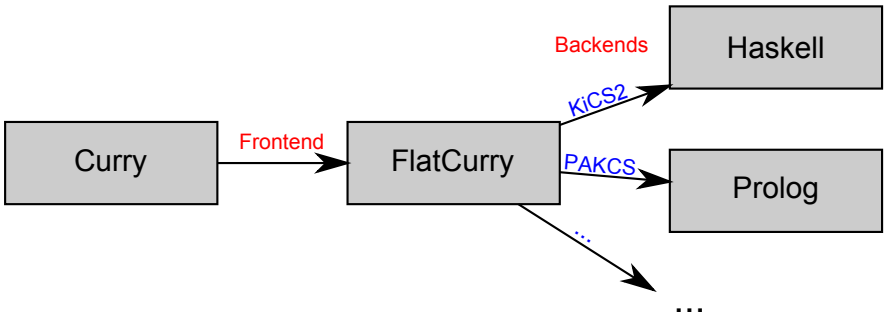

Abbildung 4.1: Aufteilung in Frontend und Backend

- Jede Funktion besitzt nur noch *genau eine* Regel. Dies wird ermöglicht, indem mehrere Regeln durch Case-Ausdrücke zusammengefasst werden.

Auch die Sprache der Ausdrücke ist reduziert. Jeder „syntaktische Zucker“ wie arithmetische Sequenzen, *Sections* und *List Comprehensions* ist entfernt worden.
Weitere Vereinfachungen sind:

- Es gibt keine `where`-Abschnitte mehr. Alle Deklarationen in `where`-Abschnitten von Funktionen werden im Kompilierprozess in `let`-Ausdrücke verschoben.
- Patterns bestehen nur noch entweder aus Datenkonstruktoren angewandt auf Variablen, oder aus Literalen. Dadurch wird die Komplexität der Case-Ausdrücke stark reduziert.
- In `let`-Ausdrücken werden nur noch einfache Variablen definiert, und keine Funktionen mehr.

4.1.2. Anpassung des KiCS2-Compilers für die Implementierung von Typklassen

Für die Implementierung von Typklassen ist bis auf eine Ausnahme lediglich eine Änderung des *Frontends* notwendig. Das Frontend kann Programme mit Typklassenelementen genauso wie Programme ohne Typklassenelemente nach FlatCurry übersetzen, da durch die beschriebenen Programmtransformationen alle Typklassenelemente aus dem Code entfernt werden können. Das KiCS2-Backend kann dann aus dem FlatCurry-Code unverändert Haskell-Code erzeugen.

Es muss nur eine einzige Änderung durchgeführt werden, die speziell mit dem KiCS2-Compiler zu tun hat. Diese wird im Folgenden geschildert:

In der Prelude werden einige Funktionen als *extern* markiert, und deren Implementierungen in Haskell in einer speziellen mit der Prelude assoziierten Datei angegeben. Durch die Einführung von Typklassen können aber viele bisher extern implementierte Funktionen nun durch Typklassenmethoden implementiert werden. So ist der Gleichheitsoperator in der originalen Prelude extern implementiert, während er nun durch Typklassen implementiert wird. Nur für den Vergleich von Basistypen wie Charactern, Integern und Floats muss noch die externe Implementierung verwendet werden. Damit keine Namenüberscheidungen auftreten, müssen die Namen der externen Funktionen geändert werden: Da die Typklasse `Eq` den Gleichheitsoperator `==` definiert, muss der externe Gleichheitsoperator zum Beispiel nach `==$` umbenannt werden, und in der mit der Prelude assoziierten Datei mit den Implementierungen der externen Funktionen muss diese Umbenennung berücksichtigt werden. Auch andere vormals extern implementierte Funktionen werden durch Typklassenmethoden ersetzt, und nur noch für Basistypen werden externe Funktionen benötigt, zum Beispiel für die Durchführung von arithmetischen Operationen auf Integern und Floats. Diese Funktionen müssen dann wieder entsprechend umbenannt werden, um Namenskonflikte mit Klassenmethoden zu vermeiden.

Die KiCS2-spezifische Änderung ist also die Umbenennung der externen Funktionen in der Prelude und ihrer assoziierten Datei; außerdem müssen für arithmetische Operationen auf Floats neue externe Funktionen eingeführt werden, da solche Operationen vormals nicht in der Prelude enthalten waren.

In Anhang A.4 ist die gesamte Prelude zu finden; dort kann man nochmals die Verzahnung von externen Funktionen und Typklassen erkennen.

4.2. Übersicht über das ursprüngliche Frontend

In diesem Abschnitt werden zuerst der Aufbau und die Struktur des *ursprünglichen* Frontends erläutert. Damit wird die Grundlage für das Verständnis des *angepassten* Frontends gelegt. Das angepasste Frontend wird in Abschnitt 4.3 erläutert.

4.2.1. Compiler-Umgebung

Während des Kompilierprozesses werden immer zwei Objekte mitgeführt: Der zu kompilierende Code in der jeweils aktuellen Form, und die *Compiler-Umgebung*. In der Compiler-Umgebung werden die Ergebnisse der einzelnen Kompilierstadien gespeichert, so dass diese in späteren Phasen verwendet werden können.

Die Compiler-Umgebung des ursprünglichen Frontends enthält die folgenden Elemente:

Bezeichner des gerade kompilierten Modules Während des Compileprozesses wird immer wieder die Information benötigt, wie das gerade kompilierte Modul heißt.

Interface-Umgebung In der Interface-Umgebung wird für jedes importierte Modul ein *Interface* gespeichert, das die exportierten Elemente des importierten Moduls enthält. Diese Umgebung ist wichtig für den Import und den Export von Modulen.

Typkonstruktor-Umgebung In dieser Umgebung werden alle *Typkonstruktoren* gespeichert, die im Programm vorkommen oder importiert wurden. Zu jedem Typkonstruktor werden zudem, wenn der Typkonstruktor zu einem algebraischen Datentyp gehört, auch die entsprechenden Datenkonstruktoren gespeichert, die von diesem algebraischen Datentyp deklariert wurden. Ist der Typkonstruktor Teil einer Typsynonymdeklaration, so wird stattdessen der Typ gespeichert, für den in dieser Deklaration ein neuer Name eingeführt wurde.

Werteumgebung In der *Werteumgebung* werden alle im Quelltext vorkommenden Variablen, Funktionen, und Datenkonstruktoren, zusammen mit ihrem Typ, ihrer Stelligkeit und ihrem Originalnamen gespeichert. Der Originalname gibt immer an, aus welchem Modul ein Wert *ursprünglich* stammt; diese Information ist wichtig, da durchaus der Name, unter dem ein Wert im Quelltext verwendet wird, sich von seinem Originalnamen unterscheiden kann, zum Beispiel, wenn ein Wert über eine Importdeklaration der Form `import Module as M` importiert wird. Die Typen der Werte werden immer in *expandierter* Form gespeichert, das heißt, dass erstens alle

Typsynonyme aufgelöst wurden, und zweitens, die Typkonstruktoren unter ihren Originalnamen aufgeführt sind. Die Stelligkeit schließlich gibt bei Funktionen an, wie viele Parameter die Funktion entgegennimmt.

Präzedenzen-Umgebung In der Präzedenzen-Umgebung sind die Präzedenzen und Assoziativitäten von Operatoren, Funktionen und Datenkonstruktoren gespeichert. Die Präzedenzen für Funktionen und Datenkonstruktoren gelten dabei für den Fall, dass diese in Infixschreibweise verwendet werden. Die Präzedenzen selber geben an, wie stark ein Infix-Operator (jeglicher Form) bindet. Die Assoziativität gibt an, ob in einem Ausdruck der Form $e_1 \circ e_2 \circ \cdots \circ e_n$ Links- oder Rechtsklammerung angenommen wird, oder ob (bei Nicht-Assoziativität des Operators) dieser Ausdruck nicht erlaubt ist.

Es wird nur für solche Operatoren ein Eintrag in der Präzedenzen-Umgebung erstellt, für die eine explizite *„Fixity-Deklaration“* existiert. Ansonsten wird eine Standard-Präzedenz und -Assoziativität angenommen.

Eine Umgebung ist eine Zuordnung von Namen zu *Entitäten* (dies können Funktionen, Konstruktoren etc. sein) [DJH02]. Unter einem Namen, der qualifiziert oder unqualifiziert sein kann, können eine oder mehrere Entitäten gespeichert sein. Zu jeder Entität wird gespeichert, ob diese aus dem lokalen Modul stammt oder importiert wurde. Wird ein Name im Code benutzt, und es existieren mehrere Entitäten zu diesem Namen, so wird geprüft, ob eine der Entitäten aus dem lokalen Modul stammt. Wenn ja, so wird diese Entität verwendet. Wenn nein, wird ein Fehler ausgegeben, dass der Name mehrdeutig ist. Hiermit implementiert Curry eine andere Semantik als Haskell; bei Haskell wird der Name schon als mehrdeutig aufgefasst, wenn mehrere Entitäten unter ihm gespeichert sind.

Jede Entität muss einen *Originalnamen* besitzen. Zwei Entitäten gelten als gleich, wenn sie denselben Originalnamen besitzen. Das Konzept der Originalnamen ist für den Import wichtig: Wird dieselbe Entität über unterschiedliche Importwege und unter demselben Namen eingebunden, so darf unter diesem Namen in der Umgebung trotzdem nur *ein* Eintrag für diese Entität existieren, da sonst die Verwendung des Namens mehrdeutig wäre. Bei algebraischen Datentypen tritt außerdem der Fall auf, dass bei verschiedenen Importspezifikationen unterschiedliche Datenkonstruktoren importiert werden können. Deshalb ist es notwendig, für algebraische Datentypen eine *Merge*-Funktion bereitzustellen, die die Vereinigung aller in verschiedenen Importspezifikationen angegebenen Datenkonstruktor-Mengen bildet.

Beispiel 4.1 (Entitätenkonzept und Mehrdeutigkeitssemantik von Curry)**:**
Es seien die folgenden Module gegeben:

```
module M1 where
fun :: a -> a
fun = ...

fun2 :: a -> a
fun2 = ...
```

```
module M2 where
fun :: a -> Bool
fun = ...

fun2 :: a -> a -> a
fun2 = ...

funM2 :: a -> a
funM2 = ...
```

```
module M where
import M1 as N
import M2

fun :: a -> a -> a
fun = ...

fun3 :: a -> a
fun3 = ...
```

Die Werteumgebung bei der Kompilation des Moduls `M` lautet (Ausschnitt):

Name	Originalname	Typ	Quelle
`fun`	`M.fun`	`a -> a -> a`	Lokal
	`M1.fun`	`a -> a`	Importiert
	`M2.fun`	`a -> Bool`	Importiert
`fun2`	`M1.fun2`	`a -> a`	Importiert
	`M2.fun2`	`a -> a -> a`	Importiert
`fun3`	`M.fun3`	`a -> a`	Lokal
`funM2`	`M2.funM2`	`a -> a`	Importiert
`M.fun`	`M.fun`	`a -> a -> a`	Lokal
`M.fun3`	`M.fun3`	`a -> a`	Lokal
`N.fun`	`M1.fun`	`a -> a`	Importiert
`N.fun2`	`M1.fun2`	`a -> a`	Importiert
`M2.fun`	`M2.fun`	`a -> Bool`	Importiert
`M2.fun2`	`M2.fun2`	`a -> a`	Importiert
`M2.funM2`	`M2.funM2`	`a -> a`	Importiert

Hier kann man gut das Konzept der Entitäten erkennen: So sind unter dem Namen `fun` zum Beispiel drei Entitäten gespeichert, die aus jeweils unterschiedlichen Modulen stammen.

Die folgenden Beispiele sollen nochmals die Mehrdeutigkeitssemantik, wie sie in Curry implementiert ist, verdeutlichen.

Im Modul `M` sind die folgenden Ausdrücke erlaubt:

- `fun x y`, da die lokale Definition von `fun` die importierten Definitionen von `fun` überschattet.
- `fun3 x`, da unter `fun3` nur eine einzige Entität gespeichert ist.

Nicht erlaubt ist der Ausdruck `fun2 x`. Hier ist `fun2` mehrdeutig, da `fun2` in zwei verschiedenen Modulen definiert wurde, und keine lokale Definition existiert.

Der Ausdruck `N.fun2 x` ist hingegen erlaubt, weil nun durch die Qualifikation des Namens `fun2` angegeben wird, welche Funktion von den möglichen Funktionen genau gemeint ist. In der Werteumgebung steht auch nur genau eine Entität unter `N.fun2`, und daher ist dieser Name nicht mehrdeutig.

Zu beachten ist, dass die Funktion `M1.fun2` *nicht* unter ihrem Originalnamen sondern unter dem Namen `N.fun2` angesprochen wird.

Hier noch ein Beispiel, das verdeutlicht, warum eine Merge-Funktion für algebraische Datentypen bereitgestellt werden muss:

Beispiel 4.2 (Merge-Funktion)**:**
Angenommen, es sind die folgenden Module definiert:

```
module M where
  data T = T1 | T2 | T3 | T4

module M2 where
  import M (T(T1))
  import M (T(T3, T4))
```

Dann ist nach dem ersten Import in Modul `M2` bereits ein Eintrag für `T` in der Compilerumgebung vorhanden; beim zweiten Import wird aber `T` nochmals mit anderen Datenkonstruktoren importiert. Damit *alle* in den Importspezifikationen angegebene Datenkonstruktoren importiert werden, muss ein Mergen des in der Compilerumgebung schon existierenden Eintrags für `T` mit dem Eintrag, der für die zweite Importspezifikation angelegt werden würde, geschehen. Damit wird erreicht, dass die Datenkonstruktoren `T1` *und* `T3` und `T4` importiert werden.

4.2.2. Kompilierphasen

Der Kompilierprozess des Frontends ist in verschiedene Phasen aufgeteilt. Zu unterscheiden sind zwei verschiedene Arten von Phasen:

Checks Checks führen jeweils eine bestimmte Überprüfung des zu kompilierenden Codes durch. Außerdem können Checks auch eine Änderung des Codes vornehmen und die Compiler-Umgebung modifizieren.

Transformationen Transformationen führen lediglich eine Code-Transformation und/oder eine Modifikation der Compiler-Umgebung durch.

Eine Übersicht über die Abfolge der einzelnen Kompilierstadien ist in Abb. 4.2 gegeben.

Im Nachfolgenden werden die einzelnen Kompilierphasen im Detail beschrieben.

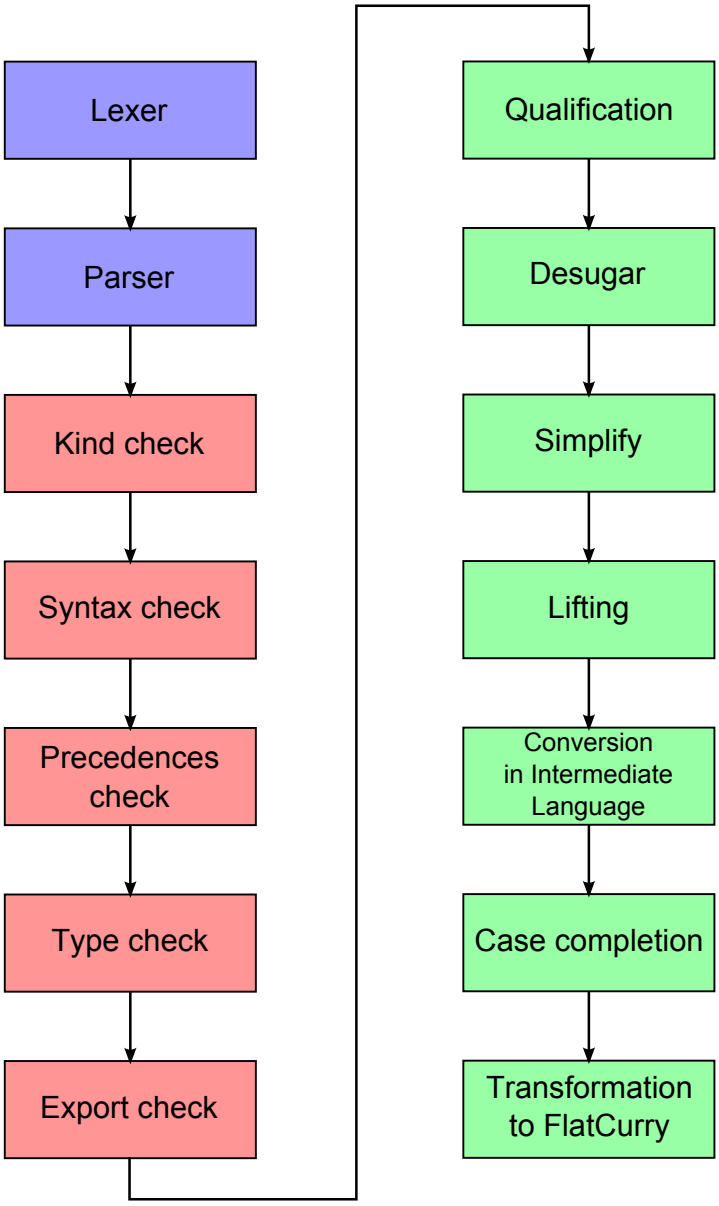

Abbildung 4.2: Ursprüngliche Abfolge der Kompilierstadien des Compilers; Checks sind rot, Transformationen grün hinterlegt.

4.2.2.1. Lexer

Der Lexer zerlegt, wie es aus dem Compilerbau bekannt ist, den Quellcode in einzelne *Token* oder *Lexeme*. Für die Implementierung des Lexers wird der *Continuation Passing Style* verwendet. Zu erwähnen ist, dass der Lexer das Layout im Haskell-Style dadurch unterstützt, indem er virtuelle Semikolons und geschweifte Klammern einfügt.

4.2.2.2. Parser

Der Parser konstruiert aus der Tokenfolge den *abstrakten Syntaxbaum (abstract syntax tree, AST)*. Auch der Parser ist im Continuation Passing Style implementiert. Die Funktionalität wird durch *Parserkombinatoren* bereitgestellt, die LL(1)-Grammatiken parsen können. Der Parser benutzt die Grammatik, die in Anhang A.1 beschrieben ist. Diese Grammatik ist nicht vollständig LL(1); daher wird ein spezieller Parserkombinator für die Alternative namens „`<|?>`“ zur Verfügung gestellt, der beide ihm übergebene Parser anwendet, und anschließend – wenn beide zu einem erfolgreichen Parse geführt haben – den Parse auswählt, der mehr Lexeme konsumiert hat. Ansonsten, wenn nur ein Parse erfolgreich war, wird dieser ausgewählt. Alle anderen Parserkombinatoren sind LL(1)-Parser.

4.2.2.3. Kind Check

Der „Kind Check“ überprüft, ob alle *Kinds* im Quellcode korrekt sind. Das Konzept der Kinds liegt noch eine Ebene über dem Konzept der Typen. $*$ ist der Kind aller nullstelligen Typkonstruktoren. Sind κ_1 und κ_2 Kinds, so ist $\kappa_1 \to \kappa_2$ der Kind eines Typs, der einen Typ des Kinds κ_1 entgegennimmt, und als Ergebnis einen Typ mit dem Kind κ_2 liefert [M$^+$10, Seite 37]. So hat der Typkonstruktor `Bool` den Kind $*$, während der Typkonstruktor `Maybe` den Kind $* \to *$ hat.

Im Kind Check wird geprüft, ob alle Typkonstruktoren *vollständig* angewandt wurden, dass also jedes Konstrukt bestehend aus einem Typkonstruktor und allen vorhandenen Parametern den Kind $*$ hat.

Beispiel 4.3:
Es sei der folgende Datentyp gegeben:

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

Dann sind die folgenden Signaturen nicht erlaubt:

- `T a b ->` τ
- `T a b c d ->` τ

Im ersten Fall hat `T a b` den Kind $* \to *$; im zweiten Fall kann überhaupt kein Kind ermittelt werden, da `T` zu viele Argumente übergeben werden. Korrekt ist nur eine Typsignatur der Form `T a b c ->` τ.

Im Curry-System können im Gegensatz zu Haskell Typkonstruktoren *nicht* syntaktisch von Typvariablen unterschieden werden. In Haskell gibt es die einfache Regel, dass Typkonstruktoren groß-, und Typvariablen kleingeschrieben werden. In Curry existiert diese Beschränkung nicht, sowohl Typkonstruktoren als auch Typvariablen können klein- oder großgeschrieben werden. Daher muss bei jedem Vorkommen eines Bezeichners, der nicht weiter appliziert wird, geprüft werden, ob dieser ein Typkonstruktor oder eine Typvariable ist. Diese Überprüfung wird ebenfalls durch den Kind Check durchgeführt.

Beispiel 4.4 (Unterscheidung nullstellige Typkonstruktoren und Typvariablen)**:**
Es sei der folgende Code gegeben:

```
data T = ...

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

Dann ist der Name `T` in der Typsignatur von `fun` der Name eines Typkonstruktors, und `S` ist eine Typvariable, da kein Typkonstruktor mit dem Namen `S` existiert.

4.2.2.4. Syntax Check

Im Syntax Check wird überprüft, ob alle Funktionen und Variablen, die im Code benutzt werden, definiert sind, und dass diese nicht *mehrdeutig* sind. Undefinierte Funktionen können zum Beispiel auftreten, wenn die Namen von Funktionen falsch geschrieben werden. Mehrdeutige Funktionen können zum Beispiel auftreten, wenn zwei Funktionen mit denselben Namen aus zwei verschiedenen Modulen importiert wurden.

Die Semantik der Mehrdeutigkeit in Curry ist unterschiedlich zu der Semantik der Mehrdeutigkeit in Haskell: Wird eine Entität egal welcher Art aus einem Modul importiert, und im aktuellen Modul eine Entität mit demselben Namen deklariert, so wird die Entität aus dem importierten Modul *verborgen*. Dies entspricht der Behandlung der Mehrdeutigkeit in `let`-Ausdrücken: Auch hier verbergen lokale Definitionen Definitionen „weiter oben“. In Haskell hingegen verbergen im aktuellen Modul definierte Entitäten *nicht* importierte Entitäten, sodass die Verwendung eines Bezeichners, der sowohl auf eine im Modul definierte als auch auf eine importierte Entität verweist, zu einem Mehrdeutigkeits-Fehler führt.

Der Syntax-Check führt weiterhin eine *Umbenennung* aller im Code vorkommenden Variablen und Funktionen, die nicht auf Top-Level-Ebene deklariert wurden, durch. Dies hat den Zweck, dass danach keine Mehrdeutigkeiten mehr auftreten, wenn zum Beispiel zwei Parameter mit demselben Namen in verschiedenen Funktionen verwendet werden, oder in zwei `let`-Deklarationen an verschiedenen Stellen Funktionen mit demselben Namen deklariert werden. Top-Level-Funktionen werden nicht umbenannt, weil deren Namen modulweit eindeutig sind. Somit kann der Compiler eine *flache* Typumgebung unterhalten, da nach der Umbenennung nie der Fall eintritt, das zwei unterschiedliche Entitäten im Modul denselben Namen besitzen. Die Umbenennung erfolgt auf einfache Weise dadurch, dass die Variablen- und Funktionsnamen mit einer fortlaufenden

Integer-Zahl indiziert werden.

Beispiel 4.5 (Umbenennung der Funktionen und Variablen)**:**
Der Code

```
fun1 x = x
  where
  fun2 x = x
  fun3 x = x
  fun4 y = x
```

wird durch Umbenennung der Variablen und Funktionen in folgenden Code transformiert:

```
fun1 x.1 = x.1
  where
  fun2.2 x.3 = x.3
  fun3.2 x.5 = x.5
  fun4.2 y.7 = x.1
```

Auch für Datenkonstruktoren und Parameter gilt in Curry, dass diese klein- *und* großgeschrieben werden können, im Gegensatz zu Haskell. Deshalb muss für jeden Bezeichner geprüft werden, ob dieser ein Datenkonstruktor oder eine Variable ist. Diese Überprüfung wird ebenfalls im Syntax Check durchgeführt.

Beispiel 4.6 (Unterscheidung nullstellige Datenkonstruktoren und Variablen)**:**
Es sei der folgende Code gegeben:

```
data T = S | T

fun S U = ...
```

In der Funktion `fun` ist dann `S` ein Pattern mit einem nullstelligen Datenkonstruktor und `U` eine Variable.

Weiterhin wird im Syntax Check überprüft, dass alle Gleichungen einer Funktionsdefinition dieselbe Stelligkeit, also die gleiche Anzahl an Parametern, besitzen. Alle Gleichungen für eine bestimmte Funktion werden danach zusammengeführt, denn die Gleichungen können – im Gegensatz zu Haskell – beliebig im Quelltext verteilt sein.

4.2.2.5. Precedences Check

Der Parser parst den Quellcode zunächst so, als ob alle Operatoren dieselbe Präzedenz haben, und rechtsassoziativ binden. Daher muss der abstrakte Syntaxbaum so modifiziert werden, dass die im Code angegebenen Präzedenzen und Assoziativitäten berücksichtigt werden. Dies geschieht durch eine *Umordnung* des ASTs. Der Precedences Check sucht zu diesem Zweck alle Fixity-Deklarationen im Quelltext und fügt die Präzedenzen in die Präzedenzen-Umgebung ein.

Beispiel 4.7:
Der Ausdruck „`5 * 3 + 2`“ wird durch den Parser wie folgt geparst: `5 * (3 + 2)`. Im Precedences Check werden dann die Präzedenzen des Mal- und des Plus-Operators berücksichtigt, und der Ausdruck in `(5 * 3) + 2` umgewandelt.

4.2.2.6. Type Check

Der Typcheck führt eine *Typüberprüfung* des Quellcodes durch. Als Typsystem wird das Damas-Milner-Typsystem verwendet, und die Typermittlung für Ausdrücke wird durch den Algorithmus $\mathcal{W}$ durchgeführt (siehe Kapitel 3).

Im Algorithmus $\mathcal{W}$ treten viele Substitutionen auf, die auf die Typannahme und die Typen angewendet werden. Um den Code leichter, wartbarer und sicherer zu gestalten (denn leicht können Substitutionen vergessen oder vertauscht werden), werden alle Substitutionen in einer *State-Monade* gekapselt. Eine State-Monade ist eine Monade, die einen Zustand besitzt, der im Code mitgeführt wird, und durch entsprechende Anweisungen verändert werden kann. Die verschiedenen Substitutionen werden in einer globalen Substitution gespeichert, die fortlaufend aktualisiert wird.

Im Typcheck wird die globale Substitution *implizit* verändert und verwendet, so zum Beispiel durch die Unifizierungsfunktion, die im Algorithmus $\mathcal{W}$ im Fall der Applikation benötigt wird. Werden zwei Typen unifiziert, so wird die globale Substitution automatisch um den *mgu* für diese Typen erweitert.

Der Prozess der Typüberprüfung läuft folgendermaßen ab:

1. Zuerst werden alle Typ- und Datenkonstruktoren in die Werte-Umgebung aufgenommen. Dies ist notwendig, da sonst die Typen der Datenkonstruktoren nicht verfügbar wären.

2. Der Code wird anschließend in sogenannte *Deklarationsgruppen* aufgeteilt. In einer Deklarationsgruppe sind – informell gesprochen – alle Funktionen voneinander abhängig (die Ermittlung von Deklarationsgruppen wird weiter unten genauer besprochen). Außerdem werden die Deklarationsgruppen topologisch sortiert, das heißt, dass wenn eine Deklarationsgruppe A von einer Deklarationsgruppe B abhängt, immer zuerst die Deklarationsgruppe B betrachtet wird.

3. Dann wird – in topologischer Reihenfolge – für alle Deklarationsgruppen die Typüberprüfung durchgeführt. Dazu werden zuerst die linken Seiten der Funktionen in der Deklarationsgruppe überprüft und deren Typen anschließend mit den Typen der rechten Seiten der Funktionen unifiziert. Schlägt dabei eine der Unifizierungen fehl, so wird eine Compilermeldung ausgegeben, die den Fehler beschreibt.

4. Sind alle Unifizierungen erfolgreich, so werden anschließend die Typschemata der Funktionen in die Werte-Umgebung geschrieben. Um das Typschema jeweils einer Funktion zu ermitteln, wird die Gen-Funktion verwendet, wie in Abschnitt 3.1.2 erläutert wurde: Das Typschema der Funktion lautet $\mathrm{Gen}(SA, \tau)$, wobei S die

aktuelle Substitution ist, die aus der State-Monade gelesen werden kann, und τ der Typ ist, der vom Algorithmus $\mathcal{W}$ ermittelt wurde.

Damit sind die Typschemata der Funktionen in der Deklarationsgruppe ermittelt und in die Werte-Umgebung geschrieben, sodass andere Deklarationsgruppen, die von dieser Deklarationsgruppe abhängig sind, auf die Typschemata der Funktionen der betrachteten Deklarationsgruppe zugreifen können.

Typvariablen In der vorliegenden Implementierung des Typchecks wird der Konvention gefolgt, dass Typvariablen durch ganze Zahlen dargestellt werden. Neue Typvariablen werden fortlaufend nummeriert, und sind negativ. Die Nummern für neue Typvariablen werden also in folgender Reihenfolge vergeben: $-1, -2, -3, \ldots$.

Die durch die Generalisierung erzeugten Typschemata für Funktionen benutzen hingegen als Typvariablen Zahlen von 0 an aufsteigend. Der Typ $\forall$ `a b. a -> b -> (a, b)` wird zum Beispiel als $\forall$ `0 1 . 0 -> 1 -> (0, 1)` gespeichert.

Die Typen von Variablen in *Pattern-Deklarationen* werden nicht generalisiert, es werden also negative Typvariablen verwendet. Pattern-Deklarationen sind Deklarationen, bei denen auf der linken Seite Patterns benutzt werden, wie in `Just x = ...`. Pattern-Deklarationen können nur in `let`-Ausdrücken oder `where`-Klauseln verwendet werden.

Auch Definitionen der Form „`x = ...`“ in `let`-Ausdrücken werden als Pattern-Deklarationen aufgefasst. Dadurch wird erreicht, dass Variablen in `let`-Ausdrücken *monomorph* behandelt werden. Die Gründe dafür wurden bereits in Abschnitt 3.1.1 aufgeführt.

Ermittlung der Deklarationsgruppen Für die Ermittlung der Deklarationsgruppen wird ein Graph betrachtet, der als Knoten die einzelnen Funktionen enthält. Eine Kante $f_1 \to f_2$ wird dann im Graph eingezeichnet, wenn die Funktion f_1 die Funktion f_2 benutzt. In dem so erhaltenen Graph entsprechen die Deklarationsgruppen genau den *starken Zusammenhangskomponenten* des Graphen. Eine starke Zusammenhangskomponente in einem Graphen ist eine Menge von Knoten $\{k_1, \ldots, k_n\}$, wobei über die gerichteten Kanten von jedem k_i zu einem k_j gelangt werden kann; diese Menge muss außerdem maximal sein, das heißt, es kann kein Knoten mehr hinzugefügt werden, ohne die erste Eigenschaft zu verletzen.

Ein Beispiel für die Ermittlung von Deklarationsgruppen ist in Abbildung 4.3 aufgeführt.

Berücksichtigung von Typsignaturen Bei der Typüberprüfung der linken Seiten der Funktionen einer Deklarationsgruppe wird eine vorhandene Typsignatur schon vorläufig zur Definierung des Funktionstyps verwendet. Bei der Generalisierung der Funktionstypen wird dann überprüft, ob die inferierten Funktionstypen mit den in den Typsignaturen angegebenen Funktionstypen übereinstimmen. Da die inferierten Funktionstypen nur spezifischer als die Typen in den Typsignaturen werden können, bedeutet für eine Funktion ein Abweichen des inferierten Typs vom deklarierten Typ, dass der Typ in der Typsignatur *zu allgemein* ist.

```
f x = g x

g x = h x && i x
h x = g x && k x

i x = j x
j x = k x
k x = i x
```

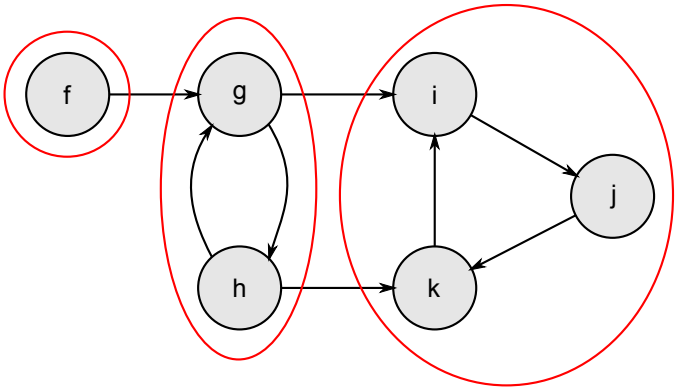

Abbildung 4.3: Beispiel für die Ermittlung von Deklarationsgruppen. Links sind die Funktionen aufgeführt, für die auf der rechten Seite der Abhängigkeitsgraph gebildet wurde. Die Deklarationsgruppen, die sich aufgrund der starken Zusammenhangskomponenten im Graphen ergeben, sind {`f`}, {`g`, `h`} und {`i`, `j`, `k`}.

Erweiterung des Algorithmus $\mathcal{W}$ auf andere Konstrukte des abstrakten Syntaxbaums
Der Algorithmus $\mathcal{W}$ kann leicht auf andere Konstrukte des abstrakten Syntaxbaums erweitert werden, die meistens lediglich syntaktischen Zucker darstellen. So können zum Beispiel `where`-Klauseln auf `let`-Ausdrücke, sowie Sections und die Infix-Anwendung von Operatoren auf die Applikation zurückgeführt werden. Auch Konstrukte wie Listen können aufgrund der Kapselung der Substitutionen in der Statemonade einfach behandelt werden, indem nacheinander die Typen der einzelnen Listenelemente unifiziert werden. Ohne die implizite Handhabung der Substitutionen müssten dabei die schon ermittelten Substitutionen immer korrekt auf die Umgebung und die schon ermittelten Typen angewandt werden, was schon bei der Beschreibung des Algorithmus $\mathcal{W}$ nicht selbstverständlich ist. Dadurch, dass durch die Unifizierung implizit die Substitutionen geändert werden, können auch andere Konstrukte leicht behandelt werden: Bei If-Then-Else's werden der Typ der Bedingung mit dem Boolean-Typ, und die Typen der beiden möglichen Zweige miteinander unifiziert. Bei Case-Ausdrücken schließlich muss der Typ des Ausdrucks, auf den gematcht wird, mit den Typen aller Patterns in den Alternativen unifiziert werden.

4.2.2.7. Export Check

Im Export-Check wird die Exportdeklaration des aktuellen Moduls überprüft. Zum einen wird geprüft, ob die in der Exportdeklaration angegebenen Entitäten definiert sind. Zum anderen wird ein Fehler gemeldet, falls eine angegebene Entität mehrdeutig ist, wenn zum Beispiel im aktuellen Modul ein Datentyp deklariert wurde und außerdem ein Datentyp mit demselben Namen importiert wurde, und in der Exportspezifikation der Name nicht qualifiziert ist. Beim Export von Datentypen über einen unqualifizierten Namen gilt *nicht*

die Regel, dass lokale Definitionen von Datentypen importierte Datentypen überdecken. Der unqualifizierte Name darf nur auf *genau einen* Datentyp verweisen.

Funktionen werden exportiert, indem ihr Name in der Exportdeklaration angegeben wird. Ein Datentyp `data T` $a_1 \ldots a_n$ `=` $\mathtt{C}_1\ \tau_{11} \ldots \tau_{1k_1}$ `|` `...` `|` $\mathtt{C}_n\ \tau_{n1} \ldots \tau_{nk_n}$ kann über drei verschiedene Arten exportiert werden:

`T` Es wird lediglich der Typkonstruktor des Datentyps exportiert. Dies entspricht der Exportspezifikation `T()`.

`T(..)` Es wird der Typkonstruktor und alle Datenkonstruktoren von `T` exportiert. Dies entspricht der Exportspezifikation `T(`$\mathtt{C}_1$`, ..., `$\mathtt{C}_n$`)`.

`T(`$\mathtt{C}_{l_1}$`, ..., `$\mathtt{C}_{l_m}$`)` Es werden der Typkonstruktor `T` und die Datenkonstruktoren $\mathtt{C}_{l_1}, \ldots, \mathtt{C}_{l_m}$ exportiert.

Beim Prüfen der Exportspezifikation werden Angaben der Form `T` und `T(..)` in `T()` bzw. `T(`$\mathtt{C}_1$`, ..., `$\mathtt{C}_n$`)` konvertiert, sodass immer explizit die zu exportierenden Datenkonstruktoren angegeben sind.

Zusätzlich ist es erlaubt, importierte Module zu *re-exportieren*, also alle importierte Entitäten eines Moduls für Nutzer des aktuellen Moduls verfügbar zu machen. Dies wird für ein importiertes Modul `M` durch die Angabe „`module M`“ in der Exportspezifikation erreicht.

4.2.2.8. Qualification

Bevor der Code die weiteren Transformationsphasen durchläuft, werden alle vorkommenden Funktionen, Daten- und Typkonstruktoren *qualifiziert*. Das heißt, dass jeweils der im Code auftretende Name durch den *Originalnamen* ersetzt wird, also durch den Namen, der angibt, in welchem Modul die Funktion oder der Konstruktor definiert wurde. Bei lokal definierten Entitäten wird der Name des Moduls, das gerade kompiliert wird, hinzugefügt; bei einer importierten Entität wird der Modulname, in dem diese Entität definiert wurde, als Qualifizierung benutzt. Parameter von Funktionen sowie lokale Funktionsdeklarationen werden nicht qualifiziert.

Beispiel 4.8:
Gegeben sei folgender Code:

```
module Module2 where
  someFun x = x

module Module1 where
  import Module2 as M

  fun x = M.someFun x
  fun2 x = fun x
```

Durch die Qualifizierung werden die Funktionsdeklarationen in Modul `Module1` in folgenden Code geändert:

```
fun x = Module2.someFun x
fun2 x = Module1.fun x
```

Auch die Werteumgebung wird aktualisiert, sodass nun auch unter den qualifizierten Namen nachgeschlagen werden kann, welchen Typ die entsprechenden Entitäten besitzen.

Diese Qualifikationstransformation ist deshalb wichtig, da sie die nachfolgenden Transformationen des abstrakten Syntaxbaumes ermöglicht. Nun ist es möglich, Funktionen unter ihrem qualifizierten Originalnamen im Code einzufügen, ohne berücksichtigen zu müssen, ob diese Funktionen überhaupt oder unter welchem Namen sie importiert wurden.

4.2.2.9. Desugarization

In dieser Phase wird jeglicher „syntaktischer Zucker“ aus dem Quellcode entfernt. Syntaktischer Zucker sind meist abkürzende Schreibweisen, die eine vereinfachte Darstellung des Quellcodes ermöglichen. Beispiele hierfür sind Sections und arithmetische Sequenzen.

Durch die „Entzuckerung“ werden folgende Konstrukte aus Ausdrücken entfernt:

Left- und Right-Sections Eine Left-Section „`(e ○)`“ wird durch „`(○) e`“ ersetzt. Eine Right-Section „`(○ e)`“ wird durch „`flip ○ e`“ ersetzt.

Infix-Operatoren Ein Ausdruck „`e ○ e'`“ wird in „`(○) e e'`“ übersetzt.

Arithmetische Sequenzen Arithmetische Sequenzen der Formen `[a ..]`, `[a, b ..]`, `[a .. b]` und `[a, b .. c]` werden jeweils durch die Prelude-Funktionen `enumFrom`, `enumFromThen`, `enumFromTo` und `enumFromThenTo` ersetzt.

Klammern Explizite Klammerungen können entfernt werden, da diese sowieso durch die Struktur des abstrakten Syntaxbaums dargestellt werden.

Tupel und Listen Tupel und Listen, für die im abstrakten Syntaxbaum besondere Datenkonstruktoren vorgesehen sind, werden in Konstruktorapplikationen übersetzt.

List comprehesions List comprehensions werden in `case`-Ausdrücke und `foldr`-Konstruktionen übersetzt.

If-Then-Else If-Then-Else-Konstrukte werden durch Case-Ausdrücke ersetzt.

Ausdrücke bestehen danach nur noch aus Literalen, Variablen, Konstruktoren, Applikationen, Let-Ausdrücken und Case-Ausdrücken.

Auch die Patterns werden vereinfacht:

Tupel- und Listenpatterns Wie bei Ausdrücken werden Tupel- und Listenpatterns durch entsprechende Konstruktorapplikationen ersetzt.

Infix-Patterns Auch Infix-Patterns werden durch entsprechende Konstruktorpatterns ersetzt.

Klammern Explizite Klammerungen von Patterns werden entfernt, da im abstrakten Syntaxbaum sowieso schon die explizite Klammerung repräsentiert ist.

Danach bestehen Patterns nur noch aus Literalen, Variablen, Konstruktorapplikationen und As-Patterns.

Weitere Transformationen sind:

- Bei Funktionsgleichungen mit *Guards* werden die Guards entfernt, indem sie in geschachtelte If-Then-Else's (bei Booleschen Guards) oder Case-Ausdrücke (bei Guards mit Constraints) übersetzt werden.
- Die Deklarationen in der `where`-Klausel einer Gleichung werden in einen neu erzeugten Let-Ausdruck, der die ursprüngliche rechte Seite der Gleichung umschließt, verschoben.

4.2.2.10. Simplification

Im Vereinfachungsschritt werden einige einfache Vereinfachungen des Codes durchgeführt:

- Wird im Code eine Funktion verwendet, die nur aus einer Variablen, einem Literal oder einer nullstelligen Funktion besteht, so wird der Funktionsrumpf dieser Funktion eingesetzt. Dieser Vorgang wird *Inlining* genannt.
- Verschachtelte Pattern-Deklarationen (die nur in lokalen Scopes erlaubt sind) werden in Gleichungen für die vorkommenden Variablen transformiert. Dabei werden Selektorfunktionen benutzt, um aus dem Gesamtergebnis die Ergebnisse für die einzelnen Variablen zu extrahieren.
- Ungenutzte Deklarationen werden entfernt.

4.2.2.11. Lifting

Durch das *Lifting* werden alle lokalen Funktionsdefinitionen in Top-Level-Definitionen transformiert. Dabei erhalten die lokalen Funktionen Parameter für ihre frei vorkommenden Variablen, und alle Aufrufe der Funktionen werden so angepasst, dass die vorher frei vorkommenden Variablen den Funktionen nun als Argumente übergeben werden. Anschließend werden alle lokalen Funktionen auf Top-Level Ebene verschoben.

Beispiel 4.9:
Der Code

```
fun x = fun2 1
  where
    fun2 y = (x, y)
```

wird in

```
fun x = fun2.lifted x 1
fun2.lifted x y = (x, y)
```

transformiert.

4.2.2.12. Übersetzung in Zwischensprache

In dieser Transformation wird der abstrakte Syntaxbaum in eine vereinfachte Zwischensprache (*intermediate language, IL*) übersetzt. Die meisten Elemente des abstrakten Syntaxbaums können ohne Änderungen in die Zwischensprache übersetzt werden.

Bei der Konversion wird aber das *Pattern Matching* in Funktionsregeln und Case-Ausdrücken entfernt bzw. vereinfacht. Die Funktionen werden so transformiert, dass sie nur noch *Variablen* als Parameter besitzen, und aus einer einzigen Gleichung bestehen. In Case-Ausdrücken werden die Alternativen so vereinfacht, dass auf den linken Seiten der Alternativen als Patterns nur noch Variablen, Literale und Konstruktorapplikationen auf Variablen vorkommen.

Beispiel 4.10:
Der Code

```
fun (Just [x]) = x
fun Nothing    = error "err"
```

wird in

```
fun x =
  case x of
    Just x' ->
      case x' of
        x'':xs ->
          case xs of
            [] -> x''
    Nothing -> error "err"
```

übersetzt.

Überlappende Patterns in Funktionsdefinitionen werden in expliziten Nichtdeterminismus übersetzt, der durch `Or`-Konstrukte dargestellt wird. So wird die Funktionsdefinition `choice = 1; choice = 2; choice = 3` übersetzt in `choice = 1 | 2 | 3`, wobei `|` für das explizite nichtdeterministische `Or` steht.

4.2.2.13. Case Completion

In dieser Phase werden unvollständige Case-Ausdrücke, also Case-Ausdrücke, in denen nicht alle möglichen Alternativen behandelt werden, vervollständigt, indem Code für die fehlenden Alternativen hinzugefügt wird. Diese Phase ist für den *PAKCS*-Compiler

notwendig, der vollständige Case-Ausdrücke erwartet. Case-Ausdrücke, die nur eine Alternative mit einer Variablen als Pattern besitzen, werden in einen Let-Ausdruck transformiert, da FlatCurry nur Konstruktorpatterns unterstützt.

4.2.2.14. Transformation in FlatCurry

Als letztes wird die Zwischensprache nach FlatCurry übersetzt. Da die Zwischensprache schon stark FlatCurry ähnelt, werden hier keine größeren Transformationen mehr durchgeführt.

4.2.3. Modulsystem

Die Aufteilung eines Programms in Module ermöglicht eine einfache Wiederverwendung von Code und die Kapselung von Funktionalitäten. Auch Curry unterstützt die Verwendung von Modulen.

4.2.3.1. Export

In jedem Modul kann angegeben werden, welche definierten Entitäten nach außen sichtbar sein sollen, und welche nicht. Somit ist es zum Beispiel möglich, abstrakte Datentypen mit einer klar definierten Schnittstelle zu definieren, und die eigentliche Implementierung zu verbergen.

Im Modulkopf kann angegeben werden, welche Entitäten aus dem Modul exportiert werden sollen. Dies können sein:

- Funktionen
- Algebraische Datentypen
- Typsynonyme

Algebraische Datentypen können mit oder ohne ihre Datenkonstruktoren exportiert werden. Ein Export nur des Typkonstruktors ermöglicht es, die interne Darstellung eines Datentyps zu verbergen. So können zum Beispiel Mengen als Listen oder als Bäume implementiert sein; wenn nur der Mengen-*Typkonstruktor* exportiert wird, können Mengen zwar verwendet werden, aber es kann nicht auf die interne Repräsentation zugegriffen werden.

Sind im Quellcode Präzedenzen für exportierte Entitäten festgelegt worden, so werden die Präzedenzen automatisch mit exportiert.

Alle exportierten Entitäten werden in eine *Interface-Datei* geschrieben. Bei Funktionen wird lediglich der Typ abgespeichert; für Datentypen werden nur die exportierten Datenkonstruktoren aufgeführt. Die Verwendung eines Interfaces ermöglicht den Import von in anderen Modulen definierten Entitäten, ohne dass diese Module neu kompiliert werden müssen.

Als Grundlage für den Export dient das transformierte Modul nach der Qualifikationsphase, das noch nicht die weiteren Transformationsschritte durchlaufen hat (siehe Abb. 4.4).

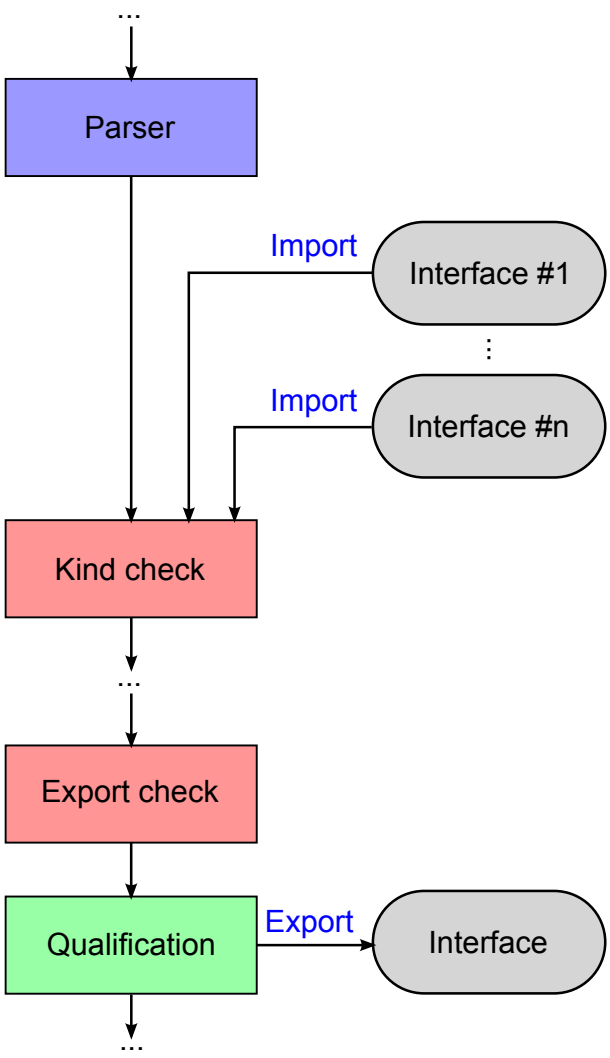

Abbildung 4.4: Übersicht über die Einbindung von Import und Export in den Kompilationsprozess

4.2.3.2. Import

Die exportierten Entitäten eines Modules können in ein anderes Modul *importiert* werden. Dies geschieht über *Importspezifikationen*. Die Backus-Naur-Form für Importspezifikationen lautet wie folgt:

$$
\begin{array}{rcll}
\textit{ImportDecl} & ::= & \texttt{import}\ [\texttt{qualified}]\ \textit{ModuleID}\ [\texttt{as}\ \textit{ModuleID}]\ [\textit{ImportRestr}] & \\
\textit{ImportRestr} & ::= & \texttt{(}\ \textit{Import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{Import}_n\ \texttt{)} & (n \geq 0) \\
& | & \texttt{hiding (}\ \textit{Import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{Import}_n\ \texttt{)} & (n \geq 0)
\end{array}
$$

Importe können alle exportierten Elemente eines Moduls umfassen, oder nur bestimmte; für ersteres wird die Import-Restriktion *ImportRestr* weggelassen; für letzteres werden in der Import-Restriktion entweder die zu importierenden Elemente aufgelistet, oder es wird angegeben, welche Elemente *nicht* importiert werden sollen (mit Hilfe des Schlüsselworts `hiding`). Bei einem Import ohne „`as`“ und „`qualified`“ sind die importierten Elemente sowohl unter ihrem unqualifizierten als auch unter dem mit dem Modulnamen qualifizierten Namen zugreifbar. Bei einem Import mit „`as`“ wird statt des Modulnamens der angegebene Name verwendet. Durch die Angabe „`qualified`“ wird schließlich erreicht, dass die Entitäten *nur* unter dem qualifizierten Namen importiert werden.

Wie beim Export können für algebraische Datentypen entweder nur der Typkonstruktor oder zusätzlich bestimmte oder alle Datenkonstruktoren importiert werden.

Der Import wird vor dem Beginn des Kompilationsprozesses durchgeführt (siehe Abb. 4.4). Für jede angegebene Import-Spezifikation wird die entsprechende Interface-Datei des zu importierenden Moduls geladen. Danach werden – gemäß der Importspezifikation – die zu importierenden Entitäten in die unterschiedlichen Umgebungen der Compiler-Umgebung eingetragen, so dass diese von Beginn an im Kompilationsprozess zur Verfügung stehen.

4.3. Implementierung von Typklassen

4.3.1. Anpassung der Compiler-Umgebung

Da nun als Entitäten auch *Typklassen* und *Instanzen* im Code vorkommen können, muss die Compiler-Umgebung entsprechend angepasst werden. Der Compiler-Umgebung wird dazu eine *Klassen-Umgebung* hinzugefügt. Außerdem wird die Werte-Umgebung angepasst.

4.3.1.1. Klassen-Umgebung

Die Klassen-Umgebung enthält die folgenden Elemente:

- Eine Umgebung im Sinne von Abschnitt 4.2.1, die als Zuordnung von Klassennamen zu Klassen dient. Diese wird ebenfalls Klassen-Umgebung genannt. Diese Umgebung verfolgt das gleiche Konzept von Entitäten wie die anderen Umgebungen auch, das heißt, ein Name kann durchaus wieder auf mehrere Klassen

verweisen. Die Originalnamen von Klassen sind wie bei Funktionen die Namen, die angeben, in welchem Modul die Klassen ursprünglich definiert wurden. Auch die Merge-Methode wird für Klassen definiert; diese wird weiter unten und in Abschnitt 4.3.3.4 genauer beschrieben. Lokal definierte Klassen verbergen importierte Klassen mit demselben Namen; damit werden in dieser Hinsicht Klassen genauso behandelt wie die anderen Entitäten auch.

- Eine Umgebung, die Instanzen speichert. Da Instanzen keine Namen besitzen, wird hier keine Zuordnung von Namen zu Instanzen vorgenommen, sondern die Instanzen werden einfach in der Umgebung abgespeichert. Ansonsten verhält sich die Umgebung wie die anderen Umgebungen auch, insbesondere wird bei einem Import derselben Instanz auf zwei Wegen diese Instanz in der Instanzumgebung nicht doppelt aufgeführt; wenn aber Instanzen für dieselbe Klasse und denselben Typ in zwei verschiedenen Modulen deklariert, und beide importiert werden, so werden sehr wohl zwei Instanzen in die Instanzumgebung aufgenommen, da diese sich voneinander unterscheiden. Um zwischen diesen beiden Fällen zu unterscheiden, wird für die Instanzen zusätzlich noch deren Ursprung gespeichert, also das Modul, in dem sie definiert wurden. Ist der Ursprung für zwei Instanzen mit derselben Klasse und demselben Typ der gleiche, so wird nur ein Eintrag in der Instanzenumgebung erstellt, ansonsten zwei.

  Wie bei den anderen Umgebungen auch kommt es zu einem *ambiguous*-Compilerfehler, wenn eine Instanz benötigt wird, aber mehrere Instanzen zur Auswahl stehen, von denen keine lokal definiert ist.

  In der Instanzumgebung werden die in den Instanzen benutzten Klassen und Typen unter ihren *Originalnamen* aufgeführt. Dies ist für die Funktionen, die auf der Instanz- und der Klassenumgebung arbeiten, wichtig, da diese nur die Originalnamen von Klassen und Typen verwenden, und nicht die Namen, unter denen diese im Code angesprochen werden.

- Eine Umgebung, die Klassenmethoden auf die Klassen abbildet, in denen sie deklariert wurden. Diese Umgebung dient dazu, in dem modifizierten Typcheck die Klassenmethoden zu der Werte-Umgebung hinzuzufügen.

- Eine *Zuordnung* (keine Umgebung!), die den *Originalnamen* von Klassen die entsprechenden Klassen zuordnet. Diese Zuordnung ist für die Kontextreduktion und andere Funktionen, die auf Klassen arbeiten, wichtig, da diese – wie oben schon erwähnt – ausschließlich die Originalnamen verwenden, und nicht die Namen, unter denen die Klassen im Code angesprochen werden. Da alle Originalnamen eindeutig sind, muss hier keine Umgebung im Sinne von Abschnitt 4.2.1 benutzt werden, sondern es reicht eine simple Zuordnung des Datentyps „`Map`“.

  Als Beispiel für die Verwendung dieser Zuordnung sei die Ermittlung von allen Superklassen einer Klasse aufgeführt. Zuerst wird für den entsprechenden Klassennamen nachgeschlagen, welche Klasse zu ihm gehört. In der ermittelten Klasse sind dann die *Originalnamen* der direkten Superklassen gespeichert. Diese werden

dann benutzt, um in der Zuordnung die Klassen für die Superklassen nachzuschlagen, die wiederum die Originalnamen ihrer Superklassen enthalten, für die wieder die entsprechenden Klassen in der Zuordnung nachgeschlagen werden usw. Dieser Vorgang wird solange wiederholt, bis alle Superklassen ermittelt wurden.

Für Klassen werden die folgenden Daten gespeichert:

- Der Originalname der Klasse
- Die Klassen im Superklassenkontext (unter deren Originalnamen)
- Die Typvariable der Klasse
- Die Typen der Klassenmethoden, wie sie im Code stehen
- Die Typschemata für die Klassenmethoden
- Ein Flag, das angibt, ob die Klasse verborgen ist
- Eine Menge, die angibt, welche Klassenmethoden sichtbar sind
- Die Default-Methoden

Der Superklassen-Kontext kann ohne die Typvariablen abgespeichert werden, da alle Variablen im Kontext der Typvariablen der Klasse entsprechen.

Klassen können *verborgen* sein, das heißt, sie werden zwar in der Klassenumgebung gespeichert, können aber nicht benutzt werden. Manche Klassen müssen deshalb in die Klassenumgebung aufgenommen werden, damit die Klassenhierarchie in der Klassenumgebung keine Lücken aufweist. Nur mit einer kompletten Klassenhierarchie ist es zum Beispiel möglich, die korrekten Typen der Wörterbücher zu ermitteln. Damit die aus diesem Grund eingefügten Klassen nicht benutzt werden können, werden sie als versteckt markiert.

Auch Klassen*methoden* können als versteckt markiert werden. Dies ist notwendig, weil immer *alle* Methoden einer Klasse bekannt sein müssen (da sonst ihr Wörterbuchtyp nicht korrekt ermittelt werden könnte), aber trotzdem der Export und Import von nur einem *Teil* der Methoden möglich sein soll (siehe Abschnitt 4.3.3).

Da auf verschiedenen Importwegen verschiedene Klassenmethoden derselben Klasse importiert werden können, muss für Klassen die für Entitäten notwendige Merge-Methode entsprechend angepasst werden. Beim Mergen werden die Mengen der sichtbaren Klassenmethoden vereinigt, so dass nun alle Klassenmethoden sichtbar sind, die in der einen oder der anderen Importspezifikation angegeben wurden. Dies entspricht dem Vorgehen beim Import von Datenkonstruktoren für einen bestimmten algebraischen Datentyp.

Die Typschemata für die Klassenmethoden einer Klasse `C` mit der Klassenvariablen `a` werden gebildet, indem zu den angegebenen Typen der Klassenmethoden der Kontext „`C a`“ hinzugefügt wird: Lautet der Typ einer Klassenmethode τ, so lautet das entsprechende

Typschema `C a =>` τ. Die Typschemata sind nie mehrdeutig, da gefordert wird, dass die Typen τ immer die Typvariable `a` enthalten.

Für Instanzen werden die folgenden Daten gespeichert:

- Der Originalname der Klasse, deren Klassenmethoden überladen werden
- Der Typ, für den die Klassenmethoden überladen werden; dabei sowohl der Typkonstruktor (wieder der Originalname) als auch die angegebenen Typvariablen
- Die Implementierungen der Klassenmethoden
- Der Instanzkontext. Da im Gegensatz zu einer Klassendeklaration hier auch unterschiedliche Typvariablen benutzt werden können, müssen diese mitgespeichert werden. Alle vorkommenden Klassen werden unter ihrem Originalnamen abgespeichert.

Ein Beispiel für eine konkrete Klassenumgebung ist in Abb. 4.5 zu finden.

4.3.1.2. Werte-Umgebung

In der Werte-Umgebung wird für Funktionen nun zusätzlich noch gespeichert, ob die jeweilige Funktion eine *Klassenmethode* ist oder nicht. Ist die Funktion eine Klassenmethode, so wird außerdem die Klasse abgespeichert, in der die Methode definiert wurde.

Die Information, ob eine Funktion eine Klassenmethode ist oder nicht, wird in der Transformation 2.7 benötigt, in der die konkreten Wörterbücher eingefügt werden. Ist die Funktion eine Klassenmethode, so wird sie durch die entsprechende Selektionsfunktion ersetzt, ansonsten wird der Funktionsname beibehalten.

4.3.2. Anpassung des Kompilationsverlaufes

Für die Implementierung von Typklassen müssen alle Kompilationsphasen bis einschließlich der Qualifikation modifiziert, und zusätzlich neue Kompilationsphasen eingefügt werden. Eine Übersicht über den neuen Kompilationsprozess ist in Abb. 4.6 zu finden. Dort sind sowohl die modifizierten Compilerphasen als auch die neu hinzukommenden Compilerphasen aufgeführt. Neu sind:

- der „Type Classes Check“, in dem Prüfungen der Typklassenelemente im Code, die Ableitung von Instanzen für algebraische Datentypen sowie die Transformation von Klassen und Instanzen durchgeführt werden;
- die „Dictionaries“-Transformation, in der die Wörterbücher und Wörterbuchparameter in den Code eingefügt werden;
- die „Typesignatures“-Transformation, in der die Typsignaturen von überladenen Funktionen im Code korrigiert werden, da in der Wörterbuch-Transformation neue Funktionsparameter eingefügt wurden, und somit die im Code stehenden Typsignaturen nicht mehr den eigentlichen Typsignaturen entsprechen.

Gegebene Module:

```
module M1 where

  class C a where
    funC :: a -> a
    funC x = x

module M2 where

  import M1 as M3

  class C a => D a where
    funD :: a -> Bool

  instance C Bool where
    funC x = not x

  instance C Int where
    funC x = x

  instance D Bool where
    funD x = x

  data T a = T a

  instance C a =>
           C (T a) where
    funC x = x
```

Klassenumgebung bei der Kompilation von Modul `M2`:

Klassenumgebung:

`C`	→ class `M1.C`
`M3.C`	→ class `M1.C`
`D`	→ class `M2.D`
`M2.D`	→ class `M2.D`

Instanzumgebung:

Klasse, Typ	→ Kontext	Typvariablen
`M1.C Prelude.Bool`	→ ∅	∅
`M1.C Prelude.Int`	→ ∅	∅
`M2.D Prelude.Bool`	→ ∅	∅
`M1.C M2.T`	→ `M1.C a`	`a`

Zuordnung Originalnamen ↦ Klassen:

`M1.C`	→ class `M1.C`
`M2.D`	→ class `M2.D`

Für die Klassen werden folgende Informationen gespeichert:

class `M1.C`:
- sichtbar
- Superklassen: ∅
- Klassenname: `M1.C`
- Klassenvariable: `a`
- Methoden: `funC :: a -> a`
- Defaultmethoden: {`funC`}
- Typschemata: `funC :: C a => a -> a`
- sichtbare Klassenmethoden: {`funC`}

class `M2.D`:
- sichtbar
- Superklassen: {`M1.C`}
- Klassenname: `M2.D`
- Klassenvariable: `a`
- Methoden: `funD :: a -> Bool`
- Defaultmethoden: ∅
- Typschemata: `funD :: D a => a -> Bool`
- sichtbare Klassenmethoden: {`funD`}

Abbildung 4.5: Beispiel für eine Klassenumgebung

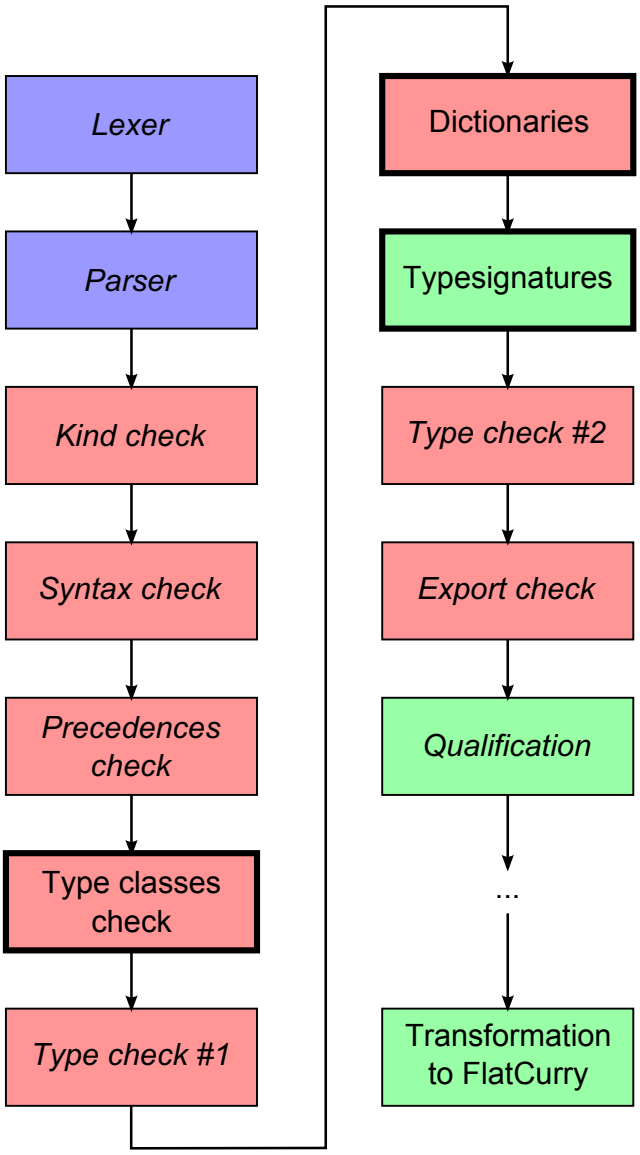

Abbildung 4.6: Modifizierter Kompilationsverlauf. Neue Kompilierstadien sind dick umrahmt, modifizierte durch kursive Schrift gekennzeichnet. Die Phasen nach der Qualification-Phase bleiben gleich, und sind daher hier ausgelassen worden.

Außerdem wird der Typcheck zweimal durchgeführt: Zuerst auf einem Programm mit Constraints und Kontexten, und danach auf dem Programm, das frei von Typklassenelementen ist. Der zweite Typcheck hat den Zweck, die Typsignaturen von überladenen Funktionen, die in der Werteumgebung gespeichert sind, zu korrigieren. Als positiver Nebeneffekt wird durch den zweiten Typcheck gleichzeitig eine Überprüfung der Korrektheit der Wörterbuchtransformation durchgeführt, was besonders in der Entwicklungsphase hilfreich ist. Wenn der erste Typcheck fehlerlos durchgelaufen ist, so sollte auch der zweite Typcheck fehlerlos durchlaufen, ansonsten ist ein Fehler bei der Wörterbuchtransformation gemacht worden.

Ein Nachteil der Einführung eines zweiten Typchecks ist, dass eventuell viele Berechnungen unnötigerweise wiederholt werden, und somit der Kompilierprozess länger dauert. Wenn im Code überhaupt keine Typklassenelemente vorkommen, so werden die beiden Typchecks auf identischem Code ausgeführt, und die zweite Ausführung des Typchecks ist quasi nutzlos. Außerdem werden zwei verschiedene Interfaces anstatt nur einem für den Import und den Export benötigt (siehe Abschnitt 4.3.3.1). Der zweite Typcheck könnte durch eine Transformation ersetzt werden, in der die Typsignaturen aller in der Werteumgebung gespeicherten überladenen Funktionen direkt angepasst werden.

Im Folgenden werden die benötigten Anpassungen der einzelnen Kompilationsphasen beschrieben bzw. die neuen Kompilationsphasen erläutert.

4.3.2.1. Lexer

Der Lexer muss um die zusätzlichen Schlüsselwörter und reservierten Operatoren erweitert werden, die durch die Einführung von Typklassen hinzukommen. Die neu eingeführten Schlüsselwörter sind `class`, `instance` und `deriving`; zusätzlich wird der Operator `=>` als Kontexttrennelement reserviert.

4.3.2.2. Parser

Der Parser muss um die Elemente, die in Anhang A.1 farbig hervorgehoben sind, erweitert werden. Die im Anhang angegebene Grammatik ist *nicht* in LL(1)-Form, sie kann also von einem LL(1)-Parser nicht erfolgreich geparst werden. Ein Beispiel für eine Stelle, bei der ein LL(1)-Parser scheitern würde, ist die Typklassendeklaration mit der folgenden EBNF:

$$
\begin{array}{rcll}
\textit{TypeClassDeclaration} & ::= & \texttt{class}\ [\textit{SimpleContext}\ \texttt{=>}]\ \textit{TypeClassID}\ \textit{BTypeVarID} & \\
 & & [\texttt{where}\ \textit{ClassDecls}] & \\
\textit{SimpleContext} & ::= & \textit{Constraint} \mid \texttt{(}\ \textit{Constraint}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{Constraint}_n\ \texttt{)} & (n \geq 0) \\
\textit{Constraint} & ::= & \textit{QTypeClassID}\ \textit{BTypeVarID} &
\end{array}
$$

Beim Parsen von „`class C a`“ kann ein LL(1)-Compiler an der markierten Stelle nicht unterscheiden, ob das `C` zum Kontext (wie in „`class C a => D a where ...`“) gehört, oder den Namen der Typklasse selber bezeichnet (wie in „`class C a where ...`“). Um das korrekte Parsen in beiden Fällen zu ermöglichen, muss der Parserkombinator `<|?>`

benutzt werden, der im Gegensatz zu dem Parserkombinator `<|>` für die einfache Alternative nicht anhand des nächsten Tokens bestimmt, welcher Parser angewendet werden soll, sondern beide Parser ausführt und dann (wenn beide erfolgreich waren) den Parser auswählt, der den längeren Präfix der zu parsenden Tokenfolge konsumiert hat.

In der konkreten Implementierung des Parsers für eine Typklassendeklaration werden die Parser für die EBNF „`class` *SimpleContext* `=>` *TypeClassID BTypeVarID*“ und die EBNF „`class` *TypeClassID BTypeVarID*“ mit dem Parserkombinator `<|?>` verknüpft und somit parallel ausgeführt; derjenige Parser, der mehr Token konsumiert hat, entspricht in diesem Fall tatsächlich dem gewünschten Parser. Auf dieselbe Weise werden auch die Instanzdeklarationen geparst, bei denen dasselbe Problem wie oben auftritt.

4.3.2.3. Kind Check

Der Kind Check prüft wie schon weiter oben geschildert die Typsignaturen und führt eine Unterscheidung zwischen nullstelligen Typkonstruktoren und Typvariablen durch. Der originale Check kann leicht darauf erweitert werden, dass er auch die Typsignaturen in Klassen- und Instanzdefinitionen prüft und dort die genannte Unterscheidung durchführt.

4.3.2.4. Syntax Check

Der Syntaxcheck muss so erweitert werden, dass auch Funktionsdefinitionen in Klassen- und Instanzdeklarationen geprüft und transformiert werden (also die Defaultmethoden in Klassendeklarationen und die Implementierungen von Klassenmethoden in Instanzdeklarationen). Dazu werden die Funktionsdeklarationen in einer Instanz- oder Klassendeklaration als eine Deklarationsgruppe aufgefasst und geprüft. Dabei muss aber die Überprüfung deaktiviert werden, dass für jede Typsignatur ein entsprechender Funktionsrumpf vorhanden ist, da in Klassendeklarationen durchaus Typsignaturen *ohne* Implementierungen auftreten können (wenn keine Default-Implementierung angegeben wurde).

Der Syntax Check prüft bei jeder im Code vorkommenden Funktion, ob diese definiert ist. Dazu zieht er eine Menge heran, in der alle Funktionen eingetragen sind, die an der entsprechenden Stelle sichtbar sind. Deshalb müssen sowohl die Klassenmethoden im aktuellen Modul als auch die importierten Klassenmethoden in diese Menge aufgenommen werden, sodass bei der Benutzung von Klassenmethoden keine Fehler gemeldet werden. Importierte Klassenmethoden können sichtbar oder nicht sichtbar sein; es werden aber nur die *sichtbaren* Klassenmethoden in die Menge der definierten Funktionen aufgenommen. Damit wird sichergestellt, dass verborgene Klassenmethoden nicht benutzt werden können.

Außerdem wird der Syntax Check um die Prüfung erweitert, dass keine Klassenmethoden und Funktionen im aktuellen Modul denselben Namen besitzen. Dass zwei *Klassenmethoden* in verschiedenen Klassen im aktuellen Modul nicht dieselben Namen besitzen, wird an anderer Stelle überprüft (nämlich im „Type Classes Check“, siehe Abschnitt 4.3.2.6).

4.3.2.5. Precedences Check

Durch den Precedences Check werden – wie schon vorher erläutert – Ausdrücke im abstrakten Syntaxbaum so umgeordnet, dass die Präzedenzen und Assoziativitäten der vorkommenden Operatoren berücksichtigt sind. Diese Umordnung der Ausdrücke muss nun ebenfalls in den Defaultmethoden von Klassen und in den Implementierungen von Klassenmethoden in Instanzdefinitionen durchgeführt werden. Der ursprüngliche Precedences Check lässt sich leicht so erweitern, dass die genannten Klassen- und Instanzelemente berücksichtigt werden.

Auch für *Klassenmethoden* können Präzedenzen angegeben werden. Diese werden dann berücksichtigt, wenn Klassenmethoden in Infixposition als Operatoren genutzt werden. Auch die Präzedenzen von Klassenmethoden können leicht in den ursprünglichen Check integriert werden, so dass diese bei den Umordnungen berücksichtigt werden.

4.3.2.6. Type Classes Check

Der Type Classes Check ist folgendermaßen aufgeteilt:

- Zuerst werden Instanzen für Datentypen mit einer `deriving`-Klausel erstellt (wie in Abschnitt 2.4.1 beschrieben).
- Danach wird eine Überprüfung aller im Code vorkommenden Typklassenelemente durchgeführt, einschließlich der erzeugten Instanzen.
- Anschließend werden die Klassen- und Instanzdefinitionen wie in Abschnitt 2.4.2 beschrieben transformiert.

Außerdem werden, wenn alle Klassennamen in Kontexten von Typsignaturen geprüft wurden und korrekt sind, all diese Klassennamen durch ihre *Originalnamen* ersetzt, ein Vorgang, der dem Vorgehen für andere Codeelemente in der Qualifizierungsphase entspricht.

Im Folgenden werden die oben aufgeführten drei Teile des Type Classes Checks erläutert.

Schritt 1: Ableitung der Instanzen für algebraische Datentypen Im Allgemeinen wird die Ableitung von Instanzen für algebraische Datentypen wie in Abschnitt 2.4.1 beschrieben durchgeführt. Bei der praktischen Umsetzung ergeben sich aber einige Probleme, die zusätzlich gelöst werden müssen.

Generierung neuer Parameternamen Ein Problem ist, dass während der Ableitung von Instanzen neue Parameter für die Funktionsimplementierungen benötigt werden. Diese können nicht einfach „`x`“ oder „`y`“ genannt werden, sondern müssen im gesamten Modul eindeutig sein, damit die Werteumgebung, die im Compileprozess mitgeführt wird, *flach* bleiben kann. Da der Syntaxcheck, der ja alle Variablen und Parameter umbenennt, *vor* dem Type Classes Check durchgeführt wird, müssen in den darauffolgenden Checks explizit eindeutige Variablennamen erstellt werden.

Bei der Ableitung von Instanzen werden neue Variablennamen mit Hilfe eines Zählers erstellt, der für jeden neuen Variablennamen inkrementiert wird. Der Zähler wird in dem Zustand einer Zustandsmonade mitgeführt, so dass das Inkrementieren und das Mitführen des Zählers sehr einfach sind.

Prelude Die Klassen, für die ein Ableiten der Instanzen möglich sein soll, werden in einem Modul mit dem Namen *Prelude* deklariert. Dieses Modul wird automatisch in alle anderen Module eingebunden.

Position der Ableitung von Instanzen im gesamten Type Classes Check Die Ableitung der Instanzen wird, wie oben schon erwähnt wurde, *vor* der Prüfung aller Typklassenelemente, also auch vor der Prüfung der Klassennamen in den `deriving`-Klauseln, durchgeführt. Deshalb wird die Ableitung einer Instanz für eine bestimmte Klasse nur dann vorgenommen, wenn der Klassenname einer der unterstützten Klassen entspricht, und die zusätzlichen Bedingungen bei der `Enum`- bzw. der `Bounded`-Klasse erfüllt sind (siehe Abschnitte 2.4.1.4 und 2.4.1.5). Sind die Bedingungen nicht erfüllt oder wird eine Klasse nicht unterstützt, so wird keine Ableitung erstellt. Dies stellt kein Problem dar, da im zweiten Schritt des Type Classes Check nochmals für alle Klassen in den `deriving`-Klauseln geprüft wird, ob für sie Instanzen abgeleitet werden können, und dann ein Compilerfehler gemeldet wird.

Da im ersten Schritt nur der Klassen*name* betrachtet wird, um zu bestimmen, welche Instanz abgeleitet werden soll, nicht aber geprüft wird, ob die Klasse tatsächlich aus der Prelude stammt oder ob sie überhaupt definiert ist, muss diese Prüfung im zweiten Schritt des Type Classes Check nachgeholt werden. Es kann nämlich der Fall eintreten, dass eine Klasse in der `deriving`-Klausel zwar denselben Namen wie eine der unterstützen Klassen besitzt, aber aus einem unterschiedlichen Modul als der Prelude stammt und somit nicht unterstützt wird, wie im folgenden Beispiel:

```
module M where
  class Enum a where ...

module N where
  import M

  data T a b = ...
    deriving M.Enum
```

Hier wird zwar zuerst eine Ableitung für die `Enum`-Klasse erstellt, aber im zweiten Schritt festgestellt, dass die Klasse „`M.Enum`“ gar nicht unterstützt wird, und eine Fehlermeldung ausgegeben.

Dass die Ableitung von Instanzen *vor* der Prüfung der Instanzen und anderer Typklassenelemente geschieht, hat außerdem den folgenden Vorteil: Außer der Prüfung, dass für die in der `deriving`-Klausel angegebenen Klassen Instanzen abgeleitet werden können, müssen keine weiteren Prüfungen mehr für `deriving`-Klauseln durchgeführt werden. Alle notwendigen Prüfungen werden auf den generierten Instanzen in der zweiten Phase des Type Classes Checks durchgeführt.

Verwendung von Prelude-Funktionen In den abgeleiteten Instanzen werden viele Funktionen aus der Prelude benutzt. Dabei ergibt sich die Problematik, unter welchem Namen diese Funktionen eingefügt werden sollen. Die Funktionen können nicht unter ihrem Originalnamen eingefügt werden, was am Beispiel des `&&`-Operators, der bei der Ableitung der `Ord`-Instanz für den Datentyp `T` im folgenden Beispiel benötigt wird, gezeigt werden soll:

```
module M where
  a && b = <some implementation>

module M2 where
  import Prelude as P
  import M as Prelude

  data T a b = T a b
    deriving P.Ord
```

Der `&&`-Operator kann deshalb nicht unter seinem Originalnamen „`Prelude.&&`“ eingefügt werden, da dieser Name auf den `&&`-Operator aus dem Modul *`M`* verweist. Würde der Operator unter seinem Originalnamen eingefügt werden, so würde in den folgenden Kompilierphasen mit einer ganz anderen Entität als der gewünschten gearbeitet werden, und nach der Qualifikationsphase würde im erzeugten Code tatsächlich der `&&`-Operator aus dem Modul `M` stehen, und nicht der `&&`-Operator aus der Prelude.

Auch die unqualifizierten Namen können nicht verwendet werden, wie folgendes Beispiel zeigt:

```
import qualified Prelude as P

data T a b = T a b
  deriving P.Ord
```

Würde bei der Ableitung der `Ord`-Instanz der `&&`-Operator unter dem Namen `&&` eingefügt werden, so würde der anschließende Typ-Check fehlschlagen, da unter dem Namen `&&` kein Eintrag in der Werteumgebung existiert.

Als Lösung werden alle in abgeleiteten Instanzen verwendete Prelude-Entitäten unter den Namen von nicht real existierenden *Dummy-Modulen* in der Werteumgebung eingetragen; bei der Generierung der abgeleiteten Instanzen werden dann die mit dem Namen der Dummy-Module qualifizierten Funktionsnamen verwendet. Die Namen der Dummy-Module lauten „`<dummy>`“ und „`<dummyTC>`“; der erste Name wird für nicht überladene Funktionen, der zweite für überladene Funktionen verwendet.

So wird zum Beispiel der And-Operator unter dem Namen „`<dummy>.&&`“ in der Werteumgebung eingetragen, wobei als Originalname „`Prelude.&&`“ angegeben wird. In den generierten Instanzen wird der And-Operator über den Namen „`<dummy>.&&`“ angesprochen. Somit wird in den nachfolgenden Kompilierphasen im Code wie gewünscht immer auf den `&&`-Operator aus der Prelude verwiesen, und in der Qualifikationsphase wird der Name „`<dummy>.&&`“ korrektermaßen durch „`Prelude.&&`“ ersetzt.

Wichtig ist, dass sich die Dummy-Modulnamen von allen möglichen Modulnamen unterscheiden. Dies wird dadurch garantiert, dass in den Dummy-Modulnamen spitze Klammern verwenden werden, die in Modulnamen im Code nicht erlaubt sind. Ohne diese Vorkehrung kann es sonst zu Konflikten kommen, wenn der Dummy-Modulname mit dem Namen eines verwendeten Moduls übereinstimmt, und Namen von Entitäten aus diesem Modul mit den Namen von Entitäten aus der Prelude übereinstimmen. In diesem Fall würden in der Werteumgebung unter den übereinstimmenden Namen mehrere Entitäten eingetragen, was zu unerwünschten Mehrdeutigkeitsfehlern führen würde.

Hier noch abschließend ein vollständiges Beispiel für die Verwendung von Dummy-Modulnamen. Die für den algebraischen Datentyp `data T a b = T1 a | T2 a b` abgeleitete `Ord`-Instanz lautet:

```
instance (Ord a, Ord b) => Ord (T a b) where
  T1 x   <= T1 x'    = x <dummyTC>.<= x'
  T1 _   <= T2 _ _   = <dummy>.True
  T2 _ _ <= T1 _     = <dummy>.False
  T2 x y <= T2 x' y' =
     x <dummyTC>.< x' <dummy>.||
    (x <dummyTC>.== x' <dummy>.&& y <dummyTC>.<= y')
```

Hierbei sei die Einführung neuer Namen für die Parameter der Übersichtlichkeit halber außer Acht gelassen.

Dieser Code wird in der Qualifikationsphase in den folgenden – und so gewünschten – Code transformiert:

```
instance (Ord a, Ord b) => Ord (T a b) where
  T1 x   <= T1 x'    = x Prelude.<= x'
  T1 _   <= T2 _ _   = Prelude.True
  T2 _ _ <= T1 _     = Prelude.False
  T2 x y <= T2 x' y' =
     x Prelude.< x' Prelude.||
    (x Prelude.== x' Prelude.&& y Prelude.<= y')
```

Schritt 2: Überprüfung der Typklassenelemente Die Überprüfung der Typklassenelemente ist selber wieder aufgeteilt, und zwar in drei Phasen:

- Zuerst werden Checks durchgeführt, für die keine Klassenumgebung notwendig ist, wie die Überprüfung der Typvariablen.
- Danach werden Checks durchgeführt, die die Informationen benötigen, ob bestimmte Klassen oder Instanzen definiert sind, aber noch nicht die komplette Klassenumgebung bzw. Klassenhierarchie benötigen.
- Am Schluss werden „globale“ Checks durchgeführt, die die gesamte Klassenumgebung einschließlich der vollständigen Klassenhierarchie benötigen.

Die einzelnen Phasen bauen aufeinander auf. So wird in der zweiten Phase unter anderem geprüft, ob die Kontexte der Instanz- und Klassendefinitionen korrekt sind, also ob die vorkommenden Klassennamen existieren und eindeutig sind. Erst wenn das und die syntaktische Korrektheit verifiziert wurden, kann die vollständige Klassenumgebung gebildet werden, die für die dritte Phase notwendig ist.

Die Aufteilung ermöglicht es außerdem, dass in der zweiten und der dritten Phase jeweils bestimmte Voraussetzungen über die Struktur der Klassen- und Instanzdefinitionen angenommen werden können. In der zweiten Phase kann angenommen werden, dass die Syntax der Klassen- und Instanzdefinitionen korrekt ist, in der dritten Phase kann außerdem angenommen werden, dass die einzelnen Klassen- und Instanzdefinitionen korrekt sind.

Checks der ersten Phase In den Checks der ersten Phase wird die syntaktische Korrektheit der Klassen- und Instanzdefinitionen sowie der Kontexte in Typsignaturen geprüft:

- In Klassendefinitionen müssen die folgenden Bedingungen erfüllt sein:
  - Die Typvariablen, die im Superklassenkontext angegeben wurden, müssen der Klassenvariable entsprechen.
  - In allen angegebenen Typsignaturen muss die Klassenvariable mindestens einmal vorkommen.
  - In den Typsignaturen dürfen keine anderen Typvariablen als die Klassenvariable vorkommen.
- Für Instanzdefinitionen müssen die folgenden Bedingungen gelten:
  - Die Menge der Typvariablen im Instanzkontext muss eine Untermenge der Typvariablen im Instanztyp sein. Wenn also eine Instanz `instance (`$\mathtt{C}_1$ $\mathtt{a}_{k_1}$`, ..., `$\mathtt{C}_n$ $\mathtt{a}_{k_n}$`) => C (`$\mathtt{a}_1$ `...` $\mathtt{a}_m$`) where ...` gegeben ist, muss $\{k_1, \ldots, k_n\} \subseteq [1, m]$ gelten.
  - Der Typ in der Instanzdefinition muss sichtbar und eindeutig sein. Außerdem darf er kein Typsynonym sein. Hier gilt wieder die Semantik, dass lokale Typen importierte Typen überschatten.
  - Für den Typ muss gelten, dass die Anzahl der Typvariablen in der Instanzdefinition der Anzahl der Typvariablen in der Typdefinition des Typs entspricht. Ist ein Typ `T` also durch `data T` $\mathtt{a}_1$ `...` $\mathtt{a}_n$ `= ...` definiert, so muss die Instanzdefinition für diesen Typen `instance (...) => C (T` $\mathtt{b}_1$ `...` $\mathtt{b}_n$`) where ...` lauten. Der Kind des Typs in der Instanzdefinition muss also $*$ sein.
  - Es dürfen keine Typvariablen im Instanztyp doppelt vorkommen. Formal: Ist die Instanz `instance (...) => C (T` $\mathtt{a}_1$ `...` $\mathtt{a}_m$`) where ...` gegeben, so müssen die $\mathtt{a}_i$ paarweise verschieden sein, es muss also gelten $\mathtt{a}_i \neq \mathtt{a}_j$ für $i \neq j$.

- Die Typsignaturen mit Kontexten müssen die folgende syntaktische Bedingung erfüllen: Alle Typvariablen, die im Kontext auftreten, müssen auch im Typ auftreten. Formal: Es sei die Typsignatur `(`$\texttt{C}_1$ $\texttt{a}_1$`, ..., `$\texttt{C}_n$ $\texttt{a}_n$`) =>` τ gegeben. Dann muss gelten: `{`$\texttt{a}_1$`, ..., `$\texttt{a}_n$`}` $\subseteq$ typevars(τ). Ansonsten ist der Kontext *mehrdeutig (ambiguous)*.
- Außerdem wird geprüft,
  - dass im aktuellen Modul keine Klassen zweimal definiert werden,
  - dass die Klassendefinitionen keine direkten Zyklen wie in `class A a => A a where ...` besitzen,
  - dass keine Klassenmethoden im aktuellen Modul mehrmals definiert werden,
  - und dass `deriving`-Klauseln nur für nicht-leere algebraische Datentypen angegeben wurden, denn für algebraische Datentypen ohne Datenkonstruktoren können keine Instanzen abgeleitet werden.

Checks der zweiten Phase Für die Checks der zweiten Phase wird die Klassenumgebung soweit aufgebaut, dass in ihr nachgeschlagen werden kann, ob bestimmte Klassen existieren. Die Klassen, auf die verwiesen wird, sind aber noch unvollständig. So wurden noch nicht die Superklassenkontexte aufgelöst und die Originalnamen (die auch als *kanonische* Namen bezeichnet werden) der dort aufgeführten Klassen eingefügt. Diese werden in dieser Phase aber auch nicht benötigt – in dieser Phase wird sogar erst *überprüft*, ob die Superklassenkontexte korrekt sind.

Es werden die folgenden Checks durchgeführt:

- In Klassendeklarationen wird überprüft, ob die Klassen im Superklassenkontext sichtbar und eindeutig sind. Auch für Klassen gilt hier wieder die Semantik, dass lokale Klassendeklarationen importierte Klassen überschatten.
- Derselbe Test wird für die Klassen im Kontext einer *Instanz* und für die eigentliche *Instanz-Klasse* durchgeführt.
- Auch für die in `deriving`-Klauseln angegebenen Klassen wird überprüft, dass diese sichtbar und eindeutig sind. Außerdem wird geprüft, ob für die angegebenen Klassen überhaupt Instanzen abgeleitet werden können. Für eine Klasse kann nur dann eine Instanz abgeleitet werden, wenn ihr Name in der Menge {`Show`, `Eq`, `Ord`, `Enum`, `Bounded`} enthalten ist, und sie aus der Prelude stammt. Wenn die `Enum`-Klasse angegeben wurde, so muss außerdem der Datentyp eine Aufzählung sein. Wenn die `Bounded`-Klasse angegeben wurde, muss der Datentyp entweder eine Aufzählung sein oder aus genau einem Konstruktor bestehen.
- Die Klassen in den Kontexten von Typsignaturen werden ebenfalls überprüft.
- Es wird überprüft, ob die in Klassen oder Instanzen angegebenen Implementierungen tatsächlich Implementierungen von Klassenmethoden sind. Die folgenden Codeausschnitte sind zum Beispiel nicht erlaubt:

```
class C a where
  funC :: a -> a
  funD x = x
```

oder

```
class C a where
  funC :: a -> a

instance C Bool where
  funD x = x
```

Hierbei wird die Information benötigt, welche Methoden eine Klasse deklariert. Diese Information wird bereits für Klassen in der noch unvollständigen Klassenumgebung gespeichert.

Checks der dritten Phase In dieser Phase werden „globale“ Überprüfungen durchgeführt, also Überprüfungen, für die die gesamte Klassenhierarchie benötigt wird.

Für jede `C`-`T`-Instanzdefinition wird überprüft, ob für alle Superklassen `C'` von `C` (genau) eine `C'`-`T`-Instanzdefinition existiert. Wäre dies nicht der Fall, so könnte entweder bei der Wörterbuchkonstruktion für die `C`-`T`-Instanz für bestimmte Superklassen kein Wörterbuch angegeben werden, oder es würden mehrere Wörterbücher zur Verfügung stehen, zwischen denen keine Auswahl getroffen werden kann.

Eine weitere, subtilere Bedingung für korrekte Programme ist, dass der Kontext einer `C`-`T`-Instanzdeklaration alle Kontexte der `C'`-`T`-Instanzen mit `C'` Superklasse von `C` *impliziert*. So ist der folgende Code nicht gültig:

```
class C a where ...
class C a => D a where ...

data T a = T a

instance C a => C (T a) where ...

instance D (T a) where ...
```

da der Kontext {} der `D`-`T`-Instanzdefinition nicht den Kontext {`C a`} der `C`-`T`-Instanzdefinition impliziert. Der Code wird gültig, indem zum Beispiel die `D`-`T`-Instanzdefinition durch „`instance D a => D (T a) where ...`“ ersetzt wird, denn der Kontext {`D a`} impliziert den Kontext {`C a`}.

Diese Einschränkung hat wieder ihren Grund in der Wörterbuchkonstruktion. Im Wörterbuch für eine `C`-`T`-Instanz sind alle Wörterbücher der `C'`-`T`-Instanzen mit `C'` Superklasse von `C` enthalten (direkt oder indirekt). Diese Wörterbücher benötigen selber wieder Wörterbücher für ihre Konstruktion gemäß den entsprechenden Instanzkontexten. Diese Wörterbücher können aber nur zur Verfügung gestellt werden, wenn aus den Wörterbüchern, die der `C`-`T`-Instanz übergeben werden, Wörterbücher für die `C'`-`T`-Instanzen

gebildet werden können. Dies ist äquivalent dazu, dass der Instanzkontext der `C-T`-Instanzdeklaration die Instanzkontexte der `C'-T`-Instanzdeklarationen impliziert.

In der dritten Phase wird auch die Überprüfung durchgeführt, dass in der Klassenhierarchie keine *Zyklen* auftreten. Der folgende Code ist zum Beispiel nicht erlaubt:

```
class E a => C a where ...
class C a => D a where ...
class D a => E a where ...
```

da der Zyklus `C` → `E` → `D` → `C` in der Klassenhierarchie gebildet wird. Um Zyklen in der Klassenhierarchie zu ermitteln, werden für den Graphen der Klassenhierarchie die *starken Zusammenhangskomponenten* ermittelt. Die Klassenhierarchie enthält genau dann keine Zyklen, wenn alle starken Zusammenhangskomponenten nur *genau ein* Element besitzen. Direkte Zyklen, wie in „`class A a => A a`“ werden dabei aber nicht ermittelt, und müssen nochmal explizit abgefangen werden (dies wird bereits in der ersten Phase durchgeführt, siehe Seite 95).

Als letzte Überprüfung wird in dieser Phase überprüft, dass für eine Klasse `C` und einen Typ `T` nicht mehr als eine Instanzdefinition lokal angegeben wurde. Diese Überprüfung kann erst hier durchgeführt werden, da vorher die kanonischen Namen der an lokalen Instanzdefinitionen beteiligten Klassen und Typen nicht bekannt waren, und im Code die Klassen und Typen eventuell unter verschiedenen Namen angesprochen werden.

Schritt 3: Transformation der Klassen und Instanzen Die Transformation der Klassen und Instanzen wird wie in Abschnitt 2.4.2 erläutert durchgeführt. Hierbei besteht wieder die Problematik, dass für vorkommende Funktionsparameter eindeutige Namen generiert werden müssen. Dies wird hier dadurch gelöst, dass die Parameter einer neu erstellen Funktion zusätzlich zum Parameternamen den Namen der Funktion als *Präfix* erhalten. Da alle Funktionsnamen modulweit eindeutig sind, sind somit auch die Namen der Parameter modulweit eindeutig.

Wie bei der Ableitung von Instanzen für Datentypen tritt hier wieder das Problem auf, dass Prelude-Funktionen in den Transformationsergebnissen verwendet werden, hier zum Beispiel der „`.`“-Operator, der in den generierten Selektionsfunktionen für nicht-direkte Superklassen verwendet wird.

Deshalb wird hier der gleiche Ansatz wie bei den abgeleiteten Instanzen verfolgt: Alle benötigten Prelude-Funktionen werden unter einem Dummy-Modulnamen in der Werteumgebung eingetragen, und in den Transformationsergebnissen werden die mit dem Namen des Dummy-Moduls qualifizierten Funktionsnamen verwendet.

Die tatsächlich durchgeführten Instanz- und Klassentransformationen weichen im folgenden Detail von den in Abschnitt 2.4.2 beschriebenen Transformationen ab: Es werden in den generierten Namen von Selektionsfunktionen, Wörterbüchern etc. immer die *qualifizierten* Klassen- bzw. Typnamen verwendet. So folgen zum Beispiel die Namen von Selektionsfunktionen für Superklassen nicht mehr dem Schema `sel.C.C'`, sondern dem Schema `sel.Module1.C.Module2.C'`; das Wörterbuch für eine Klasse `C` im Modul „`Module`“ lautet nun nicht mehr `dict.C` sondern `dict.Module.C`. Diese Erweiterung ist für die Unterstützung mehrerer Module notwendig, da durchaus in verschiedenen

Modulen Klassen mit demselben Namen deklariert werden, und somit die generierten Funktionen und Typen ohne diese Erweiterung mehrdeutig sein können. In Abschnitt 2.4.2 wurden die qualifizierten Namen der Übersichtlichkeit halber nicht verwendet.

4.3.2.7. Type Check

Der Typcheck benutzt zum Ermitteln der Typen und Constraints von Ausdrücken den erweiterten Algorithmus $\mathcal{W}$, der in Abschnitt 3.2.3 beschrieben wurde. Dazu wird er so erweitert, dass anstatt lediglich der Typen auch Kontexte von den Typcheck-Funktionen zurückgegeben werden (also insgesamt *Constrained Types*).

Fixpunktiteration für Deklarationsgruppen Die Behandlung von *Deklarationsgruppen* muss in folgender Beziehung angepasst werden: Möglicherweise reicht ein einzelnes Absteigen in den Syntaxbaum zum Ermitteln der Typen der Funktionen einer Deklarationsgruppe nicht aus. Dazu folgendes Beispiel:

```
class C a where
  funC :: a -> a

class D a where
  funD :: a -> a

fun1 x = fun2 (funC x)
fun2 x = fun1 (funD x)
```

Die Funktionen `fun1` und `fun2` befinden sich in einer Deklarationsgruppe. Da in `fun1` `funC` bzw. in `fun2` `funD` benutzt wird, lauten die ermittelten Typen von `fun1` und `fun2` „`C a => a -> a`“ bzw. „`D a => a -> a`“. Plötzlich besitzen also `fun1` und `fun2` einen Kontext, der noch nicht berücksichtigt wurde! Deshalb müssen die Constrained Types noch ein zweites Mal ermittelt werden, diesmal mit den neuen Signaturen von `fun1` und `fun2`. Als Ergebnis erhält man dadurch jetzt den Typ „`(C a, D a) => a -> a`“ sowohl für `fun1` als auch für `fun2`. Eine dritte Ermittlung der Typen verändert den Typ von `fun1` und `fun2` nicht mehr.

Für die Ermittlung der korrekten Kontexte der Funktionen einer Deklarationsgruppe muss also eine *Fixpunktiteration* durchgeführt werden. Die ermittelten Kontexte werden solange aktualisiert, bis sich diese nicht mehr ändern. In Abb. 4.7 ist dazu ein weiteres Beispiel aufgeführt.

Abspeichern der Typen von Variablen und Funktionen im AST Eine weitere Erweiterung des Typchecks ist, dass für alle Variablen oder Funktionen im abstrakten Syntaxbaum deren konkrete Typen gespeichert werden. Diese Typen werden für die Wörterbucherzeugung benötigt, wie in Abschnitt 2.4.3 beschrieben wurde. Für die Speicherung werden im Syntaxbaum spezielle Felder eingeführt, in denen die Typen gespeichert werden. Da die während des Typchecks eingefügten Typen möglicherweise noch freie Variablen enthalten, die noch nicht gebunden wurden, muss nach dem Ende des

```
class C a where
  funC :: a -> a

fun1 x = fun2 (funC x)

fun2 x = fun3 x

fun3 x = fun4 x

fun4 x = fun1 x
```

Schritt	fun1	fun2	fun3	fun4
Schritt 1	{C a}	∅	∅	∅
Schritt 2	{C a}	∅	∅	{C a}
Schritt 3	{C a}	∅	{C a}	{C a}
Schritt 4	{C a}	{C a}	{C a}	{C a}
Schritt 5	{C a}	{C a}	{C a}	{C a}

Abbildung 4.7: Beispiel für die Fixpunktiteration (angegeben sind die Kontexte der jeweiligen Funktionen)

eigentlichen Typchecks noch ein Lauf über den AST durchgeführt werden, bei dem auf die vorläufigen Typen die endgültige vom Typcheck ermittelte Substitution angewandt wird.

Da die Reduktion der Kontexte der Funktionen einer Deklarationsgruppe *am Schluss* der Behandlung der Deklarationsgruppe stattfindet, enthalten die Typannotationen im AST an den Stellen, an denen Funktionen der Deklarationsgrupe benutzt werden, noch die *unreduzierten* Kontexte. Deshalb müssen am Ende der Typüberprüfung auch nochmals alle Typannotationen aktualisiert werden, so dass sie die Kontextreduktion widerspiegeln.

Beispiel 4.11:
Es sei der folgende Code gegeben:

```
class C a where
  funC :: a -> a

class C a => D a where
  funD :: a -> a

fun1 x = funC (fun2 x)
fun2 x = funD (fun1 x)
```

Nach der Fixpunkt-Iteration lauten die Typen von `fun1` und `fun2`:

```
fun1 :: (C a, D a) => a -> a
fun2 :: (C a, D a) => a -> a
```

und in den Ausdrücken auf den rechten Funktionsseiten lauten die Typen der auftretenden Funktionen wie folgt:

- Erste Gleichung:

```
funC :: C a => a -> a
fun2 :: (C a, D a) => a -> a
```

- Zweite Gleichung:

```
funD :: D a => a -> a
fun1 :: (C a, D a) => a -> a
```

Nach der Kontextreduktion lauten die Typen von `fun1` und `fun2` folgendermaßen:

```
fun1 :: D a => a -> a
fun2 :: D a => a -> a
```

Im abstrakten Syntaxbaum stehen auf den rechten Seiten als Typen von `fun1` und `fun2` aber noch die unreduzierten Typen `(C a, D a) => a -> a`; deshalb müssen diese nochmals aktualisiert werden, um die Kontextreduktion zu berücksichtigen; danach stehen die korrekten Typen `D a => a -> a` im Syntaxbaum.

Das folgende Beispiel demonstriert nochmals, dass die *konkreten* Typen von Funktionen im Code von den allgemeinen Typen der Funktionen abweichen können, weshalb die Typen im AST gespeichert werden müssen.

Beispiel 4.12 (Konkrete Typen im AST)**:**
Es sei der folgende Code gegeben:

```
class C a where
  funC :: a -> Bool

instance C Bool where ...
instance C Int where ...

fun1   = funC True
fun2   = funC 5
fun3 x = funC x
```

Dann lauten die für „`funC`“ ermittelten und in den Syntaxbaum eingefügten Typen auf den rechten Seiten von `fun1`, `fun2` und `fun3` jeweils wie folgt:

- In `fun1`: `C Bool => Bool -> Bool`
- In `fun2`: `C Int => Int -> Bool`
- In `fun3`: `C a => a -> Bool`

Abspeichern der Deklarationsgruppen Um zu vermeiden, dass eine Deklarationsgruppe während der Fixpunktiteration mehrfach geprüft wird, werden die Ergebnisse der ersten Prüfung einer Deklarationsgruppe wie der modifizierte Syntaxbaum *gespeichert*. Dazu werden alle Funktions- und Pattern-Deklarationen durchnummeriert, so dass mit

der Menge der Nummern der Funktions- und Pattern-Deklarationen einer Deklarationsgruppe ein eindeutiger Schlüssel für die Abspeicherung der Prüfungsergebnisse vorhanden ist. Soll eine Deklarationsgruppe ein zweites Mal geprüft werden, so werden die gespeicherten Ergebnisse verwendet.

Gespeichert werden für die Deklarationsgruppen der modifizierte Syntaxbaum, also der Syntaxbaum erweitert um die Typen der vorkommenden Variablen und Funktionen, und der Kontext, der aus der Deklarationsgruppe *herauspropagiert* wird. Dieser Kontext wird zum Beispiel bei lokalen Deklarationsgruppen in `let`-Konstrukten benötigt, denn deren Kontexte werden unter bestimmten Bedingungen in den Gesamtkontext der Funktion aufgenommen, in der der `let`-Ausdruck verwendet wird (siehe nächsten Unterabschnitt).

Beispiel 4.13:
Angenommen, die Deklarationsgruppe

```
fun1 x = fun2 x
fun2 x = fun3 x
fun3 x = fun1 x
```

sei gegeben, und es seien die drei Funktionsgleichungen mit den Nummern n, m und k nummeriert. Dann wird nach der ersten Prüfung der Deklarationsgruppe unter der Menge $\{n, m, k\}$ der modifizierte Syntaxbaum gespeichert, zusammen mit dem zu propagierenden Kontext, der hier leer ist.

Unter anderem auch aufgrund der `let`-Konstrukte wurde ein Abspeichern der Ergebnisse für die Deklarationsgruppen eingeführt: Bei jedem Prüfen einer Deklarationsgruppe, die einen `let`-Ausdruck oder eine `where`-Klausel enthält, würden bei der Fixpunktiteration die Deklarationsgruppen in dem `let`-Ausdruck oder der `where`-Klausel in jeder Iteration erneut geprüft werden.

Lokale und nicht-lokale Kontextelemente Wenn in `let`-Ausdrücken *nicht-lokale* Variablen verwendet werden, also Variablen, die außerhalb des `let`-Ausdrucks definiert sind, so kann es passieren, dass auch Kontextelemente sich auf Typvariablen „von weiter oben“ beziehen. Wie solche Kontextelemente gehandhabt werden, wird anhand des folgenden Beispiels erläutert.

Im folgenden Beispiel ist die Variable `x`, die in `fun2` verwendet wird, in der Definition von `fun2` nicht-lokal, da `x` ein Parameter der Funktion `fun1` ist:

```
class C a where
  funC :: a -> a

class D a where ...

class E a where ...

instance (D a, E b) => C (a, b) where ...
```

```
instance E Bool where ...

fun1 x = fun2 True
  where
  fun2 z = funC (x, z)
```

Nach der Typüberprüfung lauten die Typen in der Funktion `fun1` wie folgt:

```
(fun1 :: D a => a -> (a, Bool)) (x :: a) =
    (fun2 :: (D a, E Bool) => Bool -> (a, Bool)) True
  where
  (fun2 :: (D a, E b) => b -> (a, b)) (z :: b) =
    (funC :: C (a, b) => (a, b) -> (a, b))
      (x :: a, z :: b)
```

Hier ist die Typvariable `a` im Typ der Funktion `fun2` ebenfalls nicht lokal, da sie sich auf den Typ des Parameters `x` der Funktion `fun1` bezieht (Typvariablen sind formal dann nicht lokal, wenn sie an der entsprechenden Stelle in der Menge der freien Variablen der Typumgebung vorkommen).

Das Kontextelement `C (a, b)` im *unreduzierten* Kontext der Funktion `fun2` enthält nun sowohl lokale als auch nicht-lokale Typvariablen. Mit dem aus diesem Element bestehenden Kontext wird nun folgendermaßen verfahren: Er wird zuerst *reduziert*, was zum Kontext `(D a, E b)` führt. Dann werden alle *nicht-lokalen* Kontextelemente nach oben propagiert, hier also das Kontextelement `D a` (ein Kontextelement der Form `K a` ist genau dann nicht lokal, wenn `a` nicht lokal ist). Deshalb steht im Kontext von `fun1` das Kontextelement `D a`, aber nicht das Element `E b`.

Im Allgemeinen wird mit den Kontexten von Funktionen, die Teil einer Deklarationsgruppe in einem `let`-Ausdruck sind, folgendermaßen umgegangen:

- Die Kontexte werden reduziert.
- Alle nicht-lokalen Kontextelemente des reduzierten Kontexts werden nach oben propagiert, die lokalen Kontextelemente werden verworfen.

Der Sinn dieser Aufteilung ist, dass Wörterbücher für lokale Kontextelemente auch lokal bezogen werden, während Wörterbücher für nicht-lokale Kontextelemente von „weiter oben“ bezogen werden. Dadurch lassen sich in Funktionen, die nicht-lokale Variablen benutzen, Wörterbuchparameter sparen. Diese Code-Optimierung wird in Abschnitt 4.3.2.8 genauer erläutert.

Kontextpropagation für unbenutzte Elemente in `let`-Ausdrücken Gemäß des Algorithmus $\mathcal{W}$ wird in einem `let`-Ausdruck der Form `let x = e1 in e2` der für `x` ermittelte Kontext nur dann propagiert, wenn `x` in e_2 verwendet wird. So würde laut dem Algorithmus $\mathcal{W}$ der Typ der Funktion `fun1` im folgenden Beispiel „`C a => a -> (a, Bool)`“ lauten, und nicht „`(C a, D a) => a -> (a, Bool)`“, da nur `fun2` benutzt wird und diese Funktion den Kontext `(C a)` hat:

```
class C a where
  funC :: a -> a

class D a where
  funD :: a -> a

fun1 x = fun2 True
  where
  fun2 y = (funC x, y)
  fun3 y = (funD x, y)
```

`funC` und `funD` beziehen ihre Wörterbücher aber von weiter oben, denn `x` ist nicht-lokal in `fun2` und `fun3`. Würde der Kontext von `fun1` nur `C a` lauten, so würde bei der Wörterbuchtransformation bei der Funktion `fun3` ein Fehler auftreten, da das Wörterbuch für `D a` von weiter oben nicht erhalten werden kann. Deshalb müssen der Funktion `fun1` sowohl das Wörterbuch für `C a` als auch das Wörterbuch für `D a` übergeben werden, der Kontext muss also `(C a, D a)` lauten. Also müssen auch aus allen *unbenutzten* Deklarationen in einem `let`-Ausdruck nicht-lokale Kontextelemente nach oben propagiert werden.

Das Ergebnis nach der Wörterbuchtransformation lautet für das obige Beispiel:

```
fun1 dict.C.a dict.D.a = fun2 True
  where fun2 y = (sel.C.funC dict.C.a x, y)
        fun3 y = (sel.D.funD dict.D.a x, y)
```

womit nochmals deutlich wird, warum das Wörterbuch für `D a` tatsächlich benötigt wird und der Kontext von `fun1 (C a, D a)` lauten muss.

Klassenmethoden Vor dem Beginn der Typüberprüfung werden alle lokal definierte und importierte Klassenmethoden in der Werteumgebung eingetragen. Die Klassenmethoden sind dabei mit den anderen in der Werteumgebung enthaltenen Elementen gleichberechtigt. Das heißt, dass weder Klassenmethoden noch normale Funktionen bei Mehrdeutigkeiten bevorzugt werden, sondern nur danach gegangen wird, ob eine Klassenmethode oder eine einfache Funktion *lokal* definiert wurde.

Alle Klassenmethoden werden in der Werteumgebung dadurch identifiziert, dass für sie zusätzlich noch die Klasse angegeben ist, in der sie definiert wurden.

Generalisierung Im Generalisierungsschritt wird, wie in Abschnitt 3.2.3 erläutert, die Generalisierung des vom Algorithmus $\mathcal{W}$ ermittelten Constrained Types durchgeführt. Lautet der für die Typannahme A und eine Funktion `f` ermittelte Constrained Type $(P \mid \tau)$, und ist S die ermittelte Substitution, so lautet das Typschema von `f` $\mathrm{Gen}(SA, P \Rightarrow \tau)$. Zusätzlich wird bei der Generalisierung auch die *Kontextreduktion* der Funktionskontexte durchgeführt. Außerdem wird geprüft, ob die reduzierten Kontexte *gültig* sind: Alle Kontextelemente müssen vollständig auf Kopfnormalform reduziert werden können, und es dürfen keine mehrdeutige (*ambiguous*) Constraints vorkommen. Mehrdeutige Constraints im Kontext `K` einer Typsignatur `K => `τ sind solche

Constraints, die eine Typvariable einschränken, welche nicht in τ und auch nicht frei in der Werteumgebung vorkommt [HB90]. So ist das Kontextelement „`C a`“ in „`C a => b -> b`“ mehrdeutig, wenn die Typvariable `a` nicht frei in der Werteumgebung vorkommt.

Ist eine Typsignatur für eine Funktion vorhanden, so wird außerdem geprüft, dass der Kontext in der Typsignatur den inferierten Kontext *impliziert*.

Reihenfolge der Wörterbücher beim Zusammensetzen von Wörterbüchern Bei der Wörterbuchcodegenerierung werden beim Zusammensetzen eines Wörterbuchs aus anderen Wörterbüchern die Wörterbücher in der Reihenfolge übergeben, die durch die Kontextelemente im reduzierten Kontext der entsprechenden Instanz angegeben wird (siehe Abschnitt 2.4.3). Damit überall diese Reihenfolge berücksichtigt wird, müssen bestimmte Vorkehrungen getroffen werden.

Betrachten wir dazu das folgende Beispiel:

```
class C a where
  funC :: a -> a

class D a where

data T a b = ...

instance (C a, D b) => C (T a b) where ...

fun :: (C a, D b) => a -> b -> T a b
fun x y = funC (T x y)
```

Da die Funktion `funC` in der Funktion `fun` den Typ `C (T a b) => T a b -> T a b` hat, muss das Wörterbuch für die `C-T`-Instanz eingefügt werden. Dieses wird, wenn der reduzierte Kontext der `C-T`-Instanz `(C a, D b)` lautet, gemäß dieses Kontexts folgendermaßen zusammengesetzt: `dict.C.T dict.C.a dict.D.b`. Der Code für `fun` lautet also nach der Wörterbuchtransformation folgendermaßen:

```
fun dict.C.a dict.D.b x y =
  sel.C.funC (dict.C.T dict.C.a dict.D.b) (T x y)
```

Für die `C-T`-Instanz wird unter anderem der folgende Code generiert, in dem das Wörterbuch für diese Instanz definiert wird:

```
dict.C.T :: (C a, D b) => Dict.C (T a b)
dict.C.T = impl.C.T.funC
```

Nun könnte es passieren, dass bei der Typermittlung für `dict.C.T` der Kontext `(D b, C a)` ermittelt wird, und nicht der Kontext `(C a, D b)`. Da der erstere den letzteren Kontext impliziert, ist auch dieser Kontext gültig. Nun würde die Wörterbuchfunktion aber die Wörterbücher in einer anderen Reihenfolge erwarten, als in der sie tatsächlich bei der Wörterbuchcodegenerierung übergeben werden! Dies würde hier zu einem Typfehler führen; hätte die obige Instanz hingegen den Kontext `(C a, C b)`, so würde kein Typfehler auftreten, sondern stillschweigend die Programmsemantik verletzt werden.

Damit ein solcher Fall nicht eintreten kann, werden die folgenden zwei Vorkehrungen getroffen:

- Bei der Typüberprüfung werden die Kontexte der Typsignaturen, falls vorhanden, als *Grundlage* genommen. Alle weiteren ermittelten Kontextelemente werden an den vorhandenen Kontext *angehängt*.
- Bei der Kontextreduktion werden die Elemente von *hinten nach vorne* entfernt.

Die folgende Überlegung zeigt, dass damit das Problem gelöst ist:

Angenommen, der Kontext einer Instanz I lautet K, und die Kontextreduktion sei mit red bezeichnet. Dann wird bei der Wörterbuchcodegenerierung die Reihenfolge der Kontextelemente in $\mathrm{red}(K)$ als Grundlage für die Reihenfolge der Wörterbücher genommen, aus denen das Wörterbuch für die Instanz I zusammengesetzt wird.

Der Kontext K wird in die Typsignatur der Wörterbuchfunktion von I übernommen. Bei der Typermittlung seien nun für die Wörterbuchfunktion zusätzlich die Kontextelemente π_1 bis π_n ermittelt worden. Diese werden an K angehängt, so dass der inferierte Kontext nun $K, \pi_1, \ldots, \pi_n$ lautet. Anschließend wird die Kontextreduktion des Kontexts durchgeführt: Da der Kontext K alle π_i impliziert (ansonsten wäre die Typsignatur inkorrekt), und bei der Kontextreduktion von hinten nach vorne vorgegangen wird, wird im jeden Schritt immer das letzte π_i entfernt, bis schließlich wieder der Kontext $\mathrm{red}(K)$ dasteht: $\mathrm{red}(K, \pi_1, \ldots, \pi_n) = \mathrm{red}(K, \pi_1, \ldots, \pi_{n-1}) = \cdots = \mathrm{red}(K, \pi_1) = \mathrm{red}(K)$. Dieser Kontext entspricht genau dem reduzierten Instanzkontext; die Wörterbuchfunktion erwartet also die Wörterbücher genau in der Reihenfolge, in der sie bei der Wörterbuchcodegenerierung übergeben werden.

Bei der oben angegebenen Funktion `fun` wird zum Beispiel der Kontext `(C a, D b)` als Grundlage genommen, und bei der Typinferenz das Kontextelement `C (T a b)` ermittelt, das an den Kontext angehängt wird, so dass dieser nun `(C a, D b, C (T a b))` lautet. Bei der Kontextreduktion wird das Element `C (T a b)` wieder entfernt, und der reduzierte Kontext lautet wie gewünscht wieder `(C a, D b)`.

Damit kann der Kontextreduktionsalgorithmus, der in Abb. 2.10 angegeben wurde, durch die Angabe des auszuwählenden Elements π komplettiert werden: Werden Kontexte als Listen betrachtet, so muss immer das *letztmögliche* Kontextelement gewählt werden, das die Schleifenbedingung erfüllt.

4.3.2.8. Wörterbuch-Transformation

Die Wörterbuchtransformation wird im Wesentlichen nach der in Abschnitt 2.4.3 angegebenen Transformation durchgeführt. Es wird aber zusätzlich eine Optimierung bei `let`-Konstrukten durchgeführt.

Optimierung von `let`-Konstrukten Funktionen, die in `let`-Ausdrücken definiert sind, können selber wieder überladen sein und Wörterbücher benötigen. In gewissen Situationen kann der Code, der durch die Transformation aus Abschnitt 2.4.3 erzeugt wird,

vereinfacht werden, indem Wörterbuchparameter wiederverwendet werden; diese Optimierung kann dann angewandt werden, wenn die in einer `let`-Deklaration definierten Funktionen *nicht-lokale* Variablen benutzen.

Beispiel 4.14:
Die Funktion `fun1` im Code

```
class C a where
  funC :: a -> a

class D a where
  funD :: a -> a

instance D Bool where ...

fun1 x = fun2 True
  where
  fun2 y = (funC x, funD y)
```

wird durch die Transformation aus Abschnitt 2.4.3 in folgenden Code übersetzt:

```
fun1 dict.C.a x = fun2 dict.C.a dict.D.Bool True
  where
  fun2 dict.C.a dict.D.b y =
    (sel.C.funC dict.C.a x, sel.D.funD dict.D.b y)
```

Dieser Code kann aber, da der Funktion `fun2` sowieso immer das Wörterbuch `dict.C.a` übergeben wird, zu folgendem Code vereinfacht werden:

```
fun1 dict.C.a x = fun2 dict.D.Bool True
  where
  fun2 dict.D.b y =
    (sel.C.funC dict.C.a x, sel.D.funD dict.D.b y)
```

Es kann also ein Wörterbuchparameter bei der Funktion `fun2` eingespart werden.

Der Wörterbuchparameter `dict.D.b` kann nicht weggelassen werden, da die Funktion `fun2` im Rumpf von `fun1` auf Werte verschiedenen Typs angewendet werden kann und dementsprechend auch verschiedene Wörterbücher übergeben werden müssen.

Welche Kontextparameter eingespart werden können, wird folgendermaßen ermittelt:

Wie schon in Abschnitt 4.2.2.6 erläutert wurde, werden Typvariablen für Parameter einer Funktion nach der Generalisierung des Funktionstyps als nicht-negative Zahlen dargestellt. Typvariablen für nicht-lokale Variablen bleiben hingegen negativ.

So lautet der generalisierte Typ von `fun2` aus obigen Beispiel `(C -1, D 0) => 0 -> (-1, 0)`, wobei jetzt explizit Nummern anstatt von Typvariablen angegeben werden.

Wann immer also in dem generalisierten Typ einer Funktion negative Typvariablen vorkommen, beziehen sich diese auf nicht-lokale Variablen. Ähnlich verhält es sich mit

den Kontextelementen eines generalisierten Typs: Wenn ein Kontextelement als Typvariable eine negative Zahl hat, so bedeutet dies, dass das entsprechende Wörterbuch von weiter außen bezogen werden kann.

Deshalb wird bei der Behandlung einer Funktion `f` der generalisierte Kontext von `f` aus der Werteumgebung herangezogen, um zu bestimmen, welche Wörterbuchparameter im Code weggelassen werden können. Bei allen Verwendungen von `f` können Wörterbuchparameter an denjenigen Stellen weggelassen werden, bei denen der generalisierte Kontext eine negative Typvariable besitzt, da dies andeutet, dass ein entsprechendes Wörterbuch von weiter außen bezogen werden kann.

Im obigen Beispiel kann also, da der generalisierte Typ von `fun2` „`(C -1, D 0) => 0 -> (-1, 0)`“ lautet, bei allen Verwendungen von `fun2` immer jeweils der Wörterbuchparameter für das erste Kontextelement weggelassen werden.

Kontextelemente der Form `C n` mit `n` negativ werden nie instanziiert, es wird also die Typvariable nie durch einen anderen Typ ersetzt. Somit werden diese Kontextelemente immer unverändert mitgeführt, auch bei Applikationen. Daher steht an einer Stelle, an der im generalisierten Typkontext eine negative Typvariable auftritt, auch im konkreten Typkontext immer dieselbe negative Typvariable, und kein Typkonstruktor. Somit entsprechen die weggelassenen Wörterbuchparameter immer demselben Wörterbuch, und nur deshalb dürfen die Wörterbuchparameter überhaupt weggelassen werden.

Probleme bei Pattern-Deklarationen Da bei Pattern-Deklarationen auf der linken Seite immer nur Variablen stehen, die keine zusätzlichen Parameter entgegennehmen, wie es bei Funktionen der Fall ist, beziehen sich Variablen auf der rechten Seite von Pattern-Deklarationen *immer* auf Variablen von „weiter oben“ oder auf lokal definierte Variablen.

Im ersten Fall treten keine Probleme auf; die benutzten Variablen sind nicht-lokal und daher werden die entsprechenden Kontexte nach oben propagiert, so dass von weiter oben die entsprechenden Wörterbücher bereitgestellt werden. Dies ist selbst dann der Fall, wenn die Variablen der Pattern-Deklaration überhaupt nicht benutzt werden, wie in

```
class C a where
  funC :: a -> a

fun x = x
  where
  Just y = Just (funC x)
```

Hier stellt die Funktion `fun` das Wörterbuch für die Klassenmethode `funC` bereit.

Im zweiten Fall sind die benutzten Variablen lokal, es werden also keine Kontexte nach oben propagiert. Dadurch ergibt sich ein Problem, wenn die in der Pattern-Deklaration definierten Variablen *nicht benutzt werden*, wie im folgenden Beispiel[1]:

```
class C a where
  funC :: a -> a
```

[1] In Haskell ist dieser Code nur dann gültig, wenn die Monomorphie-Restriktion ausgeschaltet wird.

```
fun x = x -- y wird nicht benutzt!
  where
  Just y = Just (funC y) -- y wird lokal verwendet
```

`y` wird in der Definition von `y` lokal verwendet, daher wird kein Kontext `(C a)` nach oben propagiert. Da `y` in der Pattern-Deklaration „gebunden“ ist, können für `y` auch keine zusätzlichen Parameter eingeführt werden, in denen das Wörterbuch für `C` übergeben werden könnte. Das Wörterbuch für `C`, das wegen der Methode „`funC`“ benötigt wird, ist also nicht verfügbar, was zurzeit zu einem internen Fehler führt.

Dieses Problem tritt nur auf, wenn `y` nicht weiter oben verwendet wird. Im folgenden Beispiel tritt das Problem nicht auf:

```
fun x = y
  where
  Just y = Just (funC y)
```

Da `y` verwendet wird, wird automatisch auch der Kontext von `y` bei der Typüberprüfung von `fun` mit einbezogen, so dass die Funktion `fun` nun den Kontext `(C a)` hat, und somit das Wörterbuch für `y` bereitstellen kann.

Da der interne Fehler nur dann auftritt, wenn eine rekursive Pattern-Deklaration mit Klassenmethoden vorliegt, *und* die entsprechenden Variablen nicht benutzt werden, wurde der Fehler bisher noch nicht behoben, da eine solche Situation wahrscheinlich nur sehr selten vorkommt. Der interne Fehler kann immer dadurch vermieden werden, dass die nichtbenutzte Deklaration entfernt wird, was die Programmsemantik nicht verändert.

Syntaktischer Zucker Bei der Wörterbuchtransformation muss teilweise auch syntaktischer Zucker aufgelöst werden, und zwar schon vor der eigentlichen Desugar-Phase. Dies ist deshalb der Fall, da unter Umständen bei bestimmten Konstrukten Wörterbücher eingefügt werden müssen.

Betroffen sind die folgenden Konstrukte:

Infixapplikationen, Left- und Right-Sections Die Wörterbücher, die eventuell eingefügt werden müssen, können nicht in die ursprünglichen Konstrukte, durch die diese Codeelemente im Syntaxbaum repräsentiert werden, eingebaut werden. Deshalb müssen die Codeelemente in geschachtelte Applikationen umgebaut werden, wenn Wörterbücher eingefügt werden sollen. So wird der Code “`True `funC` False`“ in den Code „`sel.C.funC dict.C.Bool True False`“ umgewandelt, wenn `funC` eine Klassenmethode des Typs `a -> a -> Bool` ist, und in der Klasse `C` deklariert wurde.

Enumerationen der Form `[a ..]`, `[a, b ..]`, `[a .. b]` und `[a, b .. c]` Durch die Verwendung von Typklassen können diese Konstrukte auf beliebige Enumerationstypen erweitert werden. Enumerationstypen sind dabei solche Typen, für die eine `Enum`-Instanz angegeben wurde. Vorher waren diese Konstrukte nur für Integer zugelassen, jetzt können aber auch zum Beispiel Character oder andere beliebige Enumerationstypen verwendet werden. So liefert `['a' .. 'c']` die Liste `['a', 'b',`

`'c']`, und die Enumeration `[T1 ..]` liefert für einen Datentyp der Form „`data T = T1 | T2 | T3 | T4`“, der die `Enum`-Klasse implementiert, die Liste `[T1, T2, T3, T4]`.

Eine Enumeration der Form `[a ..]` wird in den Code „`enumFrom a`“ aufgelöst, wobei `enumFrom` nun eine Klassenmethode von `Enum` ist. Da `enumFrom` überladen ist, müssen also an dieser Stelle Wörterbücher eingefügt werden, weshalb die Auflösung des syntaktischen Zuckers bereits hier geschehen muss. Der Code `['a' ..]` wird zum Beispiel in `sel.Enum.enumFrom dict.Enum.Char 'a'` aufgelöst.

Ebenso wird mit den Enumerationen der Form `[a, b ..]`, `[a .. b]` und `[a, b .. c]` verfahren, wobei für diese jeweils die Klassenmethoden `enumFromThen`, `enumFromTo` und `enumFromThenTo` verwendet werden.

In der Typüberprüfung wird diese Erweiterung der Enumerationen dadurch unterstützt, dass der Typ der Elemente in der Enumeration nicht mehr mit dem Typ `Int` sondern mit dem Typ „`Enum a => a`“ unifiziert wird, wobei `a` eine neue Typvariable ist.

Zahlenliterale Integer-Literale n werden durch „`fromInteger` n“ ersetzt, wobei `fromInteger` eine Klassenmethode der Klasse `Num` mit dem Typ `Int -> a` ist, die für Integers und Floats überladen ist. Mit Hilfe dieser Funktion können Integers in beliebige Zahlentypen konvertiert werden. Die Klasse `Num` ist die Basis der Klassenhierarchie für Zahlen und definiert die Operatoren `+`, `-` und `*`, die standardmäßig für Integers und Floats überladen sind (siehe Abb. 4.8). Float-Literale f werden durch „`fromFloat` f“ ersetzt, wobei `fromFloat` eine Klassenmethode der Klasse `Fractional` mit dem Typ `Float -> a` ist. Die Klasse `Fractional` ist eine Subklasse der Klasse `Num` und repräsentiert gebrochene Zahlen. Mit der Methode `fromFloat` können somit Floats in beliebige gebrochene Zahlentypen konvertiert werden. Die Klasse `Fractional` definiert auch die Division auf gebrochenen Zahlen, die für Floats überladen ist.

Mit Hilfe der angeführten Ersetzungen können nun Ausdrücke der Form „`1 + 2.3`“ geschrieben werden, obwohl der Typ „`Int`“ des ersten Literals ungleich dem Typ „`Float`“ des zweiten Literals ist, denn der Ausdruck wird durch „`fromInteger 1 + fromFloat 2.3`“ ersetzt. Der Typ dieses Ausdrucks lautet `Fractional a => a`, was bedeutet, dass das Ergebnis des Ausdrucks in jeden Typ konvertiert werden kann, der die `Fractional` Klasse implementiert. Dies kann durch die Angabe einer Typsignatur geschehen. So kann das Ergebnis des obigen Ausdrucks als Float durch den folgenden Code ermittelt werden: `(1 + 2.3) :: Float`. Dieser Ausdruck wird durch die Wörterbuchtransformation in den folgenden Code transformiert: `sel.Num.(+) dict.Num.Float (sel.Num.fromInteger dict.Num.Float 1) (sel.Fractional.fromFloat dict.Fractional.Float 2.3)`. Beide Summanden haben jetzt den Typ `Float` und können somit addiert werden.

Um die angegebenen Ersetzungen zu ermöglichen, werden beim Typcheck als Typ von Integerliteralen der Typ `Num a => a` und als Typ von Floatliteralen der Typ `Fractional a => a` angenommen.

```
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
```

```
class Num a => Fractional a where
  (/)   :: a -> a -> a
  recip :: a -> a

  recip x = 1/x
  x / y = x * recip y

  fromFloat :: Float -> a
```

Abbildung 4.8: Numerische Basisklassen: Die Klasse `Num` repräsentiert allgemeine Zahlen und die Klasse `Fractional` gebrochene Zahlen

Einstelliges Minus Ein einstelliges Minuszeichen wie in „`-1`“ wird durch die Klassenmethode `negate` der `Num`-Klasse ersetzt.

4.3.2.9. Typsignatur-Transformation

In der Typsignatur-Transformation werden alle im Code vorhandenen Typsignaturen korrigiert. Nach der Wörterbuchtransformation haben sich durch das Einfügen der Wörterbuchparameter alle Signaturen von überladenen Funktionen geändert, und dies muss nun berücksichtigt werden.

Die neuen Typsignaturen werden ermittelt, indem in der Werte-Umgebung die Typen `K =>` τ der jeweiligen Funktionen nachgeschlagen werden, und danach dem Typ τ die Wörterbuchtypen für die Kontextelemente in `K` vorangestellt werden. Der Kontext `K` selber wird entfernt.[2]

Beispiel 4.15:
Gegeben seien die folgenden Klassen:

```
class C a where
  funC :: a -> a

class C a => D a where
  funD :: a -> Bool

class E a where
  funE :: a -> a -> a
```

Die Typsignatur „`(D a, E a) =>` τ“ wird dann zuerst nach „`Dict.D a -> Dict.E a ->` τ“ übersetzt, und danach werden die Typsynonyme „`Dict.D`“ und „`Dict.E`“

[2]Derselbe Ansatz könnte auch verwendet werden, um die Typen überladener Funktionen *in der Werteumgebung* in Typen ohne Kontext zu transformieren; eine solche Transformation könnte dann anstatt des zweiten Typchecks eingebaut werden (siehe dazu auch die Diskussion in Abschnitt 4.3.2).

aufgelöst, da in τ bereits alle Typsynonyme aufgelöst sind, und in der Typsignatur nicht gleichzeitig aufgelöste und nicht aufgelöste Typsynonyme vorkommen dürfen. Das Ergebnis ist also die Typsignatur „`(a -> a, a -> Bool) -> (a -> a -> a) -> `τ“.

Die bei dieser Transformation entstehenden Typsignaturen sind also *expandiert*: Es wurden alle Typsynonyme aufgelöst und die Originalnamen von Typkonstruktoren werden verwendet. Der Typcheck muss also in der ersten Ausführung Typsignaturen behandeln, die nicht expandiert sind, während in der zweiten Ausführung Typsignaturen behandelt werden müssen, die expandiert sind. Damit der Typcheck expandierte von nicht-expandierten Typsignaturen unterscheiden kann, werden Typsignaturen mit einem Flag versehen, das angibt, ob die jeweilige Typsignatur expandiert ist oder nicht. Dann werden im Typcheck gemäß dem Flag die Typsignaturen entweder expandiert oder nicht.

Die Unterscheidung von expandierten und nicht expandierten Typsignaturen ist deshalb notwendig, weil eine bereits expandierte Typsignatur nicht nochmals expandiert werden darf. Ansonsten könnte der folgende Fall eintreten:

Angenommen, es sind die folgenden Module gegeben:

```
module M where
  data T = ...

module M2 where
  import M as N

  f :: ... N.T ...
  f = ...
```

Dann würde bei der ersten Expansion der Typsignatur von `f` „`N.T`“ durch „`M.T`“ ersetzt werden. Bei einer zweiten Expansion der Typsignatur von `f` würde versucht werden, „`M.T`“ aufzulösen, was aber nicht möglich ist und in einem Fehler resultiert.

Typsignaturen in explizit getypten *Ausdrücken* werden einfach dadurch gehandhabt, dass der Kontext aus der Typsignatur gelöscht wird, damit das entstehende Programm frei von Typklassenelementen ist.

4.3.2.10. Export Check

In der Exportspezifikation eines Moduls kann angegeben werden, welche Typklassen und Klassenmethoden exportiert werden sollen. Dabei sind für eine Typklasse `C` mit den Klassenmethoden $\texttt{fun}_1$ bis $\texttt{fun}_n$ die folgenden Exportspezifikationen erlaubt:

`C` Wird nur der Klassenname angegeben, so werden nur dieser und keine Klassenmethoden exportiert.

`C(..)` Bei dieser Angabe werden der Klassenname und *alle* Klassenmethoden exportiert.

`C(fun`$_{k_1}$`, ..., fun`$_{k_m}$`)` Hier muss gelten, dass $\{k_1, \ldots, k_m\} \subseteq [1, n]$ gilt. Es werden der Name der Typklasse sowie die angegebenen Klassenmethoden exportiert.

Es können *nicht* einzelne Klassenmethoden ohne den entsprechenden Klassennamen exportiert werden wie in der Exportspezifikation „`fun`$_i$“. Einzelne *Funktionen* können aber auf diese Weise exportiert werden. Deshalb müssen Klassenmethoden von normalen Funktionen unterschieden werden. Dies geschieht dadurch, dass in der Werteumgebung für Klassenmethoden zusätzlich die Klasse, aus der die Methode stammt, angegeben ist.

Die Syntax der Exportspezifikationen für Klassen und Klassenmethoden entspricht der Syntax der Exportspezifikationen für Datentypen und deren Konstruktoren. Deshalb werden Exportspezifikationen der Form „`C(...)`“ in zwei Stufen vom Export Check geprüft: Zuerst wird ermittelt, ob der Name `C` der Name eines Typkonstruktors oder einer Klasse ist. Ist `C` undefiniert, oder gehört der Name `C` sowohl zu einer Klasse als auch zu einem Typ, so wird eine Fehlermeldung ausgegeben. Dann werden die Angaben innerhalb der Klammern geprüft: Bei Angaben der Form „`C(t`$_1$`, ..., t`$_n$`)`“ wird bei Datentypen überprüft, ob `t`$_1$ bis `t`$_n$ Datenkonstruktoren des Datentyps sind, und wenn `C` ein Klassename ist, wird überprüft, dass `t`$_1$ bis `t`$_n$ Namen von Klassenmethoden der Klasse `C` sind.

Alle drei oben genannte zulässige Formen von Exportspezifikationen für den Export von Klassen und Klassenmethoden werden vom Export Check – wie die Datentyp-Exportangaben auch – in die Form `C(fun`$_{k_1}$`, ..., fun`$_{k_m}$`)` transformiert, es stehen also nach dem Export Check immer alle Klassenmethoden, die exportiert werden sollen, explizit im Code. Die erste mögliche Exportangabe der Form „`C`“ wird also zu „`C()`“ transformiert, die zweite mögliche Exportangabe „`C(..)`“ zu „`C(fun`$_1$`, ..., fun`$_n$`)`“. Die dritte mögliche Exportangabe wird einfach beibehalten. Dies hat unter anderem den Zweck, dass bei dem folgenden Export Klassen- und Datentypexporte von Funktionsexporten unterschieden werden können: Klassen- und Datentypexporte haben immer die Form „`C(...)`“, während Funktionsexporte die Form „`f`“ haben, also ohne Klammern geschrieben werden.

Beispiel 4.16:
Die Exportspezifikation des Moduls

```
module M (C, D(funD2, funD4), E(..), fun) where

class C a where
  funC1 :: a -> a

class D a where
  funD1 :: a -> a
  funD2 :: a -> a
  funD3 :: a -> a
  funD4 :: a -> a
```

```
class E a where
  funE1 :: a -> a
  funE2 :: a -> a

fun :: a -> a
fun = ...
```

wird nach

```
M (C(), D(funD2, funD4), E(funE1, funE2), fun)
```

transformiert.

4.3.2.11. Qualification

Bei der Qualifizierung der Compilerumgebung muss auch die Klassenumgebung berücksichtigt werden. Diese wird analog zu den anderen Umgebungen qualifiziert: Alle Klassen werden unter ihrem Originalnamen in der Klassenumgebung eingetragen.

4.3.3. Anpassung des Modulsystems

4.3.3.1. Übersicht

Der erweiterte Compiler führt *zwei* Typchecks durch, daher müssen auch zwei Interfaces bereitgestellt werden (siehe Abb. 4.9). Beide Interfaces werden in derselben Datei gespeichert. Das eine Interface enthält Funktionstypen *mit* Kontext, das andere Funktionstypen *ohne* Kontext. Ansonsten speichern die beiden Interfaces dieselben Elemente und diese in derselben Form.

Zu Beginn des Kompilationsverlaufs geschieht der erste – vollständige – Import. Importiert werden unter anderem Klassen, Instanzen und Funktionen mit Kontext. Die Klassen und Instanzen werden für den Type Classes Check benötigt. Die Funktionen mit Kontext werden im ersten Typcheck verwendet.

Nachdem alle Typklassenelemente aus dem Code herauskompiliert wurden, wird wie schon erläutert der zweite Typcheck ausgeführt. Dieser erwartet aber in der Werteumgebung Funktionen, deren Typen *keinen* Kontext besitzen, und um die Typen der Wörterbuchparameter erweitert wurden. Daher wird vor dem Ausführen des zweiten Typchecks die Werteumgebung mit den Funktionen aus dem zweiten Interface initialisiert, bei denen die Transformation von Kontexten in Wörterbuchparameter stattgefunden hat.

Wenn das erste Interface zum Beispiel eine Funktion des Typs `C a => a -> a` enthält, so lautet der Typ dieser Funktion im zweiten Interface `Dict.C a -> a -> a`.

Damit die beiden Interfaces jeweils die Funktionstypen mit bzw. ohne Kontext enthalten, müssen die Daten der Funktionen an *zwei* Stellen des Kompilierverlaufs entnommen werden (siehe Abb. 4.9). Das Interface mit Funktionen *mit* Kontext wird nach der Ausführung des ersten Typchecks ermittelt; denn nach dieser Typcheckphase stehen in der Werteumgebung alle überladenen Funktionen mit Kontext. Der Export wird vorbereitet

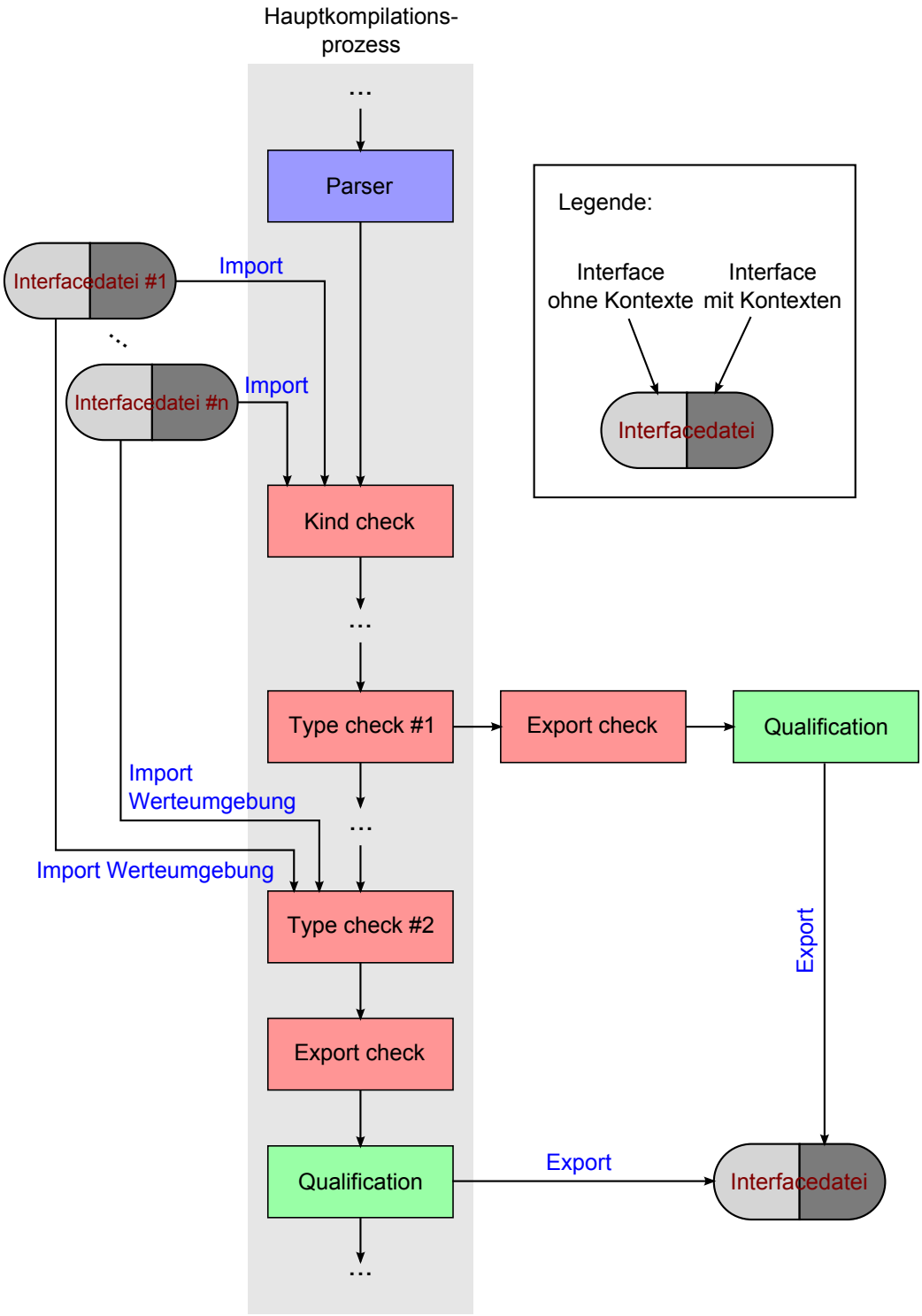

Abbildung 4.9: Import und Export beim angepassten Compiler

durch ein Ausführen des Export-Checks und der Qualifikation außerhalb des Hauptkompilierprozesses, wobei der sich ergebende abstrakte Syntaxbaum verworfen wird, denn die beiden Phasen werden nur wegen des Exports durchgeführt. Somit werden in das entsprechende Interface die Funktionstypen mit Kontext übernommen.

Der zweite Export findet nach dem *zweiten* Typcheck statt. Nach dieser Phase ist der Code frei von Typklassenelementen, und auch in der Werteumgebung gibt es keine Kontexte mehr. Deshalb werden an dieser Stelle auch nur die Funktionstypen ohne Kontexte in das entsprechende Interface übernommen. Auch hier wird der Export durch den Export-Check und die Qualifizierungstransformation vorbereitet.

4.3.3.2. Export von Klassen und Instanzen

Beim Export von Klassen und Instanzen muss berücksichtigt werden, dass implizit auch solche Entitäten versteckt exportiert werden, von denen explizit exportierte Klassen oder Instanzen *abhängen*. Dies sind bei Klassen die Superklassen, und bei Instanzen die verwendeten Klassen und deren Superklassen. Außerdem müssen auch die folgenden bei der Transformation von Klassen und Instanzen generierten Entitäten exportiert werden: die Selektionsfunktionen, Defaultmethoden und Wörterbuchtypen für Klassen, sowie die Wörterbücher von `C-T`-Instanzen und alle Wörterbücher von `D-T`-Instanzen, wobei `D` eine Superklasse von `C` ist.

Mit einer Klasse `C` wird also auch die gesamte Superklassenhierarchie dieser Klasse exportiert. Dies ist notwendig, damit beim *Import* der Klasse `C` alle Superklassen von `C` ebenfalls importiert werden können. Nur so kann die Superklassenhierarchie von `C` wieder vollständig aufgebaut werden. Die Klassenhierarchie *muss* vollständig sein, für jede Klasse müssen also alle Superklassen in der Klassenumgebung enthalten sein. Ansonsten könnten zum Beispiel für Klassen, denen eine Superklasse in der Klassenumgebung fehlt, nicht die Wörterbuchtypen generiert werden.

Beispiel 4.17:
Es sei das folgende Modul gegeben:

```
module M (D(funD1)) where

class C a where
  funC :: a -> a

class C a => D a where
  funD1 :: a -> a
  funD2 :: a -> Bool

instance C Bool where
  funC x = x
```

Hier hängt die Klasse `D` von der Klasse `C` und den Entitäten `sel.D.C`, `sel.D.funD1`, `sel.D.funD2` und `Dict.D` ab (wegen der Klasse `D`), sowie von den Entitäten `sel.C.`

`funC` und `Dict.C` wegen der Klasse `C`. Die `C-Bool`-Instanz hängt von der Klasse `C` und den Entitäten `sel.C.funC` und `Dict.C` (wegen der Klasse `C`), sowie von ihrem Wörterbuch `dict.C.Bool` ab.

In der Exportspezifikation steht nur, dass die Klasse `D` exportiert werden soll. Da `D` aber von der Klasse `C` abhängt, muss auch die Klasse `C` – verborgen – exportiert werden. Außerdem müssen die genannten Selektionsfunktionen und Wörterbuchtypen in das Interface mit aufgenommen werden.

Alle Entitäten, von denen die explizit exportierten Klassen oder die exportierten Instanzen abhängen, werden zusätzlich in den Interfaces gespeichert. Außerdem wird bei den Instanzen und Klassen angegeben, von welchen Entitäten sie abhängen.

Damit Klassen, die zwar aufgrund von Abhängigkeiten exportiert wurden, aber nicht explizit in der Exportspezifikation angegeben wurden, nicht nach dem Import des Interfaces verwendet werden können, werden alle Klassen mit einem Flag versehen, das angibt, ob sie sichtbar oder nicht sichtbar sind. Klassen, die explizit exportiert werden, sind also immer sichtbar, und Klassen, die *nur* aufgrund von Abhängigkeiten exportiert werden, werden als *versteckt* gekennzeichnet. Die Klasse `D` aus obigem Beispiel wird also sichtbar exportiert, während `C` zwar exportiert wird, aber als nicht sichtbar gekennzeichnet wird.

Beim Export kann auch nur ein *Teil* der Klassenmethoden einer Klasse exportiert werden. Es werden aber immer alle Klassenmethoden benötigt, um den korrekten Wörterbuchtyp einer Klasse ableiten zu können. Deshalb wird auch für Klassenmethoden gespeichert, ob diese sichtbar sind oder nicht. Dies geschieht dadurch, dass pro Klasse eine Menge gespeichert wird, in der die sichtbaren Klassenmethoden enthalten sind. Im obigen Beispiel werden also im Interface von Modul `M` zwar beide Klassenmethoden von `D` angegeben, die Klassenmethode `funD2` wird aber als versteckt gekennzeichnet.

Instanzdeklarationen werden *immer* exportiert, da sie unter anderem keine Namen besitzen, und daher nicht in der Exportspezifikation angegeben werden können. Es werden sogar auch alle Instanzen, die aus anderen Modulen in das aktuelle Modul importiert wurden, re-exportiert.

Eines der generierten Interfaces für das Modul `M` aus obigem Beispiel ist in Abb. 4.10 aufgeführt. Da keine überladenen Methoden im Codebeispiel vorkommen, ist das andere Interface identisch mit dem hier angegebenen. Die Namen der Selektionsfunktionen, Wörterbücher und Wörterbuchtypen werden nun in ihrer vollständigen Form angegeben, es werden also die *qualifizierten* Klassen- bzw. Typnamen in den Namen verwendet, wie in Abschnitt 4.3.2.6 beschrieben. Der Übersichtlichkeit halber wird hier außerdem der Separator „`.`“ verwendet, im Gegensatz zu dem Separator, der tatsächlich verwendet wird (siehe Seite 119). Für Funktionen ist zusätzlich zur Typsignatur auch immer deren Stelligkeit angegeben, daher die Nummern nach dem Funktionsnamen.

4.3.3.3. Format der Interfacedateien

In Interfacedateien werden zwei Interfaces gespeichert (siehe Abschnitt 4.3.3.1 und Abbildung 4.9). Das erste Interface wird unter dem Namen „`interface`“ eingeführt, und enthält Funktionstypen *ohne* Kontexte. Das zweite Interface wird unter dem Namen „`interfaceTypeClasses`“ eingeführt, und enthält Funktionstypen *mit* Kontexten.

```
interface M where {
  -- Datentypen aus der Prelude werden benötigt
  import Prelude
  -- Prelude.Bool ist ein Datenkonstruktor
  hiding data Prelude.Bool;

  -- Klasse D mit Superklasse C wird exportiert
  class [C] D a where {
    public funD1 1 :: () => a -> a; -- funD1 ist öffentlich, während
    hiding funD2 1 :: () => a -> Prelude.Bool -- funD2 versteckt ist
  } [] -- keine Default-Methoden
    [C, Dict.M.D, sel.M.D.M.C, sel.M.D.funD1, sel.M.D.funD2,
     Dict.M.C, sel.M.C.funC] -- Abhängigkeiten

  -- Wörterbuch für die C-Bool-Instanz
  dict.M.C.Prelude.Bool 0 :: () => Prelude.Bool -> Prelude.Bool;

  -- C-Bool-Instanz aus diesem Modul
  instance[] () => C (Prelude.Bool)
    -- Abhängigkeiten
    [C, Dict.M.C, sel.M.C.funC, dict.M.C.Prelude.Bool];

  -- die Klasse C wird wegen der Klasse D exportiert,
  -- aber versteckt
  hiding class [] C a where {
    hiding funC 1 :: () => a -> a
  } [] -- keine Default-Methoden
    [Dict.M.C, sel.M.C.funC]; -- Abhängigkeiten

  -- Wörterbuchtypen und Selektionsfunktionen
  type Dict.M.D b = (b -> b, b -> b, b -> Prelude.Bool);
  type Dict.M.C b = b -> b;
  sel.M.D.M.C 1 :: () =>
    (b -> b, b -> b, b -> Prelude.Bool) -> b -> b;
  sel.M.D.funD1 1 :: () =>
    (b -> b, b -> b, b -> Prelude.Bool) -> b -> b;
  sel.M.D.funD2 1 :: () =>
    (b -> b, b -> b, b -> Prelude.Bool) -> b -> Prelude.Bool;
  sel.M.C.funC 1 :: () => (b -> b) -> b -> b
}
```

Abbildung 4.10: Interface für das Modul `M` aus Beispiel 4.17

In den Interfaces wird die folgende Konvention verfolgt: Wurden die exportierten Entitäten im aktuellen Modul definiert, so stehen diese unter ihrem *unqualifizierten* Namen in den Interfaces. Bei exportierten Entitäten, die nicht aus diesem Modul stammen, wird immer der *qualifizierte* Name verwendet. Der zweite Fall kann zum Beispiel dann eintreten, wenn Elemente eines importierten Moduls *re-exportiert* werden, oder wenn Klassen aus anderen Modulen wegen Abhängigkeiten in das Interface eingetragen werden.

Der Parser, mit dem die Interfacedateien geparst werden, kann teilweise auf die schon für das Parsen von Curry-Dateien verwendeten Parsingfunktionen zurückgreifen. Deshalb müssen nur für spezielle Konstrukte in den Interfaces neue Parsingfunktionen angegeben werden.

Da die Interface-Dateien nicht von Menschen geschrieben, sondern vom Compiler erzeugt werden, kann die Anzahl der möglichen syntaktischen Konstrukte, die in den Interfaces benutzt werden dürfen, reduziert werden. So wird bei exportierten Funktionen *immer* der Kontext angegeben, auch wenn dieser leer ist.

Die Klassendefinitionen haben die folgende Form (bis auf die beiden eckigen Klammern um das Schlüsselwort `hiding` sind alle eckigen Klammern Bestandteil des Codes und drücken Listen aus):

```
[hiding] class [S1, ..., Sn] C a where {
  qual1 method1 a1 :: τ1 ;
  ⋮
  qualm methodm am :: τm ;
} [methodk1, ..., methodkl] -- Methoden mit Default-
                             -- Implementierung
  [dep1, ..., depp] -- Abhängigkeiten
```

Ein „`hiding`“ vor der Klassendefinition gibt an, dass die Klasse versteckt ist. $\mathtt{S}_1$ bis $\mathtt{S}_n$ sind die direkten Superklassen von `C`. Die $\mathtt{qual}_i$ sind Elemente der Menge {`hiding`, `public`}, und geben an, ob die jeweiligen Klassenmethoden versteckt oder öffentlich sind. $\mathtt{method}_1$ bis $\mathtt{method}_m$ sind die Klassenmethoden der Klasse `C`, zusammen mit ihren Stelligkeiten $\mathtt{a}_1$ bis $\mathtt{a}_m$ und ihren Typen τ_1 bis τ_m. In der darauf folgenden Liste werden alle Methoden angegeben, für die eine Default-Implementierung existiert. Am Schluss wird die Liste aller Abhängigkeiten angegeben.

Anhand der Liste der Methoden mit einer Default-Implementierung kann bei der Erstellung des Wörterbuchs für eine Instanz mit der Klasse `C` entschieden werden, ob bei einer fehlenden Implementierung einer Klassenmethode die Default-Methode oder nur der generierte Fehlermeldungs-Stub eingefügt werden soll (siehe Transformation 2.5).

Instanzdefinitionen haben die folgende Form:

```
instance[[M]] (C1 ak1, ..., Cm akm) => C (T a1 ... an)
  [dep1, ..., depp] -- Abhängigkeiten
```

`M` gibt an aus welchem Modul die Instanz stammt; stammt die Instanz aus dem gerade behandelten Modul, so wird das `M` weggelassen, die Instanzdefinition beginnt also dann mit `instance[]`. Anschließend werden – wie auch im Curry-Code – der Instanzkontext, die Klasse, und der Instanztyp angegeben. Am Schluss steht die Liste der Abhängigkeiten

für diese Instanz. Da die Implementierung der Klassenmethoden nicht angegeben wird, wird im Gegensatz zu Instanzdeklarationen im Curry-Code das Schlüsselwort „`where`“ nicht verwendet.

In den Interfaces und in der Werteumgebung werden generierte Funktionsnamen wie „`sel.M1.C1.M2.C2`“ tatsächlich als „`sel:_M1.C1:_M2.C2`“ geschrieben, als „Trenner“ wird also die Zeichenkette „`:_`“ benutzt, und nicht „`.`“. Da die eigentlichen Namen schwerer zu lesen sind, wird außer hier immer der Trenner „`.`“ benutzt. Die Wahl der Zeichenkette „`:_`“ als Trenner hat einen ganz praktischen Grund: Beim Lexen hat der Punkt schon eine spezielle Funktion, und zwar trennt dieser den Modulnamen vom Bezeichner. Deshalb muss ein anderer Trenner benutzt werden. Ein Operator wie „`:`“ kann als Trenner alleine nicht verwendet werden, da dann die Namen von Selektionsfunktionen für Klassenmethoden in Operatorform nicht richtig gelext würden, denn das „`:`“ wird mit dem nachfolgenden Operatornamen verschmolzen, wie in der Selektionsfunktion „`sel:M.C:$$`“ für die Klassenmethode „`$$`“. Deshalb wird der Trenner „`:_`“ verwendet. Dieser wird immer als zwei Token gelext: Als dem Operatorsymbol „`:`“ und dem nachfolgenden Identifier „`_`“, egal ob dem „`_`“ ein Operator oder ein Funktionsname folgt.

Für den Funktionsnamen „`sel:_M1.C1:_M2.C2`“ werden also folgende Lexeme generiert: `<sel> <:> <_> <M1.C1> <:> <_> <M2.C2>`. Diese werden beim Parsen wieder zu dem eigentlichen Funktionsnamen zusammengebaut. Auch Selektionsfunktionen für Klassenmethoden in Operatorform wie „`sel:_M1.C1:_$`“ werden korrekt gelext, in diesem Fall würden die Lexeme `<sel> <:> <_> <M1.C1> <:> <_> <$>` lauten.

In Anhang A.2 ist die komplette Syntax von Interfaces in Backus-Naur-Form angegeben.

4.3.3.4. Import von Klassen und Instanzen

Beim Import von Klassen und Instanzen muss berücksichtigt werden, dass alle Entitäten, von denen die Klassen und Instanzen *abhängen*, ebenfalls (versteckt) importiert werden. Die Abhängigkeiten einer bestimmten Klasse oder Instanz sind im Interface eingetragen, so dass diese leicht ermittelt werden können.

So werden für Klassen alle Superklassen mit importiert, und für Instanzen die verwendeten Klassen und deren Superklassen. Außerdem werden für alle importierten Klassen die erzeugten Selektionsfunktionen, Defaultmethoden und Wörterbuchtypen importiert, und für Instanzen die erzeugten Wörterbücher. Diese für Klassen und Instanzen erzeugten Entitäten werden immer *ausschließlich unqualifiziert* importiert. Dabei gibt es bei den Namen der für Klassen erzeugten Entitäten keine Überschneidungen, da diese den qualifizierten Namen der Klasse enthalten. So beginnen die Selektionsfunktionen einer Klasse `C` im Modul `M` mit `sel.M.C`, die Defaultmethoden mit `def.M.C`, und die Wörterbuchtypen mit `Dict.M.C`. Da die qualifizierten Klassennamen „global eindeutig“ sind, sind dies auch entsprechenden erzeugten Entitäten.

Beim Import von Klassen muss berücksichtigt werden, dass in einem Interface durchaus mehrere Klassen mit demselben Namen auftreten können. Eine Namensüberschneidung kann aber nur für *versteckte* Klassen auftreten, die aufgrund von Abhängigkeiten exportiert wurden. Für *sichtbare* Klassen im Interface gilt, dass für jeweils zwei ver-

schiedene Klassen deren Namen *unterschiedlich* sind. Dies ist deshalb der Fall, da bei der Exportspezifikation immer nur *genau eine* Entität mit einem bestimmten Namen angegeben werden kann. Da beim Import von Entitäten im originalen Frontend von der Annahme ausgegangen wird, dass es für jeden Namen nur genau ein Element im Interface gibt, das diesen Namen trägt, nun aber aufgrund der Abhängigkeiten ein Name auf mehrere Entitäten im Interface passen kann, muss der Import-Code entsprechend angepasst werden.

Da eine Klasse auf verschiedenen Importwegen importiert werden kann, muss beim Import einer Klasse auf einem bestimmten Importweg diese Klasse mit der Klasse, die schon in der Werteumgebung eingetragen ist, *gemergt* werden, falls eine solche Klasse existiert (siehe die Erläuterung des Entitätenkonzepts in Abschnitt 4.2.1). Beim Mergen müssen die Sichtbarkeiten der Klassen und der Klassenmethoden berücksichtigt werden. So kann eine Klasse einmal aufgrund von Abhängigkeiten und ein anderes Mal explizit importiert werden. In diesem Fall muss die Klasse als sichtbar markiert werden. Außerdem können beim Import einer Klasse in verschiedenen Importangaben jeweils verschiedene Klassenmethoden angegeben werden. Da *alle* in den Importangaben angegebenen Klassenmethoden verfügbar sein sollen, müssen die Mengen, in denen die sichtbaren Klassenmethoden aufgeführt sind, beim Mergen *vereinigt* werden.

Formal kann der Merge-Vorgang folgendermaßen beschrieben werden: Lautet das hidden-Flag der einen am Merge-Vorgang beteiligten Klasse hidden_1 und das hidden-Flag der anderen beteiligten Klasse hidden_2, so lautet das hidden-Flag der Ergebnisklasse $\text{hidden} = \text{hidden}_1 \;\&\&\; \text{hidden}_2$. Lautet die Menge der sichtbaren Klassenmethoden der einen Klasse VisibleMethods_1, und die der anderen Klasse VisibleMethods_2, so lautet die Menge der sichtbaren Klassenmethoden der Ergebnisklasse $\text{VisibleMethods} = \text{VisibleMethods}_1 \cup \text{VisibleMethods}_2$.

Beispiel 4.18:
Es seien die folgenden Module gegeben:

```
module M where
  class C a where
    funC :: a -> a

  class C a => D a where
    funD :: a -> a

  class D a => E a where
    funE1 :: a -> a
    funE2 :: a -> a
    funE3 :: a -> a

module M2 where
  import M (E(funE1))
  import M (E(funE2), C(..))
```

In Modul `M2` wird die Klasse `E` zusammen mit den Klassenmethoden `funE1` *und* `funE2` sichtbar importiert, da die Mengen der Klassenmethoden, die in den beiden Importspezifikationen angegeben sind, vereinigt werden.

Die Klasse `C` wird sichtbar importiert, da sie in der ersten Importspezifikation zwar nur implizit, in der zweiten Importspezifikation aber explizit importiert wird.

Die Klasse `D` wird nur versteckt importiert, da `E` von `D` abhängt, und `D` in keiner Importspezifikation angegeben wurde.

Die Sichtbarkeiten der Klassen bzw. der Klassenmethoden werden im Type Classes Check bzw. im Syntaxcheck berücksichtigt. Im Type Classes Check wird überprüft, dass alle im Code vorkommenden Klassennamen sich nur auf *sichtbare* Klassen beziehen; im Syntaxcheck werden in die Menge der gültigen Funktionsnamen nur die *sichtbaren* Klassenmethoden aufgenommen.

5. Abschlussbetrachtungen

5.1. Zusammenfassung

In der vorliegenden Arbeit wurden Typklassen für das Curry-System KiCS2 implementiert. Zu den Typklassenelementen gehören Klassendefinitionen, die angeben, unter welchem Klassennamen welche Funktionen überladen werden können. In Instanzdefinitionen können jeweils für einen bestimmten Typ konkrete Implementierungen von Klassenmethoden angegeben werden. Es können Default-Methoden in Klassen angegeben werden, die dann ausgeführt werden, wenn in einer Instanzdefinition keine entsprechende Implementierung angegeben wurde.

Überladene Funktionen besitzen einen *Kontext*. Im Kontext wird angegeben, welche Klassen welche Typvariablen einschränken. Durch Kontextreduktion wird ein Kontext soweit reduziert, dass seine Elemente nur noch in Kopf-Normalform vorliegen. Bei der Reduktion werden Instanz- und Klassendefinitionen benutzt.

Für bestimmte Prelude-Klassen und algebraische Datentypen können automatisch Instanzen abgeleitet werden, und zwar für die Klassen `Show`, `Eq`, `Ord`, `Enum` und `Bounded`.

Typklassen können mit Wörterbüchern implementiert werden. Ein Wörterbuch enthält die konkreten Implementierungen von Klassenmethoden, die in einer Instanzdefinition für einen bestimmten Typ angegeben wurden. Zudem sind in einem Wörterbuch auch alle Wörterbücher für Superklassen und denselben Typ enthalten.

Für Klassendefinitionen werden Selektionsfunktionen generiert, die aus einem Wörterbuch die gewünschten Implementierungen von Klassenmethoden oder Superklassenwörterbücher extrahieren. Außerdem wird ein Typsynonym für den Typ des Wörterbuchs angelegt. Für Instanzdefinitionen werden konkrete Wörterbücher generiert, die die angegebenen Implementierungen enthalten.

Für das Einfügen von Wörterbüchern werden die Kontexte überladener Funktionen betrachtet. Der Kontext einer Funktionsdefinition ist in Kopf-Normalform gegeben. Für jedes Kontextelement wird ein Wörterbuchparameter generiert. Auf den rechten Seiten von Gleichungen können selbst wieder überladene Funktionen benutzt werden. Die konkreten Typen und Kontexte dieser Funktionen geben an, welche Wörterbücher den überladenen Funktionen übergeben werden müssen. Bei der Codetransformation werden nun aus den Wörterbüchern, die einer überladenen Funktion übergeben werden, die Wörterbücher generiert, die von den benutzten überladenen Funktionen benötigt werden.

Dabei können Wörterbücher aus anderen Wörterbüchern abgeleitet werden. Aufgrund von Instanzdefinitionen mit einem Kontext werden Wörterbücher aus anderen Wörterbüchern zusammengesetzt. Aufgrund von Klassendefinitionen können aus einem Wörterbuch die Wörterbücher für Superklassen extrahiert werden.

Damit die Kontexte von Ausdrücken bestimmt werden können, muss das Damas-Milner-Typsystem, auf dem Curry basiert, erweitert werden. Dies geschieht allgemein durch die Einführung von Prädikaten auf Typen. Auch Constraints können als Prädikate auf Typen aufgefasst werden, und werden daher von der allgemeinen Theorie impliziert.

Der Algorithmus $\mathcal{W}$, mit dem die Typen von Ausdrücken bestimmt werden, wird um Prädikate erweitert. Neben den Typen werden nun also auch die Prädikate bestimmt, die den Typen einschränken müssen. Dies sind in unserem Anwendungsfall die Constraints.

Die konkrete Implementierung von Typklassen für den KiCS2-Compiler setzt die theoretischen Ergebnisse in konkreten Code um. Für die Implementierung muss bis auf eine Ausnahme nur das Frontend angepasst werden, da Programme mit Typklassenelementen in Programme ohne Typklassenelemente umgewandelt werden können, die gültig im Damas-Milner-Typsystem sind. Das Frontend kompiliert Curry in eine Zwischensprache, die von Backends weiterkompiliert wird. Der KiCS2-Compiler kompiliert die Zwischensprache nach Haskell.

Um Typklassen zu implementieren, muss der Compiler erweitert werden: Die Compilerumgebung muss angepasst werden, die einzelnen Kompilierstadien müssen geändert und neue hinzugefügt werden, und der Import und der Export müssen angepasst werden.

Der Compilerumgebung wird eine Klassenumgebung hinzugefügt, die die Klassen und Instanzen sowie einige Hilfselemente enthält. Zu den einzelnen Kompilierphasen wird ein Check hinzugefügt, der die Typklassenelemente prüft, Instanzen für Datentypen ableitet und Transformationen von Klassen- und Instanzdefinitionen durchführt. Außerdem wird eine Kompilierphase hinzugefügt, in der die Wörterbücher und Wörterbuchparameter eingefügt werden.

Der Typcheck wird dahingehend erweitert, dass auch Kontexte ermittelt werden. Dabei wird der erweiterte Algorithmus $\mathcal{W}$ benutzt. Um die korrekten Kontexte der Elemente einer Deklarationsgruppe zu ermitteln, wird eine Fixpunktiteration durchgeführt.

Der Parser und der Lexer müssen so erweitert werden, dass sie nun die um Typklassen erweiterte Grammatik erkennen können.

Beim Import und beim Export müssen auch Klassen- und Instanzdefinitionen berücksichtigt werden. Klassen werden in einer Umgebung gespeichert; in Umgebungen werden Namen Mengen von Entitäten zugeordnet. Beim Export und Import von Klassen muss berücksichtigt werden, dass für eine bestimmte Klasse weitere Elemente exportiert und importiert werden müssen, von denen die Klasse implizit abhängt. So werden für Klassen auch die Superklassen exportiert und importiert, und die generierten Entitäten wie Selektionsfunktionen und Wörterbuchtypen. Bei Instanzdefinitionen werden ebenfalls die benutzten Klassen und deren Superklassen, sowie die für diese Klassen erzeugten Entitäten und das Instanzwörterbuch exportiert und importiert, wobei Instanzdefinitionen aber nicht explizit für den Export bzw. Import angegeben werden können, sondern immer exportiert und importiert werden.

Damit implizit exportierte und importierte Klassen nicht benutzt werden können, werden Klassen mit einem Flag markiert, das angibt, ob die Klassen sichtbar sind oder nicht. Außerdem können Klassenmethoden ebenfalls verborgen werden, wenn beim Export oder Import nur ein Teil der Klassenmethoden exportiert oder importiert werden.

5.2. Ausblick

Die vorliegende Implementierung bietet Raum für konzeptuelle als auch die Implementierung betreffende Erweiterungen. Außerdem sind noch kleinere Bugs in der Implementierung enthalten, die ich zuerst aufführe:

- Konstrukte wie `fun = 1 where Just x = Just (funC x)`, wobei `funC` eine Klassenmethode ist, werden im Moment nicht kompiliert (siehe Abschnitt 4.3.2.8).

- Der Typcheck gibt zurzeit in manchen Fällen bei einem mehrdeutigen Kontextelement keine entsprechende Fehlermeldung aus. Dies liegt an einem Workaround, der für das folgende Problem eingebaut wurde:

  In einer Typsynonymdefinition können auf der linken Seite Typvariablen auftreten, die nicht auf der rechten Seite vorkommen, wie in `type Synonym a = ()`. Typsignaturen, die dieses Typsynonym benutzen, können zwar in der Originalform gültig sein, aber in der expandierten Form nicht. Ein Beispiel dafür ist der Typ `Eq a => Synonym a`, dessen expandierte Form `Eq a => () lautet`. Der erste Typ ist korrekt, während im zweiten Typ das Kontextelement `Eq a` mehrdeutig, und somit die Typsignatur ungültig ist.

  Der Typcheck arbeitet mit *expandierten* Typsignaturen. Daher kann wie im obigen Beispiel der Fall eintreten, dass ein inferierter Typ als ungültig betrachtet wird, obwohl er eigentlich nach der originalen Typsignatur gültig ist.

  Deshalb müssten bei einer potenziellen Mehrdeutigkeit immer die originalen Typsignaturen herangezogen werden, und es müsste geprüft werden, ob die Mehrdeutigkeit auch noch bei der unexpandierten Typsignatur auftritt. Dies ist im Moment aber nicht implementiert.

  Als Workaround werden stattdessen bei einer potenziellen Mehrdeutigkeit in dem inferierten Typ einer Funktion, wenn eine explizite Typsignatur für diese Funktion angegeben ist, *keine* Mehrdeutigkeitsfehlermeldungen ausgegeben, auch wenn tatsächlich eine Mehrdeutigkeit vorliegt; es werden dafür aber bei einer tatsächlichen Mehrdeutigkeit an anderer Stelle Fehlermeldungen generiert, die allerdings nicht den eigentlichen Fehler angeben. Somit ist garantiert, dass ein ungültiges Programm immer als ungültig erkannt wird. Wenn eine Funktion `f` *keine* explizite Typsignatur hat, und eine Mehrdeutigkeit im inferierten Typ von `f` auftritt, so ist diese Mehrdeutigkeit immer korrekt, und es wird in diesem Fall immer eine Mehrdeutigkeitsfehlermeldung ausgegeben.

- Bei einer Klassendeklaration der Form „`class A a where`“, bei der keine Klassenmethoden angegeben sind, wird eine Warnung ausgegeben, dass die Typvariable `a` nicht referenziert wird. Dies liegt daran, dass der erzeugte Wörterbuchtyp „`type Dict.A a = ()`“ lautet.

Konzeptuell könnten noch die folgenden Erweiterungen durchgeführt werden:

- In den Typsignaturen von Klassenmethoden könnten andere Typvariablen als die Typvariable der Klasse zugelassen werden. Dazu muss aber das Typsystem um existentielle Datentypen erweitert werden; ein Wörterbuch für die Klasse `class C a where funC :: a -> b -> c` würde dann zum Beispiel den folgenden Wörterbuchtyp haben: `data Dict.C a = Dict.C (∀ b c. a -> b -> c)`.

- Darauf aufbauend könnten auch Kontexte in Klassenmethoden zugelassen werden, wie in `class C a where funC :: D b => a -> b`, wobei `D` eine Klasse ist.

- Konstruktorklassen könnten unterstützt werden (siehe Abschnitt 2.2.4). Dafür müssten dann Kontextelemente der Form `C (a` τ_1 `...`τ_n`)` zugelassen werden, wobei `a` eine Typvariable (!) ist, und es wären wieder existentielle Datentypen notwendig, da die Typsignaturen aller Methoden von Konstruktorklassen automatisch andere Typvariablen als die der Klasse enthalten.

- Es könnten Multiparameterklassen unterstützt werden (siehe Abschnitt 2.2.4).

- Zurzeit muss die Kontextreduktion bei der Generalisierung des Typs einer Funktion immer vollständig sein, das heißt, alle Elemente des Kontexts müssen nach der Reduktion in Kopf-Normalform sein. Folgender Code ist nicht erlaubt:

  ```
  class C a where
    funC :: a -> a

  fun x = funC (x, x)
  ```

  da keine `C-(,)`-Instanz vorhanden ist, mit der der für `fun` ermittelte Kontext `C (a, a)` reduziert werden könnte.

  Haskell unterstützt jedoch die (undokumentierte) Erweiterung, dass dieser Code doch gültig ist. Der Kontext von `fun` wird zuerst nicht vollständig reduziert und lautet `C (a, a)`. Erst beim Anwenden der Funktion `fun` wird anscheinend versucht, den Kontext vollständig zu reduzieren, und erst dann beim Fehlschlag ein Fehler ausgegeben. Dies erlaubt es, dass bei der Anwendung von `fun` in unterschiedlichen Modulen verschiedene `C-(,)`-Instanzen benutzt werden können.

- Die `Read`-Klasse und die automatische Ableitung von Read-Instanzen sind nicht implementiert. In Read-Instanzen würden komplexere Parser und Lexer benötigt, um den Eingabestring zu parsen.

- Die Monomorphie-Restriktion, wie sie aus Haskell bekannt ist [M$^+$10, S. 56 ff.], ist nicht implementiert.

- Das *Defaulting* von Typvariablen in mehrdeutigen Kontextelementen mit numerischen Klassen, bei dem die Typvariablen durch numerische Typen ersetzt werden (siehe Haskell Report [M$^+$10, S. 48 ff.]), ist nicht implementiert. Dies bedeutet, dass unter Umständen an mehr Stellen explizit der Typ von numerischen Literalen angegeben werden muss als es der Fall wäre, wenn Defaulting implementiert wäre.

Bei der Implementierung sind noch folgende Verbesserungen möglich:

- Der zweite Typcheck könnte entfernt werden und durch eine einfache Transformation der Funktionstypen in der Werteumgebung ersetzt werden. Allerdings bietet der zweite Typcheck mehr Programmiersicherheit, da ungültige Wörterbuchtransformationen detektiert werden.
- Beim Import und Export könnte auch zugelassen werden, dass einzelne Klassenmethoden alleine für den Import bzw. Export angegeben werden dürfen, ohne die Angabe eines Klassennamens. So könnte zum Beispiel die Klassenmethode `funC` der Klasse `C` auch direkt über die Angabe „`funC`“ anstatt über „`C(funC)`“ exportiert bzw. importiert werden. Dies würde aber die Implementierung des Exports und des Imports stark verkomplizieren, und der Export bzw. der Import mit der Angabe des Klassennamens ist in den meisten Fällen ausreichend.
- Wenn bestimmte Klassenmethoden in einer Instanzdefinition nicht implementiert werden, und keine Default-Implementierungen vorhanden sind, könnte eine Warnung ausgegeben werden.
- Es könnten *algebraische Datentypen* anstatt von *Tupeln* für die Implementierung von Wörterbüchern verwendet werden, womit eine Erweiterung der Implementierung um Konstruktorklassen vereinfacht würde, da bei algebraischen Datentypen *existentielle* Typvariablen angegeben werden können.

In den Veröffentlichungen [Aug93, PJ93] werden Verbesserungen vorgeschlagen, die die *Effizienz* von Programmen mit Typklassen erhöhen können. Davon will ich hier zwei herausgreifen:

„Abflachen“ des Wörterbuchs In der vorliegenden Implementierung werden Wörterbücher als geschachtelte Tupel repräsentiert. Diese Schachtelung könnte aufgelöst werden, indem die gesamte Tupelstruktur „abgeflacht“ wird und nur noch ein einziges Tupel verwendet wird, in dem die Implementierungen aller Methoden der Klasse und der Superklassen abgelegt werden. Dies hätte zur Folge, dass für die Selektion von Methoden von Superklassen nur noch ein Funktionsaufruf notwendig ist, und keine Verkettung von Funktionen, die nacheinander die Superklassenwörterbücher extrahieren, bis die gewünschte Klassenmethode extrahiert werden kann.

Beispiel 5.1:
Es seien die folgenden Klassen gegeben:

```
class C a where
  funC1 :: a -> a
  funC2 :: a -> Bool

class C a => D a where
  funD :: a -> a -> a
```

```
class D a => E a where
  funE :: a -> a
```

Dann lautet der Typ des Wörterbuchs für `E` „`(((a -> a, a -> Bool), a -> a -> a), a -> a)`“. Durch Abflachung wird dieser Typ in den Typ „`(a -> a, a -> Bool, a -> a -> a, a -> a)`“ umgewandelt.

Die ursprünglich erzeugten Selektionsfunktionen lauten:

```
sel.C.funC1 (funC1, funC2) = funC1
sel.C.funC2 (funC1, funC2) = funC2

sel.D.C    (dict.C, funD) = dict.C
sel.D.funD (dict.C, funD) = funD

sel.E.D    (dict.D, funE) = dict.D
sel.E.funE (dict.D, funE) = funE
sel.E.C                   = selD.C . sel.E.D
```

Für das abgeflachte Wörterbuch der Klasse `E` werden nun die folgenden Selektionsfunktionen generiert:

```
sel.E.funC1 (funC1, funC2, funD, funE) = funC1
sel.E.funC2 (funC1, funC2, funD, funE) = funC2
sel.E.funD  (funC1, funC2, funD, funE) = funD
sel.E.funE  (funC1, funC2, funD, funE) = funE
```

Es sei nun die folgende Funktion gegeben:

```
fun :: E a => a -> a
fun x = funC1 x
```

Die Funktion wird ursprünglich in folgenden Code transformiert:

```
fun dict.E.a x = sel.C.funC1 (sel.E.C dict.E.a) x
```

Nach der Abflachung des Wörterbuchs kann nun für die Funktion `fun` der folgende Code generiert werden:

```
fun dict.E.a x = sel.E.funC1 dict.E.a x
```

Es werden also insgesamt zwei Funktionsaufrufe gespart.

Gerade wenn die Superklassenhierarchie tief ist, können durch diese Optimierung viele Funktionsaufrufe gespart werden.

Partielle Auswertung Bei der Übergabe von konkreten Wörterbüchern an Klassenmethoden kann eine partielle Auswertung geschehen, wie im folgenden Beispiel:

```
class C a where
  funC :: a -> a

instance C Int where
  funC x = x

fun = funC 1
```

Die Funktion `fun` wird folgendermaßen übersetzt:

```
fun = sel.C.funC dict.C.Int 1
```

Da die selektierte Funktion aus dem Wörterbuch `dict.C.Int` der Funktion `impl.C.Int.funC` entspricht, kann dies durch folgenden Code ersetzt werden:

```
fun = impl.C.Int.funC 1
```

womit ein Funktionsaufruf gespart wird.

A. Anhang

A.1. Syntax von um Typklassen erweitertem Curry

Im Folgenden ist die Syntax von um Typklassen erweitertem Curry aufgeführt. Alle für die Erweiterung um Typklassen hinzugefügten Elemente sind farbig markiert. Die Originalsyntax ohne Typklassen ist dem Handbuch von KiCS2 entnommen [H$^+$]. Diese entspricht zwar nicht vollständig der Syntax aus dem „Curry Report“ [He12]; sie entspricht aber der Syntax, die von der vorliegenden Implementierung erkannt wird, und wird deshalb als Grundlage genommen.

A.1.1. Notation

Die Syntax wird in erweiterter Backus-Naur-Form (eBNF) angegeben, wobei die folgende Notation verwendet wird:

$\mathit{NonTerm} ::= \alpha$	Produktionsregel
$\mathit{NonTerm}$	Nichtterminal-Symbol
`Term`	Terminal-Symbol
$[\alpha]$	optional
$\{\alpha\}$	keine oder mehrere Wiederholungen
(α)	Gruppierung
$\alpha \mid \beta$	Alternative
$\alpha_{\langle\beta\rangle}$	Differenz – enthält Elemente, die von α generiert werden, aber nicht Elemente, die von β generiert werden

A.1.2. Wortschatz

A.1.2.1. Bezeichner und Schlüsselwörter

Letter ::= any ASCII letter
Dashes ::= `-` {`-`}

Ident ::= *Letter* {*Letter* | *Digit* | `_` | `'`}
Symbol ::= `~` | `!` | `@` | `#` | `$` | `%` | `^` | `&` | `*` | `+` | `-` | `=` | `<` | `>` | `?` | `.` | `/` | `|` | `\` | `:`

ModuleID ::= {*Ident* `.`} *Ident*
TypeConstrID ::= *Ident*
DataConstrID ::= *Ident*
TypeVarID ::= *Ident* | `_`

$$
\begin{array}{rcl}
\mathit{BTypeVarID} & ::= & \mathit{Ident} \\
\mathit{InfixOpID} & ::= & (\mathit{Symbol}\ \{\mathit{Symbol}\})_{\langle \mathit{Dashes} \rangle} \\
\mathit{FunctionID} & ::= & \mathit{Ident} \\
\mathit{VariableID} & ::= & \mathit{Ident} \\
\mathit{LabelID} & ::= & \mathit{Ident} \\
\mathit{TypeClassID} & ::= & \mathit{Ident} \\
\\
\mathit{QTypeConstrID} & ::= & [\mathit{ModuleID}\ .]\ \mathit{TypeConstrID} \\
\mathit{QDataConstrID} & ::= & [\mathit{ModuleID}\ .]\ \mathit{DataConstrID} \\
\mathit{QInfixOpID} & ::= & [\mathit{ModuleID}\ .]\ \mathit{InfixOpID} \\
\mathit{QFunctionID} & ::= & [\mathit{ModuleID}\ .]\ \mathit{FunctionID} \\
\mathit{QVariableID} & ::= & [\mathit{ModuleID}\ .]\ \mathit{VariableID} \\
\mathit{QTypeClassID} & ::= & [\mathit{ModuleID}\ .]\ \mathit{TypeClassID}
\end{array}
$$

Die folgenden Bezeichner sind Schlüsselwörter, und können daher nicht als Bezeichner benutzt werden:

```
case        class      data        deriving
do          else       external    fcase
foreign     free       if          import
in          infix      infixl      infixr
instance    let        module      newtype
of          then       type        where
```

Die folgenden Symbole haben eine spezielle Bedeutung und können nicht als Infix-Operatoren benutzt werden:

```
..      :       ::      =       \\      |       <-      ->      @
~       :>      :=      =>
```

A.1.2.2. Kommentare

Kommentare beginnen entweder mit `--` und enden am Ende der Zeile, oder mit `{-` und enden bei einem zugehörigen `-}`, das heißt, `{-` und `-}` verhalten sich wie Klammern und können geschachtelt werden.

A.1.2.3. Zahlen- und Buchstaben-Literale

Für die Zahlen- und Buchstaben-Literale ergeben sich keine Syntax-Änderungen bezüglich der Originalsyntax.

$$
\begin{array}{rcl}
\mathit{Int} & ::= & \mathit{Decimal} \\
 & | & \texttt{0o}\ \mathit{Octal} \mid \texttt{0O}\ \mathit{Octal} \\
 & | & \texttt{0x}\ \mathit{Hexadecimal} \mid \texttt{0X}\ \mathit{Hexadecimal} \\
\\
\mathit{Float} & ::= & \mathit{Decimal}\ \texttt{.}\ \mathit{Decimal}\ [\mathit{Exponent}] \\
 & | & \mathit{Decimal}\ \mathit{Exponent} \\
\mathit{Exponent} & ::= & (\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \mathit{Decimal}
\end{array}
$$

Decimal ::= *Digit* [*Decimal*]
Octal ::= *Octit* [*Octal*]
Hexadecimal ::= *Hexit* [*Hexadecimal*]

Digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
Octit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
Hexit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`

Char ::= `'`(*Graphic*$_{\langle \text{'} | \backslash \rangle}$ | *Space* | *Escape*$_{\langle \backslash \& \rangle}$)`'`
String ::= `"`{*Graphic*$_{\langle \text{"} | \backslash \rangle}$ | *Space* | *Escape* | *Gap*}`"`
Escape ::= `\` (*CharEsc* | *Ascii* | *Decimal* | `o` *Octal* | `x` *Hexadecimal*)
CharEsc ::= `a` | `b` | `f` | `n` | `r` | `t` | `v` | `\` | `"` | `'` | `&`
Ascii ::= `^` *Cntrl* | `NUL` | `SOH` | `STX` | `ETX` | `EOT` | `ENQ` | `ACK`
| `BEL` | `BS` | `HT` | `LF` | `VT` | `FF` | `CR` | `SO` | `SI` | `DLE`
| `DC1` | `DC2` | `DC3` | `DC4` | `NAK` | `SYN` | `ETB` | `CAN`
| `EM` | `SUB` | `ESC` | `FS` | `GS` | `RS` | `US` | `SP` | `DEL`
Cntrl ::= *AsciiLarge* | `@` | `[` | `\` | `]` | `^` | `_`
AsciiLarge ::= `A` | ... | `Z`
Gap ::= `\` *WhiteChar* {*WhiteChar*} `\`

A.1.3. Layout

Curry benutzt Haskell-ähnliche Layoutregeln für Whitespace. Für Details siehe das KiCS2-Benutzerhandbuch [H$^+$]. Hier sei nur aufgeführt, dass Code in der Form `{` C_1 `;` ... `;` C_n `}` auch durch entsprechende Einrückungen ohne die geschweiften Klammern und Semikolons geschrieben werden kann. In der kontextfreien Grammatik wird immer die erste Form angegeben.

A.1.4. Kontextfreie Grammatik

Module ::= `module` *ModuleID* [*Exports*] `where` *Block*
| *Block*
ModuleID ::= see lexicon

Exports ::= `(` *Export*$_1$ `,` ... `,` *Export*$_n$ `)` $(n \geq 0)$
Export ::= *QFunctionName*
| *QTypeConstrID*
[`(` *ConsLabelName*$_1$ `,` ... `,` *ConsLabelName*$_n$ `)`] $(n \geq 0)$
| *QTypeConstrID* `(..)`
| `module` *ModuleID*
ConsLabelName ::= *LabelID* | *DataConstr*

Block ::= `{` [*ImportDecl*$_1$ `;` ... `;` *ImportDecl*$_k$ `;`] (*no fixity declarations here*)
BlockDeclaration$_1$ `;` ... `;` *BlockDeclaration*$_n$ `}` $(k, n \geq 0)$

ImportDecl ::= `import` [`qualified`] *ModuleID* [`as` *ModuleID*] [*ImportRestr*]
ImportRestr ::= `(` *Import*$_1$ `,` ... `,` *Import*$_n$ `)` $(n \geq 0)$
| `hiding (` *Import*$_1$ `,` ... `,` *Import*$_n$ `)` $(n \geq 0)$

$$
\begin{array}{rcll}
\textit{Import} & ::= & \textit{FunctionName} & \\
& | & \textit{TypeConstrID} & \\
& & \quad [\texttt{(}\ \textit{ConsLabelName}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{ConsLabelName}_n\ \texttt{)}] & (n \geq 0) \\
& | & \textit{TypeConstrID}\ \texttt{(..)} & \\
\\
\textit{BlockDeclaration} & ::= & \textit{TypeSynonymDecl} & \\
& | & \textit{DataDeclaration} & \\
& | & \textit{FixityDeclaration} & \\
& | & \textit{FunctionDeclaration} & \\
& | & \textit{TypeClassDeclaration} & \\
& | & \textit{InstanceDeclaration} & \\
\\
\textit{TypeSynonymDecl} & ::= & \texttt{type}\ \textit{SimpleType}\ \texttt{=}\ (\textit{TypeExpr} \mid \textit{RecordType}) & \\
\textit{SimpleType} & ::= & \textit{TypeConstrID}\ \textit{TypeVarID}_1\ \ldots\ \textit{TypeVarID}_n & (n \geq 0) \\
\textit{TypeConstrID} & ::= & \text{see lexicon} & \\
\\
\textit{RecordType} & ::= & \texttt{\{}\ \textit{LabelDecl}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{LabelDecl}_n\ \texttt{\}} & (n > 0) \\
\textit{LabelDecl} & ::= & \textit{LabelID}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{LabelID}_n\ \texttt{::}\ \textit{TypeExpr} & (n > 0) \\
\textit{LabelID} & ::= & \text{see lexicon} & \\
\\
\textit{DataDeclaration} & ::= & \texttt{data}\ \textit{SimpleType}\ [\texttt{deriving}\ \textit{DerivingDecl}] & (\textit{external data type}) \\
& | & \texttt{data}\ \textit{SimpleType}\ \texttt{=}\ \textit{ConstrDecl}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \textit{ConstrDecl}_n & \\
& & \quad [\texttt{deriving}\ \textit{DerivingDecl}] & (n > 0) \\
\textit{ConstrDecl} & ::= & \textit{DataConstr}\ \textit{SimpleTypeExpr}_1\ \ldots\ \textit{SimpleTypeExpr}_n & (n \geq 0) \\
& | & \textit{SimpleTypeExpr}\ \textit{ConsOp}\ \textit{TypeConsExpr} & (\textit{infix data constructor}) \\
\textit{DerivingDecl} & ::= & \textit{QTypeClassID} & \\
& | & \texttt{(}\ \textit{QTypeClassID}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{QTypeClassID}_n\ \texttt{)} & (n \geq 0) \\
\\
\textit{TypeExpr} & ::= & \textit{TypeConsExpr}\ [\texttt{->}\ \textit{TypeExpr}] & \\
\textit{TypeConsExpr} & ::= & \textit{QTypeConstrID}\ \textit{SimpleTypeExpr}_1\ \ldots\ \textit{SimpleTypeExpr}_n & (n > 0) \\
& | & \textit{SimpleTypeExpr} & \\
\textit{SimpleTypeExpr} & ::= & \textit{TypeVarID} & \\
& | & \textit{QTypeConstrID} & \\
& | & \texttt{()} & (\textit{unit type}) \\
& | & \texttt{(}\ \textit{TypeExpr}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{TypeExpr}_n\ \texttt{)} & (\textit{tuple type},\ n > 1) \\
& | & \texttt{[}\ \textit{TypeExpr}\ \texttt{]} & (\textit{list type}) \\
& | & \texttt{(}\ \textit{TypeExpr}\ \texttt{)} & (\textit{parenthesized type}) \\
\textit{TypeVarID} & ::= & \text{see lexicon} & \\
\\
\textit{FixityDeclaration} & ::= & \textit{FixityKeyword}\ \textit{Digit}\ \textit{InfixOpID}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{InfixOpID}_n & (n > 0) \\
\textit{FixityKeyword} & ::= & \texttt{infixl} \mid \texttt{infixr} \mid \texttt{infix} & \\
\textit{InfixOpID} & ::= & \text{see lexicon} & \\
\\
\textit{TypeClassDeclaration} & ::= & \texttt{class}\ [\textit{SimpleContext}\ \texttt{=>}]\ \textit{TypeClassID}\ \textit{BTypeVarID} & \\
& & \quad [\texttt{where}\ \textit{ClassDecls}] & \\
\textit{ClassDecls} & ::= & \texttt{\{}\ \textit{ClassDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \textit{ClassDecl}_n\ \texttt{\}} & (n \geq 0) \\
\textit{ClassDecl} & ::= & \textit{Signature} \mid \textit{Equat} & \\
\textit{SimpleContext} & ::= & \textit{Constraint} \mid \texttt{(}\ \textit{Constraint}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{Constraint}_n\ \texttt{)} & (n \geq 0) \\
\textit{Constraint} & ::= & \textit{QTypeClassID}\ \textit{BTypeVarID} & \\
\textit{TypeClassID} & ::= & \text{see lexicon} & \\
\textit{QTypeClassID} & ::= & \text{see lexicon} & \\
\\
\textit{InstanceDeclaration} & ::= & \texttt{instance}\ [\textit{SimpleContext}\ \texttt{=>}]\ \textit{QTypeClassID}\ \textit{InstanceType} &
\end{array}
$$

```
                                [where InstanceDecls]
InstanceDecls  ::=  { InstanceDecl_1 ; ... ; InstanceDecl_n }                (n ≥ 0)
InstanceDecl   ::=  Equat
InstanceType   ::=  GeneralTypeConstr
                |   ( GeneralTypeConstr BTypeVarID_1 ... BTypeVarID_n )      (n ≥ 0)
                |   ( BTypeVarID_1 , ... , BTypeVarID_n )          (tuple type, n ≥ 2)
                |   [ BTypeVarID ]                                      (list type)
                |   ( BTypeVarID_1 -> BTypeVarID_2 )                    (arrow type)
GeneralTypeConstr ::=  QTypeConstrID | () | [] | (->) | (,{,})

FunctionDeclaration ::=  Signature | External | Equat
External       ::=  FunctionNames external          (externally defined functions)
Signature      ::=  FunctionNames :: [Context =>] TypeExpr
Context        ::=  SimpleContext      (might change in future implementations)
FunctionNames  ::=  FunctionName_1 , ... , FunctionName_n                    (n > 0)
Equat          ::=  FunLHS = TypedExpr [where LocalDefs]
                |   FunLHS CondExprs [where LocalDefs]
FunLHS         ::=  FunctionName SimplePat_1 ... SimplePat_n                 (n ≥ 0)
                |   SimplePat InfixOpID SimplePat
CondExprs      ::=  | InfixExpr = TypedExpr [CondExprs]

Pattern        ::=  ConsPattern [QConsOp Pattern]       (infix constructor pattern)
ConsPattern    ::=  GDataConstr SimplePat_1 ... SimplePat_n     (constructor pattern)
                |   SimplePat
SimplePat      ::=  Variable
                |   _                                                  (wildcard)
                |   QDataConstr
                |   Literal
                |   - Int                                      (negative pattern)
                |   -. Float                            (negative float pattern)
                |   ()                                   (empty tuple pattern)
                |   ( Pattern_1 , ... , Pattern_n )                        (n > 1)
                |   ( Pattern )                              (parenthesized pattern)
                |   [ Pattern_1 , ... , Pattern_n ]                        (n ≥ 0)
                |   Variable @ SimplePat                          (as-pattern)
                |   ~ SimplePat                                (irrefutable pattern)
                |   ( SimplePat QFunOp SimplePat )          (infix functional pattern)
                |   ( QFunctionName SimplePat_1 ... SimplePat_n )
                                                  (functional pattern, n > 0)
                |   { FieldPat_1 , ... , FieldPat_n [| _] }              (record pattern)
FieldPat       ::=  LabelID = Pattern

LocalDefs      ::=  {ValueDeclaration_1 ; ... ; ValueDeclaration_n }         (n > 0)
ValueDeclaration ::=  FunctionDeclaration
                |   PatternDeclaration
                |   VariableID_1 , ... , VariableID_n free                   (n > 0)
                |   FixityDeclaration
PatternDeclaration ::=  Pattern = TypedExpr [where LocalDefs]

TypedExpr      ::=  InfixExpr :: [Context =>] TypeExpr     (expression type signature)
                |   InfixExpr
```

InfixExpr	::=	*Expr QOp InfixExpr*	(*infix operator application*)
	|	`-` *InfixExpr*	(*unary int minus*)
	|	`-.` *InfixExpr*	(*unary float minus*)
	|	*Expr*	
Expr	::=	`\` $SimplePat_1$ ... $SimplePat_n$ `->` *TypedExpr*	(*lambda expression*, $n > 0$)
	|	`let` *LocalDefs* `in` *TypedExpr*	(*let expression*)
	|	`if` *TypedExpr* `then` *TypedExpr* `else` *TypedExpr*	(*conditional*)
	|	`case` *TypedExpr* `of {`Alt_1 `;` ... `;` Alt_n `}`	(*case expression*, $n \geq 0$)
	|	`fcase` *TypedExpr* `of {`Alt_1 `;` ... `;` Alt_n `}`	(*fcase expression*, $n \geq 0$)
	|	`do {` $Stmt_1$ `;` ... `;` $Stmt_n$ `;` *TypedExpr* `}`	(*do expression*, $n \geq 0$)
	|	*FunctExpr*	
FunctExpr	::=	[*FunctExpr*] *BasicExpr*	(*function application*)
	|	*FunctExpr* `:>` *LabelID*	(*record selection*)
BasicExpr	::=	*QVariableID*	(*variable*)
	|	`_`	(*anonymous free variable*)
	|	*QFunctionName*	(*qualified function*)
	|	*GDataConstr*	(*general constructor*)
	|	*Literal*	
	|	`(` *TypedExpr* `)`	(*parenthesized expression*)
	|	`(` $TypedExpr_1$ `,` ... `,` $TypedExpr_n$ `)`	(*tuple*, $n > 1$)
	|	`[` $TypedExpr_1$ `,` ... `,` $TypedExpr_n$ `]`	(*finite list*, $n > 0$)
	|	`[` *TypedExpr* [`,` *TypedExpr*] `..` [*TypedExpr*] `]`	(*arithmetic sequence*)
	|	`[` *TypedExpr* `|` $Qual_1$ `,` ... `,` $Qual_n$ `]`	(*list comprehension*, $n \geq 1$)
	|	`(` *InfixExpr QOp* `)`	(*left section*)
	|	`(` $QOp_{\langle -,- \rangle}$ *InfixExpr* `)`	(*right section*)
	|	`{` $FBind_1$ `,` ... `,` $FBind_n$ `}`	(*labeled construction*, $n > 0$)
	|	`{` $FBind_1$ `,` ... `,` $FBind_n$ `|` *TypedExpr* `}`	(*labeled update*, $n > 0$)
Alt	::=	*Pattern* `->` *TypedExpr* [`where` *LocalDefs*]	
	|	*Pattern GdAlts* [`where` *LocalDefs*]	
GdAlts	::=	`|` *TypedExpr* `->` *TypedExpr* [*GdAlts*]	
FBind	::=	*LabelID* `:=` *TypedExpr*	
Qual	::=	*TypedExpr*	
	|	`let` *LocalDefs*	
	|	*Pattern* `<-` *TypedExpr*	
Stmt	::=	*TypedExpr*	
	|	`let` *LocalDefs*	
	|	*Pattern* `<-` *TypedExpr*	
Literal	::=	*Int* | *Char* | *String* | *Float*	
GDataConstr	::=	`()`	
	|	`[]`	
	|	`(,{,})`	
	|	*QDataConstr*	
FunctionName	::=	*FunctionID* | `(`*InfixOpID*`)`	(*function*)
QFunctionName	::=	*QFunctionID* | `(`*QInfixOpID*`)`	(*qualified function*)
Variable	::=	*VariableID* | `(`*InfixOpID*`)`	(*variable*)

DataConstr	::=	*DataConstrID* | `(`*InfixOpID*`)`	(*constructor*)
QDataConstr	::=	*QDataConstrID* | `(`*QConsOp*`)`	(*qualified constructor*)
QFunOp	::=	*QInfixOpID* | ‘ *QFunctionID* ‘	(*qualified function operator*)
ConsOp	::=	*InfixOpID* | ‘ *DataConstrID* ‘	(*constructor operator*)
QOp	::=	*QFunOp* | *QConsOp*	(*qualified operator*)
QConsOp	::=	*GConSym* | ‘ *QDataConstrID* ‘	(*qualified constructor operator*)
GConSym	::=	`:` | *QInfixOpID*	(*general constructor symbol*)

A.2. Syntax der Interfaces

Zu den Schlüsselwörtern, die auch als Bezeichner benutzt werden dürfen, kommen „`hiding`“, „`interface`“, „`interfaceTypeClasses`“ und „`public`“ hinzu.

Die Syntax der Interfaces lautet:

Interface	::=	(`interface` | `interfaceTypeClasses`) *ModuleId* `where`	
		`{` *ImportDecl*$_1$ `;` ... `;` *ImportDecl*$_n$ `;`	($n \geq 0$)
		InterfaceDecl$_1$ `;` ... `;` *InterfaceDecl*$_m$ `}`	($m \geq 0$)
ImportDecl	::=	`import` *ModuleId*	
InterfaceDecl	::=	*HidingDataDecl*	
	|	*DataDecl*	
	|	*TypeDecl*	
	|	*FixityDecl*	
	|	*FunctionDecl*	
	|	*ClassDecl*	
	|	*InstanceDecl*	
HidingDataDecl	::=	`hiding data` *QTypeConstrID*	
DataDecl	::=	`data` *QTypeConstrID* `=`	
		ConstrDeclOrHidden$_1$ `|` ... `|` *ConstrDeclOrHidden*$_n$	($n \geq 0$)
ConstrDeclOrHidden	::=	*ConstrDecl* | `_`	
ConstrDecl	::=	see grammar of Curry	
TypeDecl	::=	`type` *QComposedID* `=` (*TypeExpr* | *RecordType*)	
TypeExpr	::=	see grammar of Curry	
RecordType	::=	see grammar of Curry	
FixityDecl	::=	*FixityKeyword Digit InfixOpID*	
FixityKeyword	::=	see grammar of Curry	
InfixOpID	::=	see lexicon	
FunctionDecl	::=	*QComposedID Arity* `::` `(` *Context* `)` `=>` *TypeExpr*	
Arity	::=	*Int*	
Context	::=	*Constraint*$_1$ `,` ... `,` *Constraint*$_n$	($n \geq 0$)
Constraint	::=	see grammar of Curry	
ClassDecl	::=	[`hiding`] `class` *Superclasses QTypeClassID BTypeVarID* `where`	
		`{` *ClassMethods* `}` *DefaultMethods Dependencies*	
Superclasses	::=	`[` *QTypeClassID*$_1$ `,` ... `,` *QTypeClassID*$_n$ `]`	($n \geq 0$)
ClassMethods	::=	*ClassMethod*$_1$ `;` ... `;` *ClassMethod*$_n$	($n \geq 0$)

ClassMethod	::=	*Qualification FunctionDecl*	
Qualification	::=	`hiding` | `public`	
DefaultMethods	::=	`[` $FunctionID_1$ `,` ... `,` $FunctionID_n$ `]`	$(n \geq 0)$
Dependencies	::=	`[` $QComposedID_1$ `,` ... `,` $QComposedID_n$ `]`	$(n \geq 0)$
InstanceDecl	::=	`instance [` [*ModuleId*] `] (` *Context* `) =>` *QTypeClassID* `(` *InstanceType* `)` *Dependencies*	
InstanceType	::=	*GeneralTypeConstr* $BTypeVarID_1$ ... $BTypeVarID_n$	$(n \geq 0)$
GeneralTypeConstr	::=	see grammar of Curry	
QComposedID	::=	$QPart_1$ `:_` ... `:_` $QPart_n$	$(n > 0)$
QPart	::=	*QGeneralID* | *QInfixOpID*	
QGeneralID	::=	[*ModuleID* `.`] *Ident*	

A.3. Fallbeispiel für die Implementierung von Typklassen mit Wörterbüchern

A.3.1. Originalcode

Der Code des Fallbeispiels mit Typklassen lautet:

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

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

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

  x < y = x <= y && x /= y
  x >= y = y <= x
  x > y = not (x <= y)

instance Eq Int where
  x == y = primEqualInt x y

instance Ord Int where
  x <= y = primLeqInt x y

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

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

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

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

member :: Eq a => a -> [a] -> Bool
member x [] = False
member x (y:ys) | x == y = True
                | otherwise = member x ys

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | y > x = x:y:ys
                | otherwise = y : insert x ys

example1 = member 1 (insert 3 [2, 4, 6])

example2 = [1] == insert 3 [2, 4]

example3 :: Eq a => a -> a -> Bool
example3 x y = member (1, x) [(1, x), (3, x), (5, y)]
```

A.3.2. Klassen- und Instanztransformation

Der Code nach der Phase 2, also nach dem Transformieren von Klassen- und Instanzdefinitionen, lautet:

```
-- transformation results of Eq-class
type Dict.Eq a = (a -> a -> Bool, a -> a -> Bool)

sel.Eq.== :: Dict.Eq a -> a -> a -> Bool
sel.Eq.== (eq, neq) = eq

sel.Eq./= :: Dict.Eq a -> a -> a -> Bool
sel.Eq./= (eq, neq) = neq

def.Eq./= :: Eq a => a -> a -> Bool
x `def.Eq./=` y = not (x == y)

-- transformation results of Ord-class
type Dict.Ord a =
  (Dict.Eq a,
   a -> a -> Bool,
   a -> a -> Bool,
```

```
   a -> a -> Bool,
   a -> a -> Bool)

sel.Ord.Eq :: Dict.Ord a -> Dict.Eq a
sel.Ord.Eq (eqDict, leq, lt, ge, gt) = eqDict

sel.Ord.<= :: Dict.Ord a -> a -> a -> Bool
sel.Ord.<= (eqDict, leq, lt, ge, gt) = leq

sel.Ord.< :: Dict.Ord a -> a -> a -> Bool
sel.Ord.< (eqDict, leq, lt, ge, gt) = lt

sel.Ord.>= :: Dict.Ord a -> a -> a -> Bool
sel.Ord.>= (eqDict, leq, lt, ge, gt) = ge

sel.Ord.> :: Dict.Ord a -> a -> a -> Bool
sel.Ord.> (eqDict, leq, lt, ge, gt) = gt

def.Ord.< :: Ord a => a -> a -> Bool
x `def.Ord.<` y = x <= y && x /= y

def.Ord.>= :: Ord a => a -> a -> Bool
x `def.Ord.>=` y = y <= x

def.Ord.> :: Ord a => a -> a -> Bool
x `def.Ord.>` y = not (x <= y)

-- transformation results of Eq-Int-instance
impl.Eq.Int.== :: Int -> Int -> Bool
x `impl.Eq.Int.==` y = primEqualInt x y

dict.Eq.Int :: Dict.Eq Int
dict.Eq.Int = (impl.Eq.Int.==, def.Eq./=)

-- transformation results of Ord-Int-instance
impl.Ord.Int.<= :: Int -> Int -> Bool
x `impl.Ord.Int.<=` y = primLeqInt x y

dict.Ord.Int :: Dict.Ord Int
dict.Ord.Int = (dict.Eq.Int, impl.Ord.Int.<=,
  def.Ord.<, def.Ord.>=, def.Ord.>)

-- transformation results of Eq-[]-instance
impl.Eq.[].== :: Eq a => [a] -> [a] -> Bool
[]     `impl.Eq.[].==` []    = True
[]     `impl.Eq.[].==` (_:_) = False
(_:_)  `impl.Eq.[].==` []    = False
(x:xs) `impl.Eq.[].==` (y:ys) = x == y && xs == ys
```

```
dict.Eq.[] :: Eq a => Dict.Eq [a]
dict.Eq.[] = (impl.Eq.[].==, def.Eq./=)

-- transformation results of Eq-(,)-instance
impl.Eq.(,).== :: (Eq a, Eq b) => (a, b) -> (a, b) -> Bool
(x, y) `impl.Eq.(,).==` (x', y') = x == x' && y == y'

dict.Eq.(,) :: (Eq a, Eq b) => Dict.Eq (a, b)
dict.Eq.(,) = (impl.Eq.(,).==, def.Eq./=)

-- transformation results of Ord-[]-instance
impl.Ord.[].<= :: Ord a => [a] -> [a] -> Bool
[]     `impl.Ord.[].<=` []     = True
[]     `impl.Ord.[].<=` (_:_)  = True
(_:_)  `impl.Ord.[].<=` []     = False
(x:xs) `impl.Ord.[].<=` (y:ys) = x < y || (x == y && xs <= ys)

dict.Ord.[] :: Ord a => Dict.Ord [a]
dict.Ord.[] = (dict.Eq.[], impl.Ord.[].<=,
  def.Ord.<, def.Ord.>=, def.Ord.>)

-- transformation results of Ord-(,)-instance
impl.Ord.(,).<= :: (Ord a, Ord b) => (a, b) -> (a, b) -> Bool
(x, y) `impl.Ord.(,).<=` (x', y') =
  x < x' || (x == x' && y <= y')

dict.Ord.(,) :: (Ord a, Ord b) => Dict (a, b)
dict.Ord.(,) = (dict.Eq.(,), impl.Ord.(,).<=,
  def.Ord.<, def.Ord.>=, def.Ord.>)

member :: Eq a => a -> [a] -> Bool
member x [] = False
member x (y:ys) | x == y = True
                | otherwise = member x ys

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | y > x = x:y:ys
                | otherwise = y : insert x ys

example1 = member 1 (insert 3 [2, 4, 6])

example2 = [1] == insert 3 [2, 4]

example3 :: Eq a => a -> a -> Bool
example3 x y = member (1, x) [(1, x), (3, x), (5, y)]
```

A.3.3. Einfügen von Wörterbüchern

Nach dem Einfügen von Wörterbüchern lautet der Code:

```
-- Transformation results of Eq-class
type Dict.Eq a = (a -> a -> Bool, a -> a -> Bool)

sel.Eq.== :: Dict.Eq a -> a -> a -> Bool
sel.Eq.== (eq, neq) = eq

sel.Eq./= :: Dict.Eq a -> a -> a -> Bool
sel.Eq./= (eq, neq) = neq

def.Eq./= :: Dict.Eq a -> a -> a -> Bool
def.Eq./= dict.Eq.a x y = not (sel.Eq.(==) dict.Eq.a x y)

-- Transformation results of Ord-class
type Dict.Ord a =
  (Dict.Eq a,
   a -> a -> Bool,
   a -> a -> Bool,
   a -> a -> Bool,
   a -> a -> Bool)

sel.Ord.Eq :: Dict.Ord a -> Dict.Eq a
sel.Ord.Eq (eqDict, leq, lt, ge, gt) = eqDict

sel.Ord.<= :: Dict.Ord a -> a -> a -> Bool
sel.Ord.<= (eqDict, leq, lt, ge, gt) = leq

sel.Ord.< :: Dict.Ord a -> a -> a -> Bool
sel.Ord.< (eqDict, leq, lt, ge, gt) = lt

sel.Ord.>= :: Dict.Ord a -> a -> a -> Bool
sel.Ord.>= (eqDict, leq, lt, ge, gt) = ge

sel.Ord.> :: Dict.Ord a -> a -> a -> Bool
sel.Ord.> (eqDict, leq, lt, ge, gt) = gt

def.Ord.< :: Dict.Ord a -> a -> a -> Bool
def.Ord.< dict.Ord.a x y =
  sel.Ord.<= dict.Ord.a x y &&
  sel.Eq./= (sel.Ord.Eq dict.Ord.a) x y

def.Ord.>= :: Dict.Ord a -> a -> a -> Bool
def.Ord.>= dict.Ord.a x y = sel.Ord.<= dict.Ord.a y x

def.Ord.> :: Dict.Ord a -> a -> a -> Bool
def.Ord.> dict.Ord.a x y = not (sel.Ord.<= dict.Ord.a x y)
```

```
-- transformation results of Eq-Int-instance
impl.Eq.Int.== :: Int -> Int -> Bool
x `impl.Eq.Int.==` y = primEqualInt x y

dict.Eq.Int :: Dict.Eq Int
dict.Eq.Int = (impl.Eq.Int.==, def.Eq./= dict.Eq.Int)

-- transformation results of Ord-Int-instance
impl.Ord.Int.<= :: Int -> Int -> Bool
x `impl.Ord.Int.<=` y = primLeqInt x y

dict.Ord.Int :: Dict.Ord Int
dict.Ord.Int = (dict.Eq.Int, impl.Ord.Int.<=,
  def.Ord.<  dict.Ord.Int,
  def.Ord.>= dict.Ord.Int,
  def.Ord.>  dict.Ord.Int)

-- transformation results of Eq-[]-instance
impl.Eq.[].== :: Dict.Eq a -> [a] -> [a] -> Bool
impl.Eq.[].== dict.Eq.a []     []    = True
impl.Eq.[].== dict.Eq.a []     (_:_) = False
impl.Eq.[].== dict.Eq.a (_:_)  []    = False
impl.Eq.[].== dict.Eq.a (x:xs) (y:ys) =
  sel.Eq.== dict.Eq.a x y &&
  sel.Eq.== (dict.Eq.[] dict.Eq.a) xs ys

dict.Eq.[] :: Dict.Eq a -> Dict.Eq [a]
dict.Eq.[] dict.Eq.a =
  (impl.Eq.[].== dict.Eq.a,
   def.Eq./= (dict.Eq.[] dict.Eq.a))

-- transformation results of Eq-(,)-instance
impl.Eq.(,).== :: Dict.Eq a -> Dict.Eq b
               -> (a, b) -> (a, b) -> Bool
impl.Eq.(,).== dict.Eq.a dict.Eq.b (x, y) (x', y') =
  sel.Eq.== dictEq.a x x' &&
  sel.Eq.== dictEq.b y y'

dict.Eq.(,) :: Dict.Eq a -> Dict.Eq b -> Dict.Eq (a, b)
dict.Eq.(,) dict.Eq.a dict.Eq b =
  (impl.Eq.(,).== dict.Eq.a dict.Eq.b,
   def.Eq./= (dict.Eq.(,) dict.Eq.a dict.Eq.b))

-- transformation results of Ord-[]-instance
impl.Ord.[].<= :: Dict.Ord a -> [a] -> [a] -> Bool
impl.Ord.[].<= dict.Ord.a []     []     = True
impl.Ord.[].<= dict.Ord.a []     (_:_)  = True
impl.Ord.[].<= dict.Ord.a (_:_)  []     = False
impl.Ord.[].<= dict.Ord.a (x:xs) (y:ys) =
```

```
  sel.Ord.< dict.Ord.a x y ||
    (sel.Eq.== (sel.Ord.Eq dict.Ord.a) x y &&
      sel.Ord.<= (dict.Ord.[] dict.Ord.a) xs <= ys)

dict.Ord.[] :: Dict.Ord a -> Dict.Ord [a]
dict.Ord.[] dict.Ord.a =
  (dict.Eq.[] (sel.Ord.Eq dict.Ord.a),
   impl.Ord.[].<= dict.Ord.a,
   def.Ord.<  (dict.Ord.[] dict.Ord.a),
   def.Ord.>= (dict.Ord.[] dict.Ord.a),
   def.Ord.>  (dict.Ord.[] dict.Ord.a))

-- transformation results of Ord-(,)-instance
impl.Ord.(,).<= :: Dict.Ord a -> Dict.Ord b
                -> (a, b) -> (a, b) -> Bool
impl.Ord.(,).<= dict.Ord.a dict.Ord.b (x, y) (x', y') =
  sel.Ord.< dict.Ord.a x x' ||
    (sel.Eq.== (sel.Ord.Eq dict.Ord.a) x x' &&
      sel.Ord.<= dict.Ord.b y y')

dict.Ord.(,) :: Dict.Ord a -> Dict.Ord b -> Dict (a, b)
dict.Ord.(,) dict.Ord.a dict.Ord.b =
  (dict.Eq.(,) (sel.Ord.Eq dict.Ord.a) (sel.Ord.Eq dict.Ord.b),
   impl.Ord.(,).<= dict.Ord.a dict.Ord.b,
   def.Ord.<  (dict.Ord.(,) dict.Ord.a dict.Ord.b),
   def.Ord.>= (dict.Ord.(,) dict.Ord.a dict.Ord.b),
   def.Ord.>  (dict.Ord.(,) dict.Ord.a dict.Ord.b))

member :: Dict.Eq a -> a -> [a] -> Bool
member dict.Eq.a x [] = False
member dict.Eq.a x (y:ys)
  | sel.Eq.== dict.Eq.a x y = True
  | otherwise = member dict.Eq.a x ys

insert :: Dict.Ord a -> a -> [a] -> [a]
insert dict.Ord.a x [] = [x]
insert dict.Ord.a x (y:ys)
  | sel.Ord.> dict.Ord.a y x = x:y:ys
  | otherwise = y : insert dict.Ord.a x ys

example1 =
  member dict.Eq.Int 1 (insert dict.Ord.Int 3 [2, 4, 6])

example2 =
  sel.Eq.== (dict.Eq.[] dict.Eq.Int)
    [1]
    (insert dict.Ord.Int 3 [2, 4])

example3 :: Dict.Eq a -> a -> a -> Bool
```

```
example3 dict.Eq.a x y =
  member (dict.Eq.(,) dict.Eq.Int dict.Eq.a)
         (1, x) [(1, x), (3, x), (5, y)]
```

A.4. Prelude

Hier ist nicht die gesamte Prelude aufgelistet, sondern es sind nur diejenigen Teile aufgelistet, die sich aufgrund der Einführung von Typklassen geändert haben oder neu hinzugekommen sind.

```
module Prelude
  (
    -- classes and overloaded functions
    Eq(..)
  , elem, notElem, lookup
  , Ord(..)
  , Show(..), print, shows, showChar, showString, showParen
  , Read (..)
  , Bounded (..), Enum (..), boundedEnumFrom, boundedEnumFromThen
  , asTypeOf
  , Num(..), Fractional(..), Real(..), Integral(..)
  -- data types
  , Bool (..) , Char (..) , Int (..) , Float (..), String
  , Ordering (..), Success (..), Maybe (..), Either (..), IO (..)
  , IOError (..)
  -- functions
  , (.), 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,
  , negateFloat, (=:=), success, (&), (&>), maybe
  , either, (>>=), return, (>>), done, putChar, getChar, readFile
  , writeFile, appendFile
  , putStr, putStrLn, getLine, userError, ioError, showError
  , catch, doSolve, sequenceIO, sequenceIO_, mapIO
  , mapIO_, (?), unknown
  , normalForm, groundNormalForm, apply, cond, (=:<=)
  , enumFrom_, enumFromTo_, enumFromThen_, enumFromThenTo_
  ) where

-- ---------------------------------------------------------------
-- Prelude functions
-- ---------------------------------------------------------------
```

```
infixl 9 !!
infixr 9 .
infixl 7 *, `div`, `mod`, /
infixl 6 +, -
-- infixr 5 :                        -- declared together with list
infixr 5 ++
infix  4 =:=, ==, /=, <, >, <=, >=, =:<=
infix  4 `elem`, `notElem`
infixr 3 &&
infixr 2 ||
infixl 1 >>, >>=
infixr 0 $, $!, $!!, $#, $##, `seq`, &, &>, ?

-- externally defined types for numbers and characters
data Int
data Float
data Char
type String = [Char]

...

-- used for comparison of standard types like Int, Float and Char
(==$)            :: a -> a -> Bool
(==$) external

(/=$) :: a -> a -> Bool
x /=$ y = not (x ==$ y)

--- Ordering type. Useful as a result of comparison functions.
data Ordering = LT | EQ | GT
  deriving (Eq, Ord)

-- used for comparison of standard types like Int, Float and Char
(<=$) :: a -> a -> Bool
(<=$) external

(<$) :: a -> a -> Bool
x <$ y = not (y <=$ x)

...

--- Element of a list?
elem :: Eq a => a -> [a] -> Bool
elem x = any (x ==)

--- Not element of a list?
notElem :: Eq a => a -> [a] -> Bool
notElem x = all (x /=)
```

```
--- Looks up a key in an association list.
lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup _ []       = Nothing
lookup k ((x,y):xys)
      | k==x      = Just y
      | otherwise = lookup k xys

--- Generates an infinite sequence of ascending integers.
enumFrom_               :: Int -> [Int]             -- [n..]
enumFrom_ n             = n : enumFrom_ (n+1)

--- Generates an infinite sequence of integers with a
--- particular in/decrement.
enumFromThen_           :: Int -> Int -> [Int]     -- [n1,n2..]
enumFromThen_ n1 n2     = iterate ((n2-n1)+) n1

--- Generates a sequence of ascending integers.
enumFromTo_             :: Int -> Int -> [Int]     -- [n..m]
enumFromTo_ n m         =
  if n>m then [] else n : enumFromTo_ (n+1) m

--- Generates a sequence of integers with a particular
--- in/decrement.
enumFromThenTo_         :: Int -> Int -> Int -> [Int]
enumFromThenTo_ n1 n2 m = takeWhile p (enumFromThen_ n1 n2)
                          where p x | n2 >= n1  = (x <= m)
                                    | otherwise = (x >= m)


--- Converts a character into its ASCII value.
ord :: Char -> Int
ord c = prim_ord $# c

prim_ord :: Char -> Int
prim_ord external

--- Converts an ASCII value into a character.
chr :: Int -> Char
chr n = prim_chr $# n

prim_chr :: Int -> Char
prim_chr external


-- Types of primitive arithmetic functions and predicates

--- Adds two integers.
(+$)   :: Int -> Int -> Int
```

```
(+$) external

--- Subtracts two integers.
(-$)   :: Int -> Int -> Int
(-$) external

--- Multiplies two integers.
(*$)   :: Int -> Int -> Int
(*$) external

--- Integer division. The value is the integer quotient of
--- its argumentsand always truncated towards zero.
--- Thus, the value of <code>13 `div` 5</code> is <code>2</code>,
--- and the value of <code>-15 `div` 4</code> is <code>-3</code>.
div_   :: Int -> Int -> Int
div_ external

--- Integer remainder. The value is the remainder of the integer
--- division and it obeys the rule
--- <code>x `mod` y = x - y * (x `div` y)</code>.
--- Thus, the value of <code>13 `mod` 5</code> is <code>3</code>,
--- and the value of <code>-15 `mod` 4</code> is <code>-3</code>.
mod_   :: Int -> Int -> Int
mod_ external

--- Unary minus on Floats. Usually written as "-e".
negateFloat :: Float -> Float
negateFloat external

...

--- Converts an arbitrary term into an external string
--- representation.
show_    :: _ -> String
show_ x = prim_show $## x

prim_show    :: _ -> String
prim_show external

--- Converts a term into a string and prints it.
print :: Show a => a -> IO ()
print t = putStrLn (show t)

...

-- ---------------------------------------------------------------
-- Eq class and related instances and functions
-- ---------------------------------------------------------------
```

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

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

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

instance Eq Char where
  c == c' = c ==$ c'

instance Eq Int where
  i == i' = i ==$ i'

instance Eq Float where
  f == f' = f ==$ 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 Success where
  _ == _ = True

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

-- ---------------------------------------------------------------------
-- Ord class and related instances and functions
-- ---------------------------------------------------------------------

--- minimal complete definition: compare or <=
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 Bool where
  False <= False = True
  False <= True  = True
  True  <= False = False
  True  <= True  = True

instance Ord Char where
  c1 <= c2 = c1 <=$ c2

instance Ord Int where
  i1 <= i2 = i1 <=$ i2

instance Ord Float where
  f1 <= f2 = f1 <=$ f2

instance Ord Success where
  _ <= _ = True

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')

-- ----------------------------------------------------------------
-- Show class and related instances and functions
-- ----------------------------------------------------------------

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 = 10
appPrec1 = 11

instance Show Success where
  showsPrec _ _ = showString "Success"

-- ---------------------------------------------------------------------
-- Bounded and Enum classes and instances
-- ---------------------------------------------------------------------

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

-- ----------------------------------------------------------------------
-- Numeric classes and instances
-- ----------------------------------------------------------------------

-- minimal definition: all (except negate or (-))
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

-- minimal definition: fromFloat and (recip or (/))
class Num a => Fractional a where

  (/) :: a -> a -> a
  recip :: a -> a

  recip x = 1/x
  x / y = x * recip y
```

```
  fromFloat :: Float -> a

instance Fractional Float where
  x / y = x /. y
  recip x = 1.0/x

  fromFloat x = x

class (Num a, Ord a) => Real a where

class Real a => Integral a where
  div :: a -> a -> a
  mod :: a -> a -> a

  divMod :: a -> a -> (a, a)

  n `div` d = q where (q, _) = divMod n d
  n `mod` d = r where (_, r) = divMod n d

instance Real Int where
  -- no class methods to implement

instance Integral Int where
  divMod n d = (n `div_` d, n `mod_` d)

-- ---------------------------------------------------------------
-- Helper functions
-- ---------------------------------------------------------------

asTypeOf :: a -> a -> a
asTypeOf = const

-- ---------------------------------------------------------------
-- Floating point operations
-- ---------------------------------------------------------------

--- Addition on floats.
(+.)   :: Float -> Float -> Float
x +. y = (prim_Float_plus $# y) $# x

prim_Float_plus :: Float -> Float -> Float
prim_Float_plus external

--- Subtraction on floats.
(-.)   :: 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

--- Division on floats.
(/.)   :: Float -> Float -> Float
x /. y = (prim_Float_div $# y) $# x

prim_Float_div :: Float -> Float -> Float
prim_Float_div external

--- Conversion function from integers to floats.
i2f    :: Int -> Float
i2f x = prim_i2f $# x

prim_i2f :: Int -> Float
prim_i2f external
```

Literaturverzeichnis

[Aug93] Augustsson, Lennart: Implementing Haskell overloading. In: *Proceedings of the conference on Functional programming languages and computer architecture*. New York, NY, USA : ACM, 1993 (FPCA '93). – ISBN 0–89791–595–X, 65–73

[DJH02] Diatchki, Iavor S. ; Jones, Mark P. ; Hallgren, Thomas: A formal specification of the Haskell 98 module system. In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. New York, NY, USA : ACM, 2002 (Haskell '02). – ISBN 1–58113–605–6, 17–28

[DM82] Damas, Luis ; Milner, Robin: Principal type-schemes for functional programs. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1982 (POPL '82). – ISBN 0–89791–065–6, 207–212

[H$^+$] Hanus, Michael u. a.: *KiCS2 – The Kiel Curry System (Version 2)*, `http://www-ps.informatik.uni-kiel.de/kics2/Manual.pdf`

[HB90] Hammond, Kevin ; Blott, Stephen: Implementing Haskell Type Classes. In: *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. London, UK, UK : Springer-Verlag, 1990. – ISBN 3–540–19609–9, 266–286

[He12] Hanus (ed.), M.: *Curry: An Integrated Functional Logic Language (Vers. 0.8.3)*. Available at `http://www.curry-language.org`, 2012

[HHPJW96] Hall, Cordelia V. ; Hammond, Kevin ; Peyton Jones, Simon L. ; Wadler, Philip L.: Type classes in Haskell. In: *ACM Trans. Program. Lang. Syst.* 18 (1996), März, Nr. 2, 109–138. `http://dx.doi.org/10.1145/227699.227700`. – DOI 10.1145/227699.227700. – ISSN 0164–0925

[HHS02] Heeren, Bastiaan ; Hage, Jurriaan ; Swierstra, Doaitse: Generalizing Hindley-Milner Type Inference Algorithms. 2002. – Forschungsbericht

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

[Jon92a] Jones, Mark P.: A theory of qualified types. Version: 1992. `http://dx.doi.org/10.1007/3-540-55253-7_17`. In: Krieg-Brückner, Bernd (Hrsg.): *ESOP '92* Bd. 582. Springer Berlin Heidelberg, 1992. – DOI 10.1007/3–540–55253–7_17. – ISBN 978–3–540–55253–6, 287-306

[Jon92b] Jones, Mark P.: *Qualified types: theory and practice*, University of Oxford, Diss., 1992

[Jon93] Jones, Mark P.: Coherence for Qualified Types, Springer-Verlag, 1993, S. 287–306

[Jon94] Jones, Mark P.: ML typing, explicit polymorphism and qualified types. In: *In TACS '94: Conference on theoretical aspects of computer software*, Springer-Verlag, 1994, S. 56–75

[Jon95a] Jones, Mark P.: Simplifying and improving qualified types. In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture*. New York, NY, USA : ACM, 1995 (FPCA '95). – ISBN 0–89791–719–7, 160–169

[Jon95b] Jones, Mark P.: A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. In: *Journal of functional programming*, ACM Press, 1995, S. 52–61

[Jon00] Jones, Mark P.: *Typing Haskell in Haskell*. `http://web.cecs.pdx.edu/~mpj/thih/`. Version: 2000

[JW91] Jones, Simon L. P. ; Wadler, Philip: A Static Semantics for Haskell. 1991. – Forschungsbericht

[Lux] Lux, W.: *Type-classes and call-time choice vs. run-time choice*. Post to the Curry mailing list. `http://www.informatik.uni-kiel.de/~curry/listarchive/0790.html`

[M+10] Marlow, Simon u. a. ; Marlow, Simon (Hrsg.): Haskell 2010 Language Report / Microsoft Research. Version: 2010. `http://www.haskell.org/definition/haskell2010.pdf`. 2010. – Forschungsbericht

[Mil78] Milner, Robin: A theory of type polymorphism in programming. In: *Journal of Computer and System Sciences* 17 (1978), S. 348–375

[MM11] Martin-Martin, Enrique: Type classes in functional logic programming. In: *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*. New York, NY, USA : ACM, 2011 (PEPM '11). – ISBN 978–1–4503–0485–6, 121–130

[NP93] Nipkow, Tobias ; Prehofer, Christian: Type checking type classes. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1993 (POPL '93). – ISBN 0–89791–560–7, 409–418

[NP95] Nipkow, Tobias ; Prehofer, Christian: Type Reconstruction for Type Classes. In: *Journal of Functional Programming* 5 (1995), Nr. 2, S. 201–224

[NS91] Nipkow, Tobias ; Snelting, Gregor: Type classes and overloading resolution via order-sorted unification. In: *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*. New York, NY, USA : Springer-Verlag New York, Inc., 1991. – ISBN 0–387–54396–1, 1–14

[P+96] Peterson, John u. a.: Report on the Programming Language Haskell / Various institutions. Version: May 1996. `http://haskell.org/definition/haskell-report-1.3.ps.gz`. 1996. – Forschungsbericht

[PJ93] Peterson, John ; Jones, Mark: Implementing type classes. In: *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. New York, NY, USA : ACM, 1993 (PLDI '93). – ISBN 0–89791–598–4, 227–236

[Rob65] Robinson, J. A.: A Machine-Oriented Logic Based on the Resolution Principle. In: *J. ACM* 12 (1965), Januar, Nr. 1, 23–41. `http://dx.doi.org/10.1145/321250.321253`. – DOI 10.1145/321250.321253. – ISSN 0004–5411

[The13] The GHC Team: *The Glorious Glasgow Haskell Compilation System User's Guide*, April 2013. (7.6.3) . `http://www.haskell.org/ghc/docs/latest/users_guide.pdf`

[WB89] Wadler, P. ; Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1989 (POPL '89). – ISBN 0–89791–294–2, 60–76