Funktionale Programmierung

Jürgen Giesl

Sommersemester 2016

Lehr- und Forschungsgebiet Informatik 2

RWTH Aachen

© Jürgen Giesl, giesl@informatik.rwth-aachen.de

Inhaltsverzeichnis

0 Einleitung **4**

1 Einführung in die Sprache HASKELL **10**

1.1 Grundlegende Sprachkonstrukte 10

1.1.1 Deklarationen 10

1.1.2 Ausdrücke 21

1.1.3 Patterns 25

1.1.4 Typen 28

1.2 Funktionen höherer Ordnung 37

1.3 Programmieren mit Lazy Evaluation 48

1.4 Monaden 53

1.4.1 Ein- und Ausgabe mit Monaden 53

1.4.2 Programmieren mit Monaden 58

2 Semantik funktionaler Programme **68**

2.1 Vollständige Ordnungen und Fixpunkte 68

2.1.1 Partiell definierte Werte 69

2.1.2 Monotone und stetige Funktionen 72

2.1.3 Fixpunkte 81

2.2 Denotationelle Semantik von HASKELL 84

2.2.1 Konstruktion von Domains 85

2.2.2 Semantik einfacher HASKELL-Programme 90

2.2.3 Semantik komplexer HASKELL-Programme 96

3 Der Lambda-Kalkül **111**

3.1 Syntax des Lambda-Kalküls 112

3.2 Reduktionsregeln des Lambda-Kalküls 114

3.3 Reduzierung von HASKELL auf den Lambda-Kalkül 118

3.4 Der reine Lambda-Kalkül 124

4 Typüberprüfung und -inferenz **127**

4.1 Typschemata und Typannahmen 127

4.2 Der Typinferenzalgorithmus 129

4.2.1 Typinferenz bei Variablen und Konstanten 130

4.2.2 Typinferenz bei Lambda-Abstraktionen 130

4.2.3 Typinferenz bei Applikationen 133
4.2.4 Der gesamte Typinferenzalgorithmus 136
4.3 Typinferenz bei HASKELL-Programmen 138

Kapitel 0

Einleitung

Für Informatiker ist die Kenntnis verschiedener Familien von Programmiersprachen aus mehreren Gründen nötig:

- Die Vertrautheit mit unterschiedlichen Konzepten von Programmiersprachen ermöglicht es, eigene Ideen bei der Entwicklung von Software besser auszudrücken.
- Das Hintergrundwissen über verschiedene Programmiersprachen ist nötig, um in konkreten Projekten jeweils die am besten geeignete Sprache auszuwählen.
- Wenn man bereits einige konzeptionell verschiedene Programmiersprachen erlernt hat, ist es sehr leicht, sich später weitere Programmiersprachen schnell anzueignen.
- Es ist auch die Aufgabe von Informatikern, neue Programmiersprachen zu entwerfen. Dies kann nur auf der Grundlage der bereits entwickelten Sprachen geschehen.

Generell unterscheiden wir grundsätzlich zwischen *imperativen* und *deklarativen* Programmiersprachen (wobei sich deklarative Sprachen weiter in funktionale und logische Sprachen unterteilen). In imperativen Sprachen setzen sich die Programme aus einer Folge von nacheinander ausgeführten Anweisungen zusammen, die die Werte der Variablen im Speicher verändern. Die meisten der heute verwendeten Programmiersprachen beruhen auf diesem Prinzip, das auch einen direkten Zusammenhang zu der klassischen Rechnerarchitektur besitzt, die auf John von Neumann zurückgeht.

In der deklarativen Programmierung bestehen die Programme hingegen aus einer Spezifikation dessen, *was* berechnet werden soll. Die Festlegung, *wie* die Berechnung genau verlaufen soll, wird dem Interpreter bzw. dem Compiler überlassen. Deklarative Programmiersprachen sind daher problemorientiert statt maschinenorientiert.

Einerseits sind die verschiedenen Programmiersprachen alle "gleichmächtig", d.h., jedes Programm lässt sich prinzipiell in jeder der üblicherweise verwendeten Sprachen schreiben. Andererseits sind die Sprachen aber unterschiedlich gut für verschiedene Anwendungsbereiche geeignet. So werden imperative Sprachen wie C beispielsweise für schnelle maschinennahe Programmierung eingesetzt, da dort der Programmierer direkt die Verwaltung des Speichers übernehmen kann (und muss). In anderen Programmiersprachen wird diese Aufgabe automatisch (vom Compiler) durchgeführt. Dies erlaubt eine schnellere Programmentwicklung, die weniger fehleranfällig ist. Andererseits sind die dabei entstehenden Programme

meist auch weniger effizient (d.h., sie benötigen mehr Zeit und Speicherplatz). Funktionale Sprachen werden vor allem für die Prototypentwicklung, im Telekommunikationsbereich, aber auch für Multimediaanwendungen verwendet.

Das folgende Beispiel soll den Unterschied zwischen imperativen und funktionalen Programmiersprachen verdeutlichen. Wir benutzen hierfür zwei typische Sprachen, nämlich JAVA und HASKELL. Zur Illustration dieser Programmierparadigmen betrachten wir den Algorithmus zur Berechnung der Länge einer Liste. Die Eingabe des Algorithmus ist also eine Liste wie z.B. [15, 70, 36] und die Ausgabe des Algorithmus soll die Länge dieser Liste sein (in unserem Beispiel also 3). Ein imperativer Algorithmus zur Lösung dieser Aufgabe lässt sich leicht angeben.

```
class Element {

  Data    value;
  Element next;
}
```

```
class List {
  Element head;

  static int len (List l) {
    int n = 0;

    while (l.head != null) {
      l.head = l.head.next;
      n = n + 1;
    }
    return n;
  }
}
```

Anhand dieses Programms kann man die folgenden Beobachtungen machen:

- Das Programm besteht aus einzelnen Anweisungen, die nacheinander abgearbeitet werden. Hierzu existieren verschiedene Kontrollstrukturen (Bedingungen, Schleifen, etc.), um den Programmablauf zu steuern.
- Die Abarbeitung einer Anweisung ändert die Werte der Variablen im Speicher. Jede Anweisung kann daher Seiteneffekte auslösen. Beispielsweise ändert sich im Verlauf der Ausführung von `len` sowohl der Wert von `n` als auch von `l`. Letzteres hat auch außerhalb der Methode `len` Auswirkungen. Bei der Berechnung von `len(l)` wird auch der Wert des Objekts, auf das `l` zeigt, geändert. Man erhält also nicht nur einen `int`-Wert als Ergebnis, sondern als *Seiteneffekt* wird `l` dabei verändert (zum Schluss ist `l.head = null`), d.h., die Liste wird beim Berechnen der Länge geleert.
- Der Programmierer muss sich Gedanken über die Realisierung und die Speicherverwaltung bei nicht-primitiven Datentypen (wie Listen) machen. Beispielsweise ist der obige Seiteneffekt evtl. nicht gewollt. Um dies zu vermeiden, muss der Programmierer aber erwünschte und unerwünschte Seiteneffekte voraussehen. Die Seiteneffekte und die explizite Speicherverwaltung in imperativen Programmen führen daher oft zu schwer lokalisierbaren Fehlern.

Nachdem wir die Konzepte des imperativen Programmierens illustriert haben, kommen wir nun zu deklarativen Programmiersprachen, bei denen das Programm beschreibt, was

berechnet werden soll, aber die genaue Festlegung, wie die Berechnung verlaufen soll, dem Compiler oder Interpreter überlässt. Unser Ziel ist wieder, die Länge einer Liste zu berechnen. Die folgende Beschreibung macht deutlich, was die Länge `len` einer Liste `l` bedeutet:

(A) Falls die Liste `l` leer ist, so ist `len(l) = 0`.

(B) Falls die Liste `l` nicht leer ist und "`xs`" die Liste `l` ohne ihr erstes Element ist, so ist `len(l) = 1 + len(xs)`.

Wir schreiben im Folgenden `x:xs` für die Liste, die aus der Liste `xs` entsteht, indem man das Element (bzw. den Wert) `x` vorne einfügt. Wir haben also `15:[70,36] = [15,70,36]`. Jede nicht-leere Liste kann man daher als `x:xs` darstellen, wobei `x` das erste Element der Liste ist und `xs` die verbleibende Liste ohne ihr erstes Element. Wenn wir die leere Liste mit `[]` bezeichnen, so haben wir `15:70:36:[] = [15,70,36]` (wobei das Einfügen mit "`:`" von rechts nach links abgearbeitet wird, d.h., "`:`" assoziiert nach rechts).

Nun lässt sich die obige Spezifikation (Beschreibung) der Funktion `len` direkt in ein funktionales Programm übersetzen. Die Implementierung in der funktionalen Programmiersprache HASKELL lautet wie folgt:

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

Ein funktionales Programm ist eine Folge von Deklarationen. Eine Deklaration bindet eine Variable (wie `len`) an einen Wert (wie die Funktion, die die Länge von Listen berechnet). Die Zeile `len :: [a] -> Int` ist eine *Typdeklaration*, die besagt, dass `len` eine Liste als Eingabe erwartet und eine ganze Zahl als Ausgabe berechnet. Hierbei bezeichnet `a` den Typ der Elemente der Liste. Die Datenstruktur der Listen ist in HASKELL vordefiniert (aber natürlich gibt es die Möglichkeit, weitere Datenstrukturen selbst zu definieren).

Es folgen die definierenden Gleichungen von `len`. Die erste Gleichung gibt an, was das Resultat der Funktion `len` ist, falls `len` auf die leere Liste `[]` angewendet wird. (In HASKELL sind Klammern um das Argument einer Funktion nicht nötig, sie können jedoch auch geschrieben werden. Man könnte also für die erste definierende Gleichung auch `len([]) = 0` und für die zweite Gleichung `len(x:xs) = 1 + len(xs)` schreiben.) Die zweite Gleichung ist anwendbar, wenn das Argument von `len` eine nicht-leere Liste ist. Das Argument hat dann die Form `x:xs`. In diesem Fall wird nun `1 + len xs` berechnet. Man erkennt, dass `len` *rekursiv* definiert ist, d.h., zur Berechnung von `len(x:xs)` muss wiederum `len` (von einem anderen Argument, nämlich `xs`) berechnet werden.

Die Ausführung eines funktionalen Programms besteht in der Auswertung eines Ausdrucks mit Hilfe dieser Funktionen. In unserem Beispiel wird lediglich die Funktion `len` definiert. Um die Arbeitsweise des obigen Algorithmus zu veranschaulichen, betrachten wir die Berechnung von `len [15,70,36]`. Bei Ausführung des Algorithmus wird zunächst überprüft, ob das Argument die leere Liste ist (d.h., ob das Argument `[]` in der ersten definierenden Gleichung auf das aktuelle Argument `[15,70,36]` passt). Diesen Vorgang bezeichnet man als *Pattern Matching*. Da dies nicht der Fall ist, versucht man nun, die

zweite definierende Gleichung anzuwenden. Dies ist möglich, wobei in unserem Beispiel das erste Listenelement `x` der Zahl `15` entspricht und die Restliste `xs` der Liste `[70,36]`. Um `len [15,70,36]` zu berechnen, muss man also `1 + len [70,36]` bestimmen. Da das neue Argument `[70,36]` wieder eine nicht-leere Liste mit dem `x`-Wert `70` und dem `xs`-Wert `[36]` ist, führt dies zu einem erneuten Aufruf von `len` mit dem Argument `[36]`. Schließlich ergibt sich `1 + 1 + 1 + 0` = `3`.

Die wichtigsten Eigenschaften der funktionalen Programmiersprache HASKELL lassen sich wie folgt zusammenfassen:

- **Keine Schleifen**
  In (rein) funktionalen Sprachen gibt es keine Kontrollstrukturen wie Schleifen, sondern stattdessen wird nur Rekursion verwendet.

- **Polymorphes Typsystem**
  Funktionale Programmiersprachen erlauben es üblicherweise, dass eine Funktion wie `len` für Listen von Elementen beliebiger Typen verwendet werden kann. Man bezeichnet dies als *parametrischen* Polymorphismus. Für die Variable `a` im Typ von `len` kann also ein beliebiger Typ von Elementen eingesetzt werden (`a` ist eine *Typvariable*), d.h., `len` arbeitet unabhängig vom Typ der Elemente. Trotzdem garantiert das Typsystem, dass Daten nicht falsch interpretiert werden.

  Durch parametrischen Polymorphismus ergibt sich eine Reduzierung des Programmieraufwands, da entsprechende Funktionen nicht immer wieder neu geschrieben werden müssen.

- **Keine Seiteneffekte**
  Die Reihenfolge der Berechnungen beeinflusst das Ergebnis des Programms nicht. Insbesondere haben Programme keine Seiteneffekte, d.h., der Wert der Parameter ändert sich nicht bei der Ausführung einer Funktion. Das Ergebnis einer Funktion hängt also nur von den Argumenten der Funktion ab und wird eine Funktion mehrmals auf dieselben Argumente angewendet, so ist das Ergebnis immer dasselbe. Dieses Verhalten bezeichnet man als *referentielle Transparenz*.

- **Automatische Speicherverwaltung**
  Explizite Zeigermanipulation sowie Anforderung oder Freigabe des Speicherplatzes werden nicht vom Programmierer durchgeführt.

- **Gleichberechtigte Behandlung von Funktionen als Datenobjekte**
  Funktionen werden als gleichberechtigte Datenobjekte behandelt. Insbesondere können Funktionen also auch Argumente oder Ergebnisse anderer Funktionen sein. Solche Funktionen heißen Funktionen höherer Ordnung.

- **Lazy Evaluation** (Verzögerte Auswertung)
  Zur Auswertung eines Funktionsaufrufs werden nur diejenigen Teile der Argumente ausgewertet, die notwendig für die Berechnung des Ergebnisses sind. (Diese Auswertungsstrategie wird allerdings nicht in allen funktionalen Sprachen verwendet. Beispielsweise arbeiten Sprachen wie ML oder LISP und SCHEME mit sogenannter strik-

ter Auswertungsstrategie, bei der alle Argumente einer Funktion ausgewertet werden müssen, bevor die Funktion selbst angewendet werden kann.)

Insgesamt ergeben sich folgende wichtige Vorteile der funktionalen Programmierung:

- Die Programme sind kürzer, klarer und besser zu warten.
- Bei der Programmierung geschehen weniger Fehler, so dass zuverlässigere Programme entstehen. Funktionale Sprachen haben üblicherweise auch eine klare mathematische Basis und sind besser zur Verifikation geeignet als imperative Programmiersprachen.
- Die Programmentwicklung ist schneller und einfacher. (Dies liegt auch daran, dass der Programmierer sich wesentlich weniger mit der Speicherorganisation befassen muss als in imperativen Sprachen). Beispielsweise hat die Firma *Ericsson* nach Einführung der funktionalen Sprache ERLANG eine um den Faktor 10 – 25 schnellere Programmentwicklungszeit gemessen. Die entstehenden Programme waren um den Faktor 2 – 10 kürzer als zuvor. Insbesondere zeigt dies, dass funktionale Sprachen ideal zur Prototypentwicklung sind. Bei sehr zeitkritischen (Realzeit-)Anwendungen sind aus Effizienzgründen maschinennahe Sprachen oft geeigneter.
- Funktionale Programme sind oft besser wiederzuverwenden und modularer strukturiert.

Der Aufbau der Vorlesung ist wie folgt: In Kapitel 1 wird eine Einführung in die funktionale Programmiersprache HASKELL gegeben. Hierdurch werden die Möglichkeiten funktionaler Sprachen deutlich und man gewinnt einen Eindruck von einer realen funktionalen Programmiersprache.

In den folgenden Kapiteln gehen wir auf die Konzepte und Techniken ein, die funktionalen Programmiersprachen zugrunde liegen, wobei wir uns dabei wieder auf HASKELL beziehen. Im Kapitel 2 zeigen wir, wie man die Semantik solcher Sprachen definieren kann. Hierunter versteht man eine formale Festlegung, welche Bedeutung ein funktionales Programm bzw. ein Ausdruck in einem funktionalen Programm hat (d.h., welches Ergebnis dadurch berechnet wird). Solch eine Festlegung ist nötig, um die Korrektheit von Programmen bestimmen zu können und um zu definieren, was die Konstrukte der Programmiersprache bedeuten. Insbesondere ist es also die Grundlage für jede Implementierung der Sprache, da nur dadurch festgelegt werden kann, wie Interpreter oder Compiler arbeiten sollen.

Anschließend führen wir in Kapitel 3 den sogenannten Lambda-Kalkül ein. Dies ist die Grundsprache, die allen funktionalen Programmiersprachen zugrunde liegt. Diese Programmiersprachen sind lediglich lesbarere Versionen des Lambda-Kalküls. Wir zeigen daher, wie HASKELL auf den Lambda-Kalkül zurückgeführt werden kann. Der Lambda-Kalkül stellt insbesondere eine Möglichkeit dar, um funktionale Programme zu implementieren und auf bestimmte Korrektheitseigenschaften zu überprüfen.

Hierzu stellen wir in Kapitel 4 ein Verfahren vor, das untersucht, ob ein Programm (bzw. der entsprechende Ausdruck im Lambda-Kalkül) korrekt getypt ist. Mit anderen Worten, wir überprüfen, ob Funktionen immer nur auf Argumente der richtigen Sorte angewendet werden. Diese Überprüfung wird in Interpretern oder Compilern als erster Schritt vor der Ausführung eines funktionalen Programms durchgeführt. Wie erwähnt, besitzt HASKELL ein

polymorphes Typsystem. Dadurch kann man z.B. die Berechnung der Länge von Listen von Zahlen, von Listen von Zeichen, von Listen von Listen etc. mit ein- und derselben Funktion `len` realisieren, was ein hohes Maß an Wiederverwendbarkeit erlaubt. Andererseits wird aus diesem Grund die Typprüfung nicht-trivial.

Ich danke René Thiemann, Peter Schneider-Kamp, Carsten Fuhs, Darius Dlugosz und Diego Biurrun für ihre konstruktiven Kommentare und Vorschläge beim Korrekturlesen des Skripts.

Kapitel 1

Einführung in die funktionale Programmiersprache HASKELL

In diesem Kapitel geben wir eine Einführung in die Sprache HASKELL. Hierbei werden wir die Syntax der Sprache vorstellen und die Bedeutung der Sprachkonstrukte informell erklären. Eine formale Definition der Semantik der Sprache folgt in Kapitel 2. Für weitere Beschreibungen der Sprache HASKELL sei auf [Thi94, Bir98, PJH98, Tho99, HPF00, Hud00, PJ00, Pep02] verwiesen. Weitere Informationen zu HASKELL findet man auf der HASKELL-Homepage (`http://www.haskell.org`). Hier ist auch der HASKELL-Compiler und -Interpreter GHC innerhalb der *Haskell Platform* erhältlich.

Wir stellen zunächst in Abschnitt 1.1 die grundlegenden Sprachkonstrukte von HASKELL vor. Anschließend gehen wir auf funktionale Programmiertechniken ein. Hierzu betrachten wir in Abschnitt 1.2 Funktionen höherer Ordnung, d.h. Funktionen, die wiederum Funktionen verarbeiten. In Abschnitt 1.3 zeigen wir, wie man mit Lazy Evaluation programmiert und dabei unendliche Datenobjekte verwenden kann. Schließlich gehen wir in Abschnitt 1.4 auf das Konzept der Monaden ein, die insbesondere zur Ein- und Ausgabe in HASKELL verwendet werden.

1.1 Grundlegende Sprachkonstrukte

In diesem Abschnitt führen wir die grundlegenden Sprachkonstrukte von HASKELL (Deklarationen, Ausdrücke, Patterns und Typen) ein.

1.1.1 Deklarationen

Ein Programm in HASKELL ist eine Folge von Deklarationen. Die Deklarationen müssen linksbündig untereinander stehen. Der Grund dafür wird später klar, wenn wir lokale Deklarationen betrachten, die eingerückt (bzw. in der gleichen Zeile) stehen.

Eine *Deklaration* ist (im einfachsten Fall) eine Beschreibung einer Funktion. Funktionen sind gekennzeichnet durch ein Funktionssymbol (den Namen der Funktion), den Definitionsbereich, den Wertebereich des Resultats und eine Abbildungsvorschrift. Den Definitionsbereich und den Wertebereich legt man in einer sogenannten *Typdeklaration* fest und

die Abbildungsvorschrift wird in einer *Funktionsdeklaration* beschrieben. Die Syntax von Deklarationen ist daher durch folgende kontextfreie Grammatik gegeben.

$$\underline{\mathsf{decl}} \;\rightarrow\; \underline{\mathsf{typedecl}} \mid \underline{\mathsf{fundecl}}$$

Im Folgenden werden wir Nichtterminalsymbole immer durch Unterstreichung kenntlich machen. Außerdem werden wir mit einer Teilmenge von HASKELL beginnen und die Grammatikregeln sukzessive erweitern. (Einige in der HASKELL-Syntax erlaubte Programme werden wir nicht berücksichtigen — die komplette Grammatik für HASKELL-Programme findet sich in [PJH98].)

Als *Kommentare* werden in HASKELL Texte betrachtet, die zwischen `{-` und `-}` eingeschlossen sind sowie jeglicher Text zwischen `--` und dem Zeilenende.

Typdeklarationen

Als Funktionssymbole dienen in HASKELL Variablenbezeichner. Für eine Funktion zur Quadrierung von Zahlen kann z.B. der Name `square` verwendet werden. Eine Deklaration bindet eine Variable (wie `square`) an einen Wert (wie die Funktion, die Zahlen quadriert). Dann kann man die folgende Typdeklaration für `square` angeben.

```
square :: Int -> Int
```

Das erste `Int` beschreibt den Definitionsbereich und das zweite `Int` beschreibt den Wertebereich von `square`. Der Typ `Int` ist dabei in HASKELL vordefiniert. Die Deklaration $\underline{\mathsf{var}}$ `::` $\underline{\mathsf{type}}$ bedeutet, dass die Variable $\underline{\mathsf{var}}$ den Typ $\underline{\mathsf{type}}$ hat. Mit Hilfe von "`->`" wird ein Funktionstyp definiert (d.h., `Int -> Int` ist der Typ der Funktionen, die ganze Zahlen in ganze Zahlen abbilden). Als weiteres Beispiel beschreibt `[Int]` den Typ der Listen von ganzen Zahlen. Im allgemeinen existiert zu jedem Typ `a` der Typ `[a]` der Listen mit Elementen vom Typ `a`.

Man erhält die folgende Grammatikregel für die Syntax von Typdeklarationen. Hierbei legt eine Typdeklaration den Typ von einer oder mehreren Variablen fest.

$$\underline{\mathsf{typedecl}} \;\rightarrow\; \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n \;::\underline{\mathsf{type}}, \;\text{wobei } n \geq 1$$

Typdeklarationen müssen nicht mit angegeben werden. Sie werden dann durch den Interpreter oder Compiler automatisch berechnet. Allerdings sind Typdeklarationen vorteilhaft für die Verständlichkeit von Programmen und sollten daher normalerweise verwendet werden. Diese Deklarationen werden dann vom Interpreter oder Compiler überprüft.

Variablenbezeichner $\underline{\mathsf{var}}$ sind beliebige Folgen von Buchstaben und Zahlen (Strings), die mit einem Kleinbuchstaben beginnen (wie z.B. `square`).

Funktionsdeklarationen

Nach der Typdeklaration folgen die definierenden Gleichungen, d.h. die Abbildungsvorschrift. Beispielsweise könnte die Funktionsdeklaration für `square` wie folgt lauten.

```
square x = x * x
```

Die linke Seite einer definierenden Gleichung besteht aus dem Namen der Funktion und der Beschreibung des Arguments und die rechte Seite definiert das Ergebnis der Funktion. Hierbei müssen die Typen der Argumente und der Ergebnisse natürlich zum Typ der Funktion "passen" (d.h., `square` darf sowohl als Argument wie als Ergebnis nur Ausdrücke vom Typ `Int` bekommen). Arithmetische Grundoperationen wie `+`, `*`, `-`, `/`, etc. sowie Vergleichsoperationen wie `==` (für die Gleichheit), `>=`, etc. sind in HASKELL vordefiniert. Ebenso ist auch die Datenstruktur `Bool` mit den Werten `True` und `False` und den Funktionen `not`, `&&` und `||` vordefiniert. Zur Definition solcher häufig verwendeter Funktionen dienen Bibliotheken. Die oben erwähnten Funktionen sind in einer Standardbibliothek (dem sogenannten "Prelude") definiert, das bei jedem Start von HASKELL geladen wird. Allgemein werden Funktionsdeklarationen wie folgt aufgebaut.

$$
\begin{array}{lll}
\underline{\mathsf{fundecl}} & \rightarrow & \underline{\mathsf{funlhs}}\ \underline{\mathsf{rhs}} \\
\underline{\mathsf{funlhs}} & \rightarrow & \underline{\mathsf{var}}\ \underline{\mathsf{pat}} \\
\underline{\mathsf{rhs}} & \rightarrow & = \underline{\mathsf{exp}}
\end{array}
$$

Hierbei steht $\underline{\mathsf{var}}$ für den Funktionsnamen (wie `square`) und $\underline{\mathsf{pat}}$ für das Argument auf der linken Seite der definierenden Gleichung (wie z.B. `x`). Wie solche Argumente im allgemeinen Fall aussehen dürfen, wird in Abschnitt 1.1.3 erläutert. Die rechte Seite einer definierenden Gleichung ist ein beliebiger Ausdruck $\underline{\mathsf{exp}}$ (wie z.B. `x * x`).

Ausführung eines funktionalen Programms

Die Ausführung eines Programms besteht aus der Auswertung von Ausdrücken. Dies geschieht ähnlich wie bei einem Taschenrechner: Der Benutzer gibt (bei einem Interpreter) einen Ausdruck ein und der Rechner wertet ihn aus. Gibt man beispielsweise `42` ein, so wird auch als Ergebnis `42` zurückgegeben. Gibt man `6 * 7` ein, so wird ebenfalls `42` zurückgegeben, denn die Operation `*` ist vordefiniert. Aber auf dieselbe Art und Weise werden auch die benutzerdefinierten Deklarationen für die Auswertung verwendet. Bei der Eingabe von `square 11` erhält man also das Ergebnis `121` und dasselbe Resultat ergibt sich bei der Eingabe von `square (12 - 1)`. Die Bindungspriorität der Funktionsanwendung ist hierbei am höchsten, d.h., bei der Eingabe von `square 12 - 1` erhält man `143`.

Die Auswertung eines Ausdrucks erfolgt durch *Termersetzung* in zwei Schritten:

(1) Der Computer sucht einen Teilausdruck, der mit der linken Seite einer definierenden Gleichung übereinstimmt, wobei hierbei die Variablen der linken Seite durch geeignete Ausdrücke ersetzt werden müssen. Solch einen Teilausdruck bezeichnet man als *Redex* (für "reducible expression").

(2) Der Redex wird durch die rechte Seite der definierenden Gleichung ersetzt, wobei die Variablen in der rechten Seite genauso wie in (1) belegt werden müssen.

Diese Auswertungsschritte werden solange wiederholt, bis kein Ersetzungsschritt mehr möglich ist.

In Abb. 1.1 sind alle Möglichkeiten zur Auswertung des Ausdrucks `square (12 - 1)` dargestellt. Jeder Pfad durch das Diagramm entspricht einer möglichen Folge von Auswertungsschritten. Eine *Auswertungsstrategie* ist ein Algorithmus zur Auswahl des nächsten Redex.

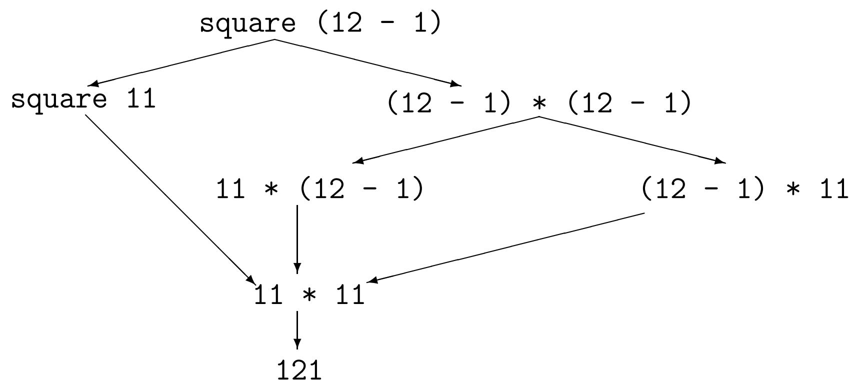


Abbildung 1.1: Auswertung eines Ausdrucks

Insbesondere unterscheiden wir zwischen *strikter* und *nicht-strikter* Auswertung. Bei strikter Auswertung wird stets der am weitesten innen links im Ausdruck vorkommende Redex gewählt. Dies entspricht dem linkesten Pfad durch das Diagramm in Abb. 1.1. Diese Strategie bezeichnet man auch als *leftmost innermost* oder *call-by-value* Strategie oder als *eager evaluation*.

Bei der nicht-strikten Auswertung wird der am weitesten außen links im Ausdruck auftretende Redex gewählt. Die Argumente von Funktionen sind nun in der Regel unausgewertete Ausdrücke. Dies entspricht dem mittleren Pfad durch das Diagramm in Abb. 1.1. Diese Strategie wird auch als *leftmost outermost* oder *call-by-name* Strategie bezeichnet.

Beide Strategien haben Vor- und Nachteile. Bei der nicht-strikten Auswertung werden nur die Teilausdrücke ausgewertet, deren Wert zum Endergebnis beiträgt, was bei der strikten Auswertung nicht der Fall ist. Andererseits muss die nicht-strikte Strategie manchmal denselben Wert mehrfach auswerten, obwohl dies in der strikten Strategie nicht nötig ist (dies geschieht hier mit dem Teilausdruck `12 - 1`).

HASKELL verfolgt das Prinzip der sogenannten *Lazy Evaluation* (verzögerte Auswertung), das beide Vorteile zu kombinieren versucht. Hierbei wird die nicht-strikte Auswertung verfolgt, jedoch werden doppelte Teilausdrücke nicht doppelt ausgewertet, wenn sie aus dem gleichen Ursprungsterm entstanden sind. Im obigen Beispiel würde der Teilterm $12 - 1$ z.B. durch einen Zeiger auf die gleiche Speicherzelle realisiert werden und damit nur einmal ausgewertet werden.

Beim Vergleich der Auswertungsstrategien erhält man das folgende wichtige Resultat: Wenn irgendeine Auswertungsstrategie terminiert, so terminiert auch die nicht-strikte Auswertung (aber nicht unbedingt die strikte Auswertung). Außerdem gilt für alle Strategien: Wenn die Berechnung endet, dann ist das Ergebnis unabhängig von der Strategie gleich. Die Strategien haben also nur Einfluss auf das Terminierungsverhalten, aber nicht auf das Ergebnis. Als Beispiel betrachten wir die folgenden Funktionen.

```
three :: Int -> Int        non_term :: Int -> Int
three x = 3                non_term x = non_term (x+1)
```

Die Auswertung der Funktion `non_term` terminiert für kein Argument. Die strikte Auswertung des Ausdrucks `three (non_term 0)` würde daher ebenfalls nicht terminieren. In

HASKELL wird dieser Ausdruck hingegen zum Ergebnis `3` ausgewertet. Weitere Vorteile der nicht-strikten Strategie werden wir später in Abschnitt 1.3 kennen lernen.

Bedingte definierende Gleichungen

Natürlich will man auch mehrstellige Funktionen und bedingte definierende Gleichungen verwenden. Hierzu betrachten wir eine Funktion `maxi` mit folgender Typdeklaration.

```
maxi :: (Int, Int) -> Int
```

Hierbei bezeichnet `(Int, Int)` das kartesische Produkt der Typen `Int` und `Int` (dies entspricht also der mathematischen Notation `Int` $\times$ `Int`). `(Int, Int) -> Int` ist demnach der Typ der Funktionen, die Paare von ganzen Zahlen auf ganze Zahlen abbilden. Die Funktionsdeklaration von `maxi` lautet wie folgt.

```
maxi(x, y) | x >= y     = x
           | otherwise  = y
```

Der Ausdruck auf der rechten Seite einer definierenden Gleichung kann also durch eine Bedingung (d.h. einen Ausdruck vom Typ `Bool`) eingeschränkt werden. Zur Auswertung verwendet man dann die erste Gleichung, deren Bedingung erfüllt ist (die Fallunterscheidung in den Gleichungen muss aber nicht vollständig sein). Der Ausdruck `otherwise` ist eine vordefinierte Funktion, die immer `True` liefert. Also muss die Grammatikregel für die Bildung von rechten Seiten <u>rhs</u> definierender Gleichungen nun wie folgt geändert werden:

$$\begin{array}{lll} \underline{\mathsf{rhs}} & \rightarrow & = \underline{\mathsf{exp}} \mid \underline{\mathsf{condrhs}}_1 \ldots \underline{\mathsf{condrhs}}_n, \quad \text{wobei } n \geq 1 \\ \underline{\mathsf{condrhs}} & \rightarrow & \mid \underline{\mathsf{exp}} = \underline{\mathsf{exp}} \end{array}$$

Currying

Um die Anzahl der Klammern in Ausdrücken zu reduzieren (und damit die Lesbarkeit zu verbessern), ersetzt man oftmals Tupel von Argumenten durch eine Folge von Argumenten. Diese Technik ist nach dem Logiker *Haskell B. Curry* benannt, dessen Vorname bereits für den Namen der Programmiersprache HASKELL benutzt wurde. Betrachten wir zur Illustration zunächst eine konventionelle Definition der Funktion `plus`.

```
plus :: (Int, Int) -> Int
plus (x, y) = x + y
```

Stattdessen könnte man nun folgende Definition verwenden:

```
plus :: Int -> (Int -> Int)
plus x y = x + y
```

Eine Überführung der ersten Definition von `plus` in die zweite bezeichnet man als *Currying*. Für den Typ `Int -> (Int -> Int)` könnte man auch einfacher `Int -> Int -> Int` schreiben, denn wir benutzen die Konvention, dass der Funktionsraumkonstruktor `->` nach

rechts assoziiert. Die Funktionsanwendung hingegen assoziiert nach links, d.h., der Ausdruck `plus 2 3` steht für `(plus 2) 3`.

Jetzt bekommt `plus` nacheinander zwei Argumente. Genauer ist `plus` nun eine Funktion, die eine ganze Zahl `x` als Eingabe erhält. Das Ergebnis ist dann die Funktion `plus x`. Dies ist eine Funktion von `Int` nach `Int`, wobei `(plus x) y` die Addition von `x` und `y` berechnet.

Solche Funktionen können also auch mit nur einem Argument aufgerufen werden (dies bezeichnet man auch als *partielle Anwendung*). Die Funktion `plus 1` ist z.B. die Nachfolgerfunktion, die Zahlen um `1` erhöht und `plus 0` ist die Identitätsfunktion auf ganzen Zahlen. Diese Möglichkeit der Anwendung auf eine geringere Zahl an Argumenten ist (neben der Klammerersparnis) der zweite Vorteil des Currying. Insgesamt ändert sich also die Grammatikregel für linke Seiten definierender Gleichungen wie folgt:

$$\underline{\mathsf{funlhs}} \rightarrow \underline{\mathsf{var}}\ \underline{\mathsf{pat}}_1 \ldots \underline{\mathsf{pat}}_n, \quad \text{wobei } n \geq 1$$

Funktionsdefinition durch Pattern Matching

Die Argumente auf der linken Seite einer definierenden Gleichung müssen im allgemeinen keine Variablen sein, sondern sie dürfen beliebige *Patterns* (Muster) sein, die als Muster für den erwarteten Wert dienen. Betrachten wir hierzu die Funktion `und`, die die Konjunktion boolescher Werte berechnet.

```
und :: Bool -> Bool -> Bool
und True  y = y
und False y = False
```

Insbesondere haben wir also jetzt mehrere Funktionsdeklarationen (d.h. definierende Gleichungen) für dasselbe Funktionssymbol.

Hierbei sind `True` und `False` vordefinierte Datenkonstruktoren des Datentyps `Bool`, d.h., sie dienen zum Aufbau der Objekte dieses Datentyps. Konstruktoren beginnen in HASKELL immer mit Großbuchstaben.

Um bei einem Funktionsaufruf `und` $\underline{\mathsf{exp}}_1$ $\underline{\mathsf{exp}}_2$ festzustellen, welche definierende Gleichung anzuwenden ist, testet man der Reihe nach von oben nach unten, welche Patterns zu den aktuellen Argumenten $\underline{\mathsf{exp}}_1$ und $\underline{\mathsf{exp}}_2$ passen (Matching). Die Frage ist also, ob es eine Substitution gibt, die die Variablen der Patterns durch konkrete Ausdrücke ersetzt, so dass dadurch die instantiierten Patterns mit $\underline{\mathsf{exp}}_1$ und $\underline{\mathsf{exp}}_2$ übereinstimmen. In diesem Fall sagt man auch, dass der Pattern $\underline{\mathsf{pat}}_i$ (auf) den Ausdruck $\underline{\mathsf{exp}}_i$ "matcht". Dann wird der Gesamtausdruck zu der entsprechend instantiierten rechten Seite ausgewertet. Beispielsweise wird also `und True True` zu `True` ausgewertet, da bei der Substitution [`y/True`] die Patterns `True` und `y` der ersten definierenden Gleichung mit den aktuellen Argumenten `True` und `True` übereinstimmen.

Da Patterns von oben nach unten ausgewertet werden, ist die Definition von `und` äquivalent zur folgenden alternativen Deklaration.

```
und :: Bool -> Bool -> Bool
und True y = y
und x    y = False
```

Wenn wir eine Funktion

```
unclear :: Int -> Bool
unclear x = not (unclear x)
```

haben, deren Auswertung nicht terminiert, so terminiert die Auswertung von `und False (unclear 0)` dennoch, denn um das Pattern Matching durchzuführen, muss `unclear 0` nicht ausgewertet werden. Hingegen terminieren `und (unclear 0) False` oder `und True (unclear 0)` nicht.

In der Funktion `und` gelingt das Pattern Matching, weil ein Wert vom Typ `Bool` nur mit den Datenkonstruktoren `True` oder `False` gebildet werden kann. Boolesche Werte werden also anhand der folgenden Regel konstruiert.[1]

$$\underline{\texttt{Bool}} \;\rightarrow\; \texttt{True} \mid \texttt{False}$$

Pattern Matching ist jedoch auch bei anderen Datentypen möglich. Um zu zeigen, wie man Pattern Matching bei Listen verwenden kann, betrachten wir wieder den Algorithmus `len`.

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

Die vordefinierte Datenstruktur der Listen hat die Datenkonstruktoren `[]` und `:`, so dass Listen wie folgt gebildet werden:

$$\underline{\texttt{[a]}} \;\rightarrow\; \texttt{[]} \mid \underline{\texttt{a}} : \underline{\texttt{[a]}}$$

Hierbei steht `[]` für die leere Liste und der (Infix-)Konstruktor “`:`” dient zum Aufbau von nicht-leeren Listen. Wie erwähnt steht der Ausdruck `x:xs` für die Liste `xs`, in der vorne das Element `x` eingefügt wurde. Das Element `x` hat hierbei einen Typ `a` und `xs` ist eine Liste von Elementen des Typs `a`. (Die Grammatik gibt also an, wie Listen vom Typ `[a]` gebildet werden.)

Bei der Auswertung von `len [15,70,36]` wird zunächst die Listenkurzschreibweise aufgelöst. Das Argument von `len` ist also `15:(70:(36:[]))`. Nun wird Pattern Matching beginnend mit der ersten definierenden Gleichung durchgeführt. Der erste Datenkonstruktor `[]` passt nicht auf den Konstruktor “`:`”, mit dem das aktuelle Argument gebildet ist. Aber der Pattern der zweiten definierenden Gleichung passt auf diesen Wert, wobei die Substitution [`x`/`15`, `xs`/`70:(36:[])`] verwendet wird. Also wertet dieser Ausdruck im ersten Schritt zu `1 + len (70:(36:[]))` aus, etc.

Analog dazu könnte man auch folgenden Algorithmus definieren:

```
second :: [Int]  -> Int
second []           = 0
second (x : [])     = 0
second (x : y : xs) = y
```

Man darf auch die Listenkurzschreibweise in diesen Patterns verwenden und die zweite Gleichung durch `second [x] = 0` ersetzen. Hierbei sei noch erwähnt, dass in HASKELL keine Vollständigkeit der definierenden Gleichungen gefordert ist.

[1] Die Grammatikregeln für <u>`Bool`</u> und <u>`[a]`</u> dienen hier nur zur Illustration des Pattern Matchings und sind nicht Teil der HASKELL-Sprachdefinition.

Patterndeklarationen

Nicht nur Funktionen, sondern auch andere Werte können in Deklarationen festgelegt werden:

```
pin :: Float
pin = 3.14159

suc :: Int -> Int
suc = plus 1

x0, y0 :: Int
(x0, y0) = (1,2)

x1, y1 :: Int
[x1,y1] = [1,2]

x2 :: Int
y2 :: [Int]
x2:y2 = [1,2]
```

Hierbei ist `Float` der vordefinierte Typ für Gleitkommazahlen.

Im allgemeinen darf einem beliebigem *Pattern* ein Ausdruck zugewiesen werden. Im einfachsten Fall ist ein Pattern eine Variable. Sonst ist es ein Ausdruck wie z.B. `(x0, y0)`, so dass bei einer Zuweisung eines Werts wie `(1,2)` an diesen Ausdruck eindeutig festliegt, welche Werte den einzelnen Variablenbezeichnern zugewiesen werden. Eine Patternbindung darf für jeden Bezeichner nur einmal vorkommen (wohingegen Funktionsbindungen mehrfach — mit verschiedenen Pattern für die Argumente — auftreten dürfen).

Wir erweitern also die Möglichkeiten für Deklarationen $\underline{\mathsf{decl}}$ nun um Patterndeklarationen wie folgt:

$$\begin{array}{rcl} \underline{\mathsf{decl}} & \rightarrow & \underline{\mathsf{typedecl}} \mid \underline{\mathsf{fundecl}} \mid \underline{\mathsf{patdecl}} \\ \underline{\mathsf{patdecl}} & \rightarrow & \underline{\mathsf{pat}}\ \underline{\mathsf{rhs}} \end{array}$$

Lokale Deklarationen

Lokale Deklarationen werden verwendet, um innerhalb einer Deklaration einen weiteren lokalen Deklarationsblock zu erstellen. In jeder rechten Seite einer Funktions- oder Patterndeklaration kann man dazu nach dem Schlüsselwort `where` eine Folge von lokalen Deklarationen angeben, die sich nur auf diese rechte Seite beziehen. Dabei werden äußere Deklarationen der gleichen Bezeichner von der lokalen Deklaration überdeckt. Die Grammatikregeln für $\underline{\mathsf{fundecl}}$ und $\underline{\mathsf{patdecl}}$ werden daher wie folgt geändert. Hierbei bedeuten eckige Klammern in der Grammatik, dass die darin befindlichen Ausdrücke optional sind.

$$\begin{array}{lcl} \underline{\mathsf{fundecl}} & \rightarrow & \underline{\mathsf{funlhs}}\ \underline{\mathsf{rhs}}\ [\mathtt{where}\ \underline{\mathsf{decls}}] \\ \underline{\mathsf{patdecl}} & \rightarrow & \underline{\mathsf{pat}}\ \underline{\mathsf{rhs}}\ [\mathtt{where}\ \underline{\mathsf{decls}}] \\ \underline{\mathsf{decls}} & \rightarrow & \{\underline{\mathsf{decl}}_1; \ldots; \underline{\mathsf{decl}}_n\},\ \text{wobei } n \geq 0 \end{array}$$

Als Beispiel betrachten wir das folgende Programm, das die Lösungen einer quadratischen Gleichung mit Hilfe der folgenden Formel berechnet.

$$ax^2 + bx + c = 0 \Longleftrightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
roots :: Float -> Float -> Float -> (Float, Float)
roots a b c = ((-b - d)/e, (-b + d)/e)
                  where { d = sqrt (b*b - 4*a*c); e = 2*a }
```

Ein wichtiger Vorteil lokaler Deklarationen ist, dass die darin deklarierten Werte nur einmal berechnet werden. Der Aufruf von `roots 1 5 3` erzeugt daher einen Graph

```
((-5 - ^d)/ ^e, (-5 + ^d)/ ^e),
```

wobei `^d` ein Zeiger auf eine Speicherzelle mit dem Ausdruck `sqrt (5*5 - 4*1*3)` und `^e` ein Zeiger auf `2*1` ist. Damit müssen diese beiden Ausdrücke also nur einmal ausgewertet werden und man kann mehrfache Auswertungen der gleichen Ausdrücke vermeiden.

Um Klammern zu vermeiden und die Lesbarkeit zu erhöhen, existiert in HASKELL die sogenannte *Offside-Regel* zur Schreibweise von (lokalen) Deklarationen:

1. Das erste Symbol in einer Sammlung <u>decls</u> von Deklarationen bestimmt den linken Rand des Deklarationsblocks.

2. Eine neue Zeile, die an diesem linken Rand anfängt, ist eine neue Deklaration in diesem Block.

3. Eine neue Zeile, die weiter rechts beginnt als dieser linke Rand, gehört zur selben Deklaration (d.h., sie ist die Fortsetzung der darüberliegenden Zeile). Beispielsweise steht

   ```
   d = sqrt (b*b -
             4*a*c)
   ```

   für

   ```
   d = sqrt (b*b - 4*a*c).
   ```

4. Eine neue Zeile, die weiter links beginnt als der linke Rand, bedeutet, dass der <u>decls</u>-Block beendet ist und sie nicht mehr zu dieser Sammlung von Deklarationen gehört.

Man kann also `decls` auch wie ein eingerücktes Programm schreiben (d.h., als Folge von Deklarationen, die linksbündig untereinander stehen). Beispielsweise ließe sich also die Deklaration von `roots` auch wie folgt schreiben:

```
roots a b c = ((-b - d)/e, (-b + d)/e)
                  where d = sqrt (b*b - 4*a*c)
                        e = 2*a
```

Operatoren und Infixdeklarationen

Manche Funktionen sollten nicht in Präfix-, sondern in Infix-Schreibweise verwendet werden, um die Lesbarkeit von Programmen zu erhöhen. Beispiele hierfür sind `+`, `*`, `==` oder auch der Listenkonstruktor `:`, der verwendet wird, um Elemente in Listen einzufügen. Solche Funktionssymbole nennt man *Operatoren*. Wie bei den Präfix-Symbolen unterscheidet man auch hier zwischen Variablen und Konstruktoren. Letztere erhalten keine Funktionsdeklaration, sondern sie werden verwendet, um Objekte einer Datenstruktur zu repräsentieren. Operatoren werden in HASKELL durch Folgen von Sonderzeichen repräsentiert. Konstruktoroperatoren (wie `:`) beginnen dabei mit einem Doppelpunkt und Variablenoperatoren (wie `+` oder `==`) beginnen mit einem anderen Zeichen.

Jeder Infix-Operator kann durch Klammerung zu einer Präfix-Funktion umgewandelt werden. So kann man "`(+) 2 3`" statt "`2 + 3`" schreiben. Analog kann auch jede zweistellige Präfix-Funktion (mit einem Typ $\underline{\texttt{type}}_1$ `->` $\underline{\texttt{type}}_2$ `->` $\underline{\texttt{type}}_3$) in einen Infix-Operator durch Verwendung von "Backquotes" gewandelt werden. So kann man "`2 ‘plus‘ 3`" statt "`plus 2 3`" schreiben. Die Verwendung von Infix-Operatoren bedeutet insofern wirklich nur eine andere Schreibweise. Wir werden daher in den folgenden Definitionen der Syntax immer nur auf Präfix-Funktionen eingehen. Die Verwendung der alternativen Schreibweise mit Infix-Operatoren ist aber in konkreten Programmen oft hilfreich. Zwei Eigenschaften sind bei Infix-Operatoren wichtig:

1. *Assoziation*
   Betrachten wir den folgenden Algorithmus.

   ```
   divide :: Float -> Float -> Float
   divide x y = x/y
   ```

   In dem Ausdruck

   ```
   36 ‘divide‘ 6 ‘divide‘ 2
   ```

   ist zunächst nicht klar, ob das Ergebnis `3` oder `12` ist. Hierzu muss man festlegen, zu welcher Seite der Operator `‘divide‘` *assoziiert*. Daher kann man bei Infix-Operatoren die Assoziation deklarieren. Falls `divide` nach links assoziieren soll, so fügt man die Deklaration

   ```
   infixl ‘divide‘
   ```

   ein. Dies ist auch der Default für Operatoren in HASKELL. In diesem Fall steht der obige Ausdruck für

   ```
   (36 ‘divide‘ 6) ‘divide‘ 2
   ```

   und das Ergebnis ist somit `3`. Deklariert man hingegen

   ```
   infixr ‘divide‘,
   ```

   so assoziiert `‘divide‘` nach rechts. Der obige Ausdruck steht dann für `36 ‘divide‘ (6 ‘divide‘ 2)`, so dass sich `12` ergibt. Eine dritte Möglichkeit ist die Deklaration

   ```
   infix ‘divide‘.
   ```

Dies bedeutet, dass `` `divide` `` gar keine Assoziation besitzt. Dann würde der Ausdruck `` 36 `divide` 6 `divide` 2 `` zu einer Fehlermeldung führen.

Das Konzept der Assoziation haben wir bereits bei dem Funktionsraumkonstruktor und der Funktionsanwendung kennen gelernt. Wie erwähnt, assoziiert der Funktionsraumkonstruktor `->` nach rechts, d.h., `Int -> Int -> Int` steht für `Int -> (Int -> Int)`. Die Funktionsanwendung assoziiert nach links. Somit würde ein Ausdruck wie `square square 3` für `(square square) 3` stehen, d.h. für einen nicht typkorrekten Ausdruck, der zu einer Fehlermeldung führt.

2. *Bindungspriorität*
   Wir definieren die folgenden beiden Funktionen.

```
(%%) :: Int -> Int -> Int
x %% y = x + y

(@@) :: Int -> Int ->Int
x @@ y = x * y
```

Die Frage ist nun, zu welchem Wert der Ausdruck

```
1 %% 2 @@ 3
```

auswertet, d.h., die Frage ist, welcher der beiden Operatoren `%%` und `@@` höhere Priorität besitzt. Hierzu kann man bei Infixdeklarationen (mit `infixl`, `infixr` oder `infix`) die Bindungspriorität mit Hilfe einer Zahl zwischen `0` und `9` angeben, wobei `9` die höchste Bindungspriorität repräsentiert. (Falls keine Bindungspriorität angegeben ist, so ist `9` der Defaultwert.) Beispielsweise könnte man folgendes deklarieren.

```
infixl 9 %%
infixl 8 @@
```

Dann steht `1 %% 2 @@ 3` für `(1 %% 2) @@ 3` und das Ergebnis ist `9`. Vertauscht man hingegen die Bindungsprioritäten `9` und `8`, so steht der Ausdruck für `1 %% (2 @@ 3)` und es ergibt sich `7`. Bei gleichen Bindungsprioritäten wird die Auswertung von links nach rechts vorgenommen.

Da es nun also auch Infixdeklarationen gibt, muss die Grammatikregel für Deklarationen noch einmal erweitert werden. Hierbei stehen große geschweifte Klammern für Wahlmöglichkeiten in der Grammatikregel.

$$\underline{\mathsf{decl}} \rightarrow \underline{\mathsf{typedecl}} \mid \underline{\mathsf{fundecl}} \mid \underline{\mathsf{patdecl}} \mid \underline{\mathsf{infixdecl}}$$

$$\underline{\mathsf{infixdecl}} \rightarrow \left\{ \begin{array}{l} \texttt{infix} \\ \texttt{infixl} \\ \texttt{infixr} \end{array} \right\} \; [\left\{ \begin{array}{c} \mathtt{0} \\ \mathtt{1} \\ \vdots \\ \mathtt{9} \end{array} \right\}] \; \underline{\mathsf{op}}_1, \ldots, \underline{\mathsf{op}}_n, \text{ wobei } n \geq 1$$

$$\underline{\mathsf{op}} \rightarrow \underline{\mathsf{varop}} \mid \underline{\mathsf{constrop}}$$

Schließlich sei noch erwähnt, dass Operatoren (ähnlich wie Präfix-Funktionen) auch partiell angewendet werden können (d.h., eine Anwendung ist auch möglich, wenn nicht alle benötigten Argumente vorliegen). So ist `(+ 2)` die Funktion vom Typ `Int -> Int`, die Zahlen um `2` erhöht. Die Funktion ``(6 `divide`)`` vom Typ `Float -> Float` nimmt ein Argument und dividiert die Zahl `6` durch dieses Argument. Die Funktion ``` `divide` 6``` hingegen ist die Funktion vom Typ `Float -> Float`, die ihr Argument durch `6` teilt.

Zusammenfassung der Syntax für Deklarationen

Zusammenfassend ergibt sich die folgende Grammatik zur Erzeugung von Deklarationen in HASKELL.

decl	→ typedecl | fundecl | patdecl | infixdecl	
typedecl	→ $\text{var}_1, \ldots, \text{var}_n$ :: type,	wobei $n \geq 1$
var	→ String von Buchstaben und Zahlen mit Kleinbuchstaben am Anfang	
fundecl	→ funlhs rhs [`where` decls]	
funlhs	→ var $\text{pat}_1 \ldots \text{pat}_n$	wobei $n \geq 1$
rhs	→ = exp | $\text{condrhs}_1 \ldots \text{condrhs}_n$	wobei $n \geq 1$
condrhs	→ | exp = exp	
decls	→ { $\text{decl}_1; \ldots; \text{decl}_n$ },	wobei $n \geq 0$
patdecl	→ pat rhs [`where` decls]	
infixdecl	→ $\left\{\begin{array}{l}\texttt{infix}\\ \texttt{infixl}\\ \texttt{infixr}\end{array}\right\}$ $[\left\{\begin{array}{c}0\\ 1\\ \vdots\\ 9\end{array}\right\}]$ $\text{op}_1, \ldots, \text{op}_n$,	wobei $n \geq 1$
op	→ varop | constrop	
varop	→ String von Sonderzeichen, der nicht mit `:` beginnt	
constrop	→ String von Sonderzeichen, der mit `:` beginnt	

1.1.2 Ausdrücke

Ausdrücke exp (Expressions) stellen das zentrale Konzept der funktionalen Programmierung dar. Ein Ausdruck beschreibt einen Wert (z.B. eine Zahl, einen Buchstaben oder eine Funktion). Die Eingabe eines Ausdrucks in den Interpreter bewirkt seine Auswertung. Darüber hinaus besitzt jeder Ausdruck einen Typ. Bei der Eingabe von "`:t` exp" im (interaktiven Modus des) GHC wird der Typ von exp berechnet und ausgegeben. Bei der Auswertung eines Ausdrucks wird ebenfalls zuerst überprüft, ob der Ausdruck korrekt getypt ist und nur im Erfolgsfall wird die Auswertung tatsächlich durchgeführt. Ein Ausdruck exp kann folgende Gestalt haben:

- var
  Variablenbezeichner wie `x` sind Ausdrücke. Wie erwähnt, werden Variablenbezeichner in HASKELL durch Strings gebildet, die mit einem Kleinbuchstaben beginnen.

- constr
  Eine andere Möglichkeit für Ausdrücke sind *Datenkonstruktoren*. Datenkonstruktoren

dienen zum Aufbau von Objekten einer Datenstruktur und werden bei der Datentypdefinition eingeführt. In HASKELL werden Bezeichner für Datenkonstruktoren durch Strings gebildet, die mit Großbuchstaben beginnen. Beispiele hierfür sind die Datenkonstruktoren `True` und `False` der vordefinierten Datenstruktur `Bool`. Ein weiteres Beispiel sind die Datenkonstruktoren `[]` und `:` für die vordefinierte Datenstruktur der Listen.

- integer
  Auch die ganzen Zahlen `0, 1, -1, 2, -2, ...` sind Ausdrücke.

- float
  Gleitkommazahlen wie `-2.5` oder `3.4e+23` sind ebenfalls Ausdrücke.

- char
  Weitere Ausdrücke sind `'a',...,'z','A',...,'Z','0',...,'9'` sowie das Leerzeichen `' '` und nicht druckbare Kontrollzeichen wie `'\n'` für das Zeilenende-Zeichen. All diese Zeichen werden zu sich selbst (in Apostrophen (Quotes)) ausgewertet.

- `[`$\underline{\text{exp}}_1, \ldots, \underline{\text{exp}}_n$`]`, wobei $n \geq 0$
  Solch ein Ausdruck bezeichnet eine Liste von n Ausdrücken. Wie erwähnt, repräsentiert `[]` hierbei die leere Liste und `[0,1,2,3]` ist eine Abkürzung für `0 : 1 : 2 : 3 : []`, wobei `:` nach rechts assoziiert. Alle Elemente einer Liste müssen denselben Typ haben. Der Typ der obigen Liste wäre z.B. `[Int]`, d.h. der Typ der Listen von ganzen Zahlen.

- string
  Ein string ist eine Liste von Zeichen char (d.h., es ist ein Ausdruck vom Typ `[Char]`). Statt `['h','a','l','l','o']` schreibt man oft `"hallo"`. Solch ein String wird zu sich selbst ausgewertet. Der vordefinierte Typ `String` in Haskell ist identisch mit dem Typ `[Char]`.

- $(\underline{\text{exp}}_1, \ldots, \underline{\text{exp}}_n)$, wobei $n \geq 0$
  Dies ist ein Tupel von Ausdrücken. Anders als bei der Liste können die Ausdrücke in einem Tupel verschiedene Typen haben. Ein Beispiel wäre der Ausdruck `(10, False)`. Dieser Ausdruck hätte z.B. den Typ `(Int, Bool)`. Einelementige Tupel (exp) werden zu exp ausgewertet. Nullelementige Tupel `()` haben den speziellen Typ `()`.

- $(\underline{\text{exp}}_1 \ \ldots \ \underline{\text{exp}}_n)$, wobei $n \geq 2$
  Solch ein Ausdruck steht für die Funktionsanwendung von Ausdrücken. Hierbei lassen wir die Klammerung soweit wie möglich weg. Die Funktionsanwendung hat die höchste Bindungspriorität und assoziiert nach links. Beispiele für solche Ausdrücke sind `square 10` (vom Typ `Int`) oder `plus 5 3` (ebenfalls vom Typ `Int`) oder `plus 5` (vom Typ `Int -> Int`). Der Wert eines Ausdrucks kann also wieder eine Funktion sein.

- `if` $\underline{\text{exp}}_1$ `then` $\underline{\text{exp}}_2$ `else` $\underline{\text{exp}}_3$
  Hierbei muss $\underline{\text{exp}}_1$ vom Typ `Bool` sein und $\underline{\text{exp}}_2$ und $\underline{\text{exp}}_3$ müssen denselben Typ

haben. Bei der Auswertung wird erst der Wert von $\underline{\mathsf{exp}}_1$ bestimmt und danach in Abhängigkeit dieses Werts der Wert von $\underline{\mathsf{exp}}_2$ oder $\underline{\mathsf{exp}}_3$. Statt

```
maxi(x, y) | x >= y     = x
           | otherwise  = y
```

kann man also auch folgendes schreiben:

```
maxi(x, y) = if x >= y then x else y
```

- `let` $\underline{\mathsf{decls}}$ `in` $\underline{\mathsf{exp}}$
  In diesem Ausdruck wird eine lokale Deklarationsfolge $\underline{\mathsf{decls}}$ für den Ausdruck $\underline{\mathsf{exp}}$ definiert. Dies ist analog zur lokalen Deklaration mit Hilfe von `where`, nur wird jetzt die lokale Deklaration voran- statt nachgestellt. Statt

```
roots a b c = ((-b - d)/e, (-b + d)/e)
              where d = sqrt (b*b - 4*a*c)
                    e = 2*a
```

  kann man also auch folgendes schreiben:

```
roots a b c = let d = sqrt (b*b - 4*a*c)
                  e = 2*a
              in ((-b - d)/e, (-b + d)/e)
```

- `case` $\underline{\mathsf{exp}}$ `of` $\{\underline{\mathsf{pat}}_1$ `->` $\underline{\mathsf{exp}}_1$`; ... ;` $\underline{\mathsf{pat}}_n$ `->` $\underline{\mathsf{exp}}_n\}$, wobei $n \geq 1$
  Bei der Auswertung dieses Ausdrucks wird versucht, den Pattern $\underline{\mathsf{pat}}_1$ auf den Ausdruck $\underline{\mathsf{exp}}$ zu matchen. Gelingt dies, so ist das Ergebnis der Ausdruck $\underline{\mathsf{exp}}_1$, wobei die Variablen mit der verwendeten Matching-Substitution instantiiert werden. Ansonsten wird anschließend versucht, den Pattern $\underline{\mathsf{pat}}_2$ auf $\underline{\mathsf{exp}}$ zu matchen, etc. Hierbei ist wieder die Offside-Regel zur Schreibweise verwendbar. Statt

```
und True  y = y
und False y = False
```

  kann man also auch folgendes schreiben:

```
und x y = case x
            of True  -> y
               False -> False
```

  Außerdem kann man statt der Ausdrücke $\underline{\mathsf{exp}}_i$ auch Folgen von bedingten Ausdrücken `|` $\underline{\mathsf{exp}}$ `->` $\underline{\mathsf{exp}}$ verwenden und darüber hinaus ist es in jeder Alternative des `case`-Ausdrucks möglich, lokale Deklarationen mit `where` zu vereinbaren.

- `\`$\underline{\mathsf{pat}}_1 \ldots \underline{\mathsf{pat}}_n$ `->` $\underline{\mathsf{exp}}$, wobei $n \geq 1$
  Solch ein Ausdruck wird als "Lambda-Ausdruck" oder "Lambda-Abstraktion" bezeichnet, denn das Zeichen \ (backslash) repräsentiert den griechischen Buchstaben λ. Der Wert dieses Ausdrucks ist die Funktion, die die Argumente $\underline{\mathsf{pat}}_1 \ldots \underline{\mathsf{pat}}_n$ auf $\underline{\mathsf{exp}}$ abbildet. Beispielsweise ist `\x -> 2 * x` die Funktion, die ein Argument nimmt und es verdoppelt. Ihr Typ ist `Int -> Int`. Mit "Lambda" bildet man also sogenannte "unbenannte Funktionen", die nur an der Stelle ihrer Definition verwendet werden können. Der Ausdruck

  ```
  (\x -> 2 * x) 5
  ```

  wertet daher zu `10` aus. Die Funktion `\x y -> x + y` ist die Additionsfunktion vom Typ `Int -> Int -> Int`. Allgemein hat der Ausdruck `\`$\underline{\mathsf{pat}}_1 \ldots \underline{\mathsf{pat}}_n$ `->` $\underline{\mathsf{exp}}$ den Typ $\underline{\mathsf{type}}_1$ `->...->` $\underline{\mathsf{type}}_n$ `->` $\underline{\mathsf{type}}$, falls $\underline{\mathsf{pat}}_i$ jeweils den Typ $\underline{\mathsf{type}}_i$ und $\underline{\mathsf{exp}}$ den Typ $\underline{\mathsf{type}}$ hat. Bei Lambda-Ausdrücken sind beliebige Patterns möglich, d.h., man kann auch Ausdrücke wie `\(x, y) -> x + y` vom Typ `(Int, Int) -> Int` bilden. An den Lambda-Ausdrücken wird deutlich, dass Funktionen in funktionalen Programmiersprachen wirklich gleichberechtigte Datenobjekte sind, denn man kann sie nun komplett durch geeignete Ausdrücke beschreiben.

  Anstelle der Funktionsdeklaration

  ```
  plus x y = x + y
  ```

  kann man nun also

  ```
  plus = \x y -> x + y
  ```

  oder

  ```
  plus x = \y -> x + y
  ```

  definieren.

Zusammenfassung der Syntax für Ausdrücke

Zusammenfassend ergibt sich die folgende Grammatik für Ausdrücke in HASKELL.

$\underline{\mathsf{exp}}$	$\rightarrow$	$\underline{\mathsf{var}}$	
	|	$\underline{\mathsf{constr}}$	
	|	$\underline{\mathsf{integer}}$	
	|	$\underline{\mathsf{float}}$	
	|	$\underline{\mathsf{char}}$	
	|	`[`$\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n$`]`,	wobei $n \geq 0$
	|	$\underline{\mathsf{string}}$	
	|	$(\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n)$,	wobei $n \geq 0$
	|	$(\underline{\mathsf{exp}}_1 \ \ldots \ \underline{\mathsf{exp}}_n)$,	wobei $n \geq 2$
	|	`if` $\underline{\mathsf{exp}}_1$ `then` $\underline{\mathsf{exp}}_2$ `else` $\underline{\mathsf{exp}}_3$	
	|	`let` $\underline{\mathsf{decls}}$ `in` $\underline{\mathsf{exp}}$	
	|	`case` $\underline{\mathsf{exp}}$ `of` $\{\underline{\mathsf{pat}}_1$ `->` $\underline{\mathsf{exp}}_1; \ldots; \underline{\mathsf{pat}}_n$ `->` $\underline{\mathsf{exp}}_n\}$,	wobei $n \geq 1$
	|	`\`$\underline{\mathsf{pat}}_1 \ldots \underline{\mathsf{pat}}_n$ `->` $\underline{\mathsf{exp}}$,	wobei $n \geq 1$

$\underline{\mathsf{constr}}$ $\rightarrow$ String von Buchstaben und Zahlen mit Großbuchstaben am Anfang

1.1.3 Patterns

Bei der Funktionsdeklaration werden sogenannte *Patterns* für die Argumente angegeben. Sie schränken die Form der erlaubten Argumente ein. Die Syntax von Patterns ist daher ähnlich wie die Syntax von Ausdrücken, denn Patterns sind Prototypen für die erwarteten Werte. Die Form der Werte wird durch die vorkommenden Datenkonstruktoren beschrieben, wobei statt mancher Teilwerte im Pattern Variablen stehen. (Wir verwenden Datenkonstruktoren nun also zur Zerlegung statt zur Konstruktion von Objekten.) Ein Pattern passt zu einem Ausdruck (bzw. er *matcht* diesen Ausdruck), wenn dieser aus dem Pattern bei einer Ersetzung der Variablen durch andere Teil-Ausdrücke hervorgeht. Als Beispiel hatten wir bereits die Algorithmen `und`, `len` und `second` in Abschnitt 1.1.1 betrachtet.

Als weiteres Beispiel betrachten wir den Algorithmus `append`. (Eine analoge (Infix)-Funktion `++` (auf Listen mit Elementen beliebigen Typs) ist in Haskell vordefiniert.)

```
append :: [Int]  -> [Int]  -> [Int]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Um `len (append [1] [2])` zu berechnen, wird das Argument `append [1] [2]` von `len` nur so weit ausgewertet, bis man entscheiden kann, welcher Pattern in der Definition von `len` matcht. Hier würde man also das Argument nur zu `1:append [] [2]` auswerten. An dieser Stelle ist bereits klar, dass nur die zweite Gleichung von `len` verwendbar ist und man erhält `1 + len (append [] [2])`, was dann weiter ausgewertet wird. Lässt sich ohne Auswertung des Arguments nicht feststellen, ob der betrachtete Pattern matcht, so wird der Argumentausdruck zunächst nur so lange ausgewertet, bis der äußerste Konstruktor des Arguments feststeht. (Man bezeichnet dies als *Weak Head Normal Form*, vgl. Kapitel 3.) Dann kann man überprüfen, ob dieser Konstruktor mit dem äußersten Konstruktor des Patterns übereinstimmt. Gegebenenfalls kann es dann zu einem weiteren rekursiven Aufruf des Pattern Matching-Verfahrens für die Teil-Argumente kommen.

Betrachten wir beispielsweise die folgenden Definitionen.

```
zeros :: [Int]
zeros = 0 : zeros

f :: [Int]  -> [Int]  -> [Int]
f [] ys = []
f xs [] = []
```

Die Auswertung von `f [] zeros` terminiert, obwohl `zeros` für sich alleine genommen nicht terminiert. Der Grund ist, dass keine Auswertung von `zeros` nötig ist, um herauszufinden, dass die erste Gleichung von `f` anwendbar ist. Aber auch `f zeros []` terminiert. Hier wird zunächst `zeros` in einem Schritt zu `0 : zeros` ausgewertet. Nun liegt der äußerste Konstruktor “`:`” von `f`’s erstem Argument fest. Da dieser Konstruktor verschieden von dem Konstruktor `[]` ist, kann die erste Gleichung nicht anwendbar sein und man verwendet daher die zweite Gleichung.

Ein Beispiel für die Anwendung des Pattern Matching in Patterndeklarationen ist

```
let x:xs = [1,2,3] in xs
```

Hier ist `x:xs` ein Pattern, der auf den Ausdruck `[1,2,3]` gematcht wird. Das Matchen ist erfolgreich bei der Substitution [`x/1, xs/[2,3]`]. Der obige Ausdruck wird daher zu `[2,3]` ausgewertet.

Eine Einschränkung an Patterns ist, dass sie *linear* sein müssen, d.h., keine Variable darf in einem Pattern mehrfach vorkommen. Der Grund dafür ist, dass sonst nicht mehr alle Auswertungsstrategien dasselbe Ergebnis liefern. Beispielsweise könnte man dann folgende Funktion deklarieren.

```
equal :: [Int] -> [Int] -> Bool
equal xs xs     = True
equal xs (x:xs) = False
```

Der Ausdruck `equal zeros zeros` könnte nun je nach Auswertungsstrategie sowohl zu `True` als auch zu `False` ausgewertet werden. Im allgemeinen kann ein Pattern pat folgende Gestalt haben:

- var
  Jeder Variablenbezeichner ist auch ein Pattern. Dieser Pattern passt auf jeden Wert, wobei die Variable beim Matching an diesen Wert gebunden wird. Ein Beispiel für eine Funktionsdeklaration, bei der solch ein Pattern verwendet wird, ist

  ```
  square x = x * x.
  ```

- _
  Das Zeichen _ (underscore) ist der Joker-Pattern. Er passt ebenfalls auf jeden Wert, aber es erfolgt keine Variablenbindung. Der Joker _ darf daher auch mehrmals in einem Pattern auftreten. Beispielsweise kann man die Funktion `und` also auch wie folgt definieren:

  ```
  und True y = y
  und _ _ = False
  ```

- integer oder float oder char oder string
  Diese Patterns passen jeweils nur auf sich selbst und es findet keine Variablenbindung beim Matching statt.

- (constr $\text{pat}_1 \ldots \text{pat}_n$), wobei $n \geq 0$
  Hierbei ist constr ein n-stelliger Datenkonstruktor. Dieser Pattern matcht Werte, die mit demselben Datenkonstruktor gebildet werden, falls jeweils pat_i das i-te Argument des Werts matcht. Beispiele hierfür hatten wir bei der Deklaration der Algorithmen `und`, `len` und `append` gesehen. (Hierbei ist "`:`" ein Infix-Konstruktor, deshalb steht er nicht außen. (Man kann stattdessen auch `((:) x xs)` schreiben.) Wie üblich lassen wir Klammern soweit möglich weg, um die Lesbarkeit zu erhöhen.

- var@pat
  Dieser Pattern verhält sich wie pat, aber falls pat auf den zu matchenden Ausdruck passt, wird zusätzlich die Variable var an den gesamten Ausdruck gebunden. Als Beispiel betrachten wir die folgende Funktion, die das erste Element einer Liste kopiert.

```
f [] = []
f (x : xs) = x : x : xs
```

  Man könnte also nun statt der zweiten definierenden Gleichung auch folgende Gleichung verwenden.

```
f y@(x : xs) = x : y
```

- [$\text{pat}_1, \ldots, \text{pat}_n$], wobei $n \geq 0$
  Solch ein Pattern matcht Listen der Länge n, falls pat_i jeweils das i-te Element der Liste matcht. Das folgende Beispiel dient dazu, Listen der Länge 3 zu erkennen:

```
has_length_three :: [Int] -> Bool
has_length_three [x,y,z] = True
has_length_three _       = False
```

- ($\text{pat}_1, \ldots, \text{pat}_n$), wobei $n \geq 0$
  Analog matcht ein solcher Tupelpattern Tupel mit n Komponenten, falls pat_i jeweils die i-te Komponente des Tupels matcht. Der Pattern `()` matcht nur den Wert `()`. Hierdurch kann man `maxi` alternativ wie folgt definieren:

```
maxi :: (Int, Int) -> Int
maxi (0,y)     = y
maxi (x,0)     = x
maxi (x,y) = 1 + maxi (x-1,y-1)
```

  Hierbei führt ein Aufruf von `maxi` mit negativen Werten natürlich zur Nichtterminierung.

Generell ist also jeder lineare Term aus Datenkonstruktoren und Variablen ein Pattern.

Zusammenfassung der Syntax für Patterns

Wir erhalten die folgenden Regeln zur Konstruktion von Patterns.

$$
\begin{array}{lll}
\text{pat} & \rightarrow & \text{var} \\
& | & _ \\
& | & \text{integer} \\
& | & \text{float} \\
& | & \text{char} \\
& | & \text{string} \\
& | & (\text{constr}\ \text{pat}_1 \ \ldots \ \text{pat}_n), \quad \text{wobei } n \geq 0 \\
& | & \text{var@pat} \\
& | & [\text{pat}_1, \ldots, \text{pat}_n], \quad \text{wobei } n \geq 0 \\
& | & (\text{pat}_1, \ldots, \text{pat}_n), \quad \text{wobei } n \geq 0
\end{array}
$$

1.1.4 Typen

Jeder Ausdruck in HASKELL hat einen Typ. Typen sind Mengen von gleichartigen Werten, die durch entsprechende Typausdrücke bezeichnet werden. Beispiele für uns bereits bekannte Typen sind die vordefinierten Typen `Bool`, `Int`, `Float` und `Char` sowie zusammengesetzte Typen wie `(Int,Int)`, `Int -> Int`, `(Int,Int) -> Int`, `[Int]`, `[Int -> Bool]`, `[[Int]]`, etc. Allgemein verwendet man die folgenden Arten von Typen type:

- (tyconstr type_1 ... type_n), wobei $n \geq 0$
  Typen werden im allgemeinen mit Hilfe von *Typkonstruktoren* tyconstr aus anderen Typen $\text{type}_1, \ldots, \text{type}_n$ erzeugt. Beispiele für nullstellige (und vordefinierte) Typkonstruktoren sind `Bool`, `Int`, `Float` und `Char`. In HASKELL werden Typkonstruktoren mit Strings bezeichnet, die mit einem Großbuchstaben beginnen (leider sind sie also syntaktisch nicht von Datenkonstruktoren zu unterscheiden, die nicht Typen, sondern Objekte eines Datentyps erzeugen). Hierbei lassen wir wieder Klammern soweit wie möglich weg.

- [type]
  Ein weiterer vordefinierter einstelliger Typkonstruktor ist `[...]`, der einen Typ als Eingabe bekommt und daraus einen neuen Typ erzeugt, dessen Objekte Listen aus Elementen des Ursprungstyps sind. Statt `[...]` type schreibt man [type]. Beispiele für solche Typen sind `[Int]` und `[[Int]]` (der Typ der Listen von Listen ganzer Zahlen).

- (type_1 -> type_2)
  Ein weiterer vordefinierter Typkonstruktor ist der Funktionsraumkonstruktor `->`, der aus zwei Typen einen neuen Typ der Funktionen zwischen ihnen generiert. Ein Beispiel hierfür ist der Typ `Int -> Int`, den beispielsweise die Funktion `square` zur Quadrierung von Zahlen hat.

- ($\text{type}_1, \ldots, \text{type}_n$), wobei $n \geq 0$
  Außerdem gibt es noch den vordefinierten und beliebigstelligen Tupelkonstruktor, mit dem Tupeltypen erzeugt werden können. Ein Beispiel hierfür ist `(Int, Bool, [Int -> Int])`. Wir werden sehen, dass neben diesen vordefinierten Typkonstruktoren auch der Benutzer beliebige weitere Typkonstruktoren definieren kann.

- var
  Schließlich ist auch eine (Typ)variable ein Typ. Dies ist nötig, um parametrische Polymorphie zu erreichen, wie im Folgenden erklärt wird.

Parametrische Polymorphie

"Polymorphie" bedeutet "Vielgestaltigkeit" und wird in der Informatik meistens verwendet, um auszudrücken, dass gleiche bzw. gleich heißende Funktionen für verschiedene Arten von Argumenten verwendet werden können. Man unterscheidet hierbei die *parametrische Polymorphie* und die *Ad-hoc-Polymorphie*. Bei der parametrischen Polymorphie wird ein und dieselbe Funktion für Argumente verschiedener Typen verwendet. Bei der Ad-hoc-Polymorphie wird zwar das gleiche Funktionssymbol für Argumente verschiedener Typen

verwendet, aber abhängig vom Typ der Argumente werden verschiedene Funktionen ausgeführt. Funktionale Sprachen wie HASKELL besitzen beide Arten der Polymorphie, wie im Folgenden erläutert wird.

Wir betrachten zunächst die parametrische Polymorphie. Hierbei wirkt eine Funktion gleichartig auf eine ganze Sammlung von Datenobjekten. Beispiele hierfür sind die folgenden Funktionen.

```
id :: a -> a
id x = x

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

Wir haben im Typ der Funktionen `id` und `len` eine Typvariable `a` verwendet. Dies bedeutet, dass diese Funktionen für jede mögliche Ersetzung der Typvariablen durch Typen definiert sind. Beispielsweise darf man nun sowohl `len [True, False]` als auch `len [1,2,3]` aufrufen.

Analog verhält es sich auch bei der Funktion `append` (bzw. `++`, die in HASKELL vordefiniert ist).

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Das mehrfache Vorkommen der gleichen Typvariable `a` im Typ `[a] -> [a] -> [a]` erzwingt die Übereinstimmung der Typen der beiden Argumente von `++`. Eine Funktion vom Typ $\underline{\mathsf{type}}_1$ `->` $\underline{\mathsf{type}}_2$ kann auf ein Argument vom Typ $\underline{\mathsf{type}}$ angewendet werden, falls es eine (allgemeinste) Ersetzung σ der Typvariablen (d.h. einen *allgemeinsten Unifikator*) gibt, so dass $\sigma(\underline{\mathsf{type}}_1) = \sigma(\underline{\mathsf{type}})$ ist. Das Ergebnis hat dann den Typ $\sigma(\underline{\mathsf{type}}_2)$.

Als Beispiel betrachten wir den Ausdruck `[True] ++ []`. Der Teilausdruck `[True]` hat den Typ `[Bool]` und das zweite Argument `[]` hat den Typ `[b]`. Die gesuchte Substitution σ mit $\sigma(\texttt{[a]}) = \sigma(\texttt{[Bool]}) = \sigma(\texttt{[b]})$ ist $\sigma = [\,\texttt{a/Bool},\ \texttt{b/Bool}\,]$. Also ist dieser Ausdruck korrekt getypt und er hat den Typ `[Bool]`. Die Untersuchung auf Typkorrektheit und die Berechnung allgemeinster Typen kann automatisch durchgeführt werden. Wir werden hierauf in Kapitel 4 genauer eingehen. Lässt man also die Typdeklaration von `append` weg, so wird automatisch der Typ `[a] -> [a] -> [a]` bestimmt.

Typdefinitionen: Einführung neuer Typen

Um neue Typen bzw. neue Typkonstruktoren einzuführen, gibt es in HASKELL eigene Formen der Deklaration. Diese Deklarationen sind aber (im Gegensatz zu den bisher betrachteten Deklarationen) nur auf der obersten Programmebene und nicht in lokalen Deklarationsblöcken möglich. Aus diesem Grund unterscheiden wir nun zwischen allgemeinen Deklarationen $\underline{\mathsf{decl}}$ und Deklarationen $\underline{\mathsf{topdecl}}$, die nur auf dieser obersten Ebene erlaubt

sind. Ein Programm ist dann eine Folge von linksbündig untereinander stehenden $\underline{\mathsf{topdecl}}$-Deklarationen. Die Grammatik für $\underline{\mathsf{topdecl}}$ lautet wie folgt:

$$
\begin{array}{lll}
\underline{\mathsf{topdecl}} & \rightarrow & \underline{\mathsf{decl}} \\
& | & \mathsf{type}\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = \underline{\mathsf{type}}, \qquad \text{wobei } n \geq 0 \\
& | & \mathsf{data}\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = \\
& & \qquad \underline{\mathsf{constr}}_1\ \underline{\mathsf{type}}_{1,1} \ldots \underline{\mathsf{type}}_{1,n_1} \\
& & \qquad | \ldots \\
& & \qquad | \ \underline{\mathsf{constr}}_k\ \underline{\mathsf{type}}_{k,1} \ldots \underline{\mathsf{type}}_{k,n_k}, \quad \text{wobei } n \geq 0, k \geq 1, n_i \geq 0
\end{array}
$$

Insbesondere können alle bislang behandelten Deklarationen $\underline{\mathsf{decl}}$ also auch auf der obersten Programmebene auftreten. Zusätzlich kann man aber nun mit Hilfe der Schlüsselworte `type` und `data` neue Typen einführen.

Die erste Möglichkeit, neue Typen bzw. Typkonstruktoren zu definieren, ist die sogenannte *Typabkürzung* (type synonym) mit Hilfe des Schlüsselworts `type`. Beispielsweise kann man durch

```
type Position = (Float, Float)
type String  = [Char]
type Pair a b = (a, b)
```

drei neue Typkonstruktoren deklarieren (wobei `String` bereits auf diese Weise in HASKELL vordefiniert ist). Der Typ (bzw. der nullstellige Typkonstruktor) `Position` ist dann lediglich eine *Abkürzung* für den Typ `(Float, Float)`, d.h., diese beiden Typen werden als gleich betrachtet. Eine Typabkürzung kann Parameter haben, d.h., `Pair` ist ein zweistelliger Typkonstruktor. Wiederum handelt es sich hierbei aber nur um Abkürzungen, d.h., die Typen `Pair Float Float` und `Position` sind identisch.

Eine Einschränkung bei Typabkürzungen ist, dass die Variablen $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$ paarweise verschieden sein müssen und dass der Typ $\underline{\mathsf{type}}$ auf der rechten Seite keine Variablen außer diesen enthalten darf (ähnliche Einschränkungen gibt es auch bei Funktionsdeklarationen). Außerdem dürfen Typabkürzungen nicht rekursiv sein, d.h., der Typ $\underline{\mathsf{type}}$ darf nicht von dem Typkonstruktor $\underline{\mathsf{tyconstr}}$, der gerade definiert wird, abhängen.

Die andere Möglichkeit zur Definition neuer Typen ist die Einführung von algebraischen Datentypen durch Angabe einer EBNF-artigen kontextfreien Grammatik. Dies geschieht mit Hilfe des Schlüsselworts `data`. Beispielsweise kann man die folgenden Aufzählungstypen definieren.

```
data Color = Red | Yellow | Green
data MyBool = MyTrue | MyFalse
```

In der Definition eines algebraischen Datentyps wie `Color` werden also die verschiedenen Möglichkeiten aufgezählt, wie mit entsprechenden Datenkonstruktoren (wie `Red`, `Yellow`, `Green`) Objekte dieses Typs konstruiert werden können. Durch diese Definitionen sind nun zwei neue nullstellige Typkonstruktoren `Color` und `MyBool` eingeführt worden. Die folgenden beiden Funktionen verdeutlichen, dass das Pattern Matching auch bei selbstdefinierten Datenstrukturen verwendbar ist. Jeder lineare Term aus Variablen und Datenkonstruktoren ist ein Pattern.

```
traffic_light :: Color -> Color
traffic_light Red    = Green
traffic_light Green  = Yellow
traffic_light Yellow = Red

und :: MyBool -> MyBool -> MyBool
und MyTrue y = y
und _      _ = MyFalse
```

Man kann Werte von selbstdefinierten Typen nicht direkt ausgeben. Bei der Ausgabe auf dem Bildschirm wird eine vordefinierte Funktion `show` aufgerufen, die den Wert in einen String umwandelt. Diese existiert bei den vordefinierten Typen. Bei eigenen Typen kann sie selbst geschrieben werden. Alternativ kann man sie automatisch erzeugen lassen. Hierzu muss bei der Datentypdeklaration `deriving Show` hinzugefügt werden. Zum eigenen Schreiben dieser Funktion muss man das Konzept des Überschreibens in HASKELL einführen, das anschließend betrachtet wird.

Betrachten wir ein weiteres Beispiel, bei dem die Datenstruktur der natürlichen Zahlen definiert wird.

```
data Nats = Zero | Succ Nats
```

Der Datentyp `Nats` besitzt zwei Datenkonstruktoren `Zero` und `Succ`. In der Datentypdeklaration werden jeweils nach dem Namen des Datenkonstruktors die Typen seiner Argumente angegeben. Somit hat `Zero` keine Argumente, d.h., sein Typ ist `Nats`. `Succ` hingegen hat den Typ `Nats -> Nats`. Wenn `Succ` die Nachfolgerfunktion repräsentiert, so steht `Succ (Succ Zero)` für die Zahl 2. Nun lassen sich auf diesem Datentyp Funktionen wie `plus` oder `half` definieren.

```
plus :: Nats -> Nats -> Nats
plus Zero     y = y
plus (Succ x) y = Succ (plus x y)

half :: Nats -> Nats
half Zero            = Zero
half (Succ Zero)     = Zero
half (Succ (Succ x)) = Succ (half x)
```

Natürlich können aber auch parametrische Typen (d.h. nicht-nullstellige Typkonstruktoren) definiert werden. Auf folgende Weise kann man einen Listentyp definieren, der den in HASKELL bereits vordefinierten Listen entspricht.

```
data List a = Nil | Cons a (List a)
```

Durch diese Deklaration wird ein neuer parametrisierter algebraischer Datentyp `List a` für Listen mit Elementen vom Typ `a` eingeführt. `List` ist ein einstelliger Typkonstruktor, d.h., `List Int` wären z.B. Listen von ganzen Zahlen. Der Datentyp `List a` besitzt zwei Datenkonstruktoren `Nil` und `Cons`. `Nil` hat keine Argumente, d.h., sein Typ ist `List a`. `Cons` hingegen hat den Typ `a -> (List a) -> (List a)`. Wenn `Cons` das Einfügen eines

neuen Elements in eine Liste repräsentiert, so steht `Cons 1 Nil` für die einelementige Liste mit dem Element `1`. Die folgenden Beispiele zeigen die Algorithmen `len` und `append` auf selbstdefinierten Listen und natürlichen Zahlen

```
len :: List a -> Nats
len Nil         = Zero
len (Cons x xs) = Succ (len xs)

append :: List a -> List a -> List a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Wie bei Typabkürzungen müssen auch bei der Definition algebraischer Datentypen die Variablen $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$ paarweise verschieden sein und der Typ $\underline{\mathsf{type}}$ auf der rechten Seite darf keine Variablen außer diesen enthalten. Anders als bei der Typabkürzung dürfen Datentypen aber rekursiv definiert sein. Dies ist z.B. bei `Nats` der Fall, da Objekte des Typs `Nats` mit Hilfe eines Konstruktors `Succ` gebildet werden, dessen Argumente wiederum vom Typ `Nats` sind. Analog verhält es sich bei `List a`.

Auch verschränkt rekursive Datentypen sind möglich. Im folgenden Beispiel wird der Typkonstruktor `Tree` mit Hilfe von `Forest` definiert und die Definition von `Forest` benötigt wiederum `Tree`. Hierbei realisiert `Tree` Vielwegbäume (d.h. Bäume mit beliebiger Anzahl von Kindern) und `Forest` repräsentiert Listen von Vielwegbäumen.

```
data Tree element = Node element (Forest element)

data Forest element = NoTrees | Trees (Tree element) (Forest element)
```

Typklassen

Eine *Typklasse* ist eine Menge von Typen. Ihre Elemente bezeichnet man als *Instanzen* der Typklasse. Die Namen der Funktionen auf diesen Instanz-Typen sind überladen, d.h., die Funktionen heißen in all diesen Typen gleich, sie können aber bei jedem Typ eine unterschiedliche Definition haben. Dies bezeichnet man als *Ad-hoc-Polymorphie*, da ad-hoc (beim Aufruf eines Funktionssymbols) aufgrund des Typs der Argumente entschieden wird, welche Funktion tatsächlich ausgeführt wird.

Die Funktionen für Gleichheit und Ungleichheit sind in HASKELL vordefiniert. Man würde zunächst erwarten, dass ihre Typdeklaration wie folgt lautet.

```
(==), (/=) :: a -> a -> Bool
```

Die Gleichheitsfunktion `==` ist zwar für viele Datentypen sinnvoll, jedoch nicht für alle (z.B. ist bei Funktionen die Gleichheit nicht entscheidbar). Hinzu kommt, dass die Implementierung von `==` für verschiedene Datentypen unterschiedlich ist. Beim Vergleich von zwei Zeichenfolgen muss eine andere Operation durchgeführt werden als beim Vergleich von zwei Zahlen.

Die Typen, auf denen die Gleichheit `==` definiert ist, werden daher in einer Typklasse `Eq` zusammengefasst. Daher hat der Typ der beiden Funktionen `==` und `/=` einen sogenannten

Kontext `Eq a`.

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

Dies bedeutet, dass für die Typvariable `a` nur Typen aus der Typklasse `Eq` eingesetzt werden dürfen.

Die Syntax zur Deklaration von Typklassen (wie `Eq`) ist wie folgt. Hierzu muss eine weitere Grammatikregel für das Nichtterminal topdecl eingeführt werden.

topdecl	$\rightarrow$	`class` tyconstr var [`where` {$\text{cdecl}_1; \ldots; \text{cdecl}_n$}], wobei $n \geq 1$
cdecl	$\rightarrow$	typedecl | fundecl | infixdecl | var rhs

Die Typklasse `Eq` ist in HASKELL vordefiniert. Ihre Definition könnte z.B. wie folgt lauten.

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

Die erste Zeile deklariert die Typklasse `Eq` mit dem Parameter `a`. Die zweite Zeile gibt den Typ der Operationen dieser Klasse an, wobei die Typvariable `a` implizit durch den Kontext `Eq a =>` eingeschränkt ist. Diese Operationen bezeichnet man auch als *Methoden* der Klasse. In der dritten Zeile wird eine Default-Implementierung von `/=` angegeben. In Instanzen dieser Klasse (d.h. in konkreten Typen, die Mitglied von `Eq` sind) können die Methoden der Klasse überschrieben werden. Nur wenn keine eigene Definition für `==` oder `/=` vorhanden ist, wird die Default-Implementierung verwendet. Dies bedeutet in unserem Beispiel, dass man bei den Instanzentypen jeweils nur die Definition von `==` angeben muss. Die Funktion `/=` berechnet dann automatisch die Ungleichheit auf korrekte Weise.

Die Typ- und Methodendeklarationen einer Klasse werden durch das Nichtterminalsymbol cdecl beschrieben. Man erkennt an der Grammatik, dass hier nahezu beliebige Deklarationen erlaubt sind. Die einzige Einschränkung im Vergleich zu decl ist, dass keine beliebigen Patterndeklarationen zugelassen sind, sondern nur Patterndeklarationen, bei denen auf der linken Seite nur eine Variable statt eines beliebigen Patterns steht.

Durch die Deklaration von `Eq` allein ist aber noch nicht klar, welche Typen zu der Typklasse `Eq` gehören. Hierzu verwendet man *Instanzendeklarationen*. Die zugehörigen Grammatikregeln lauten wie folgt.

topdecl	$\rightarrow$	`instance` tyconstr instype [`where` {$\text{idecl}_1; \ldots; \text{idecl}_n$}],	wobei $n \geq 1$
instype	$\rightarrow$	(tyconstr $\text{var}_1 \ldots \text{var}_n$),	wobei $n \geq 0$
		| `[`var`]`	
		| (var_1 `->` var_2)	
		| ($\text{var}_1, \ldots, \text{var}_n$),	wobei $n \geq 2$
idecl	$\rightarrow$	fundecl | var rhs	

Beispielsweise kann man deklarieren, dass `Int` ein Element der Typklasse `Eq` ist.

```
instance Eq Int where
  (==) = primEqInt
```

Dies besagt zum einen, dass `Int` ein Mitglied der Typklasse `Eq` ist und zum anderen wird hierdurch eine Implementierung der Gleichheit `==` auf `Int` angegeben. Wäre in `Eq` bereits eine Default-Implementierung von `==` angegeben worden, so würde sie (bei Argumenten vom Typ `Int`) durch diese Implementierung überschrieben werden. Bei der Funktion `primEqInt` handelt es sich um eine primitive vordefinierte Funktion. Die Definition der Ungleichheit wird aus der Klassendeklaration abgeleitet (d.h., `x /= y` ist auf `Int` definiert als `not (primEqInt x y)`).

Wie aus der Grammatikregel für Instanzendeklarationen hervorgeht, kann man nur eingeschränkte Formen von Typen instype als Instanzen von Typklassen deklarieren: Typkonstruktoren dürfen hier nur auf paarweise verschiedene Typvariablen und nicht auf beliebige Argumenttypen angewendet werden, Typtupel müssen mindestens zwei Komponenten haben und eine Typvariable allein kann kein Exemplar einer Typklasse sein.

Die Deklarationen idecl, um Methoden in Instanzendeklarationen zu implementieren, sind noch eingeschränkter als in der Klassendeklaration: Man kann keine Typdeklarationen (denn die Typen werden ja schon in der Klassendeklaration festgelegt), keine Infixdeklarationen und keine beliebigen Patterndeklarationen verwenden.

Generell kann man durch die Verwendung von Typklassen jetzt bei jedem Typ einen *Kontext* mit angeben. Dieser Kontext besagt, dass bestimmte Typvariablen nur mit Typen einer bestimmten Klasse instantiiert werden dürfen. Die hierfür benötigten (geänderten) Grammatikregeln sind folgende.

context → (tyconstr$_1$ var$_1$, ..., tyconstr$_n$ var$_n$), wobei $n \geq 1$

typedecl → var$_1$, ..., var$_n$ :: [context ⇒] type, wobei $n \geq 1$

topdecl → decl
| `type` ...
| `data` ...
| `class` [context ⇒] tyconstr var ...
| `instance` [context ⇒] tyconstr instype ...

Die Verwendung von Kontexten in Instanzendeklarationen ist bei Instanzendeklarationen für Typkonstruktoren mit Parametern sinnvoll. Die nächste Instanzendeklaration besagt folgendes: Falls Elemente vom Typ `a` auf ihre Gleichheit überprüft werden können, dann ist die Gleichheit auch bei Listen mit Elementen vom Typ `a` definiert. Der Typ `[a]` ist also in der Klasse `Eq`, falls auch der Typ `a` schon in der Klasse `Eq` ist. Um dies darzustellen, verwendet man den Kontext `Eq a =>` in der Instanzendeklaration.

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

Hierbei ist `&&` die vordefinierte Konjunktion auf booleschen Werten. In dem Ausdruck `x == y` auf der rechten Seite der zweiten definierenden Gleichung bezeichnet `==` die Gleichheit auf dem Typ `a`, wohingegen im zweiten Ausdruck `xs == ys` die Gleichheit auf dem Typ `[a]` gemeint ist (d.h., dies ist ein rekursiver Aufruf).

Zur Behandlung von Paaren sind Einschränkungen an zwei Typvariablen erforderlich.

```
instance (Eq a, Eq b) => Eq (a,b) where
     (x,y) == (x′,y′) = x == x′ && y == y′
```

Falls die Typen `a` und `b` Exemplare der Typklasse `Eq` sind, so ist auch der Typ `(a,b)` ein Exemplar der Typklasse `Eq`, wobei die Gleichheit komponentenweise definiert ist.

Man erkennt, dass auf diese Weise der Operator `==` in der Tat überladen ist, da er für verschiedene Datentypen unterschiedlich definiert ist. Eine besonders einfache Form der Instanzendeklaration besteht darin, dass man bei der Deklaration von algebraischen Datentypen (mittels `data`) `deriving`-Klauseln anhängt (wie z.B. `deriving Eq`). Hierdurch wird festgelegt, dass der Typ eine Instanz der jeweiligen Typklassen sein soll und es wird automatisch eine Standardimplementierung für die Methoden der Klasse erzeugt.

Hierarchische Organisation von Typklassen

Durch Verwendung von Kontexten in Klassendeklarationen kann man Unterklassen zu bereits eingeführten Typklassen deklarieren. Beispielsweise ist in HASKELL eine Typklasse `Ord` vordefiniert, in der Methoden wie `<`, `>`, `<=`, `>=`, etc. zur Verfügung stehen. Offensichtlich gilt `x <= y && y <= x` genau dann, wenn `x == y` gilt. Das bedeutet, dass nur Typen der Klasse `Eq` auch in der Klasse `Ord` liegen können. `Ord` sollte daher eine *Unterklasse* von `Eq` sein. Man könnte die Klasse `Ord` daher wie folgt deklarieren.

```
class Eq a => Ord a where
   (<),(>),(<=),(>=) :: a -> a -> Bool
   x < y = x <= y && x /= y
        ...
```

In der ersten Zeile wird die Beziehung zwischen den Klassen `Eq` und `Ord` festgelegt: Nur wenn `a` zur Typklasse `Eq` gehört, kann `a` auch zur Typklasse `Ord` gehören. Danach folgen die Typdeklarationen der Methoden und ihre Default-Implementierungen.

Eine weitere vordefinierte Typklasse ist die Klasse `Show`, die diejenigen Typen enthält, deren Objekte auf dem Bildschirm angezeigt werden können. Für diese Objekte existiert also eine Methode `show`, die die Objekte in Strings wandelt, die dann ausgegeben werden können. Die Deklaration der Klasse `Show` könnte wie folgt lauten.

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

Bei Eingabe eines Ausdrucks wertet der Interpreter diesen zunächst aus. Anschließend versucht er, den resultierenden Wert mit Hilfe der Funktion `show` auf dem Bildschirm auszugeben. Dies ist aber nur möglich, wenn der Typ dieses Werts in der Klasse `Show` enthalten ist. Ansonsten erhält man eine Fehlermeldung.

Um auch Werte von benutzerdefinierten Datenstrukturen auf dem Bildschirm ausgeben zu können, muss man diese Datenstrukturen als Instanzen der Klasse `Show` deklarieren und eine geeignete Implementierung der Methode `show` angeben. (Durch die Klausel "`deriving Show`" ist dies natürlich möglich, aber dann muss man die automatisch erzeugte Implementierung von `show` für diesen Datentyp verwenden.) Um die benutzerdefinierte Datenstruktur

```
data List a = Nil | Cons a (List a)
```

selbst als Instanz der Klasse `Show` zu deklarieren, kann man z.B. die folgende Instanzendeklaration schreiben:

```
instance Show a => Show (List a) where
         show Nil = "[]"
         show (Cons x xs) = show x ++  " : " ++ show xs
```

Gibt man nun `Cons 1 (Cons 2 Nil)` im Interpreter ein, so wird daraufhin `1 : 2 : []` auf dem Bildschirm ausgegeben.

Typklassen sind auch hilfreich bei der Überladung von arithmetischen Operatoren. Operatoren wie `+`, `-`, `*`, etc. sind sowohl auf ganzen Zahlen (`Int`) wie auf Gleitkommazahlen (`Float`) definiert. Die Typklasse, die die Zahlentypen und ihre Operationen zusammenfasst, ist in HASKELL die vordefinierte Klasse `Num`.

```
class (Eq a, Show a) => Num a where
                             ...
```

Zahlen wie `0`, `1`, `2` etc. sind vom Typ `Num a => a`, d.h., `2` kann sowohl als ganze Zahl (`Int`) als auch als Gleitkommazahl (`Float`) verwendet werden.

Zusammenfassung der Syntax für Typen

Die Syntaxregeln für Typen, Typdefinitionen und Typklassen etc. lauten zusammenfassend wie folgt:

$$
\begin{array}{lcll}
\underline{\mathsf{type}} & \rightarrow & (\underline{\mathsf{tyconstr}}\ \underline{\mathsf{type}}_1\ \ldots\ \underline{\mathsf{type}}_n), & \text{wobei } n \geq 0 \\
& | & [\underline{\mathsf{type}}] & \\
& | & (\underline{\mathsf{type}}_1 \texttt{ -> } \underline{\mathsf{type}}_2) & \\
& | & (\underline{\mathsf{type}}_1, \ldots, \underline{\mathsf{type}}_n), & \text{wobei } n \geq 0 \\
& | & \underline{\mathsf{var}} &
\end{array}
$$

$\underline{\mathsf{tyconstr}} \rightarrow$ String von Buchstaben und Zahlen mit Großbuchstaben am Anfang

$$
\begin{array}{lcll}
\underline{\mathsf{topdecl}} & \rightarrow & \underline{\mathsf{decl}} & \\
& | & \texttt{type}\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = \underline{\mathsf{type}}, & \text{wobei } n \geq 0 \\
& | & \texttt{data}\ [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = & \\
& & \quad \underline{\mathsf{constr}}_1\ \underline{\mathsf{type}}_{1,1} \ldots \underline{\mathsf{type}}_{1,n_1} & \\
& & \quad | \ldots & \\
& & \quad | \ \underline{\mathsf{constr}}_k\ \underline{\mathsf{type}}_{k,1} \ldots \underline{\mathsf{type}}_{k,n_k}, & \text{wobei } n \geq 0, k \geq 1, n_i \geq 0 \\
& | & \texttt{class}\ [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}} & \\
& & \quad [\texttt{where}\ \{\underline{\mathsf{cdecl}}_1; \ldots; \underline{\mathsf{cdecl}}_n\}], & \text{wobei } n \geq 1 \\
& | & \texttt{instance}\ [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{instype}} & \\
& & \quad [\texttt{where}\ \{\underline{\mathsf{idecl}}_1; \ldots; \underline{\mathsf{idecl}}_n\}], & \text{wobei } n \geq 1
\end{array}
$$

cdecl	$\to$	typedecl | fundecl | infixdecl | var rhs	
instype	$\to$	(tyconstr $\text{var}_1 \ldots \text{var}_n$),	wobei $n \geq 0$
		| [var]	
		| (var_1 -> var_2)	
		| ($\text{var}_1, \ldots, \text{var}_n$),	wobei $n \geq 2$
idecl	$\to$	fundecl | var rhs	
context	$\to$	($\text{tyconstr}_1\ \text{var}_1, \ldots, \text{tyconstr}_n\ \text{var}_n$),	wobei $n \geq 1$
typedecl	$\to$	$\text{var}_1, \ldots, \text{var}_n$ :: [context $\Rightarrow$] type,	wobei $n \geq 1$

1.2 Funktionen höherer Ordnung

Funktionen höherer Ordnung ("higher-order functions" oder auch "Funktionale") sind dadurch charakterisiert, dass ihre Argumente oder ihr Resultat selbst wieder Funktionen sind. Beispielsweise ist `square :: Int -> Int` eine Funktion erster Ordnung, wohingegen `plus :: Int -> Int -> Int` eine Funktion höherer Ordnung ist, denn `plus 1` (das Resultat von `plus` bei Anwendung auf das Argument `1`) ist wieder eine Funktion. Zunächst betrachten wir einige typische Funktionen höherer Ordnung.

Die Funktionskomposition "."

Eine sehr oft verwendete Funktion höherer Ordnung ist die Funktionskomposition $(f \circ g)$. In HASKELL ist die Funktionskomposition für zwei einstellige Funktionen bereits als Infix-Operator `.` vordefiniert. Wenn `g` eine Funktion vom Typ `a -> b` und `f` eine Funktion vom Typ `b -> c` ist, so ist `f.g` die Funktion, die entsteht, indem man erst `g` und dann `f` anwendet.

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Beispielsweise ist `half.square` die Funktion, die ein Argument x erst quadriert und dann halbiert (d.h., sie berechnet $\frac{x^2}{2}$). Beispielsweise ergibt die Auswertung von `(half.square) 4` das Ergebnis `8` $(= \frac{4^2}{2})$ und `((\x -> x+1).square) 5` ergibt `26`.

Die Funktionen `curry` und `uncurry`

Wie bereits erwähnt, bedeutet *Currying* die Überführung von Tupelargumenten in eine Folge von Argumenten. Beispielsweise geschieht der Schritt von der ersten der beiden folgenden Definitionen von `plus` zur zweiten durch Currying und der Schritt zurück geschieht durch Uncurrying.

```
plus :: (Int, Int) -> Int
plus (x,y) = x + y
```

bzw.

```
plus :: Int -> Int -> Int
plus x y = x + y
```

Im allgemeinen kann man diese Überführung durch (in HASKELL vordefinierte) Funktionen höherer Ordnung vornehmen.

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f = g
      where g x y = f (x,y)
```

bzw.

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry g = f
        where f (x,y) = g x y
```

Es gilt `curry (uncurry g) = g` und `uncurry (curry f) = f`. Mit Hilfe dieser beiden Funktionen ist es nun möglich, eine Funktion beliebig als curried- oder uncurried-Variante zu verwenden. Die Auswertung von `uncurry (+) (1,2)` ergibt z.B. `3`.

Die Funktion `map`

Funktionen höherer Ordnung ermöglichen es insbesondere, Probleme und Programme übersichtlicher zu strukturieren. Hierzu verwendet man typischerweise eine Menge von klassischen, oft einsetzbaren Funktionen höherer Ordnung, die bestimmte Rekursionsmuster implementieren. Programme, die auf diese Weise erstellt werden, sind besser lesbar und einfacher wiederzuverwenden. Im Folgenden sollen einige dieser klassischen Funktionen höherer Ordnung vorgestellt werden.

Betrachten wir eine Funktion `suclist`, die alle Zahlen in einer Liste von ganzen Zahlen um `1` erhöht.

```
suc :: Int -> Int
suc = plus 1

suclist :: [Int] -> [Int]
suclist []     = []
suclist (x:xs) = suc x : suclist xs
```

Bei einem Aufruf von `suclist` mit dem Argument

$$[x_1, x_2, \ldots, x_n]$$

erhält man also das Ergebnis

$$[\texttt{suc}\,x_1,\ \texttt{suc}\,x_2,\ \ldots,\texttt{suc}\,x_n].$$

Analog dazu berechnet die Funktion `sqrtlist` die Wurzel für jedes Element einer Liste von Gleitkommazahlen.

```
sqrtlist :: [Float] -> [Float]
sqrtlist []     = []
sqrtlist (x:xs) = sqrt x : sqrtlist xs
```

Bei einem Aufruf von `sqrtlist` mit dem Argument

$$[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$$

erhält man demnach das Ergebnis

$$[\mathtt{sqrt}\,\mathtt{x}_1,\ \mathtt{sqrt}\,\mathtt{x}_2,\ \ldots, \mathtt{sqrt}\,\mathtt{x}_n].$$

Man erkennt, dass die beiden Funktionen `suclist` und `sqrtlist` sehr ähnlich sind, da beide Funktionen eine Liste elementweise abarbeiten (durch die Funktionen `suc` und `sqrt`) und dann die verarbeitete Liste zurückgeben (d.h., die eigentliche Datenstruktur der Liste bleibt erhalten). Es liegt auf der Hand, diese beiden Funktionen durch ein und dieselbe Funktion zu realisieren. Dazu sind folgende Schritte notwendig:

- Abstraktion vom Datentyp der Listenelemente (`Int` bzw. `Float`). Dies ist nur in Sprachen mit (parametrischer) Polymorphie möglich.

- Abstraktion von der Funktion, die auf jedes Element der Liste angewandt werden soll (`suc` bzw. `sqrt`). Dies ist nur in Sprachen möglich, in denen Funktionen als gleichberechtigte Datenobjekte behandelt werden.

Allgemein wird also eine Funktion `g` auf alle Elemente der Liste angewandt. Man benötigt allgemein also eine Funktion `f`, so dass man beim Aufruf von `f` mit dem Argument

$$[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$$

das folgende Ergebnis erhält.

$$[\mathtt{g}\,\mathtt{x}_1,\ \mathtt{g}\,\mathtt{x}_2,\ \ldots, \mathtt{g}\,\mathtt{x}_n]$$

Die allgemeine Form von Funktionen dieser Bauart ist demnach wie folgt.

```
f :: [a] -> [b]
f [] = []
f (x:xs) = g x : f xs
```

Da `g` eine beliebige Funktion ist, sollte sie zusätzliches Eingabeargument der Funktion sein. Auf diese Weise erhält man die Funktion `map` (wobei `f` von oben nun der Funktion `map g` entspricht).

```
map :: (a -> b) -> [a] -> [b]
map g [] = []
map g (x:xs) = g x : map g xs
```

Die Funktion `map` ist eine Funktion höherer Ordnung, da sowohl ihr Resultat als auch ihr Argument Funktionen sind. Sie ist in HASKELL bereits vordefiniert. Das Ergebnis von `map g` $[x_1, x_2, \ldots, x_n]$ ist

$$[g\,x_1,\ g\,x_2,\ \ldots, g\,x_n].$$

Wir implementieren daher das Rekursionsmuster "Durchlaufe eine Liste und wende eine Funktion auf jedes Element an" mit einer eigenen Funktion `map`. Die Verwendung eines festen Satzes solcher Funktionen, die Rekursionsmuster realisieren, führt zu großer Modularisierbarkeit, Wiederverwendbarkeit und Lesbarkeit von Programmen.

Die weiter oben definierten Funktionen `suclist` und `sqrtlist` lassen sich nun wie folgt auf nicht-rekursive Weise definieren:

```
suclist  l = map suc  l
sqrtlist l = map sqrt l
```

oder noch einfacher als

```
suclist  = map suc
sqrtlist = map sqrt
```

Analog kann man entsprechende `map`-Funktionen auf anderen Datenstrukturen definieren, z.B. auf den selbst definierten Listen, die wie folgt definiert waren:

```
data List a = Nil | Cons a (List a)
```

Hier lautet die entsprechende Definition wie folgt:

```
mapList :: (a -> b) -> List a -> List b
mapList g Nil         = Nil
mapList g (Cons x xs) = Cons (g x) (mapList g xs)
```

Als weiteres Beispiel betrachten wir die folgende Datenstruktur von Vielwegbäumen.

```
data Tree a = Node a [Tree a]
```

Die Funktion `map` lautet auf dieser Datenstruktur wie folgt.

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree g (Node x ts) = Node (g x) (map (mapTree g) ts)
```

Beim Aufruf `mapTree g t` wird die Funktion `g` auf jeden Knoten des Baums `t` angewendet. Falls `t` der Baum

$$\texttt{Node}\ x_1\ \texttt{[Node}\ x_2\ \texttt{[]]}$$

ist, so ist `mapTree g t` der Baum

$$\texttt{Node}\ (g\,x_1)\ \texttt{[Node}\ (g\,x_2)\ \texttt{[]]}.$$

Um die Verwendung von `mapTree` zu verdeutlichen, implementieren wir nun eine Funktion `sucTree`, die alle Zahlen in einem Baum von ganzen Zahlen um `1` erhöht. Sie kann elegant mit Hilfe von `mapTree` formuliert werden.

```
sucTree :: Tree Int -> Tree Int
sucTree = mapTree suc
```

Generell gilt: `map`-Funktionen wenden eine Funktion `g` auf jeden Teilwert eines zusammengesetzten Datenobjekts an.

Die Funktion `zipWith`

Betrachten wir nun zwei weitere Funktionen `addlist` und `multlist`, die die Elemente zweier Listen durch Addition bzw. Multiplikation verknüpfen.

```
addlist :: Num a => [a] -> [a] -> [a]
addlist (x:xs) (y:ys) = (x + y) : addlist xs ys
addlist _      _      = []

multlist :: Num a => [a] -> [a] -> [a]
multlist (x:xs) (y:ys) = (x * y) : addlist xs ys
multlist _      _      = []
```

Da solche Funktionen wiederum sehr ähnlich implementiert werden, wollen wir nun die Funktion `zipWith` vorstellen, die das hierbei verwendete Rekursionsmuster realisiert. Diese (ebenfalls vordefinierte) Funktion arbeitet ähnlich wie `map`, sie wendet aber eine Funktion auf zwei Argumente an.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys)   = f x y : zipWith f xs ys
zipWith _ _      _        = []
```

Damit lässt sich nun das Rekursionsmuster für die Kombination von zwei Listen durch jeweiliges Anwenden einer Funktion auf Paare von Elementen beider Listen formulieren. Die nicht-rekursiven Definitionen von `addlist` und `multlist` lauten wie folgt.

```
addlist  = zipWith (+)
multlist = zipWith (*)
```

`filter`-Funktionen

Betrachten wir eine Funktion `dropEven`, die aus einer Liste von Zahlen alle geraden Zahlen löscht und eine Funktion `dropUpper`, die aus einer Liste von Zeichen alle Großbuchstaben löscht. Die Hilfsfunktionen `odd` und `isLower` sind (im Modul `Char`) vordefiniert.

```
dropEven :: [Int] -> [Int]
dropEven [] = []
dropEven (x:xs) | odd x     = x : dropEven xs
                | otherwise = dropEven xs

dropUpper :: [Char] -> [Char]
dropUpper [] = []
dropUpper (x:xs) | isLower x = x : dropUpper xs
                 | otherwise = dropUpper xs
```

Beispielsweise ergibt `dropEven [1,2,3,4]` das Resultat `[1,3]` und `dropUpper "GmbH"` das Resultat `"mb"`.

Man erkennt, dass die beiden Funktionen `dropEven` und `dropUpper` sehr ähnlich sind, da beide eine Liste durchlaufen und dabei alle Elemente löschen, die eine bestimmte Bedingung (d.h., ein bestimmtes Prädikat) nicht erfüllen. Es liegt auf der Hand, diese beiden Funktionen durch ein und dieselbe Funktion zu realisieren. Dazu sind folgende Schritte notwendig:

- Abstraktion vom Datentyp der Listenelemente (`Int` bzw. `Char`). Dies ist wiederum nur in Sprachen mit (parametrischer) Polymorphie möglich.

- Abstraktion von dem Prädikat (d.h., der booleschen Funktion), mit der die Listenelemente gefiltert werden sollen (`dropEven` bzw. `isLower`). Dies ist nur in Sprachen möglich, in denen Funktionen als gleichberechtigte Datenobjekte behandelt werden.

Die allgemeine Form von Funktionen dieser Bauart ist demnach wie folgt (wobei `g` das Prädikat ist, mit dem gefiltert wird).

```
f :: [a] -> [a]
f [] = []
f (x:xs) | g x       = x : f xs
         | otherwise = f xs
```

Da `g` eine beliebige boolesche Funktion vom Typ `a -> Bool` ist, sollte sie zusätzliches Eingabeargument der Funktion sein. Auf diese Weise erhält man die Funktion `filter` (wobei `f` von oben nun der Funktion `filter g` entspricht). Sie ist in HASKELL bereits vordefiniert.

```
filter :: (a -> Bool) -> [a] -> [a]
filter g [] = []
filter g (x:xs) | g x       = x : filter g xs
                | otherwise = filter g xs
```

Die weiter oben definierten Funktionen `dropEven` und `dropUpper` lassen sich nun wie folgt auf nicht-rekursive Weise definieren:

```
dropEven  = filter odd
dropUpper = filter isLower
```

Analog kann man entsprechende `filter`-Funktionen auf eigenen Datenstrukturen definieren, z.B. auf den selbst definierten Listen. Generell gilt: `filter`-Funktionen löschen alle Teilwerte eines zusammengesetzten Datenobjekts, die die Bedingung `g` nicht erfüllen.

`fold`-Funktionen

Betrachten wir eine Funktion `add`, die alle Zahlen in einer Liste addiert und eine Funktion `prod`, die alle Zahlen in einer Liste multipliziert. Wir illustrieren dies zunächst an unserem selbstdefinierten Datentyp für Listen, da dies einfacher ist.

```
plus :: Int -> Int -> Int
plus x y = x + y

times :: Int -> Int -> Int
times x y = x * y

add :: (List Int) -> Int
add Nil         = 0
add (Cons x xs) = plus x (add xs)

prod :: (List Int) -> Int
prod Nil         = 1
prod (Cons x xs) = times x (prod xs)
```

Bei einem Aufruf von `add` mit dem Argument

$$\texttt{Cons}\,\texttt{x}_1\,(\texttt{Cons}\,\texttt{x}_2\,(\ldots(\texttt{Cons}\,\texttt{x}_{n-1}\,(\texttt{Cons}\,\texttt{x}_n\,\texttt{Nil}))\ldots))$$

erhält man das Ergebnis

$$\texttt{plus}\,\texttt{x}_1\,(\texttt{plus}\,\texttt{x}_2\,(\ldots(\texttt{plus}\,\texttt{x}_{n-1}\,(\texttt{plus}\,\texttt{x}_n\,\texttt{0}))\ldots)).$$

Der Konstruktor `Cons` wird also durch die Funktion `plus` ersetzt und der Konstruktor `Nil` durch die Zahl `0`. Analog erhält man beim Aufruf von `prod` mit demselben Argument das Resultat

$$\texttt{times}\,\texttt{x}_1\,(\texttt{times}\,\texttt{x}_2\,(\ldots(\texttt{times}\,\texttt{x}_{n-1}\,(\texttt{times}\,\texttt{x}_n\,\texttt{1}))\ldots)).$$

Hier wird der Konstruktor `Cons` also durch `times` und der Konstruktor `Nil` durch `1` ersetzt.

Wiederum ist unser Ziel, eine Funktion höherer Ordnung anzugeben, die das Rekursionsmuster dieser beiden Funktionen implementiert. Diese Funktion kann dann zur Implementierung von `add` und `prod` wiederverwendet werden.

Allgemein wird also der Konstruktor `Cons` durch eine Funktion `g` und der Konstruktor `Nil` durch einen Initialwert `e` ersetzt. Man benötigt also eine Funktion `f`, so dass man beim Aufruf von `f` mit dem Argument

$$\texttt{Cons}\,\texttt{x}_1\,(\texttt{Cons}\,\texttt{x}_2\,(\ldots(\texttt{Cons}\,\texttt{x}_{n-1}\,(\texttt{Cons}\,\texttt{x}_n\,\texttt{Nil}))\ldots))$$

das folgende Ergebnis erhält.

$$\texttt{g}\,\texttt{x}_1\,(\texttt{g}\,\texttt{x}_2\,(\ldots(\texttt{g}\,\texttt{x}_{n-1}\,(\texttt{g}\,\texttt{x}_n\,\texttt{e}))\ldots))$$

Wiederum muss man vom Typ der Listenelemente und von den Funktionen `g` und `e` abstrahieren. Die allgemeine Form von Funktionen dieser Bauart ist demnach wie folgt.

```
f :: (List a) -> b
f Nil         = e
f (Cons x xs) = g x (f xs)
```

Hierbei ist der Initialwert `e` vom Typ `b` des Ergebnisses und die Hilfsfunktion `g` muss den Typ `a -> b -> b` haben. Da `e` und `g` beliebige Funktionen sind, sollten sie zusätzliche Eingabeargumente der Funktion sein. Auf diese Weise erhält man die Funktion `fold` (wobei `f` von oben nun der Funktion `fold g e` entspricht).

```
fold :: (a -> b -> b) -> b -> (List a) -> b
fold g e Nil         = e
fold g e (Cons x xs) = g x (fold g e xs)
```

Das Ergebnis von

$$\texttt{fold}\,\texttt{g}\,\texttt{e}\,(\texttt{Cons}\,\texttt{x}_1\,(\texttt{Cons}\,\texttt{x}_2\,(\ldots(\texttt{Cons}\,\texttt{x}_{n-1}\,(\texttt{Cons}\,\texttt{x}_n\,\texttt{Nil}))\ldots)))$$

ist also

$$\texttt{g}\,\texttt{x}_1\,(\texttt{g}\,\texttt{x}_2\,(\ldots(\texttt{g}\,\texttt{x}_{n-1}\,(\texttt{g}\,\texttt{x}_n\,\texttt{e}))\ldots)).$$

Damit sind nun neue (nicht-rekursive) Definitionen von `add` und `prod` möglich.

```
add  = fold plus  0
prod = fold times 1
```

Ein weiteres Beispiel ist die Funktion `conc`, die eine Liste von Listen als Eingabe erhält und als Ergebnis die Konkatenation aller dieser Listen liefert. Hierbei verwenden wir den folgenden Algorithmus `append`.

```
append :: List a -> List a -> List a
append Nil         ys  = ys
append (Cons x xs) ys  = Cons x (append xs ys)
```

Bei einem Aufruf von `conc` mit dem Argument

$$\texttt{Cons}\,\texttt{l}_1\,(\texttt{Cons}\,\texttt{l}_2\,(\ldots(\texttt{Cons}\,\texttt{l}_{n-1}\,(\texttt{Cons}\,\texttt{l}_n\,\texttt{Nil}))\ldots))$$

erhält man das Ergebnis

$$\texttt{append}\,\texttt{l}_1\,(\texttt{append}\,\texttt{l}_2\,(\ldots(\texttt{append}\,\texttt{l}_{n-1}\,(\texttt{append}\,\texttt{l}_n\,\texttt{Nil}))\ldots)).$$

Die Implementierung von `conc` ist mit Hilfe des durch `fold` realisierten Rekursionsmusters leicht möglich.

```
conc :: List (List a) -> List a
conc = fold append Nil
```

Eine analoge Version zu `fold` auf den vordefinierten Listen ist in HASKELL vordefiniert. Sie heißt `foldr` und ist wie folgt definiert:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr g e []     = e
foldr g e (x:xs) = g x (foldr g e xs)
```

Realisiert man z.B. `add`, `prod`, und `conc` mit den vordefinierten Listen, so ist also `foldr` statt `fold` zu verwenden. Die Funktion `add` ist unter dem Namen `sum` und `conc` ist unter dem Namen `concat` bereits in HASKELL vordefiniert.

```
add :: [Int] -> Int
add  = foldr plus  0

prod :: [Int] -> Int
prod = foldr times 1

conc :: [[a]] -> [a]
conc = foldr (++) []
```

Betrachten wir nun auch noch `fold` auf der Datenstruktur der Vielwegbäume.

```
data Tree a = Node a [Tree a]
```

Die folgende Funktion ersetzt alle Vorkommen des Konstruktors `Node` durch die Funktion `g`.

```
foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree g (Node x ts) = g x (map (foldTree g) ts)
```

Falls `t` der Baum

$$\texttt{Node}\ \texttt{x}_1\ \texttt{[Node}\ \texttt{x}_2\ \texttt{[]]}$$

ist, so ist `foldTree g t` also

$$\texttt{g}\ \texttt{x}_1\ \texttt{[g}\ \texttt{x}_2\ \texttt{[]]}.$$

Um die Verwendung von `foldTree` zu verdeutlichen, implementieren wir die Funktion `addTree`, die alle Zahlen in den Knoten eines Baums addiert. Man benötigt hierzu noch eine Funktion `addtolist`, wobei $\texttt{addtolist}\ \texttt{x}\ [\texttt{y}_1, \ldots, \texttt{y}_n] = \texttt{y}_1\texttt{+}(\texttt{y}_2\texttt{+}\ldots\texttt{+}(\texttt{y}_n\texttt{+x})\ldots)$ ist.

```
addtolist :: Num a => a -> [a] -> a
addtolist = foldr (+)

addTree :: Num a => Tree a -> a
addTree = foldTree addtolist
```

Man erhält demnach `addTree (Node 1 [Node 2 []])` = `addtolist 1 [addtolist 2 []]` = `3`.

Generell gilt also: `fold`-Funktionen ersetzen die Konstruktoren einer Datenstruktur durch anzugebende Funktionen `g`, `e`, etc.

Listenkomprehension

In der Mathematik benutzt man häufig Mengenschreibweisen wie $\{x * x \mid x \in \{1, \ldots, 5\}, odd(x)\}$. An diese Schreibweise angelehnt, bieten funktionale Sprachen wie HASKELL eine alternative Listenschreibweise (sogenannte *Listenkomprehensionen*), um Berechnungen mit Hilfe von `map` elegant auszudrücken. Der Ausdruck

```
[ x * x | x <- [1 .. 5], odd x]
```

wertet in HASKELL zu der Liste `[1,9,25]` aus. Hierbei ist `[a .. b]` allgemein eine Kurzschreibweise für die Liste `[a,a+1,...,b]`. Dann bedeutet der obige Ausdruck die Liste aller Quadratzahlen `x * x`, wobei `x` alle Zahlen von `1` bis `5` durchläuft, wir aber nur die ungeraden Zahlen davon betrachten.

Formal hat eine Listenkomprehension die Gestalt [exp|qual$_1$, ..., qual$_n$], wobei exp ein Ausdruck ist und die qual$_i$ sogenannte *Qualifikatoren* sind. Qualifikatoren teilen sich auf in *Generatoren* und *Einschränkungen* (Guards). Ein Generator hat die Form var <- exp, wobei exp ein Ausdruck von einem Listentyp ist. Ein Beispiel für einen Generator ist `x <- [1 .. 5]`. Die Bedeutung hiervon ist, dass die Variable var alle Werte aus der Liste exp annehmen kann. Eine Einschränkung ist ein boolescher Ausdruck wie `odd x`, der die möglichen Werte der im Ausdruck vorkommenden Variablen einschränkt. Wir müssen also die Grammatikregeln zur Definition von Ausdrücken um folgende Regeln erweitern:[2]

exp → [exp | qual$_1$, ..., qual$_n$], wobei $n \geq 1$

qual → var <- exp | exp

Die Bedeutung von Listenkomprehensionen ist wie folgt als Abkürzung für einen Ausdruck mit der Funktion `map` definiert. Hierbei ist `concat` wieder die Funktion, die eine Liste von Listen zu einer einzigen Liste verschmilzt. `Q` steht für eine Liste von Qualifikatoren. Falls diese leer ist, so ist [exp|Q] als [exp] zu lesen.

```
[exp | var <- exp', Q]  =  concat (map f exp') where f var = [exp | Q]
[exp | exp', Q]         =  if exp' then [exp | Q] else []
```

Die erste Regel heißt *Generatorregel* und die zweite bezeichnet man als *Einschränkungsregel*. Die Qualifikatoren werden also der Reihe nach abgearbeitet. Dabei bedeutet die Generatorregel:

```
  [exp | var <- [a1,...,an], Q]
= concat (map f [a1,...,an]) where f var = [exp | Q]
= f a1 ++...++ f an where f var = [exp | Q]
= [exp | Q] [var/a1] ++...++ [exp | Q] [var/an].
```

Man bildet also [exp | Q], wobei var alle Werte aus [a$_1$, ..., a$_n$] durchläuft.

[2] In HASKELL sind sogar Patterns statt nur Variablen in Generatoren möglich und darüber hinaus sind auch lokale Deklarationen in Qualifikatoren erlaubt.

Mit den obigen Regeln kann nun der Ausdruck `[x * x | x <- [1 .. 5], odd x]` ausgewertet werden:

```
   [x * x | x <- [1 .. 5], odd x]
=  concat (map f [1 .. 5]) where f x = [x * x | odd x]
=  concat [f 1, f 2, f 3, f 4, f 5] where f x = [x * x | odd x]
=  f 1 ++ f 2 ++ f 3 ++ f 4 ++ f 5 where f x = [x * x | odd x]
=  f 1 ++ f 2 ++ f 3 ++ f 4 ++ f 5 where f x = if odd x then [x * x] else []
=  [1] ++ [] ++ [9] ++ [] ++ [25]
=  [1,9,25]
```

Die folgenden Beispiele sollen verdeutlichen, dass die Reihenfolge von Qualifikatoren eine Rolle spielt. So wertet `[(a, b) | a <- [1 .. 3], b <- [1 .. 2]]` zu

```
[(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)]
```

aus, wohingegen `[(a, b) | b <- [1 .. 2], a <- [1 .. 3]]` zu

```
[(1,1), (2,1), (3,1), (1,2), (2,2), (3,2)]
```

auswertet. Spätere Qualifikatoren können von vorher eingeführten Variablen abhängen. So wertet `[(a,b) | a <- [1 .. 4], b <- [a+1 .. 4]]` zu

```
[(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)]
```

aus. Generatoren und Einschränkungen können in beliebiger Reihenfolge auftreten. Die Auswertung von `[(a,b) | a <- [1 .. 4], even a, b <-[a+1 .. 4], odd b]` ergibt daher nur `[(2,3)]`.

Auch die Funktion `map` lässt sich mit Listenkomprehension wie folgt definieren.

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

Als letztes Beispiel zeigen wir, wie man den bekannten *Quicksort*-Algorithmus mit Hilfe von Listenkomprehension auf äußerst einfache Weise implementieren kann.

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort l1 ++ [x] ++ qsort l2
               where l1 = [y | y <- xs, y <  x]
                     l2 = [y | y <- xs, y >= x]
```

Wenn man dies mit der entsprechenden Implementierung in einer imperativen Sprache vergleicht, wird deutlich, dass funktionale Programme oftmals wirklich deutlich kürzer und lesbarer sind. Eine Implementierung in JAVA lautet beispielsweise wie folgt.[3]

[3] Um ein Array `a` zu sortieren, muss man hierbei `qsort(a,0,a.length-1)` aufrufen.

```
static void qsort(int[] a, int lo, int hi) {
      int h, l, p, t;

      if (lo <= hi) {
         l = lo;
         h = hi;
         p = a[hi];

         do {

           while ((l < h) && (a[l] <= p))
                 l = l+1;

           while ((h > l) && (a[h] >= p))
                 h = h-1;


           if (l < h) {
              t = a[l];
              a[l] = a[h];
              a[h] = t;
           }

         } while (l < h);

         t = a[l];
         a[l] = a[hi];
         a[hi] = t;

         qsort( a, lo, l-1);
         qsort( a, l+1, hi);
      }
}
```

1.3 Programmieren mit Lazy Evaluation

Die Programmiersprache HASKELL verwendet eine nicht-strikte Auswertungsstrategie:

- Generell findet die Auswertung mit der leftmost outermost Strategie statt.
- Vordefinierte arithmetische Operatoren und Vergleichsoperatoren erfordern jedoch zunächst die Auswertung ihrer Argumente.
- Beim Pattern Matching werden die Argumente nur so weit ausgewertet, bis entschieden werden kann, welcher Pattern matcht.

Diese Punkte werden durch folgendes Beispiel verdeutlicht.

```
infinity :: Int
infinity = infinity + 1

mult :: Int -> Int -> Int
mult 0 y = 0
mult x y = x * y
```

Die Auswertung von `mult 0 infinity` terminiert mit dem Ergebnis `0`. Hingegen führt die Auswertung des Ausdrucks `0 * infinity` zur Nicht-Terminierung. Hierdurch wird der Unterschied zwischen der generellen nicht-strikten Auswertung (bei `mult`) und der Auswertung bei vordefinierten Operatoren (wie `*`) deutlich. Man erkennt auch, dass bereits ohne Auswertung des Arguments `infinity` festgestellt werden kann, dass die Patterns der ersten definierenden Gleichung von `mult` auf die Argumente `0` und `infinity` matchen.

In HASKELL ist die Definition von unendlichen Datenobjekten möglich. Betrachten wir hierzu den folgenden Algorithmus.

```
from :: Num a => a -> [a]
from x = x : from (x+1)
```

Der Ausdruck `from x` entspricht der unendlichen Liste `[x, x+1, x+2, ...]`. Sie kann in HASKELL auch als `[x ..]` geschrieben werden. Obwohl die Auswertung des Ausdrucks `from 5` natürlich nicht terminiert, können solche unendlichen Listen dennoch sehr nützlich für die Programmierung sein. In HASKELL ist eine Funktion `take` vordefiniert, die das erste Teilstück einer Liste zurückliefert. Es gilt also `take n` $[x_1, \ldots, x_n, x_{n+1}, \ldots] = [x_1, \ldots, x_n]$.

```
take :: Int -> [a] -> [a]
take _   []      = []
take n   (x:xs)  = if n <= 0 then [] else x : take (n-1) xs
```

Da beim Pattern Matching das Argument immer nur so weit wie nötig ausgewertet wird, ergibt sich

```
    take 2 (from 5)
=   take 2 (5 : from 6)
=   5 : take 1 (from 6)
=   5 : take 1 (6 : from 7)
=   5 : 6 : take 0 (from 7)
=   5 : 6 : []
=   [5,6].
```

Die Auswertung von `take 2 (from 5)` terminiert also. Funktionen, die niemals definiert sind, falls die Auswertung eines ihrer Argumente undefiniert ist, heißen *strikt*. Beispiele hierfür sind also arithmetische Grundoperationen und Vergleichsoperationen. Die Funktionen `mult` und `take` hingegen sind *nicht-strikt*.

Programmieren mit unendlichen Datenobjekten

Das generelle Vorgehen beim Programmieren mit unendlichen Datenobjekten ist wie folgt: Man erzeugt zuerst eine potentiell unendliche Liste von Approximationen an die Lösung. Anschließend filtert man daraus die wirklich gesuchte Lösung heraus.

Als Beispiel betrachten wir einen Algorithmus zur Bestimmung von Primzahlen, der ähnlich zum "*Sieb des Eratosthenes*" vorgeht. Der Algorithmus arbeitet wie folgt:

1. Erstelle die Liste aller natürlichen Zahlen beginnend mit `2`.
2. Markiere die erste unmarkierte Zahl in der Liste.
3. Streiche alle Vielfachen der letzten markierten Zahl.
4. Gehe zurück zu Schritt 2.

Man beginnt also mit der Liste `[2,3,4,...]`. Wenn wir das Markieren durch Unterstreichung deutlich machen, markiert man nun die erste (unmarkierte) Zahl `2`. Dies ergibt die Liste `[2,3,4,...]`. Nun werden alle Vielfachen der letzten markierten Zahl (d.h. `2`) gestrichen. Dies führt zu der Liste `[2,3,5,7,9,11,...]`. Nun wird die nächste unmarkierte Zahl in der Liste markiert, was `[2,3,5,7,9,11,...]` ergibt. Anschließend werden die Vielfachen der letzten markierten Zahl (d.h. `3`) gestrichen. Dies ergibt `[2,3,5,7,11,13,17,...]`. Man erkennt, dass im Endeffekt nur die Liste der Primzahlen übrig bleibt.

Diese natürlichsprachliche Beschreibung des Vorgehens kann man bei der Verwendung unendlicher Datenobjekte und der nicht-strikten Auswertung nahezu direkt in Programmcode überführen. Hierbei muss man natürlich mit unendlichen Listen umgehen (denn in der Tat entsteht dabei ja die unendliche Liste aller Primzahlen). Wenn man aber nur an den ersten 100 Primzahlen oder allen Primzahlen kleiner als `42` interessiert ist, so muss das Sieben nur für ein endliches Anfangsstück der natürlichen Zahlen durchgeführt werden.

Die Implementierung von Schritt 1 ist naheliegend, da `from 2` oder `[2 ..]` die unendliche Liste der natürlichen Zahlen ab `2` berechnet. Zum Streichen von Vielfachen einer Zahl `x` aus einer Liste `xs` benutzen wir die folgende Funktion. Sie berechnet alle Elemente `y` aus `xs`, die nicht durch `x` teilbar sind. Hierbei ist `y` durch `x` teilbar, falls sich bei der Ganzzahldivision kein Rest `y ‘mod‘ x` ergibt.

```
drop_mult :: Int -> [Int] -> [Int]
drop_mult x xs = [y | y <- xs, y ‘mod‘ x /= 0]
```

Der Ausdruck `drop_mult 2 [3 ..]` berechnet also die unendliche Liste `[3,5,7,9,11,...]`.

Um wiederholt alle Vielfachen der ersten unmarkierten Zahl zu löschen, verwenden wir die Funktion `dropall`. Sie löscht zunächst alle Vielfachen des ersten Elements aus dem Rest einer Liste. Anschließend ruft sie sich rekursiv auf der entstehenden Liste ohne ihr erstes Element auf. Nun werden also die Vielfachen des zweiten Elements gelöscht, etc.

```
dropall :: [Int] -> [Int]
dropall (x:xs) = x : dropall (drop_mult x xs)
```

Die Liste `primes` aller Primzahlen kann dann wie folgt berechnet werden.

```
primes :: [Int]
primes = dropall [2 ..]
```

Bei der Auswertung von `primes` ergibt sich also die unendliche Liste

`[2,3,5,7,11,13,17,19,23,29,31,37,41,43,...].`

Die Liste der ersten 100 Primzahlen berechnet der Ausdruck `take 100 primes`. Man erkennt also, dass man tatsächlich mit unendlichen Datenobjekten rechnen kann, falls während der Rechnung immer nur ein endlicher Teil dieser Objekte betrachtet wird.

Um die Liste aller Primzahlen zu berechnen, die kleiner als `42` sind, ist der Ausdruck `[ x | x <- primes, x < 42]` nicht geeignet. Dessen Auswertung terminiert nämlich nicht, da die Bedingung `x < 42` auf allen (unendlich vielen) Elementen von `primes` getestet werden muss. (Der Auswerter kann ja nicht wissen, dass es sich um eine monoton steigende Folge handelt.) Stattdessen sollte man die in HASKELL vordefinierte Funktion `takeWhile` verwenden.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs) | p x       = x : takeWhile p xs
                   | otherwise = []
```

Die Auswertung von `takeWhile` hält an, sobald für ein Listenelement die Bedingung `p` nicht mehr zutrifft. Die Auswertung des Ausdrucks `takeWhile (< 42) primes` terminiert daher und liefert in der Tat die Liste `[2,3,5,7,11,13,17,19,23,29,31,37,41]`.

Zirkuläre Datenobjekte

Um die Effizienz beim Rechnen mit unendlichen Datenobjekten zu erhöhen, sollte man versuchen, solche Datenobjekte (falls möglich) auf zirkuläre Weise im Speicher zu repräsentieren. Das einfachste Beispiel ist die unendliche Liste `ones` von der Form `[1,1,1,1,...]`.

```
ones :: [Int]
ones = 1 : ones
```

Wenn eine Nicht-Funktions-Variable wie `ones` in ihrer eigenen Definition auftritt, wird das entstehende Datenobjekt als zyklisches Objekt wie in Abb. 1.2 gespeichert. Der Vorteil solcher Objekte liegt in ihrem geringen Speicherbedarf und darin, dass (teilweise) Berechnungen dadurch nur einmal statt mehrmals durchgeführt werden.

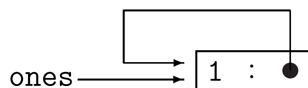


Abbildung 1.2: Zirkuläres Datenobjekt

Um diesen Effizienzgewinn zu illustrieren, betrachten wir das Hamming-Problem (nach dem Mathematiker W. R. Hamming), das sich mit Hilfe von unendlichen (und zirkulären) Datenobjekten sehr effizient lösen lässt. Die Aufgabe besteht darin, eine Liste mit folgenden Eigenschaften zu generieren:

- Die Liste ist aufsteigend sortiert und es gibt keine Duplikate.
- Die Liste beginnt mit `1`.
- Wenn die Liste das Element x enthält, dann enthält sie auch die Elemente $2x$, $3x$ und $5x$.
- Außer diesen Zahlen enthält die Liste keine weiteren Elemente.

Die Liste hat also folgende Gestalt.

```
[1,2,3,4,5,6,8,9,10,12,15,16,...]
```

Das Hamming-Problem wird oft verwendet, um Programmiersprachen auf ihre Eignung zur effizienten Implementierung bestimmter Klassen von Algorithmen zu untersuchen.

Die Idee für eine effiziente Implementierung dieses Problems ist, eine Funktion `mer` (für "merge") zu verwenden, die zwei (potentiell unendliche) geordnete Listen zu einer einzigen geordneten Liste ohne Duplikate verschmilzt.

```
mer Ord a => [a] -> [a] -> [a]
mer (x : xs) (y: ys) | x < y     = x : mer xs (y:ys)
                     | x == y    = x : mer xs ys
                     | otherwise = y : mer (x:xs) ys
```

Die Funktion `hamming` lässt sich nun wie folgt definieren:

```
hamming :: [Int]
hamming = 1 : mer (map (2*) hamming)
                  (mer (map (3*) hamming)
                       (map (5*) hamming))
```

Zu Anfang wird `hamming` durch ein zyklisches Objekt repräsentiert, in dem die drei Vorkommen von `hamming` auf der rechten Seite wieder durch Zeiger auf den Gesamtausdruck realisiert sind. Es ist eine gute Übung, einige Schritte der Auswertung von `hamming` auf diesem zyklischen Objekt nachzuverfolgen. Man erkennt, dass die Berechnung eines neuen Listenelements von `hamming` höchstens drei Multiplikationen (um die ersten Elemente von `(map (2*) hamming)`, `(map (3*) hamming)` und `(map (5*) hamming)` zu berechnen) und vier Vergleiche (für `<` und `==` in beiden `mer`-Aufrufen) benötigt. Die Berechnungszeit ist also *linear* in der Anzahl der benötigten Elemente von `hamming`. Das heißt, die Komplexität der Berechnung von `take n hamming` ist $O(\mathtt{n})$. Falls möglich, so sollte man daher immer versuchen, Nicht-Funktions-Variablen, die unendliche Datenobjekte realisieren, so zu definieren, dass diese Variablen auf der rechten Seite ihrer Definition wieder auftreten. Wie erwähnt, kann man auf diese Weise zyklische Datenobjekte erzeugen. Zusammenfassend erkennt man, dass unendliche Datenstrukturen sowohl für die Effizienz als auch für die Klarheit des Programmcodes sehr hilfreich sein können.

1.4 Monaden

Monaden sind ein Konzept aus der Kategorientheorie, das in funktionalen Programmen zur Programmierung von Sequentialität, zur Ein- und Ausgabe, zur Fehler- oder Ausnahmebehandlung und zur Erhöhung der Modularität und Änderungsfreundlichkeit verwendet wird. Wir werden zunächst in Abschnitt 1.4.1 zeigen, wie Monaden für die Ein- und Ausgabe eingesetzt werden und anschließend in Abschnitt 1.4.2 auf generelles Programmieren mit Monaden eingehen.

1.4.1 Ein- und Ausgabe mit Monaden

Eine grundlegende Eigenschaft (rein) funktionaler Programmiersprachen ist, dass Programme keine Seiteneffekte (auf die Programmvariablen) haben. Der Wert eines Ausdrucks ist also stets derselbe, d.h., er hängt nicht von der Umgebung ab. Wie erwähnt, wird dieses Konzept als *referentielle Transparenz* bezeichnet. Ein- und Ausgabe sind jedoch (gewünschte) Seiteneffekte (auf die Umwelt). Wenn wir eine Funktion hätten, die ein Zeichen von der Tastatur einliest und dieses als Resultat zurückliefert, würden wir aber das Konzept der referentiellen Transparenz verletzen. Die Auswertung dieser Funktion würde nämlich nicht jedes Mal dasselbe Resultat liefern, sondern das Ergebnis hinge von der Umwelt, d.h. dem Benutzer ab, der das Zeichen eingibt. Die Frage ist daher, wie man Ein- und Ausgabe verwenden kann, ohne die referentielle Transparenz zu zerstören.

Hierzu verwenden wir einen vordefinierten Datentyp `IO ()`, dessen Werte *Aktionen* sind. Die Auswertung eines Ausdrucks vom Typ `IO ()` bedeutet dann, dass die Aktion ausgeführt wird. Bei `IO ()` handelt es sich um einen abstrakten Datentyp, bei dem die Repräsentation seiner Werte vor dem Benutzer versteckt ist. Wichtig ist hierbei nur, welche Operationen der Typ dem Benutzer zur Verfügung stellt. So existiert z.B. eine vordefinierte Funktion `putChar` zur Ausgabe von Zeichen.

```
putChar :: Char -> IO ()
```

Der Wert des Ausdrucks `putChar '!'` ist also eine Aktion, die ein `!` ausgibt.

Weiter existiert eine vordefinierte Funktion `>>` (genannt “then”) zur Kombination von Aktionen.

```
(>>) :: IO () -> IO () -> IO ()
```

Der Wert des Ausdrucks `x >> y` ist also eine Aktion, bei der zunächst die Aktion `x` und dann die Aktion `y` durchgeführt wird. Gibt man in den Interpreter den Ausdruck

```
putChar 'a' >> putChar 'b'
```

ein, so wird dieser Ausdruck ausgewertet und der Effekt ist, dass `ab` auf dem Bildschirm ausgegeben wird.

Zur Erzeugung einer leeren Aktion existiert die Funktion `return`.

```
return () :: IO ()
```

Der Wert des Ausdrucks `return ()` ist die leere Aktion, d.h., bei der Auswertung von `putChar '!' >> return ()` wird ebenfalls nur `!` auf dem Bildschirm ausgegeben.

Selbstverständlich kann man auch rekursive Algorithmen auf dem Datentyp `IO ()` definieren. Hierzu betrachten wir die (vordefinierte) Deklaration einer Funktion, die einen String auf dem Bildschirm ausgibt.

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (x:xs) = putChar x >> putStr xs
```

Der Datentyp `IO ()` enthält nur Ausgabeaktionen. Zur Behandlung von Eingabeaktionen muss er daher verallgemeinert werden. Hierzu verwenden wir einen vordefinierten Typ `IO a` von Aktionen, die jeweils einen Wert Typ `a` berechnen. Beispielsweise ist die Funktion

```
getChar :: IO Char
```

vordefiniert. Der Wert des Ausdrucks `getChar` ist eine Aktion, die ein Zeichen von der (Standard-)eingabe (d.h. der Tastatur) einliest. Der Wert von `getChar` ist also *nicht das eingelesene Zeichen*, sondern die Aktion "lese ein". Wir stellen solche Aktionen bildlich als

dar. Dies macht deutlich, dass eine Ein-/Ausgabeaktion erfolgt und in die Aktion gekapselt dabei ein Wert des Typs `a` bestimmt wird.

Der Typ `IO ()` ist ein Spezialfall des Typs `IO a`. Der Typ `()` besitzt nur ein einziges Element (nämlich den leeren Tupel `()`). Man behandelt daher Aktionen ohne Eingabe so, als ob der feste Wert `()` eingelesen werden würde.

Wir erweitern auch die Funktion `return`, um leere Aktionen beliebiger `IO`-Typen zu generieren:

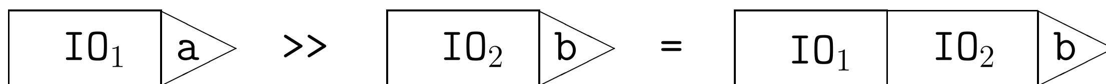
```
return :: a -> IO a
```

Damit ist `return '!'` also die Aktion vom Typ `IO Char`, die nichts tut und bei der sich das Zeichen `'!'` ergibt.

Die Erweiterung der Funktion `>>` zur Kombination von Aktionen ist ein Operator mit folgendem Typ.

```
(>>) :: IO a -> IO b -> IO b
```

Wenn `p` und `q` Aktionen sind, so ist `p >> q` eine Aktion, die zunächst `p` ausführt, den dabei bestimmen Wert ignoriert und dann `q` ausführt. Wenn `p` vom Typ `IO a` und `q` vom Typ `IO b` ist, so ergibt sich also folgendes Bild:



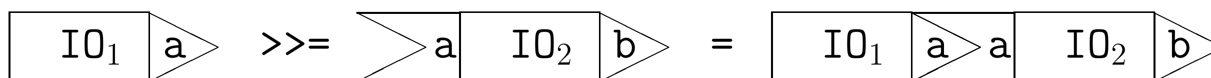
Beispielsweise wird bei Auswertung des Ausdrucks `getChar >> return ()` zunächst ein Zeichen von der Tastatur eingelesen. Anschließend wird die leere Aktion ausgeführt und die Gesamtaktion ist beendet. Der Typ des Ausdrucks `getChar >> return ()` ist

IO (). Ausdrücke dieses Typs können direkt im Interpreter ausgewertet werden und die entsprechenden Aktionen werden dabei ausgeführt. Der Wert des Ausdrucks (der ja diese Aktion ist) wird nicht angezeigt (es existiert also keine entsprechende show-Funktion für Aktionen).

Die Komposition p >> q von zwei Aktionen ist natürlich nur dann sinnvoll, wenn der in der Aktion p bestimmte Wert nicht weiter interessant ist, denn die Aktion q kann nicht von diesem Wert abhängen. Wir haben ja gesehen, dass der von p bestimmte Wert (vom Typ a) bei der Komposition von Aktionen mit >> einfach ignoriert wird. Wenn wir zwei Aktionen hintereinander ausführen wollen und die zweite Aktion vom Wert abhängt, der in der ersten Aktion bestimmt wird, so benötigen wir einen weiteren vordefinierten Operator >>= (genannt "bind").

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Wenn der Ausdruck p >>= f ausgewertet wird, so wird zunächst die Aktion p ausgeführt. Hierbei wird ein Wert x des Typs a bestimmt. Anschließend wird f x ausgewertet, wodurch schließlich ein Wert vom Typ b bestimmt wird. Als Schaubild lässt sich dies wie folgt darstellen:



Es ist leicht zu sehen, dass der "then"-Operator wie folgt mit Hilfe des "bind"-Operators definiert werden kann:

```
p >> q   =   p >>= \_ -> q
```

Nun können wir eine Funktion echo definieren, so dass der Wert des Ausdrucks echo eine Aktion ist, die erst ein Zeichen von der Tastatur liest und dieses dann auf dem Bildschirm ausgibt.

```
echo :: IO ()
echo = getChar >>= putChar
```

Die referentielle Transparenz bleibt erhalten. So ist z.B. der Wert der beiden Ausdrücke echo >> echo und let x = echo in x >> x derselbe. Dieser Wert ist die Aktion, die erst ein Zeichen einliest und dieses ausgibt und danach wieder ein (evtl. unterschiedliches) Zeichen einliest und dieses ebenfalls ausgibt.

Weitere vordefinierte primitive Funktionen zur Ein-/Ausgabe sind

```
readFile  :: String -> IO String
writeFile :: String -> String -> IO ().
```

Hierbei ist das Argument zu readFile der Name einer Datei und der Wert von readFile "myfile" ist eine Aktion, die die Datei myfile liest und ihren Inhalt bestimmt (d.h., der String ihres Inhalts ist in der entstehenden Aktion vom Typ IO String gekapselt). Die Aktion writeFile "myfile" "hallo" überschreibt den Inhalt der Datei myfile mit dem Wort hallo.

Betrachten wir nun eine Funktion zum Einlesen einer angegebenen Anzahl von Zeichen (d.h., gets n liest n Zeichen ein).

```
gets :: Int -> IO String
gets n = if n <= 0 then return []
                   else getChar >>= \x  ->
                                    gets (n-1)  >>= \xs ->
                                                    return (x:xs)
```

Falls nur `0` Zeichen eingelesen werden sollen, wird die leere Aktion mit dem leeren String als Ergebnis ausgeführt. Ansonsten wird zunächst die Aktion `getChar` ausgeführt, die ein Zeichen `x` einliest. Anschließend wird dieses Zeichen `x` genommen und die Aktion `gets (n-1)` ausgeführt. Diese Aktion liest einen String `xs` der Länge `n-1` ein. Schließlich wird auch `xs` genommen und man führt die leere Aktion `return (x:xs)` mit dem Ergebnis `(x:xs)` aus.

Diese Form der Hintereinanderausführung von Aktionen mit `>>=`, bei der jeweils das Resultat der letzten Aktion mit Hilfe eines Lambda-Ausdrucks weiterverwendet wird, kommt sehr häufig vor. Da die obige Notation mit Hilfe von `>>=` und Lambdas jedoch schlecht lesbar ist, wird dafür eine eigene Notation eingeführt. Anstelle von

```
p >>= \x ->
      q
```

schreiben wir einfach:

```
do { x <- p;
     q
   }
```

Dies bedeutet: "Setze die Variable `x` auf das bei der Aktion `p` bestimmte Ergebnis und führe dann `q` aus." Auch bei `do` kann man eine Offside-Regel verwenden, so dass man stattdessen auch

```
do x <- p
   q
```

schreiben kann. Analog kann man statt

```
p >>= \x ->
      q >>= \y ->
            r
```

nun folgendes schreiben:

```
do x <- p
   y <- q
   r
```

Allgemein verwendet man die folgenden Übersetzungsregeln, die die Bedeutung des `do`-Konstrukts definieren. Hierbei ist `S` eine nicht-leere Folge von Ausdrücken der Form `x <- p` oder `r`.

```
do {x <- p; S}  =  p >>= \x -> do {S}
do {p; S}       =  p >> do {S}
do {p}          =  p
```

Mit Hilfe dieser vereinfachten Schreibweise können wir nun die Funktion `gets` wie folgt schreiben:

```
gets :: Int -> IO String
gets n = if n <= 0 then return []
                   else do x  <- getChar
                           xs <- gets (n-1)
                           return (x:xs)
```

Auf diese Weise kann man einen imperativen Programmierstil in funktionalen Programmen verwenden (denn die `do`-Notation erinnert stark an Sequenzen von Zuweisungen in imperativen Programmen). Dies hat den Vorteil, dass dadurch die Sequentialität des Programms sichergestellt ist (d.h., `getChar` wird auf jeden Fall vor `gets n` und dies vor `return (x:xs)` ausgeführt). Solch eine Sequentialität ist bei Ein-/Ausgabe sehr wichtig (denn es soll unabhängig von der Auswertungsstrategie der funktionalen Sprache garantiert sein, dass bestimmte Aktionen vor anderen Aktionen stattfinden). Ein Beispiel hierfür wäre ein Programm, in dem erst ein bestimmtes Zeichen eingelesen werden muss, bevor es weiter verarbeitet wird. Für Ein-/Ausgabe ist daher ein Programmierstil mit solchen imperativen Eigenschaften besonders gut geeignet. Wichtig dabei ist aber, dass (anders als in wirklich imperativen Programmen) die referentielle Transparenz dennoch erhalten bleibt! Der Wert eines Ausdrucks ist stets derselbe, d.h., er hängt nicht von der Umgebung ab.

Der Typ `IO a` ermöglicht uns eine strenge Trennung zwischen Programmabschnitten mit Ein-/Ausgabe (und daher mit Seiteneffekten) und rein funktionalen Programmteilen. Seiteneffekte finden nur in `IO`-Aktionen statt, sie beeinflussen aber nie den Wert eines Ausdrucks. Es existiert also keine Funktion des Typs `IO a -> a`, die den in eine Aktion gekapselten Wert aus dieser herauszuextrahieren kann. Der gekapselte Wert kann nur in nachfolgenden Aktionen weiterverarbeitet werden. Dies stellt auch sicher, dass nur Ausdrücke eines Typs `IO a` Seiteneffekte auslösen können, während Ausdrücke anderer Typen niemals Seiteneffekte bewirken.

Der Grund dafür, dass Funktionen zur Extraktion von gekapselten Werten aus Aktionen verboten sind, ist, dass man ansonsten die referentielle Transparenz verletzen könnte. Hätte man eine Funktion `result :: IO a -> a`, die auf den in der Eingabeaktion gekapselten Wert zugreifen könnte, so wäre der Wert von `result getChar` das eingelesene Zeichen. Nun wäre die referentielle Transparenz zerstört, denn zwei verschiedene Aufrufe von `result getChar` könnten verschiedene Werte haben (der Wert hängt dann jeweils vom Zeichen ab, das der Benutzer eingibt). Beispielsweise bezeichnet `x` in dem Ausdruck

```
let x = getChar in x >> x
```

beides Mal die gleiche Aktion (`getChar`) und `x >> x` ist die Aktion "Führe `getChar` zweimal aus". `IO`-Aktionen sind also ähnlich wie Funktionen, so dass bei jedem Vorkommen die gleiche Funktion bezeichnet wird. Bei

```
let x = result getChar in ... x ... x ...
```

würde hingegen `x` einen bestimmten Wert bezeichnen, der an verschiedenen Stellen im Ausdruck `... x ... x ...` derselbe wäre. Der Wert des obigen Ausdrucks wäre also verschieden vom Wert des Ausdrucks

```
... result getChar ... result getChar ...
```

Aus diesem Grund sind Funktionen wie `result` verboten.

Funktionen zur Extraktion des gekapselten Werts fehlen nur bei der `IO`-Monade. Wir werden in Abschnitt 1.4.2 weitere Monaden kennen lernen (und der Benutzer kann Monaden auch selbst definieren). Bei anderen Monaden können Funktionen wie `result` natürlich vorhanden sein (und vom Benutzer programmiert werden). Der Grund für die Einschränkung bei der vordefinierten `IO`-Monade ist, dass nur in der `IO`-Monade Seiteneffekte stattfinden. Nur hier ist diese Einschränkung daher nötig, um die referentielle Transparenz zu erhalten. Auf Objekte des Typs `IO a` kann man nur mit Hilfe der vordefinierten Funktionen wie `return` und `>>=` zugreifen (d.h., es ist ein abstrakter Datentyp). Diese Funktionen sind bei anderen Monaden ebenfalls vorhanden, dort können aber weitere Funktionen definiert werden, die ebenfalls direkt auf Monadenobjekte zugreifen.

Ein weiterer Unterschied zwischen Ein-/Ausgabe in imperativen Sprachen und der monadischen Ein-/Ausgabe ist, dass nun Aktionen normale Werte sind. Man kann sie daher als Parameter an Funktionen übergeben, in Datenstrukturen speichern (d.h., man kann z.B. Listen von Aktionen generieren), etc.

1.4.2 Programmieren mit Monaden

Allgemein kann man Monaden wie folgt als Klasse in HASKELL deklarieren (diese Klasse ist bereits vordefiniert).

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

Anders als die Klassen `Eq`, `Show`, etc., ist `Monad` keine Typklasse (denn `m` ist kein Typ), sondern eine *Konstruktorklasse* (denn `m` ist ein einstelliger Typkonstruktor). Für jeden Typ `a` ist also auch `m a` ein Typ. Konstruktorklassen lassen sich in HASKELL natürlich auch unabhängig von Monaden verwenden.

Eine Monade dient zur Kapselung von Werten, d.h., in Objekten vom Typ `m a` ist ein Objekt vom Typ `a` gekapselt. Der Typkonstruktor `IO` ist ein Beispiel für eine Monade, d.h., das HASKELL-Prelude enthält die folgende Instanzendeklaration:

```
instance Monad IO where ...
```

Die Funktionen `return` und `>>=` sind Methoden der Klasse `Monad`, die in den jeweiligen Instanzen dieser Klasse implementiert werden müssen. Die `do`-Notation ist für alle Instanzen der Klasse `Monad` definiert (d.h. auch für benutzerdeklarierte Monaden).

Man bezeichnet einen Typkonstruktor `m` nur dann als Monade, wenn die folgenden Gesetze für `return` und `>>=` erfüllt sind (dies kann natürlich von HASKELL nicht überprüft werden, wenn Instanzen der Klasse `Monad` deklariert werden):

1. `p >>= return   =   p`
   Dieses Gesetz besagt, dass `return` das rechtsneutrale Element zu `>>=` ist. Führt man also erst eine Aktion `p` (vom Typ `m a`) aus und anschließend `return` (vom Typ `a -> m a`), so entsteht insgesamt die Aktion `p`.

2. `return x >>= f   =   f x`
   Dieses Gesetz besagt, dass `return` auch linksneutral zu `>>=` ist. Führt man erst die leere Aktion `return x` mit einem beliebigen Eingabeargument `x` durch (wobei `x :: a` und `return x :: m a`) und anschließend eine Aktion, die das gekapselte Element `x` verwendet (d.h. `f :: a -> m b`), so ist dies dasselbe, als wenn man gleich die Aktion `f x` durchführt. Zusammen mit dem vorigen Gesetz stellt dieses Gesetz sicher, dass `return` wirklich die "leere Aktion" implementiert.

3. `(p >>= \x -> q) >>= \y -> r   =   p >>= \x -> (q >>= \y -> r)`,
   falls `x` nicht in `r` auftritt.
   Dieses Gesetz bezeichnet die Assoziativität von `>>=`. Hierbei sind `p`, `q` und `r` Aktionen mit `p :: m a`, `q :: m b`, `r :: m c`. Dann sagt das Gesetz aus, dass es unerheblich ist, ob man die Aktionen `p` und `q` zusammenfasst oder die Aktionen `q` und `r`. Hierbei ist `p >>= \x -> q` die Zusammenfassung von `p` und `q`, denn es wird erst die Aktion `p` durchgeführt und dann der darin gekapselte Wert `x` (vom Typ `a`) genommen und damit `q` durchgeführt. Auf der linken Seite der Gleichung wird schließlich der insgesamt durch `p` und `q` bestimmte Wert `y` verwendet, um die Aktion `r` durchzuführen. Hingegen ist auf der rechten Seite der Gleichung `(q >>= \y -> r)` die Zusammenfassung der Aktionen `q` und `r`. Hier wird zuerst `p` ausgeführt und der dabei bestimmte Wert `x` dann in der aus `q` und `r` entstandenen Aktion verwendet. Das Gesetz lässt sich (lesbarer) auch in `do`-Notation formulieren:

```
do y <- do x <- p                          do x <- p
           q                  =               y <- q
   r                                          r
```

Modularität und Wiederverwendbarkeit von Programmcode sind zwei wichtige Ziele bei der Programmentwicklung. Hierfür sind Monaden sehr hilfreich, denn das Programmieren mit Monaden ermöglicht es, Programme leicht zu ändern und zu erweitern. Wir wollen dies im Folgenden an einem Beispiel illustrieren. Unser Ziel sei, Ausdrücke der Datenstruktur `Term` auszuwerten, die wie folgt definiert ist:

```
data Term = Con Float | Div Term Term
```

Hierbei steht ein Ausdruck wie `Con 6` für die Konstante mit dem Wert 6 und `Div (Con 6) (Con 3)` für einen Ausdruck mit dem Wert $\frac{6}{3} = 2$. Die folgenden drei Ausdrücke sind Beispiele für Objekte der Datenstruktur `Term`.

```
t1 = Div (Con 6) (Con 3)
t2 = Div (Con 1) (Con 0)
t3 = Div (Div (Con 12) (Con 2)) (Con 3)
```

Wir betrachten zunächst eine Realisierung eines Auswertungsprogramms ohne Verwendung von Monaden. Dabei werden wir das Auswertungsprogramm schrittweise verändern und verbessern. Dies ist typisch für die Entwicklung von Programmen. Man erkennt, dass solche Programmänderungen leider häufig dazu führen, dass die Struktur des Programms komplett geändert wird und große Teile des Programms reimplementiert werden müssen. Wir werden anschließend demonstrieren, dass dies bei der Verwendung von Monaden vermeidbar gewesen wäre.

Einfache Auswertung ohne Monaden

Wir wollen Ausdrücke vom Typ `Term` zu Objekten einer Datenstruktur auswerten, die auf dem Bildschirm ausgegeben werden kann. Hierfür verwenden wir die Datenstruktur `Value Float`, wobei `Value` wie folgt definiert wird:

```
data Value a = Result a

instance Show a => Show (Value a) where
         show (Result x) = "Result: " ++ show x
```

Wertet der Interpreter ein Objekt wie `Result 6` aus, so wird also

```
Result: 6
```

auf dem Bildschirm ausgegeben.

Die Auswertungsfunktion `eval1` überführt Ausdrücke des Typs `Term` in Ausdrücke des Typs `Value Float`, indem bei Objekten mit dem Konstruktor `Div` tatsächlich eine Division der zugehörigen Zahlen durchgeführt wird.

```
eval1 :: Term -> Value Float
eval1 (Con x)   = Result x
eval1 (Div t u) = Result (x/y)
                  where Result x = eval1 t
                        Result y = eval1 u
```

Die Auswertung von `eval1 t1` führt nun zur Ausgabe von

```
Result: 2.0
```

Wertet man allerdings `eval1 t2` (d.h. `eval1 (Div (Con 1) (Con 0))`) aus, so bricht das Programm mit einem Fehler ab, da versucht wird, durch `0` zu dividieren.

Fehlerbehandlung ohne Monaden

Eine Weiterentwicklung unseres Auswertungsprogramms besteht darin, solche Laufzeitfehler abzufangen. Anstelle eines Objekts der Datenstruktur `Value Float` sollte das Ergebnis der Auswertung nun ein Objekt der folgenden (vordefinierten) Datenstruktur `Maybe Float` liefern.

```
data Maybe a = Nothing | Just a

instance Show a => Show (Maybe a) where
         show Nothing   = "Nothing"
         show (Just x)  = "Just " ++ show x
```

Die Datenstruktur `Maybe a` unterscheidet sich von `Value a` dadurch, dass es neben dem Konstruktor `Result` (der hier "`Just`" heißt) noch einen weiteren nullstelligen Datenkonstruktor `Nothing` gibt, der für Fehlerwerte steht. Hiermit ergibt sich also die folgende geänderte Auswertungsfunktion `eval2`.

```
eval2 :: Term -> Maybe Float
eval2 (Con x)   = Just x
eval2 (Div t u) = case eval2 t of
                       Nothing -> Nothing
                       Just x  -> case eval2 u of
                                     Nothing -> Nothing
                                     Just y  -> if y == 0 then Nothing
                                                          else Just (x/y)
```

Die Auswertung von `eval2 t1` führt nun zu `Just 2.0` und die Auswertung von `eval2 t2` ergibt `Nothing`.

Allerdings musste hierfür der Code des Auswerters stark verändert werden, d.h., der Code von `eval2` unterscheidet sich deutlich vom Code des ursprünglichen Auswerters `eval1`. Dies wird noch deutlicher, wenn man eine Datenstruktur verwenden würde, die neben `Div` noch weitere Konstruktoren hat (z.B. `Mult`, `Plus`, `Minus`, etc.). Man müsste in allen entstehenden definierenden Gleichungen von `eval2` analoge umfangreiche (aber jeweils sehr ähnliche) Änderungen durchführen. Dies ist daher kein sehr befriedigender Ansatz für die Software-Entwicklung und wir werden später deutlich machen, wie sich dieser Aufwand durch die Verwendung von Monaden vermeiden lässt.

Zählen der Auswertungsschritte ohne Monaden

Nach der Erstellung von `eval2` wäre eine weitere Verbesserung, die Anzahl der durchgeführten Divisionen bei jeder Auswertung zu zählen und mit auszugeben. Hierzu würde man Objekte vom Typ `Term` nun zu Objekten des Typs `ST Float` auswerten.

```
data ST a = MakeST (Int -> (a, Int))
```

Hierbei steht `ST` für "State Transformer". Die Zustände (States) sind durch ganze Zahlen realisiert und ein State Transformer überführt einen Zustand vom Typ `Int` in einen neuen Zustand und berechnet dabei auch noch ein Resultat vom Typ `a`. Um einen State Transformer vom Typ `ST a` auf einen Zustand (vom Typ `Int`) anzuwenden, verwenden wir die Funktion `apply`.

```
apply :: ST a -> Int -> (a, Int)
apply (MakeST f) s = f s
```

Die folgende `show`-Funktion dient dazu, Objekte vom Typ `ST a` auszugeben.

```
instance Show a => Show (ST a) where
         show tr = "Result: " ++ show x ++ ", State: " ++ show s
                   where (x, s) = apply tr 0
```

Die Ausgabe zu einem State Transformer `tr` ist also der Wert `x` und der Zustand `s`, die sich ergeben, wenn man den State Transformer `tr` auf den initialen Zustand `0` anwendet.

Die Idee bei der Verwendung dieser Datenstruktur für unsere Zwecke ist, dass die Anzahl der Divisionen dem Zustand entspricht. Bei jeder Division muss der Zustand also um `1` erhöht werden. Der zusätzlich berechnete Wert `x` ist die Zahl, die sich bei der Auswertung eines Terms ergibt. So erhält man die folgende verbesserte Auswertungsfunktion.

```
eval3 :: Term -> ST Float
eval3 (Con x)   = MakeST (\s -> (x, s))
eval3 (Div t u) = MakeST (\s -> let (x, s') = apply (eval3 t) s
                                    (y, s'') = apply (eval3 u) s'
                                in  (x/y, s''+1))
```

Bei der Auswertung von `Con x` wird der bisherige Zustand `s` nicht verändert (denn es findet keine weitere Division statt) und das Ergebnis ist `x`. Bei der Auswertung von `Div t u` wird der bisherige Zustand zunächst zu dem Zustand `s'` verändert, der sich ergibt, indem man `s` um die Anzahl von Divisionen erhöht, die zur Auswertung von `t` nötig sind. Anschließend erhöht man `s'` zu `s''`, indem man auch die Zahl der für `u` benötigten Divisionen berücksichtigt. Die Zahl der insgesamt verwendeten Divisionen ist dann `s''+1`. Analog zur bisherigen Auswertung ergibt sich die zu `Div t u` gehörende Zahl als `x/y`, wobei `x` die Zahl ist, die `t` entspricht, und `y` die zu `u` gehörende Zahl ist.

Die Auswertung von `eval3 t1` ergibt also

```
Result: 2.0, State: 1
```

und die Auswertung von `eval3 t3` ergibt

```
Result: 2.0, State: 2
```

Wiederum bedeutete die Änderung jedoch, dass wir den bislang entwickelten Programmcode praktisch komplett umschreiben mussten. Man sieht, dass diese Form der Programmentwicklung nicht sehr änderungsfreundlich ist und kaum Wiederverwendbarkeit ermöglicht.

In den Objekten der Typen `Value a`, `Maybe a` und `ST a` ist jeweils ein Objekt des Typs `a` gekapselt. Hierzu wurde in `Value` ein Konstruktor `Result` verwendet, der der Identität entspricht. In `Maybe` gab es zwei Konstruktoren `Nothing` und `Just` zur Behandlung von Fehlern oder Ausnahmen und in `ST` haben wir einen Konstruktor `MakeST` verwendet, der zur Behandlung und Veränderung von Zuständen dient. Man erkennt, dass `Value`, `Maybe` und `ST` daher eigentlich *Monaden* sind! Daher sollte man sie auch als Instanzen der Klasse `Monad` deklarieren und geeignete Funktionen `return` und `>>=` implementieren. Auf diese Weise lassen sich die drei Interpreter `eval1`, `eval2` und `eval3` auf nahezu gleiche Weise realisieren — der Unterschied liegt jeweils nur in den Implementierungen von `return` und `>>=` in den drei verschiedenen Monaden.

Dadurch ist eine leichte Änderbarkeit und Wiederverwendbarkeit des Programms gewährleistet. Das Programm behandelt dann `Value a`, `Maybe a` und `ST a` als abstrakte Datentypen und greift (im wesentlichen) nur über die Methoden `return` und `>>=` auf ihre Objekte zu. Die Realisierung dieser Methoden ist für den restlichen Teil des Programms unerheblich. Eine Änderung des Programms bedeutet daher (fast) nur eine Änderung der Implementierung von `return` und `>>=`, wohingegen der große Rest des Programms unverändert bleiben kann. Wir demonstrieren dies nun durch eine Realisierung der verschiedenen Auswertungsfunktionen mit Monaden.

Einfache Auswertung mit Monaden

`Value` bezeichnet man auch als die *Identitätsmonade*, da sie ein Datenobjekt `x` einfach nur in dem Konstruktor `Result` als `Result x` kapselt. Die Funktion `return`, die ein Objekt des

Typs `a` als Eingabe bekommt und das entsprechend gekapselte Objekt des Typs `Value a` liefert, ist daher über die Gleichung `return = Result` definiert. Der Operator `>>=` nimmt ein Objekt wie `Result x`, extrahiert `x` aus seiner Kapselung und wendet anschließend eine Funktion `q` vom Typ `a -> Value b` darauf an. Man erhält also

```
instance Monad Value where
         return         = Result
         Result x >>= q = q x
```

Um nachzuweisen, dass `Value` bei dieser Implementierung von `return` und `>>=` tatsächlich eine Monade ist, müsste man natürlich noch die Monadengesetze für `return` und `>>=` überprüfen.

Man erhält jetzt die folgende einfache Implementierung von `eval1`.

```
eval1 :: Term -> Value Float
eval1 (Con x)   = return x
eval1 (Div t u) = do x <- eval1 t
                     y <- eval1 u
                     return (x/y)
```

Bei der Auswertung des Terms `Con x` muss einfach nur der Wert `x` zurückgeliefert werden (in der entsprechenden Kapselung, d.h., man verwendet `return x`). Bei `Div t u` wird erst `t` ausgewertet und `x` auf den darin gekapselten Wert gesetzt. Analog wird dann `u` zum darin gekapselten Wert `y` ausgewertet. Anschließend wird `x/y` (in der entsprechenden Kapselung) zurückgeliefert.

Fehlerbehandlung mit Monaden

Die *Fehlermonade* `Maybe` hat die folgenden Zugriffsfunktionen `return` und `>>=`. Ein Wert `x` vom Typ `a` wird in dem Objekt `Just x` des Typs `Maybe a` gekapselt, d.h., man erhält die Definition `return = Just`. Das Ergebnis der Hintereinanderausführung eines Objekts vom Typ `Maybe a` und einer Funktion `q :: a -> Maybe b` hängt davon ab, ob das erste Objekt das Fehlerobjekt `Nothing` ist. In diesem Fall "schlägt `Nothing` durch", d.h., aus einem Fehler kann wieder nur ein Fehler werden. Ansonsten ist `>>=` wie bei der Identitätsmonade definiert.

```
instance Monad Maybe where
         return        = Just
         Nothing >>= q = Nothing
         Just x  >>= q = q x
```

Damit ergibt sich die folgende Implementierung von `eval2`, d.h. der Auswertungsfunktion, die Divisionen durch `0` abfängt.

```
eval2 :: Term -> Maybe Float
eval2 (Con x)   = return x
eval2 (Div t u) = do x <- eval2 t
                     y <- eval2 u
                     if y /= 0 then return (x/y) else Nothing
```

Man erkennt, dass sich nun `eval1` und `eval2` kaum voneinander unterscheiden! Der einzige Unterschied ist, dass bei Divisionstermen zunächst überprüft werden muss, ob `y` die Zahl `0` ist. Das "Durchschlagen" von `Nothing` ist hingegen bereits durch die Implementierung von `>>=` realisiert (d.h., dadurch ist garantiert, dass das Ergebnis der Auswertung `Nothing` ist, falls ein Teilterm bereits zu `Nothing` auswertet). Analog würden sich auch bei einer Datenstruktur, die neben `Div` weitere Konstruktoren `Mult`, `Plus`, `Minus`, etc. besitzt, keine weiteren Unterschiede zwischen `eval1` und `eval2` ergeben.

Zählen der Auswertungsschritte mit Monaden

Die *Zustandsmonade* `ST` kapselt Werte `x` des Typs `a` in dem State Transformer `MakeST \s -> (x,s)`, der den Zustand `s` unverändert lässt, aber den Wert `x` bestimmt. Die Hintereinanderausführung eines State Transformers `tr` und einer Funktion `q :: a -> ST b` ist ein State Transformer, der einen Zustand `s` nimmt und darauf zunächst `tr` anwendet. Hierbei ergibt sich `(x, s′)`, d.h., man erhält den Zustand `s′` und der Wert `x` ist in diesem Objekt gekapselt. Nun wird `x` hieraus extrahiert und die Funktion `q` darauf angewandt. Der State Transformer `q x` wird dann auf den bislang entstandenen Zustand `s′` angewendet.

```
instance Monad ST where
         return x = MakeST (\s -> (x, s))
         tr >>= q = MakeST (\s -> let (x, s′)  = apply tr s
                                  in  apply (q x) s′)
```

Um die Auswertungsfunktion `eval3` mit Hilfe dieser Monade zu implementieren, benötigt man noch einen State Transformer, der einfach nur den Zustand um `1` erhöht. (Dieser State Transformer wird verwendet, um die Anzahl der gezählten Divisionen um `1` zu vergrößern.)

```
increase :: ST ()
increase = MakeST (\s -> ((), s+1))
```

Nun erhält man folgende Implementierung der Auswertungsfunktion `eval3`.

```
eval3 :: Term -> ST Float
eval3 (Con x)   = return x
eval3 (Div t u) = do x <- eval3 t
                     y <- eval3 u
                     increase
                     return (x/y)
```

Bei der Auswertung von `Div t u` werden also erst `t` und `u` ausgewertet und die dabei entstehenden gekapselten Werte werden in den Variablen `x` und `y` gespeichert. Anschließend wird der (bereits um die bei der Auswertung von `t` und `u` anfallenden Divisionen erhöhte) Zustand noch um `1` erhöht. Schließlich wird der Wert von `x/y` in dem Rückgabeobjekt gekapselt.

Man erkennt, dass sich die drei verschiedenen Auswertungsfunktionen nun nur ganz geringfügig unterscheiden. Die Änderungen betreffen fast nur die Definition der jeweiligen Monade. Dies bedeutet: Programmieren mit Monaden führt tatsächlich zu sehr modularen, änderungsfreundlichen und auch lesbaren Programmen (denn nun ist der Code von `eval1`, `eval2` und `eval3` in der Tat sehr leicht nachvollziehbar).

Kombination von Zustands- und Fehlermonade

Schließlich sei auch noch illustriert, wie man zwei verschiedene Monaden miteinander kombinieren kann. Das Ziel sei nun, den obigen Auswerter so zu verbessern, dass er sowohl die Anzahl der Divisionen zählt als auch den Fehler bei der Division durch `0` abfängt. Eine Lösung besteht darin, eine neue Monade `STE` (für "State Transformer with Exception") zu definieren, die durch die Kombination der Monaden `ST` und `Maybe` entsteht.

```
data STE a = MakeSTE (ST (Maybe a))
```

Die Objekte dieser Datenstruktur sind mit Hilfe eines Konstruktors `MakeSTE` gekapselt. Darin befindet sich ein State Transformer, der Zustände zu Paaren `(x, s)` überführt, wobei `s` ein neuer Zustand ist und `x` entweder der Fehlerwert `Nothing` oder ein mit `Just` gekapselter Wert vom Typ `a` ist. Um Objekte vom Typ `STE a` auszugeben, definieren wir die folgende `show`-Funktion.

```
instance Show a => Show (STE a) where
         show (MakeSTE (MakeST f)) =
              case f 0 of
                   (Just x, s) -> "Result: "  ++ show x ++
                                  ", State: " ++ show s
                   _           -> "Error"
```

Wir benötigen auch eine Funktion `recover`, die den in `STE a` gekapselten State Transformer extrahiert.

```
recover :: STE a -> ST (Maybe a)
recover (MakeSTE f) = f
```

Nun kann man die Monadenmethoden `return` und `>>=` implementieren. Um ein Objekt `x` vom Typ `a` zu kapseln, erzeugt man das Objekt `MakeSTE (return (return x))`. Hierbei hat das innere `return` den Typ `a -> Maybe a` und das äußere `return` den Typ `Maybe a -> ST (Maybe a)`. Dank der Ad-hoc-Polymorphie von HASKELL wird dabei automatisch jeweils die richtige Implementierung von `return` gewählt.

Die Definition von `MakeSTE f >>= q` wird wie folgt definiert. Hierbei ist `f` vom Typ `ST (Maybe a)` und `q` vom Typ `a -> STE b`. Wir definieren zunächst eine Funktion `r` vom Typ `(Maybe a) -> ST (Maybe b)`. Bei einer Eingabe des Fehlerwerts `Nothing` liefert `r` den in `ST` gekapselten Fehler `Nothing`. Bei einer Eingabe von `Just x` extrahiert `r` den gekapselten Wert `x`, wendet `q` darauf an und reduziert den entstehenden Wert `q x` des Typs `STE b` zu dem darin gekapselten Wert des Typs `ST (Maybe b)`. Damit arbeitet `r` also ähnlich wie `q`, nur mit dem Unterschied, dass `r` eine Eingabe des Typs `(Maybe a)` statt `a` erwartet und dass das Resultat vom Typ `ST (Maybe b)` statt vom Typ `STE b` ist. Außerdem wird in `r` das "Durchschlagen" von `Nothing` implementiert. Das Ergebnis von `MakeSTE f >>= q` erhält man demnach dadurch, dass man zuerst `f >>= r` berechnet (hier hat `>>=` den Typ `ST (Maybe a) -> ((Maybe a) -> ST (Maybe b)) -> ST (Maybe b)`, d.h., man verwendet die Implementierung von `>>=` aus der Instanzendeklaration von `ST`). Um aus dem Wert `f >>= r` des Typs `ST (Maybe b)` den benötigten Wert des Typs `STE b` zu erhalten, muss man nun noch den Konstruktor `MakeSTE` anwenden.

```
instance Monad STE where
         return x         = MakeSTE (return (return x))
         MakeSTE f >>= q  = MakeSTE (f >>= r)
                              where r Nothing  = return Nothing
                                    r (Just x) = recover (q x)
```

Schließlich muss man die Konstante `increase` nun auch auf `STE ()` definieren und man benötigt eine weitere Konstante für Fehler.

```
increase′ :: STE ()
increase′ = MakeSTE (MakeST (\s -> (Just (), s+1)))

mistake :: STE a
mistake = MakeSTE (MakeST (\s -> (Nothing, s)))
```

So ergibt sich schließlich die folgende Auswertungsfunktion.

```
eval4 :: Term -> STE Float
eval4 (Con x)   = return x
eval4 (Div t u) = do x <- eval4 t
                     y <- eval4 u
                     increase′
                     if y/=0 then return (x/y) else mistake
```

Beispielsweise ist die Ausgabe bei der Auswertung von `eval4 t3` nun

`Result: 2.0, State: 2`

und die Auswertung von `eval4 t2` ergibt die Ausgabe

`Error`.

Man erkennt, dass auch diese Auswertungsfunktion den vorhergehenden Funktionen `eval1` – `eval3` sehr ähnlich ist. Insbesondere wenn `Term` noch weitere Konstruktoren als nur `Div` hat, zeigt sich die große Änderungsfreundlichkeit dieser Realisierung mit Monaden.

Zum Abschluss fassen wir noch einmal die wichtigen Eigenschaften von Monaden zusammen:

- Eine Monade `m` kapselt Werte (d.h., in einem Objekt vom Typ `m a` ist ein Objekt vom Typ `a` gekapselt). Auf diese Weise kann man zwei verschiedene Berechnungen trennen, indem ein Teil der Berechnung innerhalb der Monade und ein Teil außerhalb stattfindet. (Im Spezialfall der `IO`-Monade können innerhalb der Monade Aktionen mit Seiteneffekten stattfinden, aber dies kann niemals außerhalb der Monade geschehen.)

- Jede Monade besitzt die Zugriffsfunktionen `return` und `>>=`. Hierbei dient `return` zum Kapseln von Objekten in Monadenobjekten (d.h., `return x` ist das Monadenobjekt, in dem `x` gekapselt ist) und `>>=` dient zum sequentiellen Hintereinanderausführen von Monadenausdrücken (diese Sequentialität ist insbesondere beim Spezialfall der `IO`-Monade wichtig, da Seiteneffekte in einer bestimmten Reihenfolge erfolgen sollen). Hierfür existiert die (lesbarere) `do`-Notation.

- Die Kapselung von Werten hat zum einen den Vorteil, dass man im Spezialfall der `IO`-Monade dadurch Seiteneffekte realisieren kann, ohne die referentielle Transparenz zu verlieren. Zum anderen hat sie allgemein den Vorteil einer besseren Strukturierung von Programmen. Dies bedeutet, dass man durch Verwendung von Monaden modularere Programme erhält, die sich leichter ändern und wiederverwenden lassen.

Kapitel 2

Semantik funktionaler Programme

In diesem Kapitel werden wir präzise festlegen, welchen Wert die Ausdrücke einer funktionalen Sprache haben. Diese Präzision ist nötig, um die Bedeutung der Programmiersprache und ihrer Sprachkonstrukte (d.h. ihre *Semantik*) zu definieren. Nur aufgrund dieser Festlegung kann man davon sprechen, dass ein Programm "korrekt" ist, denn nur dadurch ist überhaupt definiert, was ein bestimmtes Programm berechnet. Außerdem wird die Definition der Semantik für die Implementierung einer Programmiersprache benötigt, denn ansonsten ist nicht festgelegt, wie ein Interpreter oder ein Compiler für diese Sprache arbeiten sollen.

Im Folgenden betrachten wir der Einfachheit halber eine etwas eingeschränkte Teilmenge von HASKELL und geben ihre *denotationelle* Semantik an. Bei diesem Ansatz ordnet man jedem Ausdruck der Programmiersprache einen mathematischen Wert (z.B. eine Funktion) zu und beschreibt auf diese Weise ihre Bedeutung. Eine alternative Möglichkeit zur Definition der Semantik einer Programmiersprache ist die *operationelle* Semantik, bei der man ein abstraktes Modell davon verwendet, wie Ausdrücke des Programms *ausgewertet* werden. Die Bedeutung eines Programms wird dann durch das Ergebnis definiert, das der abstrakte Interpreter berechnen würde. Eine solche Art der Semantik-Definition werden wir in Kapitel 3 betrachten.

Wir führen zunächst in Abschnitt 2.1 die mathematischen Grundlagen ein, die für die Angabe der Semantik benötigt werden. Die Definition der Semantik folgt dann in Abschnitt 2.2.

2.1 Vollständige Ordnungen und Fixpunkte

In Abschnitt 2.1.1 diskutieren wir, auf welche Arten von Werten die Ausdrücke der Programmiersprache bei der Festlegung der Semantik abgebildet werden. Hierbei ist insbesondere die Behandlung von undefinierten oder partiell definierten Werten von Bedeutung. In Abschnitt 2.1.2 gehen wir auf die Werte ein, die wir für Ausdrücke von Funktionentypen wählen. Es stellt sich heraus, dass wir Ausdrücke solcher Typen nicht als beliebige Funktionen deuten, sondern nur als Funktionen, die bestimmte Eigenschaften (*Monotonie* und *Stetigkeit*) erfüllen. Schließlich zeigen wir in Abschnitt 2.1.3, wie der Wert von Funktionssymbolen (d.h. von Variablen mit Funktionentyp) bestimmt wird, die durch rekursive

Deklarationen definiert sind. Mit Hilfe dieser mathematischen Grundlagen lässt sich dann in Abschnitt 2.2 die denotationelle Semantik von HASKELL festlegen.

2.1.1 Partiell definierte Werte

Die Semantik der Programmiersprache wird durch eine Funktion $\mathcal{V}al$ festgelegt, die jeden Ausdruck der Programmiersprache auf ein mathematisches Objekt abbildet (Abb. 2.1). Beispielsweise wird der HASKELL-Ausdruck `5` auf die Zahl 5 abgebildet. Analog dazu wird bei einem Programm

```
square :: Int -> Int
square x = x * x
```

der Ausdruck `square` (d.h. das Funktionssymbol `square`) auf eine Funktion abgebildet (die Zahlen quadriert).

In einer Programmiersprache wie HASKELL kann man allerdings auch leicht Funktionssymbole oder andere Objekte einführen, deren Wert nicht oder nur partiell definiert ist.

```
non_term :: Int -> Int
non_term x = non_term (x + 1)
```

Diese Definition gibt keinerlei Information über den Wert des Ausdrucks `non_term 0`, außer dass er identisch ist zum Wert des Ausdrucks `non_term 1` (der wiederum unspezifiziert ist). Ein naheliegender Ansatz ist daher, Ausdrücke der Art `non_term x` für alle Instanzen von `x` als undefiniert zu betrachten. Aus diesem Grund ist es hilfreich, einen eigenen "undefinierten" Wert $\bot$ ("bottom") einzuführen, der nicht-terminierenden Ausdrücken wie `non_term 0` als Bedeutung zugeordnet wird. Wir gehen bei der Angabe der Semantik davon aus, dass alle Ausdrücke unseres Programms korrekt getypt sind (ein Verfahren zur Überprüfung der Typkorrektheit werden wir in Kapitel 4 betrachten). Dann genügt es für die Semantik, zu definieren, auf welche Werte solche typkorrekten Ausdrücke abgebildet werden.

Die Menge der mathematischen Objekte, die den Ausdrücken der Programmiersprache zugeordnet werden können, umfassen ganze Zahlen, boolesche Werte, undefinierte Werte wie $\bot$, Tupel von Werten, (totale) Funktionen auf den Werten, etc. Wir betrachten zunächst nur den einfachen Fall, dass wir keinerlei polymorphe Typen verwenden (solche Typen bezeichnet man als *monomorph*). Dann hat jeder Ausdruck unserer Programmiersprache einen eindeutigen Typ. Es ist dann sinnvoll, für jeden Typ eine eigene Menge an mathematischen Werten festzulegen, so dass jedem Ausdruck dieses Typs ein Wert aus dieser Menge zugeordnet wird. Diese Mengen an mathematischen Werten bezeichnet man als *Domains*, vgl. Abb. 2.1.

Für Integer-Ausdrücke (d.h. Ausdrücke, die dem Nichtterminal integer entsprechen) wählen wir den *Domain* $\mathbb{Z}_\bot = \{\bot_{\mathbb{Z}_\bot}, 0, 1, -1, 2, -2, \ldots\}$, für boolesche Ausdrücke wählen wir den Domain $\mathbb{B}_\bot = \{\bot_{\mathbb{B}_\bot}, \texttt{True}, \texttt{False}\}$, etc.

Die Struktur eines Domains D wird durch eine partielle Ordnung $\sqsubseteq_D$ festgelegt, die ihre Elemente danach ordnet, wie "stark sie definiert sind". Hierbei bedeutet also $x \sqsubseteq_D y$, dass x höchstens so sehr definiert ist wie y (bzw., dass x weniger definiert oder gleich y ist). Da $\bot_D$ den (völlig) undefinierten Wert repräsentiert, gilt offensichtlich $\bot_D \sqsubseteq_D y$ für alle Werte $y \in D$. Im allgemeinen gibt es jedoch nicht nur "normale", vollkommen definierte

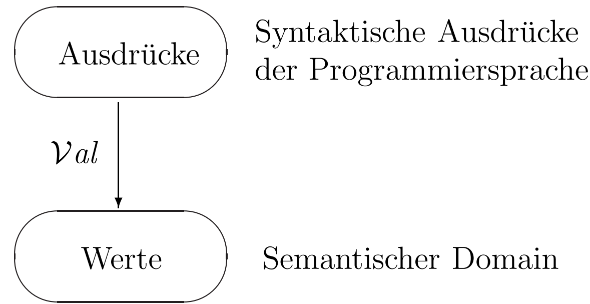


Abbildung 2.1: Denotationelle Semantik

Werte und den vollkommen undefinierten Wert $\perp_D$. Beispielsweise können Listen, Tupel und andere strukturierte Werte undefinierte Komponenten haben. Ein Wert kann also nicht nur definiert oder undefiniert sein, sondern man kann tatsächlich davon sprechen, dass ein Wert mehr definiert ist als ein anderer.

Falls D aus dem Kontext ersichtlich ist, so schreiben wir auch $\sqsubseteq$ statt $\sqsubseteq_D$ und $\perp$ statt $\perp_D$. Offensichtlich gilt für alle $x, y, z \in D$:

(1) $x \sqsubseteq x$ (Reflexivität)

(2) Aus $x \sqsubseteq y$ und $y \sqsubseteq z$ folgt $x \sqsubseteq z$. (Transitivität)

(3) Aus $x \sqsubseteq y$ und $y \sqsubseteq x$ folgt $x = y$. (Antisymmetrie)

Jede transitive und antisymmetrische Relation bezeichnet man als *Ordnung*.

In Domains für Basistypen wie `Int` gibt es lediglich einen undefinierten Wert $\perp_{\mathbb{Z}_\perp}$ und dieser ist weniger definiert als alle anderen Werte $0, 1, -1, \ldots$ Dies wird in dem Hasse-Diagramm in Abb. 2.2 deutlich.

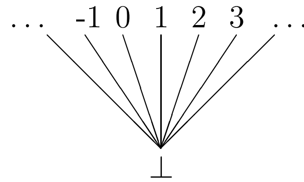


Abbildung 2.2: Partielle Ordnung auf dem flachen Domain $\mathbb{Z}_\perp$

Analog verhält es sich bei anderen Basis-Domains. Hierbei bezeichnet C die Menge der Zeichen ($C = \{\text{'a'}, \text{'b'}, \ldots\}$) und F die Menge der Gleitkommazahlen. $C_\perp$ und $F_\perp$ entstehen aus C und F durch Ergänzung eines Elements $\perp_{C_\perp}$ bzw. $\perp_{F_\perp}$. Es gilt also:

Definition 2.1.1 ($\sqsubseteq$ auf Basis-Domains) *Sei D ein Basis-Domain (d.h. $\mathbb{Z}_\perp$, $\mathbb{B}_\perp$, $C_\perp$ oder $F_\perp$). Dann gilt für alle $d, d' \in D$: $d \sqsubseteq_D d'$ gdw. $d = \perp_D$ oder $d = d'$. Man bezeichnet solche Domains und solche Ordnungen $\sqsubseteq_D$ als* flache *Domains bzw. Ordnungen.*

Man erkennt, dass $\sqsubseteq$ im allgemeinen nicht total ist, da es viele unvergleichbare Werte gibt, z.B. $0 \not\sqsubseteq 1$ und $1 \not\sqsubseteq 0$ für $0, 1 \in \mathbb{Z}_\perp$. Die Relation $\sqsubseteq$ ist demnach eine *partielle Ordnung* (oder *Halbordnung*).

Wie erwähnt, sind Domains wie $\mathbb{Z}_\perp$ und $\mathbb{B}_\perp$ flach, d.h., ihre Elemente sind entweder völlig definiert oder völlig undefiniert. Auf anderen Domains kann die Ordnung $\sqsubseteq$ jedoch wesentlich vielfältiger aussehen. Wenn $\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n$ Ausdrücke sind, so dass $\underline{\mathsf{exp}}_i$ einem Wert im Domain D_i entspricht, so ist auch der Tupel $(\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n)$ ein Ausdruck und sein Wert liegt im Domain $D_1 \times \ldots \times D_n$. Das kartesische Produkt von Domains ergibt also wieder einen Domain. Die Ordnung $\sqsubseteq_{D_1 \times \ldots \times D_n}$ auf diesem *Produkt-Domain* ist wie folgt definiert.

Definition 2.1.2 (Produkt-Domains) *Seien $D_1, \ldots, D_n$ Domains ($n \geq 2$). Dann ist auch $D_1 \times \ldots \times D_n$ ein Domain. Die Relation $\sqsubseteq_{D_1 \times \ldots \times D_n}$ ist wie folgt definiert: $(d_1, \ldots, d_n)$ $\sqsubseteq_D$ $(d'_1, \ldots d'_n)$ gdw. $d_i \sqsubseteq_{D_i} d'_i$ für alle $1 \leq i \leq n$. Das kleinste Element von $D_1 \times \ldots \times D_n$ ist demnach $(\perp_{D_1}, \ldots, \perp_{D_n})$. Es wird auch mit $\perp_{D_1 \times \ldots \times D_n}$ bezeichnet.*

Generell bezeichnet man das (bezüglich $\sqsubseteq_D$) kleinste Element eines Domains D immer mit $\perp_D$. Das Hasse-Diagramm in Abb. 2.3 illustriert (einen Teil der) partiellen Ordnung $\sqsubseteq$ auf $\mathbb{Z}_\perp \times \mathbb{Z}_\perp$.

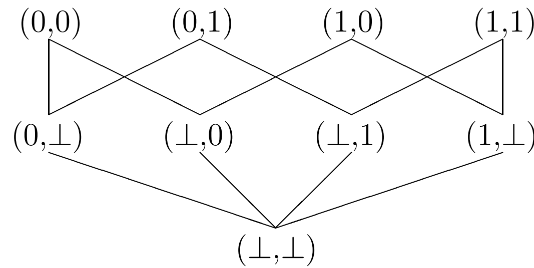


Abbildung 2.3: Partielle Ordnung auf dem Domain $\mathbb{Z}_\perp \times \mathbb{Z}_\perp$

Es ist leicht zu zeigen, dass es sich bei $\sqsubseteq$ auf Produkt-Domains ebenfalls um eine reflexive partielle Ordnung handelt.

Lemma 2.1.3 (Partielle Ordnung auf Produkt-Domains) *Wenn alle $\sqsubseteq_{D_i}$ reflexive Ordnungen sind, so ist auch $\sqsubseteq_{D_1 \times \ldots \times D_n}$ eine reflexive Ordnung.*

Beweis. Es gilt $(d_1, \ldots, d_n) \sqsubseteq_{D_1 \times \ldots \times D_n} (d_1, \ldots, d_n)$, denn aufgrund der Reflexivität der $\sqsubseteq_{D_i}$ gilt auch $d_i \sqsubseteq_{D_i} d_i$ für alle $1 \leq i \leq n$. Demnach ist $\sqsubseteq_{D_1 \times \ldots \times D_n}$ reflexiv.

Sei $(d_1, \ldots, d_n) \sqsubseteq_{D_1 \times \ldots \times D_n} (d'_1, \ldots, d'_n) \sqsubseteq_{D_1 \times \ldots \times D_n} (d''_1, \ldots, d''_n)$. Dann gilt nach Definition $d_i \sqsubseteq_{D_i} d'_i \sqsubseteq_{D_i} d''_i$ für alle $1 \leq i \leq n$. Aus der Transitivität von $\sqsubseteq_{D_i}$ folgt $d_i \sqsubseteq_{D_i} d''_i$ und somit $(d_1, \ldots, d_n) \sqsubseteq_{D_1 \times \ldots \times D_n} (d''_1, \ldots, d''_n)$. Damit ist die Transitivität von $\sqsubseteq_{D_1 \times \ldots \times D_n}$ gezeigt.

Sei nun $(d_1, \ldots, d_n) \sqsubseteq_{D_1 \times \ldots \times D_n} (d'_1, \ldots, d'_n)$ und $(d'_1, \ldots, d'_n) \sqsubseteq_{D_1 \times \ldots \times D_n} (d_1, \ldots, d_n)$. Dies bedeutet $d_i \sqsubseteq_{D_i} d'_i \sqsubseteq_{D_i} d_i$ für alle $1 \leq i \leq n$. Aus der Antisymmetrie der $\sqsubseteq_{D_i}$ folgt damit $d_i = d'_i$ und daher $(d_1, \ldots, d_n) = (d'_1, \ldots, d'_n)$. □

Die Ordnung $\sqsubseteq$ lässt sich nicht nur direkt auf Tupel erweitern, sondern auch auf Funktionen. (Allerdings besteht ein Domain für Funktionen von D_1 nach D_2 nicht aus *allen*, sondern nur aus bestimmten Funktionen von D_1 nach D_2. Dies liegt unter anderem daran, dass manche Funktionen gar nicht berechenbar sind. Die genaue Bildung von Domains für verschiedene Typen wird in Abschnitt 2.2.1 erklärt.)

Definition 2.1.4 ($\sqsubseteq$ auf Funktionen) *Seien f, g zwei Funktionen von einem Domain D_1 auf einen anderen Domain D_2. Es gilt $f \sqsubseteq_{D_1 \to D_2} g$ gdw. $f(d) \sqsubseteq_{D_2} g(d)$ für alle $d \in D_1$. Das kleinste Element des Funktionenraums von D_1 nach D_2 ist daher die Funktion, die alle Elemente aus D_1 auf $\bot_{D_2}$ abbildet. Es wird mit $\bot_{D_1 \to D_2}$ bezeichnet.*

Diese Ordnung bedeutet, dass g mehr definiert ist als f, falls g Resultate berechnet, die mehr definiert sind als diejenigen von f (für alle Argumente aus D_1).

Lemma 2.1.5 (Partielle Ordnung auf Funktionen) *Wenn $\sqsubseteq_{D_2}$ eine reflexive Ordnung ist, so ist auch $\sqsubseteq_{D_1 \to D_2}$ eine reflexive Ordnung.*

Beweis. Es gilt $f \sqsubseteq_{D_1 \to D_2} f$, denn aufgrund der Reflexivität von $\sqsubseteq_{D_2}$ haben wir $f(d) \sqsubseteq_{D_2} f(d)$ für alle $d \in D_1$.

Sei $f \sqsubseteq_{D_1 \to D_2} g \sqsubseteq_{D_1 \to D_2} h$. Damit gilt für alle $d \in D_1$, dass $f(d) \sqsubseteq_{D_2} g(d) \sqsubseteq_{D_2} h(d)$ ist. Aus der Transitivität von $\sqsubseteq_{D_2}$ folgt $f(d) \sqsubseteq_{D_2} h(d)$ und damit $f \sqsubseteq_{D_1 \to D_2} h$.

Sei nun $f \sqsubseteq_{D_1 \to D_2} g \sqsubseteq_{D_1 \to D_2} f$. Dies bedeutet $f(d) \sqsubseteq_{D_2} g(d) \sqsubseteq_{D_2} f(d)$ für alle $d \in D_1$ und somit $f = g$ aufgrund der Antisymmetrie von $\sqsubseteq_{D_2}$. □

2.1.2 Monotone und stetige Funktionen

Die Semantik eines Funktionssymbols eines funktionalen Programms ist eine Funktion zwischen Domains. Hierbei werden als semantische Funktionen (d.h. als Deutungen der HASKELL-Funktionssymbole) nur *totale* Funktionen verwendet. Ein partiell deklariertes HASKELL-Funktionssymbol hat als Semantik also eine totale Funktion, die bei manchen Argumenten das Ergebnis $\bot$ hat. Der Vorteil dieses Vorgehens ist, dass man so auch mit partiell definierten Werten umgehen kann und man auch nicht-strikte Funktionen modellieren kann.

Definition 2.1.6 (Extension und Striktheit von Funktionen) *Sei $f : A \to B$ eine Funktion. Jede Funktion $f' : A_\bot \to B$ bezeichnet man als* Extension *von f, wobei $A_\bot = A \cup \{\bot_{A_\bot}\}$ und wobei $f(d) = f'(d)$ für alle $d \in A$.*

Eine Funktion $g : D_1 \times \ldots \times D_n \to D$ für Domains $D_1, \ldots, D_n, D$ heißt strikt, *falls $g(d_1, \ldots, d_n) = \bot_D$ für alle $d_1, \ldots, d_n$ gilt, bei denen ein $d_i = \bot_{D_i}$ ist. Ansonsten heißt g* nicht-strikt.

Als Beispiel betrachten die folgende Funktionsdeklaration in HASKELL.

```
one :: Int -> Int
one x = 1
```

Dies entspricht einer Funktion f, die Zahlen aus $\mathbb{Z}$ auf die Zahl 1 abbildet. Die Frage ist, worauf der undefinierte Wert $\bot_{\mathbb{Z}_\bot}$ abgebildet werden soll. Offensichtlich gibt es hier mehrere Möglichkeiten:

(a) $f_1'(x) = 1$ für $x \in \mathbb{Z}$, $f_1'(\bot) = 1$

(b) $f_2'(x) = 1$ für $x \in \mathbb{Z}$, $f_2'(\bot) = \bot$

(c) $f'_3(x) = 1$ für $x \in \mathbb{Z}$, $f'_3(\bot) = 0$

In einer Programmiersprache wie HASKELL, die Lazy Evaluation verwendet, sollte die Semantik von `one` die Funktion f'_1 sein, da auch bei Anwendung von `one` auf ein undefiniertes Argument das Resultat 1 berechnet wird. Der Grund ist, dass die Auswertung des Arguments nicht nötig ist, um das Ergebnis zu bestimmen. In einer Programmiersprache mit strikter Semantik (wie ML oder LISP) würde man stattdessen eine Semantik wählen, bei der `one` als die Funktion f'_2 gedeutet wird.

Die Funktion f'_3 hingegen ist *nicht berechenbar*! Es gibt also kein Programm (in irgendeiner Programmiersprache), das diese Funktion realisieren würde. Der Grund ist, dass es nicht entscheidbar ist, ob die Auswertung eines Arguments terminiert (dies ist die Unentscheidbarkeit des Halteproblems). Bei einer Implementierung der Funktion f'_3 müsste man aber genau dies entscheiden können, um in Abhängigkeit davon entweder das Ergebnis 0 oder das Ergebnis 1 zurückliefern zu können. Funktionen $g : D_1 \to D_2$ mit

- $g(\bot) \neq \bot$ und
- $g(d) \neq g(\bot)$ für ein $d \neq \bot$

sind also nicht berechenbar und daher können sie niemals als Semantik einer Funktion in einer Programmiersprache auftreten.

Solche Funktionen sollten daher gar nicht erst in unseren Funktions-Domain aufgenommen werden. Allgemein wollen wir also Funktionen ausschließen, die mit weniger definierten Argumenten mehr bzw. anders definierte Resultate erzeugen. Genauer bedeutet dies, dass wir uns auf *monotone* Funktionen beschränken, da alle nicht-monotonen Funktionen nicht berechenbar sind.

Definition 2.1.7 (Monotone Funktionen) *Seien $\sqsubseteq_{D_1}$ und $\sqsubseteq_{D_2}$ partielle Ordnungen auf D_1 bzw. D_2. Eine Funktion $f : D_1 \to D_2$ ist* monoton *gdw. $f(d) \sqsubseteq_{D_2} f(d')$ für alle $d \sqsubseteq_{D_1} d'$ gilt.*

Im obigen Beispiel sind also die Funktionen f'_1 und f'_2 monoton, aber die Funktion f'_3 ist nicht monoton. Als weiteres Beispiel betrachten wir das Programm

```
id :: Int -> Int
id x = x
```

Die Semantik von `id` kann nur eine strikte Funktion f sein, denn aus $f(0) = 0$ und $f(1) = 1$ folgt wegen $\bot \sqsubseteq 0$ und $\bot \sqsubseteq 1$ und der Monotonie von f, dass $f(\bot) = \bot$ sein muss. Insgesamt ergibt sich:

Lemma 2.1.8 (Aus Striktheit folgt Monotonie) *Seien $D_1, \ldots, D_n, D$ Domains, wobei $D_1, \ldots, D_n$ flach sind. Sei $f : D_1 \times \ldots \times D_n \to D$ strikt. Dann ist f auch monoton.*

Beweis. Sei $(d_1, \ldots, d_n) \sqsubseteq (d'_1, \ldots, d'_n)$. Zu zeigen ist, dass dann auch $f(d_1, \ldots, d_n) \sqsubseteq f(d'_1, \ldots, d'_n)$ gilt. Falls $(d_1, \ldots, d_n) = (d'_1, \ldots, d'_n)$ gilt, so folgt diese Behauptung sofort aus der Reflexivität. Ansonsten folgt aus $(d_1, \ldots, d_n) \sqsubseteq (d'_1, \ldots, d'_n)$, dass es mindestens

ein i gibt mit $d_i = \bot$. Aufgrund der Striktheit von f folgt $f(d_1, \ldots, d_n) = \bot$ und damit $f(d_1, \ldots, d_n) \sqsubseteq f(d'_1, \ldots, d'_n)$. □

Da HASKELL aber eine Sprache mit lazy evaluation ist, benötigen wir auch nicht-strikte Funktionen als Deutungen. Ein Beispiel ist das folgende Programm.

```
cond :: (Bool,Int,Int) -> Int
cond (True,x,y) = x
cond (False,x,y) = y
```

Hierbei ergibt sich als Deutung eine Funktion f, so dass für alle $b \in \mathbb{B}_\bot$ und alle $x, y \in \mathbb{Z}_\bot$ gilt:

$$f(b,x,y) = \begin{cases} x, & \text{falls } b = \texttt{True} \\ y, & \text{falls } b = \texttt{False} \\ \bot_{\mathbb{Z}_\bot} & \text{falls } b = \bot_{\mathbb{B}_\bot} \end{cases}$$

Insbesondere hat man also $f(\texttt{True}, 2, \bot_{\mathbb{Z}_\bot}) = 2$, d.h., f ist nicht strikt. Trotzdem ist f monoton.

Wir haben gesehen, dass es Domains mit verschiedenen Stufen der Undefiniertheit gibt. Beispielsweise haben wir in $\mathbb{Z}_\bot \times \mathbb{Z}_\bot$

$$(\bot, \bot) \sqsubseteq (5, \bot) \sqsubseteq (5, 8).$$

Wie üblich könnte man auch $\bot_{\mathbb{Z}_\bot \times \mathbb{Z}_\bot}$ statt $(\bot_{\mathbb{Z}_\bot}, \bot_{\mathbb{Z}_\bot})$ schreiben. Eine solche Folge von Elementen eines Domains, die immer "mehr definiert werden", bezeichnet man als *Kette* (engl. *chain*).

Definition 2.1.9 (Kette) *Sei $\sqsubseteq$ eine partielle Ordnung auf einer Menge D. Eine nicht-leere Teilmenge $\{d_1, d_2, \ldots\}$ von abzählbar vielen Elementen aus D heißt* Kette *gdw.*

$$d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \ldots$$

Offensichtlich ist $\{\bot, (5, \bot), (5, 8)\}$ eine Kette. Um ein Beispiel einer unendlichen Kette zu sehen, betrachten wir die Funktionen $\mathsf{fact}_i : \mathbb{Z}_\bot \to \mathbb{Z}_\bot$ für $i \in \mathbb{N}$:

$$\begin{aligned}
\mathsf{fact}_0(x) &= \bot \text{ für alle } x \in \mathbb{Z}_\bot \\
\mathsf{fact}_1(x) &= \begin{cases} x!, & \text{für } 0 \le x < 1 \\ 1, & \text{für } x < 0 \\ \bot, & \text{für } x = \bot \text{ oder } 1 \le x \end{cases} \\
\mathsf{fact}_2(x) &= \begin{cases} x!, & \text{für } 0 \le x < 2 \\ 1, & \text{für } x < 0 \\ \bot, & \text{für } x = \bot \text{ oder } 2 \le x \end{cases} \\
&\vdots \\
\mathsf{fact}_n(x) &= \begin{cases} x!, & \text{für } 0 \le x < n \\ 1, & \text{für } x < 0 \\ \bot, & \text{für } x = \bot \text{ oder } n \le x \end{cases}
\end{aligned}$$

Damit gilt

$$\mathsf{fact}_0 \sqsubseteq \mathsf{fact}_1 \sqsubseteq \mathsf{fact}_2 \sqsubseteq \mathsf{fact}_3 \sqsubseteq \ldots$$

und $\{\mathsf{fact}_0, \mathsf{fact}_1, \ldots\}$ ist eine Kette.

Für Ketten ist es oftmals interessant, eine kleinste obere Schranke zu finden (engl. *least upper bound*, *lub*).

Definition 2.1.10 (Kleinste obere Schranke) *Sei $\sqsubseteq$ eine partielle Ordnung auf einer Menge D und S eine Teilmenge von D. Ein Element $d \in D$ ist eine* obere Schranke *von S, falls $d' \sqsubseteq d$ für alle $d' \in S$ gilt. Das Element d ist die* kleinste obere Schranke *(oder "*Supremum*"), falls darüber hinaus für alle anderen oberen Schranken e der Zusammenhang $d \sqsubseteq e$ gilt. Wir bezeichnen dieses Element als $\sqcup S$.*

Die kleinste obere Schranke einer Kette ist also der Wert, der von allen Elementen der Kette approximiert wird und der darüber hinaus auch noch kleiner als jede andere obere Schranke der Kette ist. Die einzige obere Schranke der Kette $\{\bot, (5, \bot), (5, 8)\}$ ist offensichtlich $(5, 8)$; dies ist daher auch die kleinste obere Schranke. Die Kette $\{\mathsf{fact}_0, \mathsf{fact}_1, \ldots\}$ hat unendlich viele obere Schranken f. Hierbei gilt $f(x) = x!$ für alle $x \in \mathbb{Z}$ mit $x \geq 0$, $f(x) = 1$ für alle $x < 0$, aber der Wert von $f(\bot)$ kann beliebig gewählt werden. Die kleinste obere Schranke ist die Funktion f, bei der $f(\bot) = \bot$ gilt. Man erkennt, dass dies auch die einzige der oberen Schranken ist, die monoton ist.

Es ergibt sich der folgende Zusammenhang für kleinste obere Schranken bei kartesischen Produkten und bei Funktionen. Man erkennt hierbei, dass Funktionen eigentlich (potentiell unendlichen) Tupeln entsprechen (mit je einer Komponente für jedes Element des Argument-Domains).

Lemma 2.1.11 (Kleinste obere Schranken bei Produkten und Funktionen) *Seien $D_1, \ldots, D_n, D, D'$ Domains.*

(a) Sei $S \subseteq D_1 \times \ldots \times D_n$. Für alle $1 \leq i \leq n$ sei $S_i = \{d_i |$ es existieren $d_1, \ldots, d_{i-1}, d_{i+1}, \ldots, d_n$ mit $(d_1, \ldots, d_i, \ldots, d_n) \in S\}$. Dann gilt

- *$\sqcup S$ existiert gdw. $\sqcup S_i$ existiert für alle $1 \leq i \leq n$.*
- *Falls $\sqcup S$ existiert, so gilt $\sqcup S = (\sqcup S_1, \ldots, \sqcup S_n)$.*

(b) Sei S eine Menge von Funktionen von D nach D'. Für alle $i \in D$ sei $S_i = \{f(i) | f \in S\}$. Dann gilt

- *$\sqcup S$ existiert gdw. $\sqcup S_i$ existiert für alle $i \in D$.*
- *Falls $\sqcup S$ existiert, so gilt $(\sqcup S)(i) = \sqcup S_i$.*

Beweis.

(a) Wir nehmen zunächst an, dass alle $\sqcup S_i$ existieren. Es ist zu zeigen, dass dann $(\sqcup S_1, \ldots, \sqcup S_n)$ die kleinste obere Schranke von S ist. Sei $(d_1, \ldots, d_n) \in S$. Dann gilt $d_i \sqsubseteq \sqcup S_i$ für alle i und somit $(d_1, \ldots, d_n) \sqsubseteq (\sqcup S_1, \ldots, \sqcup S_n)$, d.h., $(\sqcup S_1, \ldots, \sqcup S_n)$ ist in der Tat eine obere Schranke von S. Sei nun $(e_1, \ldots, e_n)$ eine weitere obere Schranke von S. Für alle i ist der Wert e_i eine obere Schranke von S_i (denn für alle $d_i \in S_i$ existiert ein $(d_1, \ldots, d_i, \ldots, d_n) \in S$, wobei $(d_1, \ldots, d_i, \ldots, d_n) \sqsubseteq (e_1, \ldots, e_i, \ldots, e_n)$ und damit

$d_i \sqsubseteq e_i$ ist). Daher folgt $\sqcup S_i \sqsubseteq e_i$ und damit $(\sqcup S_1, \ldots, \sqcup S_n) \sqsubseteq (e_1, \ldots, e_n)$. Somit existiert $\sqcup S$ tatsächlich und es gilt $\sqcup S = (\sqcup S_1, \ldots, \sqcup S_n)$.

Für die Rückrichtung nehmen wir nun an, dass $\sqcup S$ existiert. Sei $\sqcup S = (u_1, \ldots, u_n)$. Es ist zu zeigen, dass u_i die kleinste obere Schranke von S_i ist (für alle i). Sei $d_i \in S_i$. Dann existiert ein $(d_1, \ldots, d_i, \ldots, d_n) \in S$ und wegen $(d_1, \ldots, d_i, \ldots, d_n) \sqsubseteq (u_1, \ldots, u_i, \ldots, u_n)$ gilt auch $d_i \sqsubseteq u_i$. Daher ist u_i also eine obere Schranke von S_i. Sei nun e_i eine weitere obere Schranke von S_i. Dann ist $(u_1, \ldots, u_{i-1}, e_i, u_{i+1}, \ldots, u_n)$ auch eine weitere obere Schranke von S. Da $(u_1, \ldots, u_i, \ldots, u_n)$ die kleinste obere Schranke ist, gilt aber $u_i \sqsubseteq e_i$. Somit gilt tatsächlich $\sqcup S_i = u_i$.

(b) Wir nehmen zunächst an, dass alle $\sqcup S_i$ existieren. Sei u die Funktion mit $u(i) = \sqcup S_i$. Es ist zu zeigen, dass dann u die kleinste obere Schranke von S ist. Sei $f \in S$. Dann gilt $f(i) \sqsubseteq \sqcup S_i$ für alle i und somit $f \sqsubseteq u$, d.h., u ist in der Tat eine obere Schranke von S. Sei nun g eine weitere obere Schranke von S. Für alle i ist der Wert $g(i)$ eine obere Schranke von S_i. Daher folgt $\sqcup S_i \sqsubseteq g(i)$ und damit $u \sqsubseteq g$. Daher existiert $\sqcup S$ tatsächlich und es gilt $(\sqcup S)(i) = \sqcup S_i$.

Für die Rückrichtung nehmen wir nun an, dass $\sqcup S$ existiert. Sei $\sqcup S = u$. Es ist zu zeigen, dass $u(i)$ die kleinste obere Schranke von S_i ist (für alle i). Sei $d \in S_i$. Dann existiert ein $f \in S$ mit $f(i) = d$ und wegen $f \sqsubseteq u$ gilt auch $d = f(i) \sqsubseteq u(i)$. Daher ist $u(i)$ also eine obere Schranke von S_i. Sei nun e eine weitere obere Schranke von S_i. Dann ist die Funktion u' mit $u'(i) = e$ und $u'(j) = u(j)$ für $j \neq i$ auch eine weitere obere Schranke von S. Da u die kleinste obere Schranke ist, gilt aber $u(i) \sqsubseteq u'(i) = e$. Somit gilt tatsächlich $\sqcup S_i = u(i)$. □

Die essentielle Eigenschaft, die wir für Domains benötigen, ist, dass sie *vollständig* sein müssen. Anders ausgedrückt bedeutet das, dass die Ordnung $\sqsubseteq_D$ eine *vollständige* partielle Ordnung sein muss (engl. *complete partial order, cpo*).

Definition 2.1.12 (Vollständige partielle Ordnung) *Eine reflexive partielle Ordnung $\sqsubseteq$ auf einer Menge D heißt* vollständig *gdw.*

(1) D hat ein kleinstes Element (das mit $\bot_D$ bezeichnet wird).

(2) Für jede Kette S von D existiert die kleinste obere Schranke $\sqcup S \in D$.

Die bislang betrachteten Domains sind in der Tat cpo's.

Satz 2.1.13 (Vollständigkeit der Domains) *Seien $D_1, \ldots, D_n$ Domains.*

(a) Jede reflexive partielle Ordnung, die ein kleinstes Element hat und nur endliche Ketten besitzt, ist eine cpo. Daher sind die flachen Basis-Domains $\mathbb{Z}_\bot$, $\mathbb{B}_\bot$, $C_\bot$ und $F_\bot$ cpo's.

(b) Falls $\sqsubseteq_{D_i}$ vollständig auf D_i ist (für alle $1 \leq i \leq n$), so ist auch $\sqsubseteq_{D_1 \times \ldots \times D_n}$ vollständig auf $D_1 \times \ldots \times D_n$.

(c) Falls $\sqsubseteq_{D_2}$ eine cpo auf D_2 ist, so ist $\sqsubseteq_{D_1 \to D_2}$ eine cpo auf der Menge der Funktionen von D_1 nach D_2.

Beweis.

(a) Sei $S = \{d_1, \ldots, d_n\}$ eine Kette mit $d_1 \sqsubseteq d_2 \sqsubseteq \ldots \sqsubseteq d_n$. Die kleinste obere Schranke ist d_n. Dies ist offensichtlich eine obere Schranke (aufgrund der Transitivität). Für jede andere obere Schranke e müsste $d_n \sqsubseteq e$ gelten, so dass es auch die kleinste obere Schranke ist.

Dass $\sqsubseteq_D$ auf flachen Domains vollständig ist, folgt daraus sofort, da dort jede Kette höchstens die Länge 2 hat.

(b) Das kleinste Element von $D_1 \times \ldots \times D_n$ ist $(\bot_{D_1}, \ldots, \bot_{D_n})$. Sei nun $S \subseteq D_1 \times \ldots \times D_n$ eine Kette. Wir haben also

$$S = \{(d_1^1, \ldots, d_n^1), (d_1^2, \ldots, d_n^2), \ldots\},$$

wobei $(d_1^1, \ldots, d_n^1) \sqsubseteq (d_1^2, \ldots, d_n^2) \sqsubseteq \ldots$ Daraus folgt

$$d_i^1 \sqsubseteq d_i^2 \sqsubseteq d_i^3 \sqsubseteq \ldots,$$

d.h., die Mengen $S_i = \{d_i |$ es existieren $d_1, ..., d_{i-1}, d_{i+1}, ..., d_n$ mit $(d_1, \ldots, d_i, \ldots, d_n) \in S\}$ sind ebenfalls Ketten (für alle $1 \leq i \leq n$). Da die Ordnungen $\sqsubseteq_{D_i}$ vollständig sind, existieren somit die kleinsten oberen Schranken der Mengen S_i. Aus Lemma 2.1.11 (a) folgt dann, dass auch $\sqcup S$ existiert.

(c) Das kleinste Element von $D_1 \to D_2$ ist $\bot_{D_1 \to D_2}$, d.h. die Funktion, die alle Elemente aus D_1 auf $\bot_{D_2}$ abbildet. Sei nun $S \subseteq D_1 \to D_2$ eine Kette. Wir haben also $S = \{f_1, f_2, f_3, \ldots\}$ mit $f_1 \sqsubseteq f_2 \sqsubseteq f_3 \sqsubseteq \ldots$ Daraus folgt

$$f_1(i) \sqsubseteq f_2(i) \sqsubseteq f_3(i) \sqsubseteq \ldots,$$

d.h., die Mengen $S_i = \{f(i) \mid f \in S\}$ sind ebenfalls Ketten (für alle $i \in D_1$). Da die Ordnung $\sqsubseteq_{D_2}$ vollständig ist, existieren somit die kleinsten oberen Schranken der Mengen S_i. Aus Lemma 2.1.11 (b) folgt dann, dass auch $\sqcup S$ existiert. □

In Satz 2.1.13 (b) und (c) gelten auch die Rückrichtungen. Aus der Vollständigkeit von $\sqsubseteq_{D_1 \times \ldots \times D_n}$ folgt also auch die Vollständigkeit aller $\sqsubseteq_{D_i}$. Ebenso folgt aus der Vollständigkeit von $\sqsubseteq_{D_1 \to D_2}$ die Vollständigkeit von $\sqsubseteq_{D_2}$.

Neben der Monotonie fordern wir jetzt noch eine weitere Eigenschaft für die Funktionen, die potentiell als Deutungen von HASKELL-Funktionssymbolen in Frage kommen. Wiederum dient diese Einschränkung dazu, weitere nicht-berechenbare Funktionen auszuschließen.

Definition 2.1.14 (Stetigkeit) *Sei $\sqsubseteq_{D_1}$ vollständig auf D_1 und $\sqsubseteq_{D_2}$ vollständig auf D_2. Eine Funktion $f : D_1 \to D_2$ heißt* stetig *(engl.* continuous*) gdw. für jede Kette S von D_1 jeweils $f(\sqcup S) = \sqcup\{f(d) \mid d \in S\}$ ist. Die Menge aller stetigen Funktionen von D_1 nach D_2 wird mit $\langle D_1 \to D_2 \rangle$ bezeichnet.*[1]

[1] In der Literatur schreibt man hierfür üblicherweise $[D_1 \to D_2]$. Wir wählen jedoch die obige abweichende Schreibweise, um Verwechslungen mit der HASKELL-Notation für Listen zu vermeiden.

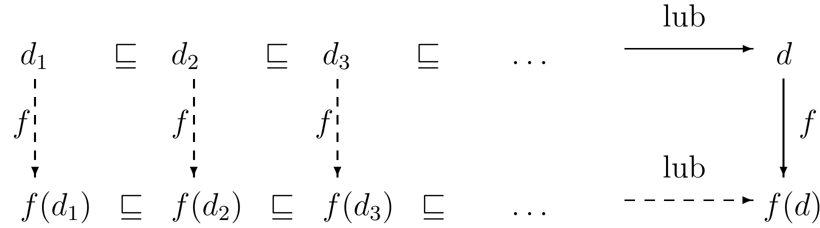


Abbildung 2.4: Stetigkeit von Funktionen f

Anstelle von $\{f(d)|d \in S\}$ schreibt man oft $f(S)$. Dann lautet die Bedingung für Stetigkeit $f(\sqcup S) = \sqcup f(S)$. Intuitiv bedeutet Stetigkeit, dass f den Grenzwert jeder Kette auf das gleiche Ergebnis abbildet, das man erhalten würde, wenn man das Ergebnis für jedes individuelle Element der Kette berechnen würde und dann den Grenzwert dieser Ergebnisse bildet. Dies wird in Abb. 2.4 verdeutlicht. Wenn man zunächst die Kette $d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ betrachtet, ihren Grenzwert d bildet und dann f darauf anwendet (durchgezogene Pfeile), so muss sich dasselbe ergeben, als wenn man erst f auf die einzelnen Elemente der Kette anwendet und dann den Grenzwert bildet (gestrichelte Pfeile).

Betrachten wir zunächst einige Beispiele, um das Konzept der Stetigkeit zu erläutern. Beispielsweise sind alle konstanten Funktionen $f : D_1 \to D_2$ mit $f(x) = e$ für alle $x \in D_1$ stetig. Der Grund ist, dass für jede Kette S aus D_1 immer $f(S) = \{e\}$ gilt und somit erhält man $\sqcup f(S) = e = f(\sqcup S)$.

Ebenso ist z.B. auch die Identitätsfunktion $i : D \to D$ stetig, wobei $i(x) = x$ für alle $x \in D$ gilt. Der Grund ist, dass für alle Ketten $S \subseteq D$ gilt $i(S) = S$. Daher folgt $\sqcup i(S) = \sqcup S = i(\sqcup S)$.

Schließlich betrachten wir die folgende Funktion $g : (\mathbb{Z}_\perp \to \mathbb{Z}_\perp) \to \mathbb{Z}_\perp$ mit

$$g(f) = \begin{cases} 0, & \text{falls } f(x) \neq \perp_{\mathbb{Z}_\perp} \text{ für alle } x \in \mathbb{Z} \\ \perp_{\mathbb{Z}_\perp}, & \text{sonst} \end{cases}$$

Diese Funktion g untersucht also, ob ihre Argumentfunktion f total ist (außer für die Eingabe $\perp$) und nur in diesem Fall liefert $g(f)$ das Ergebnis 0. Die Funktion g ist monoton: Falls $f_1 \sqsubseteq f_2$ ist, so ist entweder $f_1(x) = f_2(x)$ für alle $x \in \mathbb{Z}$ (dann gilt auch $g(f_1) = g(f_2)$) oder ansonsten ist f_1 nicht total auf $\mathbb{Z}$. In diesem Fall ist $g(f_1) = \perp$ und somit $g(f_1) \sqsubseteq g(f_2)$.

Die Funktion g ist aber nicht stetig. Betrachten wir beispielsweise die Kette $S = \{\mathsf{fact}_0, \mathsf{fact}_1, \ldots\}$. Alle Funktionen dieser Kette sind nicht total, aber die kleinste obere Schranke der Kette ist eine totale Funktion. Damit gilt $\sqcup g(S) = \perp_{\mathbb{Z}_\perp} \neq g(\sqcup S) = 0$.

Es stellt sich heraus, dass Funktionen wie g nicht berechenbar sind (da Eigenschaften wie Totalität nicht semi-entscheidbar sind). Eine Funktion g ist nur dann berechenbar, wenn sich ihr Ergebnis auf (unendlichen) Grenzwerten aus ihrem Verhalten auf endlichen Approximationen berechnen lässt. Daher ist Stetigkeit eine notwendige Voraussetzung für Berechenbarkeit. Der folgende Satz zeigt den Zusammenhang zwischen Monotonie und Stetigkeit.

Satz 2.1.15 (Stetigkeit und Monotonie) *Seien $\sqsubseteq_{D_1}$ und $\sqsubseteq_{D_2}$ vollständige Ordnungen auf D_1 bzw. D_2 und sei $f : D_1 \to D_2$ eine Funktion.*

(a) f ist stetig gdw. f monoton ist und für jede Kette S von D_1 gilt $f(\sqcup S) \sqsubseteq \sqcup f(S)$.

(b) *Falls D_1 nur endliche Ketten besitzt, dann ist f stetig gdw. f monoton ist.*

Beweis.

(a) Sei f stetig. Dass für jede Kette S der Zusammenhang $f(\sqcup S) \sqsubseteq \sqcup f(S)$ gilt, folgt sofort aus der Reflexivität von $\sqsubseteq$. Zu zeigen ist also, dass f auch monoton ist. Seien $d, d' \in D_1$ mit $d \sqsubseteq d'$. Dann ist $\{d, d'\}$ eine Kette und aufgrund der Stetigkeit folgt $f(d') = f(\sqcup\{d, d'\}) = \sqcup\{f(d), f(d')\}$. Dies bedeutet $f(d) \sqsubseteq f(d')$.

Sei nun f monoton und es gelte $f(\sqcup S) \sqsubseteq \sqcup f(S)$ für alle Ketten S. Aufgrund der Antisymmetrie reicht es, zu zeigen, dass $\sqcup f(S) \sqsubseteq f(\sqcup S)$ gilt, d.h., dass $f(\sqcup S)$ eine obere Schranke von $f(S)$ ist. Sei $e \in f(S)$, d.h., es existiert ein $d \in S$ mit $e = f(d)$. Da $d \sqsubseteq \sqcup S$ ist, folgt aus der Monotonie von f, dass $e = f(d) \sqsubseteq f(\sqcup S)$.

(b) Wegen (a) reicht es, zu zeigen, dass für jede endliche Kette $S = \{d_1, \ldots, d_n\}$ mit $d_1 \sqsubseteq d_2 \sqsubseteq \ldots \sqsubseteq d_n$ und für jede monotone Funktion f gilt, dass $f(\sqcup S) \sqsubseteq \sqcup f(S)$ ist. Es gilt $f(\sqcup S) = f(d_n) \in f(S)$ und damit folgt $f(\sqcup S) \sqsubseteq \sqcup f(S)$. □

Betrachten wir nun die folgende Funktion $ff : (\mathbb{Z}_\perp \to \mathbb{Z}_\perp) \to (\mathbb{Z}_\perp \to \mathbb{Z}_\perp)$, die wie folgt definiert ist. Hierbei seien $-$ und $\cdot$ die strikten Extensionen der Subtraktions- und der Multiplikationsfunktion.

$$(ff(g))(x) = \begin{cases} 1, & \text{falls } x \leq 0 \\ g(x-1) \cdot x, & \text{sonst} \end{cases}$$

Mit Hilfe von Satz 2.1.15 (a) lässt sich leicht zeigen, dass diese Funktion ebenfalls stetig ist. Es reicht, zu beweisen, dass ff monoton ist und dass für jede Kette S von $\mathbb{Z}_\perp \to \mathbb{Z}_\perp$ gilt $ff(\sqcup S) \sqsubseteq \sqcup ff(S)$.

Zum Beweis der Monotonie seien $f, g : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ mit $f \sqsubseteq g$. An den Stellen $x \leq 0$ liefern $ff(f)$ und $ff(g)$ dieselben Werte. Für sonstige Stellen x mit $f(x-1) = \perp_{\mathbb{Z}_\perp}$ ist der Wert von $ff(f)$ undefiniert und damit weniger definiert oder gleich dem von $ff(g)$. (Dies gilt insbesondere bei $x = \perp$, da die Subtraktionsfunktion $-$ strikt ist.) Falls $f(x-1) \neq \perp_{\mathbb{Z}_\perp}$, so folgt aus $f \sqsubseteq g$, dass $g(x-1) = f(x-1)$ ist, und damit ergibt sich $(ff(f))(x) = (ff(g))(x)$. Damit ist die Monotonie gezeigt.

Zur Stetigkeit ist nun noch zu beweisen, dass $ff(\sqcup S) \sqsubseteq \sqcup ff(S)$ ist. Sei $g = \sqcup S$. Zu zeigen ist dann $ff(g) \sqsubseteq \sqcup ff(S)$, d.h. $(ff(g))(x) \sqsubseteq \sqcup\{(ff(f))(x) \mid f \in S\}$ für alle $x \in \mathbb{Z}_\perp$ (nach Lemma 2.1.11 (b)).

Falls $(ff(g))(x) = \perp$ ist, so gilt dies offensichtlich. Wir betrachten daher nun den Fall $g(x-1) \neq \perp$. Da $g = \sqcup S$ ist, existiert also ein $f \in S$ mit $f(x-1) \neq \perp$ und da $\mathbb{Z}_\perp$ flach ist, folgt $f(x-1) = g(x-1)$. Für dieses f gilt $(ff(f))(x) = (ff(g))(x) \neq \perp$ und da es sich um einen flachen Domain handelt, gilt auch $\sqcup\{(ff(f))(x) \mid f \in S\} = (ff(f))(x) = (ff(g))(x)$.

Wir werden uns im Folgenden nur noch auf stetige Funktionen beschränken und verwenden $\langle D_1 \to D_2 \rangle$ als Funktionen-Domain. Der folgende Satz zeigt, dass $\sqsubseteq_{D_1 \to D_2}$ auch auf dieser Teilmenge der Funktionen eine vollständige partielle Ordnung ist. Dies ist nicht trivial, denn im allgemeinen ist für eine Teilmenge $D' \subseteq D$ die auf D' eingeschränkte Relation $\sqsubseteq$ nicht mehr vollständig, auch wenn sie auf D vollständig war. Der Grund ist, dass bei der Einschränkung von D auf D' die kleinsten oberen Schranken der Ketten fehlen könnten.

Beispielsweise ist $\sqsubseteq_{D_1 \to D_2}$ auf der Menge der nicht-totalen Funktionen nicht vollständig, da auf dieser Menge $\{\mathsf{fact}_0, \mathsf{fact}_1, \ldots\}$ eine Kette wäre, aber ihre kleinste obere Schranke ist total.

Satz 2.1.16 (Vollständigkeit des Funktionen-Domains) *Seien D_1, D_2 Domains mit entsprechenden cpo's. Dann ist $\langle D_1 \to D_2 \rangle$ der dazugehörige Funktionen-Domain und $\sqsubseteq_{D_1 \to D_2}$ ist eine vollständige partielle Ordnung auf $\langle D_1 \to D_2 \rangle$.*

Beweis. Das kleinste Element von $\langle D_1 \to D_2 \rangle$ ist $\bot_{D_1 \to D_2}$. Diese Funktion ist in der Tat stetig (denn jede konstante Funktion ist stetig).

Sei nun S eine Kette in $\langle D_1 \to D_2 \rangle$. Da $\sqsubseteq_{D_1 \to D_2}$ vollständig auf dem gesamten Funktionenraum ist (Satz 2.1.13 (c)), existiert eine kleinste obere Schranke $\sqcup S$, die eine Funktion von D_1 nach D_2 ist. Es ist zu zeigen, dass diese Funktion stetig ist (und somit auch in $\langle D_1 \to D_2 \rangle$ liegt).

Wir müssen also beweisen, dass $(\sqcup S)(\sqcup T) = \sqcup(\sqcup S)(T)$ für jede Kette T aus D_1 gilt. Man erhält

$$
\begin{array}{rcll}
(\sqcup S)(\sqcup T) & = & \sqcup S_{\sqcup T} & \text{nach Lemma 2.1.11 (b)} \\
& = & \sqcup\{f(\sqcup T) | f \in S\} & \\
& = & \sqcup\{\sqcup f(T) | f \in S\} & \text{da alle } f \in S \text{ stetig sind} \\
& = & \sqcup\{\sqcup\{f(x) | x \in T\} | f \in S\} & \\
& = & \sqcup\{f(x) | x \in T, f \in S\} & \\
& = & \sqcup\{\sqcup\{f(x) | f \in S\} | x \in T\} & \\
& = & \sqcup\{\sqcup S_x | x \in T\} & \\
& = & \sqcup\{(\sqcup S)(x) | x \in T\} & \text{nach Lemma 2.1.11 (b)} \\
& = & \sqcup(\sqcup S)(T) &
\end{array}
$$

Der Schritt

$$\sqcup\{\sqcup\{f(x) | x \in T\} | f \in S\} \;=\; \sqcup\{f(x) | x \in T, f \in S\}$$

lässt sich wie folgt beweisen.

Wir zeigen zunächst $\sqcup\{\sqcup\{f(x) | x \in T\} | f \in S\} \sqsubseteq \sqcup\{f(x) | x \in T, f \in S\}$, d.h., $\sqcup\{f(x) | x \in T, f \in S\}$ ist obere Schranke zu $\{\sqcup\{f(x) | x \in T\} | f \in S\}$. Dies ist offensichtlich, da $\sqcup\{f(x) | x \in T, f \in S\}$ größer oder gleich groß im Vergleich zu allen $f(x)$ mit $x \in T, f \in S$ ist. Damit ist es also eine obere Schranke zu $\{f(x) | x \in T\}$ (für beliebiges $f \in S$) und somit folgt jeweils $\sqcup\{f(x) | x \in T\} \sqsubseteq \sqcup\{f(x) | x \in T, f \in S\}$.

Nun zeigen wir $\sqcup\{f(x) | x \in T, f \in S\} \sqsubseteq \sqcup\{\sqcup\{f(x) | x \in T\} | f \in S\}$, d.h., $\sqcup\{\sqcup\{f(x) | x \in T\} | f \in S\}$ ist obere Schranke zu allen $f(x)$ mit $x \in T, f \in S$. Offensichtlich gilt für alle solchen x und f: $f(x) \sqsubseteq \sqcup\{f(x) | x \in T\} \sqsubseteq \sqcup\{\sqcup\{f(x) | x \in T\} | f \in S\}$.

Der Schritt

$$\sqcup\{f(x) | x \in T, f \in S\} \;=\; \sqcup\{\sqcup\{f(x) | f \in S\} | x \in T\}$$

wird auf analoge Weise gezeigt. □

2.1.3 Fixpunkte

In der denotationellen Semantik soll jedem Ausdruck der funktionalen Programmiersprache ein mathematisches Objekt zugeordnet werden. In diesem Abschnitt untersuchen wir die Frage, welche (stetige) Funktion einem HASKELL-Funktionssymbol zugeordnet werden soll (d.h. einer Variable, deren Bedeutung durch eine Funktionsdeklaration in HASKELL festgelegt wird). Betrachten wir zunächst das folgende nicht-rekursive Programm, das boolesche Werte in Zahlen umformt.

```
conv :: Bool -> Int
conv  = \b -> if b == True then 1 else 0
```

Der Wert des Ausdrucks `conv` sollte eine Funktion $f : \mathbb{B}_\perp \to \mathbb{Z}_\perp$ sein. Dabei sollte diese Funktion die definierende Gleichung von `conv` *erfüllen*. Die rechte Seite der Gleichung `\b -> if b == True then 1 else 0` enthält nur vordefinierte Konstrukte der Sprache, so dass wir davon ausgehen können, dass wir bereits wissen, was die Bedeutung dieses Ausdrucks ist (d.h., im allgemeinen sollte sich die Bedeutung eines Funktionssymbols aus den Bedeutungen der Funktionssymbole auf den rechten Seiten ergeben). Es wird sich herausstellen, dass der Wert des Ausdrucks `\b -> if b == True then 1 else 0` die folgende Funktion ist:

$$f(b) = \begin{cases} 1, & \text{falls } b = \texttt{True} \\ 0, & \text{falls } b = \texttt{False} \\ \perp_{\mathbb{Z}_\perp}, & \text{falls } b = \perp_{\mathbb{B}_\perp} \end{cases}$$

Die Bedeutung einer HASKELL-Deklaration wie oben ist dann, dass der Variablen `conv` auch diese Funktion f als Wert zugeordnet wird.

Betrachten wir nun ein rekursives HASKELL-Programm zur Berechnung der Fakultät.

```
fact :: Int -> Int
fact  = \x -> if x <= 0 then 1 else fact(x-1) * x
```

Der Wert des Ausdrucks `fact` sollte nun eine Funktion $f : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ sein und diesmal sollte die Funktion f die definierende Gleichung von `fact` erfüllen. Wir müssten also wie bei `conv` den Wert des Ausdrucks `\x -> if x <= 0 then 1 else fact(x-1) * x` berechnen und dann der Variablen `fact` diesen Wert zuordnen.

Der Ausdruck `\x -> if x <= 0 then 1 else fact(x-1) * x` enthält aber auf der rechten Seite die Variable `fact` selbst! Das Problem ist also, dass wir den Wert dieses Ausdrucks nur dann berechnen könnten, wenn wir schon den Wert von `fact` kennen würden.

Als Ausweg aus diesem Dilemma betrachten wir anstelle von `fact` die folgende Folge von nicht-rekursiven Definitionen. Hierbei ist `bot` ein Konstantensymbol, dessen Auswertung undefiniert ist, d.h., `bot` wird der Wert $\perp_{\mathbb{Z}_\perp}$ zugeordnet.

$$\begin{aligned}
\texttt{fact}_0 &= \texttt{\textbackslash x -> bot} \\
\texttt{fact}_1 &= \texttt{\textbackslash x -> if x <= 0 then 1 else fact}_0(\texttt{x} - 1) * \texttt{x} \\
\texttt{fact}_2 &= \texttt{\textbackslash x -> if x <= 0 then 1 else fact}_1(\texttt{x} - 1) * \texttt{x} \\
&\dots \\
\texttt{fact}_n &= \texttt{\textbackslash x -> if x <= 0 then 1 else fact}_{n-1}(\texttt{x} - 1) * \texttt{x} \\
&\dots
\end{aligned}$$

Jede der Variablen $\texttt{fact}_n$ hat nun als Wert eine Funktion $\mathsf{fact}_n : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$, die sich leicht bestimmen lässt (da es sich ja um nicht-rekursive Deklarationen handelt):

$$
\begin{aligned}
\mathsf{fact}_0(x) &= \perp \text{ für alle } x \in \mathbb{Z}_\perp \\
\mathsf{fact}_1(x) &= \begin{cases} x!, & \text{für } 0 \leq x < 1 \\ 1, & \text{für } x < 0 \\ \perp, & \text{für } x = \perp \text{ oder } 1 \leq x \end{cases} \\
\mathsf{fact}_2(x) &= \begin{cases} x!, & \text{für } 0 \leq x < 2 \\ 1, & \text{für } x < 0 \\ \perp, & \text{für } x = \perp \text{ oder } 2 \leq x \end{cases} \\
&\vdots \\
\mathsf{fact}_n(x) &= \begin{cases} x!, & \text{für } 0 \leq x < n \\ 1, & \text{für } x < 0 \\ \perp, & \text{für } x = \perp \text{ oder } n \leq x \end{cases}
\end{aligned}
$$

Man erkennt, dass jedes Funktionssymbol $\texttt{fact}_n$ mit Hilfe von $\texttt{fact}_{n-1}$ definiert ist und dass die dazugehörige Deutungsfunktion fact_n jeweils auf mehr Argumenten definiert ist als ihre Vorgängerin fact_{n-1}. Wie bereits erwähnt, bilden diese Funktionen also eine Kette

$$\mathsf{fact}_0 \sqsubseteq \mathsf{fact}_1 \sqsubseteq \ldots$$

Man erkennt auch, dass diese Funktionen immer näher an die eigentlich gewünschte Funktion herankommen, die auf allen positiven Zahlen die Fakultät berechnet und auf negativen Zahlen das Ergebnis 1 liefert. (fact_n berechnet bereits für alle $x < n$ das gewünschte Ergebnis). Jedes fact_n entspricht einer n-fachen Expansion der rekursiven Definition von `fact`, wobei der n-te rekursive Aufruf durch $\perp$ ersetzt wird, um eine potentiell unendliche Expansion zu vermeiden. Für Eingaben x, die weniger als n rekursive Aufrufe benötigen, entspricht $\mathsf{fact}_n(x)$ also bereits dem Wert von `fact x`. Beliebig gute Approximationen lassen sich erhalten, indem man die Rekursion genügend oft auffaltet. Die Deutung des Ausdrucks `fact` ist daher die Funktion f, die man als Grenzwert bei unendlich häufigem Auffalten der Rekursion erhält. Mit anderen Worten, es ist die kleinste obere Schranke der Kette $\{\mathsf{fact}_0, \mathsf{fact}_1, \ldots\}$. (Diese kleinste obere Schranke muss existieren, da es sich bei $\{\mathsf{fact}_0, \mathsf{fact}_1, \ldots\}$ um stetige Funktionen handelt und da nach Satz 2.1.16 die entsprechende Ordnung auf dem Funktionen-Domain vollständig ist.)

Der Schritt von einer Approximation fact_n zur nächsten Approximation fact_{n+1} geschieht durch Anwendung der (higher-order) Funktion $\mathit{ff} : \langle \mathbb{Z}_\perp \to \mathbb{Z}_\perp \rangle \to \langle \mathbb{Z}_\perp \to \mathbb{Z}_\perp \rangle$, die wir bereits im vorigen Abschnitt betrachtet hatten:

$$(\mathit{ff}(g))(x) = \begin{cases} 1, & \text{falls } x \leq 0 \\ g(x-1) \cdot x, & \text{sonst.} \end{cases}$$

In HASKELL ließe sich solch eine Funktion wie folgt definieren:

```
ff :: (Int -> Int) -> (Int -> Int)
ff g  = \x -> if x <= 0 then 1 else g(x-1) * x
```

Es gilt also

$$\begin{array}{rcl} \mathsf{fact}_0 & = & \mathit{ff}^0(\bot) \\ \mathsf{fact}_1 & = & \mathit{ff}^1(\bot) \\ \mathsf{fact}_2 & = & \mathit{ff}^2(\bot) \\ & \vdots & \end{array}$$

(wobei $\mathit{ff}^n(\bot)$ wieder die n-fache Anwendung von ff auf $\bot$ bezeichnet) und wir definieren die Bedeutung eines Ausdrucks wie `fact` als die kleinste obere Schranke dieser Kette, d.h. als

$$\sqcup\{\mathit{ff}^n(\bot) | n \in \mathbb{N}\}.$$

Man erkennt, dass ff eine Funktion wie `fact` als Argument bekommt und als Ergebnis eine Funktion berechnet, die wie die rechte Seite der bisherigen definierenden Gleichung von `fact` arbeitet. Die (higher-order) Funktion ff transformiert also die linke Seite der definierenden Gleichung von `fact` in den Wert der dazugehörigen rechten Seite (da der Wert der vordefinierten Funktionssymbole `-` und `*` die Funktionen $-$ und $\cdot$ sind).

Betrachten wir, wie diese Definition der Semantik von Funktionssymbolen mit unserer ursprünglichen Intuition zusammenhängt, dass Deutungen von Funktionssymbolen die dazugehörigen definierenden Gleichungen erfüllen sollen. Die Forderung, dass die Deutung f von `fact` die definierende Gleichung von `fact` erfüllen soll, ist gleichbedeutend dazu, dass f ein *Fixpunkt* der Funktion ff sein soll (d.h., dass $\mathit{ff}(f) = f$ gelten soll). Mit anderen Worten: Wenn wir ein Programm mit einer (möglicherweise rekursiven) Funktionsdeklaration wie die Deklaration von `fact` haben, so wird dem Funktionssymbol `fact` als Wert ein Fixpunkt der higher-order Funktion ff zugeordnet, die Funktionen sukzessive so transformiert, wie es die definierenden Gleichungen von `fact` beschreiben. Hierbei kann es natürlich mehrere Fixpunkte geben. Als Beispiel betrachten wir wieder die folgende Funktion.

```
non_term :: Int -> Int
non_term  = \x  -> non_term (x + 1)
```

Für die Funktion nn, die hier linke in rechte Seiten überführt, gilt $(nn(g))(x) = g(x+1)$. Alle konstanten Funktionen sind somit Fixpunkte von nn. Wir wählen als Bedeutung von `non_term` den kleinsten dieser Fixpunkte (d.h. die Fixpunktfunktion, die *am wenigsten definiert ist*). Die Motivation dafür ist, dass eine Funktion nur für solche Werte definiert sein sollte, bei denen dies auch wirklich im Programm angegeben ist. In unserem Fall ist es also der Fixpunkt, der bei allen Eingaben das Ergebnis $\bot$ liefert. Wir bezeichnen den kleinsten Fixpunkt einer Funktion f mit $\mathrm{lfp}\, f$ (für **l**east **f**ix**p**oint).

Hiermit haben wir also zunächst motiviert, warum `fact` als die kleinste obere Schranke $\sqcup\{\mathit{ff}^i(\bot) | i \in \mathbb{N}\}$ gedeutet werden sollte und anschließend haben wir motiviert, warum `fact` als der kleinste Fixpunkt $\mathrm{lfp}\, \mathit{ff}$ gedeutet werden sollte. In der Tat gilt

$$\mathrm{lfp}\, \mathit{ff} \;=\; \sqcup\{\mathit{ff}^i(\bot) | i \in \mathbb{N}\}.$$

Dieser fundamentale Zusammenhang ist der Fixpunktsatz von Tarski bzw. Kleene [Kle52].

Satz 2.1.17 (Fixpunktsatz) *Sei $\sqsubseteq$ eine vollständige partielle Ordnung auf D und sei $f : D \to D$ stetig. Dann besitzt f einen kleinsten Fixpunkt und es gilt* $\mathrm{lfp}\, f \;=\; \sqcup\{f^i(\bot) | i \in \mathbb{N}\}$.

Beweis. Wir zeigen zunächst durch Induktion über i, dass $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$ für alle $i \in \mathbb{N}$ gilt. Im Induktionsanfang ($i = 0$) gilt $f^0(\bot) = \bot \sqsubseteq f^1(\bot)$, wie gewünscht.

Im Induktionsschluss setzen wir voraus, dass $f^{i-1}(\bot) \sqsubseteq f^i(\bot)$ gilt. Da f stetig ist, ist f auch monoton (Satz 2.1.15 (a)), und daher folgt sofort $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$.

Demnach ist $\{f^i(\bot)|i \in \mathbb{N}\}$ also eine Kette und aufgrund der Vollständigkeit der Ordnung $\sqsubseteq$ existiert die kleinste obere Schranke $\sqcup\{f^i(\bot)|i \in \mathbb{N}\}$. Es bleibt zu zeigen, dass dies der kleinste Fixpunkt von f ist.

Es gilt

$$\begin{array}{rcll} f(\sqcup\{f^i(\bot)|i \in \mathbb{N}\}) & = & \sqcup f(\{f^i(\bot)|i \in \mathbb{N}\}) & \text{da } f \text{ stetig ist} \\ & = & \sqcup\{f^{i+1}(\bot)|i \in \mathbb{N}\} & \\ & = & \sqcup(\{f^{i+1}(\bot)|i \in \mathbb{N}\} \cup \{\bot\}) & \\ & = & \sqcup\{f^i(\bot)|i \in \mathbb{N}\}, & \end{array}$$

d.h., $\sqcup\{f^i(\bot)|i \in \mathbb{N}\}$ ist tatsächlich ein Fixpunkt von f.

Sei d ein weiterer Fixpunkt von f. Um $\sqcup\{f^i(\bot)|i \in \mathbb{N}\} \sqsubseteq d$ zu beweisen, reicht es, zu zeigen, dass $f^i(\bot) \sqsubseteq d$ für alle $i \in \mathbb{N}$ gilt. Dies zeigen wir durch Induktion über i. Im Induktionsanfang ($i = 0$) gilt offensichtlich $f^0(\bot) = \bot \sqsubseteq d$.

Im Induktionsschritt setzen wir als Induktionshypothese $f^{i-1}(\bot) \sqsubseteq d$ voraus. Aufgrund der Monotonie von f folgt daraus $f^i(\bot) \sqsubseteq f(d) = d$, denn d ist ein Fixpunkt von f. □

Funktionen wie ff, die aus Ausdrücken unserer Programmiersprache (nämlich den rechten Seiten von Deklarationen) gewonnen werden, sind immer stetig. Insofern ist durch den Fixpunktsatz garantiert, dass ihr kleinster Fixpunkt existiert und durch die kleinste obere Schranke der Kette $\{\bot, \mathit{ff}(\bot), \mathit{ff}^2(\bot), \ldots\}$ erhalten werden kann.

Insgesamt fassen wir also in der denotationellen Semantik die Definition einer Funktion als eine *(Fixpunkt)gleichung* über Funktionen auf. Da hier Gleichungen immer lösbar sind, erhalten wir mindestens eine Funktion als Lösung. Unter allen Lösungen wird dann eine bestimmte ausgewählt (nämlich die kleinste) und dem Funktionssymbol als Semantik zugeordnet.

2.2 Denotationelle Semantik von HASKELL

In diesem Abschnitt geben wir nun die Semantik von HASKELL-Ausdrücken an. In Abschnitt 2.2.1 zeigen wir zunächst, wie der hierfür verwendete Domain gebildet wird (d.h. aus welchem Bereich die Werte stammen, die den HASKELL-Ausdrücken zugeordnet werden). Hierbei muss man natürlich auch benutzerdefinierte Datenstrukturen und polymorphe Typen berücksichtigen. Anschließend definieren wir in Abschnitt 2.2.2 die Funktion $\mathcal{V}al$, die jedem Ausdruck einen Wert des Domains zuordnet. Dabei beschränken wir uns zunächst nur auf sogenannte einfache Programme und Ausdrücke, die kein Pattern Matching verwenden. In Abschnitt 2.2.3 stellen wir ein Verfahren vor, das (nahezu) beliebige HASKELL-Programme in einfache Programme übersetzt. Damit lässt sich dann auch die Semantik von komplexen Programmen angeben (als die Semantik des einfachen Programms, das durch die Übersetzung daraus entsteht).

2.2.1 Konstruktion von Domains

Nun geben wir an, auf welche Werte die Ausdrücke eines funktionalen Programms abgebildet werden. Insgesamt entsteht auf diese Weise nachher ein Gesamt-Domain Dom, der alle diese Werte enthält. Hierzu definieren wir zunächst den *Lift* einer Menge.

Definition 2.2.1 (Lift von Mengen) *Sei D eine Menge mit einer Relation $\sqsubseteq_D$. Der* Lift *von D ist $D_\perp = \{d^D | d \in D\} \cup \{\perp_{D_\perp}\}$. Die Relation $\sqsubseteq_{D_\perp}$ wird definiert als $e \sqsubseteq_{D_\perp} e'$ gdw. $e = \perp_{D_\perp}$ oder $e = d^D, e' = d'^D$ für $d, d' \in D$ und $d \sqsubseteq_D d'$. Falls D ein Domain ist, so ist auch $D_\perp$ ein Domain. Wir schreiben oft "d in $D_\perp$" anstelle von "d^D". Damit bezeichnet "d in $D_\perp$" also eine markierte Version von d, die in dem Lift $D_\perp$ auftritt.*

Der Lift einer Menge fügt also ein neues Bottom-Element hinzu, das kleiner als alle bisherigen Elemente (inklusive des bisherigen Bottom-Elements) ist. Dies wird in Abb. 2.5 verdeutlicht. Auf diese Weise entstanden bereits die Basis-Domains $\mathbb{Z}_\perp$, $\mathbb{B}_\perp$, $C_\perp$ und $F_\perp$ für Ausdrücke der Typen `Int`, `Bool`, `Char` und `Float`. Außerdem werden die Elemente aus der bisherigen Menge D so markiert, dass ihre Herkunft (aus D) deutlich wird.

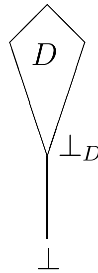


Abbildung 2.5: Lift eines Domains

Nun geben wir an, wie die Domains für Tupeltypen gebildet werden. Falls $D_1, \ldots, D_n$ die Domains für die Typen $\underline{\mathsf{type}}_1, \ldots, \underline{\mathsf{type}}_n$ sind, so ist $(D_1 \times \ldots \times D_n)_\perp$ der Domain für den Typ $(\underline{\mathsf{type}}_1, \ldots, \underline{\mathsf{type}}_n)$. Wenn d_1 der Wert ist, der einem Ausdruck $\underline{\mathsf{exp}}_1$ zugeordnet wird und d_2 dem Ausdruck $\underline{\mathsf{exp}}_2$ zugeordnet wird, so wird dem Ausdruck $(\underline{\mathsf{exp}}_1, \underline{\mathsf{exp}}_2)$ der Wert (d_1, d_2) zugeordnet. Der Unterschied zwischen $(d, \perp_{D_2})$ und $(\perp_{D_1}, \perp_{D_2})$ macht deutlich, dass die Tupelbildung eine nicht-strikte Operation ist. So kann man Funktionen auf Tupeln definieren, die nur die erste Komponente eines Tupels auswerten und deren Ergebnis auch dann definiert ist, wenn die zweite Komponente des Argumenttupels undefiniert ist.

Der Lift des Domains $(D_1 \times \ldots \times D_n)$ zu $(D_1 \times \ldots \times D_n)_\perp$ ist nötig, damit man zwischen dem völlig undefinierten Wert $\perp$ und dem n-Tupel $(\perp, \ldots, \perp)$ von undefinierten Werten unterscheiden kann. Hierzu betrachten wir die folgenden Deklarationen.

```
c :: (Int,Int)
c = c

d :: (Int,Int)
d = let bot = bot
    in  (bot,bot)
```

```
f :: (Int,Int) -> Int
f (x,y) = 0

g :: (Int,Int) -> Int
g z = 0
```

Die Semantik von `c` ist $\bot$, wohingegen `d` die Semantik $(\bot, \bot)$ hat. Der Unterschied zwischen diesen Werten wird an den Funktionen `f` und `g` deutlich. Die Auswertung von `f c` terminiert nicht, denn hier muss `f` zunächst sein Argument soweit auswerten, dass es ein Tupel mit zwei Komponenten ist. Hingegen terminiert die Auswertung von `g c`. Die Anwendung von `f` oder `g` auf `d` terminiert natürlich ebenfalls. Die Semantik von `f` ist also die Funktion $f : (\mathbb{Z}_\bot \times \mathbb{Z}_\bot)_\bot \to \mathbb{Z}_\bot$ mit $f(\bot) = \bot$ und $f(x, y) = 0$ für alle $x, y \in \mathbb{Z}_\bot$. In der Tat ist also $(\bot, \bot)$ mehr definiert als $\bot$. Hingegen hat `g` die Semantik $g : (\mathbb{Z}_\bot \times \mathbb{Z}_\bot)_\bot \to \mathbb{Z}_\bot$ mit $g(\bot) = g(x, y) = 0$ für alle $x, y \in \mathbb{Z}_\bot$.

Die Domains für Funktionentypen haben wir bereits angegeben: Falls D_1 und D_2 die Domains für die Typen $\underline{\mathsf{type}}_1$ und $\underline{\mathsf{type}}_2$ sind, so ist die Menge der stetigen Funktionen von D_1 nach D_2 (d.h. $\langle D_1 \to D_2 \rangle$) der Domain für den Typ $\underline{\mathsf{type}}_1 \to \underline{\mathsf{type}}_2$.

Wir müssen aber auch angeben, wie die Domains für benutzerdefinierte algebraische Datenstrukturen gebildet werden. Wir haben bereits gezeigt, dass $\sqsubseteq$ auf den Basis-Domains und auf den daraus gebildeten Produkt- und Funktionendomains eine vollständige partielle Ordnung ist (Satz 2.1.13 und 2.1.16). Die Eigenschaft, vollständige partielle Ordnung zu sein, bleibt mit den in funktionalen Programmiersprachen vorhandenen Möglichkeiten zur Datenkonstruktion erhalten.

Neben dem Einsatz zur Definition des Domains von Tupeltypen ist der Lift von Domains auch nötig, um einem undefinierten Ausdruck $\underline{\mathsf{exp}}$ einen anderen Wert als dem Ausdruck $\underline{\mathsf{constr}}$ $\underline{\mathsf{exp}}$ zuweisen zu können. Man benötigt dies also, um die Semantik nicht-strikt ausgewerteter Konstruktoren beschreiben zu können. Betrachten wir hierzu einen Datentyp $\underline{\mathsf{tyconstr}}$, der durch folgende Deklaration eingeführt wurde.

$$\texttt{data}\ \underline{\mathsf{tyconstr}} = \underline{\mathsf{constr}}\,\underline{\mathsf{type}}_1 \ldots \underline{\mathsf{type}}_n$$

Hier ist $\underline{\mathsf{constr}}$ also ein Konstruktor vom Typ $\underline{\mathsf{type}}_1 \to \ldots \to \underline{\mathsf{type}}_n \to \underline{\mathsf{tyconstr}}$. Wenn $D_1, \ldots, D_n$ die Domains sind, die den Typen $\underline{\mathsf{type}}_1, \ldots, \underline{\mathsf{type}}_n$ zugeordnet werden, so wählen wir

$$(\{\underline{\mathsf{constr}}\} \times D_1 \times \ldots \times D_n)_\bot$$

als den Domain des Typs $\underline{\mathsf{tyconstr}}$. Dies ist der Domain für eine Variante des Typs $(\underline{\mathsf{type}}_1, \ldots, \underline{\mathsf{type}}_n)$, dessen Objekte mit einem zusätzlichen Konstruktor $\underline{\mathsf{constr}}$ versehen sind. Wenn also $\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n$ Ausdrücke mit den Werten $d_1, \ldots, d_n$ sind, dann hat der Ausdruck $\underline{\mathsf{constr}}\,\underline{\mathsf{exp}}_1 \ldots \underline{\mathsf{exp}}_n$ als Wert den Tupel $(\underline{\mathsf{constr}}, d_1, \ldots, d_n)$. Der Konstruktor $\underline{\mathsf{constr}}$ wird wie eine Markierung verwendet, die den Argumenten mit Hilfe einer Tupelbildung angehängt wird. Wir schreiben auch $\underline{\mathsf{constr}}(d_1, \ldots, d_n)$ oder $\underline{\mathsf{constr}}\ d_1 \ldots d_n$ statt $(\underline{\mathsf{constr}}, d_1, \ldots, d_n)$, um die Lesbarkeit zu erhöhen, sofern dies zu keinen Verwechslungen führt. Der Domain enthält also im Prinzip alle *Grundterme*, die nur aus Konstruktoren und den Bottom-Symbolen $\bot$ gebildet werden. (Vordefinierte Datenstrukturen wie `Int` haben unendlich viele (nullstellige) Konstruktoren $0, 1, -1, \ldots$) Falls $\underline{\mathsf{exp}}_1$ den Wert $\bot_{D_1}$ hat, so hat demnach

$\underline{\mathsf{constr}}\,\underline{\mathsf{exp}}_1 \ldots \underline{\mathsf{exp}}_n$ den Wert $(\underline{\mathsf{constr}}, \bot_{D_1}, \ldots)$. Dieser Wert ist verschieden von dem völlig undefinierten Wert $\bot_{(\{\underline{\mathsf{constr}}\} \times D_1 \times \ldots \times D_n)_\bot}$ des Typs $\underline{\mathsf{tyconstr}}$, denn über den Wert des Ausdrucks $\underline{\mathsf{constr}}\,\underline{\mathsf{exp}}_1 \ldots \underline{\mathsf{exp}}_n$ ist ja immerhin bekannt, dass er mit Hilfe des Konstruktors $\underline{\mathsf{constr}}$ gebildet wurde (er ist also nur "partiell undefiniert").

Analog geht man bei nullstelligen Konstruktoren vor. Bei der Datentypdeklaration

$$\texttt{data}\ \underline{\mathsf{tyconstr}} = \underline{\mathsf{constr}}$$

wird dem Typ $\underline{\mathsf{tyconstr}}$ der Domain $\{\underline{\mathsf{constr}}\}_\bot$ zugeordnet.

Der Lift von Domains kann für benutzerdefinierte algebraische Datentypen mit nur einem Konstruktor verwendet werden. Im allgemeinen haben solche Datentypen aber mehrere Konstruktoren. Um hierfür Domains bilden zu können, definieren wir die *verschmolzene Summe* von Domains (engl. *union* oder *coalesced sum*).

Definition 2.2.2 (Verschmolzene Summe von Domains) *Seien $D_1, \ldots, D_n$ Domains. Die* verschmolzene Summe *von $D_1, \ldots, D_n$ wird definiert als $D_1 \oplus \ldots \oplus D_n = \{d^{D_1} | d \in D_1, d \neq \bot_{D_1}\} \cup \ldots \cup \{d^{D_n} | d \in D_n, d \neq \bot_{D_n}\} \cup \{\bot_{D_1 \oplus \ldots \oplus D_n}\}$. Die Relation $\sqsubseteq_{D_1 \oplus \ldots \oplus D_n}$ wird definiert als $e \sqsubseteq_{D_1 \oplus \ldots \oplus D_n} e'$ gdw. $e = \bot_{D_1 \oplus \ldots \oplus D_n}$ oder $e = d^{D_i}$, $e' = d'^{D_i}$ und $d \sqsubseteq_{D_i} d'$ für ein $i \in \{1, \ldots, n\}$. Die verschmolzene Summe $D_1 \oplus \ldots \oplus D_n$ von Domains ist wieder ein Domain. Wir schreiben oft "d in $D_1 \oplus \ldots \oplus D_n$" anstelle von "$d^{D_i}$", wobei $d \in D_i$. Damit bezeichnet "d in $D_1 \oplus \ldots \oplus D_n$" also eine markierte Version von d, die in der verschmolzenen Summe $D_1 \oplus \ldots \oplus D_n$ auftritt.*

Die verschmolzene Summe entspricht also der Vereinigung von $D_1, \ldots, D_n$, wobei aber die Elemente so markiert werden, dass ihre Herkunft (aus einem der D_i) deutlich wird. Außerdem werden die verschiedenen Bottom-Elemente $\bot_{D_1}, \ldots, \bot_{D_n}$ zu einem gemeinsamen neuen Bottom-Element $\bot_{D_1 \oplus \ldots \oplus D_n}$ verschmolzen. Dies wird in Abb. 2.6 verdeutlicht.

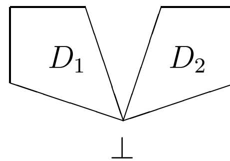


Abbildung 2.6: Verschmolzene Summe zweier Domains

Auch unendliche verschmolzene Summen sind möglich und wir werden auch den Domain

$$\mathsf{Tuples}_0 \oplus \mathsf{Tuples}_2 \oplus \mathsf{Tuples}_3 \oplus \ldots$$

mit $\mathsf{Tuples}_0 = \{(\,)\}_\bot$, $\mathsf{Tuples}_2 = (\mathsf{Dom} \times \mathsf{Dom})_\bot$, $\mathsf{Tuples}_3 = (\mathsf{Dom} \times \mathsf{Dom} \times \mathsf{Dom})_\bot, \ldots$ betrachten. Hierbei ist Dom der Domain aller Werte, den wir im Folgenden definieren werden. Den Teildomain der einstelligen Tupel (Dom) benötigen wir nicht, da ein Ausdruck $(\underline{\mathsf{exp}})$ als identisch zu dem Ausdruck $\underline{\mathsf{exp}}$ angesehen wird.

Verschmolzene Summen von Domains sind nötig, um Ausdrücken von benutzerdefinierten Datenstrukturen mit mehreren Konstruktoren geeignete Werte zuweisen zu können. Einem Datentyp $\underline{\mathsf{tyconstr}}$, der durch folgende Deklaration eingeführt wurde

$$\texttt{data}\ \underline{\mathsf{tyconstr}} = \underline{\mathsf{constr}}_1\,\underline{\mathsf{type}}_{1,1} \ldots \underline{\mathsf{type}}_{1,n_1} \mid \ldots \mid \underline{\mathsf{constr}}_k\,\underline{\mathsf{type}}_{k,1} \ldots \underline{\mathsf{type}}_{k,n_k},$$

wird der Domain $D_1 \oplus \ldots \oplus D_k$ zugeordnet. Hierbei ist D_i jeweils der Domain für die Variante des Typs $(\underline{\mathsf{type}}_{i,1}, \ldots, \underline{\mathsf{type}}_{i,n_i})$, dessen Objekte mit einem zusätzlichen Konstruktor $\underline{\mathsf{constr}}_i$ versehen sind, d.h., wir haben $D_i = (\{\underline{\mathsf{constr}}_i\} \times D_{i,1} \times \ldots \times D_{i,n_i})_\perp$, wenn $D_{i,j}$ jeweils der Domain für den Typ $\underline{\mathsf{type}}_{i,j}$ ist. Analog verhält es sich, wenn manche der vorkommenden Konstruktoren nullstellig sind. Sofern die obige Datentypdeklaration *nicht-rekursiv* ist, kann man auf diese Weise eindeutig festlegen, wie der neue Domain aus den bereits bekannten Domains $D_{i,j}$ gebildet wird.

Im allgemeinen sind solche Datentypdeklarationen aber rekursiv. Als Beispiel betrachten wir die folgende Deklaration des Typs für natürliche Zahlen.

```
data  Nats = Zero | Succ Nats
```

Sei $D_{\mathtt{Nats}}$ der Domain, den wir dem Typ `Nats` zuordnen wollen. Nach den obigen Überlegungen muss dann gelten

$$D_{\mathtt{Nats}} = \{\mathtt{Zero}\}_\perp \oplus (\{\mathtt{Succ}\} \times D_{\mathtt{Nats}})_\perp.$$

Das fundamentale Resultat der Domain-Theorie [Sco76] besagt, dass solche Domain-Gleichungen tatsächlich lösbar sind. (Hierzu ist es wichtig, dass wir z.B. nur solche Funktionenräume als Domain betrachten, in denen alle Funktionen stetig sind.) Die allgemeine Definition von Domains und der entsprechende Beweis hierzu sprengen allerdings den Rahmen der Vorlesung; der interessierte Leser sei auf [Sto81] verwiesen. Ähnlich wie bei der Semantik rekursiver Funktionen ist der Domain für den Typ `Nats` der *kleinste Fixpunkt* der Operation dd auf Domains, die einen Domain D auf den Domain $\{\mathtt{Zero}\}_\perp \oplus (\{\mathtt{Succ}\} \times D)_\perp$ abbildet. Der "kleinste" aller Domains ist der einelementige Domain $\{\perp\}$. Durch einmalige Anwendung von dd auf $\{\perp\}$ erhält man den Domain

$$\{\mathtt{Zero}\}_\perp \oplus (\{\mathtt{Succ}\} \times \{\perp\})_\perp = \{\perp, \mathtt{Zero}, (\mathtt{Succ}, \perp)\}$$

mit der Ordnung $\perp \sqsubseteq \mathtt{Zero}$ und $\perp \sqsubseteq (\mathtt{Succ}, \perp)$. Durch nochmalige Anwendung von dd erhält man

$$\{\mathtt{Zero}\}_\perp \oplus (\{\mathtt{Succ}\} \times \{\perp, \mathtt{Zero}, (\mathtt{Succ}, \perp)\})_\perp = \{\perp, \mathtt{Zero}, (\mathtt{Succ}, \perp), (\mathtt{Succ}, \mathtt{Zero}), (\mathtt{Succ}, (\mathtt{Succ}, \perp))\},$$

wobei $\perp \sqsubseteq \mathtt{Zero}$, $\perp \sqsubseteq (\mathtt{Succ}, \perp) \sqsubseteq (\mathtt{Succ}, \mathtt{Zero})$ und $(\mathtt{Succ}, \perp) \sqsubseteq (\mathtt{Succ}, (\mathtt{Succ}, \perp))$. Der kleinste Fixpunkt von dd (d.h. der Domain, der dem Typ `Nats` zugeordnet wird) ist in Abb. 2.7 dargestellt. Dies bedeutet insbesondere, dass unser Domain auch einen Wert enthält, bei dem unendlich viele `Succ`'s auf $\perp$ angewendet werden. Dieser Wert würde z.B. einem Ausdruck wie

```
let x = Succ(x) in x
```

zugeordnet werden. Insgesamt besteht der Domain also wieder im Prinzip aus allen *Grundtermen*, die mit Hilfe von Konstruktoren und $\perp$ gebildet werden, wobei aber bei reflexiven Konstruktoren (deren Ergebnistyp auch einer ihrer Argumenttypen ist) auch "unendliche Terme" mitbetrachtet werden.

Bislang haben wir uns nur auf monomorphe Ausdrücke beschränkt. Um auch polymorphe Ausdrücke behandeln zu können, betrachten wir einen Gesamt-Domain Dom, der

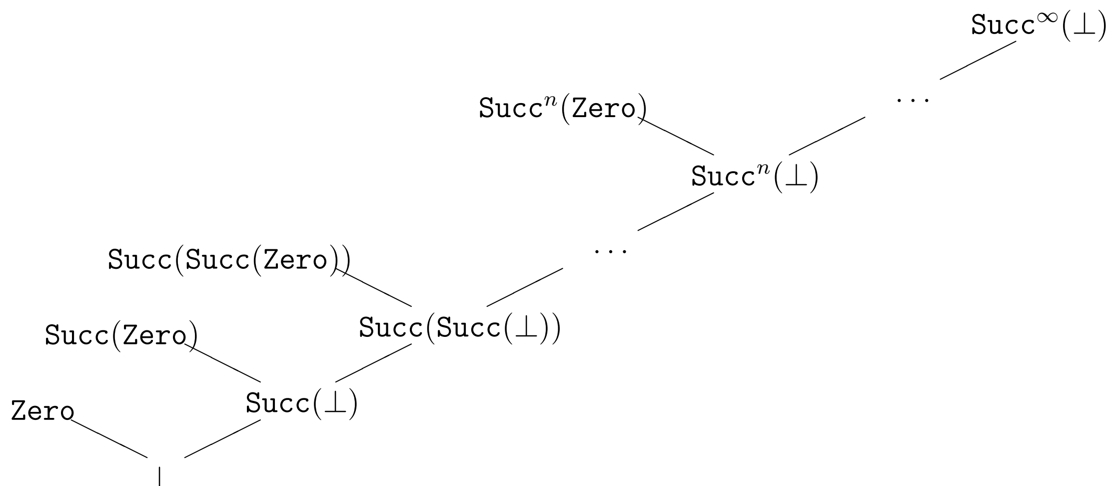


Abbildung 2.7: Domain für `Nats`

wiederum Teil-Domains für die verschiedenen Typen besitzt. Mit einem solchen Gesamt-Domain ist es möglich, polymorphen Ausdrücken auch genau einen Wert zuzuordnen; nämlich einen Wert, der auch in allen Teil-Domains für alle Instantiierungen dieses Typs liegt. Der Teil-Domain von Werten für einen polymorphen Typ $\underline{\mathsf{type}}$ ist die Schnittmenge der Teil-Domains für alle Instantiierungen von $\underline{\mathsf{type}}$. Zur Illustration betrachten wir den Teil-Domain für den Typ `[a]`, wobei `a` eine Typvariable ist. Wenn ein Ausdruck den Typ `[a]` hat, so hat er auch den Typ `[`$\underline{\mathsf{type}}$`]` für alle Typen $\underline{\mathsf{type}}$. Der Wert dieses Ausdrucks muss also in allen Domains für alle Typen der Form `[`$\underline{\mathsf{type}}$`]` liegen. Ein Beispiel für einen solchen Ausdruck ist die leere Liste `[]`. Der Wert dieses Ausdrucks liegt im Teil-Domain für `[a]` und daher auch in den Teil-Domains für `[Int]`, `[Bool]`, etc. (Der Teil-Domain für den Typ `[a]` enthält $\bot$, `[]`, $\bot$ `: []`, etc.)

Der Einfachheit halber betrachten wir in der folgenden Definition der Semantik eine eingeschränkte Version von HASKELL ohne den vordefinierten Typ der Listen. Dies bedeutet jedoch keine echte Einschränkung, da sich dieser Typ natürlich auch vom Benutzer definieren lässt.

```
data List a = Nil | Cons a  (List a)
```

Der Domain $D_{\texttt{List Nats}}$, der dem Typ `List Nats` zugeordnet wird, ist der kleinste Fixpunkt der Domain-Gleichung

$$D_{\texttt{List Nats}} = \{\texttt{Nil}\}_\bot \oplus (\{\texttt{Cons}\} \times D_{\texttt{Nats}} \times D_{\texttt{List Nats}})_\bot.$$

Dem polymorphen Typ `List a` wird hingegen der Domain

$$D_{\texttt{List a}} = \bigcap_{\underline{\mathsf{type}} \text{ ist ein Typ}} D_{\texttt{List}\,\underline{\mathsf{type}}}$$

zugeordnet.

Damit können wir nun den Domain Dom zu einem gegebenen HASKELL-Programm definieren. Wie erwähnt, beschränken wir uns auf Programme, bei denen die vordefinierten

Listenoperationen nicht auftreten. Der Einfachheit halber betrachten wir auch keine Typabkürzungen (mit `type`) und keine Typklassen.

Definition 2.2.3 (Domain eines Programms) *Zu einem* HASKELL*-Programm wie oben sei* Con_n *die Menge der n-stelligen Datenkonstruktoren (hierbei werden* $\mathbb{Z} \cup \mathbb{B} \cup C \cup F$ *auch als nullstellige Datenkonstruktoren betrachtet). Der Domain* Dom *des Programms ist definiert als der kleinste Domain, der die folgende Gleichung erfüllt:*

$$\mathsf{Dom} = \mathsf{Functions} \oplus \mathsf{Tuples}_0 \oplus \mathsf{Tuples}_2 \oplus \mathsf{Tuples}_3 \oplus \ldots \oplus \mathsf{Constructions}_0 \oplus \mathsf{Constructions}_1 \oplus \ \ldots$$

wobei

$$\begin{aligned}
\mathsf{Functions} &= \langle \mathsf{Dom} \to \mathsf{Dom} \rangle_\perp \\
\mathsf{Tuples}_0 &= \{(\,)\}_\perp \\
\mathsf{Tuples}_n &= (\mathsf{Dom}^n)_\perp \\
\mathsf{Constructions}_n &= (\mathsf{Con}_n \times \mathsf{Dom}^n)_\perp
\end{aligned}$$

Selbstverständlich enthält Dom auch Werte, die keinem (typkorrekten) HASKELL-Ausdruck zugeordnet werden (z.B. (`Succ`, `True`)).

2.2.2 Semantik einfacher HASKELL-Programme

Bei der Definition der denotationellen Semantik wird die Bedeutung eines Ausdrucks mit Hilfe der Bedeutungen seiner Teilausdrücke angegeben. Daher müssen wir auch Teilausdrücke betrachten und ihre Bedeutung unabhängig von ihrem Kontext angeben können. Im allgemeinen enthält dieser Kontext aber die Definition der Variablen, die in diesem Teilausdruck auftreten. Betrachten wir z.B. den Ausdruck `let x = 3 in plus x 2`. Der Wert des Teilausdrucks `plus x 2` lässt sich nur richtig angeben, wenn wir dabei berücksichtigen, dass wir uns hier in einer Umgebung befinden, in der die Variable `x` den Wert `3` hat. (Genauso verhält es sich bei der Variablen `plus`, deren Definition sich ebenfalls in der Umgebung befinden muss.)

Im allgemeinen kann man also den Wert eines Ausdrucks nur angeben, wenn man weiß, wie die darin vorkommenden Variablen belegt sind. Wir benötigen daher das Konzept des *Environments* (bzw. der Variablenbelegung), um anzugeben, welche Werte die im Ausdruck vorkommenden Variablen haben. Ein Environment ρ ist einfach eine Funktion, die Variablen auf Werte abbildet. Beispielsweise könnte ρ(`x`) die Zahl 3 sein und ρ(`plus`) die Additionsfunktion. Im Folgenden unterscheiden wir nicht zwischen Variablen $\underline{\mathsf{var}}$ und Variablenoperatoren $\underline{\mathsf{varop}}$ (die ja lediglich Variablen sind, die in Infix-Schreibweise verwendet werden) und verwenden die Infix-Schreibweise nur noch in den Beispielen.

Definition 2.2.4 (Environment) *Zu einem* HASKELL*-Programm mit dem Domain* Dom *ist* ρ *ein Environment, wenn* ρ *eine partielle Funktion ist, die Variablen auf Werte des Domains* Dom *abbildet, so dass* ρ *nur für endlich viele Variablen definiert ist. Wir haben also* $\rho : \mathsf{Var} \to \mathsf{Dom}$*, wenn* Var *die Menge aller (potentiellen) Variablen bezeichnet. Die Menge aller Environments zu einem Programm wird mit* Env *bezeichnet. Ein Environment* ρ *mit dem Definitionsbereich* $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$ *und mit* $\rho(\underline{\mathsf{var}}_i) = d_i$ *wird auch als* $\rho = \{\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n \,/\, d_1, \ldots, d_n\}$ *geschrieben.*

Wir schreiben $\rho_1 + \rho_2$ für das Environment, das sich wie ρ_2 für alle Elemente des Definitionsbereichs von ρ_2 verhält. Für Elemente, auf denen ρ_2 nicht definiert ist, verhält es sich wie ρ_1. Wir haben also

$$(\rho_1 + \rho_2)(\underline{\mathsf{var}}) = \begin{cases} \rho_2(\underline{\mathsf{var}}), & \textit{falls } \rho_2(\underline{\mathsf{var}}) \textit{ definiert ist} \\ \rho_1(\underline{\mathsf{var}}), & \textit{sonst.} \end{cases}$$

Sei ω das initiale Environment*, das allen Variablen, die in* HASKELL *vordefinierte Operationen bezeichnen, die jeweiligen "strikten Extensionen" dieser Operationen zuordnet. Beispielsweise ist $\omega(+) : \mathsf{Dom} \to \mathsf{Dom} \to \mathsf{Dom}$ die Funktion mit*

$$(\omega(+))\, x\, y = \begin{cases} x + y, & \textit{falls } x, y \in \mathbb{Z} \textit{ oder } x, y \in F \\ \bot, & \textit{sonst} \end{cases}$$

Auf allen anderen Variablen ist ω undefiniert.

Die Bedeutung eines Ausdrucks ist also eine Funktion von einem Environment in den Domain Dom. Wir schreiben daher $\mathcal{V}al[\![\underline{\mathsf{exp}}]\!]\rho$ für den Wert von $\underline{\mathsf{exp}}$, der von dem Environment ρ abhängt. $\mathcal{V}al$ ist also eine Funktion höherer Ordnung, die HASKELL-Ausdrücke auf eine Funktion von Env nach Dom abbildet.

Um die Präsentation der Semantik zu erleichtern, betrachten wir zunächst nur eine einfache Teilmenge von HASKELL ohne Pattern Matching. Anschließend werden wir in Abschnitt 2.2.3 aber zeigen, wie kompliziertere HASKELL-Programme (automatisch) in äquivalente Programme unserer einfacheren Teilmenge überführt werden können. Dadurch ist dann auch die Semantik dieser komplizierteren Programme festgelegt. Es zeigt sich also, dass die im Folgenden beschriebene Teilmenge von HASKELL bereits eigentlich ausreicht und die darüber hinausgehenden Sprachkonstrukte nur "syntaktischer Zucker" sind, um Programme leichter lesbar zu machen. Die hier betrachtete Teilmenge von HASKELL kann sogar noch weiter auf den sogenannten *Lambda-Kalkül* eingeschränkt werden, der allen funktionalen Sprachen zugrunde liegt. Wir werden darauf in Kapitel 3 zurückkommen.

Wir beschränken uns auf Programme, die außer Deklarationen von algebraischen Datentypen (mit `data`) nur eine einzige Pattern-Deklaration der Form

$$\underline{\mathsf{var}} = \underline{\mathsf{exp}}$$

besitzen. Bei der Form der Ausdrücke verwenden wir (wie zuvor erwähnt) nicht die vordefinierten Listenfunktionen (sondern diese müssen selbst mittels `data` definiert werden). Anstelle der n-fachen Anwendung von Ausdrücken aufeinander betrachten wir nur die Anwendung von zwei Ausdrücken. Die ist jedoch keine Einschränkung, da $(\underline{\mathsf{exp}}_1 \ldots \underline{\mathsf{exp}}_n)$ äquivalent ist zu $(\ldots((\underline{\mathsf{exp}}_1\, \underline{\mathsf{exp}}_2)\, \underline{\mathsf{exp}}_3) \ldots \underline{\mathsf{exp}}_n)$. Außerdem schließen wir im Moment Ausdrücke mit dem `case`-Konstrukt aus und betrachten nur Lambda-Ausdrücke mit Variablen anstatt von Patterns. Schließlich betrachten wir auch keine lokalen Deklarationen mit "`where`", sondern nur solche, die das `let`-Konstrukt benutzen.

Definition 2.2.5 (Einfache HASKELL-Programme) *Ein einfaches* HASKELL*-Programm ist ein Programm ohne Typabkürzungen und Typklassen und ohne die in* HASKELL *vordefinierten Listen, dessen (einzige) Deklaration anhand der folgenden Grammatik erzeugt*

wird:

$$
\begin{array}{lcl}
\underline{\mathsf{decl}} & \rightarrow & \underline{\mathsf{var}} = \underline{\mathsf{exp}} \\
\underline{\mathsf{exp}} & \rightarrow & \underline{\mathsf{var}} \\
 & | & \underline{\mathsf{constr}} \\
 & | & \underline{\mathsf{integer}} \\
 & | & \underline{\mathsf{float}} \\
 & | & \underline{\mathsf{char}} \\
 & | & (\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n), \quad n \geq 0 \\
 & | & (\underline{\mathsf{exp}}_1 \ \underline{\mathsf{exp}}_2) \\
 & | & \texttt{if}\ \underline{\mathsf{exp}}_1\ \texttt{then}\ \underline{\mathsf{exp}}_2\ \texttt{else}\ \underline{\mathsf{exp}}_3 \\
 & | & \texttt{let}\ \underline{\mathsf{var}} = \underline{\mathsf{exp}}\ \texttt{in}\ \underline{\mathsf{exp}}' \\
 & | & \backslash\ \underline{\mathsf{var}}\ \texttt{->}\ \underline{\mathsf{exp}}
\end{array}
$$

Die bereits betrachtete Definition der Fakultätsfunktion

```
fact  = \x -> if x <= 0 then 1 else fact(x-1) * x
```

ist natürlich ein einfaches HASKELL-Programm. Es enthält einige vordefinierte HASKELL-Variablen wie `<=`, `-` und `*`, deren Bedeutung aber bereits im initialen Environment ω festliegt.

Anstelle die Semantik von Ausdrücken $\underline{\mathsf{exp}}'$ bei einem zugrunde liegenden Programm mit der Deklaration $\underline{\mathsf{var}} = \underline{\mathsf{exp}}$ anzugeben, können wir dazu gleichbedeutend auch die Semantik des Ausdrucks

$$\texttt{let}\ \underline{\mathsf{var}}\ \texttt{=}\ \underline{\mathsf{exp}}\ \texttt{in}\ \underline{\mathsf{exp}}'$$

angeben. Indem man alle Deklarationen als lokal zum jeweils untersuchten Ausdruck betrachtet, kann man sich also auf die Semantik von Ausdrücken anstelle von Programmen beschränken.

Die Bedeutung des Werts `fact 2` im obigen Programm mit der Fakultätsfunktion ist also identisch zur Bedeutung des Ausdrucks

$$\texttt{let}\quad \mathsf{fact} = \backslash \mathsf{x}\ \texttt{->}\ \mathsf{if}\ \mathsf{x} \mathrel{\texttt{<=}} 0\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{fact}(\mathsf{x}-1) * \mathsf{x} \quad \texttt{in} \quad \mathsf{fact}\ 2$$

(im leeren Programm).

Wir gehen bei der Definition der Semantik davon aus, dass das Programm bereits auf Typkorrektheit überprüft wurde und auch syntaktisch korrekt ist. Dann können wir ggf. vorhandene Typdeklarationen in dem Programm ignorieren und lediglich eine Semantik für typkorrekte Ausdrücke angeben. Ein Verfahren zur automatischen Typüberprüfung werden wir in Kapitel 4 betrachten.

Wie erwähnt, besteht Dom im Prinzip aus allen (möglicherweise unendlichen) Grundtermen, die mit Hilfe von Konstruktoren und $\perp$ aufgebaut sind. Da wir alle Ausdrücke (verschiedener Typen) auf einen gemeinsamen Domain Dom abbilden, bedeutet das aber auch, dass der Domain auch Werte wie (`Succ`, `True`) (bzw. `Succ`(`True`)) enthält. Diese Werte können jedoch keinem korrekt getypten Ausdruck zugeordnet werden, so dass ihr Vorhandensein in Dom keine Probleme aufwirft.

Für die folgende Definition ist es hilfreich, die Variablen, die in einem Ausdruck auftreten, danach zu unterscheiden, ob sie *frei* oder durch ein davor stehendes Lambda oder

ein `let` gebunden sind. Ein Ausdruck, der nur eine Variable $\underline{\mathsf{var}}$ ist, hat $\underline{\mathsf{var}}$ als *freie* Variable. Ausdrücke, die durch Tupelbildung, Anwendung oder `if` entstehen, haben diejenigen freien Variablen, die in einem ihrer direkten Teilausdrücke frei sind. In einem Ausdruck `let` $\underline{\mathsf{var}} = \underline{\mathsf{exp}}$ `in` $\underline{\mathsf{exp}}'$ sind alle freien Variablen von $\underline{\mathsf{exp}}$ oder $\underline{\mathsf{exp}}'$ außer $\underline{\mathsf{var}}$ frei. In einem Ausdruck der Form \ $\underline{\mathsf{var}}$ `->` $\underline{\mathsf{exp}}$ sind alle freien Variablen von $\underline{\mathsf{exp}}$ außer $\underline{\mathsf{var}}$ frei.

Definition 2.2.6 (Freie Variablen eines HASKELL-Ausdrucks) *Für jeden einfachen HASKELL-Ausdruck* $\underline{\mathsf{exp}}$ *definieren wir* free($\underline{\mathsf{exp}}$)*, die Menge seiner* freien Variablen*, wie folgt:*

- $\mathrm{free}(\underline{\mathsf{var}}) = \{\underline{\mathsf{var}}\}$
- $\mathrm{free}(\underline{\mathsf{constr}}) = \mathrm{free}(\underline{\mathsf{integer}}) = \mathrm{free}(\underline{\mathsf{float}}) = \mathrm{free}(\underline{\mathsf{char}}) = \varnothing$
- $\mathrm{free}((\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n)) = \mathrm{free}(\underline{\mathsf{exp}}_1) \cup \ldots \cup \mathrm{free}(\underline{\mathsf{exp}}_n)$
- $\mathrm{free}((\underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2)) = \mathrm{free}(\underline{\mathsf{exp}}_1) \cup \mathrm{free}(\underline{\mathsf{exp}}_2)$
- $\mathrm{free}(\texttt{if}\ \underline{\mathsf{exp}}_1\ \texttt{then}\ \underline{\mathsf{exp}}_2\ \texttt{else}\ \underline{\mathsf{exp}}_3) = \mathrm{free}(\underline{\mathsf{exp}}_1) \cup \mathrm{free}(\underline{\mathsf{exp}}_2) \cup \mathrm{free}(\underline{\mathsf{exp}}_3)$
- $\mathrm{free}(\texttt{let}\ \underline{\mathsf{var}} = \underline{\mathsf{exp}}\ \texttt{in}\ \underline{\mathsf{exp}}') = (\mathrm{free}(\underline{\mathsf{exp}}) \cup \mathrm{free}(\underline{\mathsf{exp}}')) \setminus \{\underline{\mathsf{var}}\}$
- $\mathrm{free}(\backslash\ \underline{\mathsf{var}}\ \texttt{->}\ \underline{\mathsf{exp}}) = \mathrm{free}(\underline{\mathsf{exp}}) \setminus \{\underline{\mathsf{var}}\}$

Die folgende Definition führt die Funktion $\mathcal{V}al$ ein, die jedem Ausdruck $\underline{\mathsf{exp}}$ seinen Wert in einem Environment ρ zuordnet. Damit dieser Wert auch tatsächlich definiert ist, setzen wir voraus, dass das Environment ρ auf allen in diesem Ausdruck *frei* vorkommenden Variablen definiert ist. Wenn $\rho(\texttt{x})$ oder $\rho(\texttt{plus})$ nicht definiert ist, so ist auch der Wert des Ausdrucks `plus x 3` nicht festgelegt. Wir motivieren nun die Definition von $\mathcal{V}al$ für die verschiedenen Arten von Ausdrücken. Die gesamte Definition von $\mathcal{V}al$ wird dann in Def. 2.2.7 zusammengefasst.

Die Bedeutung einer Variable $\underline{\mathsf{var}}$ liegt nur an dem verwendeten Environment, d.h. $\mathcal{V}al[\![\underline{\mathsf{var}}]\!]\,\rho\ =\ \rho(\underline{\mathsf{var}})$.

Bei einem nullstelligen Konstruktor $\underline{\mathsf{constr}}_0$ (der z.B. auch eine ganze Zahl sein kann), ist die Semantik einfach dieser Konstruktor selbst. Solche Werte entsprechen Objekten des Teildomains $(\mathsf{Con}_0)_\perp$. Sowohl in diesem (gelifteten) Domain als auch im Gesamt-Domain Dom sind die Konstruktoren entsprechend markiert, d.h., dort befindet sich stattdessen der Wert $\underline{\mathsf{constr}}_0\ in\ \mathsf{Constructions}_0\ in\ \mathsf{Dom}$. Wir definieren daher $\mathcal{V}al[\![\underline{\mathsf{constr}}_0]\!]\,\rho\ =\ \underline{\mathsf{constr}}_0\ in\ \mathsf{Constructions}_0\ in\ \mathsf{Dom}$.

Der Wert eines n-stelligen Konstruktors $\underline{\mathsf{constr}}_n$ (mit $n > 0$) ist eine Funktion f, die "Konstruktorgrundterme" erzeugt. Die Anwendung solch einer Funktion f auf n Argumente $d_1, \ldots, d_n$ ergibt das Objekt $(\underline{\mathsf{constr}}_n, d_1, \ldots, d_n)$ (bzw. $\underline{\mathsf{constr}}_n(d_1, \ldots, d_n)$ in Kurzschreibweise). Da sich dieser Wert in dem gelifteten Teildomain $\mathsf{Constructions}_n$ und sich dieser wieder im Gesamt-Domain Dom befindet, ist er hier entsprechend markiert, so dass wir stattdessen $(\underline{\mathsf{constr}}_n, d_1, \ldots, d_n)\ in\ \mathsf{Constructions}_n\ in\ \mathsf{Dom}$ schreiben. Auf diese Weise bekommt auch die partielle Anwendung eines Konstruktors den richtigen Wert zugewiesen. Ist $\underline{\mathsf{constr}}_n$ zweistellig (d.h. $n = 2$), so ist $f(d)$ eine Funktion, die jedes Argument d' auf das Objekt $(\underline{\mathsf{constr}}_n, d, d')$ abbildet.

Der Wert eines Tupels $(\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n)$ von Ausdrücken ist einfach der Tupel der Werte der Teilausdrücke $(\mathcal{V}al\,[\![\underline{\mathsf{exp}}_1]\!]\,\rho, \ldots, \mathcal{V}al\,[\![\underline{\mathsf{exp}}_n]\!]\,\rho)$ als Element von Tuples_n *in* Dom. Eine Besonderheit ist hierbei der Fall $n = 1$. Hier werden bei der Semantik die Tupelklammern einfach ignoriert, d.h., die Semantik von $(\underline{\mathsf{exp}})$ ist dieselbe wie von $\underline{\mathsf{exp}}$.

Betrachten wir nun einen Ausdruck der Form $(\underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2)$, der die Anwendung von $\underline{\mathsf{exp}}_1$ auf $\underline{\mathsf{exp}}_2$ bezeichnet. Der Wert des ersten Ausdrucks sollte eine Funktion f sein, d.h. $\mathcal{V}al\,[\![\underline{\mathsf{exp}}_1]\!]\,\rho = f$ *in* $\mathsf{Functions}$ *in* Dom. Wir haben also $f \in \langle \mathsf{Dom} \to \mathsf{Dom} \rangle$, d.h., f ist eine stetige Funktion, die Werte des Domains wieder auf Werte des Domains abbildet. Der Wert des gesamten Ausdrucks $(\underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2)$ ist dann $f(\mathcal{V}al\,[\![\underline{\mathsf{exp}}_2]\!]\,\rho)$.

Die Bedeutung eines Ausdrucks `if` $\underline{\mathsf{exp}}_1$ `then` $\underline{\mathsf{exp}}_2$ `else` $\underline{\mathsf{exp}}_3$ hängt von der Bedeutung des ersten Teilausdrucks $\underline{\mathsf{exp}}_1$ ab. Falls dieser den Wert `True` *in* $\mathsf{Constructions}_0$ *in* Dom hat, so ergibt sich der Wert von $\underline{\mathsf{exp}}_2$ und wenn er den `False` *in* $\mathsf{Constructions}_0$ *in* Dom hat, so erhält man den Wert von $\underline{\mathsf{exp}}_3$. Ansonsten (wenn er den Wert $\bot$ hat), so ergibt sich auch $\bot$ als Gesamtwert.

Als nächstes definieren wir den Wert eines Ausdrucks in einer (lokalen) Deklaration $\mathcal{V}al\,[\![\mathtt{let}\ \underline{\mathsf{var}} = \underline{\mathsf{exp}}\ \mathtt{in}\ \underline{\mathsf{exp}}']\!]\,\rho$. Dessen Wert ist offensichtlich der Wert von $\underline{\mathsf{exp}}'$, aber nicht in dem bisherigen Environment ρ, sondern in einem Environment ρ', in dem auch die lokale Variable $\underline{\mathsf{var}}$ den durch die lokale Deklaration $\underline{\mathsf{var}} = \underline{\mathsf{exp}}$ bestimmten Wert hat. Wir untersuchen als Erstes ein Beispiel, in dem es sich um eine nicht-rekursive Deklaration handelt.

```
let x = 3 in plus x y
```

Sei $\rho(\mathtt{y}) = 2$ und sei $\rho(\mathtt{plus})$ die Additionsfunktion. Dann ist $\mathcal{V}al\,[\![\mathtt{let\,x} = \mathtt{3\,in\,plus\,x\,y}]\!]\,\rho = \mathcal{V}al\,[\![\mathtt{plus\,x\,y}]\!]\rho'$, wobei sich ρ' von ρ dadurch unterscheidet, dass neben der Belegung der Variablen `y` und `plus` nun auch der Variablen `x` ein Wert zugeordnet wird (nämlich 3). Insgesamt erhält man dann den Gesamtwert 5. Im allgemeinen ergibt sich (bei nicht-rekursiven Deklarationen) das neue Environment ρ' also dadurch, dass die Variable $\underline{\mathsf{var}}$ mit dem zu $\underline{\mathsf{exp}}$ gehörenden Wert belegt wird (dabei können frühere Belegungen derselben Variablen in ρ überdeckt werden). Die noch verbleibenden in ρ definierten Variablen werden wie im bisherigen Environment ρ gedeutet. Man erhält also

$$\mathcal{V}al\,[\![\mathtt{let}\ \underline{\mathsf{var}} = \underline{\mathsf{exp}}\ \mathtt{in}\ \underline{\mathsf{exp}}']\!]\,\rho \;=\; \mathcal{V}al\,[\![\underline{\mathsf{exp}}']\!]\,(\rho + \{\underline{\mathsf{var}}\,/\,\mathcal{V}al\,[\![\underline{\mathsf{exp}}]\!]\,\rho\}).$$

Die obige Definition der Semantik von (lokalen) Deklarationen lässt sich aber nur verwenden, wenn die Variable $\underline{\mathsf{var}}$ nicht selbst wieder frei in dem Ausdruck $\underline{\mathsf{exp}}$ auftritt. Wir müssen aber auch den Fall berücksichtigen, in dem diese Deklaration rekursiv ist. Zur Illustration betrachten wir folgende Deklaration.

```
let   fact = \x -> if x <=0 then 1 else fact(x − 1) ∗ x   in   fact 2
```

Der Wert dieses Ausdrucks in einem Environment ρ ist $\mathcal{V}al\,[\![\mathtt{fact\ 2}]\!]\,\rho'$, wobei sich das neue Environment ρ' von dem bisherigen Environment ρ dadurch unterscheidet, dass der Variablen `fact` der Wert zugeordnet wird, der sich durch die Deklaration

```
fact = \x -> if x <=0 then 1 else fact(x − 1) ∗ x
```

ergibt. Wie in Abschnitt 2.1.3 erläutert, bedeuten solche rekursiven Definitionen, dass der Variablen `fact` der *kleinste Fixpunkt* der higher-order Funktion *ff* zugeordnet werden soll,

die jede Funktion d in den Wert der entsprechenden rechten Seite `\x -> if x <=0 then 1 else fact(x - 1) * x` überführt, wenn man dort die Variable `fact` durch die Funktion d ersetzt. Demnach ist ff eine Funktion von Dom nach Dom und wir haben

$$\mathit{ff}(d) = \mathcal{V}al\,[\![\backslash\texttt{x -> if x <=0 then 1 else fact}(\texttt{x} - \texttt{1}) * \texttt{x}]\!]\,(\rho + \{\texttt{fact} / d\}).$$

Wir definieren dann

$$\begin{array}{l}\mathcal{V}al\,[\![\texttt{let}\quad\texttt{fact} = \backslash\texttt{x -> if x <=0 then 1 else fact}(\texttt{x} - \texttt{1}) * \texttt{x}\quad\texttt{in}\quad\texttt{fact 2}]\!]\,\rho\ =\\ \mathcal{V}al\,[\![\texttt{fact 2}]\!]\,(\rho + \{\texttt{fact} / \mathrm{lfp}\ \mathit{ff}\}).\end{array}$$

Insgesamt erhält man also

$$\begin{array}{rcl}\mathcal{V}al\,[\![\texttt{let}\ \underline{\mathsf{var}} = \underline{\mathsf{exp}}\ \texttt{in}\ \underline{\mathsf{exp}}']\!]\,\rho & = & \mathcal{V}al\,[\![\underline{\mathsf{exp}}']\!]\,(\rho + \{\underline{\mathsf{var}} / \mathrm{lfp}\ f\}),\\ & & \text{wobei } f(d) = \mathcal{V}al\,[\![\underline{\mathsf{exp}}]\!]\,(\rho + \{\underline{\mathsf{var}}/d\})\end{array}$$

Jede Funktion f, die einem Wert d den Wert eines Ausdrucks $\underline{\mathsf{exp}}$ (unter der Belegung von $\underline{\mathsf{var}}$ mit d) zuordnet, ist *stetig*. Intuitiv ist dies einsichtig, da solche Funktionen berechenbar sind. Für einen formalen Beweis sei der Leser auf [LS87] verwiesen. Dies stellt sicher, dass der hierbei benötigte kleinste Fixpunkt tatsächlich existiert und wie im Fixpunktsatz (Satz 2.1.17) berechnet werden kann.

Der oben zuerst behandelte Spezialfall der nicht-rekursiven Deklaration ergibt sich hieraus sofort. Tritt die Variable $\underline{\mathsf{var}}$ nämlich in dem Ausdruck $\underline{\mathsf{exp}}$ nicht auf, so spielt die Belegung dieser Variablen keine Rolle bei der Auswertung dieses Ausdrucks und man erhält $f(d) = \mathcal{V}al\ [\![\underline{\mathsf{exp}}]\!]\ \rho$. Der kleinste (und einzige) Fixpunkt von f ist daher gerade $\mathcal{V}al\,[\![\underline{\mathsf{exp}}]\!]\,\rho$, womit sich

$$\mathcal{V}al\,[\![\texttt{let}\ \underline{\mathsf{var}} = \underline{\mathsf{exp}}\ \texttt{in}\ \underline{\mathsf{exp}}']\!]\,\rho\ =\ \mathcal{V}al\,[\![\underline{\mathsf{exp}}']\!]\,(\rho + \{\underline{\mathsf{var}} / \mathcal{V}al\,[\![\underline{\mathsf{exp}}]\!]\,\rho\})$$

ergibt.

Schließlich betrachten wir noch Lambda-Ausdrücke $\backslash\,\underline{\mathsf{var}}\ \texttt{->}\ \underline{\mathsf{exp}}$. Der Wert eines solchen Ausdrucks (in einem Environment ρ) ist eine Funktion $f : \mathsf{Dom} \to \mathsf{Dom}$, die Werte d abbildet auf den Wert von $\underline{\mathsf{exp}}$. Hierbei wird das Environment verwendet, das die Variable $\underline{\mathsf{var}}$ mit dem Wert d belegt und sich ansonsten wie ρ verhält. Wir definieren also

$$f(d)\ =\ \mathcal{V}al\,[\![\underline{\mathsf{exp}}]\!]\,(\rho + \{\underline{\mathsf{var}} / d\}).$$

Definition 2.2.7 (Semantik von einfachen HASKELL-Programmen) *Sei* Dom *der Domain eines einfachen* HASKELL*-Programms, sei* $\underline{\mathsf{var}} = \underline{\mathsf{exp}}$ *die Patterndeklaration dieses Programms und sei* Exp *die Menge der einfachen* HASKELL*-Ausdrücke. Wir definieren die Funktion* $\mathcal{V}al : \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Dom}$*, die jedem Ausdruck bei jedem Environment (das auf allen seinen freien Variablen definiert ist) einen Wert zuordnet. Die Semantik eines Ausdrucks* $\underline{\mathsf{exp}}'$*, der keine freien Variablen außer* $\underline{\mathsf{var}}$ *und den in* HASKELL *vordefinierten Variablen enthält, ist bei diesem Programm definiert als*

$$\mathcal{V}al\,[\![\texttt{let}\ \underline{\mathsf{var}}\ \texttt{=}\ \underline{\mathsf{exp}}\ \texttt{in}\ \underline{\mathsf{exp}}']\!]\,\omega.$$

Für ein Environment ρ ist die Funktion $\mathcal{V}al$ wie folgt definiert. Hierbei sei $\underline{\mathsf{constr}}_0 \in \mathsf{Con}_0$ *ein nullstelliger und* $\underline{\mathsf{constr}}_n \in \mathsf{Con}_n$ *ein n-stelliger Konstruktor mit $n > 0$.*

$$\mathcal{V}al\,[\![\underline{\mathsf{var}}]\!]\,\rho = \rho(\underline{\mathsf{var}})$$

$$\mathcal{V}al\,[\![\underline{\mathsf{constr}}_0]\!]\,\rho = \underline{\mathsf{constr}}_0 \textit{ in } \mathsf{Constructions}_0 \textit{ in } \mathsf{Dom}$$

$$\mathcal{V}al\,[\![\underline{\mathsf{constr}}_n]\!]\,\rho = f \textit{ in } \mathsf{Functions} \textit{ in } \mathsf{Dom}, \textit{wobei } f\,d_1\,d_2\,\ldots\,d_n = (\underline{\mathsf{constr}}_n, d_1, \ldots, d_n) \textit{ in } \mathsf{Constructions}_n \textit{ in } \mathsf{Dom}$$

$$\mathcal{V}al\,[\![(\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n)]\!]\,\rho = (\,\mathcal{V}al\,[\![\underline{\mathsf{exp}}_1]\!]\,\rho, \ldots, \mathcal{V}al\,[\![\underline{\mathsf{exp}}_n]\!]\,\rho\,) \textit{ in } \mathsf{Tuples}_n \textit{ in } \mathsf{Dom}, \textit{ wobei } n = 0 \textit{ oder } n \geq 2$$

$$\mathcal{V}al\,[\![(\underline{\mathsf{exp}})]\!]\,\rho = \mathcal{V}al\,[\![\underline{\mathsf{exp}}]\!]\,\rho$$

$$\mathcal{V}al\,[\![(\underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2)]\!]\,\rho = f(\,\mathcal{V}al\,[\![\underline{\mathsf{exp}}_2]\!]\,\rho\,), \textit{ wobei } \mathcal{V}al\,[\![\underline{\mathsf{exp}}_1]\!]\,\rho = f \textit{ in } \mathsf{Functions} \textit{ in } \mathsf{Dom}$$

$$\mathcal{V}al\left[\!\!\left[\begin{array}{l}\texttt{if } \underline{\mathsf{exp}}_1 \texttt{ then } \underline{\mathsf{exp}}_2\\ \qquad\quad\ \texttt{else } \underline{\mathsf{exp}}_3\end{array}\right]\!\!\right]\rho = \begin{cases} \mathcal{V}al\,[\![\underline{\mathsf{exp}}_2]\!]\,\rho, & \textit{falls } \mathcal{V}al\,[\![\underline{\mathsf{exp}}_1]\!]\,\rho = \texttt{True} \textit{ in } \mathsf{Constructions}_0 \textit{ in } \mathsf{Dom}\\ \mathcal{V}al\,[\![\underline{\mathsf{exp}}_3]\!]\,\rho, & \textit{falls } \mathcal{V}al\,[\![\underline{\mathsf{exp}}_1]\!]\,\rho = \texttt{False} \textit{ in } \mathsf{Constructions}_0 \textit{ in } \mathsf{Dom}\\ \bot, & \textit{sonst}\end{cases}$$

$$\mathcal{V}al\left[\!\!\left[\begin{array}{l}\texttt{let } \underline{\mathsf{var}} = \underline{\mathsf{exp}}\\ \quad\texttt{in } \underline{\mathsf{exp}'}\end{array}\right]\!\!\right]\rho = \mathcal{V}al\,[\![\underline{\mathsf{exp}'}]\!]\,(\rho + \{\underline{\mathsf{var}}\,/\,\mathrm{lfp}\ f\}), \textit{ wobei } f : \mathsf{Dom} \to \mathsf{Dom} \textit{ mit } f(d) = \mathcal{V}al\,[\![\underline{\mathsf{exp}}]\!]\,(\rho + \{\underline{\mathsf{var}}/d\})$$

$$\mathcal{V}al\,[\![\backslash\ \underline{\mathsf{var}}\ \texttt{->}\ \underline{\mathsf{exp}}]\!]\,\rho = f \textit{ in } \mathsf{Functions} \textit{ in } \mathsf{Dom}, \textit{ wobei } f(d) = \mathcal{V}al\,[\![\underline{\mathsf{exp}}]\!]\,(\rho + \{\underline{\mathsf{var}}\,/\,d\})$$

Selbstverständlich muss bei dieser Definition sichergestellt werden, dass auf diese Weise tatsächlich allen Ausdrücken exp nur Elemente von Dom zugeordnet werden (d.h., dass alle hierbei entstehenden Funktionen stetig sind). Hierzu sei der Leser auf [LS87] und die Literatur zur denotationellen Semantik (wie z.B. [Sto81]) verwiesen.

2.2.3 Semantik komplexer HASKELL-Programme

Im Folgenden zeigen wir, wie komplexere HASKELL-Programme (mit Pattern Matching) automatisch in einfache Programme nach Def. 2.2.5 übersetzt werden können. Da diese Übersetzung automatisch durchgeführt werden kann, lässt sie sich auch für die Typüberprüfung und Implementierung funktionaler Sprachen einzusetzen. Dann kann man die nun folgende Transformation nämlich jeweils als ersten Schritt durchführen und muss dann nur noch angeben, wie man einfache HASKELL-Programme auf Typkorrektheit überprüft und implementiert. Durch die Angabe der Überführungstechnik legen wir aber insbesondere auch die Semantik von komplexen HASKELL-Programmen fest.

Bei der Übersetzung von komplexen Programmen in einfache müssen wir jedoch einige neue vordefinierte Funktionen voraussetzen (zusätzlich zu den bereits vorhandenen Funktionen wie `+`, `*`, `==`, etc.), die in den entstehenden einfachen HASKELL-Programmen auftreten können. Insbesondere benötigen wir eine Funktion `bot`, die überall undefiniert ist. Sie ließe sich z.B. durch das folgende Programm definieren.

```
bot :: a
bot = bot
```

Die Hauptaufgabe bei der Übersetzung in einfache HASKELL-Programme ist die Entfernung des Pattern Matching. Hierzu dienen die nächsten Funktionen. Für jeden n-stelligen Konstruktor $\underline{\mathsf{constr}}$ (wobei $n \geq 0$ ist), dient die Funktion $\mathtt{isa}_{\underline{\mathsf{constr}}}$ dazu, zu erkennen, ob ihr Argument mit diesem Konstruktor gebildet wurde. Diese Funktion lässt sich (bei $n > 0$) nur mit Hilfe von Pattern Matching definieren, d.h., sie kann nicht direkt als einfaches HASKELL-Programm geschrieben werden. Dies ist der Grund, warum wir diese Funktion als vordefiniert voraussetzen müssen. Für den Listenkonstruktor `Cons` könnte man (in HASKELL mit Pattern Matching) diese Funktion wie folgt definieren:

```
isa_Cons :: (List a) -> Bool
isa_Cons (Cons x y) = True
isa_Cons Nil        = False
```

Ähnlich wie die Funktion $\mathtt{isa}_{\underline{\mathsf{constr}}}$ dient die Funktion $\mathtt{argof}_{\underline{\mathsf{constr}}}$ dazu, den Tupel der Argumente des Arguments zurückzuliefern, sofern dieses mit dem Konstruktor $\underline{\mathsf{constr}}$ gebildet wurde. Auch diese Funktion ließe sich in HASKELL (bei nicht-nullstelligen Konstruktoren) nur mit Hilfe von Pattern Matching definieren.

```
argof_Cons :: (List a) -> (a, (List a))
argof_Cons (Cons x y) = (x, y)
```

Ebenso wie wir `isa`-Funktionen benötigen, die erkennen, ob ein Argument mit einem bestimmten Datenkonstruktor gebildet wurde, benötigen wir auch eine entsprechende Funktion $\mathtt{isa}_{n-\mathtt{tuple}}$, die erkennt, ob ihr Argument ein Tupel mit n Komponenten ist. Sie lässt sich wie folgt definieren:

```
isa_n_tuple  :: (a1,...,an) -> Bool
isa_n_tuple (x1,...,xn) = True
```

In einem typkorrekten Programm liefert die Auswertung von "$\mathtt{isa}_{n-\mathtt{tuple}}$ $\underline{\mathsf{exp}}$" stets `True`, außer wenn $\underline{\mathsf{exp}}$ die Semantik $\perp$ hat.

Schließlich benötigen wir auch Funktionen $\mathtt{sel}_{n,i}$, die auf das i-te Argument eines n-stelligen Tupels zugreifen (für $n \geq 2$ und alle $1 \leq i \leq n$). Solche Funktionen lassen sich ebenfalls nur dann programmieren, wenn man (Tupel-)Patterns verwendet. Beispielsweise ließe sich die Funktion $\mathtt{sel}_{3,2}$ wie folgt in (nicht-einfachem) HASKELL formulieren:

```
sel_32  :: (a1, a2, a3) -> a2
sel_32 (x1, x2, x3) = x2
```

Da wir diese Funktionen als vordefiniert betrachten (d.h. da für sie keine Deklaration in einem HASKELL-Programm angegeben wird), muss das initiale Environment ω um diese Funktionen erweitert werden. So entsteht ein neues initiales Environment ω_{tr} (hierbei steht "*tr*" für "Transformation", da dieses Environment benötigt wird, um mit den Programmen umzugehen, die durch unsere Transformation in einfache HASKELL-Programme entstehen).

Definition 2.2.8 (Vordefinierte Funktionen) *Zu einem* HASKELL*-Programm mit den Konstruktoren* Con_n $(n \geq 0)$ *seien die folgenden Funktionen vordefiniert. Hierbei sei* k *das Maximum der Stelligkeit der Funktionen des Programms, der Länge der Tupel des Programms und der Anzahl der Deklarationen des Programms.*

- $\texttt{bot} :: \texttt{a}$
- $\texttt{isa}_{\underline{\mathsf{constr}}} :: \underline{\mathsf{type}} \texttt{ -> Bool}$
  für alle $\underline{\mathsf{constr}} \in \mathsf{Con}_n$, *wobei* $\underline{\mathsf{constr}} :: \underline{\mathsf{type}}_1 \texttt{ -> } \ldots \texttt{ -> } \underline{\mathsf{type}}_n \texttt{ -> } \underline{\mathsf{type}}$
- $\texttt{argof}_{\underline{\mathsf{constr}}} :: \underline{\mathsf{type}} \texttt{ -> } (\underline{\mathsf{type}}_1, \ldots, \underline{\mathsf{type}}_n)$
  für alle $\underline{\mathsf{constr}} \in \mathsf{Con}_n$, *wobei* $\underline{\mathsf{constr}} :: \underline{\mathsf{type}}_1 \texttt{ -> } \ldots \texttt{ -> } \underline{\mathsf{type}}_n \texttt{ -> } \underline{\mathsf{type}}$
- $\texttt{isa}_{n-\texttt{tuple}} :: (\mathsf{a}_1, \ldots, \mathsf{a}_n) \texttt{ -> Bool}$
  für alle $n \in \{0, 2, 3, \ldots, k\}$
- $\texttt{sel}_{n,i} :: (\mathsf{a}_1, \ldots, \mathsf{a}_n) \texttt{ -> } \mathsf{a}_i$
  für alle $2 \leq n \leq k, 1 \leq i \leq n$.

Das initiale Environment ω *wird zu einem neuen initialen Environment* ω_{tr} *erweitert, in dem die Semantik der neuen vordefinierten Funktionen wie folgt gegeben ist. Hierbei stehen Tupel mit einer Komponente wieder für die Komponente selbst.*

$$\omega_{tr}(\texttt{bot}) = \bot$$

$$\omega_{tr}(\texttt{isa}_{\underline{\mathsf{constr}}})(d) = \begin{cases} \texttt{True} \textit{ in } \mathsf{Constructions}_0 \textit{ in } \mathsf{Dom}, \\ \quad \textit{falls } d = (\underline{\mathsf{constr}}, d_1, \ldots, d_n) \textit{ in } \mathsf{Constructions}_n \textit{ in } \mathsf{Dom} \\ \texttt{False} \textit{ in } \mathsf{Constructions}_0 \textit{ in } \mathsf{Dom}, \\ \quad \textit{falls } d = (\underline{\mathsf{constr}}', d_1, \ldots, d_m) \textit{ in } \mathsf{Constructions}_m \textit{ in } \mathsf{Dom} \\ \quad \textit{und } \underline{\mathsf{constr}} \neq \underline{\mathsf{constr}}' \\ \bot, \quad \textit{sonst} \end{cases}$$

$$\omega_{tr}(\texttt{argof}_{\underline{\mathsf{constr}}})(d) = \begin{cases} (d_1, \ldots, d_n) \textit{ in } \mathsf{Tuples}_n \textit{ in } \mathsf{Dom}, \\ \quad \textit{falls } d = (\underline{\mathsf{constr}}, d_1, \ldots, d_n) \textit{ in } \mathsf{Constructions}_n \textit{ in } \mathsf{Dom},\ n \neq 1 \\ d_1, \textit{falls } d = (\underline{\mathsf{constr}}, d_1) \textit{ in } \mathsf{Constructions}_1 \textit{ in } \mathsf{Dom} \\ \bot, \quad \textit{sonst} \end{cases}$$

$$\omega_{tr}(\texttt{isa}_{n-\texttt{tuple}})(d) = \begin{cases} \texttt{True} \textit{ in } \mathsf{Constructions}_0 \textit{ in } \mathsf{Dom}, \\ \quad \textit{falls } d = (d_1, \ldots, d_n) \textit{ in } \mathsf{Tuples}_n \textit{ in } \mathsf{Dom} \\ \bot, \quad \textit{sonst} \end{cases}$$

$$\omega_{tr}(\texttt{sel}_{n,i})(d) = \begin{cases} d_i, \quad \textit{falls } d = (d_1, \ldots, d_n) \textit{ in } \mathsf{Tuples}_n \textit{ in } \mathsf{Dom} \\ \bot, \quad \textit{sonst} \end{cases}$$

Wir zeigen nun, wie man (fast) beliebige HASKELL-Programme in einfache Programme überführen kann. Die einzigen Einschränkungen an das Ausgangsprogramm sind, dass es

keine Infixdeklarationen enthalten darf (Infix-Symbole müssen vorher durch entsprechende Präfixsymbole ersetzt werden), dass nur Patterndeklarationen mit Variablen auf den linken Seiten zugelassen sind, dass die in HASKELL vordefinierten Listen vom Benutzer selbst definiert werden müssen (mit `List`), dass wir nur lokale Definitionen mit `let` verwenden (`where` muss vorher in `let` umcodiert werden), dass wir keine bedingten rechten Seiten verwenden (diese müssen stattdessen in `if` oder `case`-Konstrukte umcodiert werden), dass wir keine Patterns mit @ benutzen und dass wir weder Typabkürzungen noch Typklassen zulassen. Entsprechende Erweiterungen der Übersetzung auf "volles" HASKELL sind aber möglich.

Definition 2.2.9 (Komplexe HASKELL-Programme) *Ein komplexes HASKELL-Programm ist ein Programm ohne Typabkürzungen, Typklassen, Infixdeklarationen und ohne die in HASKELL vordefinierten Listen. Außerdem müssen definierende Gleichungen für eine Funktion nebeneinander stehen und das Programm darf nur solche Patterndeklarationen enthalten, bei denen eine Variable auf der linken Seite steht. Weiterhin darf das Programm kein* `where`*, keine bedingten rechten Seiten und keine Patterns mit @ enthalten.*

Die Überführung von komplexen HASKELL-Programmen in einfache Programme arbeitet anhand von 12 Regeln, die in beliebiger Reihenfolge so lange wie möglich angewendet werden. Die benutzerdefinierten algebraischen Datenstrukturen legen die Menge der Konstruktoren Con fest und sind insofern wichtig, um zu bestimmen, wie die vordefinierten Funktionen für dieses Programm gebildet werden. Typdeklarationen var :: type werden bei der Semantik ignoriert - sie spielen nur bei der Typüberprüfung eine Rolle, vgl. Def. 4.3.3.

Die Eingabe des Transformationsverfahrens ist ein HASKELL-Ausdruck eines komplexen Programms. Er kann also eine Folge von (lokalen) Funktions- und Patterndeklarationen besitzen. Die Ausgabe des Verfahrens ist ein einfacher HASKELL-Ausdruck nach Def. 2.2.5. Lokale Deklarationsblöcke haben darin also nur noch eine einzige Deklaration der Form var = exp und der Ausdruck exp verwendet kein Pattern Matching und nur Lambda-Ausdrücke der Form \ var -> exp. Wir werden nun die einzelnen Regeln des Übersetzungsverfahrens motivieren und vorstellen. Die Regeln sind dann in Def. 2.2.11 zusammengefasst.

Die erste Regel überführt eine Folge von definierenden Gleichungen für eine Funktion var (d.h. eine Folge von Funktionsdeklarationen) in eine Patterndeklaration. Wie in Def. 2.2.9 gehen wir davon aus, dass definierende Gleichungen für eine Funktion nebeneinander stehen (ansonsten muss das Programm vorher umsortiert werden). Ein Programm wie

```
append Nil z = z
append (Cons x y) z = Cons x (append y z)
```

wird also überführt in

```
append = \x1 x2 -> case (x1, x2) of (Nil, z) -> z
                                    (Cons x y, z) -> Cons x (append y z)
```

Mit dieser ersten Regel lassen sich alle Funktionsdeklarationen aus dem Programm in Patterndeklarationen der Form var = exp überführen.

Wir betrachten zunächst nur den Fall einer einzigen Patterndeklaration. Dann müssen nun nur noch die Ausdrücke exp in die bei einfachen HASKELL-Programmen erlaubten

Formen gebracht werden. Dort sind nur Lambda-Ausdrücke mit einem Argument (und zwar einer Variablen) möglich. Daher werden zunächst Ausdrücke wie \ $\underline{\mathtt{pat}}_1 \ldots \underline{\mathtt{pat}}_n$ `->` $\underline{\mathtt{exp}}$ durch die dazu äquivalenten geschachtelten Lambda-Ausdrücke \ $\underline{\mathtt{pat}}_1$ `->` (\ $\underline{\mathtt{pat}}_2$ `->` ... (\ $\underline{\mathtt{pat}}_n$ `->` $\underline{\mathtt{exp}}$) ...) ersetzt. In unserem Beispiel `append` erhält man also

```
append = \x1 ->
         (\x2 -> case (x1, x2) of (Nil, z) -> z
                                  (Cons x y, z) -> Cons x (append y z))
```

Schließlich werden Lambda-Ausdrücke mit Nicht-Variablen Patterns \ $\underline{\mathtt{pat}}$ `->` $\underline{\mathtt{exp}}$ in entsprechende `case`-Ausdrücke \ $\underline{\mathtt{var}}$ `-> case` $\underline{\mathtt{var}}$ `of` $\underline{\mathtt{pat}}$ `->` $\underline{\mathtt{exp}}$ überführt. Dieser `case`-Ausdruck wird nur dann fehlerfrei ausgewertet, wenn $\underline{\mathtt{var}}$ mit einem Ausdruck belegt ist, auf den der Pattern $\underline{\mathtt{pat}}$ passt. Das Ergebnis ist dann $\underline{\mathtt{exp}}$, wobei die Variablen entsprechend der Patternbindung belegt sind. Auf diese Weise haben wir nun nur noch Lambda-Ausdrücke der Form \ $\underline{\mathtt{var}}$ `->` $\underline{\mathtt{exp}}$, wie sie in einfachen HASKELL-Programmen erlaubt sind.

Alle Pattern Matching Probleme sind nun in `case`-Ausdrücke überführt worden. Die nächsten Regeln dienen dazu, das Pattern Matching zu entfernen. Hierzu werden zunächst `case`-Ausdrücke in `match`-Ausdrücke überführt. Hierbei liefert

$$\mathtt{match}\ \underline{\mathtt{pat}}\ \underline{\mathtt{exp}}\ \underline{\mathtt{exp}}_1\ \underline{\mathtt{exp}}_2$$

folgendes: Falls der Pattern $\underline{\mathtt{pat}}$ den Ausdruck $\underline{\mathtt{exp}}$ matcht, so ist das Ergebnis $\underline{\mathtt{exp}}_1$, wobei die Variablen durch den Match von $\underline{\mathtt{pat}}$ auf $\underline{\mathtt{exp}}$ gebunden werden. Ansonsten ist das Ergebnis $\underline{\mathtt{exp}}_2$. Das neue Hilfskonstrukt `match` wird nur während der Transformation benötigt. Das zum Schluss resultierende Programm enthält also zwar die in Def. 2.2.8 eingeführten Funktionen, aber nicht `match`. Der Grund für die Verwendung von `match` statt `case` ist, dass sich damit die Transformationsregeln leichter formulieren lassen. In unserem Beispiel erhalten wir

```
append = \x1 ->
         (\x2 -> match (Nil, z) (x1, x2) z
                   (match (Cons x y, z) (x1, x2) (Cons x (append y z)) bot))
```

Die folgenden Regeln dienen dazu, die Hilfsfunktion `match` zu eliminieren. Soll ein nichtleerer Tupel-Pattern wie `(Nil, z)` auf einen Ausdruck `(x1, x2)` gematcht werden, so muss zunächst mit Hilfe der Funktion $\mathtt{isa}_{2-\mathtt{tuple}}$ überprüft werden, ob `(x1, x2)` ein Tupel mit zwei Komponenten ist. In unserem Beispiel ist dies natürlich offensichtlich der Fall. Das Matching des Tupelpatterns `(Nil, z)` auf `(x1, x2)` lässt sich nun umformen, indem zuerst die erste Komponente `Nil` auf die erste Komponente $\mathtt{sel}_{2,1}(\mathtt{x1}, \mathtt{x2})$ des Ausdrucks gematcht wird. Gelingt dies, so werden die hierbei gewonnen Bindungen verwendet, um die zweite Komponente des Patterns auf die zweite Komponente des Ausdrucks zu matchen. Diese Umformung wird durch Regel (9) realisiert. Der Lesbarkeit halber ersetzen wir Ausdrücke der Form "$\mathtt{if}\ \mathtt{isa}_{n-\mathtt{tuple}}(\underline{\mathtt{exp}}_1, \ldots, \underline{\mathtt{exp}}_n)\ \mathtt{then}\ \mathtt{exp}'\ \mathtt{else}\ \mathtt{exp}''$" durch "$\mathtt{exp}'$". Man erhält in unserem Beispiel

```
append = \x1 ->
         (\x2 ->
```

```
            match Nil
                  (sel_{2,1} (x1, x2))
                  (match z
                         (sel_{2,2} (x1, x2))
                         z
                         (match (Cons x y)
                                (sel_{2,1} (x1, x2))
                                (match z
                                       (sel_{2,2} (x1, x2))
                                       (Cons x (append y z))
                                       bot)
                                bot))
                  (match (Cons x y)
                         (sel_{2,1} (x1, x2))
                         (match z
                                (sel_{2,2} (x1, x2))
                                (Cons x (append y z))
                                bot)
                         bot))
```

Um die Lesbarkeit zu erhöhen, ersetzen wir im Folgenden `(sel`$_{2,1}$ `(x1, x2))` durch `x1` und `(sel`$_{2,2}$ `(x1, x2))` durch `x2`. So erhält man

```
append = \x1 ->
         (\x2 ->
            match Nil
                  x1
                  (match z
                         x2
                         z
                         (match (Cons x y)
                                x1
                                (match z
                                       x2
                                       (Cons x (append y z))
                                       bot)
                                bot))
                  (match (Cons x y)
                         x1
                         (match z
                                x2
                                (Cons x (append y z))
                                bot)
                         bot))
```

Das Matchen einer Variable auf einen Ausdruck gelingt stets. Daher können Ausdrücke der Form `match` $\underline{\texttt{var}}$ $\underline{\texttt{exp}}$ $\underline{\texttt{exp}}_1$ $\underline{\texttt{exp}}_2$ durch (\ $\underline{\texttt{var}}$ `->` $\underline{\texttt{exp}}_1$) $\underline{\texttt{exp}}$ ersetzt werden (Regel (5)).

Dies bedeutet, dass sich als Ergebnis von match $\underline{\text{var}}$ $\underline{\text{exp}}$ $\underline{\text{exp}}_1$ $\underline{\text{exp}}_2$ der Ausdruck $\underline{\text{exp}}_1$ ergibt, in dem aber alle Vorkommen von $\underline{\text{var}}$ durch den Ausdruck $\underline{\text{exp}}$ ersetzt werden. Im Beispiel kann man also z.B.

```
(match z x2 (Cons x (append y z)) bot)
```

durch

```
(\z -> (Cons x (append y z))) x2
```

ersetzen. So erhält man in unserem Beispiel

```
append = \x1 ->
         (\x2 ->
            match Nil
                  x1
                  (\z -> z) x2
                  (match (Cons x y)
                         x1
                         (\z -> (Cons x (append y z))) x2
                         bot))
```

Um die Lesbarkeit zu vereinfachen, ersetzen wir im Folgenden `(\z -> z) x2` durch `x2` und `(\z -> (Cons x (append y z))) x2` durch `(Cons x (append y x2))`. Man erhält also

```
append = \x1 ->
         (\x2 ->
            match Nil x1 x2
                  (match (Cons x y) x1 (Cons x (append y x2)) bot))
```

Ein Match mit dem Joker-Pattern _ würde analog zum Matchen mit einer Variablen verlaufen, nur gäbe es hierbei keine beim Matchen entstandenen Variablenbindungen.

Betrachten wir nun die Behandlung von Konstruktorpatterns, d.h.

$$\texttt{match}\ (\underline{\text{constr}}\ \underline{\text{pat}}_1\ \ldots\ \underline{\text{pat}}_n)\ \underline{\text{exp}}\ \underline{\text{exp}}_1\ \underline{\text{exp}}_2.$$

Wenn der Wert von $\underline{\text{exp}}$ mit dem Konstruktor $\underline{\text{constr}}$ gebildet wurde, dann sollte man die Argument-Pattern $(\underline{\text{pat}}_1, \ldots, \underline{\text{pat}}_n)$ mit den dazugehörigen Argumenten in $\underline{\text{exp}}$ matchen. Gelingt dies, so ergibt sich das Resultat $\underline{\text{exp}}_1$ mit den entsprechenden Variablenbindungen. Ansonsten erhält man das Ergebnis $\underline{\text{exp}}_2$. Unter Verwendung unserer vordefinierten Funktionen $\texttt{isa}_{\underline{\text{constr}}}$ und $\texttt{argof}_{\underline{\text{constr}}}$ kann man also solche Ausdrücke übersetzen in

$$\texttt{if}\ (\texttt{isa}_{\underline{\text{constr}}}\ \underline{\text{exp}})\ \texttt{then}\ (\texttt{match}\ (\underline{\text{pat}}_1, \ldots, \underline{\text{pat}}_n)\ (\texttt{argof}_{\underline{\text{constr}}}\ \underline{\text{exp}})\ \underline{\text{exp}}_1\ \underline{\text{exp}}_2)\ \texttt{else}\ \underline{\text{exp}}_2$$

In unserem Beispiel erhalten wir

```
append = \x1 ->
         (\x2 ->
            if   (isa_Nil x1)
```

```
          then (match () (argofNil x1) x2
                         (if   (isaCons x1)
                          then match (x, y)
                                     (argofCons x1)
                                     (Cons x (append y x2))
                                     bot)
                          else bot))
          else (if   (isaCons x1)
                then (match (x, y) (argofCons x1) (Cons x (append y x2)) bot)
                else bot))
```

Einen Match mit dem leeren Tupel match ( ) $\underline{\text{exp}}$ $\underline{\text{exp}}_1$ $\underline{\text{exp}}_2$ kann man in den Ausdruck if ($\texttt{isa}_{0-\texttt{tuple}}$ $\underline{\text{exp}}$) then $\underline{\text{exp}}_1$ else $\underline{\text{exp}}_2$ übersetzen. Wir können also

```
match () (argofNil x1) x2 ...
```

in den folgenden Ausdruck übersetzen:

```
if (isa0-tuple (argofNil x1)) then x2 else ...
```

Zur Vereinfachung der Lesbarkeit ersetzen wir diesen Ausdruck im Folgenden durch x2, da in unserem Beispielprogramm durch das voranstehende "if ($\texttt{isa}_{\texttt{Nil}}$ x1)" sicher gestellt ist, dass x1 den Wert Nil hat und $\texttt{argof}_{\texttt{Nil}}$ x1 daher zu () auswertet. Durch Auflösen des Matchs mit dem anderen Tupel (x, y) ergibt sich schließlich

```
append = \x1 ->
          (\x2 ->
             if   (isaNil x1)
             then x2
             else (if   (isaCons x1)
                   then (Cons (sel2,1 (argofCons x1))
                              (append (sel2,2 (argofCons x1)) x2))
                   else bot))
```

Dies ist schließlich ein einfaches HASKELL-Programm und in der Tat ist nun keine weitere der Transformationsregeln anwendbar.

Schließlich betrachten wir auch noch den Fall einer (lokalen) Deklaration eines Tupels von Variablen.

$$\texttt{let}\ (\underline{\text{var}}_1, \ldots, \underline{\text{var}}_n) = (\underline{\text{exp}}_1, \ldots, \underline{\text{exp}}_n)\ \texttt{in}\ \underline{\text{exp}}$$

Auf den ersten Blick könnte man vermuten, dass man solche Deklarationen immer in mehrere verschachtelte Deklarationen

$$\texttt{let}\ \underline{\text{var}}_1 = \underline{\text{exp}}_1\ \texttt{in}\ \texttt{let}\ \underline{\text{var}}_2 = \underline{\text{exp}}_2\ \texttt{in}\ \ldots\ \texttt{let}\ \underline{\text{var}}_n = \underline{\text{exp}}_n\ \texttt{in}\ \underline{\text{exp}}$$

überführen könnte. Solch eine Überführung wird mit Regel (10) durchgeführt. Dies gelingt jedoch bei verschränkt rekursiven Deklarationen nicht. Als Beispiel betrachten wir den folgenden Ausdruck:

```
let  even' = \x -> if x ==0 then True else odd'(x − 1)
     odd'  = \x -> if x ==0 then False else even'(x − 1)
in   even' 4
```

Hier würde eine Umformung in zwei geschachtelte `let`-Ausdrücke zu einem undefinierten Ausdruck führen, da in der äußeren Deklaration bereits der Wert von `odd'` benötigt wird, der jedoch erst in der inneren Deklaration definiert wird. Das Problem ist also, dass `even'` und `odd'` verschränkt rekursiv definiert sind. Um auch solche Deklarationen behandeln zu können, muss man die simultane Definition eines Tupels von Variablen betrachten.

Regel (11) dient daher dazu, eine Folge von solchen Patterndeklarationen $\{\underline{\mathsf{var}}_1 = \underline{\mathsf{exp}}_1; \dots; \underline{\mathsf{var}}_n = \underline{\mathsf{exp}}_n\}$ durch eine einzige Patterndeklaration $(\underline{\mathsf{var}}_1, \dots, \underline{\mathsf{var}}_n) = (\underline{\mathsf{exp}}_1, \dots, \underline{\mathsf{exp}}_n)$ eines Tupels von Variablen zu ersetzen. (Die weitere Überführung einer solchen Deklaration wird später in Regel (12) behandelt.) Man erhält daher

```
let (even', odd') = (\x -> if x ==0 then True else odd'(x − 1),
                     \x -> if x ==0 then False else even'(x − 1))   in   even' 4
```

Das obige Beispiel macht deutlich, dass Regel (11) zur gleichzeitigen Deklaration von Variablen immer dann angewendet werden muss, wenn die Deklarationen *verschränkt rekursiv* sind, d.h., wenn sie gegenseitig *voneinander abhängen*. Die folgende Definition führt diesen Begriff formal ein. Hierbei besagt $\underline{\mathsf{var}}' \gtrsim_P \underline{\mathsf{var}}$, dass $\underline{\mathsf{var}}'$ von $\underline{\mathsf{var}}$ abhängt und $\underline{\mathsf{var}}' \sim_P \underline{\mathsf{var}}$ bedeutet, dass $\underline{\mathsf{var}}'$ und $\underline{\mathsf{var}}$ verschränkt rekursiv (oder identisch) sind.

Um eine Transformation in einfaches HASKELL anzugeben, muss man untersuchen, welche Variablen voneinander abhängig sind und anschließend die Deklarationen danach *aufteilen*. Eine Aufteilung einer Deklarationsfolge P ist eine Folge von Deklarationsfolgen $P_1, \dots, P_k$, wobei in jedem P_i nur miteinander verschränkt rekursive Variablen deklariert werden und die Variablen, die in P_i deklariert werden, nur von Variablen abhängen, die bereits in $P_{i'}$ mit $i \geq i'$ deklariert wurden.

Definition 2.2.10 (Abhängigkeit von Variablen, Aufteilung von Deklarationen) *Sei* $P = \{\underline{\mathsf{var}}_1 = \underline{\mathsf{exp}}_1; \dots; \underline{\mathsf{var}}_n = \underline{\mathsf{exp}}_n\}$ *eine Folge von Deklarationen eines komplexen* HASKELL-*Programms. Wir definieren* $\underline{\mathsf{var}}_i \gtrsim_P \underline{\mathsf{var}}$ *("$\underline{\mathsf{var}}_i$ hängt von $\underline{\mathsf{var}}$ ab"), falls* $\underline{\mathsf{var}}_i = \underline{\mathsf{var}}$ *oder falls eine Variable* $\underline{\mathsf{var}}'$ *mit* $\underline{\mathsf{var}}' \gtrsim_P \underline{\mathsf{var}}$ *frei in* $\underline{\mathsf{exp}}_i$ *auftritt. Außerdem sei* $\underline{\mathsf{var}}_i \sim_P \underline{\mathsf{var}}$ *falls* $\underline{\mathsf{var}}_i \gtrsim_P \underline{\mathsf{var}}$ *und* $\underline{\mathsf{var}} \gtrsim_P \underline{\mathsf{var}}_i$. *Wir definieren auch* $\underline{\mathsf{var}}_i \succ_P \underline{\mathsf{var}}$ *falls* $\underline{\mathsf{var}}_i \gtrsim_P \underline{\mathsf{var}}$ *und* $\underline{\mathsf{var}} \not\gtrsim_P \underline{\mathsf{var}}_i$.

Zu P *ist* $P_1, \dots, P_k$ *mit* $P_i = \{\underline{\mathsf{var}}_{i,1} = \underline{\mathsf{exp}}_{i,1}, \dots, \underline{\mathsf{var}}_{i,n_i} = \underline{\mathsf{exp}}_{i,n_i}\}$ *eine* Aufteilung *(engl. separation) der Deklaration, falls*

- $P_1 \uplus \dots \uplus P_k = P$
- $P_i \neq \varnothing$ *für alle* $1 \leq i \leq k$
- $\underline{\mathsf{var}}_{i,1} \sim_P \dots \sim_P \underline{\mathsf{var}}_{i,n_i}$ *für alle* $1 \leq i \leq k$
- *falls* $\underline{\mathsf{var}}_{i,j} \gtrsim_P \underline{\mathsf{var}}_{i',j'}$, *dann* $i \geq i'$

Als Beispiel betrachten wir ein Programm mit drei Deklarationen für `a`, `f` und `g` bzw. den folgenden komplexen HASKELL-Ausdruck.

```
let a = 3
    f = \x -> g a
    g = \x -> f a
in  f (g a)
```

Wenn P die Folge der Deklarationen von `a`, `f` und `g` in unserem Beispiel ist, so gilt $\mathtt{f} \succ_P \mathtt{a}$, $\mathtt{g} \succ_P \mathtt{a}$ und $\mathtt{f} \sim_P \mathtt{g}$. Die einzige Aufteilung von P ist $P_1 = \{\mathtt{a} = \mathtt{3}\}$, $P_2 = \{\mathtt{f} = \backslash\mathtt{x} \to \mathtt{g\ a},\ \mathtt{g} = \backslash\mathtt{x} \to \mathtt{f\ a}\}$. Die Deklarationsfolge von `a`, `f` und `g` sollte dann zunächst mit Hilfe von Regel (10) in geschachtelte `let`-Ausdrücke überführt werden, bei denen nur noch diejenigen Variablen gemeinsam in einer Deklarationsfolge deklariert werden, die auch tatsächlich verschränkt rekursiv sind. Mit anderen Worten, man muss eine *Aufteilung* P_1, P_2 der Deklarationsfolge verwenden, bei der P_1 die Deklarationsfolge des äußersten `let`-Ausdrucks ist und P_2 im innersten `let`-Ausdruck verwendet wird. So ergibt sich

```
let a = 3
in  let f = \x -> g a
        g = \x -> f a
    in  f (g a)
```

Anschließend werden die Deklarationen von verschränkt rekursiven Variablen durch Regel (11) zu einer Tupeldeklaration zusammengefasst.

```
let a = 3
in  let (f,g) = (\x -> g a,
                 \x -> f a)
    in  f (g a)
```

Auf diese Weise kann man sicherstellen, dass die Semantik und die Typkorrektheit der HASKELL-Programme bei der Transformation erhalten bleibt.[2]

Im Beispiel mit `even'` und `odd'` hat die Folge P der beiden Deklarationen von `even'` und `odd'` nur P selbst als Aufteilung. Wie oben skizziert, wird hier also direkt Regel (11) angewandt.

$$\mathtt{let}\ (\mathtt{even}', \mathtt{odd}') = (\backslash\mathtt{x}\ \texttt{->}\ \mathtt{if\ x == 0\ then\ True\ else\ odd}'(\mathtt{x} - 1),\\ \backslash\mathtt{x}\ \texttt{->}\ \mathtt{if\ x == 0\ then\ False\ else\ even}'(\mathtt{x} - 1))\quad \mathtt{in}\quad \mathtt{even}'\ 4$$

Um solche Ausdrücke in einfaches HASKELL zu transformieren, muss man die Tupeldeklaration durch die Deklaration einer einzigen Variable ersetzen. In dieser Umformung benutzt man wieder das Hilfskonstrukt `match`. Statt eines Tupels von Variablen $(\mathtt{even}', \mathtt{odd}')$ verwenden wir eine neue Variable `eo`, deren Wert dann ein Tupel ist. Die erste Komponente dieses Tupels entspricht dem Wert von $\mathtt{even}'$ und die zweite Komponente entspricht dem Wert von $\mathtt{odd}'$. Anstelle der Deklaration

$$(\mathtt{even}', \mathtt{odd}') = \underline{\mathsf{exp}}$$

[2] Selbstverständlich könnte man auf Regel (10) verzichten und ausschließlich Regel (11) verwenden, um mehrere Variablendeklarationen zu einer Tupeldeklaration zusammenzufassen. Hierbei würden jedoch typkorrekte HASKELL-Programme in nicht mehr typkorrekte HASKELL-Programme überführt werden. Dies wird in Kapitel 4 genauer illustriert.

schreiben wir also

$$\texttt{eo} = \texttt{match}\ (\texttt{even}', \texttt{odd}')\ (\texttt{sel}_{2,1}\,\texttt{eo}, \texttt{sel}_{2,2}\,\texttt{eo})\ \underline{\mathsf{exp}}\ \texttt{bot}.$$

Der Effekt hiervon ist, dass der Pattern $(\texttt{even}', \texttt{odd}')$ auf den Ausdruck $(\texttt{sel}_{2,1}\,\texttt{eo}, \texttt{sel}_{2,2}\,\texttt{eo})$ gematcht wird. Dieser Match wird gelingen, wobei $\texttt{even}'$ auf $\texttt{sel}_{2,1}\,\texttt{eo}$ und $\texttt{odd}'$ auf $\texttt{sel}_{2,2}\,\texttt{eo}$ gesetzt wird. Wenn $\underline{\mathsf{exp}}$ das Paar der beiden Lambda-Ausdrücke zur Definition von $\texttt{even}'$ und $\texttt{odd}'$ ist, so ist das Ergebnis der Überführung dieser Deklaration

$$\texttt{eo} = (\backslash\texttt{even}'\ \texttt{->}\ \backslash\texttt{odd}'\ \texttt{->}\ (\backslash\texttt{x}\ \texttt{->}\ ..\texttt{odd}'(\texttt{x}-\texttt{1})..,\backslash\texttt{x}\ \texttt{->}\ ..\texttt{even}'(\texttt{x}-\texttt{1})..))\ (\texttt{sel}_{2,1}\texttt{eo})\ (\texttt{sel}_{2,2}\texttt{eo}).$$

Das Resultat der Anwendung des Lambda-Terms ist demnach

$$\texttt{eo} = (\backslash\texttt{x}\ \texttt{->}\ \ldots(\texttt{sel}_{2,2}\texttt{eo})(\texttt{x}-\texttt{1})\ldots, \backslash\texttt{x}\ \texttt{->}\ \ldots(\texttt{sel}_{2,1}\texttt{eo})(\texttt{x}-\texttt{1})\ldots).$$

Analog dazu wird auch der bisherige Ergebnisausdruck $\texttt{even}'\,\texttt{4}$ durch $\texttt{match}\ (\texttt{even}', \texttt{odd}')$ $\texttt{eo}\ (\texttt{even}'\,\texttt{4})\ \texttt{bot}$ ersetzt. Die weitere Überführung dieses Ausdrucks ergibt

$$(\backslash\texttt{even}'\ \texttt{->}\ \backslash\texttt{odd}'\ \texttt{->}\ (\texttt{even}'\texttt{4}))\ (\texttt{sel}_{2,1}\,\texttt{eo})\ (\texttt{sel}_{2,2}\,\texttt{eo})$$

was bei Anwendung des Lambda-Ausdrucks

$$(\texttt{sel}_{2,1}\,\texttt{eo})\,\texttt{4}$$

ergibt. Die folgende Definition gibt die allgemeine Form der Transformationsregeln an. Die Regeln des Übersetzungsverfahrens sind dabei so zu lesen, dass Teil-Programme oder -Ausdrücke, die oberhalb des horizontalen Balkens stehen, in die entsprechenden Teil-Programme oder -Ausdrücke unterhalb des Balkens transformiert werden.

Definition 2.2.11 (Transformation in einfache HASKELL-Programme) *Die folgenden Regeln dienen zur Transformation von komplexen* HASKELL*-Programmen bzw. von Ausdrücken in komplexen* HASKELL*-Programmen in einfache* HASKELL*-Ausdrücke. Wir sagen, ein Ausdruck* $\underline{\mathsf{exp}}$ *wird in einen Ausdruck* $\underline{\mathsf{exp}}_{tr}$ *überführt, wenn* $\underline{\mathsf{exp}}_{tr}$ *durch wiederholte Anwendung der Regeln aus* $\underline{\mathsf{exp}}$ *entsteht und wenn keine der Regeln mehr auf* $\underline{\mathsf{exp}}_{tr}$ *anwendbar ist.*

(1) **Überführung von Funktionsdeklarationen in Patterndeklarationen**

$$\frac{\underline{\mathsf{var}}\,\underline{\mathsf{pat}}_1^1 \ldots \underline{\mathsf{pat}}_n^1 \;=\; \underline{\mathsf{exp}}^1;\ \ldots;\ \underline{\mathsf{var}}\,\underline{\mathsf{pat}}_1^k \ldots \underline{\mathsf{pat}}_n^k \;=\; \underline{\mathsf{exp}}^k}{\begin{array}{l}\underline{\mathsf{var}} \;=\; \backslash\ \texttt{x}_1 \ldots \texttt{x}_n\ \texttt{->}\ \texttt{case}\ (\texttt{x}_1,\ldots,\texttt{x}_n)\ \texttt{of}\ \{(\underline{\mathsf{pat}}_1^1,\ldots,\underline{\mathsf{pat}}_n^1)\ \texttt{->}\ \underline{\mathsf{exp}}^1; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\underline{\mathsf{pat}}_1^k,\ldots,\underline{\mathsf{pat}}_n^k)\ \texttt{->}\ \underline{\mathsf{exp}}^k\}\end{array}}$$

wobei $\texttt{x}_1,\ldots,\texttt{x}_n$ *neue Variablen sein müssen,* $n > 0$*, und die obige Folge ist* maximal*, d.h., davor und dahinter stehen keine weiteren Funktionsdeklarationen für die Variable* $\underline{\mathsf{var}}$

(2) **Überführung von Lambda-Ausdrücken mit mehreren Patterns**

$$\frac{\backslash\ \underline{\mathsf{pat}}_1\ \ldots\ \underline{\mathsf{pat}}_n\ \texttt{->}\ \underline{\mathsf{exp}}}{\backslash\ \underline{\mathsf{pat}}_1\ \texttt{->}\ (\backslash\ \underline{\mathsf{pat}}_2\ \texttt{->}\ \ldots\ (\backslash\ \underline{\mathsf{pat}}_n\ \texttt{->}\ \underline{\mathsf{exp}})\ldots)} \quad \textit{wobei } n \geq 2$$

(3) **Überführung von Lambda-Patterns in** `case`

$$\frac{\backslash\ \underline{\mathsf{pat}}\ \texttt{->}\ \underline{\mathsf{exp}}}{\backslash\ \underline{\mathsf{var}}\ \texttt{->}\ \texttt{case}\ \underline{\mathsf{var}}\ \texttt{of}\ \underline{\mathsf{pat}}\ \texttt{->}\ \underline{\mathsf{exp}}}$$

falls $\underline{\mathsf{pat}}$ *keine Variable ist, wobei* $\underline{\mathsf{var}}$ *eine neue Variable sein muss*

(4) **Übersetzung von** `case` **in** `match`

$$\frac{\begin{array}{l}\texttt{case}\ \underline{\mathsf{exp}}\ \texttt{of}\ \{\underline{\mathsf{pat}}_1\ \texttt{->}\ \underline{\mathsf{exp}}_1;\\ \qquad\qquad \vdots \\ \qquad\qquad \underline{\mathsf{pat}}_n\ \texttt{->}\ \underline{\mathsf{exp}}_n\}\end{array}}{\begin{array}{l}\texttt{match}\ \underline{\mathsf{pat}}_1\ \underline{\mathsf{exp}}\ \underline{\mathsf{exp}}_1\\ \qquad (\texttt{match}\ \underline{\mathsf{pat}}_2\ \underline{\mathsf{exp}}\ \underline{\mathsf{exp}}_2\\ \qquad\qquad \ldots\\ \qquad\qquad\qquad (\texttt{match}\ \underline{\mathsf{pat}}_n\ \underline{\mathsf{exp}}\ \underline{\mathsf{exp}}_n\ \ \texttt{bot})\ldots)\end{array}}$$

(5) `match` **von Variablen**

$$\frac{\texttt{match}\ \underline{\mathsf{var}}\ \underline{\mathsf{exp}}\ \underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2}{(\backslash\ \underline{\mathsf{var}}\ \texttt{->}\ \underline{\mathsf{exp}}_1)\ \underline{\mathsf{exp}}}$$

(6) `match` **des Joker-Patterns**

$$\frac{\texttt{match}\ _\ \underline{\mathsf{exp}}\ \underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2}{\underline{\mathsf{exp}}_1}$$

(7) `match` **von Konstruktoren**

$$\frac{\texttt{match}\ (\underline{\mathsf{constr}}\ \underline{\mathsf{pat}}_1\ \ldots\ \underline{\mathsf{pat}}_n)\ \underline{\mathsf{exp}}\ \underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2}{\texttt{if}\ (\texttt{isa}_{\underline{\mathsf{constr}}}\ \underline{\mathsf{exp}})\ \texttt{then}\ (\texttt{match}\ (\underline{\mathsf{pat}}_1,\ldots,\underline{\mathsf{pat}}_n)\ (\texttt{argof}_{\underline{\mathsf{constr}}}\ \underline{\mathsf{exp}})\ \underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2)\ \texttt{else}\ \underline{\mathsf{exp}}_2}$$

(8) **`match` von leerem Tupel**

$$\frac{\texttt{match}\ (\,)\ \underline{\mathsf{exp}}\ \underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2}{\texttt{if}\ (\texttt{isa}_{\texttt{0-tuple}}\ \underline{\mathsf{exp}})\ \texttt{then}\ \underline{\mathsf{exp}}_1\ \texttt{else}\ \underline{\mathsf{exp}}_2}$$

(9) **`match` von nicht-leeren Tupeln**

$$\frac{\texttt{match}\ (\underline{\mathsf{pat}}_1, \ldots, \underline{\mathsf{pat}}_n)\ \underline{\mathsf{exp}}\ \underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2}{\begin{array}{l}\texttt{if}\quad (\texttt{isa}_{n\texttt{-tuple}}\ \underline{\mathsf{exp}})\\ \texttt{then match}\ \ \underline{\mathsf{pat}}_1\ (\texttt{sel}_{n,1}\ \underline{\mathsf{exp}})\\ \qquad\qquad\qquad (\texttt{match}\ \underline{\mathsf{pat}}_2\ (\texttt{sel}_{n,2}\ \underline{\mathsf{exp}})\\ \qquad\qquad\qquad\qquad\qquad \ldots\ (\texttt{match}\ \underline{\mathsf{pat}}_n\ (\texttt{sel}_{n,n}\ \underline{\mathsf{exp}})\ \underline{\mathsf{exp}}_1\ \underline{\mathsf{exp}}_2)\ \ldots\\ \qquad\qquad\qquad\qquad\qquad \underline{\mathsf{exp}}_2)\\ \qquad\qquad\qquad \underline{\mathsf{exp}}_2\\ \texttt{else}\ \underline{\mathsf{exp}}_2\end{array}} \qquad \textit{wobei } n \geq 2$$

(10) **Aufteilung von Deklarationen**

$$\frac{\texttt{let}\ P\ \texttt{in}\ \underline{\mathsf{exp}}}{\begin{array}{l}\texttt{let}\ P_1\ \texttt{in}\\ \qquad\qquad \texttt{let}\ P_2\ \texttt{in}\\ \qquad\qquad\qquad\qquad \ldots\\ \qquad\qquad\qquad\qquad\qquad \texttt{let}\ P_k\ \texttt{in}\ \underline{\mathsf{exp}}\end{array}}$$

wobei $P_1, \ldots, P_k$ *eine Aufteilung der Deklarationsfolge* P *ist und* $k \geq 2$

(11) **Überführung von Deklarationsfolgen in eine einzige Deklaration**

$$\frac{\{\underline{\mathsf{var}}_1\ =\ \underline{\mathsf{exp}}_1; \ldots;\ \underline{\mathsf{var}}_n\ =\ \underline{\mathsf{exp}}_n\}}{(\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n)\ =\ (\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n)}$$

wobei $n \geq 2$, $\underline{\mathsf{var}}_i \neq \underline{\mathsf{var}}_j$ *für* $i \neq j$ *und* $\underline{\mathsf{var}}_1 \sim_P \ldots \sim_P \underline{\mathsf{var}}_n$.
Hierbei bezeichnet $P = \{\underline{\mathsf{var}}_1\ =\ \underline{\mathsf{exp}}_1; \ldots;\ \underline{\mathsf{var}}_n\ =\ \underline{\mathsf{exp}}_n\}$.

(12) **Deklaration mehrerer Variablen**

$$\frac{\texttt{let}\ (\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n) = \underline{\mathsf{exp}}\ \texttt{in}\ \underline{\mathsf{exp}}'}{\begin{array}{ll}\texttt{let}\ \underline{\mathsf{var}} = & \texttt{match}\ (\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n)\ (\texttt{sel}_{n,1}\ \underline{\mathsf{var}}, \ldots, \texttt{sel}_{n,n}\ \underline{\mathsf{var}})\ \underline{\mathsf{exp}}\ \texttt{bot}\\ \texttt{in} & \texttt{match}\ (\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n)\ (\texttt{sel}_{n,1}\ \underline{\mathsf{var}}, \ldots, \texttt{sel}_{n,n}\ \underline{\mathsf{var}})\ \underline{\mathsf{exp}}'\ \texttt{bot}\end{array}}$$

wobei $n \geq 2$ *und* $\underline{\mathsf{var}}$ *eine neue Variable ist*

Der folgende Satz zeigt, dass die Anwendung der obigen Transformationregeln immer

terminiert, dass es unerheblich ist, in welcher Reihenfolge man sie anwendet und dass das Ergebnis der Überführung immer ein einfacher HASKELL-Ausdruck ist.

Satz 2.2.12 (Terminierung, Konfluenz und Ergebnis der Transformation)
Sei exp *ein komplexer* HASKELL-*Ausdruck. Dann gilt:*

(a) *Jede Anwendung der Transformationsregeln aus Def. 2.2.11 terminiert, d.h., es existiert ein Ausdruck* $\underline{\text{exp}}_{tr}$*, in den der Ausdruck* exp *überführt werden kann.*

(b) *Bis auf Regel (10) sind die Transformationsregeln "konfluent", d.h., das Ergebnis ihrer wiederholten Anwendung ist eindeutig. Bis auf unterschiedliche Reihenfolgen von Deklarationen und geschachtelten* `let`*-Ausdrücken existiert also genau ein Ausdruck* $\underline{\text{exp}}_{tr}$*, in den der Ausdruck* exp *überführt werden kann.*

(c) *Der Ausdruck* $\underline{\text{exp}}_{tr}$ *ist ein einfacher* HASKELL-*Ausdruck nach Def. 2.2.5.*

Beweisskizze.

(a) Die Terminierung kann mit Hilfe der *lexikographischen Pfadordnung* (lpo) gezeigt werden. (Details hierzu finden sich in Lehrbüchern oder Vorlesungen zu Termersetzungssystemen oder zu automatisierter Programmverifikation [BN98, Gie01, Gie02].) Dabei muss man die Ausdrücke in den Regeln als Terme betrachten und kann dann die Regeln als Termersetzungssystem darstellen. Objektvariablen (wie var oder $\mathtt{x}_i$) können selbst niemals reduziert werden; sie sind also Konstante des Termersetzungssystems.

Alle Regeln, die in ihrer linken Seite einen Lambda-Ausdruck haben, haben als erstes Argument dieses Lambdas entweder einen Nicht-Variablen-Pattern oder eine (mindestens zweielementige) Folge von Patterns. Wir können daher Regel (5) so ändern, dass im Ergebnis (\ var -> $\underline{\text{exp}}_1$) exp anstelle des (zweistelligen) Symbols \ ein neues einstelliges Symbol $\backslash_{\underline{\text{var}}}$ verwendet wird. (Dies kann nach Beendigung der Transformation wieder durch den entsprechenden Lambda-Ausdruck ersetzt werden.) Analog gehen wir in Regel (3) vor.

Dann kann man die Terminierung mit der lpo zeigen (wobei Argumente lexikographisch von links nach rechts verglichen werden). Hierbei wird folgende Präzedenz verwendet (dabei ist ";" das Kombinationssymbol für Deklarationen (das Semikolon in `case`-Ausdrücken muss anders behandelt werden - es hat niedrigste Präzedenz), `->` ist der Pfeil in `case`-Ausdrücken und (...) ist der Tupelkonstruktor).

$$\mathtt{let} \sqsupset \mathtt{;} \sqsupset \mathtt{=} \sqsupset \backslash \sqsupset \mathtt{case} \sqsupset \mathtt{match} \sqsupset$$
$$\underline{\text{var}}, \backslash_{\underline{\text{var}}}, \mathtt{isa}_{\underline{\text{constr}}}, \mathtt{isa}_{n-\mathtt{tuple}}, \mathtt{argof}_{\underline{\text{constr}}}, \mathtt{if}, \mathtt{->}, (\ldots), \mathtt{bot}$$

(b) Die Regeln sind nicht-überlappend, d.h., keine Instanz einer linken Seite unifiziert mit einem Nicht-Variablen-Teilausdruck einer linken Seite (außer mit sich selbst). Da die linken Seiten der Regeln außerdem linear sind (d.h., sie enthalten keine Variable mehr als einmal), folgt, dass die Regeln orthogonal und damit konfluent sind. Dies ist ein klassisches Resultat aus dem Bereich der Termersetzungssysteme [Ros73]. Für weitere Details sei der Leser wieder auf Vorlesungen oder Lehrbücher aus diesem Bereich verwiesen. (Hierbei ist es wieder wichtig, zu erkennen, dass Variablen var für dieses Regelsystem Konstanten sind.)

(c) Diese Behauptung folgt sofort daraus, dass auf jeden nicht-einfachen komplexen Ausdruck $\underline{\mathsf{exp}}$ eine Regel anwendbar ist. □

Damit haben wir also nun ein Verfahren, um komplexe HASKELL-Programme automatisch in einfache HASKELL-Programme zu überführen. Nun lässt sich sofort die Semantik von komplexen HASKELL-Programmen definieren.

Definition 2.2.13 (Semantik von komplexen HASKELL-Programmen) *Für ein komplexes HASKELL-Programm sei* Dom *der zugehörige Domain (nach Def. 2.2.3) und sei P die Folge der Pattern- und Funktionsdeklarationen. Sei* Exp *die Menge der komplexen HASKELL-Ausdrücke. Wir definieren die Funktion $\mathcal{V}al : \mathsf{Exp} \rightarrow \mathsf{Env} \rightarrow \mathsf{Dom}$, die jedem Ausdruck bei jedem Environment (das auf allen seinen freien Variablen definiert ist) einen Wert zuordnet. Die Semantik eines Ausdrucks $\underline{\mathsf{exp}}$, der keine freien Variablen außer den in P definierten und den in HASKELL vordefinierten Variablen enthält, ist bei diesem Programm definiert als*

$$\mathcal{V}al\,[\![(\texttt{let}\ P\ \texttt{in}\ \underline{\mathsf{exp}})_{tr}]\!]\,\omega_{tr}.$$

Die Übersetzung von Def. 2.2.11 ist nicht nur für die Definition der Semantik nützlich, sondern sie kann auch zur Typüberprüfung und zur Implementierung der Programmiersprache verwendet werden. Wir werden darauf in den folgenden Kapiteln eingehen.

Kapitel 3

Der Lambda-Kalkül

In diesem Kapitel stellen wir den Lambda-Kalkül von Church vor [Chu41]. Der Lambda-Kalkül wurde lange vor allen Programmiersprachen entwickelt und diente Church damals zur Formalisierung des Begriffs der *Berechenbarkeit*. Als sich herausstellte, dass die Ausdruckskraft dieses Kalküls genauso groß wie die Ausdruckskraft von Turing-Maschinen ist, führte ihn dies zur Churchschen These vom intuitiven Berechenbarkeitsbegriff.

Insbesondere bildet aber der Lambda-Kalkül die Grundlage für alle funktionalen Programmiersprachen: Sprachen wie HASKELL sind lediglich syntaktische (und lesbarere) Varianten des Lambda-Kalküls. Insbesondere lässt sich jedes HASKELL-Programm in den Lambda-Kalkül übersetzen. Obwohl der Lambda-Kalkül eine außerordentlich einfache Sprache ist, stellt er also eine vollständige Programmiersprache dar, in der sich jedes berechenbare Programm formulieren lässt. Dieses Resultat hat praktische Konsequenzen, da es damit möglich ist, ein HASKELL-Programm zuerst von seinem "syntaktischen Zucker" zu befreien und in den Lambda-Kalkül zu überführen. Auf dieser Ebene kann dann die Überprüfung der Typkorrektheit und anschließend die Ausführung des Programms stattfinden. Eine erste Implementierung ist durch die Reduktionsregeln des Lambda-Kalküls gegeben, die damit die Semantik von HASKELL auf eine alternative (operationelle) Weise charakterisieren. Der Lambda-Kalkül ist also ein wichtiges Hilfsmittel für die Implementierung der Programmiersprache und wir werden daher auch im nächsten Kapitel (über Typüberprüfung) nur noch auf dem Lambda-Kalkül statt auf HASKELL operieren.

Die Rechtfertigung für die Übersetzung in den Lambda-Kalkül bildet die Semantik von HASKELL aus Kapitel 2. Hierdurch wurde festgelegt, was die Konstrukte der Programmiersprache bedeuten sollen. Wenn bei der Übersetzung in den Lambda-Kalkül diese Semantik zugrunde gelegt wird, so führt dies (bei einer korrekten Implementierung des Lambda-Kalküls) auch zu einer korrekten Implementierung von HASKELL.

In Abschnitt 3.1 führen wir zunächst die Syntax des Lambda-Kalküls ein und stellen anschließend in Abschnitt 3.2 die Reduktionsregeln dieses Kalküls vor. Anschließend geben wir in Abschnitt 3.3 eine Implementierung von HASKELL an, indem wir HASKELL in den Lambda-Kalkül übersetzen und zeigen, mit welcher Strategie die Reduktionsregeln des Lambda-Kalküls angewendet werden müssen, um die Semantik von HASKELL zu realisieren. Schließlich gehen wir in Abschnitt 3.4 darauf ein, dass sich der Lambda-Kalkül noch weiter vereinfachen lässt (indem man keine Konstanten mehr zulässt) und man dennoch weiterhin eine vollständige Programmiersprache behält.

3.1 Syntax des Lambda-Kalküls

Die Syntax des Lambda-Kalküls ist die Sprache der Lambda-Terme, die wie folgt definiert sind:

Definition 3.1.1 (Lambda-Terme) *Sei $\mathcal{C}$ eine Menge von Konstanten und $\mathcal{V}$ eine (abzählbar unendliche) Menge von Variablen. Die Menge der Lambda-Terme Λ ist definiert als die kleinste Menge, so dass:*

- $\mathcal{C} \subseteq \Lambda$
- $\mathcal{V} \subseteq \Lambda$
- $(t_1\, t_2) \in \Lambda$, *falls* $t_1, t_2 \in \Lambda$
- $\lambda x.t \in \Lambda$, *falls* $x \in \mathcal{V}$ *und* $t \in \Lambda$

Terme der Form $(t_1\, t_2)$ bezeichnet man als *Anwendungen* oder *Applikationen*, denn die Semantik eines solchen Ausdrucks ist, dass der Term t_1 auf den Term t_2 *angewendet* wird. Man erkennt unmittelbar den Zusammenhang zum entsprechenden HASKELL-Ausdruck `(t_1 t_2)`.

Terme wie $\lambda x.t$ heißen *(Lambda-)Abstraktionen* und die intuitive Bedeutung eines solchen Terms ist eine Funktion, die jeden Wert x auf den Wert von t abbildet. Ein solcher Term entspricht also dem HASKELL-Ausdruck `\x -> t`. Lambda-Abstraktionen repräsentieren *anonyme* Funktionen (die durch kein Funktionssymbol bezeichnet sind).

Als Konstanten werden beliebige Symbole eingesetzt. Beispielsweise kann man sowohl Konstanten wie `Succ` und `Zero` (die in HASKELL Konstruktoren entsprechen würden) als auch Konstanten wie `sqrt`, $+$, $\mathtt{isa}_{\mathtt{Succ}}$, etc. verwenden (die in HASKELL definierten Funktionen entsprechen würden). Um Klammern zu sparen, vereinbaren wir (ähnlich wie in HASKELL) die folgenden Konventionen:

- Anwendungen assoziieren nach links. Damit steht also $(t_1\, t_2\, t_3)$ für $((t_1\, t_2)\, t_3)$.
- Der Wirkungsbereich eines Lambdas erstreckt sich so weit wie möglich nach rechts. Daher steht $\lambda x.x\, x$ für $\lambda x.(x\, x)$.
- Wir schreiben $\lambda x\, y.t$ statt $\lambda x.\lambda y.t$.

Man erkennt, dass die Menge der Lambda-Terme wirklich auf sehr einfache Weise gebildet wird. Umso erstaunlicher ist, dass sich auf diese Weise tatsächlich alle berechenbaren Funktionen schreiben lassen (d.h., dass die Programmiersprache der Lambda-Terme tatsächlich alle möglichen Programme zulässt). Bevor wir in Abschnitt 3.2 die Semantik dieser Programmiersprache zeigen, benötigen wir aber noch einige hilfreiche Begriffe.

In Def. 2.2.6 hatten wir bereits für HASKELL-Ausdrücke definiert, welche Variablen in ihnen *frei* vorkommen. Dieses Konzept benötigen wir nun auch für Lambda-Terme. Hierbei ist eine Variable in einem Term genau dann frei, wenn sie nicht durch ein darüber stehendes Lambda gebunden wird.

Definition 3.1.2 (Freie Variablen eines Lambda-Terms) *Für jeden Lambda-Term* $t \in \Lambda$ *definieren wir* $\mathrm{free}(t) \subseteq \mathcal{V}$*, die Menge seiner* freien Variablen*, wie folgt:*

- $\mathrm{free}(c) = \varnothing$ *für alle* $c \in \mathcal{C}$
- $\mathrm{free}(x) = \{x\}$ *für alle* $x \in \mathcal{V}$
- $\mathrm{free}(t_1\ t_2) = \mathrm{free}(t_1) \cup \mathrm{free}(t_2)$ *für alle* $t_1, t_2 \in \Lambda$
- $\mathrm{free}(\lambda x.t) = \mathrm{free}(t) \setminus \{x\}$ *für alle* $x \in \mathcal{V}$*,* $t \in \Lambda$*.*

Ein Term t heißt geschlossen*, falls* $\mathrm{free}(t) = \varnothing$*. Man bezeichnet geschlossene Lambda-Terme auch als* Kombinatoren*.*

Hiermit kann man nun die Definition von *Substitutionen* angeben. Eine Substitution ersetzt alle freien Vorkommen einer Variablen x durch einen Term t. Für die Anwendung einer derartigen Substitution auf einen Term r schreiben wir $r[x/t]$. Wenn r der Term $\lambda y.yx$ ist und t der Term $\lambda u.uv$, so ist $r[x/t]$ also der Term $\lambda y.y(\lambda u.uv)$. Analog dazu kann man Substitutionen angeben, die endliche Mengen von Variablen durch Terme ersetzen.

Man erkennt, dass ein Term wie $\lambda y.yx$ einfach einer Funktion entspricht, die jede Funktion y auf ihr Resultat an der Stelle x abbildet. Eine Umbenennung der gebundenen Variablen y in eine neue Variable y' (die nicht bereits frei in diesem Term vorkommt) ändert daher nichts an der Bedeutung des Terms. Der Term $r' = \lambda y'.y'x$ bezeichnet nämlich genau die gleiche Funktion wie $\lambda y.yx$. Demnach sollten auch $r[x/t]$ und $r'[x/t]$ stets dieselbe Funktion bezeichnen.

Betrachten wir nun einen modifizierten Term $t' = \lambda u.uy$, in dem die Variable y aus der Lambda-Abstraktion von r frei auftritt. Wenn man nun x durch t' ersetzt, so erhält man aus dem Term r den Term $\lambda y.y(\lambda u.uy)$ und aus dem Term r' den Term $\lambda y'.y'(\lambda u.uy)$. Diese beiden Terme bezeichnen aber nicht mehr dieselbe Funktion! Der erste Term ist geschlossen, da die Variable y durch das darüberstehende Lambda gebunden ist, während die Bedeutung des zweiten Terms vom Wert der Variablen y abhängt.

Dass die Variable y in t' identisch mit der durch das Lambda gebundenen Variable y in r ist, ist reiner Zufall, da gebundene Variablen wie y in r jederzeit umbenannt werden können. Es handelt sich hier um einen sogenannten *Namenskonflikt*. Solche Konflikte müssen stets behoben werden, bevor die Substitution durchgeführt wird. Um $r[x/t']$ zu bilden, muss man also zunächst das gebundene y aus r in eine andere Variable y' umbenennen und darf erst dann die Ersetzung von x durch t' vornehmen. Auf diese Weise bezeichnen dann $r[x/t']$ und $r'[x/t']$ tatsächlich die gleiche Funktion. Natürlich darf man bei der gebundenen Umbenennung keine Variable einführen, die bereits in dem Term frei vorkommt. Der Grund für diese Einschränkung ist, dass zwar die Terme $\lambda y.y\,x$ und $\lambda y'.y'\,x$ einander entsprechen, aber der Term, in dem y in die bereits frei vorkommende Variable x umbenannt wird (d.h. $\lambda x.x\,x$) bezeichnet eine andere Funktion.

Definition 3.1.3 (Substitution auf Lambda-Termen) *Für alle* $r, t \in \Lambda$ *und alle* $x \in \mathcal{V}$ *ist* $r[x/t]$ *wie folgt definiert:*

- $x[x/t] = t$

- $y[x/t] = y$ *für alle* $y \in \mathcal{V}$ *mit* $y \neq x$
- $c[x/t] = c$ *für alle* $c \in \mathcal{C}$
- $(r_1\, r_2)[x/t] = (r_1[x/t]\; r_2[x/t])$ *für alle* $r_1, r_2 \in \Lambda$
- $(\lambda x.r)[x/t] = \lambda x.r$
- $(\lambda y.r)[x/t] = \lambda y.(r[x/t])$, *falls* $y \neq x$ *und* $y \notin \mathrm{free}(t)$
- $(\lambda y.r)[x/t] = \lambda y'.(r[y/y'][x/t])$, *falls* $y \neq x$, $y \in \mathrm{free}(t)$, $y' \notin \mathrm{free}(r) \cup \mathrm{free}(t)$

3.2 Reduktionsregeln des Lambda-Kalküls

In diesem Abschnitt geben wir die Regeln zur Auswertung (oder *Reduktion*) von Lambda-Termen an. Damit wird deutlich, wie die Programmiersprache der Lambda-Terme arbeitet (d.h., wir erhalten ihre operationelle Semantik).

Die erste Reduktionsregel erlaubt es, gebundene Variablen beliebig umzubenennen. Wie oben erläutert, ändert dies nichts an der Bedeutung der Terme, vorausgesetzt, dass man bei der Umbenennung keine Variablen einführt, die bereits in dem Term frei vorkommen.

Definition 3.2.1 (α-Reduktion) *Die Relation* $\to_\alpha \subseteq \Lambda \times \Lambda$ *ist die kleinste Relation mit*

- $\lambda x.t \;\to_\alpha\; \lambda y.t[x/y]$, *falls* $y \notin \mathrm{free}(t)$
- *falls* $t_1 \;\to_\alpha\; t_2$, *dann auch* $(t_1\, r) \;\to_\alpha\; (t_2\, r)$, $(r\, t_1) \;\to_\alpha\; (r\, t_2)$ *und* $\lambda y.t_1 \;\to_\alpha\; \lambda y.t_2$ *für alle* $r \in \Lambda$, $y \in \mathcal{V}$.

Da diese Relation offensichtlich symmetrisch ist, spricht man neben der "α-Reduktion" auch von der "α-Konversion".

Wie im zweiten Punkt der Definition illustriert, dürfen die Reduktionsregeln des Lambda-Kalküls generell immer auch auf Teilterme eines Terms angewendet werden. Wir erhalten also z.B. $\lambda xy.x\,y \;\to_\alpha\; \lambda xy'.x\,y'$.

Die zweite Reduktionsregel ist die sogenannte β-*Reduktion*, die es erlaubt, Lambda-Abstraktionen auf beliebige Terme anzuwenden. Eine Lambda-Abstraktion $\lambda x.t$ angewendet auf ein Argument r kann reduziert werden, indem man im Rumpf t der Abstraktion alle Vorkommen der gebundenen Variablen x durch das aktuelle Argument r substituiert. Hierbei wird natürlich der Begriff der Substitution aus Def. 3.1.3 verwendet. Die Substitution führt also vor der Ersetzung ggf. eine α-Konversion durch, d.h. eine Umbenennung der gebundenen Variablen x.

Definition 3.2.2 (β-Reduktion) *Die Relation* $\to_\beta \subseteq \Lambda \times \Lambda$ *ist die kleinste Relation mit*

- $(\lambda x.t)\, r \;\to_\beta\; t[x/r]$
- *falls* $t_1 \;\to_\beta\; t_2$, *dann auch* $(t_1\, r) \;\to_\beta\; (t_2\, r)$, $(r\, t_1) \;\to_\beta\; (r\, t_2)$ *und* $\lambda y.t_1 \;\to_\beta\; \lambda y.t_2$ *für alle* $r \in \Lambda$, $y \in \mathcal{V}$.

Beispielsweise erhalten wir also

$$\begin{array}{rcl}(\lambda x.x)\,\texttt{Zero} & \rightarrow_\beta & \texttt{Zero}\\ (\lambda xy.x\,y)\,y & \rightarrow_\beta & \lambda y'.y\,y'\\ (\lambda x.\texttt{plus}\,x\,1)\,((\lambda y.\texttt{times}\,y\,y)\,3) & \rightarrow_\beta & (\lambda x.\texttt{plus}\,x\,1)\,(\texttt{times}\,3\,3)\\ & \rightarrow_\beta & \texttt{plus}\,(\texttt{times}\,3\,3)\,1\end{array}$$

Man erkennt, dass es manchmal mehrere Möglichkeiten gibt, die β-Regel auf einen Term anzuwenden. Da die Reduktion der Auswertung von Lambda-Termen entspricht, sollte das Ergebnis der Auswertung jedoch eindeutig sein. Hierzu verwendet man die folgenden Begriffe.

Definition 3.2.3 (Transitiv-Reflexive Hülle, Normalform, Konfluenz) *Sei $\rightarrow$ eine Relation über einer Menge N.*

(a) *Die Relation $\rightarrow^*$ (die* transitiv-reflexive Hülle *von $\rightarrow$) ist die kleinste Relation, so dass für alle $t_1, t_2, t_3 \in N$ gilt:*

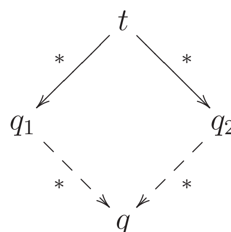
- *wenn $t_1 \rightarrow t_2$, dann $t_1 \rightarrow^* t_2$*
- *wenn $t_1 \rightarrow t_2 \rightarrow^* t_3$, dann $t_1 \rightarrow^* t_3$*
- *$t_1 \rightarrow^* t_1$.*

(b) *Ein Objekt $q \in N$ heißt $\rightarrow$-*Normalform *gdw. es kein Objekt $q' \in N$ mit $q \rightarrow q'$ gibt. Das Objekt q heißt* Normalform *eines Objekts t gdw. $t \rightarrow^* q$ und q Normalform ist.*

(c) *Die Relation $\rightarrow$ heißt* konfluent, *wenn für alle $t, q_1, q_2 \in N$ gilt:*

Wenn $t \rightarrow^* q_1$ und $t \rightarrow^* q_2$,
dann existiert ein $q \in N$ mit $q_1 \rightarrow^* q$ und $q_2 \rightarrow^* q$.

Anschaulich bedeutet Konfluenz, dass zwei unterschiedliche "Wege" von t nach q_1 und von t nach q_2 immer zu einem *gemeinsamen* "Wegpunkt" q fortgesetzt werden können. Die Existenz mehrerer Wege von t aus repräsentiert einen Indeterminismus, denn man kann ausgehend von t sowohl nach q_1 als auch nach q_2 gelangen. Die Eigenschaft der Konfluenz gewährleistet dann, dass solche Indeterminismen beliebig aufgelöst werden können. Man kann dies durch folgendes Bild illustrieren: Wenn die durchgezogenen Pfeile existieren, folgt daraus die Existenz der gestrichelten Pfeile und die Existenz eines geeigneten Objekts q.



Die Bedeutung der Konfluenz beschreibt das folgende Lemma.

Lemma 3.2.4 (Konfluenz bedeutet eindeutige Normalformen) *Sei $\to$ eine konfluente Relation über einer Menge N. Dann hat jedes Objekt $t \in N$ höchstens eine Normalform.*

Beweis. Seien q_1 und q_2 Normalformen von t. Daraus folgt $t \to^* q_1$ und $t \to^* q_2$. Aufgrund der Konfluenz muss es also ein Objekt q geben mit $q_1 \to^* q$ und $q_2 \to^* q$. Da q_1 und q_2 aber Normalformen sind, folgt $q_1 = q = q_2$. □

Der folgende grundlegende Satz von Church und Rosser besagt, dass die Auswertungsrelation $\to_\beta$ des Lambda-Kalküls in der Tat konfluent ist. Die Konfluenz (bzw. eine hierzu äquivalente Eigenschaft) wird daher auch oft als *Church-Rosser Eigenschaft* bezeichnet. Der Beweis dieses Satzes sowie eine umfassende Behandlung des Lambda-Kalküls findet sich im Standardwerk [Bar84]. Hierbei werden Terme, die sich nur durch α-Konversion unterscheiden, als gleich betrachtet.

Satz 3.2.5 (Konfluenz des Lambda-Kalküls mit β-Regeln) *$\to_\beta$ ist konfluent, d.h., wenn $t \to_\beta^* q_1$ und $t \to_\beta^* q_2$, dann existieren $q, q' \in \Lambda$ mit $q_1 \to_\beta^* q$, $q_2 \to_\beta^* q'$ und $q \to_\alpha^* q'$.*

Beweis. Siehe [Bar84]. □

Diese Eigenschaft ist entscheidend für die Auswertung von funktionalen Programmen (die ja auf dem Lambda-Kalkül basieren). Sie besagt, dass man bei jeder Auswertungsstrategie dasselbe Ergebnis erhält.

Die letzte Gruppe von Reduktionsregeln des Lambda-Kalküls sind die sogenannten δ-Regeln, die angeben, wie man Terme reduzieren kann, bei denen Konstanten aus $\mathcal{C}$ auf Argumente angewendet werden. Hierbei wollen wir sicherstellen, dass die Konfluenz des Lambda-Kalküls erhalten bleibt. Im allgemeinen ist dies natürlich offensichtlich nicht gewährleistet, wenn die δ-Regeln beliebige Form haben. So könnten wir beispielsweise für eine Konstante $c \in \mathcal{C}$ die Regeln

$$c\,x\,y \to_\delta x \quad \text{und} \quad c\,x\,y \to_\delta y$$

hinzufügen, die sofort die Konfluenz des Kalküls zerstören würden.

Man beschränkt sich daher auf δ-Regeln von bestimmter eingeschränkter Gestalt, bei denen die Erhaltung der Konfluenz gesichert ist.

Definition 3.2.6 (δ-Reduktion) *Eine Menge von Regeln δ der Form $c\,t_1 \dots t_n \to r$ mit $c \in \mathcal{C}$, $t_1, \dots, t_n, r \in \Lambda$ heißt eine Delta-Regelmenge, falls folgende Bedingungen erfüllt sind:*

- *$t_1, \dots, t_n, r$ sind jeweils geschlossene Lambda-Terme.*
- *Die t_i sind in $\to_\beta$-Normalform und sie enthalten keine linke Seite einer Regel aus δ.*
- *In δ existieren keine zwei verschiedenen Regeln $c\,t_1 \dots t_n \to r$ und $c\,t_1 \dots t_m \to r'$ mit $m \geq n$.*

Für eine solche Menge δ definieren wir die Relation $\to_\delta$ als die kleinste Relation mit

- *$l \to_\delta r$ für alle $l \to r \in \delta$*

- *falls* $t_1 \to_\delta t_2$*, dann auch* $(t_1\, r) \to_\delta (t_2\, r)$, $(r\, t_1) \to_\delta (r\, t_2)$ *und* $\lambda y.t_1 \to_\delta \lambda y.t_2$ *für alle* $r \in \Lambda$, $y \in \mathcal{V}$.

Wir bezeichnen die Kombination der β*- und der* δ*-Reduktion mit* $\to_{\beta\delta}$*, d.h., es gilt* $\to_{\beta\delta} = \to_\beta \cup \to_\delta$.

Ein Beispiel für eine δ-Regelmenge ist

$$\begin{aligned}\delta \;=\; & \{\mathtt{isa}_{\mathtt{Succ}}(\mathtt{Succ}\, t) \to \mathtt{True} \mid t \in \Lambda, t \text{ geschlossen und in } \to_{\beta\delta}\text{-Normalform}\} \cup \\ & \{\mathtt{isa}_{\mathtt{Succ}}\, \mathtt{Zero} \to \mathtt{False}\}.\end{aligned}$$

Man kann nun zeigen, dass die Konfluenz des Lambda-Kalküls (Satz 3.2.5) erhalten bleibt, wenn man zusätzlich zu den β-Regeln noch δ-Regeln erlaubt.

Satz 3.2.7 (Konfluenz des Lambda-Kalküls mit β- und δ-Regeln) $\to_{\beta\delta}$ *ist konfluent, d.h., wenn* $t \to_{\beta\delta}^* q_1$ *und* $t \to_{\beta\delta}^* q_2$*, dann existieren* $q, q' \in \Lambda$ *mit* $q_1 \to_{\beta\delta}^* q$, $q_2 \to_{\beta\delta}^* q'$ *und* $q \to_\alpha^* q'$.

Beweis. Siehe [Klo93] und [Bar84]. □

Durch die Relationen $\to_\beta$ und $\to_\delta$ haben wir also die Semantik des Lambda-Kalküls auf *operationelle* Weise festgelegt: Wir haben einen Interpreter definiert, der Lambda-Terme auswertet. Mit $\to_\alpha$ haben wir darüberhinaus angegeben, dass wir bestimmte Lambda-Terme als gleich betrachten. Eine weitere mögliche Reduktionsregel des Lambda-Kalküls ist die sogenannte η-Reduktion, mit $\lambda x.t\, x \to_\eta t$, falls $x \notin \text{free}(t)$. Sie wird allerdings im Bereich des funktionalen Programmierens nur selten verwendet. (Der Grund ist, dass Terme der Art $\lambda x.t\, x$ dort normalerweise nicht mehr weiter ausgewertet werden. Dann führt die η-Regel aber zu unerwünschten Resultaten, da t selbst wieder auswertbar sein kann und seine Auswertung sogar ggf. zur Nicht-Terminierung führen könnte. Wir werden auf diesen Effekt im nächsten Abschnitt wieder zurückkommen, wenn wir uns überlegen, welche Reduktionsstrategie des Lambda-Kalküls für eine Implementierung von HASKELL gewählt werden sollte.)

Während der Lambda-Kalkül bereits in den 30er Jahren von Church entwickelt wurde, fand Scott erst gegen Ende der 60er Jahre eine modelltheoretische Semantik für diesen Kalkül (bei der die $\alpha-$, $\beta-$, $\delta-$ und $\eta-$Regeln tatsächlich gerechtfertigt sind). Es existiert also tatsächlich eine Menge mathematischer Objekte (ein *Domain*), dessen Objekte isomorph zu den Normalformen des Lambda-Kalküls sind (wobei α-konvertierbare Terme die gleichen Objekte bezeichnen). Das Problem hierbei war, dass dieser Domain D auch Funktionen aus $D \to D$ enthalten muss, denn zu jedem Term t existiert auch der Term $(t\, t)$. Man bezeichnet den Lambda-Kalkül daher auch als eine *Logik höherer Ordnung*. Für $|D| > 1$ kann aber D nicht isomorph zu seinem eigenen Funktionenraum sein, da der Funktionenraum eine höhere Kardinalität als D hat. Die Lösung für dieses Dilemma besteht darin, sich auf cpo's und stetige Funktionen zu beschränken. Dieses Prinzip der Definition der Semantik lässt sich vom Lambda-Kalkül leicht auf (andere) funktionale Programmiersprachen übertragen. Auf diese Weise konnten wir daher die Semantik von HASKELL in Kapitel 2 angeben.

3.3 Reduzierung von HASKELL auf den Lambda-Kalkül

In diesem Abschnitt werden wir HASKELL auf den Lambda-Kalkül zurückführen. In den vorigen Abschnitten haben wir die Sprache der Lambda-Terme angegeben und die β- und δ-Regeln zur Reduktion solcher Terme vorgestellt. Es wird sich herausstellen, dass man jedes HASKELL-Programm (bzw. jeden HASKELL-Ausdruck, der in einem HASKELL-Programm ausgewertet werden soll) als Lambda-Term darstellen kann. Die Auswertung dieses Lambda-Terms geschieht dann mit der β- und δ-Reduktion. Auf diese Weise erhalten wir eine erste Implementierung von HASKELL.

Zunächst zeigt sich, dass der Lambda-Kalkül zwar konfluent ist, aber (selbst wenn man nur β-Reduktion betrachtet) nicht terminiert:

$$(\lambda x.x\,x)\,(\lambda x.x\,x) \;\to_\beta\; (\lambda x.x\,x)\,(\lambda x.x\,x) \;\to_\beta\; \dots$$

Es gibt allerdings auch Beispiele, bei denen manche Reduktionen terminieren und andere nicht. Beispielsweise erhält man

$$(\lambda x.y)((\lambda x.x\,x)\,(\lambda x.x\,x)) \;\to_\beta\; y,$$

wenn man den *außen* liegenden Term reduziert. Wenn man hingegen nur den inneren Term reduziert, erhält man eine nicht-terminierende Reduktion.

Es hängt also wieder von der *geeigneten Reduktionsstrategie* ab, ob die Reduktion terminiert. Da wir den Lambda-Kalkül ja zur Implementierung der Sprache HASKELL verwenden wollen (die eine nicht-strikte Semantik hat), verwenden wir die sogenannte *leftmost outermost* Strategie: Hierbei muss ein Lambda-Term stets soweit oben links wie möglich durch eine Reduktionsregel reduziert werden. Man kann zeigen, dass es sich hierbei um eine *sichere* Strategie handelt, d.h., falls ein Term eine Normalform hat, so wird sie mit dieser Strategie auch erreicht. (Dies bezeichnet man auch als das Standardisierungstheorem des Lambda-Kalküls.)

Darüber hinaus ist es bei der Implementierung von Programmiersprachen mit nicht-strikter Semantik jedoch nicht wünschenswert, immer bis zum Erreichen der Normalform zu reduzieren. In einem Term $f\,t_1\,\dots\,t_n$, in dem von vornherein klar ist, dass die Funktion f niemals ausgewertet werden kann, sollten die Argumente $t_1, \dots, t_n$ dann auch nicht weiter ausgewertet werden, selbst wenn auf sie noch Reduktionsregeln anwendbar wären. Ein Term wie $(\mathtt{1}+\mathtt{2}) : []$ sollte also nicht weiter ausgewertet werden, selbst wenn es eine δ-Regel $\mathtt{1}+\mathtt{2} \to_\delta \mathtt{3}$ gibt, da die Funktion : auch bei dem Argument $\mathtt{3}$ nicht ausgewertet werden kann. Der Grund ist, dass : ein Konstruktor ist, für den es keine δ-Regeln gibt. Ebenso verhält es sich bei Termen wie $x\,(\mathtt{1}+\mathtt{2})$, denn eine Variable x kann niemals "ausgewertet werden". Man spricht bei solchen Termen von der *Weak Head Normal Form* (WHNF), da hier der Kopf (head) des Terms bereits feststeht (er ist : bzw. x) und man nur solange auswerten sollte, bis dieses oberste Symbol des Terms gefunden wurde.

Ebenso sollte ein Term $\lambda x.t$ nicht weiter ausgewertet werden, auch wenn noch Regeln auf den Teilterm t anwendbar sind. Der Grund ist wiederum, dass sich das oberste Symbol λ des Terms dadurch nicht mehr ändern würde. Hingegen könnte eine Auswertung des Teilterms t ggf. zu einer (in nicht-strikten Sprachen unerwünschten) Nicht-Terminierung führen. Wir definieren die WHNF wie folgt:

Definition 3.3.1 (Weak Head Normal Form) *Ein Term ist in* Weak Head Normal Form *(WHNF) gdw. er eine Normalform oder von einer der folgenden Gestalten ist:*

- $\lambda x.t$ *für beliebiges* $t \in \Lambda$
- $c\, t_1 \, \dots \, t_n$ *für beliebige* $t_1, \dots, t_n \in \Lambda$ *und jedes* $c \in \mathcal{C}$*, für das es keine Regeln in* δ *gibt (d.h.,* c *ist ein Konstruktor)*
- $x\, t_1 \, \dots \, t_n$ *für beliebige* $t_1, \dots, t_n \in \Lambda$ *und* $x \in \mathcal{V}$

Mit Hilfe der leftmost outermost Reduktionsstrategie und der WHNF können wir jetzt die Auswertungsrelation des Lambda-Kalküls definieren, die wir benötigen, um eine nicht-strikte Sprache wie HASKELL zu implementieren.

Definition 3.3.2 (Weak Head Normal Order Reduction) *Die WHNO-Reduktion auf Lambda-Termen ist wie folgt definiert:* $t \to r$ *gdw.* t *ist nicht in WHNF und* $t \to_{\beta\delta} r$*, wobei die Reduktion soweit oben links wie möglich stattfindet ("leftmost outermost").*

Nun geben wir an, wie man HASKELL-Ausdrücke in Lambda-Terme übersetzen kann. Wir betrachten wieder zunächst nur einfache Ausdrücke (Def. 2.2.5) und erweitern die Übersetzung anschließend auf komplexe Ausdrücke, indem wir die Transformation von Def. 2.2.11 verwenden.

Variablen aus HASKELL-Ausdrücken entsprechen Variablen des Lambda-Kalküls. Ebenso entsprechen Datenkonstruktoren und vordefinierte Operationen aus HASKELL Konstanten des Lambda-Kalküls. Um n-stellige Tupel (mit $n \neq 1$) zu übersetzen, verwenden wir im Lambda-Kalkül eine neue Konstante $\mathtt{tuple}_n$, wobei $\mathtt{tuple}_n\, t_1 \, \dots \, t_n$ den Termtupel $(t_1, \dots, t_n)$ repräsentiert. Einstellige Tupel ($\underline{\mathsf{exp}}$) werden in den Term übersetzt, der $\underline{\mathsf{exp}}$ entspricht. HASKELL-Applikationen werden unmittelbar in Applikationen des Lambda-Kalküls überführt. Ausdrücke, die mit `if` gebildet werden, werden in entsprechende Lambda-Terme übersetzt, die mit der Konstanten `if` des Lambda-Kalküls konstruiert werden und die Lambda-Abstraktionen von HASKELL entsprechen direkt den Lambda-Abstraktionen des Lambda-Kalküls.

Betrachten wir nun noch die Übersetzung von Ausdrücken mit lokaler Deklaration. Zunächst untersuchen wir einen Ausdruck $\mathtt{let}\ \underline{\mathsf{var}} = \underline{\mathsf{exp}}\ \mathtt{in}\ \underline{\mathsf{exp}}'$, wobei die Variable $\underline{\mathsf{var}}$ nicht frei in dem Ausdruck $\underline{\mathsf{exp}}$ vorkommt. Wenn $\underline{\mathsf{exp}}$ in den Lambda-Term t und $\underline{\mathsf{exp}}'$ in den Lambda-Term t' übersetzt wird, dann kann man den Gesamtausdruck in

$$t'\,[\underline{\mathsf{var}}/t]$$

überführen. Damit werden also im Term t' alle (freien) Vorkommen von $\underline{\mathsf{var}}$ durch t ersetzt. Der Ausdruck

```
let x = 3 in x + 2
```

wird also in den Lambda-Term $3 + 2$ übersetzt. (Falls eine entsprechende δ-Regel vorhanden ist, kann er anschließend natürlich weiter zu 5 ausgewertet werden.)

Bei rekursiven Deklarationen (d.h. wenn die Variable $\underline{\mathsf{var}}$ selbst wieder in $\underline{\mathsf{exp}}$ frei auftritt) lässt sich diese Übersetzung jedoch nicht verwenden. Hierzu betrachten wir wieder den folgenden Ausdruck.

```
let  fact = \x -> if x <=0 then 1 else fact(x − 1) ∗ x  in  fact 2
```

Gemäß der Definition der Semantik von HASKELL (Def. 2.2.7), bedeutet diese Deklaration, dass `fact` der *kleinste Fixpunkt* derjenigen Funktion zugeordnet wird, die durch folgenden HASKELL-Ausdruck beschrieben wird:

$$\backslash\texttt{fact -> }\backslash\texttt{x -> if x <=0 then 1 else fact}(\texttt{x} - 1) * \texttt{x}.$$

Wir verwenden eine weitere Konstante `fix` des Lambda-Kalküls, die zur Berechnung des kleinsten Fixpunkts benutzt wird. Die Semantik des Terms `fix` t soll also jeweils der kleinste Fixpunkt der Funktion sein, die durch den Term t beschrieben wird. Dann können wir rekursive Deklarationen wie oben sofort in nicht-rekursive Deklarationen umformulieren:

$$\texttt{let}\quad \texttt{fact} = \texttt{fix}(\backslash\texttt{fact -> }\backslash\texttt{x -> if x <=0 then 1 else fact}(\texttt{x} - 1) * \texttt{x})\quad \texttt{in}\quad \texttt{fact 2}$$

Da diese Deklaration nicht mehr rekursiv ist (d.h., `fact` kommt nicht mehr frei auf der rechten Seite der Deklaration vor), kann man sie nun wie zuvor in den folgenden Lambda-Term übersetzen:

$$(\texttt{fix}\ (\lambda\, \texttt{fact}\, \texttt{x}.\, \texttt{if}\ (\texttt{<=}\ \texttt{x}\ 0)\ 1\ (*\ (\texttt{fact}\ (-\ \texttt{x}\ 1))\ \texttt{x})))\ 2.$$

Man kann sich überzeugen, dass nun durch β- und δ-Reduktion tatsächlich der Term `2` entsteht. Hierzu benötigen wir natürlich neben den entsprechenden δ-Regeln für `if` und die in HASKELL vordefinierten Funktionen $*, -,$ `<=` auch eine entsprechende δ-Regel für die Konstante `fix`. Diese erlaubt die Reduktion

$$\texttt{fix}\, t \quad \to^* \quad t\ (\texttt{fix}\, t).$$

Sie ist dadurch gerechtfertigt, dass `fix` t ja dem (kleinsten) Fixpunkt von t entsprechen soll.[1]

Man erkennt also, dass die Übersetzung von HASKELL in den Lambda-Kalkül voraussetzt, dass wir bereits festgelegt haben, was die Semantik rekursiver Deklarationen sein soll (nämlich jeweils der entsprechende kleinste Fixpunkt). Dann entspricht diese Übersetzung tatsächlich der gewünschten denotationellen Semantik.

In der Tat kann man die Funktion `fix` auch bei nicht-rekursiven Deklarationen verwenden. Die nicht-rekursive Deklaration $\texttt{x} = 3$ würde dann in $\texttt{x} = \texttt{fix}(\lambda\texttt{x}.3)$ übersetzt werden. Durch Anwendung der obigen Reduktion erhält man aber sofort

$$\texttt{fix}(\lambda\texttt{x}.3) \ \to^* \ (\lambda\texttt{x}.3)(\texttt{fix}(\lambda\texttt{x}.3)) \ \to_\beta 3.$$

Definition 3.3.3 (Übersetzung von einfachem HASKELL in Lambda-Terme)
Sei Exp *die Menge der einfachen* HASKELL*-Ausdrücke, seien $\mathcal{C}_0$ die Symbole der in* Exp *auftretenden vordefinierten Operationen (d.h.* `+`*,* `not`*,* `sqrt`*, etc.) und sei Λ die Menge der Lambda-Terme über den Konstanten $\mathcal{C} = \mathcal{C}_0 \cup \mathsf{Con} \cup \{\texttt{tuple}_n \mid n = 0 \ \text{ oder } n \geq 2\} \cup \{\texttt{if}, \texttt{fix}\}$, wobei* Con *die Konstruktoren in den Ausdrücken* Exp *bezeichnet. Hierbei werden auch ganze*

[1] Die Funktion `fix` lässt sich unmittelbar in HASKELL implementieren: `fix x = x (fix x)` wobei `fix :: (a -> a) -> a`. Wenn man die zugehörige Higher-Order Funktion `ff` zu `fact` ebenfalls in HASKELL implementiert hat, kann man also nun `fact = fix ff` in HASKELL definieren (oder alternativ Ausdrücke wie `(fix ff) 3` auswerten, was die Fakultät `6` von `3` ergibt).

Zahlen und Gleitkommazahlen sowie Zeichen als (nullstellige) Konstruktoren angesehen. Dann definieren wir die Funktion $\mathcal{L}am : \mathsf{Exp} \to \Lambda$, *die jeden einfachen* HASKELL*-Ausdruck in einen Lambda-Term übersetzt. Hierbei sei* $c \in \mathcal{C}_0 \cup \mathsf{Con}$ *beliebig und* $n = 0$ *oder* $n \geq 2$.

$$
\begin{array}{lcl}
\mathcal{L}am(\underline{\mathsf{var}}) & = & \underline{\mathsf{var}} \\
\mathcal{L}am(c) & = & c \\
\mathcal{L}am((\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n)) & = & \mathtt{tuple}_n\, \mathcal{L}am(\underline{\mathsf{exp}}_1) \ldots \mathcal{L}am(\underline{\mathsf{exp}}_n) \\
\mathcal{L}am((\underline{\mathsf{exp}})) & = & \mathcal{L}am(\underline{\mathsf{exp}}) \\
\mathcal{L}am((\underline{\mathsf{exp}}_1\, \underline{\mathsf{exp}}_2)) & = & (\mathcal{L}am(\underline{\mathsf{exp}}_1)\, \mathcal{L}am(\underline{\mathsf{exp}}_2)) \\
\mathcal{L}am(\mathtt{if}\ \underline{\mathsf{exp}}_1\ \mathtt{then}\ \underline{\mathsf{exp}}_2\ \mathtt{else}\ \underline{\mathsf{exp}}_3) & = & \mathtt{if}\, \mathcal{L}am(\underline{\mathsf{exp}}_1)\, \mathcal{L}am(\underline{\mathsf{exp}}_2)\, \mathcal{L}am(\underline{\mathsf{exp}}_3) \\
\mathcal{L}am(\mathtt{let}\ \underline{\mathsf{var}} = \underline{\mathsf{exp}}\ \mathtt{in}\ \underline{\mathsf{exp}}') & = & \mathcal{L}am(\underline{\mathsf{exp}}')\ [\underline{\mathsf{var}} \,/\, (\mathtt{fix}\, (\lambda \underline{\mathsf{var}}.\mathcal{L}am(\underline{\mathsf{exp}})))] \\
\mathcal{L}am(\backslash \underline{\mathsf{var}}\ \texttt{->}\ \underline{\mathsf{exp}}) & = & \lambda \underline{\mathsf{var}}.\mathcal{L}am(\underline{\mathsf{exp}})
\end{array}
$$

Um die Auswertung eines Ausdrucks $\underline{\mathsf{exp}}$ in einem HASKELL-Programm mit den Deklarationen P durchzuführen, überführt man die globalen Deklarationen wieder (ähnlich wie bei der Definition der Semantik) in lokale Deklarationen. Man wertet also stattdessen den Ausdruck

$$\mathtt{let}\ P\ \mathtt{in}\ \underline{\mathsf{exp}}$$

(im leeren Programm) aus. Für diese Auswertung muss dieser HASKELL-Ausdruck zunächst in einen Lambda-Term übersetzt werden und anschließend kann man die WHNO-Reduktion des Lambda-Kalküls für die Auswertung benutzen. Sofern der obige Ausdruck bereits ein *einfacher* HASKELL-Ausdruck ist, kann man für diese Übersetzung direkt die Übersetzungsfunktion $\mathcal{L}am$ aus Def. 3.3.3 verwenden. Ansonsten muss man diesen Ausdruck zunächst in einen einfachen HASKELL-Ausdruck transformieren. Hierzu verwenden wir das Transformationsverfahren aus Def. 2.2.11, das wir ursprünglich zur Festlegung der Semantik von komplexen HASKELL-Programmen verwendet hatten. Da diese Transformation aber automatisch durchführbar ist, kann man sie auch zur Implementierung verwenden. Wir betrachten also stattdessen den Ausdruck

$$(\mathtt{let}\ P\ \mathtt{in}\ \underline{\mathsf{exp}})_{tr}.$$

Um die Effizienz der Implementierung zu verbessern, sollte man die Transformation des Programms P natürlich vor der eigentlichen Laufzeit (d.h. als *Compilierung* des Programms) vornehmen. Sei $P_1, \ldots, P_k$ eine Aufteilung von P, d.h., P_i enthält jeweils miteinander verschränkt rekursive Deklarationen. Bei der Transformation wird (beim Vorhandensein von mehreren Deklarationen in P_i) durch Regel (12) eine neue Variable $\underline{\mathsf{var}}_{P_i}$ eingeführt, die als Wert den Tupel der einzelnen in P_i deklarierten Funktionen bzw. Konstanten hat. Wenn man sich die Namen der Variablen $\underline{\mathsf{var}}_{P_1}, \ldots, \underline{\mathsf{var}}_{P_k}$ "merkt", muss zur Laufzeit nur noch der Ausdruck $\underline{\mathsf{exp}}$ transformiert werden. Wir werden hierauf in Abschnitt 4.3 (Satz 4.3.2) genauer eingehen.

Definition 3.3.4 (Übersetzung von HASKELL in den Lambda-Kalkül) *Sei* P *die Folge der Pattern- und Funktionsdeklarationen eines komplexen* HASKELL*-Programms und sei* $\underline{\mathsf{exp}}$ *ein komplexer Ausdruck, der keine freien Variablen außer den in* P *definierten und den in* HASKELL *vordefinierten Variablen enthält. Seien* $\mathcal{C}_0$ *die Symbole der in* HASKELL *vordefinierten Operationen (d.h.* `+`, `not`, `sqrt`, *etc.). Seien* Con *die Konstruktoren*

des HASKELL*-Programms. Dann definieren wir*

$$\begin{array}{rcl} \mathcal{C} & = & \mathcal{C}_0 \cup \\ & & \mathsf{Con} \cup \\ & & \{\mathtt{bot}, \mathtt{if}, \mathtt{fix}\} \cup \\ & & \{\mathtt{isa}_{n-\mathtt{tuple}} \,|\, n \in \{0, 2, 3, \ldots\}\} \cup \\ & & \{\mathtt{isa}_{\underline{\mathsf{constr}}} \,|\, \underline{\mathsf{constr}} \in \mathsf{Con}\} \cup \\ & & \{\mathtt{argof}_{\underline{\mathsf{constr}}} \,|\, \underline{\mathsf{constr}} \in \mathsf{Con}\} \cup \\ & & \{\mathtt{sel}_{n,i} \,|\, n \geq 2, 1 \leq i \leq n\} \cup \\ & & \{\mathtt{tuple}_n \,|\, n = 0 \;\; oder \;\; n \geq 2\}. \end{array}$$

Die Übersetzung des Ausdrucks $\underline{\mathsf{exp}}$ *in dem Programm* P *liefert einen Lambda-Term über der Menge der Konstanten* $\mathcal{C}$*. Sie ist definiert als*

$$\mathcal{T}ran(P, \underline{\mathsf{exp}}) \;=\; \mathcal{L}am(\,(\mathtt{let}\, P\, \mathtt{in}\, \underline{\mathsf{exp}})_{tr}\,).$$

Nun müssen wir noch angeben, welche δ-Regeln wir bei der Implementierung von HASKELL durch den Lambda-Kalkül verwenden. Die δ-Regeln für die in HASKELL vordefinierten Funktionssymbole müssen der jeweiligen Definition entsprechen. Die Semantik der vordefinierten Funktionssymbole `bot`, $\mathtt{isa}_{n-\mathtt{tuple}}$, $\mathtt{isa}_{\underline{\mathsf{constr}}}$, $\mathtt{argof}_{\underline{\mathsf{constr}}}$ und $\mathtt{sel}_{n,i}$ wurde bei der Festlegung der Semantik (bei der Definition des initialen Environments in Def. 2.2.8) angegeben. Die Regeln für `if` müssen so gewählt sein, dass es dem `if`-Konstrukt von HASKELL entspricht und die Regel für `fix` muss dem Fixpunktoperator entsprechen (siehe oben). Man beachte, dass diese δ-Regeln nur von der Menge der Konstruktoren des HASKELL-Programms abhängen, aber nicht von den Deklarationen des Programms.

Da wir im Folgenden ja jeweils nur Reduktionen bis zur WHNF (anstatt bis zur Normalform) betrachten, kann man nun auch δ-Regeln der Form $c\, t_1 \ldots t_n \to r$ zulassen, bei denen $t_1 \ldots t_n$ nur in WHNF (statt in Normalform) sind. Die Konfluenz der WHNO bleibt damit immer noch erhalten. Beispielsweise können wir also die Regeln $\mathtt{isa}_{\underline{\mathsf{constr}}}\,(\underline{\mathsf{constr}}\, t_1 \ldots t_n) \to$ `True` für beliebige geschlossene Terme $t_1, \ldots, t_n$ verwenden, denn $(\underline{\mathsf{constr}}\, t_1 \ldots t_n)$ ist immer in WHNF. Dies ist auch nötig, um HASKELL korrekt zu implementieren. Der Grund ist, dass beim Pattern Matching jeweils nur bis zur WHNF ausgewertet wird und anschließend in Abhängigkeit des obersten Funktionssymbols der richtige Fall ausgewählt wird. Da das Pattern Matching ja in Fallunterscheidungen mit Hilfe der eingebauten Funktionen wie $\mathtt{isa}_{\underline{\mathsf{constr}}}$ übersetzt wird, müssen diese analog arbeiten. Ansonsten würde bei einer Funktion wie

```
f Zero = Zero
f (Succ x) = Zero
```

die Auswertung des Ausdrucks `f (Succ bot)` nicht terminieren.

Definition 3.3.5 (δ-Regeln für HASKELL**-Programme)** *Sei* Con *die Menge der Konstruktoren eines* HASKELL*-Programms, wobei* Con_n *wieder die Menge aller Konstruktoren der Stelligkeit* n *bezeichnet. Seien* δ_0 *die Regeln für die in* HASKELL *vordefinierten Operationen, d.h.,* δ_0 *enthält Regeln wie* `plus 1 2` $\to$ `3`, `not True` $\to$ `False`*, etc. Die gesamte Menge*

δ ergibt sich wie folgt.

$$
\begin{array}{rcl}
\delta & = & \delta_0 \cup \\
& & \{\mathtt{bot} \to \mathtt{bot}, \\
& & \mathtt{if}\,\mathtt{True} \to \lambda xy.x, \\
& & \mathtt{if}\,\mathtt{False} \to \lambda xy.y, \\
& & \mathtt{fix} \to \lambda f.\, f(\mathtt{fix}\, f)\} \cup \\
& & \{\mathtt{isa}_{n-\mathtt{tuple}}\,(\mathtt{tuple}_n\, t_1 \,\ldots\, t_n) \to \mathtt{True} \mid n \in \{0,2,3,\ldots\}, \\
& & \qquad t_1,\ldots,t_n \in \Lambda,\ t_j \textit{ geschlossen}\} \cup \\
& & \{\mathtt{isa}_{\underline{\mathsf{constr}}}\,(\underline{\mathsf{constr}}\, t_1 \,\ldots\, t_n) \to \mathtt{True} \mid \underline{\mathsf{constr}} \in \mathsf{Con}_n,\ n \geq 0, \\
& & \qquad t_1,\ldots,t_n \in \Lambda,\ t_j \textit{ geschlossen}\} \cup \\
& & \{\mathtt{isa}_{\underline{\mathsf{constr}}}\,(\underline{\mathsf{constr}'}\, t_1 \,\ldots\, t_m) \to \mathtt{False} \mid \underline{\mathsf{constr}'} \in \mathsf{Con}_m,\ \underline{\mathsf{constr}} \neq \underline{\mathsf{constr}'},\ m \geq 0, \\
& & \qquad t_1,\ldots,t_m \in \Lambda,\ t_j \textit{ geschlossen}\} \cup \\
& & \{\mathtt{argof}_{\underline{\mathsf{constr}}}\,(\underline{\mathsf{constr}}\, t_1 \,\ldots\, t_n) \to \mathtt{tuple}_n\, t_1 \,\ldots\, t_n \mid \underline{\mathsf{constr}} \in \mathsf{Con}_n,\ n \in \{0,2,3,\ldots\} \\
& & \qquad t_1,\ldots,t_n \in \Lambda,\ t_j \textit{ geschlossen}\} \cup \\
& & \{\mathtt{argof}_{\underline{\mathsf{constr}}}\,(\underline{\mathsf{constr}}\, t) \to t \mid \underline{\mathsf{constr}} \in \mathsf{Con}_1, t \in \Lambda,\ t \textit{ geschlossen}\} \cup \\
& & \{\mathtt{sel}_{n,i}\,(\mathtt{tuple}_n\, t_1 \,\ldots\, t_n) \to t_i \mid n \geq 2,\ 1 \leq i \leq n,\ t_1,\ldots,t_n \in \Lambda,\ t_j \textit{ geschlossen}\}
\end{array}
$$

Die Menge δ ist zwar unendlich, sie lässt sich aber auf endliche Weise repräsentieren. Somit kann sie für automatische Reduktionen und damit für eine Implementierung von HASKELL verwendet werden. Man erkennt auch, dass δ nur von den Datenstrukturen, aber nicht von den Algorithmen eines Programms abhängt.

Schließlich erhalten wir die folgende Implementierung von HASKELL (deren Effizienz durch eine vorangehende Compilierung des Programms P wie erwähnt verbessert werden kann). Eine solche Compilierung wird in Abschnitt 4.3 vorgestellt.

Definition 3.3.6 (Implementierung von HASKELL) *Für ein komplexes HASKELL-Programm mit den Konstruktoren Con sei δ die zu diesen Konstruktoren gehörende Regelmenge. Sei P die Folge der Pattern- und Funktionsdeklarationen des Programms und sei $\underline{\mathsf{exp}}$ ein komplexer Ausdruck, der keine freien Variablen außer den in P definierten und den in HASKELL vordefinierten Variablen enthält. Die Auswertung des Ausdrucks $\underline{\mathsf{exp}}$ im Programm P geschieht dann durch Weak Head Normal Order Reduktion (bei Verwendung der obigen Delta-Regeln δ) des Lambda-Terms $\mathcal{T}ran(P, \underline{\mathsf{exp}})$.*

Man kann zeigen, dass diese Implementierung tatsächlich korrekt ist, d.h., dass sie der (denotationellen) Semantik von HASKELL entspricht. Hierzu müssen wir die semantische Funktion $\mathcal{V}al$ auch auf Lambda-Terme erweitern. Dies gelingt jedoch sehr einfach, indem man einfach $\mathcal{V}al[\![\lambda x.t]\!]\rho$ als $\mathcal{V}al[\![\backslash\mathtt{x}\,\mathtt{->}\,\mathtt{t}]\!]\rho$, $\mathcal{V}al[\![\mathtt{tuple}_n\, t_1 \ldots t_n]\!]\rho$ als $\mathcal{V}al[\![(t_1,\ldots,t_n)]\!]\rho$ und $\mathcal{V}al[\![\mathtt{fix}\, t]\!]\rho$ als $\mathrm{lfp}\,(\mathcal{V}al[\![t]\!]\rho)$ definiert. (Für eine formale Definition müsste man die Überführung von Lambda-Termen in HASKELL-Ausdrücke natürlich rekursiv definieren.)

Satz 3.3.7 (Korrektheit der Implementierung) *Seien P und $\underline{\mathsf{exp}}$ wie in Def. 3.3.6, wobei sowohl P als auch $\underline{\mathsf{exp}}$ typkorrekt sind. Falls $\mathcal{T}ran(P, \underline{\mathsf{exp}}) \to^* q$ für einen Lambda-Term q in WHNF, so gilt $\mathcal{V}al[\![(\mathtt{let}\, P\, \mathtt{in}\, \underline{\mathsf{exp}})_{tr}]\!]\omega_{tr} = \mathcal{V}al[\![q]\!]\omega_{tr}$. Falls die WHNO-Reduktion von $\mathcal{T}ran(P, \underline{\mathsf{exp}})$ nicht terminiert, so gilt $\mathcal{V}al[\![(\mathtt{let}\, P\, \mathtt{in}\, \underline{\mathsf{exp}})_{tr}]\!]\omega_{tr} = \bot$.*

Ein Beweis von Satz 3.3.7 ist recht aufwändig. Entsprechende Beweise finden sich jedoch in vielen Büchern über den Lambda-Kalkül bzw. über die Semantik von funktionalen Programmiersprachen, z.B. [LS87]. Alternativ hätte man natürlich auch HASKELL's Semantik über den obigen Interpreter definieren können. Solch eine Festlegung der Semantik bezeichnet man als *operationelle Semantik*. Satz 3.3.7 sagt dann aus, dass sich die denotationelle und die operationelle Semantik von HASKELL entsprechen.

Man beachte, dass die WHNO-Reduktion des Lambda-Terms $\mathcal{T}ran(P, \underline{\texttt{exp}})$ tatsächlich nur dann nicht terminiert, wenn die Semantik dieses Terms der vollkommen undefinierte Wert $\bot$ ist. Bei Ausdrücken, deren Wert partiell definiert ist, würde die WHNO-Reduktion terminieren, denn die entsprechenden Lambda-Terme sind bereits in WHNF. Einem partiell definierten Wert wie $(\bot, \bot)$ würde der Lambda-Term $\texttt{tuple}_2\, t_1\, t_2$ entsprechen, bei dem der Wert von t_1 und t_2 undefiniert ist. Analog verhält es sich bei einem partiell definierten Wert wie $(\texttt{Succ}, \bot)$, dem ein Lambda-Term $\texttt{Succ}\, t$ entspricht, bei dem der Wert von t undefiniert ist.

Die obige Implementierung realisiert Undefiniertheit immer durch Nicht-Terminierung der Auswertung. Dies passiert also z.B. auch, wenn man eine Funktion auswerten will, deren definierende Gleichungen nicht alle möglichen Eingaben abdecken (d.h., das Pattern Matching ist nicht vollständig), vgl. die Transformation des Pattern Matchings in Def. 2.2.11. Alternativ könnte man natürlich die δ-Regel für `bot` ändern und stattdessen eine entsprechende Fehlermeldung ausgeben. (Dies würde z.B. im HASKELL-Interpreter des GHC geschehen.)

3.4 Der reine Lambda-Kalkül

Der Lambda-Kalkül aus den vorangegangenen Abschnitten wird oft als *erweiterter* Lambda-Kalkül bezeichnet. Der Grund ist, dass es eine Teilmenge der Lambda-Terme gibt, mit denen man immer noch alle berechenbaren Funktionen darstellen kann. Im sogenannten *reinen* Lambda-Kalkül existieren keine Konstanten $\mathcal{C}$ mehr; die einzigen Terme sind somit Variablen, Anwendungen und Lambda-Abstraktionen. Im reinen Lambda-Kalkül gibt es auch keine δ-Reduktion mehr, da δ-Regeln ja nur aussagen, wie man mit Konstanten gebildete Terme reduzieren kann.

Im reinen Lambda-Kalkül müssen die benötigten Konstanten also auch als (reine) geschlossene Lambda-Terme repräsentiert werden. Beispielsweise kann man nicht-negative ganze Zahlen wie folgt darstellen. Ein Term $f^n x$, der eine Funktion f n-mal auf ein Argument anwendet, entspricht bereits in etwa der Zahl n. Wir abstrahieren von dem konkreten Argument x und der konkreten Funktion f und erhalten so den Term $\lambda f x. f^n\, x$, der n-faches Anwenden symbolisiert. Dieser Term wird als Repräsentation $\overline{n}$ der Zahl n verwendet. Man muss also jeweils angeben, wie die Datenobjekte einer Datenstruktur repräsentiert werden sollen. Anschließend kann man jede berechenbare Funktion als reinen Lambda-Term ausdrücken. Wir erhalten also

$$\begin{array}{rcl}\overline{\texttt{0}} & = & \lambda f x. x \\ \overline{\texttt{1}} & = & \lambda f x. f\, x \\ & \vdots & \end{array}$$

Der Term $\overline{\texttt{Succ}}$ für den Nachfolgerkonstruktor `Succ` muss so definiert werden, dass $\overline{\texttt{Succ}}\, \overline{n}$

$\rightarrow_\beta^* \ \overline{n+1}$ gilt. Man beachte, dass

$$\overline{n+1} \;=\; \lambda f x. f^{n+1}\, x \;=\; \lambda f x. f(f^n x) \;\leftarrow_\beta^*\; \lambda f x. f((\lambda f x. f^n\, x) f x) \;=\; \lambda f x. f(\overline{n}\, f\, x).$$

Also folgt

$$\overline{\texttt{Succ}} \;=\; \lambda n f x.\, f\,(n\, f\, x).$$

Betrachten wir nun die Realisierung von booleschen Werten und der Konstante `if`. Wenn man `True` und `False` als Auswahlfunktionen repräsentiert, die jeweils das erste bzw. das zweite Argument auswählen, wenn man sie auf zwei Argumente anwendet, so erhält man auch eine sehr einfache Repräsentation von `if`.

$$\begin{array}{lcl} \overline{\texttt{True}} & = & \lambda xy.x \\ \overline{\texttt{False}} & = & \lambda xy.y \\ \overline{\texttt{if}} & = & \lambda x.x \end{array}$$

Es gilt nämlich

$$\begin{array}{rcl} \overline{\texttt{if}}\,\overline{\texttt{True}}\, u\, v & = & (\lambda x.x)\,(\lambda xy.x)\, u\, v \\ & \rightarrow_\beta & (\lambda xy.x)\, u\, v \\ & \rightarrow_\beta^* & u. \end{array}$$

Auf analoge Weise kann man nun auch Konstanten wie $\texttt{isa}_{\texttt{Zero}}$, $\texttt{isa}_{\texttt{Succ}}$ etc. realisieren. Für $\texttt{tuple}_n$ verwenden wir

$$\begin{array}{lcl} \overline{\texttt{tuple}_n} & = & \lambda x_1 \ldots x_n f. f\, x_1 \ldots x_n \\ \overline{\texttt{sel}_{n,i}} & = & \lambda g. g(\lambda y_1 \ldots y_n. y_i) \end{array}$$

In der Tat gilt

$$\begin{array}{rcl} \overline{\texttt{sel}_{n,i}}(\overline{\texttt{tuple}_n}\, z_1 \ldots z_n) & \rightarrow_\beta^* & (\lambda g. g(\lambda y_1 \ldots y_n. y_i))(\lambda f. f\, z_1 \ldots z_n) \\ & \rightarrow_\beta & (\lambda f. f\, z_1 \ldots z_n)\,(\lambda y_1 \ldots y_n. y_i) \\ & \rightarrow_\beta & (\lambda y_1 \ldots y_n. y_i) z_1 \ldots z_n \\ & \rightarrow_\beta^* & z_i. \end{array}$$

Betrachten wir schließlich noch die Realisierung des Fixpunktoperators `fix`. Gesucht ist ein Term $\overline{\texttt{fix}}$, so dass $\overline{\texttt{fix}}\, z \ \rightarrow_\beta^* \ z\,(\overline{\texttt{fix}}\, z)$ gilt. Die Lösung ist der folgende *Fixpunktkombinator*:[2]

$$\overline{\texttt{fix}} \;=\; (\lambda xy. y(x\, x\, y))\,(\lambda xy. y(x\, x\, y)).$$

In der Tat gilt

$$\begin{array}{rcl} \overline{\texttt{fix}}\, z & = & (\lambda xy. y(x\, x\, y))\,(\lambda xy. y(x\, x\, y))\, z \\ & \rightarrow_\beta & (\lambda y. y\,((\lambda xy. y(x\, x\, y))\,(\lambda xy. y(x\, x\, y))\, y))\, z \\ & \rightarrow_\beta & z\,((\lambda xy. y(x\, x\, y))\,(\lambda xy. y(x\, x\, y))\, z) \\ & = & z\,(\overline{\texttt{fix}}\, z). \end{array}$$

[2] Dies ist der Turingsche Fixpunktkombinator Θ. Man verwendet oft stattdessen den Fixpunktkombinator $\lambda f.(\lambda x. f(x\, x))(\lambda x. f(x\, x))$, der mit Y bezeichnet wird. Hier gilt allerdings nur $Y\, y \leftrightarrow_\beta^* y\,(Y\, y)$.

Man sollte erwähnen, dass es neben dem hier vorgestellten *ungetypten* Lambda-Kalkül auch einen getypten Lambda-Kalkül gibt, in dem jeder Term einen eindeutigen Typ hat (ähnlich wie in HASKELL). Der Fixpunktkombinator $\overline{\texttt{fix}}$ ist darin allerdings nicht darstellbar, denn einem solchen Term kann kein korrekter Typ zugeordnet werden. Dieses Problem tritt stets auf, wenn ein (Teil-)Term auf sich selbst angewendet wird.

Man erkennt, dass Wahrheitswerte, natürliche Zahlen mit der Nachfolgerfunktion sowie der Test für die Zahl `0` im reinen Lambda-Kalkül darstellbar sind. Ebenso ist auch der Fixpunktkombinator darstellbar, so dass die unbeschränkte Minimalisierung realisierbar ist. Damit können also alle μ-rekursiven und damit alle berechenbaren Funktionen im reinen Lambda-Kalkül repräsentiert werden. Eine derartig reduzierte Sprache wie der reine Lambda-Kalkül ist also bereits eine vollwertige funktionale Programmiersprache und alle anderen funktionalen Sprachen sind nur syntaktische Varianten des reinen Lambda-Kalküls. Wir werden uns im Folgenden jedoch wieder auf den erweiterten Lambda-Kalkül konzentrieren, da dort die Lambda-Terme leichter lesbar sind und außerdem Rechnungen (d.h. die Implementierung von HASKELL) effizienter durchführbar sind. Der wesentliche Vorteil des erweiterten Lambda-Kalküls ist aber, dass man ihn auch zur Typüberprüfung verwenden kann (d.h., die Typen und die Typkorrektheit bleiben bei der Übersetzung von HASKELL in den erweiterten Lambda-Kalkül erhalten). Hingegen entstehen bei der Übersetzung von typkorrekten HASKELL-Programmen in den reinen Lambda-Kalkül Terme wie $\overline{\texttt{fix}}$, die nicht typkorrekt sind.

Kapitel 4

Typüberprüfung und -inferenz

Sowohl bei der Angabe der Semantik als auch bei der Implementierung von HASKELL sind wir stets davon ausgegangen, dass wir nur typkorrekte (oder "wohlgetypte") Ausdrücke betrachten. Wir wollen nun zeigen, wie man automatisch überprüfen kann, ob ein HASKELL-Programm bzw. ein HASKELL-Ausdruck korrekt getypt ist und wie man automatisch seinen Typ feststellen kann.

Eine solche *statische* Typüberprüfung (zur Compilezeit statt zur Laufzeit) ist in allen HASKELL-Interpretern und Compilern implementiert. Ein Vorteil statisch getypter Sprachen ist, dass viele Programmierfehler bereits vor der Ausführung des Programms gefunden werden können. Außerdem hat dies Effizienzvorteile: In solchen Sprachen kann man auf eine Typüberprüfung zur Laufzeit verzichten, da durch die statische Typkorrektheit bereits zur Compilezeit garantiert werden kann, dass bei keinem Ablauf des Programms jemals ein Typfehler auftreten kann.

Der Einfachheit halber werden wir den Typinferenzalgorithmus nur für den Lambda-Kalkül (statt für HASKELL) angeben. Hierzu stellen wir zunächst in Abschnitt 4.1 die benötigten Konzepte vor und diskutieren den Typinferenzalgorithmus in Abschnitt 4.2. Eine Implementierung von HASKELL ist dann damit wie folgt möglich, vgl. Abschnitt 4.3: Zunächst übersetzt man HASKELL in einfaches HASKELL mit der Transformation von Def. 2.2.11 und dann weiter in den Lambda-Kalkül (Def. 3.3.3). Anschließend überprüft man zunächst die Typkorrektheit (dies ist äquivalent zur Typkorrektheit des ursprünglichen HASKELL-Programms). Danach kann der entstandene Lambda-Term mit WHNO-Reduktion ausgewertet werden (Def. 3.3.6).

4.1 Typschemata und Typannahmen

Wie in Abschnitt 1.1.4 über die Form von Typen in HASKELL erläutert, werden Typen mit Hilfe von *Typkonstruktoren* aus anderen Typen gebildet. In HASKELL sind bereits die nullstelligen Typkonstruktoren `Int`, `Bool`, `Float` und `Char` vordefiniert. Außerdem gibt es den zweistelligen Typkonstruktor $\rightarrow$ und einen beliebigstelligen Typkonstruktor zum Aufbau von Typtupeln. Schließlich werden durch benutzerdefinierte Datentypen beliebig viele weitere Typkonstruktoren wie `Nats` (nullstellig), `List` (einstellig), etc. eingeführt. Wie bisher gehen wir davon aus, dass anstelle der in HASKELL vordefinierten Listen die vom Benutzer definierten Listen verwendet werden.

Die Schwierigkeit der Typüberprüfung liegt daran, dass wir *polymorphe* Typen verwenden. Wir haben also auch Typvariablen a, die für beliebige Typen stehen können und die ebenfalls mit Hilfe von Typkonstruktoren zum Aufbau komplexerer Typen verwendet werden können. Jeder Typ, der Typvariablen enthält, heißt polymorph. Polymorphismus hat den Vorteil, dass man damit generische Funktionen definieren kann, die die gleiche Operation auf vielen gleichartigen Typen durchführen können.

Damit kann ein Ausdruck wie `Nil` aber nun viele Typen haben (z.B. `List` a, `List Int`, `List (List Bool)`, etc.). Allerdings hat jeder Ausdruck einen eindeutigen *allgemeinsten* Typ (bis auf Umbenennung der darin vorkommenden Typvariablen). Der allgemeinste Typ von `Nil` ist `List` a. Hierbei bedeutet "allgemeinster Typ", dass dieser Ausdruck damit auch jeden Typ hat, der durch Instantiierung der Typvariablen a durch beliebige andere Typen entsteht. Um dies deutlich zu machen, ist es hilfreich, die Allquantifizierung der Typvariable a explizit zu machen. Wir sprechen dann von sogenannten *Typschemata*. Das Typschema von `Nil` ist also $\forall a.\, \texttt{List}\ a$. Das Ziel der Typinferenz ist es, für jeden Ausdruck (bzw. jeden Lambda-Term) zu überprüfen, ob ihm ein korrekter Typ zugeordnet werden kann und ggf. den allgemeinsten Typ (bzw. das allgemeinste Typschema) zu berechnen.

Wie bisher gehen wir davon aus, dass es bereits vordefinierte Funktionen (d.h. Konstanten $\mathcal{C}$ des Lambda-Kalküls) gibt, deren Semantik und damit auch deren Typ bereits festliegt. Diese Festlegungen sammeln wir in einer sogenannten *Typannahme* (type assumption) A, die jeder Konstanten aus $\mathcal{C}$ und auch jeder Variablen aus $\mathcal{V}$ das zugehörige Typschema zuordnet. Typannahmen sind insofern ähnlich wie die Environments, die wir zur Definition der Semantik verwendet hatten. Während ein Environment jedoch Variablen mit Objekten des Domains belegt, ordnen wir nun den Variablen und Konstanten bestimmte Typschemata zu.

Um herauszufinden, ob ein bestimmter Ausdruck korrekt getypt ist, gehen wir von einer bestimmten zugrunde liegenden *initialen* Typannahme A_0 aus. So sind Variablen prinzipiell für alle Typen verwendbar, d.h., sie haben das Typschema $\forall a.\, a$.

Die vordefinierten Operationen $\mathcal{C}_0$ von HASKELL haben die dort bereits festgelegten Typen, wobei alle freien Variablen wieder allquantifiziert werden. Wir haben also z.B. $A_0(\texttt{not}) = \texttt{Bool} \to \texttt{Bool}$. Da wir der Einfachheit halber auf die Betrachtung der Typklassen von HASKELL verzichten, gehen wir davon aus, dass es verschiedene Symbole `plus` gibt, um die Addition auf ganzen Zahlen bzw. auf Gleitkommazahlen zu bezeichnen, etc. Hierbei steht `plus` wieder für `(+)`, etc. Für die Version auf den ganzen Zahlen gilt dann $A_0(\texttt{plus}) = \texttt{Int} \to \texttt{Int} \to \texttt{Int}$ und $A_0(\texttt{5}) = \texttt{Int}$.

Bei benutzerdefinierten Konstruktoren ergibt sich das zugehörige Typschema aus der Datenstrukturdeklaration. Bei

```
data List a = Nil | Cons a  (List a)
```

hätte man daher $A_0(\texttt{Nil}) = \forall a.\, \texttt{List}\, a$ und $A_0(\texttt{Cons}) = \forall a.\, a \to (\texttt{List}\, a) \to (\texttt{List}\, a)$. Für die vordefinierten Funktionen, die bei der Übersetzung von komplexem in einfaches HASKELL entstehen, haben wir die Typen bereits in Def. 2.2.8 angegeben und für die weiteren Konstanten `if`, `fix` und $\texttt{tuple}_n$ des Lambda-Kalküls sind die Typschemata offensichtlich.

Definition 4.1.1 (Typschemata und Typannahmen) *Ein* Typschema *wird anhand*

der folgenden Grammatik gebildet:

$$
\begin{array}{lll}
\underline{\mathsf{typeschema}} & \rightarrow & (\underline{\mathsf{tyconstr}}\ \underline{\mathsf{typeschema}}_1\ \ldots\ \underline{\mathsf{typeschema}}_n), \quad \textit{wobei } n \geq 0 \\
 & | & (\underline{\mathsf{typeschema}}_1 \texttt{ -> } \underline{\mathsf{typeschema}}_2) \\
 & | & (\underline{\mathsf{typeschema}}_1, \ldots, \underline{\mathsf{typeschema}}_n), \quad \textit{wobei } n \geq 0 \\
 & | & \underline{\mathsf{var}} \\
 & | & \forall \underline{\mathsf{var}}.\ \underline{\mathsf{typeschema}}
\end{array}
$$

Ein Typschema *ist also ein Typ* $\underline{\mathsf{type}}$*, wobei aber Typvariablen allquantifiziert werden können. Für ein Typschema* τ *mit den freien Variablen* $a_1, \ldots, a_n$ *bezeichnet* $\forall\tau$ *das Typschema* $\forall a_1 \ldots \forall a_n.\tau$*.*

Eine Typannahme A *ist eine (möglicherweise partielle) Funktion von* $\mathcal{V} \cup \mathcal{C}$ *in die Menge der Typschemata. Eine Typannahme* A *mit* $A(x_i) = \tau_i$ $(1 \leq i \leq n)$*, die auf anderen Argumenten undefiniert ist, wird auch als* $\{x_1 :: \tau_1, \ldots, x_n :: \tau_n\}$ *geschrieben.*

Die initiale *Typannahme* A_0 *ist wie folgt definiert. Hierbei sei* $\underline{\mathsf{constr}}$ *ein benutzerdefinierter Konstruktor, der durch die Datenstrukturdeklaration*

$$\texttt{data}\ \underline{\mathsf{tyconstr}}\, a_1 \ldots a_m = \ \ldots |\underline{\mathsf{constr}}\ \underline{\mathsf{type}}_1 \ldots \underline{\mathsf{type}}_n| \ldots$$

eingeführt wird.

$$
\begin{array}{rcl}
A_0(x) & = & \forall a.\, a \ \ \textit{für alle } x \in \mathcal{V} \\
A_0(c) & = & \textit{der in } \textsc{Haskell} \textit{ vordefinierte Typ, wobei alle freien Va-} \\
 & & \textit{riablen allquantifiziert werden, für alle } c \in \mathcal{C}_0 \\
A_0(\underline{\mathsf{constr}}) & = & \forall\ (\underline{\mathsf{type}}_1 \rightarrow \ldots \rightarrow \underline{\mathsf{type}}_n \rightarrow (\underline{\mathsf{tyconstr}}\, a_1 \ldots a_m)), \\
A_0(\texttt{bot}) & = & \forall a.\, a \\
A_0(\texttt{if}) & = & \forall a.\, \texttt{Bool} \rightarrow a \rightarrow a \rightarrow a \\
A_0(\texttt{fix}) & = & \forall a.\, (a \rightarrow a) \rightarrow a \\
A_0(\texttt{isa}_{\underline{\mathsf{constr}}}) & = & \forall\ ((\underline{\mathsf{tyconstr}}\, a_1 \ldots a_m) \texttt{ -> Bool}) \\
A_0(\texttt{argof}_{\underline{\mathsf{constr}}}) & = & \forall\ ((\underline{\mathsf{tyconstr}}\, a_1 \ldots a_m) \texttt{ -> } (\underline{\mathsf{type}}_1, \ldots, \underline{\mathsf{type}}_n)) \\
A_0(\texttt{isa}_{n-\texttt{tuple}}) & = & \forall a_1 \ldots a_n.\ (a_1, \ldots, a_n) \rightarrow \texttt{Bool} \\
A_0(\texttt{sel}_{n,i}) & = & \forall a_1 \ldots a_n.\ (a_1, \ldots, a_n) \rightarrow a_i \\
A_0(\texttt{tuple}_n) & = & \forall a_1 \ldots a_n.\ a_1 \rightarrow \ldots \rightarrow a_n \rightarrow (a_1, \ldots, a_n)
\end{array}
$$

Für zwei Typannahmen A *und* A' *definieren wir* $A + A'$ *als* $(A + A')(x) = A'(x)$*, falls* $A'(x)$ *definiert ist und als* $A(x)$ *sonst.*

4.2 Der Typinferenzalgorithmus

Die Typüberprüfung geschieht *unter einer bestimmten Typannahme* A. Wir entwickeln daher nun einen Typinferenzalgorithmus $\mathcal{W}$, der als Eingabe die aktuelle Typannahme A und den auf Typkorrektheit zu untersuchenden Term t bekommt. Falls t typkorrekt ist, so ist das Ergebnis von $\mathcal{W}$ ein Paar (θ, τ). Hierbei bezeichnet τ den allgemeinsten Typ von t und θ ist eine Substitution für die freien Variablen in der Typannahme A. Die Bedeutung hiervon ist, dass t nur dann korrekt getypt ist, falls die Typannahme A zu $\theta(A)$ verfeinert wird. Nur

in diesem Fall hat der Term t tatsächlich den allgemeinsten Typ τ. (Wir schreiben häufig auch "$A\,\theta$" statt "$\theta(A)$".)

In den Abschnitten 4.2.1 - 4.2.3 erläutern wir zunächst das Prinzip der Typinferenz bei Lambda-Termen verschiedener Bauart. Anschließend geben wir in Abschnitt 4.2.4 den kompletten Typinferenzalgorithmus an.

4.2.1 Typinferenz bei Variablen und Konstanten

Betrachten wir zunächst die Typprüfung von Variablen und Konstanten. Wenn die Typannahme für eine Konstante $c :: \forall a_1, \ldots, a_n.\, \tau$ lautet (wobei τ ein Typ ist, d.h., τ enthält keine Quantoren mehr), so ist der allgemeinste Typ von c der Typ τ, wobei man die bislang allquantifizierten Variablen $a_1, \ldots, a_n$ durch neue Variablen $b_1, \ldots, b_n$ ersetzt, die nicht frei in A oder τ vorkommen. Wir schreiben "$\tau[a_1/b_1, \ldots, a_n/b_n]$" für die Anwendung dieser Substitution auf den Typ τ. (Diese Ersetzung wird vorgenommen, um Konflikte mit ggf. anderen ebenfalls frei vorkommenden Variablen in A zu vermeiden.) Diese Typinferenz ist immer korrekt, d.h., hierfür muss die bisherige Typannahme nicht durch eine weitere Substitution verfeinert werden. Die Substitution θ ist daher hier die Identität id. Analog verhält es sich bei Variablen aus $\mathcal{V}$. So erhalten wir beispielsweise

$$\begin{array}{rcl}\mathcal{W}(A_0, x) & = & (id, b) \text{ für } x \in \mathcal{V}\\ \mathcal{W}(A_0, \texttt{not}) & = & (id, \texttt{Bool} \to \texttt{Bool})\\ \mathcal{W}(A_0, \texttt{Cons}) & = & (id, b \to (\texttt{List}\, b) \to (\texttt{List}\, b))\end{array}$$

Allgemein ergibt sich die folgende Regel für alle $c \in \mathcal{V} \cup \mathcal{C}$:

$$\boxed{\mathcal{W}(\, A + \{c :: \forall a_1, \ldots, a_n.\, \tau\},\ c\,) = (\, id,\ \tau[a_1/b_1, \ldots, a_n/b_n]\,),\ b_1, \ldots, b_n \text{ neue Variablen}}$$

4.2.2 Typinferenz bei Lambda-Abstraktionen

Als nächstes betrachten wir die Typinferenz bei der Lambda-Abstraktion. Generell ist die Idee, für eine Lambda-Abstraktion $\lambda x.t$ zunächst den Typ des Terms t zu bestimmen. Hierbei kann natürlich die Variable x selbst in t auftreten. Wir nehmen daher an, dass x einen (beliebigen) Typ b hat und bestimmen unter dieser Annahme den Typ τ des Terms t. Der gesamte Term $\lambda x.t$ hat dann den Typ $b \to \tau$.

Im allgemeinen sind hierbei jedoch einige Komplikationen möglich. Hierzu untersuchen wir zunächst als Beispiel den folgenden Term:

$$\lambda f.\, \texttt{plus}\,(f\,\texttt{True})\,(f\,\texttt{3}).$$

Wenn dieser Term korrekt getypt wäre, müsste es möglich sein, f sowohl auf ein Argument vom Typ `Bool` als auch vom Typ `Int` anzuwenden (und das Ergebnis müsste jeweils vom Typ `Int` sein). Das bedeutet, dass f in diesem Term ein Typschema besitzen muss, das eine Anwendung auf *beliebige* Argumenttypen a erlaubt. Das Typschema von f wäre also $\forall a.\, a \to \texttt{Int}$. Damit darf man für f aber nur solche Terme einsetzen, die für jede Wahl von a richtig getypt sind. Ein Term mit dem Typ $\forall a.\, a \to \texttt{Int}$ wäre z.B. $\lambda x.\, \texttt{1}$, denn diesen Term kann man auf Argumente beliebigen Typs anwenden und das Ergebnis ist stets `1` (vom Typ `Int`). Der Typ des gesamten Terms ist daher $(\forall a.\, a \to \texttt{Int}) \to \texttt{Int}$. Typschemata von solch einer Form heißen *nicht-flach*.

Definition 4.2.1 (Flache Typschemata) *Ein Typschema heißt* flach *(engl.* shallow*) gdw. es von der Form* $\forall a_1 \dots a_n.\, \tau$ *ist und* τ *keine Quantoren enthält (d.h.,* τ *ist ein Typ).*

In Programmiersprachen beschränkt man sich meist auf flache Typschemata, d.h., man betrachtet einen Term nur dann als korrekt getypt, wenn ihm ein flaches Typschema zugeordnet werden kann. Durch diese Einschränkung wird die Frage der Typinferenz entscheidbar. Der Term $\lambda f.\ \mathtt{plus}\,(f\,\mathtt{True})\,(f\,3)$ ist bei einer Einschränkung auf flache Typschemata nicht korrekt getypt (und in der Tat würde dieser Term von HASKELL als nicht typkorrekt zurückgewiesen werden).

Das nicht-flache Typschema $(\forall a.\, a \to \mathtt{Int}) \to \mathtt{Int}$ ist grundsätzlich verschieden von dem flachen Typschema $\forall a.\, (a \to \mathtt{Int}) \to \mathtt{Int}$. Hätte der obige Term dieses flache Typschema, so hätte er auch gleichzeitig jeden Typ, der sich durch Instantiierung der allquantifizierten Variablen a ergibt. Beispielsweise hätte er also auch den Typ $(\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int}$. Das bedeutet, dass die Anwendung von $\lambda f.\ \mathtt{plus}\,(f\,\mathtt{True})\,(f\,3)$ auf Terme vom Typ $\mathtt{Int} \to \mathtt{Int}$ zu keinem Typfehler führen dürfte. Dies ist aber nicht der Fall, wie die Anwendung dieses Terms auf den Term `square` (vom Typ $\mathtt{Int} \to \mathtt{Int}$) zeigt. Hierbei entsteht nämlich der nicht typkorrekte Teilterm `square True`. Ein Beispiel für einen Term, der wirklich das flache Typschema $\forall a.\, (a \to \mathtt{Int}) \to \mathtt{Int}$ hat, ist z.B. $\lambda f.\ \mathtt{plus}\,(f\,\mathtt{bot})\,(f\,\mathtt{bot})$ (bei der Typannahme $\mathtt{bot} :: \forall a.\, a$).

Dieses Beispiel macht auch deutlich, dass die explizite Quantifizierung der Typvariablen (d.h. die Betrachtung von Typ*schemata*) während der Typinferenz sehr nützlich ist, da man ansonsten flache und nicht-flache Typschemata nicht unterscheiden könnte.

Die Einschränkung auf flache Typschemata bedeutet, dass jede Lambda-Abstraktion $\lambda x.\, t$ ein Typschema der Form $\forall\,(\tau_1 \to \tau_2)$ hat, wobei τ_1 der Typ von x und τ_2 der Typ von t ist (unter der Annahme, dass x den Typ τ_1 hat). Insbesondere muss damit also x *überall in* t den gleichen Typ τ_1 haben, d.h., die Typvariablen in τ_1 müssen überall im Term $\lambda x.\, t$ gleich instantiiert werden. (Ansonsten müsste man in τ_1 eine allquantifizierte Typvariable haben – dann wäre aber $\forall\,(\tau_1 \to \tau_2)$ kein flaches Typschema.)

Damit ist solch eine Lambda-Abstraktion dann auch auf jeden Term r des Typs τ_1 (bzw. einer Instanz des Typs τ_1) anwendbar. In solch einem Term r sind die Variablen aus τ_1 auf eine bestimmte Weise instantiiert (dies liegt an den flachen Typschemata). Die Typkorrektheit muss garantieren, dass auch der durch die Anwendung entstehende Term $t[x/r]$ typkorrekt ist. Hier werden in der Tat alle Vorkommen von x durch den gleichen Term r ersetzt, d.h., alle Wahlmöglichkeiten, die der Typ von x zulässt, werden überall im Term t auf die gleiche Weise festgelegt.

Bei der Typinferenz muss man zwischen der Bindung von Variablen durch λ und durch Deklarationen unterscheiden. Um diesen Unterschied zu verdeutlichen, betrachten wir zum Vergleich einmal den HASKELL-Ausdruck

```
let f = \x -> 1 in plus (f True) (f 3).
```

Dieser Ausdruck ist im Gegensatz zu $\lambda f.\ \mathtt{plus}\,(f\,\mathtt{True})\,(f\,3)$ korrekt getypt. Durch die Deklaration `f = \x -> 1` erhält `f` den Typ $\forall a.\, a \to \mathtt{Int}$. Nun kann im Rumpf des `let`-Ausdrucks die Typvariable a jeweils unterschiedlich (einmal mit `Bool` und einmal mit `Int`) instantiiert werden. Ebenso würde es sich verhalten, wenn `f = \x -> 1` eine vordefinierte

Funktion wäre. Dann würde in der initialen Typannahme bereits $A_0(\mathtt{f}) = \forall a.\, a \to \mathtt{Int}$ gelten und der Term $\mathtt{plus}\,(\mathtt{f}\,\mathtt{True})\,(\mathtt{f}\,3)$ wäre bei dieser Typannahme korrekt getypt.

Um Typinferenzen für `let`-Ausdrücke durchzuführen, übersetzen wir solche HASKELL-Ausdrücke zunächst in Lambda-Terme. Wenn wir der Einfachheit halber berücksichtigen, dass `f` hier nicht rekursiv definiert ist, dann wird der obige `let`-Ausdruck übersetzt in

$$(\mathtt{plus}\,(\mathtt{f}\,\mathtt{True})\,(\mathtt{f}\,3))\,[\mathtt{f}/\lambda\mathtt{x}.\,1],$$

d.h. in

$$\mathtt{plus}\,((\lambda\mathtt{x}.\,1)\,\mathtt{True})\,((\lambda\mathtt{x}.\,1)\,3).$$

Dieser Term ist korrekt getypt, wobei $\lambda\mathtt{x}.\,1$ das allgemeinste Typschema $\forall a.\, a \to \mathtt{Int}$ hat und die Variable a an den beiden Stellen des Terms unterschiedlich instantiiert wird (einmal mit `Bool` und einmal mit `Int`).

Zusammenfassend lässt sich also sagen: Variablen, die durch λ gebunden werden, haben zwar (zunächst) einen beliebigen Typ, aber dieser Typ muss überall im Rumpf der Lambda-Abstraktion derselbe sein. Variablen, die durch Deklarationen gebunden werden, können hingegen anschließend verschiedene Typen haben. Die Typvariablen in solchen Typen bezeichnet man auch als "generische" Variablen. (Dies zeigt sich bei unserer Übersetzung in den Lambda-Kalkül darin, dass die lokalen Deklarationen an die entsprechenden Stellen im Term kopiert werden, so dass an diesen Stellen die jeweiligen Wahlmöglichkeiten bei der Typbestimmung unterschiedlich festgelegt werden können.) Dies ist auch der Grund, warum wir in Def. 3.3.3 einen (nicht-rekursiven) `let`-Ausdruck `let` $\underline{\mathsf{var}}$ `=` $\underline{\mathsf{exp}}$ `in` $\underline{\mathsf{exp}}'$ nicht in $(\lambda\underline{\mathsf{var}}.\underline{\mathsf{exp}}')\,\underline{\mathsf{exp}}$ übersetzt haben. Dann müssten nämlich alle Vorkommen von $\underline{\mathsf{var}}$ in $\underline{\mathsf{exp}}'$ den gleichen Typ haben. Damit würde aber der typkorrekte Ausdruck `let f = \x -> 1 in plus (f True) (f 3)` in den nicht typkorrekten Term $(\lambda\mathtt{f}.\mathtt{plus}\,(\mathtt{f}\,\mathtt{True})\,(\mathtt{f}\,3))\,(\lambda\mathtt{x}.\,1)$ übersetzt werden. Unsere Übersetzung von einfachen HASKELL-Ausdrücken in den Lambda-Kalkül ist hingegen "typerhaltend" (d.h., sie überführt typkorrekte HASKELL-Ausdrücke in typkorrekte Lambda-Terme des gleichen Typs und nicht typkorrekte HASKELL-Ausdrücke in nicht typkorrekte Lambda-Terme).

Nachdem wir die Einschränkung auf flache Typschemata motiviert haben, entwickeln wir nun die Regel, um Typinferenz für Lambda-Abstraktionen durchzuführen. Als Beispiel betrachten wir den Term $\lambda x.\, \mathtt{Cons}\,0\,(\mathtt{Cons}\,x\,x)$. Die Variable x kann hier zunächst einen beliebigen Typ b haben. Das bedeutet aber, dass sie denselben Typ b auch an allen Vorkommen im Term $\mathtt{Cons}\,0\,(\mathtt{Cons}\,x\,x)$ haben muss. Man muss also nun eine Typinferenz für den Term $\mathtt{Cons}\,0\,(\mathtt{Cons}\,x\,x)$ durchführen, wobei man als zusätzliche Typannahme $\{x :: b\}$ verwendet. Um $\mathcal{W}(\,A,\ \lambda x.\ \mathtt{Cons}\,0\,(\mathtt{Cons}\,x\,x)\,)$ zu berechnen, muss man also zunächst $\mathcal{W}(\,A + \{x :: b\},\ \mathtt{Cons}\,0\,(\mathtt{Cons}\,x\,x)\,)$ bestimmen. Mit dieser Typannahme lässt sich $\mathtt{Cons}\,0\,(\mathtt{Cons}\,x\,x)$ jedoch nicht korrekt typisieren. Man beachte, dass hierbei wieder die Verwendung von Quantoren wichtig ist. Hätte man die initiale Typannahme $\{x :: \forall b.\, b\}$ für die Variable x gehabt, wäre eine Typisierung des Terms $\mathtt{Cons}\,0\,(\mathtt{Cons}\,x\,x)$ problemlos möglich gewesen. Dann würde man aber insgesamt für den gesamten Term das nicht-flache Typschema $(\forall b.\, b) \to (\mathtt{List}\,\mathtt{Int})$ erhalten. Bei der Einschränkung auf flache Typschemata ist dieser Term hingegen nicht typkorrekt. Die Typannahme $\{x :: b\}$ bedeutet also, dass man den Typ von x beliebig instantiieren darf, aber er muss überall gleich instantiiert werden. Die Typannahme $\{x :: \forall b.\, b\}$ hingegen bedeutet, dass man den Typ von x überall beliebig und unterschiedlich instantiieren darf.

Als nächstes Beispiel betrachten wir $\lambda x.\, \texttt{tuple}_2\, x\, x$. Hier beginnen wir wieder mit der Typannahme $\{x :: b\}$. Um $\mathcal{W}(\, A,\ \lambda x.\, \texttt{tuple}_2\, x\, x\,)$ zu berechnen, muss man also zunächst $\mathcal{W}(\, A{+}\{x :: b\},\ \texttt{tuple}_2\, x\, x\,)$ bestimmen. Unter dieser Typannahme hat der Term $\texttt{tuple}_2\, x\, x$ den Typ (b, b) (d.h. $\mathcal{W}(\, A + \{x :: b\},\ \texttt{tuple}_2\, x\, x\,) = (\, id,\ (b, b)\,)$). Für den Gesamtterm erhält man also den Typ $b \to (b, b)$ bzw. das Typschema $\forall b.\, b \to (b, b)$. Man erkennt, dass die Wahl für den Typ b bei der Argumentvariable auch den Typ des Ergebnis bestimmt. Die Motivation hierfür ist wiederum, dass eine Anwendung des Terms $\lambda x.\, \texttt{tuple}_2\, x\, x$ auf z.B. ein Argument `1` des Typs `Int` die Belegung der Typvariablen b überall im entstehenden Term festlegt. Das Ergebnis der Anwendung $\texttt{tuple}_2\, \texttt{1}\, \texttt{1}$ hätte tatsächlich den Typ $(\texttt{Int}, \texttt{Int})$.

Auf diese Weise erkennt unser Typinferenzalgorithmus auch, dass der Term $\lambda f.\ \texttt{plus}\ (f\, \texttt{True})\, (f\, \texttt{3})$ nicht typkorrekt ist. Um den allgemeinsten Typ dieses Terms zu ermitteln, ordnen wir zunächst der Variablen f wieder einen möglichst allgemeinen Typ (d.h. eine Typvariable b) zu. Dieser Typ kann im Lauf der weiteren Typüberprüfung dieses Terms weiter eingeschränkt werden. Das bedeutet, dass die Variable b während der weiteren Typinferenz dieses Terms instantiiert werden kann. Wichtig dabei ist aber, dass diese Instantiierung überall in dieser Lambda-Abstraktion dieselbe ist. Durch den Teilterm $(f\, \texttt{True})$ erkennt man, dass f eine Funktion mit dem Argumenttyp `Bool` sein muss und da dies ein Argument für `plus` ist, muss das Resultat der f-Anwendung vom Typ `Int` sein. Die Variable b muss also mit $\texttt{Bool} \to \texttt{Int}$ instantiiert werden. Diese Instantiierung muss dann aber in der gesamten Lambda-Abstraktion verwendet werden. Das führt zum Typkonflikt beim Teilterm $f\, \texttt{3}$.

Betrachten wir schließlich ein weiteres Beispiel, bei dem der Typ der Variable x während der Typinferenz weiter eingeschränkt wird. Um $\mathcal{W}(A, \lambda x.\, \texttt{plus}\, x\, x)$ zu berechnen, berechnet man wieder zunächst $\mathcal{W}(\, A{+}\{x :: b\},\ \texttt{plus}\, x\, x\,)$. Der Term $\texttt{plus}\, x\, x$ hat den Typ `Int`, aber er lässt sich nur auf diese Art korrekt typisieren, falls die Typvariable b mit `Int` instantiiert wird. Man erhält also $\mathcal{W}(\, A + \{x :: b\},\ \texttt{plus}\, x\, x\,) = (\, [b/\texttt{Int}],\ \texttt{Int}\,)$. Der gesamte Term hat dann den Typ $b \to \texttt{Int}$, wobei aber die für die Typisierung des Rumpfs verwendete Instantiierung angewendet werden muss, d.h., man erhält

$$\mathcal{W}(\, A,\ \lambda x.\, \texttt{plus}\, x\, x\,) = (\, [b/\texttt{Int}],\ (b \to \texttt{Int})[b/\texttt{Int}]\,) = (\, [b/\texttt{Int}],\ \texttt{Int} \to \texttt{Int}\,).$$

Allgemein ergibt sich folgende Regel für die Typinferenz bei Lambda-Abstraktionen. Um den Typ von $\lambda x.\, t$ unter der Annahme A zu bestimmen, berechnet man zunächst den Typ von t unter derselben Annahme, wobei man jedoch die zusätzliche Annahme $\{x :: b\}$ trifft. Das Ergebnis von $\mathcal{W}(\, A + \{x :: b\},\ t\,)$ sei (θ, τ). Dies bedeutet, dass t den Typ τ hat, vorausgesetzt, dass man die bisherige Annahme $A + \{x :: b\}$ zu $(A + \{x :: b\})\theta$ verfeinert. Insbesondere hat x also den Typ $b\theta$. Der gesamte Term $\lambda x.\, t$ hat demnach den Typ $b\theta \to \tau$, falls die bisherige Annahme A zu $A\theta$ verfeinert wird. Man erhält:

$$\boxed{\mathcal{W}(\, A,\ \lambda x.\, t\,) = (\, \theta,\ b\theta \to \tau\,), \quad \text{wobei} \ \ \mathcal{W}(\, A + \{x :: b\},\ t\,) = (\theta, \tau),\ b \text{ ist neue Variable}}$$

4.2.3 Typinferenz bei Applikationen

Wir zeigen nun, wie der Typinferenzalgorithmus bei Applikationen arbeitet. Wenn wir in einem Term $(t_1\, t_2)$ wissen, dass t_1 den Typ τ_1 und t_2 den Typ τ_2 hat, so ist der Term $(t_1\, t_2)$ nur dann richtig getypt, wenn τ_1 einem Typ der Form $(\tau_2 \to \tau_3)$ “entspricht”. In diesem Fall

hat $t_1\, t_2$ den Typ τ_3. Beispielsweise ist der Term `not True` unter der initialen Typannahme richtig getypt, denn `not` hat den Typ $\tau_1 = \texttt{Bool} \to \texttt{Bool}$ und `True` hat den Typ $\tau_2 = \texttt{Bool}$. Aus der Bedingung $\tau_1 = \tau_2 \to \tau_3$ folgt $\tau_3 = \texttt{Bool}$. Dieser Term ist also typkorrekt (und er hat den Typ `Bool`), während der Term `not 3` nicht richtig getypt ist.

Die "Entsprechung" von τ_1 und $\tau_2 \to \tau_3$ kann aber natürlich auch die Instantiierung von Typvariablen bedeuten. Beispielsweise ist der Term `Cons 0` richtig getypt. Wir erhalten $\mathcal{W}(A_0, \texttt{Cons}) = (id, e \to (\texttt{List}\, e) \to (\texttt{List}\, e))$ und $\mathcal{W}(A_0, \texttt{0}) = (id, \texttt{Int})$. Die Frage ist also, ob und wie der Typ $\tau_1 = e \to (\texttt{List}\, e) \to (\texttt{List}\, e)$ einem Typ der Form $\tau_2 \to \tau_3$ entspricht, wobei $\tau_2 = \texttt{Int}$ ist. Da wir τ_3 erst bestimmen wollen und dieser Typ im Prinzip beliebig sein darf, setzen wir zunächst $\tau_3 = b$ für eine neue Typvariable b. Nun müssen wir also untersuchen, ob es eine Instanz θ der Typvariablen in τ_1, τ_2, τ_3 gibt, so dass $\tau_1\theta = (\tau_2 \to \tau_3)\theta$ gilt. Man bezeichnet eine solche Substitution als *Unifikator* der Typen τ_1 und $\tau_2 \to \tau_3$. In unserem Beispiel ist der Unifikator $\theta = [e/\texttt{Int}, b/(\texttt{List Int}) \to (\texttt{List Int})]$. Da τ_1 und $\tau_2 \to \tau_3$ also unifizierbar sind, ist dieser Term korrekt getypt und er hat den Typ $\tau_3\theta = (\texttt{List Int}) \to (\texttt{List Int})$.

Im allgemeinen kann es natürlich mehrere Unifikatoren geben. Hierzu betrachten wir den Term `Cons bot`. Hier muss nun $\tau_1 = e \to (\texttt{List}\, e) \to (\texttt{List}\, e)$ mit $\tau_2 \to \tau_3 = a \to b$ unifiziert werden. Der Unifikator θ von oben unifiziert auch diese beiden Terme, wenn er zusätzlich auch a durch `Int` ersetzt. Andererseits würde auch der Unifikator $\theta' = [a/e, b/(\texttt{List}\, e) \to (\texttt{List}\, e)]$ die beiden Terme unifizieren. Bei Verwendung von θ erhält man $\tau_3\theta = (\texttt{List Int}) \to (\texttt{List Int})$ als Typ für den Gesamtterm, während man bei Verwendung von θ' den allgemeineren Typ $\tau_3\theta' = (\texttt{List}\, e) \to (\texttt{List}\, e)$ erhält. Da wir ja insgesamt immer den allgemeinsten Typ bestimmen wollen, sollten wir also den Unifikator θ' anstelle von θ verwenden.

Generell unterscheiden sich diese beiden Unifikatoren dadurch, dass θ' allgemeiner als θ ist. In der Tat ist θ' der *allgemeinste Unifikator* der beiden Typen, d.h., für jeden anderen Unifikator θ existiert eine Substitution δ, so dass $\theta = \theta'\delta$ ist. Hierbei bezeichnet "$\theta'\delta$" die Hintereinanderausführung der beiden Substitutionen, wobei θ' zuerst und δ danach angewendet wird.

Definition 4.2.2 (Unifikation, allgemeinster Unifikator von Typen) *Sei θ eine Substitution (d.h. eine Abbildung von Typvariablen auf Typen, wobei nur endlich viele Typvariablen nicht auf sich selbst abgebildet werden). Die Substitution θ ist* Unifikator *von zwei Typen τ_1 und τ_2 falls $\tau_1\theta = \tau_2\theta$. Eine Substitution θ' heißt* allgemeinster Unifikator *(engl.* most general unifier, mgu*) von τ_1 und τ_2, falls folgende Bedingungen erfüllt sind:*

- *θ' ist Unifikator von τ_1 und τ_2*
- *Für alle Unifikatoren θ von τ_1 und τ_2 existiert eine Substitution δ mit $\theta = \theta'\delta$.*

Wir schreiben dann $\theta' = mgu(\tau_1, \tau_2)$.

Zu jedem Paar von Typen lässt sich automatisch herausfinden, ob sie unifizierbar sind und ggf. der allgemeinste Unifikator berechnen. Der allgemeinste Unifikator ist darüber hinaus eindeutig (bis auf Variablenumbenennungen). Einen Unifikationsalgorithmus und einen Beweis der Eindeutigkeit findet man z.B. in [Gie02].

Damit haben wir bereits folgenden Ansatz zur Typinferenz für Applikationen $(t_1\, t_2)$: Man bestimmt zunächst die allgemeinsten Typen τ_1 und τ_2 von t_1 bzw. t_2 und berechnet anschließend $\theta = mgu(\tau_1, \tau_2 \to b)$ für eine neue Typvariable b. Der Typ der Applikation ist dann $b\theta$.

Allerdings haben wir hierbei noch nicht berücksichtigt, dass sich bei der Typinferenz von t_1 und t_2 ja bestimmte Einschränkungen an die bisherige Typannahme ergeben können. Wie erwähnt, sollte man zur Bestimmung von $\mathcal{W}(\, A,\ (t_1\, t_2)\,)$ zunächst $\mathcal{W}(A,\, t_1) = (\theta_1, \tau_1)$ berechnen. Damit ist der Typ von t_1 zwar τ_1, aber der Term t_1 kann nur korrekt getypt werden, wenn die bisherige Typannahme zu $A\theta_1$ verfeinert wird. Deshalb sollte man anschließend, wenn der Typ von t_2 bestimmt wird, nicht die Typannahme A, sondern die verfeinerte Typannahme $A\theta_1$ verwenden. Man bestimmt also nun $\mathcal{W}(A\theta_1,\, t_2) = (\theta_2, \tau_2)$. Um auch t_2 korrekt zu typisieren, muss man also die bisherigen freien Variablen gemäß der Substitution θ_2 instantiieren. Damit muss man zum einen die Typannahme zu $A\theta_1\theta_2$ verfeinern und zum anderen ergibt sich daraus, dass bei diesen Typannahmen nun t_1 nicht mehr den Typ τ_1, sondern den Typ $\tau_1\theta_2$ hat. Man bestimmt also nun $\theta_3 = mgu(\tau_1\, \theta_2, \tau_2 \to b)$ für eine neue Typvariable b. Nur bei der Verfeinerung θ_3 ist der Gesamtterm $(t_1\, t_2)$ richtig getypt, d.h., die insgesamt zu treffende Verfeinerung ist also $\theta_1\, \theta_2\, \theta_3$. Dann ist der Typ der Applikation $b\theta_3$. Insgesamt erhält man also

$$
\mathcal{W}(\, A,\ (t_1\, t_2)\,) \;=\; (\,\theta_1\, \theta_2\, \theta_3,\ b\theta_3\,),\ \text{wobei}\quad
\begin{array}{rcl}
\mathcal{W}(\, A,\, t_1\,) & = & (\,\theta_1,\, \tau_1\,) \\
\mathcal{W}(\, A\,\theta_1,\, t_2\,) & = & (\,\theta_2,\, \tau_2\,) \\
\theta_3 & = & mgu(\tau_1\, \theta_2,\, \tau_2 \to b), \\
 & & b \text{ neue Variable}
\end{array}
$$

Betrachten wir den Term $\lambda x.\, \texttt{Cons}\, x\, x$. Um $\mathcal{W}(\, A_0,\ \lambda x.\, \texttt{Cons}\, x\, x)$ zu berechnen, bestimmt man zunächst $\mathcal{W}(\, A_0 + \{x :: a\},\ \texttt{Cons}\, x\, x)$. Es ergibt sich $\mathcal{W}(\, A_0 + \{x :: a\},\ \texttt{Cons}) = (id, e \to (\texttt{List}\, e) \to (\texttt{List}\, e))$ und $\mathcal{W}(\, A_0 + \{x :: a\},\ x) = (id, a)$. Nun berechnen wir $mgu(e \to (\texttt{List}\, e) \to (\texttt{List}\, e),\ a \to b) = [a/e, b/(\texttt{List}\, e) \to (\texttt{List}\, e)]$. Damit erhält man also $\mathcal{W}(\, A_0 + \{x :: a\},\ \texttt{Cons}\, x) = ([a/e, b/(\texttt{List}\, e) \to (\texttt{List}\, e)],\ (\texttt{List}\, e) \to (\texttt{List}\, e))$.

Als nächstes bestimmen wir $\mathcal{W}(\, A_0 + \{x :: e\},\ x) = (id, e)$. Hier haben wir die bisherige Typannahme durch die bereits berechnete Substitution $[a/e, \ldots]$ verfeinert. Damit muss man nun $(\texttt{List}\, e) \to (\texttt{List}\, e)$ und $e \to b'$ unifizieren. Es stellt sich aber heraus, dass diese beiden Typen für keine Instantiierung von e und b' gleich sind (denn e und $\texttt{List}\, e$ sind nicht unifizierbar). Es handelt sich hier um einen sogenannten *occur failure*, da die Typvariable e auch in dem Typ $\texttt{List}\, e$ auftritt, mit dem sie unifiziert werden soll. Der gesamte Term ist also nicht typkorrekt. Man erkennt, dass Typfehler zu einem Fehlschlag des Unifikationsproblems führen. Außerdem wird auch deutlich, dass man die bisher aufgesammelten Substitutionen auf die Typannahme anwenden muss, bevor man den Typ des Arguments bestimmt (d.h., man musste hier bei der Typbestimmung des zweiten x die Annahme $\{x :: e\}$ statt $\{x :: a\}$ verwenden). Ansonsten hätte man den Gesamtterm fälschlicherweise als richtig getypt erkannt.

4.2.4 Der gesamte Typinferenzalgorithmus

Nach den Überlegungen der vorangegangenen Abschnitte können wir nun den Typinferenzalgorithmus angeben. Es handelt sich um eine Variante des Algorithmus $\mathcal{W}$ von *Milner* [Mil78].

Definition 4.2.3 (Der Typinferenzalgorithmus $\mathcal{W}$) *Zu jeder Typannahme A und jedem Lambda-Term $t \in \Lambda$ berechnet $\mathcal{W}(A,t)$ entweder ein Paar aus einer Substitution auf Typvariablen und einem Typ oder die Berechnung von $\mathcal{W}$ scheitert aufgrund eines fehlschlagenden Unifikationsproblems. Hierbei sei $c \in \mathcal{C} \cup \mathcal{V}$ beliebig.*

- $\mathcal{W}(\, A + \{c :: \forall a_1, \ldots, a_n.\, \tau\},\ c\,) \;=\; (\, id,\ \tau[a_1/b_1, \ldots, a_n/b_n]\,)$, $b_1, \ldots, b_n$ *neue Variable*
- $\mathcal{W}(\, A,\ \lambda x.\, t\,) \;=\; (\, \theta,\ b\theta \to \tau\,)$, *wobei* $\mathcal{W}(\, A + \{x :: b\},\ t\,) = (\theta, \tau)$, b *ist neue Variable*
- $\mathcal{W}(\, A,\ (t_1\, t_2)\,) \;=\; (\, \theta_1\, \theta_2\, \theta_3,\ b\theta_3\,)$, *wobei*
$$\begin{array}{rcl} \mathcal{W}(\, A,\ t_1\,) & = & (\, \theta_1,\ \tau_1\,) \\ \mathcal{W}(\, A\,\theta_1,\ t_2\,) & = & (\, \theta_2,\ \tau_2\,) \\ \theta_3 & = & mgu(\tau_1\,\theta_2,\ \tau_2 \to b), \\ & & b \text{ neue Variablen.} \end{array}$$

Falls die Berechnung von $\mathcal{W}(A,t)$ erfolgreich ist, so sagen wir, der Term t ist typkorrekt *unter der Typannahme A.*

Hiermit haben wir nun also einen Algorithmus, der zu jedem Term und jeder Typannahme entscheidet, ob der Term typkorrekt ist und ggf. seinen allgemeinsten Typ berechnet (denn die Terminierung des Algorithmus $\mathcal{W}$ ist offensichtlich). Zur Illustration des Algorithmus $\mathcal{W}$ betrachten wir als größeres Beispiel die Typinferenz für den Term $\mathtt{fix}\,(\lambda\,\mathtt{fact}\,x.\ \mathtt{if}\ (x\ \mathtt{<=0})\ \mathtt{1}\ (\mathtt{fact}\,(x-\mathtt{1}) * x))$. Hierbei sind jeweils für oben stehende Typinferenzen benötigte Teilinferenzen darunter angegeben.

$$\begin{array}{rcl}
\mathcal{W}(A_0, \mathtt{fix}\,(\lambda\,\mathtt{fact}\,x.\ \mathtt{if}\ (x\ \mathtt{<=0})\ \mathtt{1}\ (\mathtt{fact}\,(x-1) * x))) & = & ([\ldots], \mathtt{Int} \to \mathtt{Int}) \\
\mathcal{W}(A_0, \mathtt{fix}) & = & (id, (a_1 \to a_1) \to a_1) \\
\mathcal{W}(A_0, \lambda\,\mathtt{fact}\,x.\ \mathtt{if}\ (x\ \mathtt{<=0})\ \mathtt{1}\ (\mathtt{fact}\,(x-1) * x)) & = & ([\ldots], (\mathtt{Int} \to \mathtt{Int}) \to (\mathtt{Int} \to \mathtt{Int})) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1\}, \lambda\,x.\ \mathtt{if}\ (x\ \mathtt{<=0})\ \mathtt{1}\ (\mathtt{fact}\,(x-1) * x)) & = & ([b_1/\mathtt{Int} \to \mathtt{Int}, \ldots], \mathtt{Int} \to \mathtt{Int}) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: b_2\}, \mathtt{if}\ (x\ \mathtt{<=0})\ \mathtt{1}\ (\mathtt{fact}\,(x-1) * x)) & = & ([b_2/\mathtt{Int}, b_1/\mathtt{Int} \to \mathtt{Int}, \ldots], \mathtt{Int}) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: b_2\}, \mathtt{if}\ (x\ \mathtt{<=0})\ \mathtt{1}) & = & ([b_2/\mathtt{Int}, \ldots], \mathtt{Int} \to \mathtt{Int}) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: b_2\}, \mathtt{if}\ (x\ \mathtt{<=0})) & = & ([b_2/\mathtt{Int}, \ldots], a_2 \to a_2 \to a_2) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: b_2\}, \mathtt{if}) & = & (id, \mathtt{Bool} \to a_2 \to a_2 \to a_2) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: b_2\}, (x\ \mathtt{<=0})) & = & ([b_2/\mathtt{Int}, \ldots], \mathtt{Bool}) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: b_2\}, (x\ \mathtt{<=})) & = & ([b_2/\mathtt{Int}, \ldots], \mathtt{Int} \to \mathtt{Bool}) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: b_2\}, \mathtt{<=}) & = & (id, \mathtt{Int} \to \mathtt{Int} \to \mathtt{Bool}) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: b_2\}, x) & = & (id, b_2) \\
mgu(\mathtt{Int} \to \mathtt{Int} \to \mathtt{Bool}, b_2 \to b_3) & = & [b_2/\mathtt{Int}, b_3/\mathtt{Int} \to \mathtt{Bool}] \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: \mathtt{Int}\}, \mathtt{0}) & = & (id, \mathtt{Int}) \\
mgu(\mathtt{Int} \to \mathtt{Bool}, \mathtt{Int} \to b_4) & = & [b_4/\mathtt{Bool}] \\
mgu(\mathtt{Bool} \to a_2 \to a_2 \to a_2, \mathtt{Bool} \to b_5) & = & [b_5/a_2 \to a_2 \to a_2] \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: \mathtt{Int}\}, \mathtt{1}) & = & (id, \mathtt{Int}) \\
mgu(a_2 \to a_2 \to a_2, \mathtt{Int} \to b_6) & = & [b_6/\mathtt{Int} \to \mathtt{Int}] \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: \mathtt{Int}\}, \mathtt{fact}\,(x-1) * x) & = & ([b_1/\mathtt{Int} \to \mathtt{Int}, \ldots], \mathtt{Int}) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: \mathtt{Int}\}, \mathtt{fact}\,(x-1) *) & = & ([b_1/\mathtt{Int} \to \mathtt{Int}, \ldots], \mathtt{Int} \to \mathtt{Int}) \\
\mathcal{W}(A_0 + \{\mathtt{fact} :: b_1, x :: \mathtt{Int}\}, *)) & = & (id, \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int})
\end{array}$$

$$\begin{array}{rcl}
\mathcal{W}(A_0 + \{\texttt{fact} :: b_1, x :: \texttt{Int}\}, \texttt{fact}\,(x - \texttt{1})) & = & ([b_1/\texttt{Int} \to b_9, \ldots], b_9) \\
\mathcal{W}(A_0 + \{\texttt{fact} :: b_1, x :: \texttt{Int}\}, \texttt{fact}) & = & (id, b_1) \\
\mathcal{W}(A_0 + \{\texttt{fact} :: b_1, x :: \texttt{Int}\}, x - \texttt{1}) & = & ([\ldots], \texttt{Int}) \\
\mathcal{W}(A_0 + \{\texttt{fact} :: b_1, x :: \texttt{Int}\}, x-) & = & ([\ldots], \texttt{Int} \to \texttt{Int}) \\
\mathcal{W}(A_0 + \{\texttt{fact} :: b_1, x :: \texttt{Int}\}, -) & = & (id, \texttt{Int} \to \texttt{Int} \to \texttt{Int}) \\
\mathcal{W}(A_0 + \{\texttt{fact} :: b_1, x :: \texttt{Int}\}, x) & = & (id, \texttt{Int}) \\
mgu(\texttt{Int} \to \texttt{Int} \to \texttt{Int}, \texttt{Int} \to b_7) & = & [b_7/\texttt{Int} \to \texttt{Int}] \\
\mathcal{W}(A_0 + \{\texttt{fact} :: b_1, x :: \texttt{Int}\}, \texttt{1}) & = & (id, \texttt{Int}) \\
mgu(\texttt{Int} \to \texttt{Int}, \texttt{Int} \to b_8) & = & [b_8/\texttt{Int}] \\
mgu(b_1, \texttt{Int} \to b_9) & = & [b_1/\texttt{Int} \to b_9] \\
mgu(\texttt{Int} \to \texttt{Int} \to \texttt{Int}, b_9 \to b_{10}) & = & [b_9/\texttt{Int}, b_{10}/\texttt{Int} \to \texttt{Int}] \\
\mathcal{W}(A_0 + \{\texttt{fact} :: \texttt{Int} \to \texttt{Int}, x :: \texttt{Int}\}, x) & = & (id, \texttt{Int}) \\
mgu(\texttt{Int} \to \texttt{Int}, \texttt{Int} \to b_{11}) & = & [b_{11}/\texttt{Int}] \\
mgu(\texttt{Int} \to \texttt{Int}, \texttt{Int} \to b_{12}) & = & [b_{12}/\texttt{Int}] \\
mgu((a_1 \to a_1) \to a_1, ((\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})) \to b_{13}) & = & [a_1/\texttt{Int} \to \texttt{Int}, b_{13}/\texttt{Int} \to \texttt{Int}]
\end{array}$$

Man kann zeigen, dass die statische Typüberprüfung des Algorithmus $\mathcal{W}$ korrekt ist. Dies bedeutet, dass bei einem (gemäß $\mathcal{W}$) korrekt getypten Lambda-Term auch keine Laufzeittypfehler auftreten können. Mit anderen Worten: Wenn t typkorrekt ist, so ist auch jeder Term, der durch Auswertung von t (mit Hilfe der β- und δ-Reduktion) entsteht, typkorrekt und er hat denselben allgemeinsten Typ wie t.

Satz 4.2.4 (Korrektheit des Algorithmus $\mathcal{W}$) *Sei t ein Lambda-Term über den Konstanten $\mathcal{C}$ von Def. 3.3.4, wobei t gemäß des Algorithmus $\mathcal{W}$ typkorrekt ist (d.h., es gilt $\mathcal{W}(A_0, t) = (\theta, \tau)$ für einen Typ τ und eine Substitution θ). Sei $t \to_{\beta\delta}^{*} t'$, wobei δ die Delta-Regelmenge aus Def. 3.3.5 ist. Dann gilt auch $\mathcal{W}(A_0, t') = (\theta', \tau)$ für eine Substitution θ'.*

Beweis. Ein Beweis einer solchen Aussage findet sich z.B. in [Thi94, Satz 15.4.10 bzw. Satz 15.5.7].

Die Umkehrung dieses Satzes gilt natürlich nicht, denn bereits durch die β-Reduktion können nicht typkorrekte Terme in typkorrekte Terme überführt werden. Ein Beispiel ist der nicht typkorrekte Term $(\lambda\texttt{f}.\texttt{plus}\,(\texttt{f}\,\texttt{True})\,(\texttt{f}\,\texttt{3}))\;(\lambda\texttt{x}.\,\texttt{1})$, der durch β-Reduktion in den typkorrekten Term $\texttt{plus}\,((\lambda\texttt{x}.\,\texttt{1})\,\texttt{True})\,((\lambda\texttt{x}.\,\texttt{1})\,\texttt{3}))$ und weiter in $\texttt{plus}\,\texttt{1}\,\texttt{1}$ überführt wird.

Terme wie die Fixpunktkombinatoren Θ oder Y sind allerdings nicht typkorrekt, da sie die Selbstapplikation $x\,x$ als Teilterm enthalten, die (bei durch λ gebundenem x) nicht typkorrekt ist. Wenn wir also HASKELL mit Hilfe von typkorrekten Lambda-Termen implementieren wollen, benötigen wir tatsächlich eine eingebaute Konstante `fix` für die Behandlung von rekursiven Definitionen.

Der in Kapitel 3 eingeführte Lambda-Kalkül war *ungetypt*, da man dort beliebige Terme bilden konnte, ohne auf Typeinschränkungen Rücksicht zu nehmen. Damit waren auch Terme wie die Fixpunktkombinatoren möglich. Man untersucht oft auch den sogenannten *getypten* Lambda-Kalkül, wobei man hierbei aber meist nur monomorphe Typen betrachtet. Das in diesem Kapitel betrachtete Typsystem geht deutlich darüber hinaus, da wir nun einen getypten Lambda-Kalkül mit polymorphen Typen betrachtet haben.

4.3 Typinferenz bei HASKELL-Programmen

Im vorigen Abschnitt haben wir angegeben, wie man Typinferenz für Lambda-Terme durchführen kann. Unser eigentliches Ziel ist jedoch, die Typkorrektheit von HASKELL-Programmen zu überprüfen. Der Ansatz hierzu ist, ein komplexes HASKELL-Programm zunächst wie in Def. 2.2.11 in ein einfaches HASKELL-Programm zu überführen und es anschließend weiter gemäß Def. 3.3.3 in einen Lambda-Term zu übersetzen. Dann kann man, wie im vorigen Abschnitt gezeigt, die Typkorrektheit des entstandenen Lambda-Terms überprüfen.

Hier zeigt sich, warum wir in Def. 3.3.3 zwei Regeln (10) und (11) zur Transformation von mehreren Variablendeklarationen verwenden mussten. Anstelle dieser beiden Regeln hätte man ja auch Regel (11) alleine benutzen können, um stets eine Folge von Patterndeklarationen $\{\underline{\mathsf{var}}_1 = \underline{\mathsf{exp}}_1; \ldots; \underline{\mathsf{var}}_n = \underline{\mathsf{exp}}_n\}$ durch eine einzige Patterndeklaration $(\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n) = (\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n)$ eines Tupels von Variablen zu ersetzen. Die dann entstehende Transformation von komplexen in einfache HASKELL-Programme ist aber *nicht typerhaltend*! Es existieren korrekt getypte komplexe HASKELL-Programme, die durch diese Transformation in nicht korrekt getypte einfache HASKELL-Programme überführt werden. Hierzu betrachten wir ein Programm mit drei Deklarationen für `i`, `a` und `b` bzw. den folgenden komplexen HASKELL-Ausdruck.

```
let i = \x -> x
    a = i True
    b = i 3
in  b
```

Dieser Ausdruck ist typkorrekt. Der allgemeinste Typ von `i` ist $a \to a$, so dass `i` sowohl auf einen booleschen Wert als auch auf einen Wert vom Typ `Int` angewendet werden darf. Da in einfachen HASKELL-Ausdrücken keine Folgen von Deklarationen zugelassen sind, würde dieser Ausdruck durch Regel (11) der Transformation aus Def. 2.2.11 überführt in

```
let (i,a,b) = (\x -> x, i True, i 3)
in  b
```

Dieser Ausdruck ist aber nicht mehr typkorrekt! Das Problem ist, dass hier `i`, `a` und `b` gleichzeitig deklariert werden. Innerhalb einer Deklaration müssen die Typvariablen von `i` aber überall gleich instantiiert werden. Dies ist analog zur Behandlung von durch Lambda gebundenen Typvariablen innerhalb eines Lambda-Ausdrucks. So würde der obige Ausdruck in folgenden Lambda-Term übersetzt werden.

$$\mathtt{sel}_{3,3}\,(\mathtt{fix}\,(\lambda v.\,\mathtt{tuple}_3\,(\lambda x.x)\,((\mathtt{sel}_{3,1}\,v)\,\mathtt{True})\,((\mathtt{sel}_{3,1}\,v)\,\mathtt{3})))$$

Hierbei entspricht die neue Variable v dem Tupel der drei Werte `i`, `a` und `b`. Man erkennt, dass v durch ein Lambda gebunden wird und daher sowohl im Term $((\mathtt{sel}_{3,1}\,v)\,\mathtt{True})$ als auch im Term $((\mathtt{sel}_{3,1}\,v)\,\mathtt{3})$ den gleichen Typ haben muss. Dies ist jedoch nicht möglich. In der Tat ist bereits der Teilterm

$$\lambda v.\,\mathtt{tuple}_3\,(\lambda x.x)\,((\mathtt{sel}_{3,1}\,v)\,\mathtt{True})\,((\mathtt{sel}_{3,1}\,v)\,\mathtt{3})$$

nicht typkorrekt, wie sich mit dem Algorithmus $\mathcal{W}$ direkt nachweisen lässt.

Die entsprechend modifizierte Transformation wäre also insofern “korrekt”, dass die resultierenden Programme dieselben Ergebnisse liefern. Daher könnte man sie auch tatsächlich zur Definition der Semantik und zur Implementierung von HASKELL verwenden. Allerdings entstehen hierbei zwischendurch nicht korrekt getypte Programme bzw. Lambda-Terme (die Resultate der WHNO-Reduktion sind allerdings wieder typkorrekt).

Das obige Beispiel macht deutlich, dass das Problem daran liegt, dass man nur solche Variablen gleichzeitig deklarieren sollte, bei denen dies unvermeidbar ist (d.h. Variablen, deren Deklarationen *verschränkt rekursiv* sind). Der “Fehler” der modifizierten Transformation ist, dass bei Regel (11) die Deklarationen der Variablen `i`, `a` und `b` in eine Deklaration zusammengefasst werden, obwohl die Variablen `a` und `b` nicht verschränkt rekursiv mit `i` sind. So hängen zwar `a` und `b` von `i` ab (denn `i` wird in ihrer Deklaration verwendet), aber `i` hängt nicht von den Variablen `a` und `b` ab. Um dies zu vermeiden, verwenden wir daher erst die Regel (10), um nicht verschränkt rekursive Variablen voneinander zu separieren und benutzen dann Regel (11), um verschränkt rekursive Variablendeklarationen in eine Tupeldeklaration zu überführen.

Wenn P die Folge der Deklarationen von `i`, `a` und `b` in unserem Beispiel ist, so gilt also $\mathtt{a} \succ_P \mathtt{i}$ und $\mathtt{b} \succ_P \mathtt{i}$. Aufteilungen von P sind $P_1 = \{\mathtt{i}\}$, $P_2 = \{\mathtt{a}\}$, $P_3 = \{\mathtt{b}\}$ oder eine Folge, bei der `a` und `b` vertauscht sind.

Um die Typkorrektheit bei der Transformation in einfaches HASKELL zu erhalten, sollte man wie bisher die Deklarationsfolge von `i`, `a` und `b` zunächst in geschachtelte `let`-Ausdrücke überführen, bei denen nur noch verschränkt rekursive Variablen gemeinsam in einer Deklarationsfolge deklariert werden. Man muss also eine Aufteilung P_1, P_2, P_3 der Deklarationsfolge verwenden, bei der P_1 die Deklarationsfolge des äußersten `let`-Ausdrucks ist, P_2 im mittleren `let`-Ausdruck deklariert wird und P_3 im innersten `let`-Ausdruck verwendet wird. So ergibt sich

```
let i = \x -> x
in  let a = i True
    in  let b = i 3
        in  b
```

Dies ist ein einfacher HASKELL-Ausdruck, der ebenfalls typkorrekt ist. In HASKELL macht es also einen großen Unterschied, ob man Variablen zusammen deklariert oder hierfür verschiedene Deklarationen verwendet. Der Grund ist, dass hier `i` bereits deklariert ist, wenn man den Teilausdruck

```
let a = i True
in  let b = i 3
    in  b
```

auswertet. Die Typvariablen in dem Typ von `i` können daher in diesem Teilausdruck verschieden instantiiert werden. Dies entspricht der Typisierung des Teilausdrucks unter der Typannahme $\{\mathtt{i} :: \forall a.a \to a\}$ bzw. der Ersetzung von `i` durch zwei verschiedene Kopien des Terms $\lambda x.x$ (wie es bei der Übersetzung in Lambda-Terme geschehen würde).

Bei der Transformation von Def. 2.2.11 bleibt also neben der Semantik auch der Typ der Ausdrücke erhalten, d.h., typkorrekte komplexe HASKELL-Ausdrücke werden in typkorrekte

einfache HASKELL-Ausdrücke (desselben Typs) überführt und nicht typkorrekte komplexe HASKELL-Ausdrücke werden in nicht typkorrekte einfache HASKELL-Ausdrücke überführt.

Nun können wir daher formal definieren, wann wir einen HASKELL-Ausdruck als *typkorrekt* bezeichnen. Wie erläutert, verwenden wir hierzu die obige Übersetzung in einfaches HASKELL und die weitere Überführung in Lambda-Terme. Dann kann man den Algorithmus $\mathcal{W}$ zur Typinferenz und zur Definition der Typkorrektheit benutzen.

Definition 4.3.1 (Typkorrektheit bei komplexen HASKELL-Programmen) *Für ein komplexes HASKELL-Programm ohne Typdeklarationen sei P die Folge der Pattern- und Funktionsdeklarationen und $\underline{\texttt{exp}}$ ein komplexer HASKELL-Ausdruck. Der Ausdruck $\underline{\texttt{exp}}$ ist* typkorrekt *beim Programm P gdw. der Lambda-Term $\mathcal{T}ran(P, \underline{\texttt{exp}})$ typkorrekt ist.*

Mit den beiden Übersetzungsschritten (in einfaches HASKELL und in den Lambda-Kalkül) haben wir nun die Möglichkeit, sowohl die Typüberprüfung als auch die anschließende Ausführung des Programms im Lambda-Kalkül durchzuführen. Allerdings ist dieses Vorgehen noch recht ineffizient, da bei einem Programm mit den Deklarationen P jedes Mal, wenn ein neuer Ausdruck $\underline{\texttt{exp}}$ auf seinen Typ überprüft und ausgewertet werden soll, das gesamte Programm P mit übersetzt und seine Typkorrektheit überprüft wird. Der Grund ist, dass jedes Mal ein neuer Ausdruck `let` P `in` $\underline{\texttt{exp}}$ betrachtet wird.

Es ist natürlich wesentlich günstiger, das Programm P nur einmal zu übersetzen und auf seinen Typ zu überprüfen. Dies kann in einer Compilationsphase erfolgen. Später muss dann nur noch der jeweilige neue Ausdruck $\underline{\texttt{exp}}$ übersetzt, auf seinen Typ untersucht und ausgewertet werden.

Solch eine Verbesserung des Vorgehens ist leicht möglich. Betrachten wir hierzu das folgende Programm. Hier wurde bereits Regel (1) aus Def. 2.2.11 eingesetzt, um Funktionsdeklarationen in Patterndeklarationen zu überführen, so dass wir nur noch Patterndeklarationen mit einer Variablen auf der linken Seite haben.

```
even′ = \x  -> if x == 0 then True else odd′(x-1)

odd′ = \x  -> if x == 0 then False else even′(x-1)

half = \x  -> if x == 0
                then 0
                else if even′ x then (half (x-2)) + 1 else half (x-1)
```

Wenn P die Folge der obigen Deklarationen ist, so gilt $\texttt{half} \succ_P \texttt{even}' \sim_P \texttt{odd}'$. Es ergibt sich daher die Aufteilung $P_1 = \{\texttt{even}', \texttt{odd}'\}$, $P_2 = \{\texttt{half}\}$.

Die Deklaration P_1 würde in eine einfache Deklaration einer neuen Variable `eo` überführt werden:

$$\texttt{eo} = \backslash\texttt{even}' \texttt{ -> } \backslash\texttt{odd}' \texttt{ -> } (\backslash\texttt{x -> ..odd}'(\texttt{x}-1)..,\backslash\texttt{x -> ..even}'(\texttt{x}-1)..)\ (\texttt{sel}_{2,1}\,\texttt{eo})\ (\texttt{sel}_{2,2}\,\texttt{eo})$$

bzw. (bei Anwendung einer β-Reduktion)

$$\texttt{eo} = (\backslash\texttt{x -> } \ldots(\texttt{sel}_{2,2}\texttt{eo})\,(\texttt{x}-1)\ldots, \backslash\texttt{x -> } \ldots(\texttt{sel}_{2,1}\texttt{eo})\,(\texttt{x}-1)\ldots)$$

Bei der Übersetzung in den Lambda-Kalkül muss diese Deklaration in eine nicht-rekursive Deklaration (mit `fix`) überführt werden. Die rechte Seite dieser Deklaration wird daher in

$$\texttt{fix}\,(\lambda \texttt{eo}.\ \mathcal{L}am((\backslash x \texttt{ -> } \ldots (\texttt{sel}_{2,2}\texttt{eo})\,(x-1)\ldots, \backslash x \texttt{ -> } \ldots (\texttt{sel}_{2,1}\texttt{eo})\,(x-1)\ldots)))$$

übersetzt. Wir bezeichnen diesen Term mit t_1, da er den Deklarationen aus dem Programmteil P_1 entspricht (d.h. dem Tupel der Werte für $\texttt{even}'$ und $\texttt{odd}'$). Will man nachher einen Ausdruck $\underline{\texttt{exp}}$ in diesem Programm auswerten, kann man daher zunächst einmal $\underline{\texttt{exp}}$ in einfaches HASKELL und weiter in einen Lambda-Term überführen. Dies kann ohne Berücksichtigung der Deklarationen des Programms geschehen. Anschließend muss man nur noch die in diesem Lambda-Term vorkommenden Variablen $\texttt{even}'$ und $\texttt{odd}'$ durch die jeweiligen Komponenten von t_1 ersetzen. Wir definieren hierzu die Substitution σ, die jeder im Programm deklarierten Variable den Lambda-Term zuordnet, der das Teilprogramm dieser Variablendeklaration implementiert. Es gilt also $\sigma(\texttt{even}') = (\texttt{sel}_{2,1}\ t_1)$, $\sigma(\texttt{odd}') = (\texttt{sel}_{2,2}\ t_1)$. Die Substitution σ, die die im Programm bestimmten Werte für $\texttt{even}'$ und $\texttt{odd}'$ beinhaltet, muss also nur einmal (während der Compilezeit) berechnet werden.

In unserem Beispiel muss σ natürlich auch die Variable `half` mit dem richtigen Wert instantiieren. Hierzu bildet man den Term t_2, der den Deklarationen in P_2 entspricht. Es ergibt sich

$$t_2 = \texttt{fix}\,(\lambda\ \texttt{half}\ x.\ \ldots \texttt{even}'\,x\ldots).$$

Somit ist $t_2\sigma$ der Wert der Variablen `half` (hierbei dient σ dazu, die Variable $\texttt{even}'$ richtig zu instantiieren). Zur Auswertung eines Ausdrucks $\underline{\texttt{exp}}$ in diesem Programm muss also nun der Lambda-Term $\mathcal{L}am(\,(\underline{\texttt{exp}})_{tr}\,)\sigma$ ausgewertet werden.

Durch diese Form der Compilierung lässt sich auch die Typinferenz wesentlich effizienter durchführen. Bislang muss bei jedem Ausdruck $\underline{\texttt{exp}}$ der um das Programm P erweiterte Ausdruck `let` P `in` $\underline{\texttt{exp}}$ auf Typkorrektheit untersucht werden. Dies bedeutet zum einen, dass die Typen für das Programm P jedes Mal neu bestimmt werden. Zum anderen wird bei jedem Vorkommen einer in P deklarierten Variable in $\underline{\texttt{exp}}$ ihre komplette Deklaration bei der Übersetzung in den Lambda-Kalkül kopiert. Für jede dieser Kopien wird der Typ separat erneut bestimmt.

Um dies zu vermeiden, sollte man daher zuerst Typinferenz für die Terme t_1 und t_2 durchführen. Dabei berechnet man für jede im Programm deklarierte Variable $\underline{\texttt{var}}$ ihren allgemeinsten Typ τ. Will man nun später einen Ausdruck $\underline{\texttt{exp}}$ bei dem Programm P auf Typkorrektheit untersuchen, reicht es, lediglich die zusätzliche Typannahme $\{\underline{\texttt{var}} :: \forall\,\tau\}$ zu verwenden. Man muss also nicht noch einmal das gesamte Programm auf seinen Typ untersuchen und man muss auch die Programmvariablen nicht durch ihre Deklaration ersetzen. Beispielsweise ergibt sich

$$\mathcal{W}(A_0, t_1) = (\ldots, (\texttt{Int} \to \texttt{Bool}, \texttt{Int} \to \texttt{Bool})).$$

Wenn man nun t_2 untersucht, in dem die in P_1 deklarierten Variablen $\texttt{even}'$ und $\texttt{odd}'$ auftreten, sollte man die zusätzliche Typannahme verwenden, die diesen Variablen die jeweiligen Komponenten des Typs von t_1 zuordnet. Man bestimmt also

$$\mathcal{W}(A_0 + \{\texttt{even}' :: \texttt{Int} \to \texttt{Bool}, \texttt{odd}' :: \texttt{Int} \to \texttt{Bool}\}, t_2) = (\ldots, \texttt{Int} \to \texttt{Int}).$$

Um den Typ eines Ausdrucks $\underline{\mathsf{exp}}$ in diesem Programm zu bestimmten, berechnet man also nun

$$\mathcal{W}(A_0 + \{\texttt{even}' :: \texttt{Int} \to \texttt{Bool}, \texttt{odd}' :: \texttt{Int} \to \texttt{Bool}, \texttt{half} :: \texttt{Int} \to \texttt{Int}\}, \mathcal{L}am(\,(\underline{\mathsf{exp}})_{tr}\,)).$$

Satz 4.3.2 (Compilierung von komplexen HASKELL-Programmen) *Sei P die Folge der Patterndeklarationen eines komplexen HASKELL-Programms (ohne Typdeklarationen) nach Anwendung von Regel (1) so oft wie möglich (d.h., es gibt nur noch Patterndeklarationen, die eine Variable auf der linken Seite haben). Sei $P_1, \ldots, P_k$ eine Aufteilung der Deklaration mit $P_i = \{\underline{\mathsf{var}}_{i,1} = \underline{\mathsf{exp}}_{i,1}, \ldots, \underline{\mathsf{var}}_{i,n_i} = \underline{\mathsf{exp}}_{i,n_i}\}$. Sei*

$$\begin{array}{lcll} t_i & = & \texttt{fix}\,(\lambda\underline{\mathsf{var}}_{i,1}.\,\mathcal{L}am(\,(\underline{\mathsf{exp}}_{i,1})_{tr}\,)), & \textit{falls } n_i = 1 \\ t_i & = & \texttt{fix}\,(\lambda\underline{\mathsf{var}}.\,\mathcal{L}am(\,(\underline{\mathsf{exp}}_{i,1}, \ldots, \underline{\mathsf{exp}}_{i,n_i})_{tr}\,)[\underline{\mathsf{var}}_{i,j}/(\texttt{sel}_{n_i,j}\;\underline{\mathsf{var}})]), & \textit{falls } n_i > 1, \end{array}$$

wobei $\underline{\mathsf{var}}$ eine neue Variable ist. Wir definieren die Substitution σ mit $\sigma(\underline{\mathsf{var}}_{i,j}) = (\texttt{sel}_{n_i,j}\; t_i\sigma)$, falls $n_i > 1$ und mit $\sigma(\underline{\mathsf{var}}_{i,1}) = t_i\sigma$ sonst. Die Compilierung liefert eine korrekte Implementierung, d.h. $\mathcal{V}al[\![\mathcal{L}am((\underline{\mathsf{exp}})_{tr})\sigma]\!]\rho = \mathcal{V}al[\![\texttt{let}\ P\ \texttt{in}\ \underline{\mathsf{exp}}]\!]\rho$.

Diese Compilierung lässt sich auch zur Untersuchung der Typkorrektheit verwenden. Das Programm P ist typkorrekt gdw.

$$\begin{array}{rcl} \mathcal{W}(A_0, t_1) & = & (\theta_1, (\tau_{1,1}, \ldots, \tau_{1,n_1})) \\ \mathcal{W}(A_0 + \{\underline{\mathsf{var}}_{1,1} :: \forall\tau_{1,1}, \ldots, \underline{\mathsf{var}}_{1,n_1} :: \forall\tau_{1,n_1}\}, t_2) & = & (\theta_2, (\tau_{2,1}, \ldots, \tau_{2,n_2})) \\ & \vdots & \\ \mathcal{W}(A_0 + \{\underline{\mathsf{var}}_{1,1} :: \forall\tau_{1,1}, \ldots, \underline{\mathsf{var}}_{k-1,n_{k-1}} :: \forall\tau_{k-1,n_{k-1}}\}, t_k) & = & (\theta_k, (\tau_{k,1}, \ldots, \tau_{k,n_k})) \end{array}$$

für bestimmte Substitutionen $\theta_1, \ldots, \theta_k$ und Typen $\tau_{1,1}, \ldots, \tau_{k,n_k}$ gilt. Der Ausdruck $\underline{\mathsf{exp}}$ ist typkorrekt beim Programm P, falls

$$\mathcal{W}(A_0 + \{\underline{\mathsf{var}}_{1,1} :: \forall\tau_{1,1}, \ldots, \underline{\mathsf{var}}_{k,n_k} :: \forall\tau_{k,n_k}\}, \mathcal{L}am((\underline{\mathsf{exp}})_{tr})) \;=\; (\theta, \tau)$$

für bestimmte θ und τ.

Wir haben bislang nur solche Programme bei der Untersuchung der Typkorrektheit betrachtet, die keine Typdeklarationen der Form $\underline{\mathsf{var}} :: \underline{\mathsf{type}}$ enthalten. Hierzu untersuchen wir das folgende Beispiel.

```
i :: Int -> Int
i = \x -> x
a = i True
```

Ein erster Ansatz wäre, erst das Programm ohne die Typdeklarationen auf Typkorrektheit zu überprüfen. Dies würde im obigen Beispiel gelingen und man würde für `i` den allgemeinsten Typ $a \to a$ bestimmen. Anschließend müsste man untersuchen, ob die deklarierten Typen τ' jeweils Spezialfälle der berechneten allgemeinsten Typen τ sind, d.h. ob es eine Substitution δ gibt mit $\tau' = \tau\delta$. Dies ist hier ebenfalls der Fall, denn wir haben $\texttt{Int} \to \texttt{Int} = (a \to a)[a/\texttt{Int}]$. Dennoch ist dieses HASKELL-Programm offensichtlich

nicht typkorrekt. Das Problem ist, dass die Typdeklaration `i :: Int -> Int` im ganzen Programm verwendet werden müsste. Dann ist aber der Term `i True` nicht korrekt getypt.

Eine Typdeklaration $\underline{\mathsf{var}} :: \underline{\mathsf{type}}$ bedeutet also zweierlei: Zum einen muss man überprüfen, ob bei einer Deklaration der Variable $\underline{\mathsf{var}}$ der allgemeinste Typ allgemeiner oder gleich $\underline{\mathsf{type}}$ ist. Diese Überprüfung betrifft also den *Deklarationsteil* des Programms, d.h. den Teil, in dem diese Variable (und ihr Typ) eingeführt wird. Zum anderen muss man sicherstellen, dass bei jeder Verwendung der Variable $\underline{\mathsf{var}}$ der Typ $\underline{\mathsf{type}}$ (oder Instanzen davon) benutzt werden. Diese Aufgabe betrifft also den *Verwendungsteil* des Programms, d.h. den Teil, in dem diese Variable nicht mehr deklariert, sondern nur noch benutzt wird. Wenn wir uns wieder nur auf HASKELL-Ausdrücke statt Programme beschränken, so ist der Unterschied zwischen Deklarations- und Verwendungsteil deutlich. In

$$\mathtt{let}\ \{\underline{\mathsf{var}} :: \underline{\mathsf{type}}; \underline{\mathsf{var}} = \underline{\mathsf{exp}}\}\ \mathtt{in}\ \underline{\mathsf{exp}}'$$

ist $\{\underline{\mathsf{var}} :: \underline{\mathsf{type}}; \underline{\mathsf{var}} = \underline{\mathsf{exp}}\}$ der Deklarationsteil von $\underline{\mathsf{var}}$, in dem man überprüfen muss, ob $\underline{\mathsf{exp}}$ tatsächlich einen Typ hat, von dem $\underline{\mathsf{type}}$ eine Instanz ist. Im Verwendungsteil $\underline{\mathsf{exp}}'$ muss dann sichergestellt werden, dass man den Typ $\underline{\mathsf{type}}$ für die Variable $\underline{\mathsf{var}}$ benutzt. Diese zweifache Wirkung von Typdeklarationen (im Deklarations- und im Verwendungsteil) führt dazu, dass wir die Wirkung von Typdeklarationen nicht direkt im Lambda-Kalkül nachbilden können. Solche Typdeklarationen können nur dann korrekt behandelt werden, wenn man Deklarations- und Verwendungsteile trennen kann. Eine Möglichkeit hierzu wäre, einen um `let` erweiterten Lambda-Kalkül zu betrachten, da dadurch die Trennung zwischen diesen beiden Teilen deutlich wird. (Dies entspräche allerdings nicht mehr dem normalen Lambda-Kalkül, da man die Auswertung von `let` nicht als (typkorrekte) Delta-Regel formulieren kann.)

Solange wir uns allerdings nur auf Typdeklarationen auf der obersten Ebene des Programms beschränken (d.h., wir schließen lokale Typdeklarationen in `let`-Ausdrücken aus), ergibt sich durch die Verwendung der Compilierungstechnik auch die Möglichkeit, Typdeklarationen mit zu berücksichtigen. Der Grund ist, dass durch die Aufteilung des Programms in $P_1, \ldots, P_k$ deutlich wird, welches die Deklarations- und welches die Verwendungsteile für die jeweiligen Variablen sind. Die für die Typdeklarationen benötigten Überprüfungen und Typannahmen lassen sich also (auf der Meta-Ebene) in den Typüberprüfungsalgorithmus des Compilers mit integrieren.

Die Modifikation des Typüberprüfungsverfahrens von Satz 4.3.2 verläuft wie folgt: Um zu überprüfen, ob die deklarierten Typen jeweils Spezialfälle der berechneten allgemeinsten Typen $\tau_{i,j}$ der in P deklarierten Variablen $\underline{\mathsf{var}}_{i,j}$ sind, muss man jeweils untersuchen, ob es Substitutionen $\delta_{i,j}$ mit $\tau_{i,j}\delta_{i,j} = \underline{\mathsf{type}}_{i,j}$ für alle Typdeklarationen "$\underline{\mathsf{var}}_{i,j} :: \underline{\mathsf{type}}_{i,j}$" gibt. Dies stellt sicher, dass die Deklarationsteile der Bedingung der Typdeklaration genügen. Außerdem muss man im jeweiligen Verwendungsteil der Variablen $\underline{\mathsf{var}}_{i,j}$ die entsprechende Typdeklaration mit in die Typannahme aufnehmen (wobei man evtl. vorher in der Typannahme vorhandene Typen für diese Variable überschreibt). Dies betrifft nicht nur die Typbestimmung von Ausdrücken, die unter diesem Programm ausgewertet werden sollen, sondern auch die Deklarationen von Variablen, die in späteren Programmteilen eingeführt werden. In der Tat kann man alle Typdeklarationen sogar bei allen Deklarationsteilen bereits mit in die Typannahme aufnehmen. Der Grund ist, dass im Deklarationsteil von $\underline{\mathsf{var}}_{i,j}$

diese Variable selbst sowieso nicht frei auftritt (sondern durch ein Lambda gebunden wird, dass vorherige Typannahmen ohnehin überschreibt).

Definition 4.3.3 (Typkorrektheit bei Programmen mit Typdeklarationen) *Seien P, $P_1, \ldots, P_k$ und t_i wie in Satz 4.3.2, wobei das Programm aber nun auch Typdeklarationen T für die Variablen enthalten darf, die in P auf oberster Ebene deklariert werden (d.h. Deklarationen der Art $\underline{\mathsf{var}}_{i,j} :: \underline{\mathsf{type}}_{i,j}$, die natürlich auch ganz oder teilweise fehlen können). Sei $\forall T$ die Typannahme, die aus T entsteht, indem die freien Variablen der Typschemata allquantifiziert werden, d.h., $\forall T = \{v :: \forall \tau \mid v :: \tau \in T\}$. Das Programm ist typkorrekt, falls*

$$
\begin{array}{rcl}
\mathcal{W}(A_0 + \forall T, t_1) & = & (\theta_1, (\tau_{1,1}, \ldots, \tau_{1,n_1})) \\
\mathcal{W}(A_0 + \forall T + \{\underline{\mathsf{var}}_{1,1} :: \forall \tau_{1,1}, \ldots, \underline{\mathsf{var}}_{1,n_1} :: \forall \tau_{1,n_1}\}, t_2) & = & (\theta_2, (\tau_{2,1}, \ldots, \tau_{2,n_2})) \\
& \vdots & \\
\mathcal{W}(A_0 + \forall T + \{\underline{\mathsf{var}}_{1,1} :: \forall \tau_{1,1}, \ldots, \underline{\mathsf{var}}_{k-1,n_{k-1}} :: \forall \tau_{k-1,n_{k-1}}\}, t_k) & = & (\theta_k, (\tau_{k,1}, \ldots, \tau_{k,n_k}))
\end{array}
$$

sowie

$$\tau_{i,j}\delta_{i,j} = \underline{\mathsf{type}}_{i,j}\delta'_{i,j} \text{ für alle } \text{“}\underline{\mathsf{var}}_{i,j} :: \underline{\mathsf{type}}_{i,j}\text{”} \in T$$

für bestimmte Substitutionen $\theta_1, \ldots, \theta_k$, $\delta_{i,j}, \delta'_{i,j}$ und Typen $\tau_{1,1}, \ldots, \tau_{k,n_k}$ gilt.

Der Ausdruck $\underline{\mathsf{exp}}$ ist typkorrekt beim Programm P, falls

$$\mathcal{W}(A_0 + \forall T + \{\underline{\mathsf{var}}_{1,1} :: \forall \tau_{1,1}, \ldots, \underline{\mathsf{var}}_{k,n_k} :: \forall \tau_{k,n_k}\}, \mathcal{L}am((\underline{\mathsf{exp}})_{tr})) \;=\; (\theta, \tau)$$

für bestimmte θ und τ.

Hiermit haben wir nun auch definiert, wann ein Programm mit Typdeklarationen (auf der obersten Ebene) typkorrekt ist und einen Algorithmus angegeben, um die Typinferenz auch für solche Programme durchzuführen. Wie bei der Implementierung von HASKELL lässt sich natürlich auch die Typinferenz noch weiter verbessern, indem man z.B. gemeinsame Teilterme nur einmal repräsentiert, etc. Auch eine Typinferenz direkt auf der HASKELL-Ebene anstatt auf der Ebene des Lambda-Kalküls ist natürlich möglich. Techniken zur effizienten Implementierung von funktionalen Sprachen wurden und werden stark untersucht und es gibt darüber hinaus auch viele Verfahren, um durch automatische Programmtransformationen die Effizienz des erzeugten Codes zu vergrößern. Für weitergehende Literatur sei auf [PJ87, FH88, Rea89] etc. verwiesen.

Literaturverzeichnis

[BN98] F. Baader & T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[Bar84] H. P. Barendregt. *The lambda calculus, its syntax and semantics*, North Holland, 1984.

[Bir98] R. Bird. *Introduction to functional programming using* HASKELL. Prentice Hall, 1998.

[Chu41] A. Church. The calculi of Lambda-Conversion, *Annals of Mathematical Studies*, No. 6, Princeton University Press, 1941.

[FH88] A. Field, P. Harrison. *Functional programming*. Addison-Wesley, 1988.

[Gie01] J. Giesl. *Automatisierte Programmverifikation*. Skript zur Vorlesung, RWTH Aachen, 2001.

[Gie02] J. Giesl. *Termersetzungssysteme*. Skript zur Vorlesung, RWTH Aachen, 2002.

[HPF00] P. Hudak, J. Peterson, J. Fasel. A gentle introduction to HASKELL, 2000. Available from `http://www.haskell.org/tutorial/`

[Hud00] P. Hudak. *The* HASKELL *school of expression: learning functional programming through multimedia*, Cambridge University Press, 2000.

[Hut07] G. Hutton. *Programming in* HASKELL, Cambridge University Press, 2007.

[Kle52] S. C. Kleene. *Introduction to metamathematics*. North-Holland, 1952.

[Klo93] J. W. Klop, V. van Oostrom, F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1,2):279 - 308, 1993.

[LS87] J. Loeckx & K. Sieber. *The foundations of program verification*. Wiley-Teubner, 1987.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348-375, 1978.

[Pep02] P. Pepper. *Funktionale Programmierung*. Springer, 2002.

[PJ87] S. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, 1987.

[PJH98] S. Peyton Jones & J. Hughes (eds.). *Report on the programming language* HASKELL 98, 1998. Available from `http://www.haskell.org/definition`.

[PJ00] S. Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in* HASKELL, Marktoberdorf Summer School 2000. Available from `http://research.microsoft.com/users/simonpj/papers/marktoberdorf`.

[Rea89] C. Reade. *Elements of functional programming*. Addison-Wesley, 1989.

[Ros73] B. K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20:160-187, 1973.

[Sco76] D. S. Scott. Data types as lattices. *SIAM Journal on Comp. Sci.*, 5(3):522-587, 1976.

[Sto81] J. E. Stoy. *Denotational semantics: The Scott-Strachey approach to programming language theory*. MIT Press, 1981.

[Thi94] P. Thiemann. *Grundlagen der funktionalen Programmierung*. Teubner, 1994.

[Tho99] S. J. Thompson. HASKELL*: The craft of functional programming*. Addison-Wesley, 1999.

Index

$D_\perp$, 85
Λ, 112
$\backslash$, 24
$\perp$, 69
$\perp_D$, 76
λ, 24, 112
$\langle D_1 \to D_2 \rangle$, 77
$\mathcal{C}$, 112
$\mathcal{V}$, 112
ω, 91
ω_{tr}, 98
$\oplus$, 87
ρ, 90
$\sqcup S$, 75
$\sqsubseteq_D$, 69
$\to_\beta$, 114
$\to_\delta$, 116
$\to_\eta$, 117
$\to_\alpha$, 114
$\to_{\beta\delta}$, 117
(), 22
(+), 19
*, 12, 19
+, 12, 19
++, 25, 29
-, 12
--, 11
->, 11, 28
., 37
/, 12, 18
/=, 32
:, 6, 16, 19
::, 11
:t, 21
<-, 46
==, 12, 19, 32
>=, 12, 35
>>, 53
>>=, 55
[], 6, 16
&&, 12, 34
_, 26
||, 12
'...', 22
(...), 14, 22, 28
[...], 6, 22, 28
"...", 22
`...`, 19

Abstraktion, 112
add, 43
addlist, 41
addtolist, 45
addTree, 45
Aktion, 53
allgemeinster Unifikator, 29, 134
Antisymmetrie, 70
Anwendung, 112
    partielle, 15, 21
append, 25, 29, 32, 44, 99
Applikation, 112
apply, 61
argof$_{\text{constr}}$, 97
Assoziation
    Funktionsanwendung, 15, 22, 112
    Funktionsraumkonstruktor ->, 14
    Listenkonstruktor :, 6, 22
    von Operatoren, 19
Ausdruck, 21
    Auswertung, 12
    Syntax, 24
Ausführung von Programmen, 12
Ausgabe, 53
Auswertung, 6
    nicht-strikt, 13, 48

strikt, 8, 13
Auswertungsstrategie, 12

Baum, 32
bedingte Gleichung, 14
berechenbar, 111
bind, 55
Bindungspriorität
Funktionsanwendung, 12, 22
von Operatoren, 20
Bool, 12, 15, 28
bot, 97

C, 4
call-by-name, 13
call-by-value, 13
case, 23
cdecl, 33
Char, 28
char, 22
Church, Alonzo, 111
Church-Rosser Satz, 116
class, 33
coalesced sum, 87
Color, 30
Compilierung, 121, 142
bei Typdeklarationen, 144
complete partial order, 76
Con, 90
conc, 44
concat, 45, 46
condrhs, 14
Cons, 31
constr, 21
Constructions, 90
context, 34
continuous, 77
cpo, 76
curry, 38
Curry, Haskell B., 14
Currying, 14, 37

data, 30
Datenkonstruktor, 15, 21
Datentyp, 30
rekursiv, 32
verschränkt rekursiv, 32
dd, 88
decl, 11, 17, 20, 21
Deklaration, 6, 10
Funktionsdeklaration, 11
Infixdeklaration, 19
Instanzendeklaration, 33
lokale, 17, 23
Patterndeklaration, 17, 26
Syntax, 21
Topdeklaration, 29
Typdeklaration, 6, 11
von Typklassen, 33
deklarative Programmiersprache, 4
denotationelle Semantik, 68
deriving, 31, 35
divide, 19
do, 56
Dom, 87, 90
Domain, 69, 117
Basis-, 70
eines Programms, 90
für Tupeltypen, 85
flach, 70
Produkt-, 71
dropall, 50
dropEven, 41
drop_mult, 50
dropUpper, 41

Eager Evaluation, 13
EBNF, 30
echo, 55
einfache HASKELL-Programme, 91
Übersetzung, 96, 106
Eingabe, 53
Einschränkung (Guard), 46
Einschränkungsregel, 46
Env, 90
Environment, 90
initiales, 91, 98
Eq, 33, 35
equal, 26
ERLANG, 8
eval, 60

Exp, 95, 110
$\underline{\text{exp}}_{tr}$, 106
exp, 24, 46
Expression, 21
Extension, 72

fact, 74, 81, 92
False, 12, 15
Fehlerbehandlung, 60
Fehlermonade, 63
ff , 79, 82, 94
filter, 42
fix, 120
Fixpunkt, 83
  kleinster, 83, 88, 94, 120
Fixpunktkombinator, 125
Fixpunktsatz, 83
Float, 17, 28
float, 22
fold, 44
foldr, 44
foldTree, 45
Forest, 32
freie Variablen, 92, 113
from, 49
Functions, 90
fundecl, 11, 12, 17
Funktion höherer Ordnung, 7, 37
Funktional, 37
Funktionsanwendung, 22
Funktionsbibliothek, 12
Funktionskomposition, 37
funlhs, 12, 15

Generator, 46
Generatorregel, 46
geschlossener Term, 113
getChar, 54
gets, 55
GHC, 10
Grammatik, 11, 30
Grundoperation, 12
Grundterm, 86, 88

Halbordnung, 70
half, 31
hamming, 52
Hamming-Problem, 51
has_length_three, 27
HASKELL, 6, 10
  einfaches, 91
  Implementierung, 123
  komplexes, 99
  Semantik, 68, 95, 96
  Syntax, 10
  Typüberprüfung, 127
Hasse-Diagramm, 71

id, 29
idecl, 33
Identitätsmonade, 62
if then else, 22
imperative Programmiersprache, 4, 57
infinity, 49
infix, 19
infixdecl, 20
infixl, 19
infixr, 19
innermost Auswertung, 13
instance, 33
Instanz, 32
instype, 33
Int, 6, 11, 28
integer, 22
IO, 53
$\text{isa}_{\underline{\text{constr}}}$, 97
$\text{isa}_{n-\text{tuple}}$, 97
isLower, 41

JAVA, 5, 47
Just, 60

Kapselung, 58
kartesisches Produkt von Typen, 14
Kategorientheorie, 53
Kette, 74
kleinste obere Schranke, 75
kleinster Fixpunkt, 83, 88, 94, 120
Kommentar, 11
komplexe HASKELL-Programme, 99
Konfluenz, 109, 115
  des Lambda-Kalküls, 116

Konstruktorklasse, 58
Konstruktoroperator, 19
Kontext, 33, 34
Kontrollstruktur, 5, 7

$\mathcal{L}am$, 121
Lambda, 24
Lambda-Kalkül, 91, 111
  getypt, 126
  Reduzierung von HASKELL, 118
  reiner, 124
Lambda-Term, 112
Lazy Evaluation, 7, 13, 48
least fixpoint, 83
least upper bound, 75
len, 6, 16, 25, 29, 32
let, 23
lexikographische Pfadordnung, 109
lfp, 83
Lift, 85
linear, 26
LISP, 7
List, 31, 35, 40, 89
Liste, 16, 22, 31
  unendliche, 49
Listenkomprehension, 46
Logik höherer Ordnung, 117
lub, 75

map, 39, 46
mapList, 40
mapTree, 40
match, 100
Matching, 15, 25
maxi, 14, 23, 27
Maybe, 60
mer, 52
Methode, 33
ML, 7
mod, 50
Monad, 58
Monade, 53
  Ein-/Ausgabe, 53
  Fehlermonade, 63
  Identitätsmonade, 62
  Kombination, 65
  Programmierung, 58
  Zustandsmonade, 64
Monadengesetze, 58
monomorph, 69
Monotonie, 73, 78
most general unifier (mgu), 134
mult, 49
multlist, 41
Muster, 15
MyBool, 30

Nats, 31, 88
nicht-überlappend, 109
Nichtterminalsymbol, 11
Nil, 31
non_term, 13
Normalform, 115
not, 12
Nothing, 60
Num, 36

odd, 41
Offside-Regel, 18
ones, 51
op, 20
operationelle Semantik, 68, 111
Operator, 19, 48
Ord, 35
Ordnung, 70
  flach, 70
  partiell, 70
  vollständig, 76
otherwise, 14
outermost Auswertung, 13, 48, 118

pat, 27
patdecl, 17
Pattern, 15, 25
  Deklaration, 17
  Matching, 6, 15, 25, 30, 48
  Syntax, 27
plus, 14, 24, 31
Polymorphismus, 7, 28
  ad-hoc, 28, 32
  parametrischer, 28
Prelude, 12

primEqInt, 34
primes, 50
prod, 43
putChar, 53
putStr, 54

qsort, 47
qual, 46
Qualifikator, 46
Quicksort, 47

readFile, 55
Redex, 12
Reduktion, 114
  α, 114
  β, 114
  δ, 116
  η, 117
Reduktionsstrategie, 118
referentielle Transparenz, 7, 53
Reflexivität, 70
Rekursion, 6, 7, 32
Result, 60
return, 53
rhs, 12, 14
roots, 18, 23

SCHEME, 7
Scott, Dana, 117
second, 16
Seiteneffekt, 5, 7, 53
$\mathtt{sel}_{n,i}$, 97
Semantik, 68
Sequentialität, 57
Show, 31, 35
Sieb des Eratosthenes, 50
Speicherverwaltung, 5, 7
sqrt, 18
sqrtlist, 38
square, 11
ST, 61
State Transformer, 61
STE, 65
Stetigkeit, 77, 78, 95
strikt, 49, 72
String, 30
string, 22
Substitution, 15, 113
suc, 17, 38
Succ, 31
suclist, 38
sucTree, 40
sum, 45
Supremum, 75

take, 49
takeWhile, 51
Term, 59
Termersetzung, 12, 109
Terminierung, 13, 109
then, 53
three, 13
topdecl, 30, 33, 34
$\mathcal{T}ran$, 122
transitiv-reflexive Hülle, 115
Transitivität, 70
Tree, 32, 40, 45
True, 12, 15
Tupel
  von Ausdrücken, 22
  von Typen, 14
$\mathtt{tuple}_n$, 119
Tuples_n, 87, 90
Typ, 21, 28
  allgemeinster, 29, 128
  parametrischer, 31
  Syntax, 36
Typüberprüfung, 127
  statisch, 127
Typabkürzung, 30
Typannahme, 128, 129
  initiale, 128, 129
type, 30
typedecl, 11, 34
Typeinführung, 29
Typinferenz
  bei Applikationen, 133
  bei Lambda-Abstraktionen, 130
  bei Typdeklarationen, 144
  bei Variablen und Konstanten, 130
  bei HASKELL, 138

Typinferenzalgorithmus $\mathcal{W}$, 136
Typklasse, 32
  hierarchische Organisation, 35
Typkonstruktor, 28
Typkorrektheit, 29
  der Transformation in einfaches HASKELL, 138
  von HASKELL-Programmen, 140
Typschema, 128
  flach, 131
Typüberprüfung, 29
Typvariable, 28

überladen, 32
unclear, 16
uncurry, 38
und, 15, 23
unendliches Datenobjekt, 50
Unifikation, 29, 134
Unterklasse, 35

$\mathcal{V}al$, 69, 95, 110
Value, 60
var, 11
Variablenbezeichner, 11
Variablenoperator, 19
Verifikation, 109
verschmolzene Summe, 87
verzögerte Auswertung, 7, 13

$\mathcal{W}$, 136
Weak Head Normal Form, 25, 118
Weak Head Normal Order Reduction, 119
where, 17, 33
WHNF, 118
writeFile, 55

Zero, 31
zeros, 25
zipWith, 41
zirkuläres Datenobjekt, 51
Zustandsmonade, 64