Einführung in die Funktionale Programmierung

Wintersemester 2023/24

Prof. Dr. Manfred Schmidt-Schauß

Institut für Informatik
Fachbereich Informatik und Mathematik
Goethe-Universität Frankfurt am Main
Postfach 11 19 32
D-60054 Frankfurt am Main
Email: schauss@ki.informatik.uni-frankfurt.de

Stand: 3. Dezember 2023

Dank an PD Dr. David Sabel für viele Beiträge und Mithilfe

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel erläutern wir den Begriff der funktionalen Programmierung bzw. funktionalen Programmiersprachen und motivieren, warum die tiefer gehende Beschäftigung mit solchen Sprachen lohnenswert ist. Wir geben einen Überblick über verschiedene funktionale Programmiersprachen. Außerdem geben wir einen kurzen Überblick über den (geplanten) Inhalt der Vorlesung und einige Literaturempfehlungen.

1.1 Was sind funktionale Programmiersprachen?

Um den Begriff der funktionalen Programmiersprachen einzugrenzen, geben wir zunächst eine Übersicht über die Klassifizierung von Programmiersprachen bzw. Programmierparadigmen. Im allgemeinen unterscheidet man zwischen *imperativen* und *deklarativen* Programmiersprachen. Objektorientierte Sprachen kann man zu den imperativen Sprachen hinzuzählen, wir führen sie gesondert auf:

Imperative Programmiersprachen Der Programmcode ist eine Folge von Anweisungen (Befehlen), die sequentiell ausgeführt werden und den *Zustand* des Rechners (Speicher) verändern. Eine Unterklasse sind sogenannte prozedurale Programmiersprachen, die es erlauben den Code durch Prozeduren zu strukturieren und zu gruppieren.

Objektorientierte Programmiersprachen In objektorientierten Programmiersprachen werden Programme (bzw. auch Problemstellungen) durch Klassen (mit Methoden und Attributen) und durch Vererbung strukturiert. Objekte sind Instanzen von Klassen. Zur Laufzeit versenden Objekte Nachrichten untereinander durch Methodenaufrufe. Der Zustand der Objekte und daher auch des Systems wird bei Ausführung des Programms verändert.

Deklarative Programmiersprachen Der Programmcode beschreibt hierbei im Allgemeinen nicht, *wie* das Ergebnis eines Programms berechnet

wird, sondern eher *was* berechnet werden soll. Hierbei manipulieren deklarative Programmiersprachen im Allgemeinen nicht den Zustand des Rechners, sondern dienen der Wertberechnung von Ausdrücken. Deklarative Sprachen lassen sich grob aufteilen in funktionale Programmiersprachen und logische Programmiersprachen.

Logische Programmiersprachen Ein Programm ist eine Menge logischer Formeln (in Prolog: so genannte Hornklauseln der Prädikatenlogik), zur Laufzeit werden mithilfe logischer Schlüsse (der so genannten Resolution) neue Fakten hergeleitet.

Funktionale Programmiersprachen Ein Programm in einer funktionalen Programmiersprache besteht aus einer Menge von Funktionsdefinitionen (im engeren mathematischen Sinn). Das Ausführen eines Programms entspricht dem Auswerten eines Ausdrucks, d.h. das Resultat ist ein einziger Wert. In rein funktionalen Programmiersprachen gibt es keinen Zustand des Rechners, der manipuliert wird, d.h. es treten bei der Ausführung keine Seiteneffekte (d.h. sichtbare Speicheränderungen) auf. Vielmehr gilt das Prinzip der *referentiellen Transparenz*: Das Ergebnis einer Anwendung einer Funktion auf Argumente, hängt ausschließlich von den Argumenten ab, oder umgekehrt: Die Anwendung einer gleichen Funktion auf gleiche Argumente liefert stets das gleiche Resultat. Variablen in funktionalen Programmiersprachen bezeichnen keine Speicherplätze, sondern stehen für (unveränderliche) Werte.

1.2 Warum Funktionale Programmierung?

Es stellt sich die Frage, warum die Beschäftigung mit funktionalen Programmiersprachen lohnenswert ist. Wir führen im Folgenden einige Gründe auf:

- Andere Sichtweise der Problemlösung: Während beim Programmieren in imperativen Programmiersprachen ein Großteil in der genauen Beschreibung besteht, wie auf dem Speicher operiert wird, wird in funktionalen Programmiersprachen eher das Problem bzw. die erwartete Antwort beschrieben. Man muss dabei meist einen anderen Blick auf das Problem werfen und kann oft von speziellen Problemen abstrahieren und zunächst allgemeine Methoden (bzw. Funktionen) implementieren. Diese andere Sichtweise hilft später auch beim Programmieren in imperativen Programmiersprachen, da man auch dort teilweise funktional programmieren kann.

- Elegantere Problemlösung: Im Allgemeinen sind Programme in funktionalen Programmiersprachen verständlicher als in imperativen Programmiersprachen, auch deshalb, da sie „mathematischer“ sind.

- Funktionale Programme sind im Allgemeinen mathematisch einfacher handhabbar als imperative Programme, da kein Zustand betrachtet werden muss. Aussagen wie “mein Programm funktioniert korrekt” oder “Programm A und Programm B verhalten sich gleich” lassen sich in funktionalen Programmiersprachen oft mit (relativ) einfachen Mitteln nachweisen.

- Weniger Laufzeitfehler: Stark und statisch typisierte funktionale Programmiersprachen erlauben es dem Compiler viele Programmierfehler schon während des Compilierens zu erkennen, d.h. viele falsche Programme werden erst gar nicht ausgeführt, da die Fehler schon frühzeitig während des Implementierens entdeckt und beseitigt werden können.

- Testen und Debuggen ist einfacher in (reinen) funktionalen Programmiersprachen, da keine Seiteneffekte und Zustände berücksichtigt werden müssen. Beim Debuggen kann so ein Fehler unabhängig vom Speicherzustand o.ä. reproduziert, gefunden und beseitigt werden. Beim Testen ist es sehr einfach möglich, einzelne Funktionen unabhängig voneinander zu testen, sofern referentielle Transparenz gilt.

- Wiederverwendbarkeit: Funktionen höherer Ordnung (d.h. Funktionen, die Funktionen als Argumente verwenden und auch als Ergebnisse liefern können) erlauben eine hohe Abstraktion, d.h. man kann allgemeine Funktionen implementieren, und diese dann mit entsprechenden Argumenten als Instanz verwenden. Betrachte z.B. die Funktionen `summe` und `produkt`, die die Summe und das Produkt einer Liste von Zahlen berechnen (hier als Pseudo-Code):

```
summe liste =
  if leer(liste) then 0 else
      "erstes Element der Liste"
           +  (summe "Liste ohne erstes Element")

produkt liste =
  if leer(liste)  then 1  else
     "erstes Element der Liste"
         * (produkt "Liste ohne erstes Element")
```

Hier kann man abstrahieren und allgemein eine Funktion implementieren, die die Elemente einer Liste mit einem Operator verknüpft:

```
reduce e f liste =
 if leer(liste) then e else
     f "erstes Element der Liste"
        (reduce e f "Liste ohne erstes Element")
```

Die Funktionen `summe` und `produkt` sind dann nur Spezialfälle:

```
summe liste = reduce 0 (+) liste

produkt liste = reduce 1 (*) liste
```

Die Funktion `reduce` ist eine Funktion höherer Ordnung, denn sie erwartet eine Funktion für das Argument f.

- Einfache Syntax: Funktionale Programmiersprachen haben im Allgemeinen eine einfache Syntax mit wenigen syntaktischen Konstrukten und wenigen Schlüsselwörtern.
- Kürzerer Code: Implementierungen in funktionalen Programmiersprachen sind meist wesentlich kürzer als in imperativen Sprachen.
- Modularität: Die meisten funktionalen Programmiersprachen bieten Modulsysteme, um den Programmcode zu strukturieren und zu kapseln, aber auch die Zerlegung des Problems in einzelne Funktionen alleine führt meist zu einer guten Strukturierung des Codes. Durch Funkti-

onskomposition können dann aus mehreren (kleinen) Funktionen neue Funktionen erstellt werden.

- Parallelisierbarkeit: Da die Reihenfolge innerhalb der Auswertung oft nicht komplett festgelegt ist, kann man funktionale Programme durch relativ einfache Analysen parallelisieren, indem unabhängige (Unter-) Ausdrücke parallel ausgewertet werden.

- Multi-Core Architekturen: Funktionale Programmiersprachen bzw. auch Teile davon liegen momentan im Trend, da sie sich sehr gut für die parallele bzw. nebenläufige Programmierung eignen, da es keine Seiteneffekte gibt. Deshalb können Race Conditions oder Deadlocks oft gar nicht auftreten, da diese direkt mit Speicherzugriffen zusammenhängen.

- Nicht zuletzt gibt es viele Forscher, die sich mit funktionalen Programmiersprachen beschäftigen und dort neueste Entwicklungen einführen. Diese finden oft Verwendung auch in anderen (nicht funktionalen) Programmiersprachen.

Zur weiteren Motivation für funktionale Programmiersprachen sei der schon etwas ältere Artikel "Why Functional Programming Matters" von John Hughes empfohlen (Hughes, 1989). Aktuelle Berichte über die Popularität von funktionalen Programmiersprachen sind beispielsweise:

- John Puopolo, *Introduction to Functional Programming*, Dr. Dobbs, 6. Januar 2010, http://www.drdobbs.com/windows/222200479

- Michael Swaine, *It's Time to Get Good at Functional Programming*, Dr. Doobs, 3. Dezember 2008, http://www.drdobbs.com/tools/212201710

- Christian Kücherer, Uwe Schirmer *Renaissance des F*, iX – Magazin für Informationstechnik, S. 54-57, 2009

1.3 Klassifizierung funktionaler Programmiersprachen

In diesem Abschnitt geben wir einen Überblick über funktionale Programmiersprachen. Um diese zu klassifizieren erläutern wir zunächst einige wesentliche Eigenschaften funktionaler Programmiersprachen:

Pure/Impure: Pure (auch reine) funktionale Programmiersprachen erlauben keine Seiteneffekte, während impure Sprachen Seiteneffekte erlauben.

Meistens gilt, dass pure funktionale Programmiersprachen die nicht-strikte Auswertung (siehe unten) verwenden, während strikte funktionale Programmiersprachen oft impure Anteile haben.

Typsysteme: Es gibt verschiedene Aspekte von Typsystemen anhand derer sich funktionale Programmiersprachen unterscheiden lassen:

stark /schwach: In starken Typsystemen müssen alle Ausdrücke (d.h. alle Programme und Unterprogramme) getypt sein, d.h. der Compiler akzeptiert nur korrekt getypte Programme. Bei schwachen Typsystemen werden (manche) Typfehler vom Compiler akzeptiert und (manche) Ausdrücke dürfen ungetypt sein.

dynamisch/statisch: Bei statischem Typsystem muss der Typ eines Ausdrucks (Programms) zur Compilezeit feststehen, bei dynamischen Typsystemen wird der Typ zur Laufzeit ermittelt. Bei statischen Typsystemen ist im Allgemeinen keine Typinformation zur Laufzeit nötig, da bereits während des Compilierens erkannt wurde, dass das Programm korrekt getypt ist, bei dynamischen Typsystemen werden Typinformation im Allgemeinen zur Laufzeit benötigt. Entsprechend verhält es sich mit Typfehlern: Bei starken Typsystemen treten diese nicht zur Laufzeit auf, bei dynamischen Typsystemen ist dies möglich.

monomorph / polymorph: Bei monomorphen Typsystemen haben Funktionen einen festen Typ, bei polymorphen Typsystemen sind schematische Typen (mit Typvariablen) erlaubt.

first-order / higher-order Bei first-order Sprachen dürfen Argumente von Funktionen nur Datenobjekte sein, bei higher-order Sprachen sind auch Funktionen als Argumente erlaubt.

Auswertung: strikt / nicht-strikt Bei der strikten Auswertung (auch call-by-value oder applikative Reihenfolge genannt), darf die Definitionseinsetzung einer Funktionsanwendung einer Funktion auf Argumente erst erfolgen, wenn sämtliche Argumente ausgewertet wurden. Bei nicht-strikter Auswertung (auch Normalordnung, lazy oder call-by-name bzw. mit Optimierung call-by-need oder verzögerte Reihenfolge genannt) werden Argumente nicht vor der Definitionseinsetzung ausgewertet, d.h. Unterausdrücke werden nur dann ausgewertet, wenn ihr Wert für den Wert des Gesamtausdrucks benötigt wird.

Speicherverwaltung Die meisten funktionalen Programmiersprachen haben eine automatische Speicherbereinigung (Garbage Collector). Bei nichtpuren funktionalen Programmiersprachen mit expliziten Speicherreferenzen kann es Unterschiede geben.

Wir geben eine Übersicht über einige funktionale Programmiersprachen bzw. -sprachklassen. Die Auflistung ist in alphabetischer Reihenfolge:

Agda Lazy Funktionale Sprache, mit dependent types (als Erweiterung von polymophen Typen). Typ-System kann zB Array-Verwendgunen mit Dimensionsangaben formulieren und typisieren. . Aber kann Korrektheitsbehauptungen in den Typen formulieren und Verifikation von Funktionenn nachweisen.

Alice ML: ML Variante, die sich als Erweiterung von SML versteht, mit Unterstützung für Nebenläufigkeit durch sogenannte Futures, call-by-value Auswertung, stark und statisch typisiert, polymorphes Typsystem, entwickelt an der Uni Saarbrücken 2000-2007, http://www.ps.uni-saarland.de/alice/

Clean: Nicht-strikte funktionale Programmiersprache, stark und statisch getypt, polymorphe Typen, higher-order, pure, http://wiki.clean.cs.ru.nl/Clean.

Clojure: relativ neue Sprache, Lisp-Dialekt, der direkt in Java-Bytecode compiliert wird, dynamisch getypt, spezielle Konstrukte für multithreaded Programmierung (software transactional memory system, reactive agent system) http://clojure.org/

Curry Logisch-funktionale Programmiersprache, die auch Nichtdeterminismus erlaubt. https://www.informatik.uni-kiel.de/ mh/FLP/

Common Lisp: Lisp-Dialekt, erste Ausgabe 1984, endgültiger Standard 1994, dynamisch typisiert, auch OO- und prozedurale Anteile, Seiteneffekte erlaubt, strikt, http://common-lisp.net/

Erlang: Strikte funktionale Programmiersprache, dynamisch typisiert, entwickelt von Ericsson für Telefonnetze, sehr gute Unterstützung zur parallelen und verteilten Programmierung, Ressourcen schonende Prozesse, entwickelt ab 1986, open source 1998, hot swapping (Code-Austausch zur Laufzeit), Zuverlässigkeit "nine nines"= 99,9999999 %, http://www.erlang.org/

F#: Funktionale Programmiersprache entwickelt von Microsoft ab 2002, sehr angelehnt an OCaml, streng typisiert, auch objektorientierte und imperative Konstrukte, call-by-value, Seiteneffekte möglich, im Visual Studio 2010 offiziell integriert, http://fsharp.net

Haskell: Pure funktionale Programmiersprache, keine Seiteneffekte, strenges und statisches Typsystem, call-by-need Auswertung, Polymorphismus, Komitee-Sprache, erster Standard 1990, Haskell 98: 1999 und nochmal 2003 leichte Revision, neuer Standard Haskell 2010 (veröffentlicht Juli 2010), http://haskell.org

Lisp: (ende 1950er Jahre, von John McCarthy): steht für (List Processing), Sprachfamilie, Sprache in Anlehnung an den Lambda-Kalkül, ungetypt, strikt, bekannte Dialekte Common Lisp, Scheme

ML: (Metalanguage) Klasse von funktionalen Programmiersprachen: statische Typisierung, Polymorphismus, automatische Speicherbereinigung, im Allgemeinen call-by-value Auswertung, Seiteneffekte erlaubt, entwickelt von Robin Milner 1973, einige bekannte Varianten: Standard ML (SML), Lazy ML, Caml, OCaml,

OCaml: (Objective Caml), Weiterentwicklung von Caml (Categorically Abstract Machine Language), ML-Dialekt, call-by-value, stark und statische Typisierung, polymorphes Typsystem, automatische Speicherbereinigung, nicht Seiteneffekt-frei, unterstützt auch Objektorientierung, entwickelt: 1996, http://caml.inria.fr/

Scala: Multiparadigmen Sprache: funktional, objektorientiert, imperativ, 2003 entwickelt, läuft auf der Java VM, funktionaler Anteil: Funktionen höherer Ordnung, Anonyme Funktionen, Currying, call-by-value Auswertung, statische Typisierung, http://www.scala-lang.org/

Scheme: Lisp-Dialekt, streng und dynamisch getypt, call-by-value, Seiteneffekte erlaubt, eingeführt im Zeitraum 1975-1980, letzter Standard aus dem Jahr 2007, http://www.schemers.org/

SML: Standard ML, ML-Variante: call-by-value, entwickelt 1990, letzte Standardisierung 1997, Sprache ist vollständig formal definiert, Referenz-Implementierung: Standard ML of New Jersey http://www.smlnj.org/.

Wir werden in dieser Vorlesung die nicht-strikte funktionale Programmiersprache Haskell verwenden und uns auch von der theoretischen Seite an Haskell orientieren. Als Compiler bzw. Interpreter verwenden wir den Glasgow Haskell Compiler (GHC) bzw. dessen interaktive Variante GHCi. Für die Installation sei die „Haskell Platform“ (http://hackage.haskell.org/platform) empfohlen, da diese neben GHC und GHCi weitere Bibliotheken und Hilfsmittel beinhaltet. Lohnenswert ist auch ein Blick in die Dokumentation der Standardbibliotheken (z.B. online unter http://www.haskell.org/ghc/docs/latest/html/libraries/index.html).

1.4 Inhalte und Ziele

Innerhalb dieser Veranstaltung soll zum Einen ein tiefer gehender Einblick in die funktionale Programmiersprache Haskell erfolgen, zum Anderen sollen die theoretischen Grundlagen von Haskell und funktionalen Programmiersprachen erörtert werden.

Aus praktischer Sicht wird ein Überblick über die verschiedenen Sprachkonstrukte und Merkmale von Haskell gegeben. Im Vergleich zur Veranstaltung „Grundlagen der Programmierung 2“ werden im Rahmen dieser Veranstaltung insbesondere tiefer gehende Details wie Haskells Typklassensystem und die Behandlung von Ein- und Ausgabe in Haskell mittels monadischem IO behandelt. Auf der eher theoretischen Seite werden verschiedene Kernsprachen von Haskell vorgestellt und deren Bezug untereinander und zu Haskell erläutert. Für die Sprachen werden operationale Semantiken eingeführt, um zu verstehen, wie Haskell-Programme ausgewertet werden. Ein weiteres wichtiges Thema behandelt die Typisierung von Haskell-Programmen. Hierbei werden insbesondere zwei Typisierungsverfahren für die polymorphe Typisierung erläutert und wesentliche Eigenschaften dieser Verfahren erörtert.

1.5 Literatur

Es gibt kein Buch, das genau zur Vorlesung passt. Zu Haskell und auch zu einigen weiteren Themen zu Haskell und Funktionalen Programmiersprachen gibt es einige Bücher, die je nach Geschmack und Vorkenntnissen einen Blick wert sind:

Bird, R.
Introduction to Functional Programming using Haskell.

Prentice Hall PTR, 2 edition, 1998.
Gutes Buch, deckt viele Themen der Vorlesung ab.

Davie, A. J. T.
An introduction to functional programming systems using Haskell.
Cambridge University Press, New York, NY, USA, 1992.
Älteres aber gutes Buch zur Programmierung in Haskell.

Thompson, S.
Haskell: The Craft of Functional Programming.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
Guter Überblick über Haskell

Will Kurt
Get Programming with Haskell.
Manning Publications, 2018.
Aktuelle gute Einführung in Haskell. Auch als E-Book und Online-Buch.

Hutton, G.
Programming in Haskell.
Cambridge University Press, 2007.
Knapp geschriebene Einführung in Haskell.

Chakravarty, M. & Keller, G.
Einführung in die Programmierung mit Haskell.
Pearson Studium, 2004
Einführung in die Informatik mit Haskell, eher ein Anfängerbuch.

Bird, R. & Wadler, P.
Introduction to Functional Programming.
Prentice-Hall International Series in Computer Science. Prentice-Hall International, Upper Saddle River, NJ, USA, 1988.
Ältere aber gute Einführung in die funktionale Programmierung, Programmiersprache: Miranda.

O'Sullivan, B., Goerzen, J., & Stewart, D.
Real World Haskell.
O'Reilly Media, Inc, 2008.
Programmierung in Haskell für Real-World Anwendungen

Pepper, P..
Funktionale Programmierung in OPAL, ML, HASKELL und GOFER.
Springer-Lehrbuch, 1998. ISBN 3-540-64541-1.
Einführung in die funktionale Programmierung mit mehreren Programmiersprachen.

Pepper, P. & Hofstedt, P.
Funktionale Programmierung – Weiterführende Konzepte und Techniken.
Springer-Lehrbuch. ISBN 3-540-20959-X, 2006.
beschreibt einige tiefergehende Aspekte, deckt Teile der Vorlesung ab.

Peyton Jones, S. L.
The Implementation of Functional Programming Languages.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
Sehr gutes Buch über die Implementierung von Funktionalen Programmiersprachen.

Zum Lambda-Kalkül seien u.a. empfohlen:

Barendregt, H. P.
The Lambda Calculus. Its Syntax and Semantics.
North-Holland, Amsterdam, New York, 1984.
Standardwerk zum Lambda-Kalkül, eher Nachschlagewerk, teilweise schwer zu lesen.

Hankin, C.
An introduction to lambda calculi for computer scientists.
Number 2 in Texts in Computing. King's College Publications, London, UK, 2004.
Sehr gute Einführung zum Lambda-Kalkül

Lohnenswert ist auch ein Blick in das Buch:

Pierce, B. C.,
Types and programming languages.
MIT Press, Cambridge, MA, USA, 2002.

das sich allerdings an ML als Programmiersprache orientiert.

2

Funktionale Kernsprachen

In diesem Kapitel werden verschiedene Kernsprachen (für Haskell) erörtert. Teilweise werden wir dabei auch schon auf einige Konstrukte und Eigenschaften von Haskell eingehen. Im anschließenden Kapitel werden wir die wesentlichen Konstrukte und Konzepte der Sprache Haskell genauer darstellen und uns davon überzeugen, dass wir diese in die Kernsprache KFPTSP+seq übersetzen können.

Kernsprachen werden auch oft als *Kalküle* bezeichnet. Sie bestehen im Wesentlichen aus der Syntax, die festlegt welche Ausdrücke in der Sprache gebildet werden dürfen und der Semantik, die angibt, welche Bedeutung die Ausdrücke haben. Diese Kalküle kann man als abgespeckte Programmiersprachen auffassen. „Normale“ Programmiersprachen werden oft während des Compilierens in solche einfacheren Sprachen übersetzt, da diese besser überschaubar sind und sie u.a. besser mathematisch handhabbar sind. Mithilfe einer Semantik kann man (mathematisch korrekte) Aussagen über Programme und deren Eigenschaften nachweisen. Das Gebiet der „Formalen Semantiken“ unterscheidet im Wesentlichen drei Ansätze:

- Eine *axiomatische Semantik* beschreibt Eigenschaften von Programmen mithilfe logischer Axiome bzw. Schlussregeln. Weitere Eigenschaften von Programmen können dann mithilfe von logischen Schlussregeln hergeleitet werden. Im Allgemeinen werden von axiomatischen Semantiken nicht alle Eigenschaften, sondern nur einzelne Eigenschaften der Programme erfasst. Ein prominentes Beispiel für eine axiomatische Semantik ist der Hoare-Kalkül (Hoare, 1969).

- *Denotationale Semantiken* verwenden mathematische Räume um die Bedeutung von Programmen festzulegen. Eine *semantische Funktion* bildet Programme in den entsprechenden mathematischen Raum ab. Für funktionale Programmiersprachen werden als mathematische Räume oft „Domains“, d.h. partiell geordnete Mengen, verwendet. Denotationale Semantiken sind mathematisch elegant, allerdings für umfangrei-

chere Sprachen oft schwer zu definieren.

- Eine *operationale Semantik* legt die Bedeutung von Programmen fest, indem sie definiert wie Programme *ausgewertet* werden. Es gibt verschiedene Möglichkeiten operationale Semantiken zu definieren: *Zustandsübergangssysteme* geben an, wie sich der Zustand der Maschine (des Speichers) beim Auswerten verändert, *Abstrakte Maschinen* geben ein Maschinenmodell zur Auswertung von Programmen an und *Ersetzungssysteme* definieren, wie der Wert von Programmen durch Term-Ersetzungen „ausgerechnet“ werden kann. Man kann operationale Semantik noch in *big-step* und *small-step*-Semantiken unterscheiden. Während small-step-Semantiken die Auswertung in kleinen Schritten festlegen, d.h. Programme werden schrittweise ausgewertet, legen big-step-Semantik diese Auswertung in größeren (meist einem) Schritt fest. Oft erlauben big-step-Semantiken Freiheit bei der Implementierung eines darauf basierenden Interpreters, während bei small-step-Semantiken meist ein Interpreter direkt aus den Regeln ablesbar ist.

Wir werden operationale Semantiken betrachten, die als Ersetzungssysteme aufgefasst werden können und small-step-Semantiken sind. Im ersten Abschnitt dieses Kapitels betrachten wir kurz den Lambda-Kalkül, der ein weit verbreitetes Modell für Programmiersprachen ist. Insbesondere lässt sich Haskell fast gänzlich auf den Lambda-Kalkül zurückführen. Wir werden jedoch verschiedene Erweiterungen des Lambda-Kalküls betrachten, die sich besser als Kernsprachen für Haskell eignen.

2.1 Der Lambda-Kalkül

In diesem Abschnitt betrachten wir den Lambda-Kalkül und werden hierbei verschiedene Auswertungsstrategien (also operationale Semantiken) darstellen. Der Lambda-Kalkül wurde von Alonzo Church in den 1930er Jahren eingeführt (Church, 1941).

Ausdrücke des Lambda-Kalküls können mit dem Nichtterminal **Expr** mithilfe der folgenden kontextfreien Grammatik gebildet werden:

$$\mathbf{Expr} ::= V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}\ \mathbf{Expr})$$

Hierbei ist V ein Nichtterminal, welches eine Variable (aus einer unendlichen Menge von Variablennamen) generiert. $\lambda x.s$ wird als *Abstraktion* bezeichnet.

Durch den Lambda-Binder λx wird die Variable x innerhalb des Unterausdrucks s (den man als *Rumpf* der Abstraktion bezeichnet) gebunden, d.h. umgekehrt: Der Gültigkeitsbereich von x ist s. Eine Abstraktion wirkt wie eine anonyme Funktion, d.h. wie eine Funktion ohne Namen. Z.B. kann die Identitätsfunktion $id(x) = x$ im Lambda-Kalkül durch die Abstraktion $\lambda x.x$ dargestellt werden.

Das Konstrukt $(s\ t)$ wird als *Anwendung* (oder auch *Applikation*) bezeichnet. Mithilfe von Anwendungen können Funktionen auf Argumente angewendet werden. Hierbei darf sowohl die Funktion (der Ausdruck s) als auch das Argument t ein beliebiger Ausdruck des Lambda-Kalküls sein. D.h. insbesondere auch, dass Abstraktionen selbst Argumente von Anwendungen sein dürfen. Deshalb spricht man auch vom Lambda-Kalkül höherer Ordnung (bzw. higher-order Lambda-Kalkül). Z.B. kann man die Identitätsfunktion auf sich selbst anwenden, d.h. die Anwendung $id(id)$ kann in Lambda-Notation als $(\lambda x.x)\ (\lambda x.x)$ geschrieben werden.

Um Klammern zu sparen legen wir die folgenden Assoziativitäten und Prioritäten fest: Die Anwendung ist links-assoziativ, d.h. $s\ t\ r$ entspricht $((s\ t)\ r)$ und *nicht* $(s\ (t\ r))$. Der Rumpf einer Abstraktion erstreckt sich soweit wie möglich, z.B. entspricht $\lambda x.s\ t$ dem Ausdruck $\lambda x.(s\ t)$ und *nicht* dem Ausdruck $((\lambda x.s)\ t)$. Als weitere Abkürzung verwenden wir die Schreibweise $\lambda x_1, \ldots, x_n.t$ für die geschachtelten Abstraktionen $\lambda x_1.(\lambda x_2. \ldots (\lambda.x_n.t) \ldots))$.

Beispiel 2.1.1. *Einige prominente Ausdrücke des Lambda-Kalküls sind:*

$$
\begin{array}{lcl}
I & := & \lambda x.x \\
K & := & \lambda x.\lambda y.x \\
K_2 & := & \lambda x.\lambda y.y \\
\Omega & := & (\lambda x.(x\ x))\ (\lambda x.(x\ x)) \\
Y & := & \lambda f.(\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x))) \\
S & := & \lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z) \\
Y' & := & \lambda f.(\lambda x.(\lambda y.(f\ (x\ x\ y))))\ (\lambda x.(\lambda y.(f\ (x\ x\ y))))
\end{array}
$$

Der I-Kombinator stellt gerade die Identitätsfunktion dar. Der Kombinator K erwartet zwei Argumente und bildet auf das Erste ab, das Gegenstück dazu ist K_2, der auf das zweite Argument abbildet. Der Ausdruck Ω hat die Eigenschaft, dass dessen Auswertung nicht terminiert (siehe unten). Y Ist ein Fixpunktkombinator, für ihn gilt $Y\ f\ =\ f\ (Y\ f)$[1]. S ist der S-Kombinator des SKI-Kalküls, und Y' ist ein call-by-value Fixpuntkombinator (siehe unten).

[1] Beachte, dass wir eigentlich noch keinen Gleichheitsbegriff für den Lambda-Kalkül definiert

Um die Gültigkeitsbereiche von Variablen formal festzuhalten, definieren wir die Funktionen FV und BV. Für einen Ausdruck t ist $BV(t)$ die Menge seiner *gebundenen Variablen* und $FV(t)$ die Menge seiner *freien Variablen*, wobei diese (induktiv) durch die folgenden Regeln definiert sind:

$$\begin{array}{lcllcl} FV(x) & = & x & BV(x) & = & \emptyset \\ FV(\lambda x.s) & = & FV(s) \setminus \{x\} & BV(\lambda x.s) & = & BV(s) \cup \{x\} \\ FV(s\ t) & = & FV(s) \cup FV(t) & BV(s\ t) & = & BV(s) \cup BV(t) \end{array}$$

Beispiel 2.1.2. *Sei s der Ausdruck $(\lambda x.\lambda y.\lambda w.(x\ y\ z))\ x$. Dann ist gilt $FV(s) = \{x, z\}$ und $BV(s) = \{x, y, w\}$.*

Gilt $FV(t) = \emptyset$ für einen Ausdruck t, so sagen wir t ist *geschlossen* oder auch t ist ein *Programm*. Ist ein Ausdruck t nicht geschlossen, so nennen wir t *offen*. Ein Vorkommen eine Variablen x in einem Ausdruck s ist *gebunden*, falls es im Geltungsbereich eines Binders λx steht, anderenfalls ist das Vorkommen *frei*.

Beispiel 2.1.3. *Sei s der Ausdruck $(\lambda x.\lambda y.\lambda w.(x\ y\ z))\ x$. Wir markieren die Vorkommen der Variablen (nicht an den Bindern): $(\lambda x.\lambda y.\lambda w.(\underbrace{x}_{1}\ \underbrace{y}_{2}\ \underbrace{z}_{3}))\ \underbrace{x}_{4}$. Das Vorkommen von x markiert mit 1 und das Vorkommen von y markiert mit 2 sind gebundene Vorkommen von Variablen. Das Vorkommen von z markiert mit 3 und das Vorkommen von x markiert mit 4 sind freie Vorkommen von Variablen.*

Übungsaufgabe 2.1.4. *Sei s der Ausdruck $(\lambda y.(y\ x))\ (\lambda x.(x\ y))\ (\lambda z.(z\ x\ y))$. Berechne die freien und gebundenen Variablen von s. Welche der Vorkommen von x, y, z sind frei, welche sind gebunden?*

Definition 2.1.5. *Um die operationale Semantik des Lambda-Kalküls zu definieren, benötigen wir den Begriff der* Substitution*: Wir schreiben $s[t/x]$ für den Ausdruck, der entsteht, indem alle freien Vorkommen der Variable x in s durch den Ausdruck t ersetzt werden. Um Namenskonflikte bei dieser Ersetzung zu vermeiden, nehmen wir an, dass $BV(s) \cap FV(t) = \emptyset$ gilt. Unter diesen Annahmen kann man die Substitution formal durch die folgenden Gleichungen definieren:*

$$\begin{array}{lcl} x[t/x] & = & t \\ y[t/x] & = & y, \text{ falls } x \neq y \\ (\lambda y.s)[t/x] & = & \begin{cases} \lambda y.(s[t/x]) & \text{falls } x \neq y \\ \lambda y.s & \text{falls } x = y \end{cases} \\ (s_1\ s_2)[t/x] & = & (s_1[t/x]\ s_2[t/x]) \end{array}$$

haben. Syntaktische Gleichheit gilt für $Y\ f$ und $f\ (Y\ f)$ nicht. In einem späteren Abschnitt werden wir einen Gleichheitsbegriff einführen.

Z.B. ergibt $(\lambda x.z\ x)[(\lambda y.y)/z]$ den Ausdruck $(\lambda x.((\lambda y.y)\ x))$.

Übungsaufgabe 2.1.6. *Sei* $s = (x\ z)\ (\lambda y.x)$ *und* $t = \lambda w.(w\ w)$. *Welchen Ausdruck erhält man für* $s[t/x]$*?*

Kontexte C sind Ausdrücke, wobei genau ein Unterausdruck durch ein Loch (dargestellt mit $[\cdot]$) ersetzt ist. Man kann Kontexte auch mit der folgenden kontextfreien Grammatik definieren:

$$\mathbf{C} = [\cdot] \mid \lambda V.\mathbf{C} \mid (\mathbf{C}\ \mathbf{Expr}) \mid (\mathbf{Expr}\ \mathbf{C})$$

In Kontexte kann man Ausdrücke *einsetzen*, um so einen neuen Ausdruck zu erhalten. Sei C ein Kontext und s ein Ausdruck. Dann ist $C[s]$ der Ausdruck, der entsteht, indem man in C anstelle des Loches den Ausdruck s einsetzt. Diese Einsetzung kann Variablen einfangen. Betrachte als Beispiel den Kontext $C = \lambda x.[\cdot]$. Dann ist $C[\lambda y.x]$ der Ausdruck $\lambda x.(\lambda y.x)$. Die freie Variable x in $\lambda y.x$ wird beim Einsetzen eingefangen.

Nun können wir α-Umbenennung definieren: Ein α-Umbenennungsschritt hat die Form:

$$C[\lambda x.s] \xrightarrow{\alpha} C[\lambda y.s[y/x]] \text{ falls } y \notin BV(C[\lambda x.s]) \cup FV(C[\lambda x.s])$$

Die reflexiv-transitive Hülle solcher α-Umbenennungen heißt α-*Äquivalenz*. Wir unterscheiden α-äquivalente Ausdrücke nicht. Vielmehr nehmen wir an, dass die *Distinct Variable Convention* (DVC) gilt:

> In einem Ausdruck haben alle gebundenen Variablen unterschiedliche Namen die Namen gebundener Variablen sind stets verschieden von Namen freier Variablen.

Mithilfe von α-Umbenennungen kann diese Konvention stets eingehalten werden.

Beispiel 2.1.7. *Betrachte den Ausdruck* $(y\ (\lambda y.((\lambda x.(x\ x))\ (x\ y))))$. *Er erfüllt die DVC nicht, da* x *und* y *sowohl frei als auch gebunden vorkommen. Benennt man die gebundenen Variablen mit frischen Namen um* (*das sind alles* α*-Umbenennungen*)*, erhält man*

$$\begin{array}{ll} & (y\ (\lambda y.((\lambda x.(x\ x))\ (x\ y)))) \\ \xrightarrow{\alpha} & (y\ (\lambda y_1.((\lambda x.(x\ x))\ (x\ y_1)))) \\ \xrightarrow{\alpha} & (y\ (\lambda y_1.((\lambda x_1.(x_1\ x_1))\ (x\ y_1)))) \end{array}$$

Der Ausdruck $(y\ (\lambda y_1.((\lambda x_1.(x_1\ x_1))\ (x\ y_1))))$ *erfüllt die DVC.*

Übungsaufgabe 2.1.8. *Sei* $s = ((\lambda x.(x\ \lambda y.(x\ z)\ (y\ x)))\ (\lambda z.y))$. *Führe* α*-Umbenennungen für* s *durch, so dass der entstehende Ausdruck die DVC erfüllt.*

Jetzt kann man die Substitution korrekt definieren so dass diese in allen Fällen ein korrektes Ergebnis liefert:

Definition 2.1.9. *Wenn* $BV(s) \cap FV(t) = \emptyset$*, dann definiert man* $s[t/x]$ *wie in Definition 2.1.5. Wennn die Bedingung nicht erfüllt ist, dann sei* $s' =_\alpha s$ *eine Umbenennung von* s*, so dass* $BV(s') \cap FV(t) = \emptyset$*. Ein solches* s' *gibt es! Dann definiere* $s[t/x]$ *als* $s'[t/x]$ *entsprechend Definition 2.1.5.*

Die klassische Reduktionsregel des Lambda-Kalküls ist die β-Reduktion, die die Funktionsanwendung auswertet:

$$(\beta) \qquad (\lambda x.s)\ t \to s[t/x]$$

Wenn $s \xrightarrow{\beta} t$, so sagt man auch s *reduziert unmittelbar* zu t.

Beispiel 2.1.10. *Den Ausdruck* $(\lambda x.x)\ (\lambda y.y)$ *kann man* β*-reduzieren:*

$$(\lambda x.x)\ (\lambda y.y) \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

Den Ausdruck $(\lambda y.y\ y\ y)\ (x\ z)$ *kann man* β*-reduzieren:*

$$(\lambda y.y\ y\ y)\ (x\ z) \xrightarrow{\beta} (y\ y\ y)[(x\ z)/y] = (x\ z)\ (x\ z)\ (x\ z)$$

Damit ein Ausdruck nach einer β*-Reduktion die DVC erfüllt muss man umbenennen: Betrachte z.B. den Ausdruck* $(\lambda x.(x\ x))\ (\lambda y.y)$*. Eine* β*-Reduktion ergibt*

$$(\lambda x.(x\ x))\ (\lambda y.y) \xrightarrow{\beta} (\lambda y.y)\ (\lambda y.y)$$

Allerdings erfüllt $(\lambda y.y)\ (\lambda y.y)$ *nicht die DVC.* α*-Umbenennung ergibt jedoch* $(\lambda y_1.y_1)\ (\lambda y_2.y_2)$*, der die DVC erfüllt.*

Allerdings reicht es nicht aus, nur β-Reduktionen auf oberster Ebene eines Ausdrucks durchzuführen, man könnte dann z.B. den Ausdruck $((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z)$ nicht weiter reduzieren. Zur Festlegung der operationalen Semantik muss man deshalb noch definieren, wo im Ausdruck die β-Reduktionen angewendet werden sollen. Betrachte z.B. den Ausdruck $((\lambda x.xx)((\lambda y.y)(\lambda z.z)))$. Er enthält zwei verschiedene Positionen, die man reduzieren könnte (die entsprechenden Unterausdrücke nennt

man *Redex*): $\underline{((\lambda x.xx)((\lambda y.y)(\lambda z.z)))} \rightarrow ((\lambda y.y)(\lambda z.z))\ ((\lambda y.y)(\lambda z.z))$ oder $((\lambda x.xx)\underline{((\lambda y.y)(\lambda z.z))}) \rightarrow ((\lambda x.xx)(\lambda z.z))$.

Diese Festlegung bezeichnet man auch als Reduktionsstrategie.

2.1.1 Normalordnungsreduktion

Die Auswertung in *Normalordnung* (auch *call-by-name Auswertung* und *nicht-strikte Auswertung*) sucht immer den am weitesten oben und am weitesten links stehenden Redex. Formal kann man die Normalordnungsreduktion mithilfe von Reduktionskontexten definieren.

Definition 2.1.11. *Reduktionskontexte* $\mathbf{R}$ *werden durch die folgende Grammatik definiert:*

$$\mathbf{R} ::= [\cdot] \mid (\mathbf{R}\ \mathbf{Expr})$$

Wenn $s \xrightarrow{\beta} t$*, dann ist* $R[s] \xrightarrow{no} R[t]$ *eine* Normalordnungsreduktion (*oft auch call-by-name Reduktionsschritt*).

Übungsaufgabe 2.1.12. *Sei* $s = (\lambda w.w)\ (\lambda x.x)\ ((\lambda y.((\lambda u.y)\ y))\ (\lambda z.z))$*. Gebe alle Reduktionskontexte* R *und Ausdrücke* t *an für die gilt:* $R[t] = s$*. Führe einen Normalordnungsschritt für* s *aus.*

Bemerkung 2.1.13. *Wenn man geschlossene Ausdrücke mittels Normalordnung reduziert, dann kann man im Prinzip auf die Vorsichtsmaßnahme der Variablen-Umbenennung verzichten, denn eine Einfangen von freien Variablen ist dann nicht möglich. Aber: wenn man innerhalb eines Ausdrucks eine Nicht-Normalordnungsreduktion macht, dann braucht man i.a. die Vorsichtsmaßnahmen der Variablenumbenennung um eine korrekte Beta-Reduktion durchzuführen.*

Wir beschreiben ein weiteres alternatives (intuitives) Verfahren: Sei s ein Ausdruck. Markiere zunächst s mit einem Stern: $s^\star$. Wende nun die folgende Verschiebung der Markierung

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

so oft an, wie es geht. Das Ergebnis ist von der Form $(s_1^\star\ s_2\ \ldots\ s_n)$, wobei s_1 keine Anwendung ist. Nun gibt es die folgenden Fälle:

- s_1 ist eine Abstraktion $\lambda x.s_1'$: Falls $n \geq 2$ dann reduziere s wie folgt: $(\lambda x.s_1')\ s_2\ \ldots s_n \xrightarrow{no} (s_1'[s_2/x]\ \ldots s_n)$. Falls $n = 1$, dann ist keine Reduktion möglich, da der Gesamtausdruck eine Abstraktion ist.

- s_1 ist eine Variable. Dann ist keine Reduktion möglich: eine freie Variable wurde entdeckt. Dieser Fall kommt bei geschlossenen Ausdrücken nicht vor.

Wir betrachten als Beispiel den Ausdruck $((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))$. Der Markierungsalgorithmus verschiebt die Markierung wie folgt:

$$((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))^\star \Rightarrow ((\lambda x.\lambda y.x)^\star((\lambda w.w)(\lambda z.z)))$$

Nun ist eine Reduktion möglich, da der mit $\star$ markierte Unterausdruck eine Abstraktion ist und es ein Argument gibt, d.h. die Normalordnungsreduktion wertet aus:

$$((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z))) \xrightarrow{no} \lambda y.((\lambda w.w)(\lambda z.z))$$

Da dieser erste Auswertungsschritt mit einer Abstraktion endet, ist die Normalordnungsreduktion erfolgreich beendet.

Übungsaufgabe 2.1.14. *Werte die folgenden Ausdrücke in Normalordnung aus:*

- $(\lambda f.(\lambda x.f\ (x\ x)))\ (\lambda.f.(\lambda x.f\ (x\ x)))(\lambda w.\lambda z.w)$
- $(\lambda f.(\lambda x.f\ (x\ x)))\ (\lambda.f.(\lambda x.f\ (x\ x)))((\lambda z.z)\ (\lambda w.w))$

Eine Eigenschaft der Normalordnungsreduktion ist, dass diese stets *deterministisch* ist, d.h. für einen Ausdruck s gibt es höchstens einen Ausdruck t (modulo α-Gleichheit), so dass $s \xrightarrow{no} t$. Es gibt auch Ausdrücke für die keine Reduktion möglich ist. Dies sind zum Einen Ausdrücke, bei denen die Auswertung auf eine freie Variable stößt (z.B. ist $(x\ (\lambda y.y))$ nicht reduzibel). Zum Anderen sind dies Abstraktionen, die auch als FWHNFs (functional weak head normal forms, funktionale schwache Kopfnormalformen) bezeichnet werden. D.h. sobald wir eine FWHNF mithilfe von $\xrightarrow{no}$-Reduktionen erreicht haben, ist die Auswertung beendet. Ausdrücke, die man so in eine Abstraktion überführen kann, *konvergieren* (oder alternativ *terminieren*). Wir definieren dies formal, wobei $\xrightarrow{no,+}$ die transitive Hülle von $\xrightarrow{no}$ und $\xrightarrow{no,*}$ die reflexiv-transitive Hülle von $\xrightarrow{no}$ sei[2]

Definition 2.1.15. *Ein Ausdruck s (call-by-name)* konvergiert *genau dann, wenn es eine Folge von call-by-name Reduktionen gibt, die s in eine Abstraktion überführt, wir schreiben dann $s{\Downarrow}$. D.h. $s{\Downarrow}$ gdw. $\exists$Abstraktion $v : s \xrightarrow{no,*} v$. Falls s nicht konvergiert, so schreiben wir $s{\Uparrow}$ und sagen s* divergiert.

[2]Formal kann man definieren: Sei $\rightarrow \subseteq (M \times M)$ eine binäre Relation über einer Menge M,

Haskell hat die call-by-name Auswertung als semantische Grundlage, wobei Implementierungen die verbesserte Strategie der call-by-need Auswertung verwenden, die Doppelauswertungen von Argumenten vermeidet (siehe Abschnitt 2.1.3). Der Lambda-Kalkül selbst ist jedoch syntaktisch zu eingeschränkt, um als Kernsprache für Haskell zu dienen, wir werden später erweiterte Lambda-Kalküle betrachten, die sich besser als Kernsprachen für Haskell eignen. Für die Normalordnungsreduktion gilt die folgende Eigenschaft:

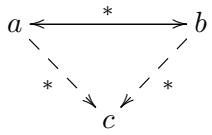
Satz 2.1.16. [*Standardisierungseigenschaft der Normalordnung*] *Sei s ein Lambda-Ausdruck. Wir nehmen an, dass s mit beliebigen β-Reduktionen (an beliebigen Positionen) in eine Abstraktion v überführt werden. Dann gilt $s\Downarrow$.*

Diese Aussage zeigt, dass die call-by-name Auswertung bezüglich der Terminierung eine optimale Strategie ist.

Eine weitere Aussage ist die Konfluenz bzw das Church-Rosser Theorem:

Theorem 2.1.17 (Church-Rosser Theorem)**.** *Für den Lambda-Kalkül gilt (bzgl. der Gleichheit $=_\alpha$:)*

Wenn $a \overset{*}{\leftrightarrow} b$, dann existiert c, so dass $a \overset{*}{\to} c$ und $b \overset{*}{\to} c$



Hierbei bedeutet $\overset{*}{\to}$ eine beliebige Folge von β-Reduktionen (in bel. Kontext) und $\overset{*}{\leftrightarrow}$ eine beliebige Folge von β-Reduktionen (vorwärts und ruckwärts) (in bel. Kontext).

2.1.2 Anwendungsordnung (call-by-value)

Wir betrachten eine weitere wichtige Auswertungsstrategie. Die *Anwendungsordnung* (auch *call-by-value Auswertung* oder *strikte Auswertung*) verlangt, dass

---

dann ist

$$\begin{aligned}
\overset{0}{\to} &:= \{(s,s) \mid s \in M\} \\
\overset{i}{\to} &:= \{(s,t) \mid s \to r \text{ und } r \overset{i-1}{\longrightarrow} t\} \\
\overset{+}{\to} &:= \bigcup_{i>0} \overset{i}{\to} \\
\overset{*}{\to} &:= \bigcup_{i\geq 0} \overset{i}{\to}.
\end{aligned}$$

eine β-Reduktion nur dann durchgeführt werden darf, wenn das Argument eine Abstraktion ist. Wir definieren hierfür die β_{cbv}-Reduktion als:

$$(\beta_{cbv}) \qquad (\lambda x.s)\ v \to s[v/x], \text{ wobei } v \text{ eine Abstraktion oder Variable}$$

Jede β_{cbv}-Reduktion ist auch eine β-Reduktion. Die Umkehrung gilt nicht. Wie bei der Normalordnungsreduktion definieren wir eine Anwendungsordnungsreduktion mithilfe von Reduktionskontexten. Diese müssen jedoch angepasst werden, damit Argumente vor dem Einsetzen ausgewertet werden.

Definition 2.1.18. *Call-by-value Reduktionskontexte* $\mathbf{E}$ *werden durch die folgende Grammatik generiert:*

$$\mathbf{E} ::= [\cdot] \mid (\mathbf{E}\ \mathbf{Expr}) \mid ((\lambda V.\mathbf{Expr})\ \mathbf{E})$$

Wenn $s \xrightarrow{\beta_{cbv}} t$*, dann ist* $E[s] \xrightarrow{ao} E[t]$ *eine* Anwendungsordnungsreduktion (*auch call-by-value Reduktionsschritt*).

Alternativ kann man die Suche nach dem Redex einer Anwendungsordnungsreduktion auch wie folgt definieren: Sei e ein Ausdruck, starte mit $s^\star$ und wende die folgenden beiden Regeln solange an, bis es nicht mehr geht:

$$\begin{array}{ll} (s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2) & \\ (v^\star\ s) \Rightarrow (v\ s^\star) & \text{falls } v \text{ eine Abstraktion ist} \\ & \text{und } s \text{ keine Abstraktion oder Variable ist} \end{array}$$

Ist danach eine Variable mit * markiert, so ist keine Reduktion möglich, da eine freie Variable gefunden wurde. Ist s selbst markiert, dann ist s eine Variable oder eine Abstraktion, und es ist keine Reduktion möglich. Anderenfalls ist eine Abstraktion markiert, dessen direkter Oberterm eine Anwendung ist, wobei das Argument eine Abstraktion oder eine Variable ist. Die Anwendungsordnungsreduktion reduziert diese Anwendung.

Wir betrachten als Beispiel den Ausdruck $((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))$. Die Redexsuche mit Markierung verschiebt die Markierung wie folgt:

$$\begin{aligned} & ((\lambda x.\lambda y.x)\ ((\lambda w.w)\ (\lambda z.z)))^\star \\ \Rightarrow & ((\lambda x.\lambda y.x)^\star\ ((\lambda w.w)\ (\lambda z.z))) \\ \Rightarrow & ((\lambda x.\lambda y.x)\ ((\lambda w.w)\ (\lambda z.z))^\star) \\ \Rightarrow & ((\lambda x.\lambda y.x)\ ((\lambda w.w)^\star\ (\lambda z.z))) \end{aligned}$$

Die erste Reduktion erfolgt daher im Argument:

$$((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z))) \xrightarrow{ao} (\lambda x.\lambda y.x)(\lambda z.z)$$

Anschließend ist das Argument ein Wert, und die oberste Anwendung wird reduziert:

$$(\lambda x.\lambda y.x)(\lambda z.z) \xrightarrow{ao} \lambda y.\lambda z.z$$

Nun hat man eine Abstraktion erhalten, und die Auswertung in Anwendungsordnung damit erfolgreich beendet.

Auch die Anwendungsordnungsreduktion ist deterministisch, also eindeutig. Werte sind ebenfalls FWHNFs (also Abstraktionen). Der Begriff der Konvergenz ergibt sich wie folgt:

Definition 2.1.19. *Ein Ausdruck s* konvergiert in Anwendungsordnung *genau dann, wenn es eine Folge von Anwendungsordnungsreduktion gibt, die s in eine Abstraktion überführt, wir schreiben dann $s{\Downarrow_{ao}}$. D.h. $s{\Downarrow_{ao}}$ gdw. $\exists$Abstraktion $v : s \xrightarrow{ao,*} v$. Falls s nicht konvergiert, so schreiben wir $s{\Uparrow_{ao}}$ und sagen s* divergiert in Anwendungsordnung.

Satz 2.1.16 zeigt sofort: $s{\Downarrow_{ao}} \implies s{\Downarrow}$. Die Umkehrung gilt nicht, da es Ausdrücke gibt, die in Anwendungsordnung divergieren, bei Auswertung in Normalordnung jedoch konvergieren:

Beispiel 2.1.20. *Betrachte den Ausdruck $\Omega := (\lambda x.x\ x)\ (\lambda x.x\ x)$. Es lässt sich leicht nachprüfen, dass $\Omega \xrightarrow{no} \Omega$ als auch $\Omega \xrightarrow{ao} \Omega$. Daraus folgt sofort, dass $\Omega{\Uparrow}$ und $\Omega{\Uparrow_{ao}}$. Betrachte nun den Ausdruck $t := ((\lambda x.(\lambda y.y))\ \Omega)$. Dann gilt $t \xrightarrow{no} \lambda y.y$, d.h. $t{\Downarrow}$. Da die Auswertung in Anwendungsordnung jedoch zunächst das Argument Ω auswerten muss, gilt $t{\Uparrow_{ao}}$.*

Trotz dieser eher schlechten Eigenschaft der Auswertung in Anwendungsordnung, spielt sie eine wichtige Rolle für Programmiersprachen. Die Reihenfolge der Auswertung ist bei der Anwendungsordnung statisch und modular festgelegt: Wenn man z.B. eine Funktion f auf geschlossenen Ausdrücke s_1, s_2, s_3 anwendet, dann wird zunächst s_1 in Anwendungsordnung ausgewertet, dann s_2, dann s_3 und danach wird f auf die entsprechenden Werte angewendet. Im Gegensatz zur Normalordnung: wenn man die Normalordnungsreihenfolge der Auswertung von s_i kennt, kann die Reihenfolge der Auswertung von $(f\ s_1\ s_2\ s_3)$ diese Auswertung zB so sein: zunächst wird s_1 ausgewertet, dann abhängig vom Ergebnis s_3 und dann s_2 oder evtl. sogar

s_2 überhaupt nicht. Bei Sharing bzw call-by-need sind die Reihenfolgen noch schwerer vorherzusagen.

Durch die festgelegte Reihenfolge der Auswertung bei call-by-value kann man direkte Seiteneffekte in Sprachen mit strikter Auswertung zulassen. Einige wichtige Programmiersprachen mit strikter Auswertung sind die strikten funktionalen Programmiersprachen ML (mit den Dialekten SML, OCaml), Scheme und Microsofts F#.

2.1.3 Verzögerte Auswertung

Die verzögerte Auswertung (auch call-by-need Auswertung) stellt eine Optimierung der Auswertung in Normalordnung dar. Wir stellen in diesem Abschnitt eine einfache Variante der verzögerten Auswertung dar. Wie wir sehen werden, ist die Auswertung trotzdem kompliziert. Deshalb werden wir im Anschluss nicht mehr auf die call-by-need Auswertung eingehen (insbesondere bei syntaktisch reicheren Kernsprachen). Zur einfacheren Darstellung erweitern wir die Syntax des Lambda-Kalküls um `let`-Ausdrücke:

$$\mathbf{Expr} ::= V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}\ \mathbf{Expr}) \mid \texttt{let}\ V = \mathbf{Expr}\ \texttt{in}\ \mathbf{Expr}$$

Der Bindungsbereich von x in $\texttt{let}\ x = s\ \texttt{in}\ t$ ist t, d.h. es muss gelten $x \notin FV(s)$. Daher ist dieses `let` *nicht* rekursiv[3]. Reduktionskontexte $\mathbf{R}_{need}$ der verzögerten Auswertung werden durch die folgende Grammatik gebildet, wobei zwei weitere Kontextklassen zur Hilfe genommen werden:

$$\begin{array}{rcl} \mathbf{R}_{need} & ::= & \mathbf{LR}[\mathbf{A}] \mid \mathbf{LR}[\texttt{let}\ x = \mathbf{A}\ \texttt{in}\ \mathbf{R}_{need}[x]] \\ \mathbf{A} & ::= & [\cdot] \mid (\mathbf{A}\ \mathbf{Expr}) \\ \mathbf{LR} & ::= & [\cdot] \mid \texttt{let}\ V = \mathbf{Expr}\ \texttt{in}\ \mathbf{LR} \end{array}$$

Die **A**-Kontexte entsprechen hierbei gerade den call-by-name Reduktionskontexten, d.h. deren Loch ist immer rechts in einer Anwendung. Die **LR**-Kontexte haben ihr Loch immer im `in`-Ausdruck eines `let`-Ausdrucks[4]. Die Reduktionskontexte springen zunächst möglichst weit in die `in`-Ausdrücke eines (möglicherweise verschachtelten) `let`-Ausdrucks, in `let`-Bindungen wird zurück gesprungen, wenn der Wert einer Bindung $x = t$ benötigt wird.

Statt der (β)-Reduktion werden die folgenden Reduktionsregeln verwendet, genauer jede der folgenden Reduktionen ist ein *verzögerter Auswertungs-*

[3] Beachte: In Haskell sind allgemeinere `let`-Ausdrücke erlaubt, die u.A. rekursiv sein dürfen.
[4] Der Name **LR** spiegelt das wider, er steht für „let rechts“

schritt, den wir mit $\xrightarrow{need}$ notieren:

$$
\begin{array}{ll}
(lbeta) & R_{need}[(\lambda x.s)\ t] \to R_{need}[\texttt{let}\ x = t\ \texttt{in}\ s] \\
(cp) & LR[\texttt{let}\ x = \lambda y.s\ \texttt{in}\ R_{need}[x]] \to LR[\texttt{let}\ x = \lambda y.s\ \texttt{in}\ R_{need}[\lambda y.s]] \\
(llet) & LR[\texttt{let}\ x = (\texttt{let}\ y = s\ \texttt{in}\ t)\ \texttt{in}\ R_{need}[x]] \\
 & \qquad \to LR[\texttt{let}\ y = s\ \texttt{in}\ (\texttt{let}\ x = t\ \texttt{in}\ R_{need}[x])] \\
(lapp) & R_{need}[(\texttt{let}\ x = s\ \texttt{in}\ t)\ r] \to R_{need}[\texttt{let}\ x = s\ \texttt{in}\ (t\ r)]
\end{array}
$$

Wir erläutern die Regeln: Anstelle der einsetzenden (β)-Reduktion wird die $(lbeta)$-Reduktion verwendet, die das Argument nicht einsetzt, sondern mit einer neuen `let`-Bindung „speichert“. Die (cp)-Reduktion kopiert dann eine solche Bindung, falls sie benötigt wird, allerdings muss vorher die rechte Seite der Bindung zu einer Abstraktion ausgewertet werden. Die Regeln $(llet)$ und $(lapp)$ dienen dazu, die `let`-Verschachtelungen zu justieren.

Einen Markierungsalgorithmus zur Redexsuche kann man wie folgt definieren: Für einen Ausdruck s starte mit $s^\star$. Der Algorithmus benutzt weitere Markierungen $^\diamond$ und $^\circledcirc$, die Notation $\star \vee \diamond$ meint dabei, dass die Markierung $\star$ oder $\diamond$ sein kann. Die folgenden Regeln zur Markierungsverschiebung werden anschließend solange auf s angewendet wie möglich, wobei falls Regel (2) und Regel (3) anwendbar sind, immer Regel (2) angewendet wird.

$$
\begin{array}{ll}
(1) & (\texttt{let}\ x = s\ \texttt{in}\ t)^\star \Rightarrow (\texttt{let}\ x = s\ \texttt{in}\ t^\star) \\
(2) & (\texttt{let}\ x = y^\diamond\ \texttt{in}\ C[x^\circledcirc]) \Rightarrow (\texttt{let}\ x = y^\diamond\ \texttt{in}\ C[x]) \\
(3) & (\texttt{let}\ x = s\ \texttt{in}\ C[x^{\star\vee\diamond}]) \Rightarrow (\texttt{let}\ x = s^\diamond\ \texttt{in}\ C[x^\circledcirc]) \\
(4) & (s\ t)^{\star\vee\diamond} \Rightarrow (s^\diamond\ t)
\end{array}
$$

Zur Erläuterung: Regel (1) springt in den `in`-Ausdruck eines `let`-Ausdrucks. Die Markierung $\star$ wird zu $\diamond$ nachdem einmal in die Funktionsposition einer Anwendung bzw. in eine `let`-Bindung gesprungen wurde. Hierdurch wird verhindert, dass anschließend wieder `let`-Ausdrücke mit Regel (1) gesprungen werden kann. Regel (3) führt die Markierung $\circledcirc$ ein, um das Kopierziel einer evtl. (cp)-Reduktion zu markieren. Regel (2) verschiebt das Kopierziel, wenn eine `let`-Bindung der Form $x = y$ gefunden wurde: In diesem Fall wird zunächst in die Bindung kopiert (y ersetzt).

Nach Abschluss des Markierungsalgorithmus werden die Reduktionsregeln (falls möglich) wie folgt angewendet (wir geben hier nur den Redex an,

ein äußerer Kontext ist durchaus möglich):

$$
\begin{array}{ll}
(lbeta) & ((\lambda x.s)^{\diamond}\ t) \to \texttt{let}\ x = t\ \texttt{in}\ s \\
(cp) & \texttt{let}\ x = (\lambda y.s)^{\diamond}\ \texttt{in}\ C[x^{\circledcirc}]] \to \texttt{let}\ x = \lambda y.s\ \texttt{in}\ C[\lambda y.s] \\
(llet) & \texttt{let}\ x = (\texttt{let}\ y = s\ \texttt{in}\ t)^{\diamond}\ \texttt{in}\ C[x^{\circledcirc}] \to \texttt{let}\ y = s\ \texttt{in}\ (\texttt{let}\ x = t\ \texttt{in}\ C[x])] \\
(lapp) & ((\texttt{let}\ x = s\ \texttt{in}\ t)^{\diamond}\ r) \to \texttt{let}\ x = s\ \texttt{in}\ (t\ r)
\end{array}
$$

Wir betrachten ein Beispiel.

Beispiel 2.1.21. *Die verzögerte Auswertung (mit Markierungsalgorithmus) des Ausdrucks* $\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)\ x)$ *ist:*

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)\ x))^{\star} \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)\ x)^{\star}) \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta} & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^{\star} \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^{\star}) \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y^{\star})) \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc})) \\
\Rightarrow & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y)) \\
\Rightarrow & (\texttt{let}\ x = ((\lambda u.u)\ (\lambda w.w))^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\Rightarrow & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta} & (\texttt{let}\ x = (\texttt{let} u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^{\star} \\
\Rightarrow & (\texttt{let}\ x = (\texttt{let} u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^{\star}) \\
\Rightarrow & (\texttt{let}\ x = (\texttt{let} u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc})) \\
\Rightarrow & (\texttt{let}\ x = (\texttt{let} u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y)) \\
\Rightarrow & (\texttt{let}\ x = (\texttt{let} u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet} & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)))^{\star} \\
\Rightarrow & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^{\star}) \\
\Rightarrow & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^{\star})) \\
\Rightarrow & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y^{\star}))) \\
\Rightarrow & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc}))) \\
\Rightarrow & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y))) \\
\Rightarrow & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)))
\end{aligned}
$$

$$
\begin{aligned}
&\Rightarrow(\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)))\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)))\\
\xrightarrow{need,cp}&(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)))^{\star}\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^{\star})\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^{\star}))\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y^{\star})))\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc})))\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y)))\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)))\\
\xrightarrow{need,cp}&(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y)))^{\star}\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y))^{\star})\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y)^{\star}))\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y^{\star})))\\
&\Rightarrow(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)^{\diamond}\ \texttt{in}\ y^{\circledcirc})))\\
\xrightarrow{need,cp}&(\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ (\lambda w.w))))
\end{aligned}
$$

FWHNFs für die call-by-need Auswertung sind Ausdrücke der Form $LR[\lambda x.s]$, d.h. Abstraktionen, die von einer `let`-Umgebung umgeben sein dürfen.

Definition 2.1.22. *Ein Ausdruck s* call-by-need konvergiert (*geschrieben als $s{\Downarrow_{need}}$), gdw. er mit einer Folge von $\xrightarrow{need}$-Reduktionen in eine FWHNF überführt werden kann, d.h. $s{\Downarrow_{need}} \iff \exists$ FWHNF $v : s \xrightarrow{need,*} v$*

Man kann zeigen, dass sich die Normalordnungsreduktion und die verzögerte Auswertung bezüglich der Konvergenz gleich verhalten, d.h.

Satz 2.1.23. *Sei s ein (`let`-freier) Ausdruck, dann gilt $s{\Downarrow} \iff s{\Downarrow_{need}}$.*

2.1.4 Programmieren mit Let-Ausdrücken in Haskell

Als kleinen Einschub betrachten wir `let`-Ausdrücke in Haskell und einige Programmiertechniken im Bezug zu diesen. `let`-Ausdrücke in Haskell sind viel allgemeiner, als die im letzten Abschnitt verwendeten `let`-Ausdrücke im call-by-need Lambda-Kalkül.

2.1.4.1 Lokale Funktionsdefinitionen mit `let`

Let-Ausdrücke in Haskell können für *lokale Funktionsdefinitionen* innerhalb von (globalen) Funktionsdefinitionen verwendet werden, d.h. die Syntax ist

$$\begin{array}{ll} \texttt{let} & f_1\ x_{1,1}\ \ldots\ x_{1,n_1} = e_1 \\ & f_2\ x_{2,1}\ \ldots\ x_{2,n_2} = e_2 \\ & \ldots \\ & f_m\ x_{m,1}\ \ldots\ x_{m,n_m} = e_m \\ \texttt{in} \ldots & \end{array}$$

Hierdurch werden die Funktionen $f_1, \ldots, f_m$ definiert. Ein einfaches Beispiel ist

```
f x y =
   let quadrat z = z*z
   in quadrat x + quadrat y
```

Die Funktionen dürfen allerdings auch verschränkt rekursiv sein, d.h. z.B. darf e_2 Aufrufe von f_1, f_2 und f_3 enthalten usw. Ein Beispiel ist

```
quadratfakultaet x =
   let quadrat z = z*z
       fakq    0 = 1
       fakq    x = (quadrat x)*fakq (x-1)
   in fakq x
```

Hierbei ist die Einrückung wichtig. Definiert man eine „0-stellige Funktion“ (d.h. ohne Argumente), dann wird der entsprechende Ausdruck „geshared“, d.h. durch die Verwendung von `let` in Haskell kann man das Sharing explizit angeben[5]. Ein einfaches Beispiel ist:

```
verdopplefak x =
   let fak 0  = 1
       fak x  = x*fak (x-1)
       fakx   = fak x
   in fakx + fakx
```

[5] Ein schlauer Compiler kann dieses Sharing u.U. wieder aufheben oder ein solches Einführen (diese Transformation wird i.A. als „Common Subexpression Elimination“ bezeichnet).

Ein Aufruf `verdopplefak 100` wird nur einmal `fak 100` berechnen und anschließend das Ergebnis verdoppeln. Testet man diese Funktion mit großen Werten und im Vergleich dazu die Funktion

```
verdopplefakLangsam x =
   let fak 0  = 1
       fak x  = x*fak (x-1)
   in fak x + fak x
```

so lässt sich schon ein leichter Vorteil in der Laufzeit feststellen.

2.1.4.2 Pattern-Matching mit `let`

Anstelle einer linken Seite $f_i\ x_{i,1}\ \ldots\ x_{i,n_i}$ kann auch ein Pattern stehen, z.B. $(a, b) = \ldots$, damit kann man komfortabel auf einzelne Komponenten zugreifen. Will man z.B. die Summe von 1 bis n und das Produkt von 1 bis n in einer rekursiven Funktion als Paar berechnen, kann man mit einem `let`-Ausdruck auf das rekursive Ergebnis und zurückgreifen und das Paar mittels Pattern in die einzelnen Komponenten zerlegen:

```
sumprod 1 = (1,1)
sumprod n =
   let (s',p') = sumprod (n-1)
   in (s'+n,p'*n)
```

2.1.4.3 Memoization

Mit explizitem Sharing kann man dynamisches Programmieren sehr gut umsetzen. Betrachte z.B. die Berechnung der n-ten Fibonacci-Zahl. Der naive Algorithmus ist

```
fib 0 = 0
fib 1 = 1
fib i = fib (i-1) + fib (i-2)
```

Schlauer (und auch schneller) ist es, sich die bereits ermittelten Werte für `(fib i)` zu merken und nicht jedes mal neu zu berechnen. Man kann eine Liste verwenden und sich dort die Ergebnisse speichern. Diese Liste shared man mit einem `let`-Ausdruck. Das ergibt die Implementierung:

```
-- Fibonacci mit Memoization

fibM i =
     let fibs = [0,1] ++ [fibs!!(i-1) + fibs!!(i-2) | i <- [2..]]
     in fibs!!i -- i-tes Element der Liste fibs
```

Vergleicht man `fib` und `fibM` im Interpreter, so braucht `fib` für den Wert 30 schon einige Sekunden und die Laufzeit wächst sehr schnell an bei größeren Werten. Eine Tabelle mit gemessenen Laufzeiten:

n	gemessene Zeit im ghci für `fib` n
30	9.75sec
31	15.71sec
32	25.30sec
33	41.47sec
34	66.82sec
35	108.16sec

`fibM` verbraucht für diese kleine Zahlen nur wenig Zeit. Einige gemessene Werte:

n	gemessene Zeit im ghci für `fibM` n
1000	0.05sec
10000	1.56sec
20000	7.38sec
30000	23.29sec

Die Definition von `fibM` kann noch so verbessert werden, dass die Liste der Fibonacci-Zahlen `fibs` auch über mehrere Aufrufe der Funktion gespeichert wird. Dies funktioniert, indem man das Argument i „über die Liste zieht“:

```
fibM'  =
     let fibs = [0,1] ++ [fibs!!(i-1) + fibs!!(i-2) | i <- [2..]]
     in \i -> fibs!!i -- i-tes Element der Liste fibs
```

Führt man nun zweimal den gleichen Aufrufe nacheinander im Interpreter aus (ohne neu zu laden), so verbraucht der zweite Aufruf im Grunde gar keine Zeit, da der Wert bereits berechnet wurde (allerdings bleibt der Speicher dann mit der Liste blockiert.)

```
*Main> fibM' 20000 ↩
...
(7.27 secs, 29757856 bytes)
*Main> fibM' 20000 ↩
...
(0.06 secs, 2095340 bytes)
```

2.1.4.4 `where`-Ausdrücke

Haskell bietet neben `let`-Ausdrücken auch `where`-Ausdrücke zur Programmierung an. Diese verhalten sich ähnlich zu `let`-Ausdrücken sind jedoch Funktionsrümpfen nachgestellt. Z.B. könnten wir definieren:

```
sumprod' 1 = (1,1)
sumprod' n = (s'+n,p'*n)
  where (s',p') = sumprod' (n-1)
```

Allerdings ist zu beachten, dass (`let` ... `in` e) einen Ausdruck darstellt, während e `where`... kein Ausdruck ist, sondern das `where` gültig für die Funktionsdefinition ist, und daher für alle Guards wirken kann. Z.B. kann man definieren:

```
f x
 | x == 0    = a
 | x == 1    = a*a
 | otherwise = a*f (x-1)
 where a = 10
```

während man einen `let`-Ausdruck nicht um die Guards herum schreiben kann. Andererseits darf man in einem `where`-Ausdruck nur die Parameter der Funktion, und nicht durch den Funktionsrumpf eingeführte Parameter verwenden, z.B. ist

```
f x = \y -> mul
   where mul = x * y
```

kein gültiger Ausdruck, da y nur im Rumpf nicht aber im `where`-Ausdruck gebunden ist. Oft ist die Verwendung von `let`- oder `where` jedoch auch Geschmackssache.

2.1.5 Gleichheit von Programmen

Bisher haben wir den call-by-name, den call-by-value und den call-by-need Lambda-Kalkül vorgestellt, die Syntax und die jeweilige operationale Semantik definiert. Wir können damit (Lambda-Kalkül-)Programme ausführen (d.h. auswerten) allerdings fehlt noch ein Begriff der Gleichheit von Programmen. Dieser ist z.B. notwendig um nachzuprüfen, ob ein Compiler korrekt optimiert, d.h. ob ein ursprüngliches Programm und ein optimiertes Programm gleich sind.

Nach dem Leibnizschen Prinzip sind zwei Dinge gleich, wenn sie die gleichen Eigenschaften bezüglich aller Eigenschaften besitzen. Dann sind die Dinge beliebig in jedem Kontext austauschbar. Für Programmkalküle kann man das so fassen: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, egal in welchem Kontext man sie benutzt. Formaler ausgedrückt sind s und t gleich, wenn für alle Kontexte C gilt: $C[s]$ und $C[t]$ verhalten sich gleich. Hierbei fehlt noch ein Begriff dafür, welches Verhalten wir beobachten möchten. Für deterministische Sprachen reicht die Beobachtung der Terminierung, die wir bereits als Konvergenzbegriff für die Kalküle definiert haben.

Wir definieren nach diesem Muster die *Kontextuelle Gleichheit*, wobei man zunächst eine Approximation definieren kann und die Gleichheit dann die Symmetrisierung der Approximation darstellt.

- $s \leq_c t$ gdw. $\forall C : C[s]{\Downarrow} \implies C[t]{\Downarrow}$
- $s \sim_c t$ gdw. $s \leq_c t$ und $t \leq_c s$

Definition 2.1.24 (Kontextuelle Approximation und Gleichheit)**.** *Für den call-by-name Lambda-Kalkül definieren wir die* Kontextuelle Approximation $\leq_c$ *und die* Kontextuelle Gleichheit $\sim_c$ *als:*

- $s \leq_c t$ *gdw.* $\forall C : C[s]{\Downarrow} \implies C[t]{\Downarrow}$
- $s \sim_c t$ *gdw.* $s \leq_c t$ *und* $t \leq_c s$

Analog definieren wir für den call-by-value Lambda-Kalkül die Kontextuelle Approximation $\leq_{c,ao}$ *und die* Kontextuelle Gleichheit $\sim_{c,ao}$ *als:*

- $s \leq_{c,ao} t$ *gdw.* $\forall C$: *Wenn* $C[s], C[t]$ *geschlossen und* $C[s]{\Downarrow_{ao}}$, *dann auch* $C[t]{\Downarrow_{ao}}$
- $s \sim_{c,ao} t$ *gdw.* $s \leq_{c,ao} t$ *und* $t \leq_{c,ao} s$

Für den call-by-need Lambda-Kalkül mit `let` kann man eine analoge Definition angeben, wir lassen sie jedoch an dieser Stelle weg.

Bei call-by-value Kalkülen macht es einen Unterschied ob man $C[s], C[t]$ geschlossen oder nicht verlangt in der Definition, im Gegensatz zu call-by-name und call-by-need. In call-by-value Kalkülen werden dadurch die richtigen Gleichungen erfasst, die in (geschlossenen) Programmen die Austauschbarkeit von Ausdrücken rechtfertigen.

Die kontextuelle Gleichheit kann als gröbste Gleichheit betrachtet werden (d.h. möglichst viele Programme sind gleich), die offensichtlich unterschiedliche Programme unterscheidet. Eine wichtige Eigenschaft der kontextuellen Gleichheit ist die folgende:

Satz 2.1.25. *$\sim_c$ und $\sim_{c,ao}$ sind* Kongruenzen*, d.h. sie sind Äquivalenzrelationen und kompatibel mit Kontexten, d.h. $s \sim t \implies C[s] \sim C[t]$.*

Die Kongruenzeigenschaft erlaubt es u.a. Unterprogramme eines großen Programms zu transformieren (bzw. optimieren) und dabei ein gleiches Gesamtprogramm zu erhalten (unter der Voraussetzung, dass die lokale Transformation die kontextuelle Gleichheit erhält).

Die kontextuelle Gleichheit liefert einen allgemein anerkannten Begriff der Programmgleichheit. Ein Nachteil der Gleichheit ist, dass Gleichheitsbeweise im Allgemeinen schwierig sind, da man alle Kontexte betrachten muss. Generell ist kontextuelle Gleichheit unentscheidbar, da man mit der Frage, ob $s \sim_c \Omega$ gilt, das Halteproblem lösen würde. Wir gehen nicht genauer auf die Beweistechniken ein. Es ist jedoch noch erwähnenswert, dass Ungleichheit i.a. leicht nachzuweisen ist, da man in diesem Fall nur einen Kontext angeben muss, der Ausdrücke bezüglich ihrer Konvergenz unterscheidet.

Es lässt sich nachweisen, dass die (β)-Reduktion die Gleichheit im call-by-name Lambda-Kalkül erhält, d.h. es gilt $(\beta) \subseteq \sim_c$. Für den call-by-value Lambda-Kalkül gilt $(\beta_{cbv}) \subseteq \sim_{c,ao}$ aber $(\beta) \not\subseteq \sim_{c,ao}$, denn z.B. konvergiert $((\lambda x.(\lambda y.y))\ \Omega)$ im leeren Kontext unter call-by-value Auswertung nicht, während $\lambda y.y$ sofort konvergiert.

Bezüglich der Gleichheiten in beiden Kalkülen gilt keinerlei Beziehung, d.h. $\sim_c \not\subseteq \sim_{c,ao}$ und $\sim_{c,ao} \not\subseteq \sim_c$. Die erste Aussage folgt sofort aufgrund der Korrektheit der (β)-Reduktion. Für die zweite Aussage kann man als Beispiel nachweisen, dass $((\lambda x.(\lambda y.y))\ \Omega) \sim_{c,ao} \Omega$ während $((\lambda x.(\lambda y.y))\ \Omega) \not\sim_c \Omega$.

Mit dem Begriff der Kontextuellen Gleichheit werden wir uns im Rahmen dieser Veranstaltung nur wenig beschäftigen. In der Veranstaltung „M-

CEFP Programmtransformationen und Induktion in funktionalen Programmen“ wird sich hiermit tiefer gehend beschäftigt.

2.1.6 Kernsprachen von Haskell

In den folgenden Abschnitten führen wir verschiedene Kernsprachen (oder Kalküle) ein, die einer Kernsprache von Haskell näher kommen als der Lambda-Kalkül. Nichtsdestotrotz sind alle diese Sprachen Erweiterungen des Lambda-Kalküls. Sämtliche Sprachen beginnen mit dem Kürzel „KFP“, was als „Kern einer Funktionalen Programmiersprache“ gedeutet werden kann.

Bemerkung 2.1.26. *Zunächst stellt sich die Frage, warum sich der Lambda-Kalkül nicht sonderlich als Kernsprache für Haskell eignet. Hierfür gibt es mehrere Gründe:*

- *Der Lambda-Kalkül verfügt nicht über Daten: Will man Daten wie z.B. Zahlen, aber auch kompliziertere Strukturen wie beispielsweise Listen darstellen, so muss man dies durch Funktionen tun. Dies ist zwar möglich (was im Allgemeinen als Church-Kodierung bezeichnet wird), aber man kann Funktionen dann nicht von Daten unterscheiden und außerdem ist diese Methode eher schwer zu überblicken.*
- *Rekursive Funktionen: Man kann zwar im Lambda-Kalkül durch so genannte Fixpunktkombinatoren Rekursion darstellen, allerdings ist auch dies sehr unübersichtlich und die Kodierung aufwändig.*
- *Typisierung: Haskell ist eine polymorph getypte Sprache, der bisher vorgestellte Lambda-Kalkül war ungetypt.*
- `seq`*: Der in Haskell verfügbare Operator* `seq` *lässt sich nicht im reinen Lambda-Kalkül kodieren.*

2.2 Die Kernsprache KFPT

2.2.1 Syntax von KFPT

Als erste Erweiterung beheben wir den ersten Kritikpunkt aus Bemerkung 2.1.26 und fügen *Datenkonstruktoren* und `case`-Ausdrücke zum Lambda-Kalkül hinzu. Wir nehmen an, es gibt eine Menge von Typen, wobei jeder Typ einen Namen hat, z.B. `Bool`, `List`, ... Zu jedem Typ gibt es eine (endliche) Menge von Datenkonstruktoren, die wir im Allgemeinen mit c_i darstellen. Konkret hat z.B. `Bool` die Datenkonstruktoren `True` und `False`, und zum Typ

`List` gehören die Datenkonstruktoren `Nil` und `Cons` (die in Haskell-Notation durch `[]` und `:` (infix) dargestellt werden[6]). Jeder Datenkonstruktor hat eine feste Stelligkeit $\mathrm{ar}(c_i) \in \mathbb{N}_0$ (ar kürzt dabei das englische Wort „arity“ ab). Z.B. ist $\mathrm{ar}(\texttt{True}) = \mathrm{ar}(\texttt{False}) = 0$ und $\mathrm{ar}(\texttt{Nil}) = 0$ und $\mathrm{ar}(\texttt{Cons}) = 2$. Wir fordern stets, dass Datenkonstruktoren in Ausdrücken nur *gesättigt* vorkommen, d.h. es sind stets $\mathrm{ar}(c_i)$ viele Argumente für den Konstruktor c_i angegeben[7].

Definition 2.2.1. *Die Syntax der* Kernsprache KFPT *wird durch folgende Grammatik gebildet, wobei V bzw. V_i Variablen sind:*

$$
\begin{array}{lcl}
\mathbf{Expr} & ::= & V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}_1\ \mathbf{Expr}_2) \\
& \mid & (c_i\ \mathbf{Expr}_1\ \ldots\ \mathbf{Expr}_{\mathrm{ar}(c_i)}) \\
& \mid & (\texttt{case}_{\texttt{Typname}}\ \mathbf{Expr}\ \texttt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n\}) \\
& & \quad \textit{Hierbei ist } \mathbf{Pat}_i \textit{ ein Pattern für Konstruktor i,} \\
& & \quad \mathbf{Pat}_i \to \mathbf{Expr}_i \textit{ heißt auch } \texttt{case}\textit{-Alternative.} \\
& & \quad \textit{Für jeden Konstruktor des Typs } \texttt{Typname} \textit{ kommt} \\
& & \quad \textit{genau eine } \texttt{case}\textit{-Alternative vor} \\
\\
\mathbf{Pat}_i & ::= & (c_i\ V_1 \ldots V_{ar(c_i)}) \textit{ wobei die Variablen } V_i \textit{ alle verschieden sind.}
\end{array}
$$

Hierbei sind die Nebenbedingungen zu beachten: Das `case`*-Konstrukt ist mit einem Typ gekennzeichnet, und die case-Alternativen sind vollständig und disjunkt, d.h. für jeden Konstruktor des entsprechenden Typs kommt genau eine case-Alternative vor, die den Konstruktor abdeckt.*

Das „T“ in KFPT steht für geTyptes `case`. Allerdings ist KFPT ansonsten ungetypt und kann daher auch als „schwach getypt“ bezeichnet werden. KFPT verfügt genau wie der Lambda-Kalkül über Variablen, Abstraktionen und Anwendungen, erweitert ihn jedoch um `case`-Ausdrücke (auch Fallunterscheidungen genannt) und Konstruktoranwendungen. Wir vergleichen KFPT-`case`-Ausdrücke mit Haskells `case`-Ausdrücken: Die Syntax für case-Ausdrücke in Haskell ist ähnlich, wobei $\to$ durch `->` ersetzt ist. In Haskell haben `case`-Ausdrücke keine Typmarkierung (die durch Alternativen abgedeckten Konstruktoren müssen jedoch alle vom gleichem Typ stammen). In

[6] In Haskell ist die übliche Schreibweise für eine n-elementige Liste $[a_1, a_2, \ldots, a_n]$, allerdings ist dies nur eine Abkürzung für die mit `(:)` und `[]` aufgebaute Liste a_1:($a2$:($\ldots$:(a_n:[])$\ldots$)). Beachte, dass `(:)` rechts-assoziativ ist, d.h. a_1:a_2:[] entspricht a_1:(a_2:[]) und *nicht* (a_1:a_2):[] .

[7] Man beachte, dass dies in Haskell nicht immer nötig ist, dort sind ungesättigte Konstruktoranwendungen (außerhalb von Pattern) erlaubt.

Haskell ist es zudem nicht notwendig alle Konstruktoren eines Typs abzudecken. Man erhält dann u.U. einen Fehler zur Laufzeit, wenn keine Alternative vorhanden ist. Z.B.

```
(case True of False -> False) ↩
*** Exception: Non-exhaustive patterns in case
```

Haskell erlaubt in den `case`-Alternativen im Gegensatz zu KFPT auch *geschachtelte* und auch überlappende Pattern. Z.B. ist

```
case [] of {[] -> []; (x:(y:ys)) -> [y]}
```

ein gültiger Haskell-Ausdruck. In Haskell kann man das Semikolon und die geschweiften Klammern weglassen, wenn man die richtige Einrückung beachtet:

```
 case [] of
           [] -> []
           (x:(y:ys)) -> [y]
```

aber das Konstrukt

$$\texttt{case}_{\texttt{List}}\ \texttt{Nil}\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ x\ (\texttt{Cons}\ y\ ys)) \to (\texttt{Cons}\ y\ \texttt{Nil})\}$$

ist *kein* KFPT-Ausdruck, da das geschachtelte Pattern $(\texttt{Cons}\ x\ (\texttt{Cons}\ y\ ys))$ nicht erlaubt ist. Ein Übersetzung von geschachtelten in einfache Pattern ist jedoch möglich durch verschachtelte `case`-Ausdrücke. Der obige Ausdruck kann beispielsweise in KFPT dargestellt als:

```
case_List Nil of {Nil -> Nil;
                  (Cons x z) -> case_List z of
                         {Nil -> Omega
                          (Cons y ys) -> (Cons y Nil)
                         }
                 }
```

Diese Übersetzung zeigt auch, wie man nicht vorhandene `case`-Alternativen behandeln kann: Man fügt diese hinzu, wobei die rechte Seite der Alternative auf einen nicht terminierenden Ausdruck, z.B. Ω,

abgebildet wird. Im weiteren benutzen wir $\bot$ (gesprochen als „bot“) als Repräsentanten für einen geschlossenen nicht terminierenden Ausdruck. Als weitere Abkürzung benutzen wir für `case`-Ausdrücke die Notation $(\texttt{case}_{Typ}\ s\ \texttt{of}\ Alts)$, wobei $Alts$ für irgendwelche – syntaktisch korrekte – `case`-Alternativen steht.

Wir geben einige KFPT-Beispielausdrücke an:

Beispiel 2.2.2. *Eine Funktion, die eine Liste erwartet, und das erste Element der Liste liefert, kann man in* KFPT *darstellen als:*

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to y\}$$

Analog dazu kann eine Funktion, die den Tail einer Liste (also die Liste ohne erstes Element) liefert, definiert werden als

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to ys\}$$

Eine Funktion, die testet, ob eine Liste leer ist, kann definiert werden als:

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{True}; (\texttt{Cons}\ y\ ys) \to \texttt{False}\}$$

Beispiel 2.2.3. *In Haskell gibt es* `if-then-else`*-Ausdrücke der Form* `if` e `then` s `else` t*. Solche Ausdrücke gibt es in* KFPT *nicht, sie können jedoch durch den folgenden* `case`*-Ausdruck vollständig simuliert werden:*

$$\texttt{case}_{\texttt{Bool}}\ e\ \texttt{of}\ \{\texttt{True} \to s; \texttt{False} \to t\}$$

Beispiel 2.2.4. *Paare können in* KFPT *durch einen Typ* `Paar`*, der einen zweistelligen Konstruktor* `Paar` *besitzt, dargestellt werden. Z.B. kann das Paar* $(\texttt{True}, \texttt{False})$ *durch* $(\texttt{Paar}\ \texttt{True}\ \texttt{False})$ *in* KFPT *dargestellt werden. Projektionsfunktionen* fst *und* snd*, die das erste bzw. das zweite Element eines Paares liefern, können definiert werden als Abstraktionen:*

$$\begin{aligned} fst &:= \lambda x.\texttt{case}_{\texttt{Paar}}\ x\ \texttt{of}\ \{(\texttt{Paar}\ a\ b) \to a\} \\ snd &:= \lambda x.\texttt{case}_{\texttt{Paar}}\ x\ \texttt{of}\ \{(\texttt{Paar}\ a\ b) \to b\} \end{aligned}$$

Analog kann man diese Definition auf mehrstellige Tupel erweitern.

In Haskell sind Paare und Tupel bereits vorhanden (eingebaut), die Schreibweise in Haskell ist $(a_1, \ldots, a_n)$*. Beachte: In Haskell gibt es mehrstellige Tupel, aber auch*

0-stellige Tupel (dargestellt als `()`*), allerdings gibt es* keine *einstelligen Tupel*[8].

2.2.2 Freie und gebundene Variablen in KFPT

Verglichen mit dem Lambda-Kalkül enthält KFPT ein weiteres Konstrukt, das Variablen bindet: In einer `case`-Alternative $(c_i\ x_1\ \ldots\ x_{\mathrm{ar}(c_i)}) \to s$ sind die Variablen $x_1, \ldots, x_{\mathrm{ar}(c_i)}$ gebunden. Formal kann man die Menge FV der *freien Variablen* und die Menge BV der *gebundenen Variablen* für einen Ausdruck definieren durch:

$$
\begin{aligned}
&FV(x) && = x \\
&FV(\lambda x.s) && = FV(s) \setminus \{x\} \\
&FV(s\ t) && = FV(s) \cup FV(t) \\
&FV(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)}) && = FV(s_1) \cup \ldots \cup FV(s_{\mathrm{ar}(c_i)}) \\
&FV(\texttt{case}_{Typ}\ t\ \texttt{of} && = FV(t) \cup (\bigcup_{i=1}^{n} (FV(s_i) \setminus \{x_{i,1}, \ldots, x_{i,\mathrm{ar}(c_i)}\})) \\
&\quad \{(c_1\ x_{1,1}\ \ldots\ x_{1,\mathrm{ar}(c_1)}) \to s_1; \\
&\quad \ldots \\
&\quad (c_n\ x_{n,1}\ \ldots\ x_{n,\mathrm{ar}(c_n)}) \to s_n\})
\end{aligned}
$$

$$
\begin{aligned}
&BV(x) && = \emptyset \\
&BV(\lambda x.s) && = BV(s) \cup \{x\} \\
&BV(s\ t) && = BV(s) \cup BV(t) \\
&BV(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)}) && = BV(s_1) \cup \ldots \cup BV(s_{\mathrm{ar}(c_i)}) \\
&BV(\texttt{case}_{Typ}\ t\ \texttt{of} && = BV(t) \cup (\bigcup_{i=1}^{n} (BV(s_i) \cup \{x_{i,1}, \ldots, x_{i,\mathrm{ar}(c_i)}\})) \\
&\quad \{(c_1\ x_{1,1}\ \ldots\ x_{1,\mathrm{ar}(c_1)}) \to s_1; \\
&\quad \ldots \\
&\quad (c_n\ x_{n,1}\ \ldots\ x_{n,\mathrm{ar}(c_n)}) \to s_n\})
\end{aligned}
$$

Wie im Lambda-Kalkül, sagen wir ein Ausdruck ist *geschlossen*, wenn die Menge seiner freien Variablen leer ist, und nennen den Ausdruck anderenfalls *offen*. Durch α-Umbenennungen (wir verzichten auf die formale Definition) ist es stets möglich auch in KFPT, die Distinct Variable Convention zu erfüllen.

Beispiel 2.2.5. *Für den* KFPT*-Ausdruck*

$$s := ((\lambda x.\texttt{case}_{\texttt{List}}\ x\ \texttt{of}\ \{\texttt{Nil} \to x; \texttt{Cons}\ x\ xs\ \to \lambda u.(x\ \lambda x.(x\ u))\})\ x)$$

[8] Der Grund hierfür ist vor allem, dass man die Syntax (a) nicht verwenden kann, da man dann Klammerung und einstellige Tupel nicht unterscheiden kann.

gilt

$$
\begin{aligned}
& FV(s) \\
=& (FV(\lambda x.\texttt{case}_{\texttt{List}}\ x\ \texttt{of}\ \{\texttt{Nil} \to x; \texttt{Cons}\ x\ xs\ \to \lambda u.(x\ \lambda x.(x\ u)))) \cup FV(x) \\
=& (FV(\lambda x.\texttt{case}_{\texttt{List}}\ x\ \texttt{of}\ \{\texttt{Nil} \to x; \texttt{Cons}\ x\ xs\ \to \lambda u.(x\ \lambda x.(x\ u)))) \cup \{x\} \\
=& (FV(\texttt{case}_{\texttt{List}}\ x\ \texttt{of}\ \{\texttt{Nil} \to x; \texttt{Cons}\ x\ xs\ \to \lambda u.(x\ \lambda x.(x\ u))) \setminus \{x\}) \cup \{x\} \\
=& ((FV(x) \cup (FV(x) \setminus \emptyset) \cup (FV(\lambda u.(x\ \lambda x.(x\ u))) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup (\{x\} \setminus \emptyset) \cup (FV(\lambda u.(x\ \lambda x.(x\ u))) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup (FV(\lambda u.(x\ \lambda x.(x\ u))) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((FV(x\ \lambda x.(x\ u)) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((FV(x) \cup FV(\lambda x.(x\ u)) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup FV(\lambda x.(x\ u)) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup (FV(x\ u) \setminus \{x\}) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup (FV(x) \cup FV(u)) \setminus \{x\}) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup (\{x\} \cup \{u\}) \setminus \{x\}) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup (\{x, u\} \setminus \{x\}) \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup ((\{x\} \cup \{u\} \setminus \{u\}) \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup (\{x\} \setminus \{x, xs\})) \setminus \{x\}) \cup \{x\} \\
=& ((\{x\} \cup \emptyset) \setminus \{x\}) \cup \{x\} \\
=& \{x\}.
\end{aligned}
$$

Man kann nachrechnen, dass $BV(s) = \{x, xs, u\}$. D.h. die Variable x kommt sowohl frei als auch gebunden vor. Der Ausdruck s erfüllt die DVC nicht. *Benennt man die gebundenen Variablen von s um, so sieht man die Bindungsbereiche:*

$$s' := ((\lambda x_1.\texttt{case}_{\texttt{List}}\ x_1\ \texttt{of}\ \{\texttt{Nil} \to x_1; \texttt{Cons}\ x_2\ xs\ \to \lambda u.(x_2\ \lambda x_3.(x_3\ u))\})\ x)$$

Dieser Ausdruck erfüllt die DVC. Es gilt: $FV(s') = \{x\}$ und $BV(s') = \{x_1, x_2, xs, x_3, u\}$.

2.2.3 Operationale Semantik für KFPT

Als nächsten Schritt definieren wir Reduktionen für KFPT und im Anschluss definieren wir die Normalordnungsreduktion für KFPT. Sei $s[t/x]$ der Ausdruck s nach Ersetzen aller freien Vorkommen von x durch t und sei $s[t_1/x_1, \ldots, t_n/x_n]$ die (parallele) Ersetzung der freien Vorkommen der Varia-

blen x_i durch die Terme t_i im Ausdruck s.

Definition 2.2.6. *Die Reduktionsregeln* (β) *und* (*case*) *sind in KFPT definiert als:*

$$\begin{array}{ll} (\beta) & (\lambda x.s)\ t \to s[t/x] \\ (\mathit{case}) & \texttt{case}_{Typ}\ (c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ \texttt{of}\ \{\ldots;\ (c\ x_1 \ldots\ x_{\mathrm{ar}(c)}) \to t;\ \ldots\} \\ & \quad \to t[s_1/x_1, \ldots, s_{\mathrm{ar}(c)}/x_{\mathrm{ar}(c)}] \end{array}$$

Die (β)-Reduktion ist analog zum Lambda-Kalkül definiert. Die (case)-Regel dient zur Auswertung eines `case`-Ausdrucks, sofern das Argument eine passende Konstruktoranwendung ist (d.h. c zum entsprechenden Typ gehört).

Beispiel 2.2.7. *Der Ausdruck*

$$(\lambda x.\texttt{case}_{\texttt{Paar}}\ x\ \texttt{of}\ \{(\texttt{Paar}\ a\ b \to a\}))\ (\texttt{Paar}\ \texttt{True}\ \texttt{False})$$

kann durch (β) *und* (*case*)*-Reduktionen in* `True` *überführt werden:*

$$\begin{array}{ll} & (\lambda x.\texttt{case}_{\texttt{Paar}}\ x\ \texttt{of}\ \{(\texttt{Paar}\ a\ b) \to a\})\ (\texttt{Paar}\ \texttt{True}\ \texttt{False}) \\ \xrightarrow{\beta} & \texttt{case}_{\texttt{Paar}}\ (\texttt{Paar}\ \texttt{True}\ \texttt{False})\ \texttt{of}\ \{(\texttt{Paar}\ a\ b) \to a\} \\ \xrightarrow{\mathit{case}} & \texttt{True} \end{array}$$

Wenn $s \to t$ mit einer (β)- oder (case)-Reduktion, dann sagt man s *reduziert unmittelbar* zu t. Kontexte sind auch in KFPT Ausdrücke, die an einer Stelle ein Loch $[\cdot]$ haben, sie sind durch die folgende Grammatik definiert:

$$\begin{array}{ll} \mathbf{C} ::= & [\cdot] \mid \lambda V.\mathbf{C} \mid (\mathbf{C}\ \mathbf{Expr}) \mid (\mathbf{Expr}\ \mathbf{C}) \\ \mid & (c_i\ \mathbf{Expr}_1\ \ldots \mathbf{Expr}_{i-1}\ \mathbf{C}\ \mathbf{Expr}_{i+1}\ \mathbf{Expr}_{\mathrm{ar}(c_i)}) \\ \mid & (\texttt{case}_{\texttt{Typname}}\ \mathbf{C}\ \texttt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n\}) \\ \mid & (\texttt{case}_{\texttt{Typname}}\ \mathbf{Expr}\ \texttt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_i \to \mathbf{C}; \ldots, \mathbf{Pat}_n \to \mathbf{Expr}_n\}) \end{array}$$

Mit $C[s]$ bezeichnen wir den Ausdruck, der entsteht nachdem im Kontext C das Loch durch den Ausdruck s ersetzt wurde. Wendet man eine (β)- oder eine (case)-Reduktion in einem Kontext an, d.h. $C[s] \to C[t]$ wobei $s \xrightarrow{\beta} t$ oder $s \xrightarrow{\mathit{case}} t$, so bezeichnet man den Unterausdruck s in $C[s]$ (mit seiner Position, d.h. an der Stelle des Lochs von C), als *Redex*. Die Bezeichnung Redex ist ein Akronym für Reducible expression.

Im folgenden werden wir die Normalordnungsreduktion für KFPT definieren. Wir führen keine weiteren Strategien (wie bspw. die strikte Auswertung)

ein, diese können jedoch leicht aus den für den Lambda-Kalkül definierten Strategien abgeleitet werden.

Zur Definition einer eindeutigen Reduktionsstrategie verwenden wir (call-by-name) Reduktionskontexte:

Definition 2.2.8. *Reduktionskontexte R in* KFPT *werden durch die folgende Grammatik erzeugt:*

$$\mathbf{R} ::= [\cdot] \mid (\mathbf{R}\ \mathbf{Expr}) \mid (\texttt{case}_{Typ}\ \mathbf{R}\ \texttt{of}\ Alts)$$

Die Normalordnungsreduktion ist in KFPT definiert als:

Definition 2.2.9. *Seien s und t Ausdrücke, so dass s unmittelbar zu t reduziert, dann ist $R[s] \to R[t]$ für jeden Reduktionskontext R eine* Normalordnungsreduktion. *Wir notieren Normalordnungsreduktionen mit $\xrightarrow{no}$.*

Um kenntlich zu machen, welche Reduktionsregel verwendet wurde, benutzen wir auch die Notation $\xrightarrow{no,\beta}$ bzw. $\xrightarrow{no,case}$. Des Weiteren bezeichne $\xrightarrow{no,+}$ die transitive, und $\xrightarrow{no,*}$ die reflexiv-transitive Hülle von $\xrightarrow{no}$.

Beispiel 2.2.10. *Die Reduktion $(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) \to (\lambda y.y)\ (\lambda z.z)$ ist eine Normalordnungsreduktion. Die Reduktion $(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) \to (\lambda x.x)\ (\lambda z.z)$ ist hingegen* keine *Normalordnungsreduktion, da der Kontext $(\lambda x.x)\ [\cdot]$* kein *Reduktionskontext ist.*

Die Auswertung in Normalordnung besteht aus einer Folge von Normalordnungsreduktionen. Sie ist erfolgreich beendet sobald eine WHNF (schwache Kopfnormalform) erreicht ist. Wir definieren verschiedene Arten von Normalformen:

Definition 2.2.11. *Ein* KFPT*-Ausdruck s ist eine*

- Normalform (*NF = normal form*), *wenn s keinerlei (β)- oder (case)-Redexe enthält und eine WHNF ist,*
- Kopfnormalform (*HNF = head normal form*), *wenn s eine Konstruktoranwendung oder eine Abstraktion $\lambda x_1, \ldots x_n.s'$ ist, wobei s' entweder eine Variable oder eine Konstruktoranwendung $(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})$ oder von der Form $(x\ s')$ ist (wobei x eine Variable ist).*
- schwache Kopfnormalform (*WHNF = weak head normal form*), *wenn s eine FWHNF oder eine CWHNF ist.*

- funktionale schwache Kopfnormalform (*FWHNF = functional whnf*), *wenn s eine Abstraktion ist.*
- Konstruktor-schwache Kopfnormalform (*CWHNF = constructor whnf*), *wenn s eine Konstruktoranwendung $(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})$ ist.*

Beachte, dass jede Normalform auch eine Kopfnormalform ist, die Umkehrung gilt jedoch nicht: eine HNF kann Redexe in Argumenten enthalten, während diese für NFs nicht erlaubt ist. Ebenso gilt: Eine HNF ist auch stets eine WHNF aber nicht umgekehrt. Wir verwenden nur WHNFs (keine NFs, keine HNFs).

Definition 2.2.12 (Konvergenz, Terminierung)**.** *Ein* KFPT*-Ausdruck s* konvergiert (*oder* terminiert, *notiert als $s{\Downarrow}$*) *genau dann, wenn er mit Normalordnungsreduktionen in eine WHNF überführt werden kann, d.h.*

$$s{\Downarrow} \iff \exists\, \mathit{WHNF}\ t : s \xrightarrow{no,*} t$$

Falls ein Ausdruck s nicht konvergiert, so sagen wir s divergiert *und notieren dies mit $s{\Uparrow}$.*

Als Sprechweise benutzen wir auch: Wir sagen *s hat eine WHNF* (bzw. FWHNF, CWHNF), wenn s zu einer WHNF (bzw. FWHNF, CWHNF) mit endlichen vielen (oder auch 0) Normalordungsreduktionen reduziert werden kann.

2.2.4 Dynamische Typisierung

Die Normalordnungsreduktion kann unter Umständen stoppen (d.h. es ist keine $\xrightarrow{no}$-Reduktion mehr anwendbar), *ohne* dass eine WHNF erreicht wurde. In diesem Fall wurde entweder eine freie Variable an Reduktionsposition gefunden, d.h. der Ausdruck ist von der Form $R[x]$, wobei R ein Reduktionskontext ist, oder es wurde ein Typfehler beim Auswerten gefunden. Da der Typfehler erst beim Auswerten entdeckt wird (Erinnerung: KFPT ist nur schwach getypt), spricht man von einem *dynamischen Typfehler*. Auf dieser Beobachtung aufbauend, kann man dynamische Typregeln für KFPT definieren:

Definition 2.2.13 (Dynamische Typregeln für KFPT)**.** *Sei s ein* KFPT*-Ausdruck. Wir sagen s ist* direkt dynamisch ungetypt, *falls s von einer der folgenden Formen ist (hierbei ist R ein beliebiger Reduktionskontext):*

- $R[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ \mathit{Alts}]$ *und c ist* nicht *vom Typ T*

- $R[\texttt{case}_T\ \lambda x.t\ \texttt{of}\ Alts]$.
- $R[(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ t]$

Ein KFPT-*Ausdruck* s *ist* dynamisch ungetypt, *falls er sich mittels Normalordnungsreduktionen in einen direkt dynamisch ungetypten Ausdruck überführen lässt, d.h.*

$$s \text{ ist dynamisch ungetypt}$$
$$\Longleftrightarrow$$
$$\exists t : s \xrightarrow{no,*} t \wedge t \text{ ist direkt dynamisch ungetypt}$$

Beachte, dass dynamisch ungetypte Ausdrücke stets divergieren. Es gilt zudem der folgende Satz:

Satz 2.2.14. *Ein* geschlossener KFPT-*Ausdruck* s *ist irreduzibel* (*bzgl. der Normalordnung*) *genau dann, wenn eine der folgenden Bedingungen auf ihn zutrifft:*

- *Entweder ist* s *eine WHNF, oder*
- s *ist direkt dynamisch ungetypt.*

Beachte, dass obiger Satz nur über geschlossene Ausdrücke spricht. Ebenfalls sei erwähnt, dass nicht jeder geschlossene divergente Ausdruck auch dynamisch ungetypt ist: Betrachte z.B. $\Omega := (\lambda x.x\ x)\ (\lambda x.x\ x)$, dieser Ausdruck divergiert, da gilt $\Omega \xrightarrow{no} \Omega \xrightarrow{no} \Omega \ldots$, aber Ω ist nicht dynamisch ungetypt.

Des Weiteren sei anzumerken, dass Haskell dynamisch ungetypte Ausdrücke zur Compilezeit als ungetypt erkennt. Allerdings ist Haskells Typsystem restriktiver, d.h. es gibt KFPT-Ausdrücke, die nicht dynamisch ungetypt sind, jedoch nicht Haskell-typisierbar sind. Ein Beispiel ist gerade der Ausdruck Ω[9]. Ein anderes Beispiel ist der Ausdruck $\texttt{case}_{\texttt{Bool}}\ \texttt{True}\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{Nil}\}$.

2.2.5 Suche nach dem Normalordnungsredex mit einem Markierungsalgorithmus

Wie bereits für den Lambda-Kalkül geben wir für KFPT eine alternative Möglichkeit an, den Normalordnungsredex mithilfe eines Markierungsalgorithmus zu finden. In Compilern bzw. Interpretern für Funktionale Programmiersprachen, werden Ausdrücke als Termgraphen dargestellt. Dort wird ein äquivalentes Verfahren durchgeführt, welches man gemeinhin als (Graph)-Unwinding bezeichnet.

[9] In einem späteren Kapitel werden wir erfahren, warum dies so ist.

Für einen KFPT-Ausdruck s startet unser Markierungs-Algorithmus mit $s^\star$. Anschließend werden die folgenden beiden Regeln solange wie möglich angewendet:

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_{Typ}\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_{Typ}\ s^\star\ \texttt{of}\ Alts)$

Nach Ausführen des Markierungsalgorithmus können folgende Fälle auftreten:

- Die Markierung ist an einer Abstraktion. Dann gibt es drei Fälle:
  - s selbst ist die markierte Abstraktion. Dann wurde eine WHNF erreicht, es ist keine Normalordnungsreduktion möglich.
  - Der direkte Oberterm der Abstraktion ist eine Anwendung, d.h. s ist von der Form $C[(\lambda x.s')^\star\ t]$. Dann reduziere $C[(\lambda x.s')\ t] \xrightarrow{no,\beta} C[s'[t/x]]$.
  - Der direkte Oberterm der Abstraktion ist ein `case`-Ausdruck, d.h. s ist von der Form $C[\texttt{case}_{Typ}\ (\lambda x.s')^\star\ \texttt{of}\ Alts]$. Dann ist keine Normalordnungsreduktion möglich. s ist direkt dynamisch ungetypt.
- Die Markierung ist an einer Konstruktoranwendung. Dann gibt es die Fälle:
  - s selbst ist die markierte Konstruktoranwendung. Dann wurde eine WHNF erreicht, es ist keine Normalordnungsreduktion möglich.
  - Der direkte Oberterm der Konstruktoranwendung ist ein `case`-Ausdruck, d.h. s ist von der Form $C[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ Alts]$. Es gibt zwei Fälle:
    - c ist vom Typ T, d.h. es gibt eine passende Alternative für c, dann reduziere:
      $C[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ \{\ldots;(c\ x_1\ \ldots\ x_n) \to t\ ;\ldots\}]$
      $\xrightarrow{no,case} C[t[s_1/x_1,\ldots,s_n/x_n]]$.
    - c ist nicht vom Typ T. Dann ist keine Normalordnungsreduktion möglich für s. s ist direkt dynamisch ungetypt.
  - Der direkte Oberterm der Konstruktoranwendung ist eine Anwendung, d.h. s ist von der Form $C[(c\ s_1\ \ldots\ s_n)^\star\ t]$. In diesem Fall ist keine Normalordnungsreduktion möglich für s. Der Ausdruck s ist direkt dynamisch ungetypt.

- Die Markierung ist an einer Variablen, d.h. s ist von der Form $C[x^\star]$. Dann ist keine Normalordnungsreduktion möglich, da eine freie Variable entdeckt wurde. Der Ausdruck s ist keine WHNF, aber auch nicht direkt dynamisch ungetypt.

Beispiel 2.2.15. *Die Auswertung in Normalordnung für den Ausdruck*

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

wobei sämtliche Suchschritte zum Finden des Normalordnungsredex mit angegeben sind, ist:

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))^\star$$

$$\Rightarrow(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))\ (\lambda u, v.v))^\star\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\Rightarrow(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^\star\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{no,\beta}((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))^\star$$

$$\Rightarrow((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^\star\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{no,\beta}(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})^\star$$

$$\Rightarrow(\begin{pmatrix}\texttt{case}_{\texttt{List}}\,(\texttt{Cons}\,(\lambda w.w)\,\texttt{Nil})\,\texttt{of}\,\{\\ \texttt{Nil}\to\texttt{Nil};\\ (\texttt{Cons}\,z\,zs)\to((\lambda u,v.v)\,z)\}\end{pmatrix}^{\star}\,\texttt{True})$$

$$\Rightarrow(\begin{pmatrix}\texttt{case}_{\texttt{List}}\,(\texttt{Cons}\,(\lambda w.w)\,\texttt{Nil})^{\star}\,\texttt{of}\,\{\\ \texttt{Nil}\to\texttt{Nil};\\ (\texttt{Cons}\,z\,zs)\to((\lambda u,v.v)\,z)\}\end{pmatrix}\,\texttt{True})$$

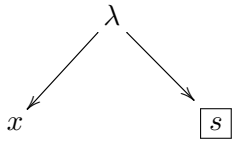
$$\begin{aligned}
&\xrightarrow{no,case}(((\lambda u,v.v)\,(\lambda w.w))\,\texttt{True})^{\star}\\
&\quad\Rightarrow(((\lambda u,v.v)\,(\lambda w.w))^{\star}\,\texttt{True})\\
&\quad\Rightarrow(((\lambda u,v.v)^{\star}\,(\lambda w.w))\,\texttt{True})\\
&\xrightarrow{no,\beta}((\lambda v.v)\,\texttt{True})^{\star}\\
&\quad\Rightarrow((\lambda v.v)^{\star}\,\texttt{True})\\
&\xrightarrow{no,\beta}\texttt{True}
\end{aligned}$$

2.2.6 Darstellung von Termen als Termgraphen

Wir erläutern kurz die Darstellung von KFPT-Ausdrücken als Baum. Jeder Knoten eines solchen Baumes stellt ein syntaktisches Konstrukt des Ausdrucks dar (von der Wurzel beginnend).

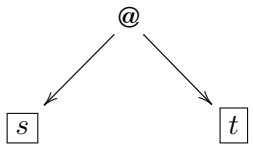
- Variablen werden durch einen Knoten, der mit der Variablen markiert ist, dargestellt.

- Abstraktionen werden durch einen mit „λ“ markierten Knoten dargestellt, der zwei Kinder besitzt: Das linke Kind ist die durch die Abstraktion gebundene Variable, das zweite Kind entspricht dem Rumpf der Abstraktion. D.h $\lambda x.s$ wird dargestellt durch



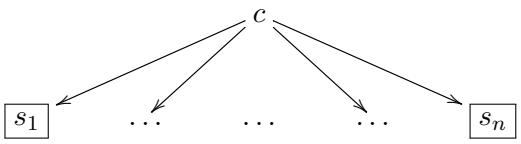
  wobei $\boxed{s}$ der Baum für s ist.

- Applikationen werden durch einen mit „@“ markierten Knoten dargestellt, der zwei Kinder für den Ausdruck an Funktionsposition und dem Argument besitzt. D.h. $(s\ t)$ wird dargestellt durch



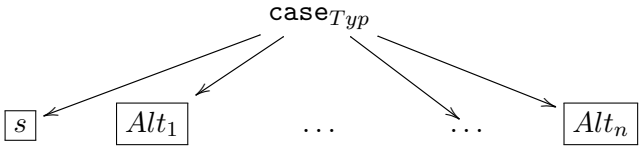
  wobei $\boxed{s}$ und $\boxed{t}$ die Bäume für s und t sind.

- Konstruktoranwendungen haben als Knotenbeschriftung den Konstruktornamen, und n Kinder, wenn der Konstruktor Stelligkeit n besitzt. D.h. $(c\ s_1 \ldots\ s_n)$ wird dargestellt durch



  wobei $\boxed{s_i}$ die Bäume für s_i sind.

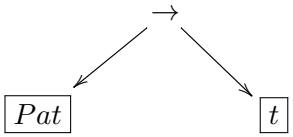
- `case`-Ausdrücke haben $n + 1$ Kinder, wenn n die Anzahl der Alternativen ist: Das erste Kind ist das erste Argument des `case`-Ausdrucks, die anderen Kinder entsprechen den Alternativen. D.h. $\texttt{case}_{Typ}\ s\ \texttt{of}\ \{Alt_1; \ldots; Alt_n\}$ wird dargestellt durch



  wobei $\boxed{s}$ der Baum für s und $\boxed{Alt_i}$ der Baum für die i-te Alternative ist.

- eine `case`-Alternative „Pattern“ $\rightarrow$ „Ausdruck“ wird durch einen mit $\rightarrow$ markierten Knoten dargestellt, der zwei Kinder besitzt: einen für das Pattern und einen für die rechte Seite der Alternativen. D.h. $Pat \rightarrow t$

wird durch

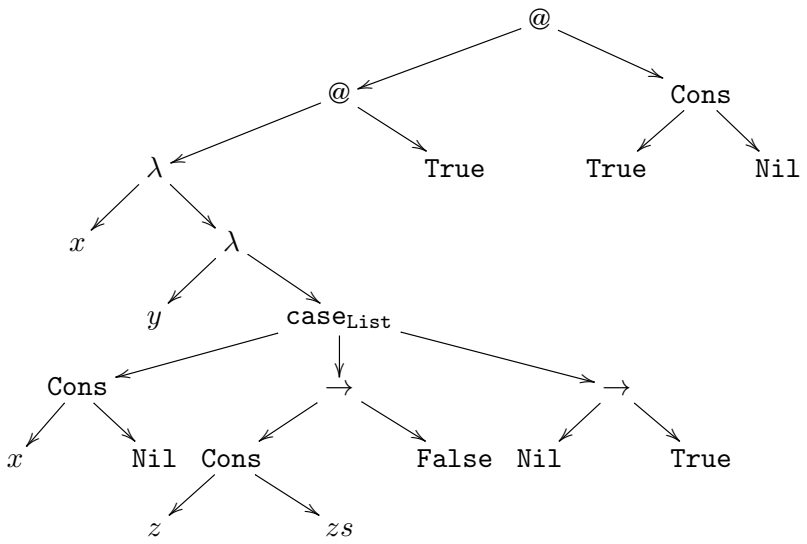


dargestellt, wobei $\boxed{Pat}$ der Baum für das Pattern und $\boxed{t}$ der Baum für t ist.

Beispiel 2.2.16. *Der Baum für den Ausdruck*

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \{\\ \qquad (\texttt{Cons}\ z\ zs) \rightarrow \texttt{False};\\ \qquad \texttt{Nil} \rightarrow \texttt{True}\}\end{array}\right)\ \texttt{True}\right)\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})$$

ist:



Die Suche nach dem Normalordnungsredex kann man sich in der Baumdarstellung wie folgt verdeutlichen: Man läuft den linkesten Pfad solang herab, bis man auf einen Knoten trifft der

- entweder mit einer Variablen markiert ist, oder
- mit einem Konstruktor markiert ist, oder
- mit λ markiert ist.

Der direkte Oberterm (bzw. Oberbaum) ist dann der Normalordnungsredex, falls eine Normalordnungsreduktion möglich ist.

Für oben gezeigten Beispielbaum steigt die Suche entlang des linkesten Pfades bis zum ersten λ herab, der Normalordnungsredex ist deren Oberterm, d.h. die Anwendung $(\lambda x. \ldots)$ `True`. Der Vollständigkeit halber zeigen wir die gesamte Auswertung des Beispielausdrucks:

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \{\\ \qquad (\texttt{Cons}\ z\ zs) \to \texttt{False};\\ \qquad \texttt{Nil} \to \texttt{True}\}\end{array}\right)\ \texttt{True}\right)\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})$$

$$\xrightarrow{no,\beta} \quad \left(\begin{array}{l}\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})\ \texttt{of}\ \{\\ \qquad (\texttt{Cons}\ z\ zs) \to \texttt{False};\\ \qquad \texttt{Nil} \to \texttt{True}\}\end{array}\right)\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})$$

$$\begin{array}{ll}\xrightarrow{no,\beta} & \texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})\ \texttt{of}\ \{(\texttt{Cons}\ z\ zs) \to \texttt{False}; \texttt{Nil} \to \texttt{True}\}\\ \xrightarrow{no,case} & \texttt{False}\end{array}$$

2.2.7 Eigenschaften der Normalordnungsreduktion

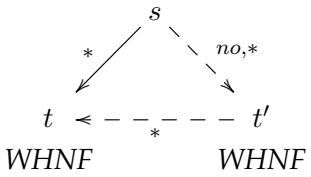
Für die Normalordnungsreduktion gilt:

- Die Normalordnungsreduktion ist deterministisch, d.h. für jeden KFPT-Ausdruck s gibt es höchstens einen Ausdruck t mit $s \xrightarrow{no} t$.
- Eine WHNF ist irreduzibel bezüglich der Normalordnungsreduktion.

Das folgende Theorem zeigt, dass die Normalordnungsreduktion standardisierend ist, d.h. falls eine WHNF mit (β)- und (`case`)-Reduktionen gefunden werden kann, dann findet auch die Normalordnungsreduktion eine WHNF. Wir beweisen das Theorem im Rahmen dieser Vorlesung nicht.

Theorem 2.2.17 (Standardisierung)**.** *Sei s ein* KFPT*-Ausdruck. Wenn $s \xrightarrow{*} t$ mit beliebigen (β)- und (case)-Reduktionen (in beliebigem Kontext angewendet), wobei t eine WHNF ist, dann existiert eine WHNF t', so dass $s \xrightarrow{no,*} t'$ und $t' \xrightarrow{*} t$ (wobei das modulo alpha-Gleichheit gemeint ist). D.h. das folgende Diagramm gilt, wobei*

die gestrichelten Pfeile Existenz-quantifizierte Reduktionen sind:



Aus dem Theorem folgt auch, dass man an beliebigen Positionen eine (β)- oder (case)-Reduktion anwenden kann, ohne die Konvergenzeigenschaft zu verändern.

2.3 Die Kernsprache KFPTS

Als nächste Erweiterung von KFPT betrachten wir die Sprache KFPTS. Das „S“ steht hierbei für Superkombinatoren. Superkombinatoren sind Namen (bzw. Konstanten), die (rekursive) Funktionen bezeichnen. Mit KFPTS beheben wir den zweiten Kritikpunkt aus Bemerkung 2.1.26.

2.3.1 Syntax

Wir nehmen an, es gibt eine Menge von Superkombinatornamen $\mathcal{SK}$.

Definition 2.3.1. *Ausdrücke der Sprache* KFPTS *werden durch die folgende Grammatik gebildet, wobei die Bedingungen an* `case`*-Ausdrücke genau wie in* KFPT *gelten müssen:*

$$
\begin{array}{lcl}
\mathbf{Expr} & ::= & V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}_1\ \mathbf{Expr}_2) \\
& \mid & (c_i\ \mathbf{Expr}_1\ \ldots\ \mathbf{Expr}_{\mathrm{ar}(c_i)}) \\
& \mid & (\mathtt{case}_{\mathtt{Typname}}\ \mathbf{Expr}\ \mathtt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n\}) \\
& \mid & SK \text{ wobei } SK \in \mathcal{SK}
\end{array}
$$

$$
\mathbf{Pat}_i \quad ::= \quad (c_i\ V_1 \ldots V_{ar(c_i)}) \text{ wobei die Variablen } V_i \text{ alle verschieden sind.}
$$

Zu jedem verwendeten Superkombinator muss es genau eine Superkombinatordefinition geben:

Definition 2.3.2. *Eine* Superkombinatordefinition *ist eine Gleichung der Form:*

$$SK\ V_1\ \ldots\ V_n = \mathbf{Expr}$$

wobei V_i paarweise verschiedene Variablen sind und **Expr** *ein* KFPTS*-Ausdruck ist. Außerdem muss gelten: $FV(\mathbf{Expr}) \subseteq \{V_1, \ldots, V_n\}$, d.h. nur die Variablen $V_1, \ldots, V_n$ kommen frei in* **Expr** *vor. Mit $\mathrm{ar}(SK) = n$ bezeichnen wir die Stelligkeit des Superkombinators.*

Wir nehmen stets an, dass die Namensräume der Superkombinatoren, der Variablen und der Konstruktoren paarweise disjunkt sind.

Definition 2.3.3. *Ein* KFPTS-Programm *besteht aus:*

- *Einer Menge von Typen und Konstruktoren,*
- *einer Menge von Superkombinator-Definitionen.*
- *und aus einem* KFPTS*-Ausdruck s. (Diesen könnte man auch als Superkombinator* `main` *mit Definition* `main` $= s$ *definieren).*

Dabei muss gelten, dass alle Superkombinatoren, die in rechten Seiten der Definitionen und auch in s auftreten, auch definiert sind.

2.3.2 Auswertung von KFPTS-Ausdrücken

Die KFPT-Normalordnung muss für KFPTS erweitert werden, um Superkombinatoranwendungen auszuwerten. Die Reduktionskontexte für KFPTS sind analog zu KFPT definiert als:

$$\mathbf{R} ::= [\cdot] \mid (\mathbf{R}\ \mathbf{Expr}) \mid \texttt{case}_{Typ}\ \mathbf{R}\ \texttt{of}\ Alts$$

Die Reduktionsregeln werden erweitert um eine neue Regel.

Definition 2.3.4. *Die Reduktionsregeln (β), (case) und (SK-β) sind in* KFPTS *definiert als:*

$$\begin{array}{ll}
(\beta) & (\lambda x.s)\ t \to s[t/x] \\
(case) & \texttt{case}_{Typ}\ (c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ \texttt{of}\ \{\ldots;\ (c\ x_1 \ldots\ x_{\mathrm{ar}(c)}) \to t;\ \ldots\} \\
 & \quad \to t[s_1/x_1, \ldots, s_{\mathrm{ar}(c)}/x_{\mathrm{ar}(c)}] \\
(SK\text{-}\beta) & (SK\ s_1\ \ldots\ s_n) \to e[s_1/x_1, \ldots, s_n/x_n], \\
 & \quad \textit{wenn } SK\ x_1\ \ldots\ x_n = e \textit{ die Definition von } SK \textit{ ist}
\end{array}$$

Die Normalordnungsreduktion in KFPTS wendet nun eine der drei Regeln in einem Reduktionskontext an:

Definition 2.3.5. *Wenn $s \to t$ mit einer (β)-, (case)- oder (SK-β)-Reduktion, dann ist $R[s] \to R[t]$ eine* (KFPTS)*-Normalordnungsreduktion. Wir notieren dies mit $R[s] \xrightarrow{no} R[t]$.*

Auch die Definition von WHNFs muss leicht angepasst werden: Eine CWHNF ist weiterhin eine Konstruktoranwendung, eine FWHNF ist eine Abstraktion oder ein Ausdruck der Form $SK\ s_1\ \ldots\ s_m$, wenn $\mathrm{ar}(SK) > m$, d.h. der Superkombinator ist nicht gesättigt. Eine WHNF ist eine FWHNF oder eine CWHNF.

Für die dynamischen Typregeln fügen wir hinzu:

- $R[\mathtt{case}_T\ SK\ s_1\ \ldots\ s_m\ \mathtt{of}\ Alts]$ ist direkt dynamisch ungetypt falls $\mathrm{ar}(SK) > m$.

Der Markierungsalgorithmus zur Redexsuche funktioniert wie in KFPT, mit dem Unterschied, dass neue Fälle auftreten können:

- Der mit $\star$ markierte Unterausdruck ist ein Superkombinator, d.h. der markierte Ausdruck ist von der Form $C[SK^\star]$. Es gibt nun drei Unterfälle:
  - $C = C'[[\cdot]\ s_1\ \ldots\ s_n]$ und $\mathrm{ar}(SK) = n$. Dann wende die (SK-β)-Reduktion an: $C'[SK\ s_1\ \ldots\ s_n] \xrightarrow{no,SK-\beta} C'[e[s_1/x_1, \ldots, s_n/x_n]]$ (wenn $SK\ x_1\ \ldots\ x_n = e$ die Definition von SK ist).
  - $C = [[\cdot]\ s_1\ \ldots\ s_m]$ und $\mathrm{ar}(SK) > m$. Dann ist der Ausdruck eine WHNF.
  - $C = C'[\mathtt{case}_T\ [\cdot]\ s_1\ \ldots\ s_m\ \mathtt{of}\ Alts]$ und $\mathrm{ar}(SK) > m$. Dann ist der Ausdruck direkt dynamisch ungetypt.

Beispiel 2.3.6. *Die Superkombinatoren map und not seien definiert als:*

$$map\ f\ xs = \mathtt{case}_{\mathtt{List}}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \mathtt{Nil}; (\mathtt{Cons}\ y\ ys) \to \mathtt{Cons}\ (f\ y)\ (map\ f\ ys)\}$$
$$not\ x \quad = \mathtt{case}_{\mathtt{Bool}}\ x\ \mathtt{of}\ \{\mathtt{True} \to \mathtt{False}; \mathtt{False} \to \mathtt{True}\}$$

Die Auswertung des Ausdrucks map not (Cons True (Cons False Nil)) *in* KFPTS*-Normalordnung ist:*

$$\begin{aligned}
& map\ not\ (\mathtt{Cons}\ \mathtt{True}\ (\mathtt{Cons}\ \mathtt{False}\ \mathtt{Nil})) \\
\xrightarrow{no,SK\text{-}\beta}\ & \mathtt{case}_{\mathtt{List}}\ (\mathtt{Cons}\ \mathtt{True}\ (\mathtt{Cons}\ \mathtt{False}\ \mathtt{Nil}))\ \mathtt{of}\ \{ \\
& \quad \mathtt{Nil} \to \mathtt{Nil}; \\
& \quad (\mathtt{Cons}\ y\ ys) \to \mathtt{Cons}\ (not\ y)\ (map\ not\ ys)\} \\
\xrightarrow{no,case}\ & \mathtt{Cons}\ (not\ \mathtt{True})\ (map\ not\ (\mathtt{Cons}\ \mathtt{False}\ \mathtt{Nil}))
\end{aligned}$$

Beachte, dass der erreichte Ausdruck eine WHNF ist. Will man die Liste komplett auswerten, müsste man dies durch eine Funktion erzwingen. Auch Haskell wertet so aus, im `ghci` *bemerkt man diesen Effekt oft nicht, da dort zum* Anzeigen *des Ergebnisses die Daten voll ausgewertet werden.*

2.4 Erweiterung um `seq`

Haskell stellt den Operator `seq` zur Verfügung. Dessen Semantik kann man definieren als

$$(\texttt{seq}\ a\ b) = \begin{cases} b & \text{falls } a{\Downarrow} \\ \bot & \text{falls } a{\Uparrow} \end{cases}$$

Operational kann man dies so darstellen: Die Auswertung eines Ausdrucks $(\texttt{seq}\ a\ b)$ wertet zunächst a zur WHNF aus, danach erhält man $\texttt{seq}\ v\ b$ wobei v eine WHNF ist. Nun darf man $(\texttt{seq}\ v\ b) \to b$ reduzieren. Ein analoger Operator ist der Operator `strict` (in Haskell infix geschrieben als `!$`). Angewendet auf eine Funktion, macht `strict` die Funktion strikt in ihrem ersten Argument[10], d.h. `strict` erzwingt die Auswertung des Arguments bevor die Definitionseinsetzung bzw. β-Reduktion durchgeführt werden kann. `seq` und `strict` sind äquivalent, da jeder der beiden Operatoren mit dem jeweilig anderen kodiert werden kann[11]:

$$\begin{array}{lcl} \texttt{strict}\ f\ x & = & \texttt{seq}\ x\ (f\ x) \\ \texttt{seq}\ a\ b & = & \texttt{strict}\ (\lambda x.b)\ a \end{array}$$

Die Operatoren `seq` bzw. `strict` sind nützlich, um die strikte Auswertung zu erzwingen. Dies ist z.B. vorteilhaft um Platz während der Auswertung zu sparen. Wir betrachten als Beispiel die Fakultätsfunktion: Die naive Imple-

[10] Formal definiert man Striktheit einer Funktion wie folgt: Eine n-stellige Funktion f ist strikt im i-ten Argument, falls gilt $f\ s_1\ \ldots\ s_{i-1}\ \bot\ s_{i+1}\ \ldots\ s_n = \bot$ für alle beliebigen geschlossene Ausdrücke s_i. Dabei ist $\bot$ ein geschlossener nichtterminierender Ausdruck und $=$ die Gleichheit der entsprechenden Sprache (z.B. die kontextuelle Gleichheit). Es gilt: Wertet f ihr i-tes Argument stets aus, so ist obige Defintion erfüllt, d.h. f ist strikt im i-ten Argument. Allerdings kann f auch strikt im i-ten Argument sein, ohne das Argument auszuwerten. Betrachte z.B. die Funktion:

$$f\ x\ = f\ x$$

f ist strikt im ersten Argument, da f niemals terminiert. Methoden und Verfahren um Striktheit im Rahmen von funktionalen Programmiersprachen zu erkennen, werden u.a. in der Veranstaltung „M-SAFP: Semantik und Analyse von funktionalen Programmen“ erörtert.

[11] In Haskell-Notation:

```
f $! x   =   seq x (f x)
seq a b  =   (\x -> b) $! a
```

mentierung ist:

```
fak 0 = 1
fak x = x*(fak (x-1))
```

Wertet man `fak` n aus so wird zunächst der Ausdruck $n * (n-1) * \ldots * 1$ erzeugt, der anschließend ausgerechnet wird. Das Aufbauen des Ausdrucks benötigt linear viel Platz. Die endrekursive Definition von `fak` ist:

```
fak x = fakER x 1
 where fakER 0 y = y
       fakER x y = fakER (x-1) (x*y)
```

Diese behebt das Platzproblem noch *nicht*, allerdings kann man mit `let`- und `seq` konstanten Platzbedarf erzwingen, indem vor dem rekursiven Aufruf das Auswerten der Argumente erzwungen wird:

```
fak x = fakER x 1
 where fakER 0 y = y
       fakER x y = let x' = x-1
                       y' = x*y
                   in seq x' (seq y' (fakER x' y'))
```

Der Operator `seq` wurde aufgrund dieser Nützlichkeit zum Haskell-Standard hinzugefügt. Unsere Kernsprachen KFPT und KFPTS unterstützen diesen Operator nicht, er kann dort auch nicht simuliert werden. Deshalb bezeichnen wir mit KFPT+seq bzw. KFPTS+seq die Sprachen KFPT bzw. KFPTS zu denen der Operator hinzugefügt wurde. Wir verzichten an dieser Stelle weitgehend auf die formale Angabe dieser Erweiterung (Erweiterung der Syntax, der Reduktionskontexte und der operationalen Semantik), es ist jedoch erwähnswert, dass man als zusätzliche Reduktionsregel benötigt:

`seq` $v\ t \to t$, wenn v eine Abstraktion oder eine Konstruktoranwendung ist

Bemerkung 2.4.1. *Die Kernsprache* KFP *ist definiert durch die Syntax*

$$\begin{aligned}\mathbf{Expr} ::= {} & V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}_1\ \mathbf{Expr}_2) \\ \mid {} & (c_i\ \mathbf{Expr}_1\ \ldots\ \mathbf{Expr}_{\mathrm{ar}(c_i)}) \\ \mid {} & (\texttt{case}\ \mathbf{Expr}\ \texttt{of} \\ & \quad \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n; \texttt{lambda} \to \mathbf{Expr}_{n+1}\}) \\ & \textit{wobei } Pat_1, \ldots, Pat_n \textit{ alle Konstruktoren abdecken}\end{aligned}$$

$\mathbf{Pat}_i$ ::= $(c_i\ V_1 \ldots V_{ar(c_i)})$ *wobei die Variablen* V_i *alle verschieden sind.*

Der Unterschied zu KFPT *liegt in den* `case`*-Ausdrücken:*

- *Statt getypten* `case`*-Ausdrücken gibt es nur ein* `case`*-Konstrukt.*
- *Die Alternativen eines* `case`*-Ausdrucks decken alle Konstruktoren ab.*
- *Es gibt eine zusätzliche* `case`*-Alternative* `lambda` $\to$ **Expr***, die den Fall abdeckt, dass eine Abstraktion im ersten Argument des* `case`*-Ausdrucks steht, d.h. die neue Reduktionsregel passend dazu ist:*

$$\texttt{case}\ \lambda x.s\ \texttt{of}\ \{\ldots, \texttt{lambda} \to t\} \quad \to \quad t$$

In KFP *ist* `seq` (*und auch* `strict`) *definierbar:*

$$\texttt{seq}\ a\ b := \texttt{case}\ a\ \texttt{of}\ \{Pat_1 \to b; \ldots; Pat_n \to b; \texttt{lambda} \to b\}$$

Wir verwenden KFP *jedoch in dieser Veranstaltung nicht.*

2.5 KFPTSP: Polymorphe Typen

Die bisher vorgestellten Kernsprachen waren ungetypt bzw. sehr schwach getypt. Haskell verwendet jedoch polymorphe Typisierung. Mit KFPTSP (bzw. KFPTSP+seq) bezeichnen wir die Kernsprache KFPTS (bzw. KFPTS+seq), die nur korrekt getypte Ausdrücke und Programme zulässt. Wir werden an dieser Stelle nur ganz allgemeine Grundlagen zur polymorphen Typisierung und Haskells Typsystem einführen. Genauer werden wir Haskells Typisierung in einem späteren Kapitel untersuchen.

Definition 2.5.1. *Die Syntax von polymorphen Typen kann durch die folgende Grammatik beschrieben werden:*

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1 \to \mathbf{T}_2$$

wobei TV für eine Typvariable steht und TC ein Typkonstruktor mit Stelligkeit n ist.

Typkonstruktoren sind Namen wie `Bool`, `List`, `Paar`, etc. Typkonstruktoren können Argumente verlangen, z.B. erwartet der Typkonstruktor `List` ein Argument, dass den Typ der Listenelemente angibt, z.B. polymorph `List` a (in Haskell-Schreibweise `[a]`, wir verwenden diese Schreibweise auch für KFPTSP). Die Typen der Datenkonstruktoren sind entsprechend (vom Programmierer) vorgegeben. Z.B. hat `Cons` den polymorphen Typ $a \to$ `List` $a \to$ `List` a, d.h. `Cons` erwartet als Argument einen Ausdruck vom Elementtyp und eine Liste (in dieser Reihenfolge) und liefert dann die zusammengesetzte Liste. Das Konstrukt $T_1 \to T_2$ könnte man auch als speziellen Typkonstruktor ansehen, durch $\to$ wird ein *Funktionstyp* dargestellt, die Bedeutung ist: Ein Ausdruck vom Typ $T_1 \to T_2$ erwartet ein Argument vom Typ T_1 und liefert bei Anwendung auf ein solches Argument einen Ausdruck vom Typ T_2. Z.B. hat die Fakultätsfunktion den Typ `Int` $\to$ `Int`: sie erwartet eine Zahl und liefert als Ergebnis eine Zahl.

Beachte, dass der $\to$-Konstruktor recht-assoziativ ist, d.h. $T_1 \to T_2 \to T_3$ entspricht $T_1 \to (T_2 \to T_2)$ und *nicht* $(T_1 \to T_2) \to T_3$. Die so definierten Typen heißen *polymorphe* Typen, da sie Typvariablen enthalten. Sie stellen daher eine Menge von (monomorphen) Typen dar. Diese erhält man, indem man für die Typvariablen Typen einsetzt, d.h. eine Typsubstitution auf den polymorphen Typ anwendet. Z.B. kann man den Typ $a \to a$ mit der Substitution $\sigma = \{a \mapsto$ `Bool`$\}$ zu $\sigma(a \to a) =$ `Bool` $\to$ `Bool` instanziieren.

Einige Beispiel-Ausdrücke mit Typen sind:

`True`	::	`Bool`
`False`	::	`Bool`
`not`	::	`Bool` $\to$ `Bool`
`map`	::	$(a \to b) \to [a] \to [b]$
$(\lambda x.x)$	::	$(a \to a)$

Wir werden später im Detail sehen, wie man überprüft, ob ein Ausdruck typisierbar bzw. korrekt getypt ist. An dieser Stelle geben wir uns mit den folgenden einfachen Typisierungsregeln zufrieden. Die Notation der Regeln ist

$$\frac{\text{Voraussetzung}}{\text{Konsequenz}},$$

d.h. um die Konsequenz herleiten zu dürfen, muss die Voraussetzung erfüllt

sein. Die vereinfachten Regeln sind:

- Für die Anwendung:
$$\frac{s :: T_1 \to T_2, t :: T_1}{(s\ t) :: T_2}$$

- Instanziierung
$$\frac{s :: T}{s :: T'} \quad \text{wenn } T' = \sigma(T), \text{ wobei } \sigma \text{ eine Typsubstitution ist, die Typen für Typvariablen ersetzt.}$$

- Für `case`-Ausdrücke:
$$\frac{s :: T_1, \quad \forall i : Pat_i :: T_1, \quad \forall i : t_i :: T_2}{(\texttt{case}_T\ s\ \texttt{of}\ \{Pat_1 \to t_1; \ldots; Pat_n \to t_n\}) :: T_2}$$

Beispiel 2.5.2. *Die booleschen Operatoren* `and` *und* `or` *kann man in* KFPTS *als Abstraktionen definieren:*

$$\begin{aligned} \texttt{and} &:= \lambda x, y.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to y; \texttt{False} \to \texttt{False}\} \\ \texttt{or} &:= \lambda x.y.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to y\} \end{aligned}$$

In Haskell sind diese bereits als `(&&)` *und* `(||)` *vordefiniert. Beide Operatoren haben den Typ* `Bool` $\to$ `Bool` $\to$ `Bool`*. Den Ausdruck* `and True False` *kann man nun typisieren durch zweimaliges Anwenden der Anwendungsregel:*

$$\frac{\dfrac{\texttt{and} :: \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}, \texttt{True} :: \texttt{Bool}}{(\texttt{and}\ \texttt{True}) :: \texttt{Bool} \to \texttt{Bool}}, \texttt{False} :: \texttt{Bool}}{(\texttt{and}\ \texttt{True}\ \texttt{False}) :: \texttt{Bool}}$$

Beispiel 2.5.3. *Der Ausdruck*

$$\texttt{case}_{\texttt{Bool}}\ \texttt{True}\ \texttt{of}\ \{\texttt{True} \to (\texttt{Cons}\ \texttt{True}\ \texttt{Nil}); \texttt{False} \to \texttt{Nil}\}$$

lässt sich mit obigen Regeln folgendermaßen typisieren:

$$\dfrac{\dfrac{\dfrac{\texttt{Cons} :: a \to [a] \to [a]}{\texttt{Cons} :: \texttt{Bool} \to [\texttt{Bool}] \to [\texttt{Bool}]}, \texttt{True} :: \texttt{Bool}}{(\texttt{Cons}\ \texttt{True}) :: [\texttt{Bool}] \to [\texttt{Bool}]}, \dfrac{\texttt{Nil} :: [a]}{\texttt{Nil} :: [\texttt{Bool}]}}{(\texttt{Cons}\ \texttt{True}\ \texttt{Nil}) :: [\texttt{Bool}]}, \dfrac{\texttt{Nil} :: [a]}{\texttt{Nil} :: [\texttt{Bool}]}}{\texttt{case}_{\texttt{Bool}}\ \texttt{True}\ \texttt{of}\ \{\texttt{True} \to (\texttt{Cons}\ \texttt{True}\ \texttt{Nil}); \texttt{False} \to \texttt{Nil}\} :: [\texttt{Bool}]}$$

Beispiel 2.5.4. *Der Operator* `seq` *hat den Typ* $a \to b \to b$*, da er sein erstes Argument im Ergebnis ignoriert.*

Beispiel 2.5.5. *Hat man die Typen von* `map` *und* `not` *bereits gegeben, so kann man den Typ der Anwendung* (`map not`) *mit obigen Regeln berechnen als:*

$$\dfrac{\dfrac{\texttt{map} :: (a \to b) \to [a] \to [b]}{\texttt{map} :: (\texttt{Bool} \to \texttt{Bool}) \to [\texttt{Bool}] \to [\texttt{Bool}]}, \texttt{not} :: \texttt{Bool} \to \texttt{Bool}}{(\texttt{map}\ \texttt{not}) :: [\texttt{Bool}] \to [\texttt{Bool}]}$$

Beachte, dass die Verwendung der Instanziierung-Regel nicht algorithmisch ist, da man die passende Substitution quasi „raten“ muss. Später werden wir algorithmische Verfahren hierfür erläutern.

In den obigen einfachen Typregeln gibt es noch keine Regel für die Abstraktion und die `case`-Regel ist auch noch zu ungenau, da die Abhängigkeit der Variablen in den Pattern und den rechten Seiten der Alternativen noch nicht erfasst wird. (Eigentlich können wir Pattern mit Variablen noch gar nicht typisieren.) Es fehlen zudem Regeln, um rekursive Superkombinatoren zu typisieren. All dies werden wir in einem späteren Kapitel nachholen, wir nehmen zunächst einfach an, es gibt einen Algorithmus, der dies für uns entscheiden kann.

2.6 Zusammenfassung

In diesem Kapitel haben wir verschiedene Varianten des Lambda-Kalküls und Erweiterungen des call-by-name Lambda-Kalküls kennen gelernt. Die für Haskell geeignete Kernsprache ist KFPTSP+seq. Die folgende Tabelle gibt nochmals eine Übersicht über die verschiedenen Kernsprachen (ohne Lambda-Kalkül):

Kernsprache	Besonderheiten
KFP	Erweiterung des call-by-name Lambda-Kalküls um ungetyptes `case` und Datenkonstruktoren, spezielles `case`-Pattern `lambda` ermöglicht Kodierung von `seq`.
KFPT	Erweiterung des call-by-name Lambda-Kalküls um (schwach) getyptes `case` und Datenkonstruktoren, `seq` ist nicht kodierbar.
KFPTS	Erweiterung von KFPT um rekursive Superkombinatoren, `seq` nicht kodierbar.
KFPTSP	KFPTS, polymorph getypt; `seq` nicht kodierbar.
KFPT+seq	Erweiterung von KFPT um den `seq`-Operator
KFPTS+seq	Erweiterung von KFPTS um den `seq`-Operator
KFPTSP+seq	KFPTS+seq mit polymorpher Typisierung, geeignete Kernsprache für Haskell

2.7 Quellennachweis und weiterführende Literatur

Zum Lambda-Kalkül gibt es viele Quellen. Ein nicht ganz leicht verständliches Standardwerk ist (Barendregt, 1984). Ein gute Einführung ist in (Hankin, 2004) zu finden. Call-by-need Lambda-Kalküle mit `let` sind in (Ariola et al., 1995; Ariola & Felleisen, 1997) und (Maraist et al., 1998) beschrieben. Nichtdeterministische Erweiterungen sind in (Kutzner & Schmidt-Schauß, 1998; Kutzner, 2000; Mann, 2005b; Mann, 2005a) zu finden. Die vorgestellte call-by-need Reduktion orientiert sich dabei an (Mann, 2005a). Call-by-need Kalküle mit rekursivem `let` wurden u.a. in (Schmidt-Schauß et al., 2008; Schmidt-Schauß et al., 2010) behandelt, nichtdeterministische Erweiterungen sind z.B. in (Sabel & Schmidt-Schauß, 2008; Schmidt-Schauß & Machkasova, 2008; Sabel, 2008) zu finden. Ein polymorph getypter Lambda-Kalkül mit rekursivem `let` ist in (Sabel et al., 2009) zu finden. Der Begriff der kontextuellen Äquivalenz geht auf (Morris, 1968) zurück und wurde im Rahmen des Lambda-Kalküls insbesondere von (Plotkin, 1975) untersucht.

3 Haskell

In diesem Kapitel werden wir die verschiedenen Konstrukte der funktionalen Programmiersprache Haskell erörtern und deuten an, wie diese Konstrukte in die Sprache KFPTSP+seq „verlustfrei“ übersetzt werden können. Umgekehrt bedeutet dies, wir können die Haskell-Konstrukte benutzen, da wir durch die Übersetzung in KFPTSP+seq deren Semantik kennen.

Innerhalb dieses Kapitels werden wir auch weitere Eigenschaften und Methoden von Haskell kennen lernen, die eher unabhängig von KFPTSP+seq sind, z.B. das Modulsystem von Haskell und Haskells Typklassensystem.

Die Übersetzung von `let`- und `where`-Konstrukten behandeln wir direkt, da wir die entsprechenden Verfahren nicht genauer im Rahmen dieser Veranstaltung untersuchen werden. Ein nicht-rekursives `let` lässt sich leicht übersetzen: $\texttt{let}\ x = s\ \texttt{in}\ t \rightsquigarrow (\lambda x.t)\ s$. Für rekursive `let`-Ausdrücke ist das Verfahren schwieriger, da diese im Bindungsbereich anderer Variablen stehen können. Man kann hierfür die Rekursion komplett auflösen (durch Verwendung von Fixpunktkombinatoren). Wir betrachten diese Übersetzung nicht weiter, aber gehen davon aus, dass sie möglich ist. Für `where`-Ausdrücke kann man sich analoge Übersetzungen überlegen.

3.1 Arithmetische Operatoren und Zahlen

Haskell verfügt über eingebaute Typen für ganze Zahlen beschränkter Größe: `Int`, ganze Zahlen beliebiger Länge: `Integer`, Gleitkommazahlen: `Float`, Gleitkommazahlen mit doppelter Genauigkeit: `Double`, und rationale Zahlen `Rational` (bzw. verallgemeinert `Ratio` α, wobei α eine Typvariable ist, der Typ `Rational` ist nur ein Synonym für `Ratio Integer`, die Darstellung ist x `%` y). Die üblichen Rechenoperationen sind verfügbar:

- `+` für die Addition
- `-` für die Subtraktion
- $*$ für die Multiplikation

- `/` für die Division

- `mod` für die Restberechnung

- `div` für den ganzzahligen Anteil einer Division mit Rest

Die Operatoren können für alle Zahl-Typen benutzt werden, d.h. sie sind *überladen*. Diese Überladung funktioniert durch Typklassen (genauer sehen wir das später in Abschnitt 3.7). Die Operationen `(+)`, `(-)`, `(*)` sind für alle Typen der Typklasse `Num` definiert, daher ist deren Typ `Num a => a -> a -> a`.

Eine kleine Anmerkung zum Minuszeichen: Da dieses sowohl für die Subtraktion als auch für negative Zahlen verwendet wird, muss man hier öfter Klammern setzen, damit ein Ausdruck geparst werden kann, z.B. ergibt `1 + -2` einen Parserfehler, richtig ist `1 + (-2)`.

Die üblichen Vergleichsoperationen sind auch bereits eingebaut:

- `==` für den Gleichheitstest (der Typ ist `(==) :: (Eq a) => a -> a -> Bool`, d.h. die zugehörige Typklasse ist `Eq`)

- `/=` für den Ungleichheitstest

- `<`, `<=`, `>`, `>=`, für kleiner, kleiner gleich, größer und größer gleich (der Typ ist `(Ord a) => a -> a -> Bool`, d.h. die zugehörige Typklasse ist `Ord`).

In Haskell sind Prioritäten und Assoziativitäten für die Operatoren vordefiniert und festgelegt (durch die Schlüsselwörter `infixl` (links-assoziativ), `infixr` (rechts-assoziativ) und `infix` (nicht assoziativ, Klammern müssen immer angegeben werden). Die Priorität wird durch eine Zahl angegeben. Z.B. sind in der Prelude vordefiniert:

```
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -

-- The (:) operator is built-in syntax, and cannot
-- legally be given a fixity declaration; but its
-- fixity is given by:
--   infixr 5  :

infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`
```

Diese können auch für selbst definierte Operatoren verwendet werden.

3.1.1 Darstellung von Zahlen in KFPTSP+seq

Zahlen mit endlichem Wertebereich (z.B. `Int`) können direkt in KFPTSP+seq dargestellt werden, indem ein entsprechender Typ mit (0-stelligen) Konstruktoren hinzugefügt wird. Zahlen unbeschränkter Länge können auf diese Weise nicht dargestellt werden, da wir stets angenommen haben, dass die Menge der Konstruktoren zu einem Typ endlich ist (dies ist sehr sinnvoll, anderenfalls hätten wir unendlich große `case`-Ausdrücke für diesen Typ definiert!).

Natürliche Zahlen unbeschränkter Länge können jedoch mit der so genannten *Peano*-Kodierung[1] dargestellt werden:

Wir nehmen an, es gibt einen Typ `Pint` mit zwei Datenkonstruktoren: `Zero` :: `Pint` und `Succ` :: `Pint` $\rightarrow$ `Pint`, d.h `Zero` ist nullstellig und `Succ` ist einstellig. In Haskell kann man dies auch als Datentyp definieren:

```
data Pint = Zero | Succ Pint
 deriving(Eq,Show)
```

[1] nach dem italienischen Mathematiker Giuseppe Peano benannt

Sei n eine natürliche Zahl (einschließlich 0), dann lässt sich die Peano-Darstellung $\mathcal{P}(n)$ definieren durch:

$$\begin{aligned}\mathcal{P}(0) &:= \texttt{Zero}\\ \mathcal{P}(n) &:= \texttt{Succ}(\mathcal{P}(n-1)) \text{ für } n > 0\end{aligned}$$

Z.B. wird 3 dargestellt als `Succ(Succ(Succ(Zero)))`. Wir können nun Operationen auf Peano-Zahlen definieren, der Einfachheit halber geben wir diese als Haskell-Programme an (die Übersetzung in KFPTSP+seq ist offensichtlich).

Eine Funktion, die testet, ob es sich um eine Peano-Zahl handelt (d.h. die Datenstruktur wird komplett ausgewertet), kann definiert werden als:

```
istZahl :: Pint -> Bool
istZahl x  =  case x of {Zero  -> True; (Succ y) ->  istZahl y}
```

Beachte, dass man diese Funktion bei jeder Operation (wie Addition etc.) vorher auf die Zahlen anwenden muss, damit sichergestellt ist, dass die Zahlen zur Gänze ausgewertet werden. Betrachte zum Beispiel den Ausdruck `Succ bot` wobei `bot` als `bot = bot` definiert ist. Dann ist `Succ bot` ein Ausdruck vom Typ `Pint`, aber keine Zahl. Genauso könnte man definieren

```
unendlich :: Pint
unendlich = Succ unendlich
```

was auch keine natürliche Zahl darstellt. Die Verwendung von `istZahl` garantiert, dass sich z.B. + verhält wie in Haskell: + ist strikt in beiden Argumenten, d.h. $\bot + s$ und $s + \bot$ müssen stets nichtterminieren. Die Addition auf Peanozahlen kann definiert werden als:

```
peanoPlus :: Pint -> Pint -> Pint
peanoPlus x y  = if istZahl x &&  istZahl y then plus x y else bot
  where
   plus x  y  =  case x of
                     Zero   -> y
                     Succ z -> Succ (plus z y)
bot = bot
```

Die Multiplikation kann analog definiert werden durch

```
peanoMult :: Pint -> Pint -> Pint
peanoMult x y  = if istZahl x &&  istZahl y then mult x y else bot
  where
   mult x  y  =  case x of
                     Zero   -> Zero
                     Succ z -> peanoPlus y (mult z y)
```

Vergleichsoperationen können auf Peano-Zahlen wie folgt implementiert werden:

```
peanoEq :: Pint -> Pint -> Bool
peanoEq x y = if istZahl x &&  istZahl y then eq x y else bot
 where
  eq Zero Zero          = True
  eq (Succ x) (Succ y)  = eq x y
  eq _        _         = False

peanoLeq :: Pint -> Pint -> Bool
peanoLeq x y = if istZahl x &&  istZahl y then leq x y else bot
 where
  leq Zero y            = True
  leq x    Zero         = False
  leq (Succ x) (Succ y) = leq x y
```

Es gibt auch noch andere Möglichkeiten ganze Zahlen in KFPTSP+seq darzustellen, z.B. als Listen von Bits.

3.2 Algebraische Datentypen in Haskell

Haskell stellt (eingebaute und benutzerdefinierte) Datentypen bzw. Datenstrukturen zur Verfügung.

3.2.1 Aufzählungstypen

Ein Aufzählungstyp ist ein Datentyp, der aus einer Aufzählung von Werten besteht. Die Konstruktoren sind dabei Konstanten (d.h. sie sind nullstellig). Ein vordefinierter Aufzählungstyp ist der Typ `Bool` mit den Konstruktoren `True` und `False`. Allgemein lässt sich ein Aufzählungstyp mit der `data`-Anweisung wie folgt definieren:

```
data Typname = Konstante1 | Konstante2 | ... | KonstanteN
```

Die Darstellung in KFPTSP+seq ist offensichtlich: Der Typ `Typname` hat die 0-stelligen Konstruktoren `Konstante1`, `Konstante2`, ..., `KonstanteN`. Ein Beispiel für einen selbst-definierten Datentyp ist der Typ Wochentag:

```
data Wochentag = Montag
               | Dienstag
               | Mittwoch
               | Donnerstag
               | Freitag
               | Samstag
               | Sonntag
  deriving(Show)
```

Bemerkung 3.2.1. *Fügt man einer Data-Definition die Zeile* `deriving(...)` *hinzu, wobei* `...` *verschiedene Typklassen sind, so versucht der Compiler automatisch Instanzen für den Typ zu generieren, damit man die Operationen wie == für den Datentyp automatisch verwenden kann. Oft fügen wir* `deriving(Show)` *hinzu. Diese Typklasse erlaubt es, Werte des Typs in Strings zu verwandeln, d.h. man kann sie anzeigen. Im Interpreter wirkt sich das Fehlen einer Instanz so aus:*

```
*Main> Montag

<interactive>:1:0:
    No instance for (Show Wochentag)
      arising from a use of ‘print’ at <interactive>:1:0-5
    Possible fix: add an instance declaration for (Show Wochentag)
    In a stmt of a ’do’ expression: print it
```

Mit Show-Instanz kann der Interpreter die Werte anzeigen:

```
*Main> Montag
Montag
```

In KFPTSP+seq können die Werte eines Summentyps mit einem `case`-Ausdruck abgefragt werden (und eine entsprechende Fallunterscheidung durchgeführt werden). Das geht auch in Haskell, z.B.

```
istMontag :: Wochentag -> Bool
istMontag x = case x of
                Montag -> True
                Dienstag -> False
                Mittwoch -> False
                Donnerstag -> False
                Freitag -> False
                Samstag -> False
                Sonntag -> False
```

Haskell erlaubt es jedoch auch, eine default-Alternative im `case` zu verwenden. Dabei wird anstelle des Patterns einfach eine Variable verwendet, diese bindet den gesamten Ausdruck, d.h. in Haskell kann `istMontag` kürzer definiert werden:

```
istMontag' :: Wochentag -> Bool
istMontag' x = case x of
                Montag -> True
                y      -> False
```

Allerdings kann man solche `case`-Ausdrücke einfach in KFPTSP+seq übersetzen, indem man anstelle der default-Alternativen alle Pattern mit der entsprechenden rechten Seite aufführt.

Haskell bietet auch die Möglichkeit (verschachtelte) Pattern links in einer Funktionsdefinition zu benutzen und die einzelnen Fälle durch mehrere Definitionsgleichungen abzuarbeiten z.B.

```
istMontag'' :: Wochentag -> Bool
istMontag'' Montag = True
istMontag'' _      = False
```

Hierbei ist `_` eine namenslose Variable („Wildcard“), die wie eine Variable wirkt aber rechts nicht benutzt werden kann. Die Übersetzung in KFPTSP+seq ist möglich durch Verwendung von `case`-Ausdrücken (also gerade die Ersetzung von `istMontag''` durch `istMontag`). Auf verschachtelte und allgemeinere Pattern gehen wir später ein.

3.2.2 Produkttypen

Produkttypen fassen verschiedene Werte zu einem neuen Typ zusammen. Die bekanntesten Produkttypen sind Paare und mehrstellige Tupel. Diese sind – wie bereits erwähnt – in Haskell vordefiniert. Mit `data` können auch Produkttypen definiert werden. Eine einfache Syntax hierfür ist

```
data Typname = KonstruktorName Typ1 Typ2 ... TypN
```

Ein weiteres Beispiel ist der Datentyp Student:

```
data Student = Student
                  String  -- Name
                  String  -- Vorname
                  Int     -- Matrikelnummer
```

In diesem Beispiel gibt es eine (in Haskell erlaubte) Namensüberlappung: Sowohl der Typ als auch der Datenkonstruktor heißen `Student`. Mit Pattern-Matching und `case`-Ausdrücken kann man die Datenstruktur zerlegen und auf die einzelnen Komponenten zugreifen, z.B.

```
setzeName :: Student -> String -> Student
setzeName x name' =
 case x of
  (Student name vorname mnr) name'
            -> Student name' vorname mnr
```

Alternativ mit Pattern auf der linken Seite der Funktionsdefinition:

```
setzeName :: Student -> String -> Student
setzeName (Student name vorname mnr) name' =
  Student name' vorname mnr
```

Für Produkttypen bietet Haskell noch die so genannte *Record*-Syntax, die wir im nächsten Abschnitt behandeln werden.

Zuvor sei noch erwähnt, dass man Aufzählungstypen und Produkttypen verbinden kann, z.B. kann man einen Datentyp für 3D-Objekte definieren:

```
data DreiDObjekt = Wuerfel Int -- Kantenlaenge
                 | Quader Int Int Int -- Drei Kantenlaengen
                 | Kugel Int -- Radius
```

Allgemein kann man einen solchen Typ definieren als

```
data Typ = Typ1 | ... | Typn
```

wobei `Typ1,...,Typn` selbst komplexe Typen sind, z.B. Produkttypen. Man nennt diese Klasse von Typen oft auch *Summentypen*.

3.2.2.1 Record-Syntax

Haskell bietet neben der normalen Definition von Datentypen auch die Möglichkeit eine spezielle Syntax zu verwenden, die insbesondere dann sinnvoll ist, wenn ein Datenkonstruktor viele Argumente hat.

Wir betrachten zunächst den normal definierten Datentypen `Student` als Beispiel:

```
data Student = Student
                 String  -- Vorname
                 String  -- Name
                 Int     -- Matrikelnummer
```

Ohne die Kommentare ist nicht ersichtlich, was die einzelnen Komponenten darstellen. Außerdem muss man zum Zugriff auf die Komponenten neue Funktionen definieren. Beispielsweise

```
vorname :: Student -> String
vorname (Student vorname name mnr) = vorname
```

Wenn nun Änderungen am Datentyp vorgenommen werden – zum Beispiel eine weitere Komponente für das Hochschulsemester wird hinzugefügt – dann müssen alle Funktionen angepasst werden, die den Datentypen verwenden:

```
data Student = Student
                 String  -- Vorname
                 String  -- Name
                 Int     -- Matrikelnummer
                 Int     -- Hochschulsemester

vorname :: Student -> String
vorname (vorname name mnr hsem) = vorname
```

Um diese Nachteile zu vermeiden, bietet es sich an, die Record-Syntax zu verwenden. Diese erlaubt zum Einen die einzelnen Komponenten mit Namen zu versehen:

```
data Student = Student {
                 vorname        :: String,
                 name           :: String,
                 matrikelnummer :: Int
               }
```

Eine konkrete Instanz würde mit der normalen Syntax initialisiert mittels

```
Student  "Hans" "Mueller" 1234567
```

Für den Record-Typen ist dies genauso möglich, aber es gibt auch die Möglichkeit die Namen zu verwenden:

```
Student{vorname="Hans",
        name="Mueller",
        matrikelnummer=1234567}
```

Hierbei spielt die Reihenfolge der Einträge keine Rolle, z.B. ist

```
Student{vorname="Hans",
        matrikelnummer=1234567,
        name="Mueller"
       }
```

genau dieselbe Instanz.

Zugriffsfunktionen für die Komponenten brauchen nicht zu definiert werden, diese sind sofort vorhanden und tragen den Namen der entsprechenden

Komponente. Z.B. liefert die Funktion `matrikelnummer` angewendet auf eine Student-Instanz dessen Matrikelnummer. Wird der Datentyp jetzt wie oben erweitert, so braucht man im Normalfall wesentlich weniger Änderungen am bestehenden Code.

Die Schreibweise mit Feldnamen darf auch für das Pattern-Matching verwendet werden. Hierbei müssen nicht alle Felder spezifiziert werden. So ist z.B. eine Funktion, die testet, ob der Student einen Nachnamen beginnend mit `'A'` hat, implementierbar als

```
nachnameMitA Student{nachname = 'A':xs} = True
nachnameMitA _ = False
```

Diese Definition ist äquivalent zur Definition

```
nachnameMitA (Student ('A':xs) _ _) = True
nachnameMitA _                      = False
```

Die Record-Syntax kann auch benutzt werden, um „Updates“ durch zu führen[2]. Hierfür verwendet man die nachgestellte Notation `{feldname = ...}`. Die nicht veränderten Werte braucht man dabei nicht neu zu setzen. Wir betrachten ein konkretes Beispiel:

```
setzeName :: Student -> String -> Student
setzeName student neuername = student {name = neuername}
```

Die Funktion setzt den Namen eines Studenten neu. Sie ist äquivalent zu:

```
setzeName :: Student -> String -> Student
setzeName student neuername =
  Student {vorname = vorname student,
           name    = neuername,
           matrikelnummer = matrikelnummer student}
```

Auch wenn KFPTSP+seq keine Record-Syntax unterstützt, so ist jedoch einsichtig, dass diese nur syntaktischer Zucker ist (die Zugriffsfunktionen müssen automatisch erzeugt werden), und daher eine Übersetzung in KFPTSP+seq unproblematisch.

[2] In Wahrheit ist das kein Update, sondern das erstellen eines neuen Werts, da Haskell keine Seiteneffekte erlaubt!

3.2.3 Parametrisierte Datentypen

Haskell bietet die Möglichkeit, Datentypen mit (polymorphen) Parametern zu versehen. D.h. einzelne Komponenten haben eine Typvariable als Argument. Betrachte z.B. den Typ `Maybe`, der definiert ist als:

```
data Maybe a = Nothing | Just a
```

Dieser Datentyp ist polymorph über dem Parameter `a`. Eine konkrete Instanz ist z.B. der Typ `Maybe Int`. Der `Maybe`-Typ kann sinnvoll verwendet werden, wenn man partielle Funktionen definiert, also Funktionen, die nicht für alle Eingaben einen sinnvollen Wert liefern. Dann bietet es sich an, dass Ergebnis als Wert vom Typ `Maybe` zu verpacken: Wenn es ein Ergebnis x gibt, dann wird `Just` x zurückgegeben, anderenfalls `Nothing`. Z.B. kann man damit die Funktion `safeHead`-definieren, die das erste Element einer Liste liefert, und `Nothing`, falls die Liste leer ist.

```
safeHead :: [a] -> Maybe a
safeHead xs = case xs of
                [] ->     Nothing
                (x:xs) -> Just x
```

3.2.4 Rekursive Datenstrukturen

Es ist in Haskell auch möglich, rekursive Datenstrukturen zu definieren. Das prominenteste Beispiel hierfür sind Listen. Auch der bereits definierte Datentyp `Pint` ist rekursiv. Rekursive Datentypen zeichnen sich dadurch aus, dass der gerade definierte Typ wieder als Argument unter einem Typkonstruktor vorkommt. Listen könnte man in Haskell definieren durch

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

Haskell verwendet jedoch die eigene Syntax, d.h. die Definition entspricht eher

```
data [a] = [] | a:[a]
```

In Haskell steht auch die Syntax $[a_1, \ldots, a_n]$ als Abkürzung für $a_1 : (a_2 : (\ldots : []) \ldots)$ zur Verfügung. Haskell erlaubt im Gegensatz zu KFPTSP+seq verschachtelte Pattern, z.B. kann man definieren

```
viertesElement (x1:(x2:(x3:(x4:xs)))) = Just x4
viertesElement _                      = Nothing
```

Für die Übersetzung in KFPTSP+seq müssen diese verschachtelten Pattern durch geschachtelte case-Ausdrücke ersetzt werden, im Beispiel:

```
viertesElement ys = case ys of
                      [] -> Nothing
                      (x1:ys') ->
                        case ys' of
                          [] -> Nothing
                          (x2:ys'') ->
                            case ys'' of
                              [] -> Nothing
                              (x3:ys''') ->
                                case ys''' of
                                  [] -> Nothing
                                  (x4:xs) -> Just x4
```

Es gibt noch weitere sehr sinnvolle rekursive Datentypen, die wir später erörtern werden.

3.3 Listenverarbeitung in Haskell

In diesem Abschnitt erläutern wir die Verarbeitung von Listen und einige prominente Operationen auf Listen.

3.3.1 Listen von Zahlen

Haskell bietet eine spezielle Syntax, um Listen von Zahlen zu erzeugen:

- `[startwert..endwert]` erzeugt die Liste der Zahlen von `startwert` bis `endwert`, z.B. ergibt `[10..15]` die Liste `[10,11,12,13,14,15]`.
- `[startwert..]` erzeugt die unendliche Liste ab dem `startwert`, z.B. erzeugt `[1..]` die Liste aller natürlichen Zahlen.
- `[startwert,naechsterWert..endwert]` erzeugt die Liste `[startwert,startWert+delta,startWert+2delta,...,endwert]`,

wobei `delta=naechsterWert - startWert`. Z.B. ergibt `[10,12..20]` die Liste `[10,12,14,16,18,20]`.

- Analog dazu erzeugt `[startWert,naechsterWert..]` die unendlich lange Liste mit der Schrittweite `naechsterWert - startWert`.

Diese Möglichkeiten sind syntaktischer Zucker, sie können als „normale“ Funktionen definiert werden (in Haskell sind diese Funktionen über die Typklasse `Enum` verfügbar), z.B. für den Datentyp `Integer`:

```
from :: Integer -> [Integer]
from start = start:(from (start+1))

fromTo :: Integer -> Integer -> [Integer]
fromTo start end
 | start > end     = []
 | otherwise = start:(fromTo (start+1) end)

fromThen :: Integer -> Integer -> [Integer]
fromThen start next = start:(fromThen next (2*next - start))

fromThenTo :: Integer -> Integer -> Integer  -> [Integer]
fromThenTo start next end
 | start > end = []
 | otherwise   = start:(fromThenTo next (2*next - start) end)
```

Beachte: In dieser Definition haben wir *Guards* benutzt. Diese bieten eine elegante Möglichkeit, Fallunterscheidungen durchzuführen. Die allgemeine Syntax ist

```
 f pat1 ... patn
    | guard1 = e1
    | ...
    | guardn = en
```

Hierbei sind `guard1` bis `guardn` Boolesche Ausdrücke, die die Variablen der Pattern `pat1,...,patn` benutzen dürfen. Die Guards werden von oben nach unten ausgewertet. Liefert ein Guard `True`, so wird die entsprechende rechte Seite `ei` als Resultat übernommen. Im Beispiel oben haben wir den Guard `otherwise` verwendet, dieser ist nur ein Synonym für `True`, d.h.

```
otherwise = True
```

ist vordefiniert. Guards lassen sich in KFPTSP+seq übersetzen, indem man verschachtelte `if-then-else`-Ausdrücke benutzt (und diese durch $\texttt{case}_{\texttt{Bool}}$-Ausdrücke darstellt). Etwa in der Form

```
if guard1 then e1 else
 if guard2 then e2 else
  ...
   if guardn then en else s
```

Hierbei ist `s = bot`, wenn keine weitere Funktionsdefinition für `f` kommt, anderenfalls ist `s` die Übersetzung anderer Definitionsgleichungen.

Wir betrachten ein Beispiel:

```
f (x:xs)
  | x > 10  = True
  | x < 100 = True
f ys        = False
```

Die korrekte Übersetzung in KFPTSP+seq (mit `if-then else`) ist:

```
f = case x of {
      Nil -> False;
      (x:xs) -> if x > 10 then True else
                  if x < 100 then True else False
      }
```

3.3.2 Strings

Neben Zahlenwerten gibt es in Haskell den eingebauten Typ `Char` zur Darstellung von Zeichen. Die Darstellung erfolgt in einfachen Anführungszeichen, z.B. `'A'`. Es gibt spezielle Zeichen wie Steuersymbole wie `'\n'` für Zeilenumbrüche, `'\\'` für den Backslash \ etc. Zeichenketten vom Typ `String` sind nur vordefiniert. Sie sind Listen vom Typ `Char`, d.h. `String = [Char]`. Allerdings wird syntaktischer Zucker verwendet durch Anführungszeichen: `"Hallo"` ist eine abkürzende Schreibweise für die Liste `'H':'a':'l':'l':'o':[]`. Man kann Strings genau wie jede andere Liste verarbeiten.

3.3.3 Standard-Listenfunktionen

Wir erläutern einige Funktionen auf Listen, die in Haskell bereits vordefiniert sind und sehr nützlich sind.

3.3.3.1 Append

Der Operator ++ vereinigt zwei Listen (gesprochen als „append“). Einige Beispielverwendungen sind

```
*Main> [1..10] ++ [100..110]
[1,2,3,4,5,6,7,8,9,10,100,101,102,103,104,105,106,107,108,109,110]
*Main> [[1,2],[2,3]] ++ [[3,4,5]]
[[1,2],[2,3],[3,4,5]]
*Main> "Infor" ++ "matik"
"Informatik"
```

Die Definition in Haskell ist

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

Das Laufzeitverhalten ist linear in der Länge der ersten Liste (wenn man alle append-Operationen auswertet).

3.3.3.2 Zugriff auf ein Element

Der Operator `!!` erlaubt den Zugriff auf ein Listenelement an einer bestimmten Position. Die Indizierung beginnt dabei mit 0. Die Implementierung ist:

```
(!!) :: [a] -> Int -> a
[]     !! _ = error "Index too large"
(x:xs) !! 0 = x
(x:xs) !! i = xs !! (i-1)
```

Umgekehrt kann man mit `elemIndex` den Index eines Elements berechnen. Wenn das Element mehrfach in der Liste vorkommt, so zählt das erste Vorkommen. Um den Fall abzufangen, dass das Element gar nicht vorkommt,

wird das Ergebnis durch den `Maybe`-Typ verpackt: Ist das Element nicht vorhanden, so wird `Nothing` zurück geliefert, andernfalls `Just i`, wobei `i` der Index ist:

```
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
elemIndex a xs = findInd 0 a xs
  where findInd i a [] = Nothing
        findInd i a (x:xs)
          | a == x    = Just i
          | otherwise = findInd (i+1) a xs
```

Wir geben einige Beispielaufrufe an:

```
*Main Data.List> [1..10]!!0
1
*Main Data.List> [1..10]!!9
10
*Main Data.List> [1..10]!!10
*** Exception: Prelude.(!!): index too large

*Main Data.List> elemIndex 5 [1..10]
Just 4
*Main Data.List> elemIndex 5 [5,5,5,5]
Just 0
*Main Data.List> elemIndex 5 [1,5,5,5,5]
Just 1
*Main Data.List> elemIndex 6 [1,5,5,5,5]
Nothing
```

3.3.3.3 Map

Die Funktion `map` wendet eine Funktion auf die Elemente einer Liste an:

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

Einige Beispielverwendungen:

```
*Main> map (*3) [1..20]
[3,6,9,12,15,18,21,24,27,30,33,36,39,42,45,48,51,54,57,60]
*Main> map not [True,False,False,True]
[False,True,True,False]
*Main> map (^2) [1..10]
[1,4,9,16,25,36,49,64,81,100]
*Main> :m + Char
*Main Char> map toUpper "Informatik"
"INFORMATIK"
```

3.3.3.4 Filter

Die Funktion `filter` erhält eine Testfunktion und filtert die Elemente in die Ergebnisliste, die den Test erfüllen.

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

Einige Beispielaufrufe:

```
*Main> filter (> 15) [10..20]
[16,17,18,19,20]
*Main> filter even [10..20]
[10,12,14,16,18,20]
*Main> filter odd [1..9]
[1,3,5,7,9]
*Main Char> filter isAlpha "2010 Informatik 2010"
"Informatik"
```

Man kann analog eine Funktion `remove` definieren, die die Elemente entfernt, die einen Test erfüllen:

```
remove p xs = filter (not . p) xs
```

Beachte: Für die Definition haben wir den Kompositionsoperator `(.)` benutzt, der dazu dient Funktionen zu komponieren. Er ist definiert als

```
(f . g) x = f (g x)
```

3.3.3.5 Length

Die Funktion `length` berechnet die Länge einer Liste als `Int`-Wert.

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

Beispielverwendungen sind

```
*Main Char> length "Informatik"
10
*Main Char> length [2..20002]
20001
```

3.3.3.6 Reverse

Die Funktion `reverse` dreht eine (endliche) Liste um. Beachte, auf unendlichen Listen terminiert `reverse` nicht. Eine eher schlechte Definition ist:

```
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]
```

Da die Laufzeit von ++ linear in der linken Liste ist, folgt, dass der Aufwand von `reverse` quadratisch ist. Besser (linear) ist die Definition mit einem Akkumulator (Stack):

```
reverse :: [a] -> [a]
reverse xs = rev xs []
 where
   rev [] acc     = acc
   rev (x:xs) acc = rev xs (x:acc)
```

Beispielaufrufe:

```
*Main> reverse "Informatik"
"kitamrofnI"
*Main> reverse [1..10]
[10,9,8,7,6,5,4,3,2,1]
*Main> reverse "RELIEFPFEILER"
"RELIEFPFEILER"
*Main> reverse [1..]
^CInterrupted.
```

3.3.3.7 Repeat und Replicate

Die Funktion `repeat` erzeugt eine unendliche Liste von gleichen Elementen, `replicate` erzeugt eine bestimmte Anzahl von gleichen Elementen:

```
repeat :: a -> [a]
repeat x = x:(repeat x)

replicate :: Int -> a -> [a]
replicate 0 x = []
replicate i x = x:(replicate (i-1) x)
```

Beispiele:

```
repeat 1
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,^C1Interrupted.
*Main> replicate 10 [1,2]
[[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2]]
*Main> replicate 20 'A'
"AAAAAAAAAAAAAAAAAAAA"
```

3.3.3.8 Take und Drop

Die Funktion `take` nimmt eine Anzahl der ersten Elemente einer Liste, die Funktion `drop` wirft eine Anzahl an Elementen weg. Analog dazu nimmt `takeWhile` die Elemente einer Liste, solange eine Prädikat erfüllt ist und `dropWhile` verwirft Elemente, solange ein Prädikat erfüllt ist:

```
take :: Int -> [a] -> [a]
take i [] = []
take 0 xs = []
take i (x:xs) = x:(take (i-1) xs)

drop i []     = []
drop 0 xs     = xs
drop i (x:xs) = drop (i-1) xs

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

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

Einige Beispielaufrufe:

```
*Main> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
*Main> drop 5 "Informatik"
"matik"
*Main> takeWhile (> 5) [5,6,7,3,6,7,8]
[]
*Main> takeWhile (> 5) [7,6,7,3,6,7,8]
[7,6,7]
*Main> dropWhile (< 10) [1..20]
[10,11,12,13,14,15,16,17,18,19,20]
```

3.3.3.9 Zip und Unzip

Die Funktion `zip` vereint zwei Listen zu einer Liste von Paaren, `unzip` arbeitet umgekehrt: Sie zerlegt eine Liste von Paaren in das Paar zweier Listen:

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

unzip :: [(a, b)] -> ([a], [b])
unzip [] = ([],[])
unzip ((x,y):xs) = let (xs',ys') = unzip xs
                   in (x:xs',y:ys')
```

Beispielaufrufe:

```
*Main> zip [1..10] "Informatik"
[(1,'I'),(2,'n'),(3,'f'),(4,'o'),(5,'r'),
 (6,'m'),(7,'a'),(8,'t'),(9,'i'),(10,'k')]
*Main> unzip [(1,'I'),(2,'n'),(3,'f'),(4,'o'),(5,'r'),
(6,'m'),(7,'a'),(8,'t'),(9,'i'),(10,'k')]
([1,2,3,4,5,6,7,8,9,10],"Informatik")
```

Beachte: Man kann zwar diese Implementierung übertragen zu z.B. `zip3` zur Verarbeitung von drei Listen, man kann jedoch in Haskell keine Implementierung für `zipN` zum Packen von `n` Listen angeben, da diese nicht typisierbar ist.

Ein allgemeinere Variante von `zip` ist die Funktion `zipWith`. Diese erhält als zusätzliches Argument einen Operator, der angibt, wie die beiden Listen miteinander verbunden werden sollen. Z.B. kann `zip` mittels `zipWith` definiert werden durch:

```
zip = zipWith (\x y -> (x,y))
```

Die Definition von `zipWith` ist:

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

Interpretiert man Listen von Zahlen als Vektoren, so kann man die Vektoraddition mit `zipWith` implementieren:

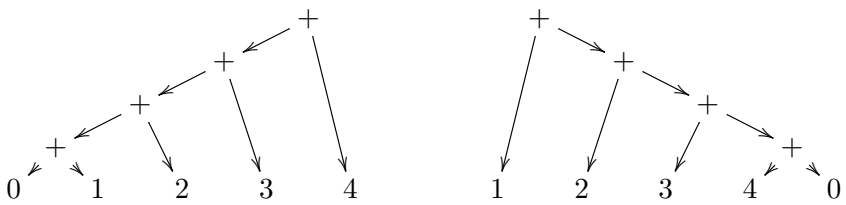
```
vectorAdd :: (Num a) => [a] -> [a] -> [a]
vectorAdd = zipWith (+)
```

3.3.3.10 Die Fold-Funktionen

Die Funktionen `foldl` und `foldr` „falten“ eine Liste zusammen. Dafür erwarten sie einen binären Operator (um die Listenelemente miteinander zu verbinden) und ein Element für den Einstieg. Man kann sich merken, dass `foldl` die Liste links-faltet, während `foldr` die Liste rechts faltet, genauer:

- `foldl` $\otimes$ e $[a_1, \ldots, a_n]$ ergibt $(\ldots((e \otimes a_1) \otimes a_2)\ldots) \otimes a_n$
- `foldr` $\otimes$ e $[a_1, \ldots, a_n]$ ergibt $a_1 \otimes (a_2 \otimes (\ldots \otimes (a_n \otimes e)\ldots))$

Zur Veranschaulichung kann man die Ergebnis auch als Syntaxbäume aufzeichnen: Wir betrachten als Beispiel `foldll (+) 0 [1,2,3,4]` und `foldlr (+) 0 [1,2,3,4]`. Die Ergebnisse werden entsprechend der folgenden Syntaxbäume berechnet:



Die Definitionen in Haskell sind[3]:

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

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

[3]Beachte: In Haskell kann man Präfix-Operatoren auch infix verwenden, indem man sie in Hochkommata setzt, z.B. ist `mod 5 2` äquivalent zu `5 `mod` 2`. Umgekehrt kann man Infix-Operatoren auch Präfix verwenden, indem man sie einklammert, z.B. ist `1 + 2` äquivalent zu `(+) 1 2`

Ein Beispiel zur Verwendung von `fold` ist die Definition von `concat`. Diese Funktion nimmt eine Liste von Listen und vereint die Listen zu einer Liste:

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

Beachte das man auch `foldl` verwenden könnte (da ++ assoziativ und `[]` links- und rechts-neutral ist), allerdings wäre dies ineffizienter, da die Laufzeit von ++ linear in der linken Liste ist.

Weitere Beispiele sind `sum` und `product`, die die Summe bzw. das Produkt einer Liste von Zahlen berechnen.

```
sum     = foldl (+) 0
product = foldl (*) 1
```

Vom Platzbedarf ist hierbei die Verwendung von `foldl` und `foldr` zunächst identisch, da zunächst die gesamte Summe bzw. das gesamte Produkt als Ausdruck aufgebaut wird, bevor die Berechnung stattfindet. D.h. der Platzbedarf ist linear in der Länge der Liste. Man kann das optimieren, indem man eine optimierte Version von `foldl` verwendet, die strikt im zweiten Argument ist. Diese Funktion ist als `fold'` im Modul `Data.List` definiert, sie erzwingt mittels `seq` die Auswertung von `(e `f` x)` vor dem rekursiven Aufruf:

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f e []     = e
foldl' f e (x:xs) = let e' = e `f` x in e' `seq` foldl' f e' xs
```

Diese Variante verbraucht nur konstanten Platz.

Allerdings ist zu beachten, dass sich `foldl` und `foldl'` nicht immer semantisch gleich verhalten: Durch die erzwungene Auswertung innerhalb des `foldl'` terminiert `foldl` öfter als `foldl'`. Betrachte z.B. die Ausdrücke:

```
foldl  (\x y -> y) 0 [undefined, 1]
foldl' (\x y -> y) 0 [undefined, 1]
```

Die zum Falten genutzte Funktion bildet konstant auf das zweite Argument ab. Der `foldl`-Aufruf ergibt daher zunächst `(\x y -> y) ((\x y -> y) 0 undefined) 1`. Anschließend wertet die verzögerte Auswertung den Ausdruck direkt zu `1` aus. Der `foldl'`-Aufruf

wertet jedoch durch die Erzwingung mit `seq` zunächst das Zwischenergebnis `((\x y -> y) 0 undefined)` aus. Da dieser Ausdruck jedoch keine WHNF besitzt, terminiert die gesamte Auswertung des `foldl'`-Ausrufs nicht[4].

Es gibt spezielle Varianten für `foldl` und `foldr`, die ohne „neutrales Element“ auskommen. Bei leerer Liste liefern diese Funktionen einen Fehler.

```
foldr1                  :: (a -> a -> a) -> [a] -> a
foldr1 _ []             =  error "foldr1 on an empty list"
foldr1 _ [x]            =  x
foldr1 f (x:xs)         =  f x (foldr1 f xs)

foldl1                  :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)         =  foldl f x xs
foldl1 _ []             =  error "foldl1 on an empty list"
```

Beachte, dass in der Definition von `foldr1` ein spezielles Pattern verwendet wird: `[x]`. Solche Pattern sind auch für längere Listen erlaubt. Die Übersetzung in KFPTSP+seq ist jedoch offensichtlich.

3.3.3.11 Scanl und Scanr

Ähnlich zu `foldl` und `foldr` gibt es die Funktionen `scanl` und `scanr`, die eine Liste auf die gleiche Art „falten“, jedoch die Zwischenergebnisse in einer Liste zurückgegeben, d.h.

- `scanl` $\otimes\ e\ [a_1, a_2, ..., a_n] = [e, e \otimes a_1, (e \otimes a_1) \otimes a_2, \ldots]$
- `scanr` $\otimes\ e\ [a_1, a_2, ..., a_n] = [\ldots, a_{n-1} \otimes (a_n \otimes e), a_n \otimes e, e]$

Beachte, dass gilt:

- `last (scanl f e xs) = foldl f e xs`[5] und
- `head (scanr f e xs) = foldr f e xs`.

Die Definitionen sind:

[4] Man kann auch anstelle von `undefined` den Ausdruck `bot` benutzen, wobei die Definition in Haskell `bot = bot` sei. `undefined` hat zum Testen den Vorteil, dass anstelle einer Endlosschleife ein Laufzeitfehler gemeldet wird.

[5] `last` berechnet das letzte Element einer Liste

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f e xs = e:(case xs of
                    [] ->  []
                    (y:ys) -> scanl f (e `f` y) ys)

scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ e []           =  [e]
scanr f e (x:xs)       =  f x q : qs
                           where qs@(q:_) = scanr f e xs
```

Die Definition von `scanr` verwendet ein so genanntes „as“-Pattern `qs@(q:_)`. Dabei wird der Liste, die durch das Pattern `(q:_)` gematcht wird, zusätzlich der Name `qs` gegeben.

Beispielaufrufe für die Funktionen `scanl` und `scanr`:

```
Main> scanr (++) [] [[1,2],[3,4],[5,6],[7,8]]
[[1,2,3,4,5,6,7,8],[3,4,5,6,7,8],[5,6,7,8],[7,8],[]]
Main> scanl (++) [] [[1,2],[3,4],[5,6],[7,8]]
[[],[1,2],[1,2,3,4],[1,2,3,4,5,6],[1,2,3,4,5,6,7,8]]
Main> scanl (+) 0 [1..10]
[0,1,3,6,10,15,21,28,36,45,55]
Main> scanr (+) 0 [1..10]
[55,54,52,49,45,40,34,27,19,10,0]
```

3.3.3.12 Partition und Quicksort

Ähnlich wie `filter` und `remove` arbeitet die Funktion `partition`: Sie erwartet ein Prädikat und eine Liste und liefert ein Paar von Listen, wobei die erste Liste die Elemente der Eingabe enthält, die das Prädikat erfüllen und die zweite Liste die Elemente der Eingabe enthält, die das Prädikat nicht erfüllen. Eine einfache aber nicht ganz effiziente Implementierung ist:

```
partition p xs = (filter p xs, remove p xs)
```

Diese Implementierung geht die Eingabeliste jedoch zweimal durch. Besser ist die folgende Definition, die die Liste nur einmal liest:

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition p [] = ([],[])
partition p (x:xs)
  | p x       = (x:r1,r2)
  | otherwise = (r1,x:r2)
  where (r1,r2) = partition p xs
```

Mithilfe von `partition` kann man recht einfach den Quicksort-Algorithmus implementieren, wobei wir das erste Element als Pivot-Element wählen:

```
quicksort :: (Ord a) => [a] -> [a]
quicksort []  = []
quicksort [x] = [x]
quicksort (x:xs) = let (kleiner,groesser) = partition (<x) xs
                   in quicksort kleiner ++ (x:(quicksort groesser))
```

Der Typ von `quicksort` hat eine Typklassenbeschränkung, denn es können nur Elemente sortiert werden, die auch verglichen werden können.

3.3.3.13 Listen als Ströme

Da Listen in Haskell auch unendlich lang sein können, diese aber nicht unbedingt ausgewertet werden müssen (und daher nicht sofort zu Nichtterminierung führen), kann man Listen auch als potentiell unendlich lange Ströme von Elementen auffassen. Verarbeitet man solche Ströme, so muss man darauf achten, die Verarbeitung so zu programmieren, dass sie die Listen auch nicht bis zum Ende auswerten wollen. Funktionen, die Listen bis zum Ende auswerten, also insbesondere nichtterminieren auf unendlichen Listen oder auf Listen, deren n-ter Tail $\bot$ ist (Listen der Form $a_1 : \ldots : a_{n-1} : \bot$) nennt man *tail-strikt*. Wir haben schon einige tail-strikte Listenfunktionen betrachtet, z.B. sind `reverse`, `length` und die fold-Funktionen tail-strikt. Funktionen f, die Ströme in Ströme verwandeln und bei den `take` n $(f\ list)$ für jede Eingabeliste $list$ und jedes n terminiert, nennt man *strom-produzierend*. Wir sehen das etwas weniger restriktiv und akzeptieren auch solche Funktionen, die nur für fast alle Eingabeströme diese Eigenschaft haben.

Von den vorgestellten Funktionen eignen sich zur Stromverarbeitung z.B. `map`, `filter`, `zip` und `zipWith` (zur Verarbeitung mehrerer Ströme), `take`, `drop`, `takeWhile`, `dropWhile`. Für Strings gibt es die nützlichen Funktionen

`words :: String -> [String]` (Zerlegen einer Zeichenkette in eine Liste von Wörtern), `lines :: String -> [String]` (Zerlegen einer Zeichenkette in eine Liste der Zeilen), `unlines :: [String] -> String` (einzelne Zeilen in einer Liste zu einem String zusammenfügen (mit Zeilenumbrüchen)).

Übungsaufgabe 3.3.1. *Gegeben sei als Eingabe ein Text in Form eines Strings (als Strom). Implementiere eine Funktion, die den Text mit Zeilennummern versieht. Die Funktion sollte dabei auch für unendliche Ströme funktionieren. Tipp: Verwende* `lines`, `unlines` *und* `zipWith`.

All diese Funktionen sind nicht tail-rekursiv und produzieren die Ausgabeliste nach und nach.

Wir betrachten als weiteres Beispiel das Mischen von sortierten Strömen:

```
merge :: (Ord t) => [t] -> [t] -> [t]
merge []       ys  = ys
merge xs       []  = xs
merge a@(x:xs) b@(y:ys)
  | x <= y    = x:merge xs b
  | otherwise = y:merge a ys
```

Ein Beispielaufruf ist:

```
*Main> merge [1,3,5,6,7,9] [2,3,4,5,6]
[1,2,3,3,4,5,5,6,6,7,9]
```

Die Funktion `nub` entfernt doppelte Elemente eines (auch unsortierten) Stromes:

```
nub xs  = nub' xs []
  where
    nub' [] _           = []
    nub' (x:xs) seen
        | x `elem` seen = nub' xs seen
        | otherwise     = x : nub' xs (x:seen)
```

In der Liste `seen` merkt sich `nub'` dabei die schon gesehenen Elemente. Dabei wird die vordefinierte Funktion `elem` verwendet. Sie prüft ob ein Element in einer Liste enthalten ist; sie ist definiert als:

```
elem e []    = False
elem e (x:xs)
  | e == x   = True
  | otherwise = elem e xs
```

Die Laufzeit von `nub` ist u.U. quadratisch. Bei sortierten Listen geht dies mit folgender Funktion in linearer Zeit:

```
nubSorted (x:y:xs)
  | x == y   = nubSorted (y:xs)
  | otherwise = x:(nubSorted (y:xs))
nubSorted y = y
```

Beachte: Die letzte Zeile fängt zwei Fälle ab: Sowohl den Fall einer einelementigen Liste als auch den Fall einer leeren Liste.

Ein Verwendungsbeispiel ist die alte Aufgabe, die Mehrfachen con 3,5,7 so zu mergen, dass Elemente inaufsteigender Reihenfoge und nicht doppelt erscheinen.

```
*> nubSorted $ merge (map (3*) [1..])
*>    (merge (map (5*) [1..]) (map (7*) [1..]))
 [3,5,6,7,9,10,12,14,15,18,20,..
```

3.3.3.14 Lookup

Die Funktion `lookup` sucht in einer Liste von Paaren nach einem Element mit einem bestimmten Schlüssel. Ein Paar der Liste soll dabei von der Form (Schlüssel,Wert) sein. Die Rückgabe von `lookup` ist vom `Maybe`-Typ: Wurde ein passender Eintrag gefunden, wird `Just` Wert geliefert, anderenfalls `Nothing`. Die Implementierung ist:

```
lookup                  :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup _key []          =  Nothing
lookup  key ((x,y):xys)
    | key == x          =  Just y
    | otherwise         =  lookup key xys
```

3.3.3.15 Mengenoperation

Die Bibliothek `Data.List` stellt einige Operationen zur Verfügung, die Listen wie Mengen behandeln:

- Die Funktionen `any` und `all` wirken wie Existenz- und Allquantoren. Sie testen ob es ein Element gibt bzw. ob alle Elemente einen Test erfüllen:

```
any _ []     = False
any p (x:xs)
 | (p x)     = True
 | otherwise = any xs

all _ []     = True
all p (x:xs)
 | (p x)     = any xs
 | otherwise = False
```

- `delete` löscht ein passendes Element aus der Liste:

```
delete :: (Eq a) => a -> [a] -> [a]
delete e (x:xs)
 | e == x    = xs
 | otherwise = x:(delete e xs)
```

  Beachte, dass nur das erste Vorkommen gelöscht wird.

- Der Operator `(\\)` berechnet die „Mengendifferenz“ zweier Listen. Er ist implementiert als:

```
(\\) :: (Eq a) => [a] -> [a] -> [a]
(\\) =  foldl (flip delete)
```

  Die Funktion `flip` dreht die Argumente einer Funktion um, d.h.

```
flip :: (a -> b -> c) -> b -> a -> c
flip f a b = f b a
```

- Die Funktion `union` vereinigt zwei Listen:

```
union :: (Eq a) => [a] -> [a] -> [a]
union xs ys =  xs ++ (ys \\ xs)
```

- Die Funktion `intersect` berechnet den Schnitt zweier Listen:

```
intersect :: (Eq a) => [a] -> [a] -> [a]
intersect xs ys = filter (\y -> any (== y) ys) xs
```

Einige Beispielaufrufe:

```
Main> union [1,2,3,4] [9,6,4,3,1]
[1,2,3,4,9,6]
Main> union [1,2,3,4,4] [9,6,4,3,1]
[1,2,3,4,4,9,6]
Main> union [1,2,3,4,4] [9,6,4,4,3,1]
[1,2,3,4,4,9,6]
Main> union [1,2,3,4,4] [9,9,6,4,4,3,1]
[1,2,3,4,4,9,6]
Main> delete 3 [1,2,3,4,5,3,4,3]
[1,2,4,5,3,4,3]
Main>  [1,2,3,4,4] \\ [9,6,4,4,3,1]
[2]
Main>  [1,2,3,4] \\ [9,6,4,4,3,1]
[2]
Main> intersect [1,2,3,4,4] [9,9,6,4,4,3,1]
[1,3,4,4]
Main> intersect [1,2,3,4,4] [9,9,6,4,3,1]
[1,3,4,4]
Main> intersect [1,2,3,4] [9,9,6,4,3,1]
[1,3,4]
```

3.3.4 List Comprehensions

Haskell bietet noch eine weitere syntaktische Möglichkeit zur Erzeugung und Verarbeitung von Listen. Mit *List Comprehensions* ist es möglich, Listen

in ähnlicher Weise wie die übliche Mengenschreibweise – so genannten ZF-Ausdrücke[6] – zu benutzen.

List Comprehensions haben die folgende Syntax:

```
[Expr | qual1,...,qualn]
```

wobei `Expr` ein Ausdruck ist (dessen freie Variablen durch `qual1,...,qualn` gebunden sein müssen) und $qual_i$ entweder:

- ein *Generator* der Form `pat <- Expr`, oder
- ein *Guard*, d.h. ein Ausdruck booleschen Typs,
- oder eine *Deklaration lokaler Bindungen* der Form `let x1=e1,...,xn=en` (ohne `in`-Ausdruck!) ist.

Wir betrachten einige einfache Beispiele:

- `[x | x <- [1..]]` erzeugt die Liste der natürlichen Zahlen
- `[(x,y) | x <- [1..], y <- [1..]]` erzeugt das kartesische Produkt der natürlichen Zahlen, ein Aufruf:

  ```
  Prelude> take 10 $ [(x,y) | x <- [1..], y <- [1..]]
  [(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)]
  ```

- `[x | x <- [1..], odd x]` erzeugt die Liste aller ungeraden natürlichen Zahlen.
- `[x*x | x <- [1..]]` erzeugt die Liste aller Quadratzahlen.
- `[(x,y)| x <- [1..], let y = x*x]` erzeugt die Liste aller Paare (Zahl,Quadrat der Zahl).
- `[a | (a,_,_,_) <- [(x,x,y,y) | x <- [1..3], y <- [1..3]]]` ergibt `[1,1,1,2,2,2,3,3,3]`
- Eine alternative Definition für `map` mit List Comprehension ist `map f xs = [f x | x <- xs]`
- Eine alternative Definition für `filter` mit List Comprehension ist `filter p xs = [x | x <- xs, p x]`

[6] nach der Zermelo-Fränkel Mengenlehre

- Auch `concat` kann mit List Comprehensions definiert werden: `concat xs = [y | xs <- xss, y <- xs]`
- Quicksort mit List Comprehensions:

```
qsort (x:xs) =    qsort [y | y <- xs, y <= x]
               ++ [x]
               ++ qsort [y | y <- xs, y > x]
qsort x = x
```

Die obige Definition des kartesischen Produkts und der Beispielaufruf zeigt die Reihenfolge in der mehrere Generatoren abgearbeitet werden. Zunächst wird ein Element des ersten Generators mit allen anderen Elementen des nächsten Generators verarbeitet, usw. Deswegen wird obige Definition nie das Paar `(2,1)` generieren.

Man kann List Comprehensions in ZF-freies Haskell übersetzen, d.h. sie sind nur syntaktischer Zucker. Die Übersetzung entsprechend dem Haskell Report ist:

```
[ e | True ]         =   [e]
[ e | q ]            =   [ e | q, True ]
[ e | b, Q ]         =   if b then [ e | Q ] else []
[ e | p <- l, Q ]    =   let ok p = [ e | Q ]
                             ok _ = []
                         in concatMap ok l
[ e | let decls, Q ] =   let decls in [ e | Q ]
```

Hierbei ist `ok` eine neue Variable, `b` ein Guard, `q` ein Generator, eine lokale Bindung oder ein Guard (aber nicht `True`), und `Q` eine Folge von Generatoren, Deklarationen und Guards.

Beispiel 3.3.2. `[x*y | x <- xs, y <- ys, x > 2, y < 3]` *wird übersetzt zu*

```
   [x*y | x <- xs, y <- ys, x > 2, y < 3]

=  let ok x = [x*y | y <- ys, x > 2, y < 3]
       ok _ = []
   in concatMap ok xs
```

```
=  let ok x = let ok' y = [x*y | x > 2, y < 3]
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs

=  let ok x = let ok' y = if x > 2 then [x*y | y < 3] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs

=  let ok x = let ok' y = if x > 2 then [x*y | y < 3, True] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs

=  let ok x = let ok' y = if x > 2 then
                            (if y < 3 then [x*y | True] else [])
                          else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs


=  let ok x = let ok' y = if x > 2 then
                            (if y < 3 then [x*y] else [])
                          else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Die Übersetzung ist nicht optimal, da Listen generiert und wieder abgebaut werden.

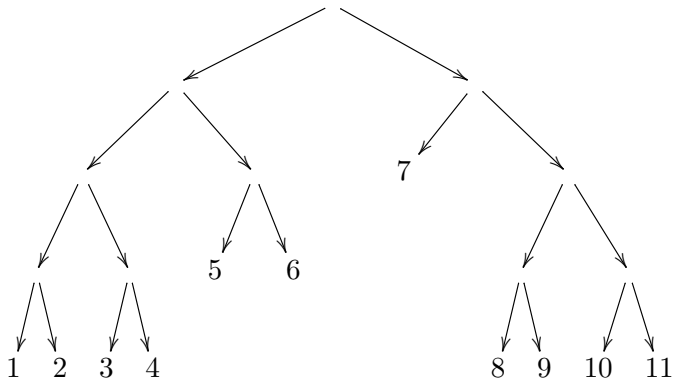
3.4 Rekursive Datenstrukturen: Bäume

Binäre Bäume mit (polymorphen) Blattmarkierungen kann man in Haskell als rekursiven Datentyp wie folgt definieren:

```
data BBaum a = Blatt a | Knoten (BBaum a) (BBaum a)
 deriving(Eq,Show)
```

Hierbei ist `BBaum` der Typkonstruktor und `Blatt` und `Knoten` sind Datenkonstruktoren.

Den Baum



kann man als Instanz des Typs `BBaum Int` in Haskell angeben als:

```
beispielBaum =
  Knoten
    (Knoten
      (Knoten
          (Knoten (Blatt 1) (Blatt 2))
          (Knoten (Blatt 3) (Blatt 4))
      )
      (Knoten (Blatt 5) (Blatt 6))
    )
    (Knoten
      (Blatt 7)
      (Knoten
          (Knoten (Blatt 8) (Blatt 9))
          (Knoten (Blatt 10) (Blatt 11))
      )
    )
```

Wir betrachten einige Funktionen auf solchen Bäumen. Die Summe der Blätter kann man berechnen mit

```
bSum (Blatt a) = a
bSum (Knoten links rechts) = (bSum links) + (bSum rechts)
```

Ein Beispielaufruf:

```
*Main> bSum beispielBaum
66
```

Die Liste der Blätter erhält man analog:

```
bRand (Blatt a) = [a]
bRand (Knoten links rechts) = (bRand links) ++ (bRand rechts)
```

Für den Beispielbaum ergibt dies:

```
Main> bRand beispielBaum
[1,2,3,4,5,6,7,8,9,10,11]
```

Analog zur Listenfunktion `map` kann man auch eine `bMap`-Funktion implementieren, die eine Funktion auf alle Blätter anwendet:

```
bMap f (Blatt a) = Blatt (f a)
bMap f (Knoten links rechts) = Knoten (bMap f links) (bMap f rechts)
```

Z.B. kann man alle Blätter des Beispielbaums quadrieren:

```
*Main> bMap (^2) beispielBaum
Knoten (Knoten (Knoten (Knoten (Blatt 1) (Blatt 4))
(Knoten (Blatt 9) (Blatt 16))) (Knoten (Blatt 25) (Blatt 36)))
(Knoten (Blatt 49) (Knoten (Knoten (Blatt 64) (Blatt 81))
(Knoten (Blatt 100) (Blatt 121))))
```

Die Anzahl der Blätter eines Baumes kann man wie folgt berechnen:

```
anzahlBlaetter = bSum . bMap (\x -> 1)
```

Für den Beispielbaum ergibt dies:

```
*Main> anzahlBlaetter beispielBaum
11
```

Eine Funktion, die testet, ob es ein Blatt mit einer bestimmten Markierung gibt:

```
bElem e (Blatt a)
 | e == a          = True
 | otherwise       = False
bElem e (Knoten links rechts) = (bElem e links) || (bElem e rechts)
```

Einige Beispielaufrufe:

```
*Main> 11 `bElem` beispielBaum
True
*Main> 1 `bElem` beispielBaum
True
*Main> 20 `bElem` beispielBaum
False
*Main> 0 `bElem` beispielBaum
False
```

Man kann ein fold über solche Bäume definieren, die die Blätter mit einem binären Operator entsprechend der Baumstruktur verknüpft:

```
bFold op (Blatt a) = a
bFold op (Knoten a b) = op (bFold op a) (bFold op b)
```

Damit kann man z.B. die Summe und das Produkt berechnen:

```
*Main> bFold (+) beispielBaum
66
*Main> bFold (*) beispielBaum
39916800
```

Offensichtlich kann man Datentypen für Bäume mit anderem Grad (z.B. ternäre Bäume o.ä.) analog zu binären Bäumen definieren. Bäume mit beliebigem Grad (und auch unterschiedlich vielen Kindern pro Knoten) kann man mithilfe von Listen definieren:

```
data NBaum a = NBlatt a | NKnoten [NBaum a]
 deriving(Eq,Show)
```

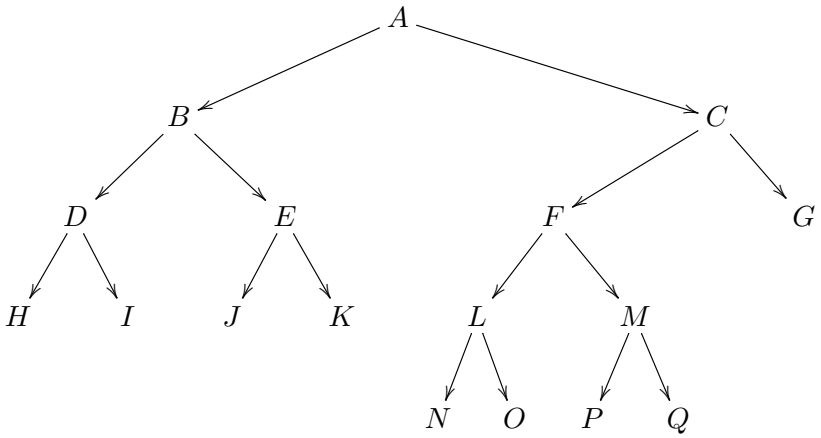
Ein Knoten erhält als Argument eine Liste seiner Unterbäume.

Übungsaufgabe 3.4.1. *Definiere Funktionen* `nRand`, `nFold`, `nMap` *für n-äre Bäume.*

Die bisher betrachteten Bäume hatten nur Markierungen an den Blättern, d.h. die inneren Knoten waren nicht markiert. Binäre Bäume mit Markierungen an allen Knoten kann man definieren durch:

```
data BinBaum a = BinBlatt a | BinKnoten a (BinBaum a) (BinBaum a)
 deriving(Eq,Show)
```

Der folgende Baum



kann als `BinBaum Char` wie folgt dargestellt werden:

```
beispielBinBaum =
 BinKnoten 'A'
  (BinKnoten 'B'
     (BinKnoten 'D' (BinBlatt 'H') (BinBlatt 'I'))
     (BinKnoten 'E' (BinBlatt 'J') (BinBlatt 'K'))
  )
  (BinKnoten 'C'
     (BinKnoten 'F'
        (BinKnoten 'L' (BinBlatt 'N') (BinBlatt 'O'))
        (BinKnoten 'M' (BinBlatt 'P') (BinBlatt 'Q'))
     )
     (BinBlatt 'G')
  )
```

Zum Beispiel kann man die Liste aller Knotenmarkierungen für solche Bäume in den Reihenfolgen pre-order (Wurzel, linker Teilbaum, rechter Teilbaum), in-order (linker Teilbaum, Wurzel, rechter Teilbaum), post-order (linker Teilbaum, rechter Teilbaum, Wurzel), level-order (Knoten stufenweise, wie bei Breitensuche) berechnen:

```
preorder :: BinBaum t -> [t]
preorder (BinBlatt a)      = [a]
preorder (BinKnoten a l r) = a:(preorder l) ++ (preorder r)

inorder (BinBlatt a) = [a]
inorder (BinKnoten a l r) = (inorder l) ++ a:(inorder r)

postorder (BinBlatt a) = [a]
postorder (BinKnoten a l r) =
   (postorder l) ++ (postorder r) ++ [a]
```

```
levelorderSchlecht b =
   concat [nodesAtDepthI i b | i <- [0..depth b]]
 where
  nodesAtDepthI 0 (BinBlatt a) = [a]
  nodesAtDepthI i (BinBlatt a) = []
  nodesAtDepthI 0 (BinKnoten a l r) = [a]
  nodesAtDepthI i (BinKnoten a l r) = (nodesAtDepthI (i-1) l)
                                      ++ (nodesAtDepthI (i-1) r)
  depth (BinBlatt _) = 0
  depth (BinKnoten _ l r) = 1+(max (depth l) (depth r))
```

Die Implementierung der level-order Reihenfolge ist eher schlecht, da der Baum für jede Stufe neu von der Wurzel beginnend durch gegangen wird. Besser ist die folgende Implementierung:

```
levelorder b = loForest [b]
 where
  loForest xs = map root xs
                  ++ concatMap (loForest . subtrees) xs
  root (BinBlatt a) = a
  root (BinKnoten a _ _) = a
  subtrees (BinBlatt _) = []
  subtrees (BinKnoten _ l r) = [l,r]
```

Für den Beispielbaum ergibt dies:

```
*Main> inorder beispielBinBaum
"HDIBJEKANLOFPMQCG"
*Main> preorder beispielBinBaum
"ABDHIEJKCFLNOMPQG"
*Main> postorder  beispielBinBaum
"HIDJKEBNOLPQMFGCA"
*Main> levelorderSchlecht  beispielBinBaum
"ABCDEFGHIJKLMNOPQ"
*Main> levelorder  beispielBinBaum
"ABCDEHIJKFGLMNOPQ"
```

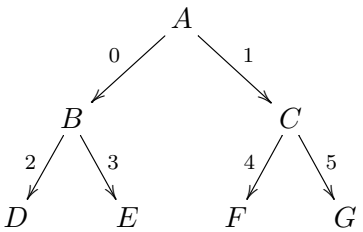
Übungsaufgabe 3.4.2. *Geben Sie einen Datentyp in Haskell für n-äre Bäume mit Markierung aller Knoten an. Implementieren Sie die Berechnung aller Knotenmarkierungen als Liste in pre-order, in-order, post-order und level-order Reihenfolge.*

Man kann auch Bäume definieren, die zusätzlich zur Beschriftung der Knoten auch Beschriftungen der Kanten haben. Hierbei ist es durchaus sinnvoll für die Kanten- und die Knotenbeschriftungen auch unterschiedliche Typen zu zulassen, indem man zwei verschiedene Typvariablen benutzt:

```
data BinBaumMitKM a b =
   BiBlatt a
 | BiKnoten a (b, BinBaumMitKM a b) (b,BinBaumMitKM a b)
 deriving(Eq,Show)
```

Beachte, dass man die Paare eigentlich nicht benötigt, man könnte auch `BiKnoten a b (BinBaumMitKM a b) b (BinBaumMitKM a b)` schreiben, was allerdings eher unübersichtlich ist.

Der Baum



kann als `BinBaumMitKM Char Int` dargestellt werden durch:

```
beispielBiBaum =
  BiKnoten 'A'
   (0,BiKnoten 'B' (2,BiBlatt 'D') (3,BiBlatt 'E'))
   (1,BiKnoten 'C' (4,BiBlatt 'F') (5,BiBlatt 'G'))
```

Man kann eine Funktion `biMap` implementieren, die zwei Funktionen erwartet und die erste Funktion auf die Knotenmarkierungen und die zweite Funktion auf die Kantenmarkierungen anwendet:

```
biMap f g (BiBlatt a) = (BiBlatt $ f a)
biMap f g (BiKnoten a (kl,links) (kr,rechts) =
 BiKnoten (f a) (g kl, biMap f g links) (g kr, biMap f g rechts)
```

Wir haben in der Definition von `biMap` den in Haskell vordefinierten Operator `$` verwendet. Dieser ist definiert wie eine Anwendung:

```
f $ x = f x
```

Der (eher Parse-) Trick dabei ist jedoch, dass die Bindungspräzedenz für `$` ganz niedrig ist, daher werden die Symbole nach dem `$` zunächst als syntaktische Einheit geparst, bevor der `$`-Operator angewendet wird. Der Effekt ist, dass man durch `$` Klammern sparen kann. Z.B. wird

```
map (*3) $ filter (>5) $ concat [[1,1],[2,5],[10,11]]
```

dadurch geklammert als

```
map (*3) (filter (>5) ( concat [[1,1],[2,5],[10,11]]))
```

Grob kann man sich merken, dass gilt: `f $ e` wird als `f (e)` geparst.

Ein Beispielaufruf für `biMap` ist

```
*Main Char> biMap toLower even beispielBiBaum
BiKnoten 'a'
 (True,BiKnoten 'b' (True,BiBlatt 'd') (False,BiBlatt 'e'))
 (False,BiKnoten 'c' (True,BiBlatt 'f') (False,BiBlatt 'g'))
```

3.4.1 Syntaxbäume

Ein Syntaxbaum ist im Allgemeinen die Ausgabe eines Parsers, d.h. er stellt syntaktisch korrekte Ausdrücke eine Sprache dar. In Haskell kann man Datentypen für Syntaxbäume ganz analog zu den bisher vorgestellten Bäumen definieren. Die Manipulation (z.B. die Auswertung) von Syntaxbäumen kann dann durch Funktionen auf diesen Datentypen bewerkstelligt werden. Durch Haskells Pattern-matching sind solche Funktionen recht einfach zu implementieren.

Wir betrachten als einfaches Beispiel zunächst arithmetische Ausdrücke, die aus Zahlen, der Addition und der Multiplikation bestehen. Man kann deren Syntax z.B. durch die folgende kontextfreie Grammatik angeben (das Startsymbol ist dabei E):

$$\begin{array}{lcl} E & ::= & (E + E) \mid (E * E) \mid Z \\ Z & ::= & 0Z' \mid \ldots \mid 9Z' \\ Z' & ::= & \varepsilon \mid Z \end{array}$$

Als Haskell-Datentyp für Syntaxbäume für solche arithmetischen Ausdrücke kann man z.B. definieren:

```
data ArEx = Plus ArEx ArEx
          | Mult ArEx ArEx
          | Zahl Int
```

Haskell bietet auch die Möglichkeit infix-Konstruktoren zu definieren, diese müssen stets mit einem `:` beginnen. Damit kann man eine noch besser lesbare Datentyp definition angeben:

```
data ArEx = ArEx :+: ArEx
          | ArEx :*: ArEx
          | Zahl Int
```

Anstelle der Präfix-Konstruktoren `Plus` und `Mult` verwendet diese Variante die Infix-Konstruktoren `:+:` und `:*:`. Der arithmetische Ausdruck $(3 + 4) * (5 + (6 + 7))$ wird dann als Objekt vom Typ `ArEx` dargestellt durch: `((Zahl 3) :+: (Zahl 4)) :*: ((Zahl 5) :+: ((Zahl 6) :+: (Zahl 7)))`.

Man kann nun z.B. einen Interpreter für arithmetische Ausdrücke sehr leicht implementieren, indem man die einzelnen Fälle (d.h. verschiedenen Konstruktoren) durch Patter-Matching abarbeitet:

```
interpretArEx :: ArEx -> Int
interpretArEx (Zahl i) = i
interpretArEx (e1 :+: e2) =
            (interpretArEx e1) + (interpretArEx e2)
interpretArEx (e1 :*: e2) =
            (interpretArEx e1) * (interpretArEx e2)
```

Z.B. ergibt:

```
*> interpretArEx
    ((Zahl 3) :+: (Zahl 4)) :*: ((Zahl 5) :+: ((Zahl 6)
             :+: (Zahl 7)))
126
```

Als komplexeres Beispiel betrachten wir den ungetypten Lambda-Kalkül (siehe Kapitel 2). Als Datentyp für Lambda-Kalkül-Ausdrücke könnte man definieren:

```
data LExp v =
   Var v                  -- x
 | Lambda v (LExp v)      -- \x.s
 | App (LExp v) (LExp v)  -- (s t)
 deriving(Eq,Show)
```

Dieser Datentyp ist polymorph über den Namen der Variablen, wir werden im Folgenden Strings hierfür verwenden. Z.B. kann man den Ausdruck $s = (\lambda x.x)\ (\lambda y.y)$ als Objekt vom Typ `LExp String` darstellen als:

```
s :: LExp String
s =  App (Lambda "x" (Var "x")) (Lambda "y" (Var "y"))
```

Wir erläutern im Folgenden, wie man einen Interpreter für den Lambda-Kalkül einfach implementieren kann, der in Normalordnung auswertet. Im Grunde muss man hierfür die β-Reduktion implementieren und diese an der richtigen Stelle durchführen. Wir beginnen jedoch zunächst mit zwei Hilfsfunktionen. Die Funktion `rename` führt die Umbenennung eines Ausdrucks mit frischen Variablennamen durch. D.h. sie erwartet einen Ausdruck und eine Liste von neuen Namen (eine Liste von Strings) und liefert den umbenannten Ausdruck und die noch nicht verbrauchten Namen aus der Liste:

```
rename :: (Eq b) => LExp b -> [b] -> (LExp b, [b])
rename expr freshvars = rename_it expr [] freshvars
 where rename_it (Var v) renamings freshvars =
           case lookup v renamings of
             Nothing -> (Var v,freshvars)
             Just v'  -> (Var v',freshvars)
       rename_it (App e1 e2) renamings freshvars =
           let (e1',vars') = rename_it e1 renamings freshvars
               (e2',vars'') = rename_it e2 renamings vars'
           in (App e1' e2', vars'')
       rename_it (Lambda v e) renamings (f:freshvars) =
           let (e',vars') = rename_it e ((v,f):renamings) freshvars
           in (Lambda f e',vars')
```

Die lokal definierte Funktion `rename_it` führt dabei als weiteres Argument die Liste `renamings` mit. In diese Liste wird an jedem Lambda-Binder die Substitution (alter Name, neuer Name) eingefügt. Erreicht man ein Vorkommen der Variable, so wird in dieser Liste nach der richtigen Substitution gesucht.

Als zweite Hilfsfunktion definieren wir die Funktion `substitute`. Diese führt die Substitution $s[t/x]$ durch, wie sie später bei der β-Reduktion benötigt wird. Um die DVC einzuhalten, wird jedes eingesetzte t dabei mit `rename` frisch umbenannt. Deshalb erwartet auch die Funktion `substitute` eine Liste frischer Variablennamen und liefert als Ergebnis zusätzlich die nicht benutzten Namen.

```
-- substitute freshvars expr1 expr2 var
-- f"uhrt die Ersetzung expr1[expr2/var] durch

substitute  :: (Eq b) => [b] -> LExp b -> LExp b ->
                b -> (LExp b, [b])

substitute freshvars (Var v) expr2 var
 | v == var =  rename (expr2) freshvars
 | otherwise = (Var v,freshvars)
```

```
substitute freshvars (App e1 e2) expr2 var =
 let (e1',vars')  = substitute freshvars e1 expr2 var
     (e2',vars'') = substitute vars' e2 expr2 var
 in (App e1' e2', vars'')

substitute freshvars (Lambda v e) expr2 var =
 let (e',vars')   = substitute freshvars e expr2 var
 in (Lambda v e',vars')
```

Hierbei ist zu beachten, dass die Implementierung davon ausgeht, dass alle Ausdrücke der Eingabe die DVC erfüllen (sonst wäre die Implementierung für den Fall `Lambda v e` aufwändiger).

Nun implementieren wir die Ein-Schritt-Normalordnungsreduktion in Form der Funktion `tryBeta`. Diese erwartet einen Ausdruck und eine Liste frischer Variablennamen. Anschließend sucht sie nach einem Normalordnungsredex, indem sie in Anwendungen in das linke Argument rekursiv absteigt. Sobald sie einen Redex findet, führt sie die β-Reduktion durch. Da der Ausdruck auch irreduzibel sein kann, verpacken wir das Ergebnis mit dem `Maybe`-Typen: Die Funktion liefert `Just ...`, wenn eine Reduktion durchgeführt wurde, und liefert ansonsten `Nothing`.

```
-- Einfachster Fall: Beta-Reduktion ist auf Top-Level m"oglich:
tryNOBeta (App (Lambda v e) e2) freshvars =
 let (e',vars) = substitute freshvars e e2 v
 in Just (e',vars)

-- Andere Anwendungen: gehe links ins Argument (rekursiv):
tryNOBeta (App e1 e2) freshvars =
 case tryNOBeta e1 freshvars of
  Nothing -> Nothing
  Just (e1',vars) -> (Just ((App e1' e2), vars))

-- Andere Faelle: Keine Reduktion m"oglich:
tryNOBeta _ vars = Nothing
```

Schließlich kann man die Normalordnungsauswertung implementieren, indem man `tryNOBeta` solange anwendet bis keine Reduktion mehr möglich ist:

```
tryNO e vars = case tryNOBeta e vars of
                  Nothing -> e
                  Just (e',vars') -> tryNO e' vars'
```

Die Hauptfunktion ruft nun `tryNO` mit einer Liste von frischen Variablennamen auf und benennt zur Sicherheit die Eingabe vor der ersten Reduktion um:

```
reduceNO e = let (e',v') = rename e fresh
             in  tryNO e' v'
      where fresh = ["x_" ++ show i | i <- [1..]]
```

Einige Beispielauswertungen:

```
example_id  = (Lambda "v" (Var "v"))
example_k   = (Lambda "x" (Lambda "y" (Var "x")))

*Main> reduceNO example_id
Lambda "x_1" (Var "x_1")
*Main> reduceNO (App example_id example_id)
Lambda "x_3" (Var "x_3")
*Main> reduceNO (App example_k example_id)
Lambda "x_2" (Lambda "x_4" (Var "x_4"))
```

3.5 Typdefinitionen: data, type, newtype

In Haskell gibt es drei Konstrukte um neue (Daten-)Typen zu definieren. Die allgemeinste Methode ist die Verwendung der `data`-Anweisung, für die wir schon einige Beispiele gesehen haben. Mit der `type`-Anweisung kann man *Typsynonyme* definieren, d.h. man gibt bekannten Typen (auch zusammengesetzten) einen neuen Namen. Z.B. kann man definieren

```
type IntCharPaar = (Int,Char)

type Studenten = [Student]

type MyList a = [a]
```

Der Sinn dabei ist, dass Typen von Funktionen dann diese Typsynonyme verwenden können und daher leicht verständlicher sind. Z.B.

```
alleStudentenMitA :: Studenten -> Studenten
alleStudentenMitA = map nachnameMitA
```

Das `newtype`-Konstrukt ist eine Mischung aus `data` und `type`: Es wird ein Typsynonym erzeugt, dass einen zusätzlichen Konstruktor erhält, z.B.

```
newtype Studenten' = St [Student]
```

Diese Definition ist aus Benutzersicht völlig äquivalent zu:

```
data Studenten' = St [Student]
```

Der Haskell-Compiler kann jedoch mit `newtype` definierte Datentypen effizienter behandeln, als mit `data` definierte Typen, da er „weiß“, dass es sich im Grunde nur um ein verpacktes Typsynonym handelt. Eine andere Frage ist, warum man `newtype` anstelle von `type` verwenden sollte. Dies liegt am Typklassensystem von Haskell (das sehen wir später genauer): Für mit `type` definierte Typsynonyme können keine speziellen Instanzen für Typklassen erstellt werden, bei `newtype` (genau wie bei `data`) jedoch schon. Z.B. kann man keine eigene `show`-Funktion für `Studenten` definieren, für `Studenten'` jedoch schon. Der Unterschied ist auch, dass case und pattern match für Objekte vom `newtype`-definierten Typ immer erfolgreich sind.

3.6 Haskells hierarchisches Modulsystem

Module dienen zur

Strukturierung / Hierarchisierung: Einzelne Programmteile können innerhalb verschiedener Module definiert werden; eine (z. B. inhaltliche) Unterteilung des gesamten Programms ist somit möglich. Hierarchisierung ist möglich, indem kleinere Programmteile mittels Modulimport zu größeren Programmen zusammen gesetzt werden.

Kapselung: Nur über Schnittstellen kann auf bestimmte Funktionalitäten zugegriffen werden, die Implementierung bleibt verdeckt. Sie kann somit unabhängig von anderen Programmteilen geändert werden, solange die Funktionalität (bzgl. einer vorher festgelegten Spezifikation) erhalten bleibt.

Wiederverwendbarkeit: Ein Modul kann für verschiedene Programme benutzt (d.h. importiert) werden.

3.6.1 Moduldefinitionen in Haskell

In einem Modul werden Funktionen, Datentypen, Typsynonyme, usw. definiert. Durch die Moduldefinition können diese Konstrukte exportiert werden, die dann von anderen Modulen importiert werden können.

Ein Modul wird mittels

$$\texttt{module}\ \mathit{Modulname}(\mathit{Exportliste})\ \texttt{where}\ \left.\begin{array}{l}\mathit{Modulimporte},\\ \mathit{Datentypdefinitionen},\\ \mathit{Funktionsdefinitionen},\ldots\end{array}\right\}\ \mathit{Modulrumpf}$$

definiert. Hierbei ist `module` das Schlüsselwort zur Moduldefinition, *Modulname* der Name des Moduls, der mit einem Großbuchstaben anfangen muss. In der *Exportliste* werden diejenigen Funktionen, Datentypen usw. definiert, die durch das Modul exportiert werden, d.h. von außen sichtbar sind.

Für jedes Modul muss eine separate Datei angelegt werden, wobei der Modulname dem Dateinamen ohne Dateiendung entsprechen muss[7].

Ein Haskell-Programm besteht aus einer Menge von Modulen, wobei eines der Module ausgezeichnet ist, es muss laut Konvention den Namen `Main` haben und eine Funktion namens `main` definieren und exportieren. Der Typ von `main` ist auch per Konvention festgelegt, er muss `IO ()` sein, d.h. eine Ein-/Ausgabe-Aktion, die nichts (dieses „Nichts“ wird durch das Nulltupel `()` dargestellt) zurück liefert. Der Wert des Programms ist dann der Wert, der durch `main` definiert wird. Das Grundgerüst eines Haskell-Programms ist somit von der Form:

```
module Main(main) where
 ...
 main = ...
 ...
```

Im Folgenden werden wir den Modulexport und -import anhand folgendes Beispiels verdeutlichen:

[7] Bei hierarchischen Modulen muss der Dateipfad dem Modulnamen entsprechen, siehe Abschnitt 3.6.4)

Beispiel 3.6.1. *Das Modul* `Spiel` *sei definiert als:*

```
module Spiel where
  data Ergebnis = Sieg | Niederlage | Unentschieden
  berechneErgebnis a b = if a > b then Sieg
                         else if a < b then Niederlage
                              else Unentschieden
  istSieg Sieg = True
  istSieg _    = False
  istNiederlage Niederlage = True
  istNiederlage _          = False
```

3.6.2 Modulexport

Durch die *Exportliste* bei der Moduldefinition kann festgelegt werden, was exportiert wird. Wird die Exportliste einschließlich der Klammern weggelassen, so werden alle definierten, bis auf von anderen Modulen importierte, Namen exportiert. Für Beispiel 3.6.1 bedeutet dies, dass sowohl die Funktionen `berechneErgebnis`, `istSieg`, `istNiederlage` als auch der Datentyp `Ergebnis` samt aller seiner Konstruktoren `Sieg`, `Niederlage` und `Unentschieden` exportiert werden. Die Exportliste kann folgende Einträge enthalten:

- Ein Funktionsname, der im Modulrumpf definiert oder von einem anderem Modul importiert wird. Operatoren, wie z.B. + müssen in der Präfixnotation, d.h. geklammert `(+)` in die Exportliste eingetragen werden.

  Würde in Beispiel 3.6.1 der Modulkopf

  ```
  module Spiel(berechneErgebnis) where
  ```

  lauten, so würde nur die Funktion `berechneErgebnis` durch das Modul `Spiel` exportiert.

- Datentypen die mittels `data` oder `newtype` definiert wurden. Hierbei gibt es drei unterschiedliche Möglichkeiten, die wir anhand des Beispiels 3.6.1 zeigen:
  - Wird nur `Ergebnis` in die Exportliste eingetragen, d.h. der Modulkopf würde lauten

```
module Spiel(Ergebnis) where
```

so wird der Typ `Ergebnis` exportiert, nicht jedoch die Datenkonstruktoren, d.h. `Sieg`, `Niederlage`, `Unentschieden` sind von außen nicht sichtbar bzw. verwendbar.

- Lautet der Modulkopf

```
module Spiel(Ergebnis(Sieg, Niederlage))
```

so werden der Typ `Ergebnis` und die Konstruktoren `Sieg` und `Niederlage` exportiert, nicht jedoch der Konstruktor `Unentschieden`.

- Durch den Eintrag `Ergebnis(..)`, wird der Typ mit sämtlichen Konstruktoren exportiert.

- Typsynonyme, die mit `type` definiert wurden, können exportiert werden, indem sie in die Exportliste eingetragen werden, z.B. würde bei folgender Moduldeklaration

```
module Spiel(Result) where
  ... wie vorher ...
  type Result = Ergebnis
```

der mittels `type` erzeugte Typ `Result` exportiert.

- Schließlich können auch alle exportierten Namen eines importierten Moduls wiederum durch das Modul exportiert werden, indem man `module` *Modulname* in die Exportliste aufnimmt, z.B. seien das Modul `Spiel` wie in Beispiel 3.6.1 definiert und das Modul `Game` als:

```
module Game(module Spiel, Result) where
  import Spiel
  type Result = Ergebnis
```

Das Modul `Game` exportiert alle Funktionen, Datentypen und Konstruktoren, die auch `Spiel` exportiert sowie zusätzlich noch den Typ `Result`.

3.6.3 Modulimport

Die exportierten Definitionen eines Moduls können mittels der `import` Anweisung in ein anderes Modul importiert werden. Diese steht am Anfang des Modulrumpfs. In einfacher Form geschieht dies durch

`import` *Modulname*

Durch diese Anweisung werden sämtliche Einträge der Exportliste vom Modul mit dem Namen *Modulname* importiert, d.h. sichtbar und verwendbar.

Will man nicht alle exportierten Namen in ein anderes Modul importieren, so ist dies auf folgende Weisen möglich:

Explizites Auflisten der zu importierenden Einträge: Die importierten Namen werden in Klammern geschrieben aufgelistet. Die Einträge werden hier genauso geschrieben wie in der Exportliste.

Z.B. importiert das Modul

```
module Game where
  import Spiel(berechneErgebnis, Ergebnis(..))
  ...
```

nur die Funktion `berechneErgebnis` und den Datentyp `Ergebnis` mit seinen Konstruktoren, nicht jedoch die Funktionen `istSieg` und `istNiederlage`.

Explizites Ausschließen einzelner Einträge: Einträge können vom Import ausgeschlossen werden, indem man das Schlüsselwort `hiding` gefolgt von einer Liste der ausgeschlossenen Einträge benutzt.

Den gleichen Effekt wie beim expliziten Auflisten können wir auch im Beispiel durch Ausschließen der Funktionen `istSieg` und `istNiederlage` erzielen:

```
module Game where
  import Spiel hiding(istSieg,istNiederlage)
  ...
```

Die importierten Funktionen sind sowohl mit ihrem (unqualifizierten) Namen ansprechbar, als auch mit ihrem qualifizierten Namen: *Modulname*.*unqualifizierter Name*, manchmal ist es notwendig den qualifizierten Namen zu verwenden, z.B.

```
module A(f) where
  f a b = a + b

module B(f) where
  f a b = a * b

module C where
  import A
  import B
  g = f 1 2 + f 3 4 -- funktioniert nicht
```

führt zu einem Namenskonflikt, da `f` mehrfach (in Modul `A` und `B`) definiert wird.

```
Prelude> :l C.hs

ERROR C.hs:4 - Ambiguous variable occurrence "f"
*** Could refer to: B.f A.f
```

Werden qualifizierte Namen benutzt, wird die Definition von g eindeutig:

```
module C where
  import A
  import B
  g = A.f 1 2 + B.f 3 4
```

Durch das Schlüsselwort `qualified` sind nur die qualifizierten Namen sichtbar:

```
module C where
  import qualified A
  g = f 1 2   -- f ist nicht sichtbar

Prelude> :l C.hs

ERROR C.hs:3 - Undefined variable "f"
```

Man kann auch *lokale Aliase* für die zu importierenden Modulnamen angeben, hierfür gibt es das Schlüsselwort `as`, z.B.

```
import LangerModulName as C
```

Eine durch `LangerModulName` exportierte Funktion `f` kann dann mit `C.f` aufgerufen werden.

Abschließend eine Übersicht: Angenommen das Modul `M` exportiert `f` und `g`, dann zeigt die folgende Tabelle, welche Namen durch die angegebene `import`-Anweisung sichtbar sind:

Import-Deklaration	definierte Namen
`import M`	`f, g, M.f, M.g`
`import M()`	keine
`import M(f)`	`f, M.f`
`import qualified M`	`M.f, M.g`
`import qualified M()`	keine
`import qualified M(f)`	`M.f`
`import M hiding ()`	`f, g, M.f, M.g`
`import M hiding (f)`	`g, M.g`
`import qualified M hiding ()`	`M.f, M.g`
`import qualified M hiding (f)`	`M.g`
`import M as N`	`f, g, N.f, N.g`
`import M as N(f)`	`f, N.f`
`import qualified M as N`	`N.f, N.g`

3.6.4 Hierarchische Modulstruktur

Die hierarchische Modulstruktur erlaubt es, Modulnamen mit Punkten zu versehen. So kann z.B. ein Modul `A.B.C` definiert werden. Allerdings ist dies eine rein syntaktische Erweiterung des Namens und es besteht nicht notwendigerweise eine Verbindung zwischen einem Modul mit dem Namen `A.B` und `A.B.C`.

Die Verwendung dieser Syntax hat lediglich Auswirkungen wie der Interpreter nach der zu importierenden Datei im Dateisystem sucht: Wird `import A.B.C` ausgeführt, so wird das Modul `A/B/C.hs` geladen, wobei `A` und `B` Verzeichnisse sind.

Die „Haskell Hierarchical Libraries[8]“ sind mithilfe der hierarchischen Modulstruktur aufgebaut, z.B. sind Funktionen, die auf Listen operieren, im Modul `Data.List` definiert.

[8] siehe `http://www.haskell.org/ghc/docs/latest/html/libraries`

3.7 Haskells Typklassensystem

Haskells Typklassensystem dient zur Implementierung von so genanntem *ad hoc* Polymorphismus. Wir grenzen die Begriffe ad hoc Polymorphismus und parametrischer Polymorphismus voneinander ab:

Ad hoc Polymorphismus: Ad hoc Polymorphismus tritt auf, wenn eine Funktion (bzw. Funktionsname) mehrfach für verschiedene Typen definiert ist, wobei sie sich die Implementierungen für verschiedene Typen völlig anders verhalten können, also auch i.a. anders definiert sind. Die Stelligkeit ist dabei in der Regel unabhängig von den Argumenttypen. Ein Beispiel ist der Additionsoperator `+`, der z.B. für `Integer`- aber auch für `Double`-Werte implementiert ist und verwendet werden kann. Besser bekannt ist ad hoc-Polymorphismus als *Überladung*.

Parametrischer Polymorphismus: Parametrischer Polymorphismus tritt auf, wenn eine Funktion für eine Menge von verschiedenen Typen definiert ist, aber sich für alle Typen gleich verhält, also die Implementierung vom konkreten Typ abstrahiert. Ein Beispiel für parametrischen Polymorphismus ist die Funktion `++`, da sie für alle Listen (egal welchen Inhaltstyps) definiert ist und verwendet werden kann.

Typklassen in Haskell dienen dazu, einen oder mehrere überladene Operatoren zu definieren (und zusammenzufassen). Im Grunde wird in der Typklasse nur der Typ der Operatoren festgelegt, aber es sind auch default-Implementierungen möglich. Eine *Typklassendefinition* beginnt mit

```
class [OBERKLASSE =>] Klassenname a where
  ...
```

Hierbei ist `a` eine Typvariable, die für die Typen steht. Beachte, dass nur eine solche Variable erlaubt ist (es gibt Erweiterungen die mehrere Variablen erlauben). `OBERKLASSE` ist eine Klassenbedingung, sie kann auch leer sein, der Pfeil `=>` entfällt dann. Im Rumpf der Klassendefinition stehen die Typdeklarationen für die Klassendefinitionen und optional default-Implementierungen für die Operationen. Beachte, dass die Zeilen des Rumpfs eingerückt sein müssen. Wir betrachten die vordefinierte Klasse `Eq`, die den Gleichheitstest `==` (und den Ungleichheitstest `/=`) überlädt:

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

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

Diese Klasse hat keine Klassenbedingung (`OBERKLASSE` ist leer) und definiert zwei Operatoren == und /=, die beide den Typ `a -> a -> Bool` haben (beachte: a ist durch den Kopf der Klassendefinition definiert). Für beide Operatoren (oder auch *Klassenmethoden*) sind default-Implementierungen angegeben. Das bedeutet, für die Angabe einer Instanz genügt es eine der beiden Operationen zu implementieren (man darf aber auch beide angeben). Die nicht verwendete default-Implementierung wird dann durch die Instanz *überschrieben*.

In Haskell kann man einer `data`-Deklaration das Schlüsselwort `deriving` gefolgt von einer Liste von Typklassen nachstellen. Dadurch generiert der Compiler automatisch Typklasseninstanzen des Datentyps. Typischerweise benutzt man das für `deriving(Show,Eq)` (s.u.), damit man die Objekte drucken kann und mit == auch vergleichen kann.

Für die Typisierung unter Typklassen (die wir nicht genau betrachten werden) muss man die Syntax von Typen erweitern: Bisher wurden polymorphe Typen entsprechend der Syntax

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1 \to \mathbf{T}_2$$

gebildet. Zur Abbildung der Typklassenbedingungen könnte man erweiterte Typen $\mathbf{T_e}$ wie folgt definieren:

$$\mathbf{T_e} ::= \mathbf{T} \mid \mathbf{Kon}\ \texttt{=>}\ \mathbf{T}$$

wobei $\mathbf{T}$ ein polymorpher Typ (ohne Typklassen) ist und $\mathbf{Kon}$ ein sogenannter Typklassenkontext ist, den man nach der folgenden Grammatik bilden kann

$$\mathbf{Kon} ::= \text{Klassenname}\ TV \mid (\text{Klassenname}_1\ TV, \ldots, \text{Klassenname}_n\ TV)$$

D.h. der Klassenkontext besteht aus einer oder mehreren Typklassenbeschränkungen für Typvariablen.

Für einen Typ der Form *Kontext* `=>` *Typ* muss dabei zusätzlich gelten, dass alle Typvariablen des Kontexts auch als Typvariablen im Typ *Typ* vorkommen.

Z.B. hat die Funktion `elem`, die testet ob ein bestimmtes Element in einer Liste enthalten ist, den erweiterten Typ `(Eq a) => a -> [a] -> Bool`. Die richtige Interpretation dafür ist: `elem` kann auf allen Listen verwendet werden, für deren Elementtyp der Gleichheitstest (also eine `Eq`-Instanz) definiert ist. Die Beschränkung rührt daher, dass innerhalb der Definition von `elem` der Gleichheitstest == verwendet wird.

Wir betrachten nun, wie man Typklasseninstanzen definiert. Das Schema hierfür ist:

```
instance [KLASSENBEDINGUNGEN => ] KLASSENINSTANZ where
  ...
```

Hierbei können mehrere oder keine Klassenbedingungen angegeben werden. Erlaubt für KLASSENINSTANZ sind nur: Tykonstruktoren, oder ein Typkonstruktor angewendet auf verschiedene Typ-Variablen. Die Klasseninstanz ist gerade der Typ der Klasse nach einsetzen des Instanztyps für den Parameter `a`, d.h. z.B. für den Typ `Int` und die Klasse `Eq` ist die Klasseninstanz `Eq Int`. Im Rumpf der Instanzdefinition müssen die Klassenmethoden implementiert werden (außer jenen, die von default-Implementierungen abgedeckt sind). Die `Eq`-Instanz für `Int` ist definiert als:

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

Hierbei ist `primEQInt` eine primitive eingebaute Funktion, die wir nicht genauer betrachten. Wir können z.B. eine `Eq`-Instanz für den Wochentag-Typ angeben

```
instance Eq Wochentag where
  Montag     == Montag     = True
  Dienstag   == Dienstag   = True
  Mittwoch   == Mittwoch   = True
  Donnerstag == Donnerstag = True
  Freitag    == Freitag    = True
  Samstag    == Samstag    = True
  Sonntag    == Sonntag    = True
  _          == _          = False
```

Anschließend kann man == und /= auf Wochentagen verwenden, z.B.:

```
*Main> Montag == Montag
True
*Main> Montag /= Dienstag
True
*Main> Montag == Dienstag
False
```

3.7.1 Vererbung und Mehrfachvererbung

Bei Klassendefinitionen können eine oder mehrere Oberklassen angegeben werden. Die Syntax ist

```
class (Oberklasse1 a, ..., OberklasseN a) => Klassenname a where
  ...
```

Die Schreibweise ist etwas ungewöhnlich, da man => nicht als logische Implikation auffassen darf. Vielmehr ist die Semantik: Ein Typ kann nur dann Instanz der Klasse `Klassenname` sein, wenn für ihn bereits Instanzen des Typs für die Klassen `Oberklassen1`, ..., `OberklassenN` definiert sind. Dadurch darf man überladene Funktionen der Oberklassen in der Klassendefinition für die Unterklasse verwenden. Beachte, dass auch Mehrfachvererbung erlaubt ist, da mehrere Oberklassen möglich sind.

Ein Beispiel für eine Klasse mit Vererbung ist die Klasse `Ord`, die Vergleichsoperationen zur Verfügung stellt. `Ord` ist eine Unterklasse von `Eq`, daher darf der Ungleichheitstest und Gleichheitstest in der Definition der default-Implementierungen verwendet werden. Die Definition von `Ord` ist:

```
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a

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

    x <= y  = compare x y /= GT
    x <  y  = compare x y == LT
    x >= y  = compare x y /= LT
    x >  y  = compare x y == GT

    -- Note that (min x y, max x y) = (x,y) or (y,x)
    max x y | x <= y    =  y
            | otherwise =  x
    min x y | x <= y    =  x
            | otherwise =  y
```

Instanzen müssen entweder <= oder die Funktion `compare` definieren. Der Rückgabewert von `compare` ist vom Typ `Ordering`, der ein Aufzählungstyp ist, und definiert ist als:

```
data Ordering = LT | EQ | GT
```

Hierbei steht `LT` für „lower than“ (kleiner), `EQ` für „equal“ (gleich) und `GT` für „greater than“ (größer).

Wir können z.B. eine Instanz für den Wochentag-Typ angeben:

```
instance Ord Wochentag where
 a <= b =
   (a,b) `elem`  [(a,b) | i <- [0..6],
                          let a = ys!!i,
                          b <- drop i ys]
  where ys = [Montag, Dienstag, Mittwoch,
              Donnerstag, Freitag, Samstag, Sonntag]
```

Wir betrachten die folgende Funktion

```
f x y = (x == y) && (x <= y)
```

Da `f` in der Definition sowohl == als auch <= verwendet, würde man als Typ für `f` erwarten:

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

Durch die Vererbung (da `Ord` Unterklasse von `Eq` ist) genügt jedoch der Typ

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

da die `Eq`-Instanz dadurch automatisch gefordert wird.

3.7.2 Klassenbeschränkungen bei Instanzen

Auch bei Instanzdefinitionen kann man Voraussetzungen angeben. Diese dienen jedoch *nicht zur Vererbung*, sondern dafür Instanzen für rekursive polymorphe Datentypen definieren zu können. Wollen wir z.B. eine Instanzdefinition des Typs `BBaum a` für die Klasse `Eq` angeben, so ist dies nur sinnvoll für solche Blattmarkierungstypen `a` für die selbst schon der Gleichheitstest definiert ist. Allgemein ist die Syntax

```
instance (Klasse1 a1, ..., KlasseN aN) => Klasse Typ where
  ...
```

Hierbei müssen die Typvariablen `a1`, ..., `aN` alle im polymorphen Typ `Typ` vorkommen, und dieser muss von der Form (Typkonstruktor Typvar_1 ... Typvar_n) sein. Die `Eq`-Instanz für Bäume kann man damit definieren als:

```
instance Eq a => Eq (BBaum a) where
 Blatt a == Blatt b           = a == b
 Knoten l1 r1 == Knoten l2 r2 = l1 == l2 && r1 == r2
 _ == _                       = False
```

Die Beschränkung `Eq a` besagt daher, dass der Gleichheitstest auf Bäumen nur dann definiert ist, wenn der Gleichheitstest auf den Blattmarkierungen definiert ist.

Einige Beispielaufrufe:

```
*Main List> Blatt 1 == Blatt 2
False
*Main List> Blatt 1 == Blatt 1
True
*Main List> Blatt (\x ->x) == Blatt (\y -> y)

<interactive>:1:0:
    No instance for (Eq (t -> t))
      arising from a use of ‘==’ at <interactive>:1:0-32
    Possible fix: add an instance declaration for (Eq (t -> t))
    In the expression: Blatt (\ x -> x) == Blatt (\ y -> y)
    In the definition of ‘it’:
        it = Blatt (\ x -> x) == Blatt (\ y -> y)
```

Beachte, dass der letzte Ausdruck nicht typisierbar ist, da die Klasseninstanz für `BBaum` nicht greift, da für den Funktionstyp `a -> a` keine `Eq`-Instanz definiert ist.

Übungsaufgabe 3.7.1. *Warum ist es nicht sinnvoll eine* `Eq`*-Instanz für den Typ* `a -> a` *zu definieren?*

Es ist durchaus möglich, mehrere Einschränkungen für verschiedene Typvariablen zu benötigen. Betrachte als einfaches Beispiel den Datentyp `Either`, der definiert ist als:

```
data Either a b = Left a | Right b
```

Dieser Typ ist ein Summentyp, er ermöglicht es zwei völlig unterschiedliche Typen in einem Typ zu verpacken. Z.B. kann man damit eine Liste von `Integer`- und `Char`-Werten darstellen:

```
[Left 'A',Right 0,Left 'B',Left 'C',Right 10,Right 12]
*Main List> :t it
it :: [Either Char Integer]
```

Will man eine sinnvolle `Eq`-Instanz für `Either` implementieren, so benötigt man als Voraussetzung, das beide Argumenttypen bereits `Eq`-Instanzen sind:

```
instance (Eq a, Eq b) => Eq (Either a b) where
 Left x  == Left y  = x == y -- benutzt Eq-Instanz f\"ur a
 Right x == Right y = x == y -- benutzt Eq-Instanz f\"ur b
 _       == _       = False
```

Eine analoge Problemstellung ergibt sich bei der `Eq`-Instanz für `BinBaumMitKM`.

Übungsaufgabe 3.7.2. *Definiere eine* `Eq`*-Instanz für* `BinBaumMitKM`*, wobei Bäume nur dann gleich sind, wenn ihre Knoten- und Kantenmarkierungen identisch sind.*

3.7.3 Die `Read`- und `Show`-Klassen

Die Klassen `Read` und `Show` dienen dazu, Typen in Strings zum Anzeigen zu konvertieren und umgekehrt Strings in Typen zu konvertieren. Die einfachsten Funktionen hierfür sind:

```
show :: Show a => a -> String
read :: Read a => String -> a
```

Die Funktion `show` ist tatsächlich eine Klassenmethode der Klasse `Show`, die Funktion `read` ist in der Prelude definiert, aber keine Klassenmethode der Klasse `read`, aber sie benutzt die Klassenmethoden. Es sollte stets gelten `read (show a) = a`.

Für die Klassen `Read` und `Show` werden zunächst zwei Typsynonyme definiert:

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

Der Typ `ReadS` stellt gerade den Typ eines *Parsers* dar: Als Eingabe erwartet er einen String und als Ausgabe liefert eine Erfolgsliste: Eine Liste von Paaren, wobei jedes Paar aufgebaut ist als

(erfolgreich geparster Ausdruck, Reststring).

Der Typ `ShowS` stellt eine Funktion dar, die einen String als Eingabe erhält und einen String als Ergebnis liefert. Die Idee dabei ist, Funktionen zu definieren, die einen String mit dem nächsten verbinden.

Die Funktionen `reads` und `shows` sind genau hierfür gedacht:

```
reads :: Read a => ReadS a
shows :: Show a => a -> ShowS
```

In den Typklassen sind diese Funktionen noch allgemeiner definiert als `readsPrec` und `showsPrec`, die noch eine Zahl als zusätzliches Argument erwarten, welche eine Präzedenz angibt. Solange man keine infix-Operatoren verwendet, kann man dieses Argument vernachlässigen. Die Klassen sind definiert als

```
class  Read a  where
    readsPrec :: Int -> ReadS a
    readList  :: ReadS [a]
    -- ... default decl for readList given in Prelude

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

    showsPrec _ x s   = show x ++ s
    show x            = showsPrec 0 x ""
    -- ... default decl for showList given in Prelude
```

Es ist oft sinnvoller `showsPrec` anstelle von `show` zu definieren, da man bei der Verwendung von `show` oft `++` verwendet, welches lineare Laufzeit im ersten Argument benötigt. Bei Verwendung von `showsPrec` kann man dies vermeiden.

Betrachte als Beispiel den Datentyp `BBaum`. Wir könnten eine `show` Funktion definieren durch:

```
showBBaum :: (Show t) => BBaum t -> String
showBBaum (Blatt a)    = show a
showBBaum (Knoten l r) =
   "<" ++ showBBaum l ++ "|" ++ showBBaum r ++ ">"
```

Allerdings kann diese Funktion u.U. quadratische Laufzeit haben, da `++` lineare Zeit im ersten Argument hat. Verwenden wir `shows` und die Funktionskomposition können wir dadurch quasi umklammern[9]:

[9] Die Funktion `showChar` ist bereits vordefiniert, sie ist vom Typ `Char -> ShowS`

```
showBBaum' :: (Show t) => BBaum t -> String
showBBaum' b = showsBBaum b []

showsBBaum :: (Show t) => BBaum t -> ShowS
showsBBaum (Blatt a)    = shows a
showsBBaum (Knoten l r) =
  showChar '<' . showsBBaum l . showChar '|'
    . showsBBaum r . showChar '>'
```

Beachte, dass die Funktionskomposition rechts-geklammert wird. Dadurch erzielen wir den Effekt, dass das Anzeigen lineare Laufzeit benötigt. Verwendet man hinreichend große Bäume, so lässt sich dieser Effekt im Interpreter messen:

```
*Main> last $ showBBaum t
'>'
(73.38 secs, 23937939420 bytes)
*Main> last $ showBBaum' t
'>'
(0.16 secs, 10514996 bytes)
*Main>
```

Hierbei ist `t` ein Baum mit ca. 15000 Knoten.

D.h. man sollte die folgende Instanzdefinition verwenden:

```
instance Show a => Show (BBaum a) where
 showsPrec _ = showsBBaum
```

Analog zur Darstellung der Bäume in der `Show`-Instanz, kann man eine `Read`-Instanz für Bäume definieren. Dies lässt sich sehr elegant mit Erfolgslisten und List Comprehensions bewerkstelligen:

```
instance Read a => Read (BBaum a) where
 readsPrec _ = readsBBaum

readsBBaum :: (Read a) => ReadS (BBaum a)
readsBBaum ('<':xs) =
   [(Knoten l r, rest) | (l, '|':ys)   <- readsBBaum xs,
                         (r, '>':rest) <- readsBBaum ys]
readsBBaum s         =
    [(Blatt x, rest)    | (x,rest)      <- reads s]
```

Übungsaufgabe 3.7.3. *Entwerfe eine gut lesbare String-Repräsentation für Bäume mit Knoten- und Kantenmarkierungen vom Typ* `BinBaumMitKM`*. Implementiere Instanzen der Klassen* `Read` *und* `Show` *für den Datentyp* `BinBaumMitKM`

Beachte, dass man bei der Verwendung von `read` manchmal explizit den Ergebnistyp angeben muss, damit der Compiler weiß, welche Implementierung er verwenden muss:

```
Prelude> read "10"

<interactive>:1:0:
 Ambiguous type variable ‘a’ in the constraint
 ‘Read a’ arising from a use of ‘read’ at <interactive>:1:0-8
 Probable fix: add a type signature that
               fixes these type variable(s)

Prelude> (read "10")::Int
10
```

Ein ähnliches Problem tritt auf, wenn man eine überladene Zahl eingibt. Z.B. eine `0`. Der Compiler weiß dann eigentlich nicht, von welchem Typ die Zahl ist. Die im `ghc` durchgeführte Lösung ist *Defaulting*: Wenn der Typ unbekannt ist, aber benötigt wird, so wird für jede Konstante ein default-Typ verwendet. Für Zahlen ist dies der Typ `Integer`[10].

[10] Mit dem Schlüsselwort `default` kann man das Defaulting-Verhalten auf Modulebene anpassen, siehe (Peyton Jones, 2003, Abschnitt 4.3.4)

3.7.4 Die Klassen `Num` und `Enum`

Die Klasse `Num` stellt Operatoren für Zahlen zur Verfügung. Sie ist definiert als:

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
```

Die Funktion `fromInteger` konvertiert dabei eine `Integer`-Zahl in den entsprechenden Typ. Beim Defaulting (s.o.) wird diese Funktion oft eingesetzt (vom Compiler), um eine Definition möglichst allgemein zu halten, obwohl Zahlenkonstanten verwendet werden: Die Konstante erhält den Typ `Integer` wird jedoch durch die Funktion `fromInteger` verpackt.

Betrachte z.B. die Definition der Längenfunktion ohne Typangabe:

```
length [] = 0
length (x:xs) = 1+(length xs)
```

Der Compiler kann dann den Typ `length :: (Num a) => [b] -> a` herleiten, da die Zahlenkonstanten `0` und `1` eigentlich für `fromInteger (0::Integer)` usw. stehen.

Die Klasse `Enum` fasst Typen zusammen, deren Werte aufgezählt werden können. Für Instanzen der Klasse stehen Funktionen zum Erzeugen von Listen zur Verfügung. Die Definition von `Enum` ist:

```
class  Enum a  where
    succ, pred     :: a -> a
    toEnum         :: Int -> a
    fromEnum       :: a -> Int
    enumFrom       :: a -> [a]            -- [n..]
    enumFromThen   :: a -> a -> [a]       -- [n,n'..]
    enumFromTo     :: a -> a -> [a]       -- [n..m]
    enumFromThenTo :: a -> a -> a -> [a]  -- [n,n'..m]
```

Eine Instanz für `Wochentag` kann man definieren als:

```
instance Enum Wochentag where
 toEnum i = tage!!(i `mod` 7)
 fromEnum t = case elemIndex t tage of
               Just i -> i

tage = [Montag, Dienstag, Mittwoch, Donnerstag,
        Freitag, Samstag, Sonntag]
```

Einige Beispielaufrufe:

```
*Main> succ Dienstag
Mittwoch
*Main> pred Montag
Sonntag
*Main> enumFromTo  Montag Sonntag
[Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Samstag,Sonntag]
*Main> enumFromThenTo Montag Mittwoch Sonntag
[Montag,Mittwoch,Freitag,Sonntag]
```

3.7.5 Konstruktorklassen

Die bisher vorgestellten Typklassen abstrahieren über einen Typ, genauer: Die Variable in der Klassendefinition steht für einen Typ. Man kann dies erweitern und hier auch Typkonstruktoren zulassen, über die abstrahiert wird. Konstruktorklassen ermöglichen genau dies. Z.B. ist bei Listen der Typ `[a]` der Typkonstruktor ist `[]` oder einfacher, da keine extra Schreibweise verwendet wird: Bei binären Bäumen ist `BBaum a` der Typ und `BBaum` der Typkonstruktor.

Die Klasse `Functor` ist eine Konstruktorklasse, d.h. sie abstrahiert über einen Typkonstruktor. Sie fasst solche Typkonstruktoren zusammen, für die man eine `map`-Funktion definieren kann (für die Klasse heißt die Funktion allerdings `fmap`). Die Klasse `Functor` ist definiert als:

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

Die Instanz für binäre Bäume kann definiert werden als

```
instance Functor BBaum where
 fmap = bMap
```

Beachte, dass auch in der Instanzdefinition für eine Konstruktorklasse der Typkonstruktor und *nicht* der Typ angegeben wird (also im Beispiel `BBaum` und nicht `BBaum a`). Gibt man die falsche Definition an:

```
instance Functor (BBaum a) where
 fmap = bMap
```

so erhält man eine Fehlermeldung:

```
    Kind mis-match
    The first argument of ‘Functor’ should have kind ‘* -> *’,
    but ‘BBaum a’ has kind ‘*’
    In the instance declaration for ‘Functor (BBaum a)’
```

Wie aus der Fehlermeldung hervorgeht, verwendet Haskell zum Prüfen der richtigen Instanzdefinition so genannte *Kinds*. Man kann sich Kinds wie Typen über Typen vorstellen, d.h. ein Typsystem welches prüft, ob das eigentliche Typsystem richtig verwendet wird. Typen haben dabei den Kind `*`, der Typkonstruktor `BBaum` hat den Kind `* -> *`, denn er erwartet einen Typen als Argument und bei Anwendung auf einen Typ erhält man einen Typ. Der Typkonstruktor `Either` hat den Kind `* -> * -> *`, denn er erwartet zwei Typen.

Man kann allerdings eine `Functor`-Instanz für `Either a` (die Typanwendung ist vom Kind `* -> *`) angeben:

```
instance Functor (Either a) where
 fmap f (Left a) = Left a
 fmap f (Right a) = Right (f a)
```

Für den Datentyp `Maybe` ist eine `Functor`-Instanz bereits vordefiniert als

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

Instanzen von `Functor` sollten die folgenden beiden Gesetze erfüllen:

```
fmap id = id
fmap (f . g)  = fmap f . fmap g
```

Eine weitere vordefinierte Konstruktorklasse ist die Klasse `Monad`. Auf diese werden wir später eingehen.

Übungsaufgabe 3.7.4. *Definiere eine Konstruktorklasse* `Tree`*, die Operationen für Baum-artige Datentypen überlädt. Als Klassenmethoden sollen dabei zur Verfügung stehen:*

- `subtrees` *liefert die Liste der Unterbäume der Wurzel.*
- `isLeaf` *die testet, ob ein Baum nur aus einem Blatt besteht und einen Booleschen Wert liefert.*

Definiere eine Unterklasse `LabeledTree` *von* `Tree`*, die Operatoren für Bäume mit Knotenmarkierungen überlädt. Die Klassenmethoden sind:*

- `label` *liefert die Beschriftung der Wurzel.*
- `nodes` *liefert alle Knotenmarkierungen eines Baums als Liste.*
- `edges` *liefert alle Kanten des Baumes, wobei eine Kante als Paar von Knotenmarkierungen dargestellt wird.*

Gebe default-Implementierungen für `nodes` *und* `edges` *innerhalb der Klassendefinition an.*

Gebe Instanzen für `BinBaum` *für beide Klassen an.*

3.7.6 Übersicht über einige der vordefinierten Typklassen

Abbildung 3.1 zeigt die Klassenhierarchie einiger der vordefinierten Haskell-Typklassen, sowie für welche Datentypen Instanzen der entsprechenden Klassen vordefiniert sind. Es sind einige in den neuesten Versionen hinzugekommen
z.B. `Monoid` und `Foldable`, die auch den Typ einiger eingebauten FUnktionen beeinflussen. Zum Beispiel der Typ von `fold` und `sum` ist aktuell (Version 7.10.2, 2016):

```
fold :: (Foldable t, Monoid m) => t m -> m
sum :: (Num a, Foldable t) => t a -> a
```

3.7.7 Auflösung der Überladung

Jede Programmiersprache mit überladenen Operationen muss irgendwann die Überladung auflösen und die passende Implementierung für einen konkreten Typ verwenden. Beispielsweise wird in Java die Auflösung sowohl zur

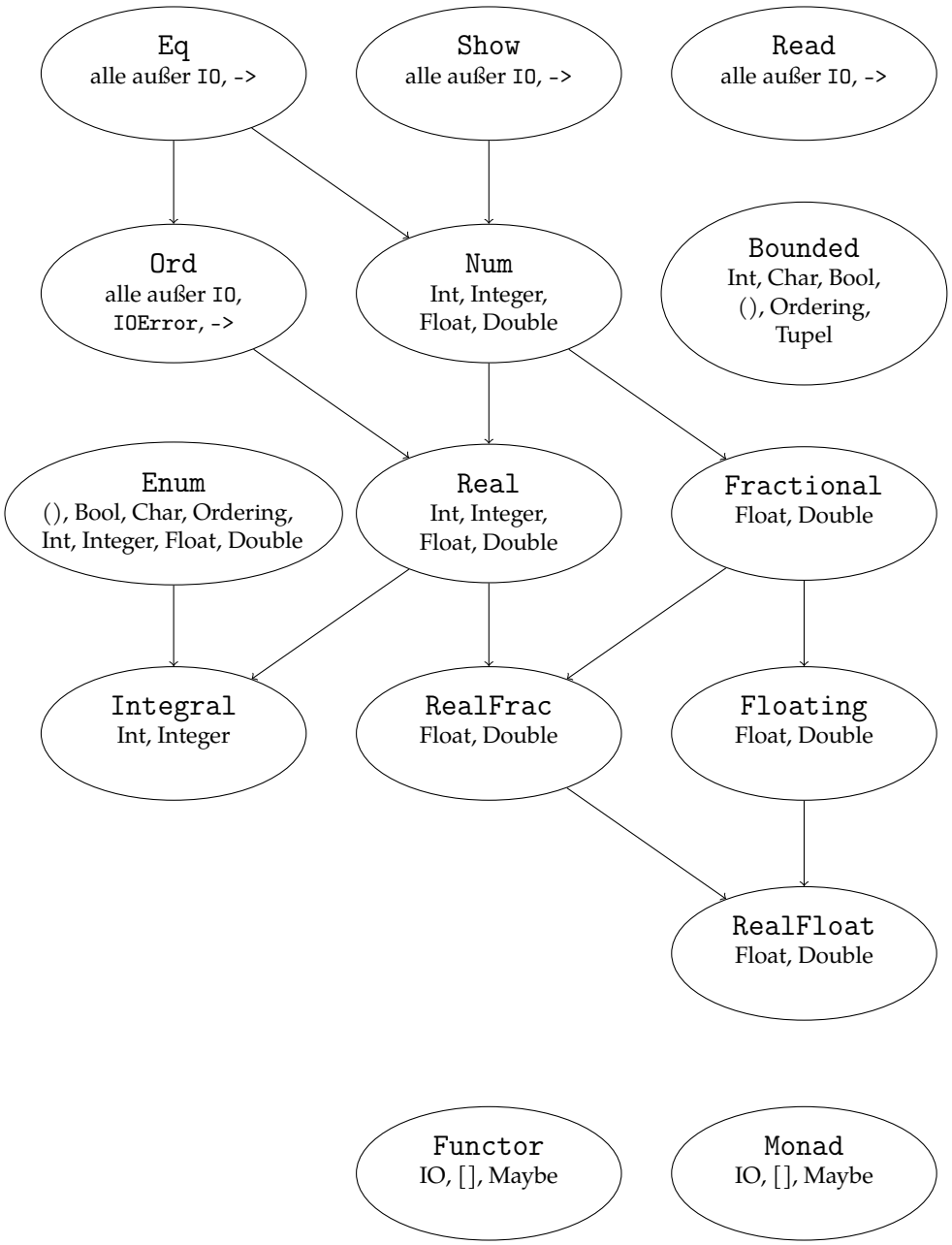


Abbildung 3.1: Typklassenhierarchie der meisten vordefinierten Haskell-Typklassen und Instanzen der vordefinierten Typen

Compilezeit aber manchmal auch erst zur Laufzeit durchgeführt. Je nach Zeitpunkt spricht man in Java von „early binding“ oder „late binding“. In Haskell gibt es keinerlei Typinformation zur Laufzeit, da Typinformationen zur Laufzeit nicht nötig sind. D.h. die Auflösung der Überladung muss zur Compilezeit stattfinden. Das Typklassensystem wird in Haskell in Verbindung mit dem Typcheck vollständig weg transformiert. Die Transformation kann nicht vor dem Typcheck stattfinden, da Typinformationen notwendig sind, um die Überladung aufzulösen. Wir erläutern diese Auflösung anhand von Beispielen und gehen dabei davon aus, dass der Typcheck vorhanden ist (wie er genau funktioniert erläutern wir in einem späteren Kapitel). Beachte, dass die Transformation mit einem Haskell-Programm endet, das keine Typklassen und Instanzen mehr enthält, d.h. die Übersetzung ist „Haskell“ $\rightarrow$ „Haskell ohne Typklassen“.

Für die Auflösung der Überladung werden so genannte Dictionaries eingeführt. An die Stelle einer Typklassendefinition tritt die Definition eines Datentyps eines passenden Dictionaries. Wir werden zu Einfachheit hierfür Datentypen mit Record-Syntax verwenden.

Wir betrachten als Beispiel die Typklasse `Eq`. Wir wiederholen die Klassendefinition:

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

Anstelle der Typklasse tritt nun die Definition eines Dictionary-Datentyps, der ein Produkttyp (genauer: Record-Typ) ist und für jede Klassenmethode eine Komponente des entsprechenden Typs erhält:

```
data EqDict a = EqDict {
                  eqEq  :: a -> a -> Bool, -- f"ur ==
                  eqNeq :: a -> a -> Bool  -- f"ur /=
                }
```

Die default-Implementieren werden als Funktionen definiert, die als zusätzliches Argument ein Dictionary erwarten. Durch Zugriff auf das Dictionary kann == auf /= und umgekehrt zugreifen:

```
-- Default-Implementierung f"ur ==:
default_eqEq eqDict  x y = not (eqNeq eqDict x y)

-- Default-Implementierung f"ur /=:
default_eqNeq eqDict x y = not (eqEq eqDict x y)
```

Anstelle der überladenen Operatoren `(==)` und `(/=)` treten nun die Operatoren:

```
-- Ersatz f"ur ==
overloadedeq :: EqDict a -> a -> a -> Bool
overloadedeq dict a b = eqEq dict a b

-- Ersatz f"ur /=
overloadedneq :: EqDict a -> a -> a -> Bool
overloadedneq dict a b = eqNeq dict a b
```

Beachte, wie sich der Typ verändert hat: Aus der Klassenbeschränkung `Eq a` in `(==) :: Eq a => a -> a -> Bool` wurde ein zusätzlicher Parameter: Das Dictionary für `Eq`.

Jetzt kann man überladene Funktionen entsprechend anpassen und überall, wo `(==)` (mit allgemeinem Typ) stand, einen zusätzlichen Dictionary-Parameter einfügen und `overloadedeq` verwenden. Z.B. wird die Funktion `elem` wie folgt modifiziert: Aus

```
elem :: (Eq a) => a -> [a] -> Bool
elem e []    = False
elem e (x:xs)
  | e == x   = True
  | otherwise = elem e xs
```

wird

```
elemEq :: EqDict a -> a -> [a] -> Bool
elemEq dict e []    = False
elemEq dict e (x:xs)
  | (eqEq dict) e  x   = True
  | otherwise          = elemEq dict e xs
```

Es gibt jedoch stellen im Programm, wo (==) anders ersetzt werden muss. Z.B. muss `True == False` ersetzt werden durch `overloadedeq` mit der Dictionary-Instanz für `Bool`, d.h. aus

```
... True == False ...
```

wird

```
... overloadedeq eqDictBool True False
```

wobei `eqDictBool` ein für `Bool` definiertes Dictionary vom Typ `EqDict Bool` ist. Bevor wir auf die Instanzgenerierung eingehen, sei hier angemerkt, dass das richtige Ersetzen Typinformation benötigt (wir müssen wissen, dass wir das Dictionary für `Bool` an dieser Stelle einsetzen müssen). Deshalb findet die Auflösung der Überladung mit dem Typcheck statt.

Für eine Instanzdefinition wird eine Instanz des Dictionary-Typs angelegt. Wir betrachten zunächst die `Eq`-Instanz für Wochentag:

```
instance Eq Wochentag where
 Montag     == Montag     = True
 Dienstag   == Dienstag   = True
 Mittwoch   == Mittwoch   = True
 Donnerstag == Donnerstag = True
 Freitag    == Freitag    = True
 Samstag    == Samstag    = True
 Sonntag    == Sonntag    = True
 _          == _          = False
```

Diese wird ersetzt durch ein Dictionary vom Typ `EqDict Wochentag`. Da die Instanz keine Definition für /= angibt, wird die Default-Implementierung, also `default_eqNeq` verwendet, wobei das gerade erstellte Dictionary rekursiv angewendet wird. Das ergibt:

```
eqDictWochentag :: EqDict Wochentag
eqDictWochentag =
 EqDict {
   eqEq = eqW,
   eqNeq = default_eqNeq eqDictWochentag
 }
  where
    eqW Montag     Montag     = True
    eqW Dienstag   Dienstag   = True
    eqW Mittwoch   Mittwoch   = True
    eqW Donnerstag Donnerstag = True
    eqW Freitag    Freitag    = True
    eqW Samstag    Samstag    = True
    eqW Sonntag    Sonntag    = True
    eqW _          _          = False
```

Analog kann man ein Dictionary für `Ordering` erstellen (wir geben es hier an, da wir es später benötigen):

```
eqDictOrdering :: EqDict Ordering
eqDictOrdering =
 EqDict {
  eqEq = eqOrdering,
  eqNeq = default_eqNeq eqDictOrdering
 }
  where
   eqOrdering LT LT = True
   eqOrdering EQ EQ = True
   eqOrdering GT GT = True
   eqOrdering _  _  = False
```

Für Instanzen mit Klassenbeschränkungen kann man die Klassenbeschränkung als Parameter für ein weiteres Dictionary auffassen. Wir betrachten die Eq-Instanz für `BBaum`:

```
instance Eq a => Eq (BBaum a) where
 Blatt a == Blatt b           = a == b
 Knoten l1 r1 == Knoten l2 r2 = l1 == l2 && r1 == r2
 _ == _                       = False
```

Die Instanz verlangt, dass der Markierungstyp `a` bereits Instanz der Klasse `Eq` ist. Analog erwartet das `EqDict`-Dictionary für `BBaum a` ein `EqDict a`-Dictionary als Parameter:

```
eqDictBBaum :: EqDict a -> EqDict (BBaum a)
eqDictBBaum dict = EqDict {
                     eqEq  = eqBBaum dict,
                     eqNeq = default_eqNeq (eqDictBBaum dict)
                }
 where
  eqBBaum dict (Blatt a) (Blatt b) =
    overloadededeq dict a b
  eqBBaum dict (Knoten l1 r1) (Knoten l2 r2) =
    eqBBaum dict l1 l2 && eqBBaum dict r1 r2
  eqBBaum dict x y =  False
```

Beachte, dass auf den passenden Gleichheitstest mit `overloadededeq dict` zugegriffen wird.

Als nächsten Fall betrachten wir eine Unterklasse. Der zu erstellende Datentyp für das Dictionary erhält in diesem Fall neben Komponenten für die Klassenmethoden je eine weitere Komponente (ein Dictionary) für jede direkte Oberklasse. Wir betrachten hierfür die von `Eq` abgeleitete Klasse `Ord`:

```
class  (Eq a) => Ord a  where
  compare              :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

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

  max x y | x <= y    =  y
          | otherwise =  x
  min x y | x <= y    =  x
          | otherwise =  y
```

Der Dictionary-Datentyp für `Ord` enthält neben Komponenten für die Klassenmethoden `compare`, `(<)`, `(<=)`, `(>=)`, `(>)`, `compare`, `max` und `min` eine Komponente für ein `EqDict`-Dictionary, da `Eq` Oberklasse von `Ord` ist:

```
data OrdDict a =
 OrdDict {
  eqDict :: EqDict a,
  ordCompare :: a -> a -> Ordering,
  ordL  :: a -> a -> Bool,
  ordLT :: a -> a -> Bool,
  ordGT :: a -> a -> Bool,
  ordG  :: a -> a -> Bool,
  ordMax :: a -> a -> a,
  ordMin :: a -> a -> a
  }
```

Die Default-Implementierungen der Klassenmethoden lassen sich übersetzen als:

```
default_ordCompare dictOrd x y
  | (eqEq (eqDict dictOrd)) x y = EQ
  | (ordLT dictOrd) x y         = LT
  | otherwise                   = GT

default_ordLT dictOrd x y = let compare = (ordCompare dictOrd)
                                nequal  = eqNeq (eqDictOrdering)
                            in (compare x y) `nequal` GT

default_ordL  dictOrd x y = let compare = (ordCompare dictOrd)
                                equal    = eqEq eqDictOrdering
                            in (compare x y) `equal` LT

default_ordGT dictOrd x y = let compare = (ordCompare dictOrd)
                                nequal  = eqNeq eqDictOrdering
                            in (compare x y) `nequal` LT

default_ordG  dictOrd x y = let compare = (ordCompare dictOrd)
                                equal    = eqEq eqDictOrdering
                            in (compare x y) `equal` GT

default_ordMax dictOrd x y
 | (ordLT dictOrd) x y = y
 | otherwise           = x

default_ordMin dictOrd x y
 | (ordLT dictOrd) x y = x
 | otherwise           = y
```

Hierbei ist zu beachten, dass die Definition von `(<)`, `(<=)`, `(>=)` und `(>)` den Gleichheitstest bzw. Ungleichheitstest auf dem Datentyp `Ordering` verwenden. Dementsprechend muss hier das Dictionary `eqDictOrdering` verwendet werden. Für die überladenen Methoden können nun die folgenden Funktionen als Ersatz definiert werden:

```
overloaded_compare :: OrdDict a -> a -> a -> Ordering
overloaded_compare dict = ordCompare dict

overloaded_ordL    :: OrdDict a -> a -> a -> Bool
overloaded_ordL dict = ordL dict

overloaded_ordLT   :: OrdDict a -> a -> a -> Bool
overloaded_ordLT dict = ordLT dict

overloaded_ordGT   :: OrdDict a -> a -> a -> Bool
overloaded_ordGT dict = ordGT dict

overloaded_ordG    :: OrdDict a -> a -> a -> Bool
overloaded_ordG  dict = ordG dict

overloaded_ordMax  :: OrdDict a -> a -> a -> a
overloaded_ordMax dict = ordMax dict

overloaded_ordMin  :: OrdDict a -> a -> a -> a
overloaded_ordMin  dict = ordMin dict
```

Ein Dictionary für die `Ord`-Instanz von `Wochentag` kann nun definiert werden als:

```
ordDictWochentag = OrdDict {
  eqDict = eqDictWochentag,
  ordCompare = default_ordCompare ordDictWochentag,
  ordL  = default_ordL ordDictWochentag,
  ordLT = wt_lt,
  ordGT = default_ordGT ordDictWochentag,
  ordG  = default_ordG ordDictWochentag,
  ordMax = default_ordMax ordDictWochentag,
  ordMin = default_ordMin ordDictWochentag
  }
  where
   wt_lt  a b =
      (a,b) `elem` [(a,b) | i <- [0..6],
                            let a = ys!!i,
                            b <- drop i ys]
   ys = [Montag, Dienstag, Mittwoch, Donnerstag,
         Freitag, Samstag, Sonntag]
```

Hier ist noch etwas Mogelei im Spiel, da `elem` noch aufgelöst werden müsste.

Als abschließenden Fall betrachten wir die Auflösung einer Konstruktorklasse. Wir betrachten die Klasse `Functor`:

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

Hier müssen wir dem Datentyp für das Dictionary auch die Typvariablen `a` und `b` (neben `f`) hinzufügen.

```
data FunctorDict a b f = FunctorDict {
  functorFmap :: (a -> b) -> f a -> f b
}
```

Die überladene Funktion als Ersatz für `fmap` ist:

```
overloaded_fmap :: (FunctorDict a b f) -> (a -> b) -> f a -> F b
overloaded_fmap dict = functorFmap dict
```

Für die `BBaum`-Instanz von `Functor` wird das folgende Dictionary erstellt:

```
functorDictBBaum = FunctorDict {
 functorFmap = bMap
 }
```

3.7.8 Erweiterung von Typklassen

Die vorgestellten Typklassen entsprechen dem Haskell-Standard. Es gibt jedoch einige Erweiterungen, die zwar nicht im Standard definiert sind, jedoch in Compilern implementiert sind. Z.B. kann man mehrere Typvariablen in einer Klassendefinition verwenden, man erhält so genannte *Multiparameterklassen*.

Überlappende bzw. flexible Instanzen sind in Haskell verboten, z.B. darf man nicht definieren:

```
instance Eq (Bool,Bool) where
 (a,b) == (c,d) = ...
```

da der Typ `(Bool,Bool)` zu speziell ist. Überlappende Instanzen sind als Erweiterung des Haskell-Standards verfügbar. Eine weitere Erweiterung sind *Funktionale Abhängigkeiten* etwa in der Form

```
class MyClass a b | a -> b where
```

was ausdrücken soll, dass der Typ `b` durch den Typen `a` bestimmt werden kann.

All diese Erweiterungen haben als Problem, dass das Typsystem kompliziert und u.U. unentscheidbar wird. Man kann mit diesen Erweiterungen zeigen, dass das Typsystem selbst Turing-mächtig wird, also daher unentscheidbar ist, ob der Typcheck terminiert. Dies spricht eher gegen die Einführung solcher Erweiterungen, auch wenn sie mehr Flexibilität beim Programmieren ermöglichen. Umgekehrt bedeutet es: Wenn man die Erweiterungen verwendet, muss man selbst (als Programmierer) darauf achten, dass der Typcheck seines Programms entscheidbar ist.

3.8 Quellennachweise und weiterführende Literatur

Die vorgestellte Übersicht in Haskell ist in den meisten Haskell-Büchern zu finden (siehe Kapitel 1). Formal definiert sind sie im

Haskell-Report ((Peyton Jones, 2003) bzw. (Marlow, 2010)). Die Dokumentation der Standard-Implementierungen findet man unter http://www.haskell.org/ghc/docs/6.12.2/html/libraries/index.html, wobei von besonderer Bedeutung die Bibliothek `Data.List` ist. Das hierachische Modulsystem ist seit Haskell 2010 Teil des Standards.

Haskells Typklassensystem wurde eingeführt in (Wadler & Blott, 1989). Die Standardklassen sind im Haskell Report dokumentiert. In (Wadler & Blott, 1989) ist auch die Auflösung der Überladung zu finden. Tiefergehend formalisiert wurden Haskells Typklassen in (Hall et al., 1996). Konstruktorklassen wurden in (Jones, 1995) vorgeschlagen.

4

Probabilistisches Lazy Programmieren

In diesem Kapitel wird eine probabilistische Erweiterung von call-by-need Programmiersprachen wie Haskell (bzw. den Kernsprachen) definiert und erklärt. Die einfache Idee dazu ist die Einführung einer Operation die einem Münzwurf entspricht.

Als Basis nehmen wir die (ungetypte) pure funktionale Kernsprache KFPTS, d.h. mit Case, Konstruktoren, und rekursiv definierten Superkombinatoren, und ein Konstrukt `let`. (Ob mit oder ohne den strict-Operator ist erstmal nicht wesentlich.)

4.1 Probabilistische Programme

Damit man leichter interessante probabilistische Programme schreiben kann, ohne zu komplexe Konstrukte einzuführen, wird das programmiersprachliche Konstrukt

$$\texttt{coin}\ p\ s\ t$$

eingeführt, bei dem p eine rationale Zahl ist mit $0 \leq p \leq 1$; und s, t Ausdrücke. Sinnvoll ist es, wenn diese den gleichen Typ haben.

Die Beta-Reduktion operiert mit sharing des Arguments (genannt lbeta):
$(\lambda x.s)\ t \xrightarrow{lbeta} \texttt{let}\ x = t\ \texttt{in}\ s$

Die Sprache nennen wir `KFPTSprob`.

Für Untersuchungen zur Ausdruckskraft von Programmen führen wir unterschiedliche Varianten ein:

- p muss eine explizite rationale Zahl sein.
- p darf ein beliebiger Ausdruck sein, der zu einer rationalen Zahl auswertet.

Ein Grund für die Beschränkung ist, dass man die Wahrscheinlichkeiten eher explizit haben will, und dass bei der Möglichkeit Wahrscheinlichkeit erst durch eine Programmausführung zu berechnen, die operationale Semantik

komplexer wird und die Gleichheit von Programmen schwerer handhabbar wird.

Die Auswertung von $(\texttt{c}oin\ p\ s\ t)$ wählt entweder s aus oder t aus und macht dann weiter. D.h. die Auswertung ist nicht-deterministisch, was bedeutet, dass verschiedene Programmabläufe möglich sind.

Die Probabilistische call-by-need Auswertung muss jetzt erstmal geklärt werden, da diese nicht-determinstisch ist und einige Vorsichtsmaßnahmen braucht, damit diese richtig auswertet.

Die Auswertung ist **call-by-need** in `KFPTSprob`. Das ist in etwa Normalordnung mit Sharing. Um es genau zu machen, muss man einige Prinzipien beachten, und für mehrere Fälle die Regeln definieren.

Eine detaillierte exakte Definition kann man finden in der Literatur zu nicht-deterministschen call-by-need functional programming languages.

Einige Prinzipien zur call-by-need Auswertung:

1. let-Ausdrücke werden nach oben geschoben wenn nötig.
2. Bei LBeta und Case-Reduktion wird die Einsetzung mit let geshared.
3. Echt kopiert werden dürfen nur Abstraktionen und Konstruktorapplikationen der Form $c\, x_1 \ldots, x_n$

Details zur call-by-need Auswertung:

1. Das `let` ist normalerweise mit mehreren Bindungen. Es kann rekursive sein oder nicht-rekursiv je nach Sprache.
2. Zunächst wird ein Normalordnungs-Redex gesucht, indem man von oben den nächsten Redex sucht. Bei rekursivem let kann diese Suche in einen Schleife geraten. In dem Fall terminiert der aktuelle Ausdruck nicht.
3. Wenn nicht rekursiv, dann muss man einen extra Fixpunktkombinator hinzufügen, und den im wesentlichen auf lambda-Ausdrucke beschränken; wenn getypt, dann sollte der Typ der gefixpunkten Ausdrücke so sein: $(a \to a) \to (a \to a)$.
4. Wenn rekursives let, dann sind mehr und etwas komplexere Reduktionsregeln nötgi
5. let-Ausdrücke werden nach oben geschoben wenn nötig.

6. Bei LBeta und Case-Reduktion wird Einsetzung geshared.
7. Sharing wird erreicht durch let-Referenzen.
8. Echt kopiert werden dürfen nur Abstraktionen und Konstruktorapplikationen der Form $c\ x_1 \ldots, x_n$

Man kann es genau hinschreiben: Die Beschreibung sind Regeln zum Reduktionskontext finden und ca 15 Reduktionsregeln.

Eine Auswahl von Reduktionsregeln zur probabilistische call-by-need Auswertung ist, die das Typische illustriert.:

$(\lambda x.s)\ t \xrightarrow{lbeta}$ `let` $x = t$ `in` s

$(\texttt{case}\ (c\ s_1\ s_2)\ \texttt{of}\ (c\ x_1\ x_2) \to t; \ldots) \xrightarrow{lcase}$ `let` $x_1 = s_1; x_2 = s_2$ `in` t

$((\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ s)\ t) \xrightarrow{let}$ $(\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ (s\ t))$

$((\texttt{let}\ x_1 = (\texttt{let}\ y_1 = r_1, \ldots, y_m = r_n\ \texttt{in}\ s_1), x_2 = s_2, \ldots, x_n = s_n\ \texttt{in}\ r))$
$\xrightarrow{let}$ $(\texttt{let}\ y_1 = r_1, \ldots, y_m = r_n,\ \ x_1 = s_1, x_2 = s_2, \ldots, x_n = s_n\ \texttt{in}\ r$

• Es gibt noch weitere Regeln um let-Bindungen nach oben zu verschieben

Beispiel 4.1.1. *Zu probabilistischer call-by-need Auswertung*

```
let y = coin 0.75 1 2; z = 0.25 3 4 in
   let x = coin 0.5 y z in x*y+z
```

$\longrightarrow$

```
let y = coin 0.75 1 2; z = 0.25 3 4,
   x = coin 0.5 y z in x*y+z
```

$\longrightarrow x$ zuerst; Wahl: wird zu y. $p = 0.5$

```
let y = coin 0.75 1 2; z = 0.25 3 4,
  x = y in x*y+z
```

$\longrightarrow y$ auswerten, $p = 0.5 * 0.25$

```
let y = 2; z = 0.25 3 4,
  x = y in x*y+z
```

$\longrightarrow z$ auswerten, $p = 0.5 * 0.25 * 0.25$

```
let y = 2; z = 3,
  x = y in x*y+z
```

$\longrightarrow 7$, $p = 0.5 * 0.25 * 0.25 = 1/32$

Anmerkungen dazu:

- Es kommen Bindungen $x = y$ vor:
  $\Longrightarrow$ Kalkülregeln verfeinern, damit das abgedeckt ist.
- Es gibt 8 mögliche Ausführungen. Wahrscheinlichkeiten: $\{0.25, 0.75\} * \{0.25, 0.75\} * 0.5$
- Nur eine Möglichkeit ist auf der Folie.
- Auswertung hängt vom Ausdruck $x * y + z$ ab.
- Auswertung hängt auch vom Zufall ab:
  Wenn Ausdruck = x, dann wird y oder z ausgewertet.

Macht man viele Durchläufe desselben Programms, dann soll s mit Wahrscheinlichkeit p ausgewählt werden und t mit Wahrscheinlichkeit $1 - p$. Wen p nicht im Intervall $[0, 1]$ ist, dann behandelt man Werte ≤ 0 wie 0 und Werte ≥ 1 wie 1.

Man kann das Programm mittels einer sogenannten Monte-Carlo Simulation auswerten (d.h. viele Durchläufe und die coin-Ausdrücke werden zufällig mit der entsprechenden Wahrscheinlichkeit ausgewertet) und man erhält dann Statistiken zu den berechneten Werten und Häufigkeiten und damit auch Annäherungen an die Wahrscheinlichkeit mit der diese Werte ausgewählt und berechnet werden.

Beispiel 4.1.2. *Das Programm* $(coin\ 0.5\ 0\ 1)$ *entspricht einem fairen Münzwurf. Lässt man es sehr oft ablaufen ist die Häufigkeit mit der* 0 *als Ergebnis berechnet wird, in etwa 50%, ebenso die Häufigkeit, mit der* 1 *berechnet wird. Diese berechnete Häufigkeit nähert sich den Wahrscheinlichkeiten wenn man das Programm öfter ausführt.*

Den Vorteil den man aus lazy probabilistischen Programmiersprachen ziehen kann, ist die sichere Programmoptimierung und Programmabänderung, ohne das Ergebnis und ohne die Wahrscheinlichkeiten zu verfälschen.

Die Auswertung von geschlossenen Ausdrücken (bzw. KFPTSProb-Programmen) muss man technisch etwas genauer definieren:
Man geht bei der Definition der Normalordnungsreduktion genauso vor wie bei KFPTS. Allerdings gibt es zu einem festen Programm und Ausdruck i.a. verschiedene Normalordnungsreduktionen, da ja bei Auswertung von $(\texttt{coin}\ p\ s\ t)$ (nicht-deterministisch) nur eine der Möglichkeiten ausgewählt wird. D.h. Verschiedene Auswertung des Programms erzeugen verschiedenen Reduktionsfolgen.

Betrachtet man alle Möglichkeiten, dann erhält man einen Auswertungsbaum, dessen Blätter WHNFs oder nicht weiter auswertbare Ausdrücke sind, aber auch unendlich lange Zweige können vorkommen.
Nimmt man die getypte Variante, hat der Auswertungsbaum nur Blätter mit WHNFs, wobei aber unendliche Zweige können immer noch vorkommen.

Bei der Auswertung, die Wahrscheinlichkeiten mitberechnet: betrachtet man Paare (s, p) von Ausdruck s und einer Zahl $p \in [0, 1]$. Eine deterministische Normalordnungs-Reduktion von $s \to s'$ bedeutet $(s, p) \to (s', p)$, und eine Reduktion eines Ausdrucks der einen Unterausdruck `coin` auswertet, wird so berechnet:

- $(R[\texttt{coin}\ q\ s\ t], p) \to (R[s], p * q)$ (wobei C ein Reduktionskontext ist.)
- $(R[\texttt{coin}\ q\ s\ t], p) \to (R[t], p * (1 - q))$

Die Auswertung eines Programms s startet dann mit $(s, 1)$. Im potentiell unendlichen Reduktionsbaum kann man dann im Prinzip die Wahrscheinlichkeiten der Ausgänge eintragen.

Terminierung einer Einzelreduktion gilt, wenn die Programmausgabe eine WHNF ist. Der unendliche Auswertungsbaum kann dann benutzt werden zur Definition der Terminierungswahrscheinlichkeit: Es ist die Summe der Wahrscheinlichkeiten p aller Äste mit WHNF, und die Divergenzwahrscheinlichkeit ist $1 - P_{conv}$. Unendliche Äste (Teilbäume) ohne eine einzige WHNF betrachtet man als divergent, d.h. deren Wahrscheinlichkeit zu terminieren ist 0.

Das *Programmergebnis* ist wie folgt definiert, wobei es zwei Varianten gibt:

- Eine Wahrscheinlichkeit $0 \leq p \leq 1$, mit der das Programm erfolgreich terminiert.

- Eine diskrete Verteilung: D.h. $(p_i, w_i), i = 1, \ldots$, wobei w_i WHNFs sind. D.h. eine Zuordnung von Wahrscheinlichkeiten zu den Werten (WHNFs).

Wir definieren genauer:

Definition 4.1.3. *Sei s ein geschlossener Ausdruck. Dann ist die (Expected Convergence) $EC(s)$ definiert als die Wahrscheinlichkeit, dass s mit einer WHNF terminiert.*

Definition 4.1.4. *Sei s ein geschlossener Ausdruck.*
Sei $W(s)$ die Menge aller WHNFs w, so dass $s \xrightarrow{*} w$. Die Menge der Paare (w, p) so dass $s \xrightarrow{*} w$ mit Wahrscheinlichkeit p ist die Multi-Verteilung *zu s.*
(Damit ist gemeint, dass bei der Zuordnung $w \to p$ das w mehrfach vorkommen kann. Das Problem ist dass manchmal verschiedene Werte als gleich betrachtet werden, aber die Gleichheit unentscheidbar sein darf.)
Wenn in der Menge der erreichbaren WHNFs die Gleichheit entscheidbar ist, kann man auch sinnvoll eine Verteilung definieren, $EV(s)$.

Beispiel 4.1.5. *Der Münzwurf mit einer fairen Münze entspricht dem Programm* $(\texttt{coin}\ 0.5\ Kopf\ Zahl)$*, wobei $Kopf$ und $Zahl$ in dem Fall Konstanten sind.*
Ein Münzwurf mit einer anderen Wahrscheinlichkeit ist auch möglich: $(\texttt{coin}\ p\ Kopf\ Zahl)$*, wobei p rational sein kann. Wenn p zwischen 0 und 1 ist, ist es die Wahrscheinlichkeit. Wenn $p \leq 0$, dann ist die Wahrscheinlichkeit 0, wenn $p \geq 1$, dann ist die Wahrscheinlichkeit 1.*

Beispiel 4.1.6. *Einen Würfel mit den Zahlen $1, .., 6$ der fair gewürfelt wird, programmiert man mittels:*

$$\texttt{wuerfel} = coin\ (1/6)\ 1\ (\texttt{coin}\ (1/5)\ 2\ (coin\ (1/4)\ 3\ (\texttt{coin}\ (1/3)\ 4\ (\texttt{coin}\ (1/2)\ 5\ 6))))$$

Nachrechnen ergibt, dass die Zahlen $1, \ldots, 6$ mit jeweils Wahrscheinlichkeit $1/6$ benutzt werden.

Bemerkung 4.1.7. *Hat man als Ergebnis eines Programms nur Integer, dann kann man den Erwartungswert wie üblich definieren als $\sum_i i * p_i$, wobei die Verteilung (i, p_i) für ganze Zahlen ist (oder für Zahlen $0, 1, 2, 3, \ldots$) .*

Eine Berechnung ist möglich analog zu einem Monte-Carlo Simulationsprogramm, das den Erwartungswert annähern kann.

Bemerkung 4.1.8. *Verschiedenes Verhalten der Auswertungsreihenfolgen: Call-by-value wertet bei Funktionsanwendung $f\ s$ zuerst das Argument s aus. Call-by-name*

setzt das Argument in die Funktion ein.

	call-by-value	*call-by-need*	*call-by-name*
$(\lambda x.1)\bot$	terminiert nicht	*1*	*1*
let w = coin 1 2 in w+ w	*2,4*	*2,4*	*2,3,4*

Übungsaufgabe 4.1.9. *Schreiben Sie ein funktionales lazy Programm wuerfel in KFPTSProb mit den bisher definierten Mitteln, das zwei faire Würfel mit den Zahlen* $1, .., 6$ *wirft und dann das Produkt ausgibt. Wie sieht die Verteilung der Ergebnissen aus?*

Das Programm soll aus einer programmierten Funktion `wuerfel` *zum Würfeln bestehen und die Ausgabe soll durch* (`wuerfel` ()) $*$ (`wuerfel` ()) *definiert werden.*

Welche Ausgaben haben Wahrscheinlichkeit $1/9$*?*

Wieso darf man nicht `wuerfel` $*$ `wuerfel` *programmieren?*

Beispiel 4.1.10. *Ein Programm das Werte 1,2,3,4,... mit Wahrscheinlichkeit* 0.5, 0.25, 2^{-3}, 2^{-4}, *usw. erzeugt:*

$$\begin{array}{lcl} \texttt{result} & = & \texttt{numbers}\ 1 \\ \texttt{numbers}\ n & = & \texttt{coin}\ 0.5\ n\ (\texttt{numbers}\ (n+1)) \end{array}$$

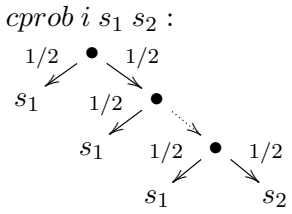
Man erhält $EC(\texttt{result}) = 1$, *und als Verteilung* $EV(\texttt{result})$ *der Ergebnisse die Zuordnung* $i \mapsto 2^{-i}$.

D.h. das Programm kann unendlich lange laufen, wenn $coin$ *immer falsch fällt, aber die Wahrscheinlichkeit dafür ist* 0.

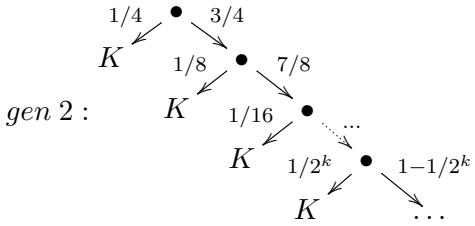
Aber Vorsicht: Es gibt Programme die relativ harmlos aussehen, aber trotzdem eine nicht-0 Wahrscheinlichkeit haben, dass sie nicht terminieren. Hier ein solches Beispiel:

Beispiel 4.1.11. (*Das Originalbeispiel von L. Maio*)
Die Graphik unten zeigt rechts die Verzweigung durch Zufallsoperatoren. Programmieren kann man das, ohne die Wahrscheinlichkeiten wie $1/8$ *usw. explizit als rationale Zahlen anzugeben, indem man* `cprob` *definiert und verwendet.*
Sei $K = \lambda x, y.x$ *die Funktion die konstante Funktionen erzeugt.*

$$\begin{array}{l} s := \texttt{let}\ cprob\ i = if\ i = 0\ then\ K\ else\ \lambda x, y.coin\ 0.5\ (cprob\ (i{-}1)\ x\ y)\ \ y, \\ \qquad\quad gen\ i = cprob\ i\ K\ (gen\ (i{+}1)) \\ \quad \texttt{in}\ gen\ 2 \end{array}$$



D.h., cprob i s_1 s_2 wird mit Wahrscheinlichkeit $1/2^i$ zu s_2 und mit Wahrscheinlichkeit $(1 - 1/2^i)$ zu s_1.



Der Aufruf $(gen\ 2)$ wird mit Wahrscheinlichkeit $1/4$ zu K und mit Wahrscheinlichkeit $3/4$ geht es dann mit $(gen\ 3)$ weiter.

Grobe Abschätzung liefert, dass die Konvergenz (d.h. Terminierung) von (`gen 2`) mit maximal $1/4 + 1/8 + \ldots = 1/2$ eintritt. Wenn man die geometrischen Reihen exakt summiert, ergibt sich, dass der Aufruf `gen 2` mit Wahrscheinlichkeit $5/12$ konvergiert.

Insbesondere divergiert (d.h. terminiert nicht) `gen 2` mit Wahrscheinlichkeit $7/12$, d.h. etwas mehr als 50%.

Das zeigt, dass es Programme gibt, die mit positiver Wahrscheinlichkeit nicht terminieren, auch ohne explizite Nichtterminierung wie zB $\bot$ bzw. Ω.

Das Beispiel lässt sich auch abändern, so dass die Wahrscheinlichkeit der Nichtterminierung beiebig nahe an 1 kommt.

Folgend ein Vergleich mit den Terminierungs-Begriffen zu nicht-deterministischen funktionalen Programmen:

- may-convergent: Der Ausdruck hat eine Möglichkeit mit WHNF zu terminieren.
- may-divergent: Der Ausdruck hat eine Möglichkeit zu divergieren (d.h. unendlich oder Ende ist keine WHNF)
- must-convergent: Jede Reduktionsfolge endet mit einer WHNF.
- must-divergent: keine Reduktionsfolge endet mit einer WHNF.

- should-convergent: Jeder Reduktionsnachfolger ist may-convergent.

Neu in KFPTSProb im Vergleich zu rein nd-Programmen:

- Es gibt should-konvergente geschlossene Ausdrücke, die mit mehr als 50% Wahrscheinlichkeit nicht terminieren (siehe Bspl.)
- Es gibt may-divergente Ausdrücke, die mit Wahrscheinlichkeit 1 konvergieren. Siehe Beispiel oben.

4.1.1 Korrekte Programmtransformationen unter Wahrscheinlichkeiten.

Obige Argumente deuten darauf hin, dass die Begriffe "gleiches Programmen" bzw "gleiche Funktionen" sich gegenüber der Definition die sich nur auf Reduktionseigenschaften bezieht, verändert, wenn man Wahrscheinlichkeiten mit berücksichtigt.

Eine genaue Beschreibung der Sprache erfordert etwas mehr Formalismus: Welche Programme kann man bilden?

- Man kann die Kernsprache KFPTS von Haskell nehmen, mit let, lambda, case, Konstruktoren usw. und die call-by-need Auswertung.
- Diese Sprache kann getypt oder auch ungetypt sein.
- Rekursion kann man mit dem let erreichen bzw. mit der rekursiven Definition der Superkombinatoren.
- Zusätzlich gibt es den Münzwurf $(\texttt{coin}\ p\ s\ t)$, wobei p ein rationaler Ausdruck ist und nur das Intervall $[0, 1]$ interessant ist. Man nimmt $p \leq 0$ als Wahrscheinlichkeit 0, und $p \geq 1$ als Wahrscheinlichkeit 1.

Definition 4.1.12. *Man definiert zwei Ausdrücke s, t als gleich, wenn für alle Kontexte C: $EC(C[s]) = EC(C[t])$, d.h. wenn die Konvergenzwahrscheinlichkeit sich nicht ändert, wenn man s durch t ersetzt oder umgekehrt. Mit Kontext C sind KFPTSProb-Programme gemeint, die eine Leerstelle an einem Ausdruck haben.*

In einer speziellen lazy funktionalen Programmiersprache mit Wahrscheinlichkeiten kann man ziemlich viele Programmtransformationen als korrekt nachweisen, d.h. man kann Programme wirklich umformen, weiter auswerten usw., ohne dass man das Programm vom Effekt her verändert. Natürlich darf man intern (im Compiler) nicht die Münzwürfe ausführen.

Programmäquivalenz ist nicht über Terminierung bzw Konvergenz definiert, sondern über die Wahrscheinlichkeit mit der Programme/Ausdrücke terminieren.

Definition 4.1.13. *Zwei offene Programme sind* **kontextuell äquivalent**, *wenn für jedes Programm R die Konkatenationen $P|R$ (mit main) und $Q|R$ die gleiche Wahrscheinlichkeit der Konvergenz haben.*

Mit Konkatenation zweier Programme ist folgendes gemeint: Das KFPTS-Prob Programm das die Superkombinator-Definition vereinigt, und die main-Funktion des ersten vorher umbenennt.

Definition 4.1.14. *Zwei geschlossene Programme sind* **Verteilungs-äquivalent**, *wenn sie die gleiche Verteilung von Werten erzeugen. (ohne Kontext)*

- Welche Programmtransformationen sind korrekt in KFPTSProb?

  Man kann (in einer anderen Sprachvariante) zeigen, dass folgende Transformationen korrekt sind:

  - LBeta-Reduktion
  - LCase-Reduktionen
  - Weitere, z.B. die Vertauschung der Ausdrücke in `coin` Ausdrücken.

- Welche weiteren Vorteile hat das lazy (call-by-need) probabilistic Programmieren?

  Man kann in nicht zu komplizierten Fällen durch die Auswertung von Programmen manchmal deren Konvergenz-Wahrscheinlichkeit direkt bestimmen, ohne Monte-Carlo Methoden zu verwenden.

  Die Programme sind unmittelbar geeignet zur zufälligen Auswertung mittels Monte-Carlo Methode.

  Es gibt eine umfangreiche Forschung und Literatur zur probabilistischen Programmierung, wobei es zu lazy funktionalen Programmiersprachen noch nicht soviele Untersuchungen gibt.

  Die Haskell Bibliothek `.../packages/probability` hat den Ansatz die diskreten Verteilungen zu berechnen. Das hat Vor- und Nachteile: Man kann weitere Zufallsprozesse zusammensetzen und berechnet direkt deren Verteilung. z.B. die Verteilung der Ergebnisse wenn man zwei Würfel wirft. Es gibt weitere Kombinationsmöglichkeiten von Zufallsvariablen, bzw. deren Verteilungen.

  Diese Sichtweise ist etwas anders als die direkte (rekursive) Programmierung von Zufallsexperimenten. Es ist auf der Basis des packages

leichter, Verteilungen zu Experimenten zu berechnen, aber es ist vermutlich nicht so allgemein wie die direkte Programmierung.

4.1.2 Literatur-Auswahl, und Hinweise auf funktionale Programmpakete.

- (Übersichtsartikel) Ugo Dal Lago. 2020. On Probabilistic Lambda-Calculi. Cambridge University Press, 121-144. `https://doi.org/10.1017/9781108770750.005`
- (Artikel zum Anwendungspotential) Goodman, N. D., Tenenbaum, J. B., & Gerstenberg, T. (2014). Concepts in a probabilistic language of thought. Center for Brains, Minds and Machines (CBMM).
- (Konferenzartikel zu call-by-need probabilistic programs) D. Sabel, M. Schmidt-Schauß, and L. Maio. 2022. Contextual Equivalence in a Probabilistic Call-by-Need Lambda-Calculus. In 24th PPDP 2022, Tbilisi, Georgia. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3551357.3551374
- (Implementierung in Haskell) Luca Maio, The Probabilistic Lambda Calculus with Call-by-Need-Evaluation, thesis, LMU München, 2021
- Martin Erwig, Steve Kollmannsberger, J. Funct. Program. 16(1): 21–34 (2006) `web.engr.oregonstate.edu/(tilde)erwig/papers/PFP_JFP06.pdf`
- Artikel zur Programmäquivalenz: in einer probabilistischen, getypten funktionalen Programmiersprache, analog zu `KFPTSprob` (2023)
  M. Schmidt-Schauß, D. Sabel, Program equivalence in a typed probabilistic call-by-need functional language, J. Log. Algebraic Methods Program. 135, 2023, `https://doi.org/10.1016/j.jlamp.2023.100904`
- Haskell probability:
  https://hackage.haskell.org/packages/probability
- Die Webseite zum Haskell Code der functional pearl zu probability von Martin Erwig, Steve Kollmannsberger:
  https://web.engr.oregonstate.edu/{tilde}erwig/pfp/

5 Typisierung

In diesem Kapitel stellen wir zwei Verfahren zur polymorphen Typisierung von Haskell bzw. KFPTS+seq Programmen vor. Zunächst motivieren wir, warum Typsysteme sinnvoll sind und warum der Begriff „dynamisch ungetypt“ nicht ausreicht. Anschließend führen wir schrittweise Grundbegriffe und notwendige Methoden zur Typisierung ein (u.a. die Unifikation von Typen). Nachdem wir die Typisierung von KFPTS+seq-*Ausdrücken* eingeführt haben, stellen wir zwei Verfahren zur Typisierung von (rekursiven) Superkombinatoren vor: Das iterative Verfahren berechnet allgemeinste polymorphe Typen, aber die zugrundeliegende Aufgabe ist nicht entscheidbar. Das Milner-Verfahren wird in Haskell verwendet, berechnet eingeschränktere Typen als das iterative Verfahren, ist aber terminierend, und somit ein Entscheidungsverfahren.

5.1 Motivation

Wir erörtern zunächst, warum typisierte Programmiersprachen sinnvoll sind, welche Anforderungen man an ein Typsystem stellen sollte und welche Grenzen es dabei gibt. KFPTS+seq-Programme sind nicht typisiert, daher können bei der Auswertung dynamische Typfehler zur Laufzeit auftreten. Solche dynamischen Typfehler sind Programmierfehler, d.h. i.A. rechnet der Programmierer nicht damit, dass sein Programm einen Typfehler hat. Ein starkes statisches Typsystem unterstützt daher den Programmierer, Fehler im Programm möglichst früh (also vor der Laufzeit) zu erkennen. Typen können jedoch auch den Charakter einer Dokumentation übernehmen: Oft kann man am Typ einer Funktion schon ausmachen, welche Funktionalität durch die entsprechende Funktion implementiert wird und wie die Funktion verwendet werden kann. Daher ergeben sich die folgenden Grundanforderungen für die Typisierung von Programmiersprachen:

- Die Typisierung sollte zur Compilezeit entschieden werden.

- Korrekt getypte Programme erzeugen keine Typfehler zur Laufzeit.

Dies sind Mindestanforderungen an ein starkes statisches Typsystem. Gleichzeitig gibt es jedoch weitere wünschenswerte Eigenschaften

- Das Typsystem sollte beim Programmieren möglichst wenig einschränken, d.h. sinnvolle Programme sollten möglichst nicht nur aufgrund des Typsystems als falsche (ungetypte) Programme zurück gewiesen werden.

- Idealerweise muss der Programmierer die Typen nicht (oder nur selten) selbst vorgeben. Selbst Typen anzugeben (und tlw. auch berechnen) ist oft mühsam. In vielen Fällen kann dies jedoch auch sinnvoll sein. Trotzdem ist es wünschenswert, wenn der Compiler (der Typchecker) die Typen des Programms selbst berechnen kann (diesen Vorgang nennt man *Typinferenz*). Auch hier erwartet man, dass der Compiler möglichst die allgemeinsten Typen berechnet.

Es gibt einige Typsysteme, die diese beiden wünschenswerten Eigenschaften nicht erfüllen, z.B. gibt es den *einfach getypten* (simply-typed) Lambda-Kalkül: Für ihn ist der Typcheck zur Compilezeit entscheidbar, aber die Menge der getypten Ausdrücke ist nicht mehr Turing-mächtig, da alle korrekt getypten Programme terminieren (das kann man beweisen!). Der einfach getypte Lambda-Kalkül schränkt daher den Programmierer sehr stark ein. In Haskell-Implementierungen gibt es Erweiterungen des Typsystems (nicht im Haskell-Standard), die vollständige Typinferenz nicht mehr zulassen. Der Programmierer muss dann Typen vorgeben. Es gibt auch Erweiterungen bzw. Typsysteme, die nicht entscheidbar sind, dann kann es passieren, dass der Typcheck (also der Compiler) nicht terminiert.

Ein erster Ansatz für ein allgemeines Typsystem für KFPTSP+seq ist das folgende:

> Ein KFPTSP+seq-Programm ist korrekt getypt, wenn es keine dynamischen Typfehler zur Laufzeit erzeugt.

Dies schränkt die Menge der korrekt getypten Programme, auf solche Programme ein, die nicht dynamisch ungetypt sind.

Leider ist dieser Begriff eher ungeeignet, da die Frage, ob ein beliebiges KFPTS-Programm dynamisch ungetypt ist, unentscheidbar ist. Wir skizzieren den Beweis:

Sei `tmEncode` eine KFPTS+seq-Funktion, die sich wie eine universelle Turingmaschine verhält, d.h. sie erhält eine Turingmaschinenbeschreibung und eine Eingabe für die Turingmaschine und simuliert anschließend die Turingmaschine. Sie liefert `True`, falls die Turingmaschine auf dieser Eingabe anhält und akzeptiert. Wir nehmen an, dass `tmEncode` angewendet auf passende Argumente niemals dynamisch ungetypt ist[1].

Für eine Turingmaschinenbeschreibung b und eine Eingabe e für diese Turingmaschine sei der Ausdruck s definiert durch:

$$
\begin{aligned}
s := \quad & \texttt{if tmEncode } b\ e \\
& \texttt{then case}_{\text{Bool}}\ \texttt{Nil of } \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\} \\
& \texttt{else case}_{\text{Bool}}\ \texttt{Nil of } \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\}
\end{aligned}
$$

Da $(\texttt{tmEncode}\ b\ e)$ nicht dynamisch ungetypt ist, gilt: s ist genau dann dynamisch ungetypt, wenn die Auswertung von $(\texttt{tmEncode}\ b\ e)$ mit `True` endet. Umgekehrt bedeutet dies: Wenn wir entscheiden könnten, ob s dynamisch ungetypt ist, dann können wir auch das Halteproblem für Turingmaschinen entscheiden, was bekanntlich nicht möglich ist. Daher folgt:

Satz 5.1.1. *Die dynamische Typisierung von* KFPTS+seq*-Programmen ist unentscheidbar.*

Wir werden daher Typsysteme betrachten, die die Programmierung weiter einschränken, d.h. für die beiden von uns im folgenden betrachteten Typsysteme gilt:

- Ein korrekt getypter Ausdruck ist nicht dynamisch ungetypt (sonst wäre das Typsystem nicht viel wert)
- Es gibt Ausdrücke, die nicht dynamisch ungetypt sind, aber trotzdem nicht korrekt getypt sind.

[1] Diese Annahme scheint zunächst etwas unrealistisch, da wir ja gerade nachweisen möchten, dass diese Frage unentscheidbar ist. Aber: Wir zeigen, dass die Frage, ob ein beliebiger KFPTS+seq-Ausdruck dynamisch ungetypt ist, unentscheidbar ist. `tmEncode` ist glücklicherweise Milner-Typisierbar (das folgt, da `tmEncode` in Haskell typisierbar ist), und für Milner-getypte KFPTS-Programme gilt: Solche Programme sind niemals dynamisch ungetypt (was wir in einem späteren Abschnitt auch zeigen werden). Im Anhang ist zur Vollständigkeit eine Haskell-Implementierung für `tmEncode` angegeben.

5.2 Typen: Sprechweisen, Notationen und Unifikation

In diesem Abschnitt führen wir einige Sprechweisen und Notationen für Typen ein. Wir wiederholen zunächst die Syntax von polymorphen Typen:

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1 \to \mathbf{T}_2$$

wobei TV für eine Typvariable steht und TC ein Typkonstruktor ist[2].

Wir führen die folgenden Sprechweisen ein:

Definition 5.2.1. *Ein* Basistyp *ist ein Typ der Form TC, wobei TC ein nullstelliger Typkonstruktor ist. Ein* Grundtyp (*oder alternativ* monomorpher Typ) *ist ein Typ, der keine Typvariablen enthält.*

Beispiel 5.2.2. *Die Typen* `Int`, `Bool` *und* `Char` *sind Basistypen. Die Typen* `[Int]` *und* `Char -> Int` *sind keine Basistypen aber Grundtypen. Die Typen* `[a]` *und* `a -> a` *sind weder Basistypen noch Grundtypen.*

Wir verwenden als Notation auch *all-quantifizierte Typen*. Sei τ ein polymorpher Typ mit Vorkommen der Typvariablen $\alpha_1, \ldots, \alpha_n$, dann ist $\forall\alpha_1, \ldots, \alpha_n.\tau$ der *all-quantifizierte Typ* für τ. Man kann polymorphe Typen als allquantifiziert ansehen, da die Typvariablen für beliebige Typen stehen können. Wir benutzen die Quantor-Syntax, um (im Typisierungsverfahren) zwischen Typen zu unterscheiden die „kopiert“ (also umbenannt) werden dürfen und Typen für die wir im Typisierungsverfahren keine Umbenennung erlauben. Die Reihenfolge der allquantifizierten Typvariablen am Quantor spielt keine Rolle. Deshalb verwenden wir auch die Notation $\forall\mathcal{X}.\tau$, wobei $\mathcal{X}$ eine Menge von Typvariablen ist.

Definition 5.2.3. *Eine Typsubstitution ist eine Abbildung $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ einer endlichen Menge von Typvariablen auf Typen. Sei σ eine Typsubstitution. Die homomorphe Erweiterung σ_E von σ ist die Erweiterung der Typsubstitution als Abbildung von Typen auf Typen, die definiert ist durch:*

$$\begin{aligned}
\sigma_E(TV) &:= \sigma(TV), \textit{ falls } \sigma \textit{ die Variable } TV \textit{ abbildet} \\
\sigma_E(TV) &:= TV, \textit{ falls } \sigma \textit{ die Variable } TV \textit{ nicht abbildet} \\
\sigma_E(TC\ T_1\ \ldots\ T_n) &:= TC\ \sigma_E(T_1)\ \ldots\ \sigma_E(T_n) \\
\sigma_E(T_1 \to T_2) &:= \sigma_E(T_1) \to \sigma_E(T_2)
\end{aligned}$$

[2] Es sei nochmals daran erinnert, dass der Listentyp in Haskell als `[a]` dargestellt wird, der Typkonstruktor ist hierbei `[]`. Ebenso wird für Tupel-Typen eine spezielle Syntax verwendet: `(,...,)` ist der Typkonstruktor, z.B. ist `(a,b)` der Typ für Paare. Wir verwenden diese Syntax auch im Folgenden

Wir unterscheiden im folgenden nicht zwischen σ und der Erweiterung σ_E.

Eine Typsubstitution ist eine Grundsubstitution für einen Typ τ *genau dann, wenn σ auf Grundtypen abbildet und alle Typvariablen, die in τ vorkommen, durch σ auf Grundtypen abgebildet werden.*

Mit Grundsubstitutionen kann man eine Semantik für polymorphe Typen angeben, die auf monomorphe Typen zurückführt. Sei τ ein polymorpher Typ, dann ist die Grundtypensemantik $\mathrm{sem}(\tau)$ von τ die Menge aller möglichen Grundtypen, die durch Substitution, angewendet auf τ, erzeugt werden können, d.h.

$$\mathtt{sem}(\tau) := \{\sigma(\tau) \mid \sigma \text{ ist Grundsubstitution für } \tau\}$$

Das entspricht gerade der Vorstellung von *schematischen* Typen: Ein polymorpher Typ beschreibt das Schema einer Menge von Grundtypen.

Wir haben bereits in Kapitel 2 (Abschnitt 2.5) einfache Typregeln eingeführt. Betrachten wir erneut die Typregel für die Anwendung

$$\frac{s :: T_1 \to T_2, t :: T_1}{(s\ t) :: T_2}$$

Diese Regel verlangt dass der Argumenttyp von s schon passend zum Typ von t ist. Betrachtet man z.B. die Anwendung (`map not`) unter der Annahme, dass wir die (allgemeinsten) Haskell-Typen von `map` und `not` schon kennen:

```
map :: (a -> b) -> [a] -> [b]
not :: Bool -> Bool
```

Damit die Anwendungsregel verwendbar ist, müssen wir vorher den Typ von `map` instanziieren (d.h. die passende Substitution ist $\sigma = \{\mathtt{a} \mapsto \mathtt{Bool}, \mathtt{b} \mapsto \mathtt{Bool}\}$). Hier haben wir die passende Substitution durch Hinschauen geraten. Ein allgemeines (und automatisierbares) Verfahren zum „Gleichmachen“ von Typen (und i.A. auch anderen Termen) ist die *Unifikation*. Diese werden wir nun formal definieren:

Definition 5.2.4. *Ein* Unifikationsproblem *auf Typen ist gegeben durch eine Menge von Gleichungen der Form $\tau_1 \doteq \tau_2$, wobei τ_1 und τ_2 polymorphe Typen sind. Eine Lösung eines Unifikationsproblem auf Typen ist eine Substitution σ (bezeichnet als* Unifikator*), so dass $\sigma(\tau_1) = \sigma(\tau_2)$ für alle Gleichungen $\tau_1 \doteq \tau_2$ des Problems. Eine* allgemeinste Lösung (*allgemeinster Unifikator, mgu = most general unifier*)

von Γ ist ein Unifikator σ, so dass gilt: Für jeden anderen Unifikator ρ gibt es eine Substitution γ so dass $\rho(x) = \gamma \circ \sigma(x)$ für all $x \in FV(\Gamma)$.

Man kann zeigen, dass allgemeinste Unifikatoren eindeutig bis auf Umbenennung von Variablen sind.

Beispiel 5.2.5. *Für das Unifikationsproblem $\{\alpha \doteq \beta\}$ sind $\sigma = \{\alpha \mapsto \beta\}$ und $\sigma' = \{\beta \mapsto \alpha\}$ allgemeinste Unifikatoren. Die Substitution $\sigma'' = \{\alpha \mapsto \texttt{Bool}, \beta \mapsto \texttt{Bool}\}$ ist ein Unifikator aber kein allgemeinster.*

Der folgende Unifikationsalgorithmus berechnet einen allgemeinsten Unifikator, falls dieser existiert. Die Datenstruktur ist eine Multimenge von zu lösenden Gleichungen. (Multimengen sind analog zu Mengen, aber mehrfaches Vorkommen von Elementen ist erlaubt.) Wir verwenden dabei Γ für solche (Multi-)Mengen und $\Gamma \cup \Gamma'$ stellt die disjunkte Vereinigung zweier Multi-Mengen dar. Die Notation $\Gamma[\tau/\alpha]$ bedeutet, dass in allen Gleichungen in Γ die Typvariable α durch τ ersetzt wird (d.h. auf alle linken und rechten Seiten der Gleichungen wird die Substitution $\{\alpha \mapsto \tau\}$ angewendet). Der Algorithmus startet mit dem zum Unifikationsproblem passenden Gleichungssystem und wendet anschließend die folgenden Schlussregeln (nichtdeterministisch) solange an, bis

- keine Regel mehr anwendbar ist,
- oder Fail auftritt.

$$\textsc{Fail1}\ \frac{\Gamma \cup \{(TC_1\ \tau_1\ \ldots\ \tau_n) \doteq (TC_2\ \tau'_1\ \ldots\ \tau'_m)\}}{\text{Fail}} \quad \text{wenn } TC_1 \neq TC_2$$

$$\textsc{Fail2}\ \frac{\Gamma \cup \{(TC_1\ \tau_1\ \ldots\ \tau_n) \doteq (\tau'_1 \to \tau'_2)\}}{\text{Fail}}$$

$$\textsc{Fail3}\ \frac{\Gamma \cup \{(\tau'_1 \to \tau'_2) \doteq (TC_1\ \tau_1\ \ldots\ \tau_n)\}}{\text{Fail}}$$

$$\textsc{Decompose1}\ \frac{\Gamma \cup \{TC\ \tau_1\ \ldots\ \tau_n \doteq TC\ \tau'_1\ \ldots\ \tau'_n\}}{\Gamma \cup \{\tau_1 \doteq \tau'_1, \ldots, \tau_n \doteq \tau'_n\}}$$

$$\textsc{Decompose2}\ \frac{\Gamma \cup \{\tau_1 \to \tau_2 \doteq \tau'_1 \to \tau'_2\}}{\Gamma \cup \{\tau_1 \doteq \tau'_1, \tau_2 \doteq \tau'_2\}}$$

$$\textsc{Orient}\ \frac{\Gamma \cup \{\tau_1 \dot{=} \alpha\}}{\Gamma \cup \{\alpha \dot{=} \tau_1\}}$$

wenn τ_1 keine Typvariable und α Typvariable

$$\textsc{Elim}\ \frac{\Gamma \cup \{\alpha \dot{=} \alpha\}}{\Gamma}$$

wobei α Typvariable

$$\textsc{Solve}\ \frac{\Gamma \cup \{\alpha \dot{=} \tau\}}{\Gamma[\tau/\alpha] \cup \{\alpha \dot{=} \tau\}}$$

wenn Typvariable α nicht in τ vorkommt,
aber α kommt in Γ vor

$$\textsc{OccursCheck}\ \frac{\Gamma \cup \{\alpha \dot{=} \tau\}}{\text{Fail}}$$

wenn $\tau \neq \alpha$ und Typvariable α kommt in τ vor

Man kann zeigen, dass der Algorithmus die folgenden Eigenschaften erfüllt (wir verzichten auf einen Beweis):

- Der Algorithmus endet mit Fail gdw. es keinen Unifikator für die Eingabe gibt.
- Der Algorithmus endet erfolgreich gdw. es einen Unifikator für die Eingabe gibt. Das Gleichungssystem Γ ist dann von der Form

  $$\{\alpha_1 \dot{=} \tau_1, \ldots, \alpha_n \dot{=} \tau_n\},$$

  wobei α_i paarweise verschiedene Typvariablen sind und kein α_i in irgendeinem τ_j vorkommt. Der Unifikator kann dann abgelesen werden als $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$.
- Liefert der Algorithmus einen Unifikator, dann ist es ein allgemeinster Unifikator.
- Man braucht keine alternativen Regelanwendungen auszuprobieren! Der Algorithmus kann deterministisch implementiert werden.
- Der Algorithmus terminiert für jedes Unifikationsproblem auf Typen[3].

[3] Für den interessierten Leser skizzieren wir den Beweis: Sei Γ ein Unifikationsproblem und

- Die Typen in der Resultat-Substitution können exponentiell groß werden.

- der Unifikationsalgorithmus kann so implementiert werden, dass er Zeit $O(n * \log n)$ benötigt. Man muss Sharing dazu beachten; und dazu eine andere Solve-Regel benutzen.

- Das Unifikationsproblem (d.h. die Frage, ob eine (Multi-)Menge von Typgleichungen unifizierbar ist) ist P-complete. D.h. man kann im wesentlichen alle PTIME-Probleme als Unifikationsproblem darstellen: Interpretation ist: Unifikation ist nicht effizient parallelisierbar.

Beispiel 5.2.6. *Das Unifikationsproblem* $\{(a \to b) \doteq \texttt{Bool} \to \texttt{Bool}\}$ *wird durch den Unifikationsalgorithmus in einem Schritt gelöst:*

$$\textsc{Decompose2}\ \frac{\{(a \to b) \doteq \texttt{Bool} \to \texttt{Bool}\}}{\{a \doteq \texttt{Bool}, b \doteq \texttt{Bool}\}}$$

Beispiel 5.2.7. *Betrachte das Unifikationsproblem* $\{a \to [a] \doteq \texttt{Bool} \to c, [d] \doteq c\}$. *Ein Ablauf des Unifikationsalgorithmus ist:*

- $\mathit{Var}(\Gamma)$ die Anzahl an *ungelösten* Typvariablen in Γ, wobei eine Variable α gelöst ist, falls sie nur einmal in Γ in einer Gleichung auf der linken Seite vorkommt (d.h. $\Gamma = \Gamma' \cup \{\alpha \doteq \tau\}$ wobei α weder in Γ' noch in τ vorkommt).
- $\mathit{Size}(\Gamma)$ ist die Summe alle Typtermgrößen aller Typen der rechten und linken Seiten in Γ, wobei die Typtermgröße $\texttt{ttg}$ eines Typen definiert ist als:

$$\begin{array}{lcl} \texttt{ttg}(TV) & = & 1 \\ \texttt{ttg}(TC\ T_1\ \ldots\ T_n) & = & 1 + \sum_{i=1}^{n} \texttt{ttg}(T_i) \\ \texttt{ttg}(T_1 \to T_2) & = & 1 + \texttt{ttg}(T_1) + \texttt{ttg}(T_2) \end{array}$$

- $\mathit{OEq}(\Gamma)$ ist die Anzahl der ungerichteten Gleichungen in Γ, wobei eine Gleichung gerichtet ist, falls sie von der Form $\alpha \doteq \tau$ ist, wobei α eine Typvariable ist.
- $M(\Gamma) = (\mathit{Var}(\Gamma), \mathit{Size}(\Gamma), \mathit{OEq}(\Gamma))$, wobei $M(\text{Fail}) := (-1, -1, -1)$.

Für den Terminierungsbeweis zeigen wir, dass für jede Regel $\frac{\Gamma}{\Gamma'}$ gilt: $M(\Gamma') <_{lex} M(\Gamma)$, wobei $<_{lex}$ die lexikographische Ordnung auf 3-Tupeln sei. Vorher muss man noch verifizieren, dass das Maß M wohl-fundiert ist, was jedoch einsichtig ist. Wir gehen die Regeln durch: Die Fail-Regeln und die Regel (OccursCheck) verkleinern das Maß stets. Die Decompose-Regeln vergrößern $\mathit{Var}(.)$ nicht, verkleinern aber stets $\mathit{Size}(.)$ (und vergrößern evtl. $\mathit{OEq}(.)$ was aber nicht stört aufgrund der lexikographischen Ordnung). Die (Orient)-Regel vergrößert $\mathit{Var}(.)$ nicht, lässt $\mathit{Size}(.)$ unverändert, aber verkleinert $\mathit{OEq}(.)$. Die (Elim)-Regel vergrößert $\mathit{Var}(.)$ nicht, aber verkleinert $\mathit{Size}(.)$. Die (Solve)-Regel verkleinert $\mathit{Var}(.)$ (und vergrößert $\mathit{Size}(.)$).

$$\textsc{Decompose1}\ \dfrac{\textsc{Solve}\ \dfrac{\textsc{Solve}\ \dfrac{\textsc{Orient}\ \dfrac{\textsc{Decompose2}\ \dfrac{\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}}{\{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\}}}{\{[d] \doteq c, a \doteq \texttt{Bool}, c \doteq [a]\}}}{\{[d] \doteq [a], a \doteq \texttt{Bool}, c \doteq [a]\}}}{\{[d] \doteq [\texttt{Bool}], a \doteq \texttt{Bool}, c \doteq [\texttt{Bool}]\}}}{\{d \doteq \texttt{Bool}, a \doteq \texttt{Bool}, c \doteq [\texttt{Bool}]\}}$$

Der Unifikator ist $\{d \mapsto \texttt{Bool}, a \mapsto \texttt{Bool}, c \mapsto [\texttt{Bool}]\}$.

Beispiel 5.2.8. *Das Unifikationsproblem* $\{a \doteq [b], b \doteq [a]\}$ *hat keine Lösung:*

$$\textsc{OccursCheck}\ \dfrac{\textsc{Solve}\ \dfrac{\{a \doteq [b], b \doteq [a]\}}{\{a \doteq [[a]], b \doteq [a]\}}}{\mathit{Fail}}$$

Das Unifikationsproblem $\{a \to [b] \doteq a \to c \to d\}$ *hat ebenfalls keine Lösung:*

$$\textsc{Fail2}\ \dfrac{\textsc{Elim}\ \dfrac{\textsc{Decompose2}\ \dfrac{\{a \to [b] \doteq a \to c \to d\}}{\{a \doteq a, [b] \doteq c \to d\}}}{\{[b] \doteq c \to d\}}}{\mathit{Fail}}$$

Übungsaufgabe 5.2.9. *Löse die Unifikationsprobleme:*

- $\{\texttt{BBaum}\ a \doteq \texttt{BBaum}\ (b \to c), [b] \doteq [[c]], d \to e \to f = \texttt{Bool} \to c\}$
- $\{a \to g \to (b \to c) \to (d \to e) \doteq a \to f, ([b] \to [c] \to [d] \to ([e] \to [g])) \to h \doteq f \to h\}$

Wir verwenden ab jetzt die Unifikation in den Beispielen ohne den Rechenweg genauer anzugeben.

5.3 Polymorphe Typisierung für KFPTSP+seq-Ausdrücke

Wir betrachten zunächst die Typisierung von Ausdrücken, später werden wir auf die Typisierung von Superkombinatoren eingehen. Zunächst nehmen wir an, Superkombinatoren seien getypt.

Da wir nun wissen, wie Typen unifiziert werden, können wir die Regel für

die Anwendung für polymorphe Typen allgemeiner formulieren:

$$\frac{s :: \tau_1, t :: \tau_2}{(s\ t) :: \sigma(\alpha)}$$
wenn σ allgemeinster Unifikator für $\tau_1 \doteq \tau_2 \to \alpha$ ist
und α neue Typvariable ist.

Allerdings reicht dieses Vorgehen noch nicht aus, um Ausdrücke mit Bindern zu typisieren. Betrachte z.B. die Typisierung einer Abstraktion $\lambda x.s$. Will man $\lambda x.s$ typisieren, so muss man den Rumpf s typisieren, und dann einen entsprechenden Funktionstypen konstruieren. Z.B. für $\lambda x.\texttt{True}$ kann man `True` mit `Bool` typisieren, und dann die Abstraktion mit $\alpha \to \texttt{True}$ typisieren. Dieser Fall ist aber zu einfach, denn x kann im Rumpf s vorkommen und alle Vorkommen von x müssen gleich typisiert werden. Deswegen ist die bessere Regel von der Form:

$$\frac{\text{Typisierung von } s \text{ unter der Annahme } x \text{ hat Typ } \tau \text{ ergibt } s :: \tau'}{\lambda x.s :: \tau \to \tau'}$$

Allerdings muss man nun den richtigen Typ τ für x erraten, was (zumindest algorithmisch) wenig sinnvoll ist. Deshalb gibt man zunächst einen allgemeinsten Typ für x vor und berechnet den passenden Typen durch Unifikation. Um dies effizient zu implementieren (und nicht zu oft zu unifizieren), kann man alle Typisierungsschritte durchführen und sich dabei zu unifizierende Gleichungen merken und erst ganz am Ende eine Unifikation durchführen. Genau dieses Verfahren wenden wir an. Die Herleitungsregeln werden daher verändert: Neben einem Typ leiten wir Gleichungssysteme her (die wir mit E notieren). Die obige Typregel für Abstraktionen zeigt auch, dass wir uns Annahmen merken müssen, was zu einer weiteren Notationserweiterung führt. Die Notation ist

$$A \vdash s :: \tau, E.$$

Sie bedeutet: Unter der Annahme A (die eine Menge von Annahmen darstellt) kann für den Ausdruck s der Typ τ und die Typgleichungen E hergeleitet werden.

Am Anfang bestehen die Annahmen aus schon bekannten Typen: Dies sind Typen für Superkombinatoren und für die Konstruktoren. Wir verwenden hierbei all-quantifizierte polymorphe Typen, um zu verdeutlichen, dass diese Typen umbenannt werden dürfen. Z.B. können wir für `map` in die Annahme

einfügen: `map` $:: \forall a, b.(a \to b) \to [a] \to [b]$. Konstruktoranwendungen werden in den folgenden Typisierungsregeln wie Anwendungen auf eine Konstante behandelt, z.B. wird `Cons True Nil` wie (`(Cons True)` `Nil`) behandelt. Die Typannahme für `Cons` ist z.B. `Cons` $:: \forall a.a \to [a] \to [a]$.

Wir geben nun Regeln zur Typisierung von KFPTSP+seq-Ausdrücken an, wobei auftretende Superkombinatoren alle als bereits typisiert angenommen werden (und daher in den Annahmen enthalten sind):

Axiom für Variablen:

$$\text{(AxV)}\ \frac{}{A \cup \{x :: \tau\} \vdash x :: \tau, \emptyset}$$

Axiom für Konstruktoren:

$$\text{(AxK)}\ \frac{}{A \cup \{c :: \forall \alpha_1 \ldots \alpha_n.\tau\} \vdash c :: \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n], \emptyset}$$

wobei β_i neue Typvariablen sind

Axiom für Superkombinatoren:

$$\text{(AxSK)}\ \frac{}{A \cup \{SK :: \forall \alpha_1 \ldots \alpha_n.\tau\} \vdash SK :: \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n], \emptyset}$$

wobei β_i neue Typvariablen sind

Regel für Anwendungen:

$$\text{(RApp)}\ \frac{A \vdash s :: \tau_1, E_1 \quad \text{und} \quad A \vdash t :: \tau_2, E_2}{A \vdash (s\ t) :: \alpha, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha\}}$$

wobei α neue Typvariable

Regel für `seq`:

$$\text{(RSeq)}\ \frac{A \vdash s :: \tau_1, E_1 \quad \text{und} \quad A \vdash t :: \tau_2, E_2}{A \vdash (\texttt{seq}\ s\ t) :: \tau_2, E_1 \cup E_2}$$

Regel für Abstraktionen:

$$\text{(RAbs)}\ \frac{A \cup \{x :: \alpha\} \vdash s :: \tau, E}{A \vdash \lambda x.s :: \alpha \to \tau, E}$$

wobei α eine neue Typvariable

Regel für `case`:

$$\text{(RCase)}\ \frac{\begin{array}{l} A \vdash s :: \tau, E \\ \text{für alle } i = 1, \ldots, m\text{:} \\ \quad A \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,\mathrm{ar}(c_i)} :: \alpha_{i,\mathrm{ar}(c_i)}\} \vdash (c_i\ x_{i,1}\ \ldots\ x_{i,\mathrm{ar}(c_i)}) :: \tau_i, E_i \\ \text{für alle } i = 1, \ldots, m\text{:} \\ \quad A \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,\mathrm{ar}(c_i)} :: \alpha_{i,\mathrm{ar}(c_i)}\} \vdash t_i :: \tau_i', E_i' \end{array}}{A \vdash \left(\begin{array}{l} \texttt{case}_{Typ}\ s\ \texttt{of}\{ \\ \quad (c_1\ x_{1,1}\ \ldots\ x_{1,\mathrm{ar}(c_1)}) \to t_1; \\ \quad \ldots; \\ \quad (c_m\ x_{m,1}\ \ldots\ x_{m,\mathrm{ar}(c_m)}) \to t_m\} \end{array} \right) :: \alpha, E'}$$

$$\text{wobei } E' = E \cup \bigcup_{i=1}^{m} E_i \cup \bigcup_{i=1}^{m} E_i' \cup \bigcup_{i=1}^{m} \{\tau \doteq \tau_i\} \cup \bigcup_{i=1}^{m} \{\alpha \doteq \tau_i'\}$$

und $\alpha_{i,j}, \alpha$ neue Typvariablen sind

Damit lässt sich der Typ eines Ausdrucks s wie folgt berechnen:

Algorithmus 5.3.1 (Typisierung von KFPTSP+seq-Ausdrücken)**.** *Sei s ein geschlossener* KFPTSP+seq*-Ausdruck, wobei die Typen für alle in s benutzten Superkombinatoren und Konstruktoren bekannt sind (d.h. schon berechnet wurden)*

1. *Starte mit Anfangsannahme A, die Typen für die Konstruktoren und die Superkombinatoren enthält.*
2. *Leite $A \vdash s :: \tau, E$ mit den Typisierungsregeln her.*
3. *Löse E mit Unifikation.*
4. *Wenn die Unifikation mit Fail endet, ist s nicht typisierbar; Andernfalls: Sei σ ein allgemeinster Unifikator von E, dann gilt $s :: \sigma(\tau)$.*

Als Optimierung kann man zu obigen Herleitungsregeln hinzufügen:
Typberechnung:

$$\text{(RUnif)}\ \frac{A \vdash s :: \tau, E}{A \vdash s :: \sigma(\tau), E_\sigma}$$

wobei E_σ das gelöste Gleichungssystem zu E ist
und σ der ablesbare Unifikator ist

Damit kann man jederzeit zwischendrin schon einmal unifizieren.

Definition 5.3.2. *Ein* (KFPTSP+seq) *Ausdruck s ist* wohl-getypt*, wenn er sich mit obigem Verfahren typisieren lässt.*

Wir betrachten einige Beispiele:

Beispiel 5.3.3. *Betrachte die Konstruktoranwendung* `Cons True Nil`*. In der Anfangsannahme sind die Typen für* `Cons`*,* `Nil` *und* `True` *enthalten:* $A_0 = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} :: \forall a.[a], \texttt{True} :: \texttt{Bool}\}$.

Zur Typisierung von `Cons True Nil` *fängt man von unten nach oben an, den Herleitungsbaum aufzubauen, d.h. der erste Schritt ist:*

$$(RApp)\ \frac{A_0 \vdash (\texttt{Cons True}) :: \tau_1, E_1, \quad A_0 \vdash \texttt{Nil} :: \tau_2, E_2}{A_0 \vdash \texttt{Cons True Nil} :: \alpha_4, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha_4\}}$$

Die genauen Typen τ_1, τ_2 *und die Gleichungssysteme* E_1, E_2 *erhält man erst beim nach oben Steigen im Herleitungsbaum. Insgesamt muss man zweimal die Anwendungsregel und dreimal das Axiom für Konstanten anwenden und erhält dann den vollständigen Herleitungsbaum:*

$$\dfrac{\dfrac{\dfrac{(AxK)\ \overline{A_0 \vdash \texttt{Cons} :: \alpha_1 \to [\alpha_1] \to [\alpha_1], \emptyset}, \quad (AxK)\ \overline{A_0 \vdash \texttt{True} :: \texttt{Bool}, \emptyset}}{(RApp)\ \overline{A_0 \vdash (\texttt{Cons True}) :: \alpha_2, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \texttt{Bool} \to \alpha_2\}}}, \quad (AxK)\ \overline{A_0 \vdash \texttt{Nil} :: [\alpha_3], \emptyset}}{(RApp)\ A_0 \vdash (\texttt{Cons True Nil}) :: \alpha_4, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \texttt{Bool} \to \alpha_2, \alpha_2 \doteq [\alpha_3] \to \alpha_4\}}$$

Um den Typ von `(Cons True Nil)` *zu berechnen muss man noch unifizieren. Das ergibt:*

$$\sigma = \{\alpha_1 \mapsto \texttt{Bool}, \alpha_2 \mapsto [\texttt{Bool}] \to [\texttt{Bool}], \alpha_3 \mapsto \texttt{Bool}, \alpha_4 \mapsto [\texttt{Bool}]\}$$

und daher $(\texttt{Cons True Nil}) :: \sigma(\alpha_4) = [\texttt{Bool}]$.

Beispiel 5.3.4. *Der Ausdruck* $\Omega := (\lambda x.(x\ x))\ (\lambda y.(y\ y))$ *ist* nicht *wohl-getypt: Die Anfangsannahme ist leer, da keine Konstruktoren oder Superkombinatoren vorkommen.*

$$\dfrac{\dfrac{\dfrac{(AxV)\ \overline{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}, \quad (AxV)\ \overline{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}}{(RApp)\ \overline{\{x :: \alpha_2\} \vdash (x\ x) :: \alpha_4, \{\alpha_2 \doteq \alpha_2 \to \alpha_4\}}}}{(RAbs)\ \overline{\emptyset \vdash (\lambda x.(x\ x)) :: \alpha_2 \to \alpha_4, \{\alpha_2 \doteq \alpha_2 \to \alpha_4\}}}, \quad \dfrac{\dfrac{(AxV)\ \overline{\{y :: \alpha_3\} \vdash y :: \alpha_3, \emptyset}, \quad (AxV)\ \overline{\{y :: \alpha_3\} \vdash y :: \alpha_3, \emptyset}}{(RApp)\ \overline{\{y :: \alpha_3\} \vdash (y\ y) :: \alpha_5, \{\alpha_3 \doteq \alpha_3 \to \alpha_5\}}}}{(RAbs)\ \overline{\emptyset \vdash (\lambda y.(y\ y)) :: \alpha_3 \to \alpha_5, \{\alpha_3 \doteq \alpha_3 \to \alpha_5\}}}}{(RApp)\ \emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, \{\alpha_2 \doteq \alpha_2 \to \alpha_4, \alpha_3 \doteq \alpha_3 \to \alpha_5, \alpha_2 \to \alpha_4 \doteq (\alpha_3 \to \alpha_5) \to \alpha_1\}}$$

Die Unifikation schlägt jedoch fehl:

$$OccursCheck\ \frac{\{\alpha_2 \doteq \alpha_2 \to \alpha_4, \ldots\}}{Fail}$$

Übungsaufgabe 5.3.5. *Zeige, dass der Fixpunktkombinator*

$$Y := \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

nicht typisierbar ist.

Beispiel 5.3.6. *Seien* `map` *und* `length` *bereits typisierte Superkombinatoren. Wir typisieren den Ausdruck*

$$t := \lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{map}\ \texttt{length}\ ys\}$$

Als Anfangsannahme benutzen wir:

$$\begin{aligned} A_0 = \{&\texttt{map} :: \forall a, b.(a \to b) \to [a] \to [b], \\ &\texttt{length} :: \forall a.[a] \to \texttt{Int}, \\ &\texttt{Nil} :: \forall a.[a] \\ &\texttt{Cons} :: \forall a.a \to [a] \to [a] \\ \}& \end{aligned}$$

Zur Typisierung fängt man im Herleitungsbaum von unten an, und arbeitet sich dann nach oben, d.h. die unterste Regel ist (RAbs)*, da t eine Abstraktion ist:*

$$(RA_{BS})\ \frac{A_0 \cup \{xs :: \alpha_1\} \vdash \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to 1; (\texttt{Cons}\ y\ ys) \to \texttt{map}\ \texttt{length}\ ys\} :: \tau, E}{A_0 \vdash \alpha_1 \to \tau, E}$$

Zu diesem Zeitpunkt kennt man den Typ τ und das Gleichungssystem E noch nicht. Diese ergeben sich beim Berechnen. Der nächste Schritt ist nun die Voraussetzung zu zeigen, wofür man die (RCase)*-Regel anwenden muss usw.*

Aus Platzgründen geben wir den Herleitungsbaum und seine Beschriftung getrennt an:

$$\dfrac{\dfrac{\overline{B_3}^{(AxV)},\ \overline{B_4}^{(AxK)},\ \dfrac{\overline{B_8}^{(AxK)},\ \overline{B_9}^{(AxV)}}{B_6}^{(RApp)},\ \overline{B_7}^{(AxV)}}{B_5}^{(RApp)},\ \overline{B_{10}}^{(AxK)},\ \dfrac{\overline{B_{14}}^{(AxSK)},\ \overline{B_{15}}^{(AxSK)}}{B_{12}}^{(RApp)},\ \overline{B_{13}}^{(AxV)}}{B_{11}}^{(RApp)}}{B_2}^{(RCase)}}{B_1}^{(RAbs)}$$

Die Beschriftungen sind dabei:

$$
\begin{aligned}
B_1 =& A_0 \vdash t :: \alpha_1 \to \alpha_{13},\\
&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\
&(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\
&\alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}\\
B_2 =& A_0 \cup \{xs :: \alpha_1\} \vdash\\
&\quad \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{map}\ \texttt{length}\ ys\} :: \alpha_{13},\\
&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\
&(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\
&\alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}\\
B_3 =& A_0 \cup \{xs :: \alpha_1\} \vdash\ xs :: \alpha_1, \emptyset\\
B_4 =& A_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: [\alpha_2], \emptyset\\
B_5 =& A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{Cons}\ y\ ys) :: \alpha_7,\\
&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7\}\\
B_6 =& A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{Cons}\ y) :: \alpha_6,\\
&\quad \{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6\}\\
B_7 =& A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset\\
B_8 =& A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset\\
B_9 =& A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash y :: \alpha_3, \emptyset\\
B_{10} =& A_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: [\alpha_{14}], \emptyset\\
B_{11} =& A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{map}\ \texttt{length})\ ys :: \alpha_{12},\\
&\quad \{(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12}\}\\
B_{12} =& A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{map}\ \texttt{length}) :: \alpha_{11},\\
&\quad \{(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}\}\\
B_{13} =& A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset\\
B_{14} =& A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{map} :: (\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9], \emptyset\\
B_{15} =& A_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{length} :: [\alpha_{10}] \to \texttt{Int}, \emptyset
\end{aligned}
$$

Um den Typ von t zu berechnen müssen wir das Gleichungssystem

$$\begin{aligned}&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\ &(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\ &\alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}\}\end{aligned}$$

noch unifizieren. Man erhält dann als Lösung die Substitution:

$$\begin{aligned}\sigma \quad = \quad &\{\alpha_1 \mapsto [[\alpha_{10}]], \alpha_2 \mapsto [\alpha_{10}], \alpha_3 \mapsto [\alpha_{10}], \alpha_4 \mapsto [[\alpha_{10}]], \alpha_5 \mapsto [\alpha_{10}],\\ &\alpha_6 \mapsto [[\alpha_{10}]] \to [[\alpha_{10}]], \alpha_7 \mapsto [[\alpha_{10}]], \alpha_8 \mapsto [\alpha_{10}], \alpha_9 \mapsto \texttt{Int},\\ &\alpha_{11} \mapsto [[\alpha_{10}]] \to [\texttt{Int}], \alpha_{12} \mapsto [\texttt{Int}], \alpha_{13} \mapsto [\texttt{Int}], \alpha_{14} \mapsto \texttt{Int}\}\end{aligned}$$

Damit erhält man $t :: \sigma(\alpha_1 \to \alpha_{13}) = [[\alpha_{10}]] \to [\texttt{Int}]$.

Beispiel 5.3.7. *Die Funktion (bzw. in* KFPTSP+seq *der Superkombinator)* `const` *ist definiert als*

```
const :: a -> b -> a
const x y = x
```

Der Ausdruck $\lambda x.\texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'})$ *ist nicht wohl-getypt: Sei* $A_0 = \{\texttt{const} :: \forall a, b.a \to b \to a, \texttt{True} :: \texttt{Bool}, \texttt{'A'} :: \texttt{Char}\}$.

Sei $A_1 = A_0 \cup \{x :: \alpha_1\}$. *Die Typisierung ergibt:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_1 \vdash \texttt{const} :: \alpha_2 \to \alpha_3 \to \alpha_2, \emptyset}(\textit{AxK}) \quad \dfrac{\dfrac{}{A_1 \vdash x :: \alpha_1,}(\textit{AxV}) \quad \dfrac{}{A_1 \vdash \texttt{True} :: \texttt{Bool}}(\textit{AxK})}{A_1 \vdash (x\ \texttt{True}) :: \alpha_4, E_1}(\textit{RApp})}{A_1 \vdash \texttt{const}\ (x\ \texttt{True}) :: \alpha_5, E_2}(\textit{RApp}) \quad , \quad \dfrac{\dfrac{}{A_1 \vdash x :: \alpha_1,}(\textit{AxV}) \quad \dfrac{}{A_1 \vdash \texttt{'A'} :: \texttt{Char}}(\textit{AxK})}{A_1 \vdash (x\ \texttt{'A'}) :: \alpha_6, E_3}(\textit{RApp})}{A_1 \vdash \texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'}) :: \alpha_7, E_4}(\textit{RApp})}{A_0 \vdash \lambda x.\texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'}) :: \alpha_1 \to \alpha_7, E_4}(\textit{RAbs})$$

wobei:

$$\begin{aligned}E_1 \quad &= \quad \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4\}\\ E_2 \quad &= \quad \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4, \alpha_2 \to \alpha_3 \to \alpha_2 \doteq \alpha_4 \to \alpha_5\}\\ E_3 \quad &= \quad \{\alpha_1 \doteq \texttt{Char} \to \alpha_6\}\\ E_4 \quad &= \quad \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4, \alpha_2 \to \alpha_3 \to \alpha_2 \doteq \alpha_4 \to \alpha_5, \alpha_1 \doteq \texttt{Char} \to \alpha_6,\\ &\qquad \alpha_5 \doteq \alpha_6 \to \alpha_7\}\end{aligned}$$

Die Unifikation schlägt fehl, da die beiden Gleichungen für α_1 *nicht zueinander passen.*

Auch in Haskell erhält man den entsprechenden Typfehler:

```
Main> \x -> const (x True)  (x 'A')
```

```
<interactive>:1:23:
Couldn't match expected type ‘Char’ against inferred type ‘Bool’
      Expected type: Char -> b
      Inferred type: Bool -> a
 In the second argument of ‘const’, namely ‘(x 'A')’
 In the expression: const (x True) (x 'A')
```

Das letzte Beispiel verdeutlicht, dass das Typisierungsverfahren Lambda-gebundene Variablen quasi monomorph behandelt: Für eine Abstraktion $\lambda x.s$ darf x im Rumpf nur mit dem gleichen Typ verwendet werden. Beachte: Für den Ausdruck $\lambda x.\texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'})$ gibt es durchaus sinnvolle Argumente, z.B. eine Funktion die ihr Argument ignoriert und konstant auf einen Wert abbildet. Da Lambda-gebundene Variablen nur monomorph typisiert werden dürfen (und Patternvariablen in `case`-Alternativen genauso) spricht man auch von `let`-*Polymorphismus*, da nur `let`-gebundene Variablen (die in KFPTSP+seq nur als Superkombinatoren auftreten) wirklich polymorph typisiert werden.

Würde man versuchen, das Typsystem dahingehend anzupassen, dass obiger Ausdruck typisierbar ist, so müsste der Typ für die Variable x in etwa aussagen: Lasse nur Ausdrücke zu, die Funktionen sind, die für beliebigen Argumenttypen den gleichen Ergebnistyp liefern. Unser Typsystem kann eine solche Bedingung nicht ausdrücken, da sie nicht zu unserer Mengensemantik $\texttt{sem}(.)$ passt. Man spricht hierbei auch von *prädikativem* Polymorphismus, da Typvariablen nur für Grundtypen stehen, aber nicht selbst für polymorphe Typen (wie dies in so genanntem imprädikativen Polymorphismus der Fall ist).

In erweiterten Typsystemen (mit imprädikativem Polymorphismus) kann man obigen Ausdruck typisieren als

```
(\x -> const (x True) (x 'A'))::(forall b.(forall a. a -> b) -> b)
```

Beachte, dass der innere Allquantor in unserer Typsyntax nicht erlaubt ist. Die Typisierung des Ausdrucks funktioniert auch im `ghci`, wenn man Typsystemerweiterungen anschaltet. Beachte: Lässt man solche Erweiterungen zu, erhält der Programmierer mehr Freiheit, die Typinferenz ist im Allgemeinen jedoch nicht mehr möglich, d.h. der Programmierer muss Typen vorgeben.

5.4 Typisierung von nicht-rekursiven Superkombinatoren

Bisher haben wir nur KFPTSP+seq-Ausdrücke typisiert und uns nicht um die Typisierung von Superkombinatoren gekümmert. Wir betrachten in diesem Abschnitt den einfachsten Fall: Wir betrachten Superkombinatoren, die sich selbst nicht im Rumpf aufrufen und auch nicht verschränkt rekursiv sind.

Sei $\mathcal{SK}$ eine Menge von Superkombinatoren. Für $SK_i, SK_j \in \mathcal{SK}$ sei $SK_i \preceq SK_j$ genau dann, wenn SK_j den Superkombinator SK_i im Rumpf benutzt. Sei $\preceq^+$ der transitive Abschluss von $\preceq$ ($\preceq^*$ der reflexiv-transitive Abschluss).

Ein Superkombinator SK_i ist direkt rekursiv, wenn $SK_i \preceq SK_i$ gilt. Er ist rekursiv, wenn $SK_i \preceq^+ SK_i$ gilt.

Eine Teilmenge $\{SK_1, \ldots, SK_m\} \subseteq \mathcal{SK}$ von Superkombinatoren ist verschränkt rekursiv, wenn $SK_i \preceq^+ SK_j$ für alle $i, j \in \{1, \ldots, m\}$ gilt.

Nicht-rekursive Superkombinatoren kann man analog zu Abstraktionen typisieren (andere im Rumpf des Superkombinators benutzte Superkombinatoren müssen bereits getypt sein, damit deren Typ in der Annahme verfügbar ist). Wir benutzen die Notation $A \models SK :: \tau$, wenn man für den Superkombinator SK den Typ τ unter der Annahme A herleiten kann[4]. Die Regel für nicht rekursive Superkombinatoren ist:

$$\text{(RSK1)}\ \frac{A \cup \{x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{A \models SK :: \forall \mathcal{X}.\sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)}$$

wenn σ Lösung von E, $SK\ x_1\ \ldots\ x_n = s$ die Definition von SK
und SK nicht rekursiv ist
und $\mathcal{X}$ die Typvariablen in $\sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)$ sind

Beispiel 5.4.1. *Wir betrachten die Typisierung des Kompositionsoperators in Haskell. Er ist definiert als*

```
(.) f g x = f (g x)
```

Die Anfangsannahme A_0 ist leer, da `(.)` *im Rumpf keine Konstruktoren und keine anderen Superkombinatoren benutzt. Sei $A_1 = \{f :: \alpha_1, g :: \alpha_2, x :: \alpha_3\}$. Die Typherleitung für* `(.)` *ist:*

[4] Im Gegensatz zur Notation $A \vdash s :: \tau, E$ hat die Notation $A \models SK :: \tau$ kein Gleichungssystem als Komponente

$$\dfrac{\dfrac{\dfrac{}{A_1 \vdash f :: \alpha_1, \emptyset,}(AxV) \quad \dfrac{\dfrac{}{A_1 \vdash g :: \alpha_2, \emptyset,}(AxV) \quad \dfrac{}{A_1 \vdash x :: \alpha_3, \emptyset}(AxV)}{A_1 \vdash (g\ x) :: \alpha_5, \{\alpha_2 \doteq \alpha_3 \to \alpha_5\}}(RAPP)}{A_1 \vdash (f\ (g\ x)) :: \alpha_4, \{\alpha_2 \doteq \alpha_3 \to \alpha_5, \alpha_1 = \alpha_5 \to \alpha_4\}}(RAPP)}{\emptyset \models (.) :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4) = (\alpha_5 \to \alpha_4) \to (\alpha_3 \to \alpha_5) \to \alpha_3 \to \alpha_4}(RSK1)$$

Hierbei ist $\sigma = \{\alpha_2 \mapsto \alpha_3 \to \alpha_5, \alpha_1 \mapsto \alpha_5 \to \alpha_4\}$. Den erhaltenen Typ darf man nun allquantifizieren (und umbenennen) und anschließend zur Typisierung von Superkombinatoren benutzen, die `(.)` verwenden. Z.B.

$$(.) :: \forall a, b, c.(a \to b) \to (c \to a) \to c \to b$$

5.5 Typisierung rekursiver Superkombinatoren

Für die Typisierung rekursiver Superkombinatoren, bzw. einer Gruppe von verschränkt rekursiven Superkombinatoren ergibt sich zunächst das Problem, dass man beim Typisieren des Rumpfs einer entsprechenden Superkombinatordefinition den Typ des gerade eben zu typisierenden Superkombinators nicht kennt. Deshalb fängt man mit den allgemeinsten Annahmen an und versucht diese so einzuschränken, bis man eine so genannte *konsistente Annahme* erreicht hat. Wir werden zwei Verfahren erörtern.

5.5.1 Das iterative Typisierungsverfahren

Das iterative Typisierungsverfahren fängt mit allgemeinsten Annahmen an, und iteriert den Typcheck (und berechnet in jedem Iterationsschritt eine neue Annahme) so lange bis die Annahme konsistent ist, d.h. aus der Annahme mittels Typisierung genau die gleiche Annahme folgt. Die Regel für die Typisierung eines Superkombinators ist analog zur Regel für nicht-rekursive Superkombinatoren:

$$(\text{SKR}_{\text{EK}})\ \frac{A \cup \{x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{A \models SK :: \sigma(\alpha_1 \to \ldots \alpha_n \to \tau)}$$

wenn $SK\ x_1\ \ldots\ x_n = s$ die Definition von SK, σ Lösung von E

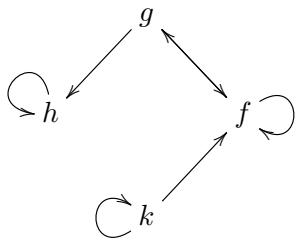
Der Unterschied liegt darin, dass die Annahme A bereits einen Typ für SK (bzw. alle zugehörigen verschränkt rekursiven Superkombinatoren) enthält (siehe Algorithmus 5.5.1).

Global gesehen wird man für eine Menge von Superkombinatordefinitionen zunächst eine Abhängigkeitsanalyse durchführen, die im Aufrufgraph die starken Zusammenhangskomponenten berechnet. Genauer: Sei $\simeq$ die Äquivalenzrelation passend zu $\preceq^*$, dann sind die starken Zusammenhangskomponenten gerade die Äquivalenzklassen zu $\simeq$. Anschließend fängt man mit den kleinsten Elementen bzgl. der Ordnung auf $\simeq$ an (diese hängen nicht von anderen Superkombinatoren ab) und arbeitet sich entsprechend der Abhängigkeiten voran. Wichtig ist, dass man stets eine Gruppe von verschränkt rekursiven Superkombinatoren (d.h. alle Superkombinatoren einer Äquivalenzklasse zu $\simeq$) während der Typisierung gemeinsam behandeln muss.

Betrachte z.B. die Superkombinatoren:

```
f x y = if x<=1 then y else f (x-y) (y + g x)
g x   = if x==0 then (f 1 x) + (h 2) else 10
h x   = if x==1 then 0 else h (x-1)
k x y = if x==1 then y else k (x-1) (y+(f x y))
```

Der Aufrufgraph ist



Die Äquivalenzklassen (mit Ordnung) sind $\{h\} \preceq^+ \{f, g\} \preceq^+ \{k\}$. Man typisiert daher zuerst h alleine, dann f und g zusammen und schließlich zuletzt k.

Das *iterative Typisierungsverfahren* geht wie folgt vor:

Algorithmus 5.5.1 (Iterativer Typisierungsalgorithmus)**.** *Als Eingabe erhält man eine Menge von verschränkt rekursiven Superkombinatoren $SK_1, \ldots, SK_m$ wobei alle anderen in den Definitionen verwendeten Superkombinatoren bereits typisiert sind.*

1. *Die Anfangsannahme A besteht aus den Typen der Konstruktoren und den Typen der bereits bekannten Superkombinatoren.*
2. $A_0 := A \cup \{SK_1 :: \forall\alpha_1.\alpha_1, \ldots, SK_m :: \forall\alpha_m.\alpha_m\}$ *und* $j = 0$.
3. *Verwende für jeden Superkombinator SK_i (mit $i = 1, \ldots, m$) die Regel (SKRek) und Annahme A_j, um SK_i zu typisieren.*

4. *Wenn die m Typisierungen erfolgreich waren, d.h.*

$$A_j \models SK_1 :: \tau_1, \ldots, A_j \models SK_m :: \tau_m.$$

Dann allquantifiziere die Typen τ_i indem alle Typvariablen quantifiziert werden. O.B.d.A ergebe dies $SK_1 :: \forall \mathcal{X}_1.\tau_1, \ldots, SK_m :: \forall \mathcal{X}_m.\tau_m$. Sei

$$A_{j+1} := A \cup \{SK_1 :: \forall \mathcal{X}_1.\tau_1, \ldots, SK_m :: \forall \mathcal{X}_m.\tau_m\}$$

5. *Wenn $A_j \neq A_{j+1}$ (wobei Gleichheit auf Typen die Gleichheit bis auf Umbenennung meint), dann gehe mit $j := j + 1$ zu Schritt (3). Anderenfalls ($A_j = A_{j+1}$) war A_j konsistent, die Typen der SK_i sind entsprechend in A_j zu finden.*

Sollte irgendwann ein `Fail` *in der Unifikation auftreten, dann sind $SK_1, \ldots, SK_m$ nicht typisierbar.*

Beachte: Die Annahme am Anfang in Schritt (2) ist die *allgemeinste* die man treffen kann, denn der polymorphe Typ $\forall a.a$ steht für jeden beliebigen Typen (`sem`(a) ist die Menge aller Grundtypen).

Folgende Eigenschaften gelten für den Algorithmus (wir verzichten auf einen Beweis):

- Die berechneten Typen pro Iterationsschritt sind eindeutig bis auf Umbenennung. Daher folgt: Wenn das Verfahren terminiert, sind die berechneten Typen der Superkombinatoren eindeutig.

- In jedem Schritt werden die neu berechneten Typ spezieller (oder bleiben unverändert). D.h. bzgl. der Grundtypensemantik ist das Verfahren monoton (die Mengen von Grundtypen werden immer kleiner).

- Wenn das Verfahren nicht terminiert (endlos läuft), dann gibt es keinen polymorphen Typ für die zu typisierenden Superkombinatoren. Das folgt, da die Typen dann immer spezieller werden (und daher die Mengen der Grundtypensemantik immer kleiner werden) und da man mit den allgemeinsten Annahmen angefangen hat. Man muss noch ausschließen, dass der Fixpunkt bzgl der Grundtypensemantik nicht die leere Menge ist: Jeder Iterationsschritt berechnet ja einen polymorphen Typ τ, den man syntaktisch hinschreiben kann. Es ist leicht nachzuweisen, dass man bei syntaktisch gleich großen Typen nur eine endliche

Kette bekommen kann. Also muss eine unendliche Iteration dann irgendwann einen syntaktisch größeren polymorphen Typ erzeugen. Bei nichtterminierendem Iterationsverfahren werden die Typen also syntaktisch immer größer, also kann es keinen Grundtyp geben der in allen polymorphen Typen, der von der Iteration erzeugt wird, enthalten ist.

- Das iterative Verfahren berechnet einen *größten Fixpunkt* (bzgl. der Grundtypensemantik): Man startet mit der Menge aller Grundtypen und schränkt die Menge solange durch iteriertes Typisieren ein, bis sich die Menge nicht mehr verkleinert. D.h. es wird der *allgemeinste* polymorphe Typ berechnet.

5.5.2 Beispiele für die iterative Typisierung und Eigenschaften des Verfahrens

Wir betrachten einige Beispiele, wobei wir meistens nur einen rekursiven Superkombinator (und keine verschränkt rekursiven) betrachten.

Beispiel 5.5.2. *Wir typisieren die Funktion* `length`*, die definiert ist als:*

$$\texttt{length}\ xs = \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ ys\}$$

Die Anfangsannahme für die Konstruktoren und $(+)$ *ist:* $A = \{\texttt{Nil} :: \forall a.[a], (:) :: \forall a.a \to [a] \to [a], 0 :: \texttt{Int}, 1 :: \texttt{Int}, (+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}\}$*. Der iterative Typisierungsalgorithmus nimmt am Anfang (für* $j = 0$*) den allgemeinsten Typ für* `length` *an, d.h.* $A_0 = A \cup \{\texttt{length} :: \forall \alpha.\alpha\}$ *und verwendet nun die (SKREK)-Regel um* `length` *im ersten Iterationsschritt zu typisieren:*

$$\dfrac{\dfrac{\begin{array}{ll}(a) & A_0 \cup \{xs :: \alpha_1\} \vdash xs :: \tau_1, E_1 \\ (b) & A_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: \tau_2, E_2 \\ (c) & A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (y : ys) :: \tau_3, E_3 \\ (d) & A_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \tau_4, E_4 \\ (e) & A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}\} \vdash (1 + \texttt{length}\ ys) :: \tau_5, E_5\end{array}}{A_0 \cup \{xs :: \alpha_1\} \vdash (\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ xs\}) :: \alpha_3,\ E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \dot{=} \tau_2, \tau_1 \dot{=} \tau_3, \alpha_3 \dot{=} \tau_4, \alpha_3 \dot{=} \tau_5\}}\ (RC_{ASE})}{A_0 \models \texttt{length} :: \sigma(\alpha_1 \to \alpha_3)\ \textit{wobei } \sigma \textit{ Lösung von } E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \dot{=} \tau_2, \tau_1 \dot{=} \tau_3, \alpha_3 \dot{=} \tau_4, \alpha_3 \dot{=} \tau_5\}}\ (SKR_{EK})$$

Zur besseren Darstellung zeigen wir die Voraussetzungen (a) *bis* (e) *getrennt (die Typen* $\tau_1, \ldots, \tau_5$ *und die Gleichungen* $E_1, \ldots, E_5$ *werden dabei berechnet):*

$$(a): \frac{}{A_0 \cup \{xs :: \alpha_1\} \vdash xs :: \alpha_1, \emptyset}\,(AxV)$$

D.h $\tau_1 = \alpha_1$ *und* $E_1 = \emptyset$

$$(b): \frac{}{A_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: [\alpha_6], \emptyset}\,(AxK)$$

D.h. $\tau_2 = [\alpha_6]$ *und* $E_2 = \emptyset$

$$(c)\ \frac{\dfrac{\dfrac{}{A_0' \vdash (:) :: \alpha_9 \to [\alpha_9] \to [\alpha_9], \emptyset}\,(AxK) \quad \dfrac{}{A_0' \vdash y :: \alpha_4, \emptyset}\,(AxV)}{A_0' \vdash ((:)\ y) :: \alpha_8, \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8\}}\,(RAPP) \quad \dfrac{}{A_0' \vdash ys :: \alpha_5, \emptyset}\,(AxV)}{A_0' \vdash (y : ys) :: \alpha_7, \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}}\,(RAPP)$$

wobei $A_0 = A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$

D.h. $\tau_3 = \alpha_7$ *und* $E_3 = \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}$

$$(d)\ \frac{}{A_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \texttt{Int}, \emptyset}\,(AxK)$$

D.h. $\tau_4 = \texttt{Int}$ *und* $E_4 = \emptyset$

$$(e)\ \frac{\dfrac{\dfrac{}{A_0' \vdash (+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}, \emptyset}\,(AxK) \quad \dfrac{}{A_0' \vdash 1 :: \texttt{Int}, \emptyset}\,(AxK)}{A_0' \vdash ((+)\ 1) :: \alpha_{11}, \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}\}}\,(RAPP) \quad \dfrac{\dfrac{}{A_0' \vdash \texttt{length} :: \alpha_{13}, \emptyset}\,(AxSK) \quad \dfrac{}{A_0' \vdash (ys) :: \alpha_5, \emptyset}\,(AxV)}{A_0' \vdash (\texttt{length}\ ys) :: \alpha_{12}, \{\alpha_{13} \doteq \alpha_5 \to \alpha_{12}\}}\,(RAPP)}{A_0' \vdash (1 + \texttt{length}\ ys) :: \alpha_{10}, \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}}\,(RAPP)$$

wobei $A_0' = A_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$

D.h. $\tau_5 = \alpha_{10}$ *und*

$E_5 = \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}$

Setzt man die erhaltenen Werte ein, ergibt das für die Anwendung der (SKREK)*-Regel:*

$$\frac{\begin{array}{l} A_0 \cup \{xs :: \alpha_1\} \vdash (\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ xs\}) :: \alpha_3, \\ \quad \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\} \\ \cup\{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\} \\ \cup\{\alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \texttt{Int}, \alpha_3 \doteq \alpha_{10}\} \end{array}}{\begin{array}{c} A_0 \models \texttt{length} :: \sigma(\alpha_1 \to \alpha_3) \\ \textit{wobei } \sigma \textit{ Lösung von} \\ \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7, \\ \texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}, \\ \alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \texttt{Int}, \alpha_3 \doteq \alpha_{10}\} \end{array}}\,(SKREK)$$

Die Unifikation ergibt als Unifikator

$$\{\alpha_1 \mapsto [\alpha_9], \alpha_3 \mapsto \texttt{Int}, \alpha_4 \mapsto \alpha_9, \alpha_5 \mapsto [\alpha_9], \alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto [\alpha_9], \alpha_8 \mapsto [\alpha_9] \to [\alpha_9],$$
$$\alpha_{10} \mapsto \texttt{Int}, \alpha_{11} \mapsto \texttt{Int} \to \texttt{Int}, \alpha_{12} \mapsto \texttt{Int}, \alpha_{13} \mapsto [\alpha_9] \to \texttt{Int}\}$$

Der neue berechnete Typ von `length` *ist daher* $\sigma(\alpha_1 \to \alpha_3\} = [\alpha_9] \to \texttt{Int}$*, und* $A_1 = A \cup \{\texttt{length} :: \forall\alpha.[\alpha] \to \texttt{Int}\}$*. Da* $A_0 \neq A_1$ *muss man mit* A_1 *erneut iterieren.*

Wir führen die erneute Typisierung nicht komplett auf: Die einzige Stelle, wo sich etwas ändert ist im Teil (e)*: statt*

$$(AxSK)\ \frac{}{A_0' \vdash \texttt{length} :: \alpha_{13}, \emptyset}$$

erhält man

$$(AxSK)\ \frac{}{A_1' \vdash \texttt{length} :: [\alpha_{13}] \to \texttt{Int}, \emptyset}$$

Natürlich muss man die Gleichungssysteme entsprechend anpassen und A_0 *durch* A_1 *ersetzen. Die Verwendung der* (SKRek)*-Regel ergibt:*

$$(SKR_{EK})\ \frac{\begin{array}{l} A_1 \cup \{xs :: \alpha_1\} \vdash (\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ xs\}) :: \alpha_3, \\ \quad \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\} \\ \cup\{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, : [\alpha_{13}] \to \texttt{Int} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\} \\ \cup\{\alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \texttt{Int}, \alpha_3 \doteq \alpha_{10}\} \end{array}}{\begin{array}{c} A_1 \models \texttt{length} :: \sigma(\alpha_1 \to \alpha_3) \\ \textit{wobei } \sigma \textit{ Lösung von} \\ \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7, \\ \texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, [\alpha_{13}] \to \texttt{Int} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}, \\ \alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \texttt{Int}, \alpha_3 \doteq \alpha_{10}\} \end{array}}$$

Die Unifikation ergibt für σ*:*

$$\{\alpha_1 \mapsto [\alpha_9], \alpha_3 \mapsto \texttt{Int}, \alpha_4 \mapsto \alpha_9, \alpha_5 \mapsto [\alpha_9], \alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto [\alpha_9], \alpha_8 \mapsto [\alpha_9] \to [\alpha_9],$$
$$\alpha_{10} \mapsto \texttt{Int}, \alpha_{11} \mapsto \texttt{Int} \to \texttt{Int}, \alpha_{12} \mapsto \texttt{Int}, \alpha_{12} \mapsto \texttt{Int}, \alpha_{13} \mapsto \alpha_9\}$$

Daher ergibt dies $\texttt{length} :: \sigma(\alpha_1 \to \alpha_3) = [\alpha_9] \to \texttt{Int}$ *und* $A_2 = A \cup \{\texttt{length} :: \forall\alpha.[\alpha] \to \texttt{Int}\}$*. Da* $A_1 = A_2$ *ist* A_1 *eine konsistente Annahme und* $[\alpha] \to \texttt{Int}$ *der iterative Typ von* `length`*.*

5.5.2.1 Das iterative Verfahren ist allgemeiner als Haskell

Beispiel 5.5.3. *Wir typisieren den Superkombinator* g *mit der Definition*

```
g x = 1 : (g (g 'c'))
```

Die Anfangsannahme ist $A = \{1 :: \texttt{Int}, \texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$. *Der iterative Typisierungsalgorithmus startet mit* $A_0 = A \cup \{\texttt{g} :: \forall \alpha.\alpha\}$ *und typisiert* g *mit der (SKRek)-Regel (sei* $A_0' = A_0 \cup \{x :: \alpha_1\}$*):*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_0' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}(AxK) \quad \dfrac{}{A_0' \vdash 1 :: \texttt{Int}, \emptyset}(AxK)}{A_0' \vdash (\texttt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3}(RApp) \quad \dfrac{\dfrac{}{A_0' \vdash \texttt{g} :: \alpha_6, \emptyset}(AxSK) \quad \dfrac{\dfrac{}{A_0' \vdash \texttt{g} :: \alpha_8, \emptyset}(AxSK) \quad \dfrac{}{A_0' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}(AxK)}{A_0' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7\}}(RApp)}{A_0' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4\}}(RApp)}{A_0' \vdash \texttt{Cons}\ 1\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}(SKRek)$$

$$A_0 \models \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = \alpha_1 \to [\texttt{Int}]$$

wobei $\sigma = \{\alpha_2 \mapsto [\texttt{Int}], \alpha_3 \mapsto [\texttt{Int}] \to [\texttt{Int}], \alpha_4 \mapsto [\texttt{Int}], \alpha_5 \mapsto \texttt{Int}, \alpha_6 \mapsto \alpha_7 \to [\texttt{Int}], \alpha_8 \mapsto \texttt{Char} \to \alpha_7\}$ *die Lösung von* $\{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$ *ist.*

D.h. $A_1 = A \cup \{\texttt{g} :: \forall \alpha.\alpha \to [\texttt{Int}]\}$. *Die nächste Iteration (wir lassen sie an dieser Stelle weg) zeigt, dass* A_1 *konsistent ist. Beachte: Für die Funktion* g *kann Haskell keinen Typ herleiten:*

```
Prelude> let g x = 1:(g(g 'c'))

<interactive>:1:13:
Couldn't match expected type `[t]' against inferred type `Char'
      Expected type: Char -> [t]
      Inferred type: Char -> Char
    In the second argument of `(:)', namely `(g (g 'c'))'
    In the expression: 1 : (g (g 'c'))
```

Aber: Haskell kann den Typ verifizieren, wenn man ihn angibt:

```
let g::a -> [Int]; g x = 1:(g(g 'c'))
Prelude> :t g
g :: a -> [Int]
```

Das liegt daran, dass Haskell dann keine Typinferenz mehr durchführt, sondern nur den Rumpf wie einen Ausdruck mit bekanntem Typ für den Superkombinator g *behandelt.*

5.5.2.2 Das iterative Verfahren benötigt mehrere Iterationen

Beispiel 5.5.4. *Wir typisieren den Superkombinator* g *mit der Definition*

```
g x = x : (g (g 'c'))
```

Die Anfangsannahme ist $A = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$. *Der iterative Typisierungsalgorithmus startet mit* $A_0 = A \cup \{\texttt{g} :: \forall\alpha.\alpha\}$ *und typisiert* g *mit der (SKRek)-Regel (sei* $A'_0 = A_0 \cup \{x :: \alpha_1\}$*):*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A'_0 \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}(AxK) \quad \dfrac{}{A'_0 \vdash x :: \alpha_1, \emptyset}(AxV)}{A'_0 \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}(RApp) \quad \dfrac{\dfrac{}{A'_0 \vdash \texttt{g} :: \alpha_6, \emptyset}(AxSK) \quad \dfrac{\dfrac{}{A'_0 \vdash \texttt{g} :: \alpha_8, \emptyset}(AxSK) \quad \dfrac{}{A'_0 \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}(AxK)}{A'_0 \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7\}}(RApp)}{A'_0 \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4\}}(RApp)}{A'_0 \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}(SKRek)$$

$$A_0 \models \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = \alpha_5 \to [\alpha_5]$$

wobei $\sigma = \{\alpha_1 \mapsto \alpha_5, \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto [\alpha_5], \alpha_6 \mapsto \alpha_7 \to [\alpha_5], \alpha_8 \mapsto \texttt{Char} \to \alpha_7\}$ *die Lösung von* $\{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$ *ist.*

D.h. $A_1 = A \cup \{\texttt{g} :: \forall\alpha.\alpha \to [\alpha]\}$. *Da* $A_0 \neq A_1$ *muss eine weitere Iteration durchgeführt werden: Sei* $A'_1 = A_1 \cup \{x :: \alpha_1\}$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A'_1 \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}(AxK) \quad \dfrac{}{A'_1 \vdash x :: \alpha_1, \emptyset}(AxV)}{A'_1 \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}(RApp) \quad \dfrac{\dfrac{}{A'_1 \vdash \texttt{g} :: \alpha_6 \to [\alpha_6], \emptyset}(AxSK) \quad \dfrac{\dfrac{}{A'_1 \vdash \texttt{g} :: \alpha_8 \to [\alpha_8], \emptyset}(AxSK) \quad \dfrac{}{A'_1 \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}(AxK)}{A'_1 \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7\}}(RApp)}{A'_1 \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4\}}(RApp)}{A'_1 \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}(SKRek)$$

$$A_1 \models \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = [\texttt{Char}] \to [[\texttt{Char}]]$$

wobei $\sigma = \{\alpha_1 \mapsto [\texttt{Char}], \alpha_2 \mapsto [[\texttt{Char}]], \alpha_3 \mapsto [[\texttt{Char}]] \to [[\texttt{Char}]], \alpha_4 \mapsto [[\texttt{Char}]], \alpha_5 \mapsto [\texttt{Char}], \alpha_6 \mapsto [\texttt{Char}], \alpha_7 \mapsto [\texttt{Char}], \alpha_8 \mapsto \texttt{Char}\}$
die Lösung von $\{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$ *ist.*

Daher ist $A_2 = A \cup \{\texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]]\}$. *Da* $A_1 \neq A_2$ *muss eine weitere Iteration durchgeführt werden: Sei* $A'_2 = A_2 \cup \{x :: \alpha_1\}$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A'_2 \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}(AxK) \quad \dfrac{}{A'_2 \vdash x :: \alpha_1, \emptyset}(AxV)}{A'_2 \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}(RApp) \quad \dfrac{\dfrac{}{A'_2 \vdash \texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]], \emptyset}(AxSK) \quad \dfrac{\dfrac{}{A'_2 \vdash \texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]], \emptyset}(AxSK) \quad \dfrac{}{A'_2 \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}(AxK)}{A'_2 \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7\}}(RApp)}{A'_2 \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4\}}(RApp)}{A'_2 \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}(SKRek)$$

$$A_2 \models \texttt{g} :: \sigma(\alpha_1 \to \alpha_2)$$

wobei σ *die Lösung von*

$\{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$ *ist.*

Aber die Unifikation des letzten Gleichungssystem schlägt fehl. Daher ist g *nicht typisierbar.*

Satz 5.5.5. *Das iterative Typisierungsverfahren benötigt unter Umständen mehrere Iterationen, bis ein Ergebnis (untypisiert / konsistente Annahme) gefunden wurde.*

Beweis. Beispiel 5.5.4 zeigt, dass mehrere Iterationen benötigt werden, um Ungetyptheit festzustellen. Es gibt auch Beispiele, die zeigen, dass mehrere Iterationen nötig sind, um eine konsistente Annahme zu finden (siehe Übungsaufgabe 5.5.6). □

Übungsaufgabe 5.5.6. *Typisiere den Superkombinator* `fix` *mit dem iterativen Typisierungsverfahren, Die Definition von* `fix` *ist:*

```
fix f = f (fix f)
```

Zeige, dass der iterative Typisierungsalgorithmus mehrere Iterationen benötigt, bis eine konsistente Annahme gefunden wird.

5.5.2.3 Das iterative Verfahren muss nicht terminieren

Beispiel 5.5.7. *Seien* `f` *und* `g` *Superkombinatoren mit den Definitionen:*

```
f = [g]
g = [f]
```

Es gilt $\texttt{f} \simeq \texttt{g}$*, d.h. das iterative Verfahren typisiert die beiden Superkombinatoren gemeinsam.*

Die Anfangsannahme ist $A = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} : \forall a.a\}$*. Die iterative Typisierung startet mit* $A_0 = A \cup \{\texttt{f} :: \forall\alpha.\alpha, \texttt{g} :: \forall\alpha.\alpha\}$ *und wendet die (SKREK)-Regel für* `f` *und* `g` *an:*

$$\dfrac{\dfrac{\dfrac{(AxK)\ \dfrac{}{A_0 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset,}\ (AxSK)\ \dfrac{}{A_0 \vdash \texttt{g} :: \alpha_5}}{(RAPP)\ A_0 \vdash (\texttt{Cons g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3\}}\ ,\ (AxK)\ \dfrac{}{A_0 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{(RAPP)\ A_0 \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}{(SKREK)\ \dfrac{A_0 \models \texttt{f} :: \sigma(\alpha_1) = [\alpha_5]}{\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto \alpha_5\} \textit{ ist}\\ \textit{Lösung von } \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}$$

$$\dfrac{\dfrac{\dfrac{(AxK)\ \dfrac{}{A_0 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset,}\ (AxSK)\ \dfrac{}{A_0 \vdash \texttt{f} :: \alpha_5}}{(RAPP)\ A_0 \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3\}}\ ,\ (AxK)\ \dfrac{}{A_0 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{(RAPP)\ A_0 \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}{(SKREK)\ \dfrac{A_0 \models \texttt{g} :: \sigma(\alpha_1) = [\alpha_5]}{\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto \alpha_5\} \textit{ ist}\\ \textit{Lösung von } \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}}$$

Daher ist $A_1 = A \cup \{\texttt{f} :: \forall a.[a], \texttt{g} :: \forall a.[a]\}$. *Da* $A_1 \neq A_0$ *muss man weiter iterieren.*

$$\dfrac{\dfrac{\dfrac{(AxK)\ \overline{\quad}}{A_1 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset,}\ (AxSK)\ \overline{A_1 \vdash \texttt{g} :: [\alpha_5]}}{A_1 \vdash (\texttt{Cons g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3\}}(RAPP)\ ,\ (AxK)\ \overline{A_1 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{A_1 \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}(RAPP)}{A_1 \models \texttt{f} :: \sigma(\alpha_1) = [[\alpha_5]]}(SKREK)$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \to [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$ *ist Lösung von* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{(AxK)\ \overline{\quad}}{A_1 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset,}\ (AxSK)\ \overline{A_1 \vdash \texttt{f} :: [\alpha_5]}}{A_1 \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3\}}(RAPP)\ ,\ (AxK)\ \overline{A_1 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{A_1 \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}(RAPP)}{A_1 \models \texttt{g} :: \sigma(\alpha_1) = [[\alpha_5]]}(SKREK)$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \to [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$ *ist Lösung von* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

Daher ist $A_2 = A \cup \{\texttt{f} :: \forall a.[[a]], \texttt{g} :: \forall a.[[a]]\}$. *Da* $A_2 \neq A_1$ *muss man weiter iterieren. Man kann vermuten, dass dieses Verfahren nicht endet. Wir beweisen es: Sei* $[a]^i$ *die* i*-fache Listenverschachtelung. Durch Induktion zeigen wir, dass* $A_i = A \cup \{\texttt{f} :: \forall a.[a]^i, \texttt{g} :: \forall a.[a]^i\}$. *Die Basis haben wir bereits gezeigt, der Induktionsschritt ist:*

$$\dfrac{\dfrac{\dfrac{(AxK)\ \overline{\quad}}{A_i \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset,}\ (AxSK)\ \overline{A_i \vdash \texttt{g} :: [\alpha_5]^i}}{A_i \vdash (\texttt{Cons g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3\}}(RAPP)\ ,\ (AxK)\ \overline{A_i \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{A_i \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}(RAPP)}{A_i \models \texttt{f} :: \sigma(\alpha_1) = [[\alpha_5]^i]}(SKREK)$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \to [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$ *ist Lösung von* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{(AxK)\ \overline{\quad}}{A_i \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset,}\ (AxSK)\ \overline{A_i \vdash \texttt{f} :: [\alpha_5]^i}}{A_i \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3\}}(RAPP)\ ,\ (AxK)\ \overline{A_i \vdash \texttt{Nil} :: [\alpha_2], \emptyset}}{A_i \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}(RAPP)}{A_i \models \texttt{g} :: \sigma(\alpha_1) = [[\alpha_5]^i]}(SKREK)$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \to [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$ *ist Lösung von* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

D.h. $A_{i+1} = A \cup \{\texttt{f} :: \forall a.[a]^{i+1}, \texttt{g} :: \forall a.[a]^{i+1}\}$.

Satz 5.5.8. *Das iterative Typisierungsverfahren terminiert nicht immer.*

Beweis. Siehe Beispiel 5.5.7. □

Man kann zeigen, dass die iterative Typisierung im Allgemeinen unentscheidbar ist.

Theorem 5.5.9. *Die iterative Typisierung ist unentscheidbar.*

Beweis. Wir führen den Beweis nicht durch. Er folgt aus der Unentscheidbarkeit der so genannten Semi-Unifikation von First-Order Termen. Die entsprechenden Beweise sind in (Kfoury et al., 1990b; Kfoury et al., 1993; Henglein, 1993) zu finden. □

Abschließend zeigen wir, dass die so genannte „Type safety“ für das iterative Verfahren gilt. Diese besteht aus zwei Eigenschaften: Zum einen, dass die Typisierung unter der Reduktion erhalten bleibt (sogenannte „Type Preservation“) und dass getypte geschlossene Ausdrücke reduzibel sind, solange sie keine WHNF sind (das wird i.A. als „Progress Lemma“ bezeichnet).

Lemma 5.5.10. *Sei s ein direkt dynamisch ungetypter* KFPTS+seq*-Ausdruck. Dann kann das iterative Typsystem keinen Typ für s herleiten.*

Beweis. Wenn s direkt dynamisch ungetypt ist, trifft einer der folgenden Fälle zu:

- $s = R[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ Alts]$ und c ist nicht vom Typ T. Wenn die Typisierung den `case`-Ausdruck typisieren will, wird sie Gleichungen hinzufügen, so dass der Typ von $(c\ s_1\ \ldots\ s_n)$ gleich zum Typ der Pattern in den `case`-Alternativen gemacht wird. Da die Typen der Pattern aber zu Typ T gehören und c nicht, wird die Unifikation scheitern.
- $s = R[\texttt{case}_T\ \lambda x.t\ \texttt{of}\ Alts]$: Bei der Typisierung des `case`-Ausdrucks und von $\lambda x.t$ werden für $\lambda x.t$ Gleichungen erzeugt, die Implizieren, dass $\lambda x.t$ einen Funktionstyp erhält. Dieser Typ wird mit den Typen der Pattern in den `case`-Alternativen unifiziert, was scheitern muss.
- $R[(c\ \ s_1\ \ \ldots\ \ s_{\mathrm{ar}(c)})\ \ t]$: Die Typisierung typisiert die Anwendung $((c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ t)$ wie verschachtelte Anwendung $(((c\ s_1)\ \ldots)\ s_{\mathrm{ar}(c)})\ t)$. Hierbei werden Gleichungen hinzugefügt, die implizieren, dass c höchstens $\mathrm{ar}(c)$ Argumente verarbeiten kann. Da die Anwendung ein weiteres Argument hat, wird die Unifikation scheitern.

□

Lemma 5.5.11 (Type Preservation)**.** *Sei s ein wohl-getypter, geschlossener* KFPTS+seq*-Ausdruck (wobei auch die verwendeten Superkombinatoren wohlgetypt sind) und $s \xrightarrow{no} s'$. Dann ist s' wohl-getypt.*

Beweis. Hierfür muss man die einzelnen Fälle einer (β)-, $(SK-\beta)$- und (case)-Reduktion durchgehen. Für die Typherleitung von s kann man aus der Typherleitung einen Typ für jeden Unterterm von s ablesen. Bei der Reduktion werden diese Typen einfach mitkopiert. Man kann dann leicht nachvollziehen, dass eine Typherleitung immer noch möglich ist. □

Genauer: Für einen Grundtyp τ: $t : \tau$ vor der Reduktion $t \to t'$, dann auch $t'\tau$ danach.
D.h. polymorphe Typen können nach Reduktion auch allgemeiner werden.

Aus den letzten beiden Lemmas folgt:

Satz 5.5.12. *Sei s ein wohlgetypter, geschlossener* KFPTSP+seq*-Ausdruck. Dann ist s nicht dynamisch ungetypt.*

Lemma 5.5.13 (Progress Lemma)**.** *Sei s ein wohl-getypter, geschlossener* KFPTS+seq*-Ausdruck. Dann gilt:*

- *s ist eine WHNF, oder*
- *s ist normalordnungsreduzibel, d.h. $s \xrightarrow{no} s'$ für ein s'.*

Beweis. Betrachtet man die Fälle, wann ein geschlossener KFPTS+seq-Ausdruck irreduzibel ist, so erhält man: s ist eine WHNF oder s ist direkt-dynamisch ungetypt. Daher folgt das Lemma. □

Theorem 5.5.14. *Die iterative Typisierung für* KFPTSP+seq *erfüllt die „Type-safety"-Eigenschaft.*

5.5.2.4 Erzwingen der Terminierung: Milner Schritt

Will man die Terminierung des iterativen Typisierungsverfahren nach einigen Schritten erzwingen, so kann man einen so genannten Milner-Schritt durchführen. Beachte: Im nächsten Abschnitt betrachten wir das Milner-Typisierungsverfahren. Der Unterschied zwischen dem iterativen Typisierungsverfahren mit Milner-Schritt und dem Milner-Typisierungsverfahren liegt darin, dass man beim iterativen Verfahren den Milner-Schritt erst nach einigen Iterationen (wenn man nicht mehr weitermachen will) durchführt, das

Milner-Verfahren jedoch sofort einen Milner-Schritt durchführt (und deswegen gar nicht iteriert).

Definition 5.5.15 (Milner-Schritt). *Sei $SK_1, \ldots, SK_m$ eine Gruppe verschränkt rekursiver Superkombinatoren und $A_i \models SK_1 :: \tau_1, \ldots, A_i \models SK_m :: \tau_m$ seien die durch die i. Iteration hergeleiteten Typen für $SK_1, \ldots, SK_m$.*

Führe einen Milner-Schritt wie folgt durch:

Typisiere alle Superkombinatoren $SK_1, \ldots, SK_m$ auf einmal, wobei die Typannahme ist $A_M = A \cup \{SK_1 :: \tau_1, \ldots, SK_m :: \tau_m\}$ (wobei A die Annahmen für Konstruktoren und bereits typisierte Superkombinatoren enthält). Verwende dafür die folgende Regel:

$$(SKREkM)\ \frac{\begin{array}{c}\text{für } i = 1, \ldots, m: \\ A_M \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i', E_i\end{array}}{A_M \models \text{für } i = 1, \ldots, m\ SK_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i')}$$

$$\text{wenn } \sigma \text{ Lösung von } E_1 \cup \ldots \cup E_m \cup \bigcup_{i=1}^{m} \{\tau_i \doteq \alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i'\}$$

$$\begin{array}{rcl} \text{und} \quad SK_1\ x_{1,1}\ \ldots\ x_{1,n_1} & = & s_1 \\ \ldots & & \\ SK_m\ x_{m,1}\ \ldots\ x_{m,n_m} & = & s_m \end{array}$$

$$\text{die Definitionen von } SK_1, \ldots, SK_m \text{ sind}$$

Als zusätzliche Regel muss im Typisierungsverfahren hinzugefügt werden:

$$(AxSK2)\ \frac{}{A \cup \{SK :: \tau\} \vdash SK :: \tau}$$

$$\text{wenn } \tau \text{ nicht allquantifiziert ist}$$

Wir erläutern den Milner-Schritt: In der Annahme A_M werden die zu typisierenden Superkombinatoren *nicht* mit allquantifiziertem Typ verwendet, sondern ohne Allquantor. Die Regel (AxSK2) wird dann benutzt, um einen solchen Typ aus der Annahme herauszuholen. Dabei findet *keine* Umbenennung statt. D.h. es wird nur ein fester Typ für jedes Auftreten eines Superkombinators verwendet und keine Kopien gemacht im Gegensatz zum iterativen Verfahren. Außerdem werden durch (SKREkM)-Regel alle Gleichungssysteme vereint und gemeinsam unifiziert und es werden *neue Gleichungen* hinzugefügt: Für jeden Superkombinator werden die angenommenen Typen mit den hergeleiteten Typen unifiziert.

Daraus folgt: Die neue Annahme, die man durch die (SKREKM)-Regel herleiten kann ist *stets konsistent*, d.h. man braucht keine weitere Iteration durch zu führen.

Beachte, dass im Unterschied zu (SKREK) nun alle Superkombinatoren einer verschränkt rekursiven Gruppe in der Regel (SKREKM) gleichzeitig erfasst werden müssen, um die Unifikation über alle Gleichungssysteme zu ermöglichen.

Der Nachteil bei diesem Verfahren ist zum Einen, dass man nicht weiß, wann man den Milner-Schritt durchführen sollte und zum Anderen, dass eingeschränktere Typen hergeleitet werden können, als beim iterativen Verfahren (das sehen wir später). Es kann auch durchaus passieren, dass man durch weitere Iteration einen Typ herleiten hätte können, der Milner-Schritt jedoch einen Fehler bei der Unifikation fest stellt und den Superkombinator als ungetypt zurückweist.

5.5.3 Das Milner-Typisierungsverfahren

Das Typisierungsverfahren von Milner geht eigentlich auf drei Autoren zurück: Robin Milner, Luis Damas und Roger Hindley. Je nach Laune findet man daher auch die Bezeichnungen Hindley-Milner oder Damas-Milner Typisierung usw. Wir verwenden „Milner-Typisierung“. Das Verfahren lässt sich einfach beschreiben: Anstelle des iterativen Typisierungsverfahrens mit Milner-Schritt zur Terminierung, wird sofort ein Milner-Schritt durchgeführt:

Algorithmus 5.5.16 (Milner-Typisierungsverfahren)**.** *Seien $SK_1, \ldots, SK_m$ alle Superkombinatoren einer Äquivalenzklasse bzgl. $\simeq$ und seien die Superkombinatoren die kleiner bzgl. dem strikten Anteil der Ordnung $\preceq$ als SK_i sind bereits typisiert.*

1. *In die Typisierungsannahme A werden die allquantifizierten Typen der bereits typisierten Superkombinatoren und der Konstruktoren eingefügt.*

2. *Typisiere $SK_1, \ldots, SK_m$ mit der Regel (MSKREK):*

$$(\mathit{MSKREK})\ \frac{\begin{array}{c}\textit{für } i = 1, \ldots, m\textit{:}\\ A \cup \{SK_1 :: \beta_1, \ldots, SK_m :: \beta_m\}\\ \qquad \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i, E_i\end{array}}{A \models \textit{für } i = 1, \ldots, m\ SK_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i)}$$

$$\textit{wenn } \sigma \textit{ Lösung von } E_1 \cup \ldots \cup E_m \cup \bigcup_{i=1}^{m} \{\beta_i \doteq \alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i\}$$

$$\begin{array}{rcl}\textit{und} \quad SK_1\ x_{1,1}\ \ldots\ x_{1,n_1} & = & s_1\\ \ldots & & \\ SK_m\ x_{m,1}\ \ldots\ x_{m,n_m} & = & s_m\end{array}$$

$$\textit{die Definitionen von } SK_1, \ldots, SK_m \textit{ sind}$$

Schlägt die Unifikation fehl, so sind $SK_1, \ldots, SK_m$ nicht Milner-typisierbar, anderenfalls sind die Milner-Typen von $SK_1, \ldots, SK_m$ genau die Typen die man durch die Regel (MSKREK) erhalten hat. Beachte: Die Regel (AxSK2) wird bei der Typherleitung benötigt.

Um die Regel zu Milner-Typisierung verständlicher zu machen, schreiben wir sie noch einmal für einen einzelnen Superkombinator auf (d.h. wir gehen davon aus, dass dieser Superkombinator rekursiv aber nicht verschränkt rekursiv mit anderen Superkombinatoren ist).

$$(\text{MSKREK1})\ \frac{A \cup \{SK :: \beta, x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{A \models SK :: \sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)}$$

$$\text{wenn } \sigma \text{ Lösung von } E \cup \{\beta \doteq \alpha_1 \to \ldots \to \alpha_n \to \tau\}$$

$$\text{und } SK\ x_1\ \ldots\ x_n = s \text{ die Definition von } SK \text{ ist}$$

Wir beschreiben die Regel: Die Typ-Annahme für den Superkombinator ist β, also der allgemeinste Typ, aber dieser ist nicht (im Gegensatz zum iterativen Verfahren) allquantifiziert. Die Folge ist, dass bei der Herleitung alle Vorkommen von SK im Rumpf s mit dem gleichen Typ β (und nicht umbenannten Kopien) typisiert werden (im iterativen Verfahren wird die (AxSK)-Regel benutzt, während im Milner-Verfahren die (AxSK2)-Regel benutzt wird). Als weitere Veränderung (gegenüber dem iterativen Verfahren) wird bei der Unifikation die Gleichung $\beta \doteq \alpha_1 \to \ldots \to \alpha_n \to \tau$ hinzugefügt: Diese erzwingt, dass der angenommene Typ (β) gleich zum hergeleiteten Typ ist.

Für das Milner-Typisierungsverfahren gelten die folgenden Eigenschaften:

- Das Verfahren terminiert.

- Das Verfahren liefert eindeutige Typen (bis auf Umbenennung von Variablen)
- Die Milner-Typisierung ist entscheidbar.
- Das Problem, ob ein Ausdruck Milner-Typisierbar ist, ist DEXPTIME-vollständig (siehe (Mairson, 1990; Kfoury et al., 1990a)).
- Das Verfahren liefert u.U. eingeschränktere Typen als das iterative Verfahren. Insbesondere kann ein Ausdruck iterativ typisierbar, aber nicht Milner-typisierbar sein.

Aufgrund der Entscheidbarkeit wird das Milner-Typverfahren in Haskell und auch anderen funktionalen Programmiersprachen z.B. ML eingesetzt.

Beispiel 5.5.17. *Man benötigt manchmal exponentiell viele Typvariablen:*

```
(let x0 = \z->z in
  (let x1 = (x0,x0) in
    (let x2 = (x1,x1) in
      (let x3 = (x2,x2) in
         (let x4 = (x3,x3) in
            (let x5 = (x4,x4) in
               (let x6 = (x5,x5) in x6)))))
```

Die Anzahl der Typvariablen im Typ ist 2^6. Verallgemeinert man das Beispiel, dann sind es 2^n.

Wir betrachten einige Beispiele zur Milner-Typisierung:

Beispiel 5.5.18. *Wir typisieren den Superkombinator* `map` *mit dem Milner-Verfahren. Die Definition ist*

```
map f xs = case xs of {
            [] -> []
            (y:ys) -> (f y):(map f ys)
           }
```

Die Annahme für die Konstruktoren ist $A = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} :: \forall a.[a]\}$.

Sei $A' = A \cup \{\texttt{map} :: \beta, f :: \alpha_1, xs :: \alpha_2\}$ *und* $A'' = A' \cup \{y : \alpha_3, ys :: \alpha_4\}$.

Die Typisierung beginnend mit der (MSKREK)-Regel ist:

$$
\dfrac{\dfrac{\begin{array}{ll}(a) & A' \vdash xs :: \tau_1, E_1\\ (b) & A' \vdash \texttt{Nil} :: \tau_2, E_2\\ (c) & A'' \vdash (\texttt{Cons}\ y\ ys) :: \tau_3, E_3\\ (d) & A' \vdash \texttt{Nil} :: \tau_4, E_4\\ (e) & A'' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \tau_5, E_5\end{array}}{A' \vdash \texttt{case}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; \texttt{Cons}\ y\ ys \to \texttt{Cons}\ y\ (\texttt{map}\ f\ ys)\} :: \alpha, E}\ (RC\textit{ASE})}{A \models \texttt{map} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha)}\ (MSKR\textit{EK1})
$$

wenn σ Lösung von $E \cup \{\beta \doteq \alpha_1 \to \alpha_2 \to \alpha\}$

wobei $E = E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha \doteq \tau_4, \alpha \doteq \tau_5\}$. Die Typen $\tau_1, \ldots, \tau_5$ und die Gleichungen $E_1, \ldots, E_5$ ergeben sich beim Herleiten der Voraussetzungen $(a), \ldots, (e)$. Wir machen das zur Übersichtlichkeit getrennt:

(a)
$$\dfrac{}{A' \vdash xs :: \alpha_2, \emptyset}\ (AxV)$$
D.h. $\tau_1 = \alpha_2$ und $E_1 = \emptyset$.

(b)
$$\dfrac{}{A' \vdash \texttt{Nil} :: [\alpha_5], \emptyset}\ (AxK)$$
D.h. $\tau_2 = [\alpha_5]$ und $E_2 = \emptyset$

(c)
$$\dfrac{\dfrac{\dfrac{}{A'' \vdash \texttt{Cons} :: \alpha_6 \to [\alpha_6] \to [\alpha_6], \emptyset}\ (AxK), \dfrac{}{A'' \vdash y :: \alpha_3, \emptyset}\ (AxV)}{A'' \vdash (\texttt{Cons}\ y) :: \alpha_7, \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7\}}\ (RA\textit{PP}), \dfrac{}{A'' \vdash ys :: \alpha_4, \emptyset}\ (AxV)}{A'' \vdash (\texttt{Cons}\ y\ ys) :: \alpha_8, \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8\}}\ (RA\textit{PP})$$
D.h. $\tau_3 = \alpha_8$ und $E_3 = \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8\}$

(d)
$$\dfrac{}{A' \vdash \texttt{Nil} :: [\alpha_9], \emptyset}\ (AxK)$$
D.h. $\tau_4 = [\alpha_9]$ und $E_4 = \emptyset$.

(e)

$$\dfrac{\dfrac{\dfrac{}{A'' \vdash \texttt{Cons} :: \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}], \emptyset}\ (AxK), \dfrac{\dfrac{}{A'' \vdash f :: \alpha_1, \emptyset}\ (AxV), \dfrac{}{A'' \vdash y :: \alpha_3, \emptyset}\ (AxV)}{A'' \vdash (f\ y) :: \alpha_{15}, \{\alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}\ (RA\textit{PP})}{A'' \vdash (\texttt{Cons}\ (f\ y)) :: \alpha_{11}, \{\alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}\ (RA\textit{PP}), \dfrac{\dfrac{}{A'' \vdash \texttt{map} :: \beta, \emptyset}\ (AxSK_2), \dfrac{}{A'' \vdash f :: \alpha_1, \emptyset}\ (AxV)}{A'' \vdash (\texttt{map}\ f) :: \alpha_{12}, \{\beta \doteq \alpha_1 \to \alpha_{12}\}}\ (RA\textit{PP}), \dfrac{}{A'' \vdash ys :: \alpha_4, \emptyset}\ (AxV)}{A'' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \alpha_{14}, \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}, \beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}}\ (RA\textit{PP})$$

D.h. $\tau_5 = \alpha_{14}$ und
$E_5 = \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15},$
$\beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}$

Insgesamt müssen wir das Gleichungssystem $E \cup \{\beta \doteq \alpha_1 \to \alpha_2 \to \alpha\}$ durch

Unifikation lösen, d.h.

$$\{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8, \alpha_{11} \doteq \alpha_{13} \to \alpha_{14},$$
$$\alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}, \beta \doteq \alpha_1 \to \alpha_{12},$$
$$\alpha_{12} \doteq \alpha_4 \to \alpha_{13}, \alpha_2 \doteq [\alpha_5], \alpha_2 \doteq \alpha_8, \alpha \doteq [\alpha_9], \alpha \doteq \alpha_{14}\}$$

Die Unifikation ergibt

$$\sigma = \{\alpha \mapsto [\alpha_{10}], \alpha_1 \mapsto \alpha_6 \to \alpha_{10}, \alpha_2 \mapsto [\alpha_6], \alpha_3 \mapsto \alpha_6, \alpha_4 \mapsto [\alpha_6], \alpha_5 \mapsto \alpha_6,$$
$$\alpha_7 \mapsto [\alpha_6] \to [\alpha_6], \alpha_8 \mapsto [\alpha_6], \alpha_9 \mapsto \alpha_{10}, \alpha_{11} \mapsto [\alpha_{10}] \to [\alpha_{10}],$$
$$\alpha_{12} \mapsto [\alpha_6] \to [\alpha_{10}], \alpha_{13} \mapsto [\alpha_{10}], \alpha_{14} \mapsto [\alpha_{10}], \alpha_{15} \mapsto \alpha_{10},$$
$$\beta \mapsto (\alpha_6 \to \alpha_{10}) \to [\alpha_6] \to [\alpha_{10}],$$

D.h. $map :: \sigma(\alpha_1 \to \alpha_2 \to \alpha) = (\alpha_6 \to \alpha_{10}) \to [\alpha_6] \to [\alpha_{10}]$.

Beispiel 5.5.19. *Wir betrachten Beispiel 5.5.4 erneut und führen die Milner Typisierung dafür durch. Der Superkombinator* `g` *sei definiert als*

```
g x = x : (g (g 'c'))
```

Die Anfangsannahme ist $A = \{$`Cons` $:: \forall a.a \to [a] \to [a],$ `'c'` $::$ `Char`$\}$. *Sei* $A' = A \cup \{x :: \alpha, \mathtt{g} :: \beta\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}(AxK) \quad \dfrac{}{A' \vdash x :: \alpha, \emptyset}(AxV)}{A' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3}(RApp) \quad \dfrac{\dfrac{}{A' \vdash \mathtt{g} :: \beta, \emptyset}(AxSK_2) \quad \dfrac{\dfrac{}{A' \vdash \mathtt{g} :: \beta, \emptyset}(AxSK_2) \quad \dfrac{}{A' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}(AxK)}{A' \vdash (\mathtt{g}\ \texttt{'c'}) :: \alpha_7, \{\beta \doteq \texttt{Char} \to \alpha_7\}}(RApp)}{A' \vdash (\mathtt{g}\ (\mathtt{g}\ \texttt{'c'})) :: \alpha_4, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4\}}(RApp)}{A' \vdash \texttt{Cons}\ x\ (\mathtt{g}\ (\mathtt{g}\ \texttt{'c'})) :: \alpha_2, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}(MSKRek)$$

$$A \models \mathtt{g} :: \sigma(\alpha \to \alpha_2)$$

wobei σ *die Lösung von*

$\{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2, \beta \doteq \alpha \to \alpha_2\}$ *ist.*

Die Unifikation schlägt jedoch fehl, da `Char` *mit einer Liste unifiziert werden soll. D.h.* `g` *ist nicht Milner-typisierbar.*

Beispiel 5.5.20. *Wir betrachten die Funktion* `g` *aus Beispiel 5.5.3 erneut:*

```
g x = 1 : (g (g 'c'))
```

Sei $A' = A \cup \{x :: \alpha, \mathtt{g} :: \beta\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,(AxK) \quad \dfrac{}{A' \vdash 1 :: \texttt{Int}, \emptyset}\,(AxK)}{A' \vdash (\texttt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3}\,(RApp) \quad , \quad \dfrac{\dfrac{}{A' \vdash \texttt{g} :: \beta, \emptyset}\,(AxSK_2) \quad \dfrac{\dfrac{}{A' \vdash \texttt{g} :: \beta, \emptyset}\,(AxSK_2) \quad \dfrac{}{A' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\,(AxK)}{A' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\beta \doteq \texttt{Char} \to \alpha_7\}}\,(RApp)}{A' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4\}}\,(RApp)}{A' \vdash \texttt{Cons}\ 1\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,(SKRek)$$

$$A \models \texttt{g} :: \sigma(\alpha \to \alpha_2)$$

wobei σ die Lösung von

$\{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2, \beta \doteq \alpha \to \alpha_2\}$ *ist.*

Die Unifikation schlägt fehl, da $[\alpha_5] \doteq \texttt{Char}$ unifiziert werden sollen. Beachte g *ist iterativ typisierbar.*

Als abschließendes Beispiel betrachten wir eine Funktion, die sowohl iterativ als auch Milner-typisierbar ist, aber der iterative Typ ist echt allgemeiner als der Milner-Typ:

Beispiel 5.5.21. *Sei der Datentyp* Baum *definiert als*

```
data Baum a = Leer | Knoten a (Baum a) (Baum a)
```

Die Typen für die Konstruktoren sind Leer $:: \forall a.$Baum a *und* Knoten $:: \forall a.a \to$ Baum a $\to$ Baum a $\to$ Bauma

Wir typisieren die Funktion g*, die definiert ist als*

```
g x y = Knoten True (g x y) (g y x)
```

zunächst mit dem Milner-Verfahren: Seien A die Annahmen für die Konstruktoren und sei $A' = A \cup \{x :: \alpha_1, y :: \alpha_2, \texttt{g} :: \beta\}$. Wir zerlegen die Herleitungsbäume zur Übersichtlichkeit:

$$\dfrac{\dfrac{(c), (a)}{\begin{array}{c}A' \vdash (\texttt{Knoten}\ \texttt{True}\ (\texttt{g}\ x\ y)\ (\texttt{g}\ y\ x)) :: \alpha_3, \\ \{\alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \doteq \texttt{Bool} \to \alpha_9, \\ \beta \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10}, \\ \beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3\}\end{array}}\,(RApp)}{\begin{array}{c}A \models \texttt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3) \\ \textit{wobei } \sigma \textit{ Lösung von} \\ \{\alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \doteq \texttt{Bool} \to \alpha_9, \\ \beta \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10}, \\ \beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3 \beta = \alpha_1 \to \alpha_2 \to \alpha_3\}\end{array}}\,(MSKRek)$$

(*a*)

$$(RAPP)\ \frac{(RAPP)\ \dfrac{(AxSK2)\ \dfrac{}{A' \vdash g :: \beta, \emptyset}, (AxV)\ \dfrac{}{A' \vdash y :: \alpha_2, \emptyset}}{A' \vdash (\mathtt{g}\ y) :: \alpha_4, \{\beta \doteq \alpha_2 \to \alpha_4\}}, (AxV)\ \dfrac{}{A' \vdash x :: \alpha_1, \emptyset}}{(\mathtt{g}\ y\ x)) :: \alpha_5, \{\beta \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5\}}$$

(*b*)

$$(RAPP)\ \frac{(RAPP)\ \dfrac{(AxSK2)\ \dfrac{}{A' \vdash g :: \beta, \emptyset}, (AxV)\ \dfrac{}{A' \vdash x :: \alpha_1, \emptyset}}{A' \vdash (\mathtt{g}\ x) :: \alpha_6, \{\beta \doteq \alpha_1 \to \alpha_6\}}, (AxV)\ \dfrac{}{A' \vdash y :: \alpha_2, \emptyset}}{(\mathtt{g}\ x\ y)) :: \alpha_7, \{\beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}}$$

(*c*)

$$(RAPP)\ \frac{\begin{array}{c}(AxK)\ \dfrac{}{A' \vdash \mathtt{Knoten} :: \alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8, \emptyset}, (AxK)\ \dfrac{}{A' \vdash \mathtt{True} :: \mathtt{Bool}, \emptyset} \\ (RAPP)\ \dfrac{}{\begin{array}{c}A' \vdash (\mathtt{Knoten}\ \mathtt{True}) :: \alpha_9, \\ \{\alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9\}\end{array}}\end{array}}{(b)}, \quad (RAPP)\ \frac{}{\begin{array}{c}A' \vdash (\mathtt{Knoten}\ \mathtt{True}\ (\mathtt{g}\ x\ y)) :: \alpha_{10}, \\ \{\alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9, \\ \alpha_9 \doteq \alpha_7 \to \alpha_{10}, \beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}\end{array}}$$

Die Unifikation ergibt

$$\begin{array}{rcl}\sigma & = & \{\alpha_1 \mapsto \alpha_2, \alpha_3 \mapsto \mathtt{Baum}\ \mathtt{Bool}, \alpha_4 \mapsto \alpha_2 \to \mathtt{Baum}\ \mathtt{Bool}, \\ & & \alpha_5 \mapsto \mathtt{Baum}\ \mathtt{Bool}, \alpha_6 \mapsto \alpha_2 \to \mathtt{Baum}\ \mathtt{Bool}, \alpha_8 \mapsto \mathtt{Bool}, \\ & & \alpha_9 \mapsto \mathtt{Baum}\ \mathtt{Bool} \to \mathtt{Baum}\ \mathtt{Bool} \to \mathtt{Baum}\ \mathtt{Bool}, \alpha_7 \mapsto \mathtt{Baum}\ \mathtt{Bool}, \\ & & \alpha_{10} \mapsto \mathtt{Baum}\ \mathtt{Bool} \to \mathtt{Baum}\ \mathtt{Bool}\}\end{array}$$

D.h. $A \models \mathtt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3) = \alpha_2 \to \alpha_2 \to \mathtt{Baum}\ \mathtt{Bool}$.

Daher ergibt der Milner-Typcheck $\mathtt{g} :: a \to a \to \mathtt{Baum}\ \mathtt{Bool}$.

Als nächstes betrachten wir die iterative Typisierung von g*: Sei* $A_0 = A \cup \{\mathtt{g} :: \forall\alpha.\alpha\}$*, die erste Iteration für* A_0 *ergibt, wobei* $A_0' = A_0 \cup \{x :: \alpha_1, y :: \alpha_2\}$*:*

$$(SKR_{EK})\ \frac{(RA_{PP})\ \dfrac{(c),(a)}{\begin{array}{c}A_0' \vdash (\texttt{Knoten}\ \texttt{True}\ (\texttt{g}\ x\ y)\ (\texttt{g}\ y\ x)) :: \alpha_3,\\ \{\alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \doteq \texttt{Bool} \to \alpha_9,\\ \beta_1 \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10},\\ \beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3\}\end{array}}}{\begin{array}{c}A_0 \models \texttt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3)\\ \textit{wobei } \sigma \textit{ Lösung von}\\ \{\alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \doteq \texttt{Bool} \to \alpha_9,\\ \beta_1 \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10},\\ \beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3\}\end{array}}$$

(a)

$$(RA_{PP})\ \frac{(RA_{PP})\ \dfrac{(AxSK)\ \dfrac{}{A_0' \vdash g :: \beta_1, \emptyset}, (AxV)\ \dfrac{}{A_0' \vdash y :: \alpha_2, \emptyset}}{A_0' \vdash (\texttt{g}\ y) :: \alpha_4, \{\beta_1 \doteq \alpha_2 \to \alpha_4\}}, (AxV)\ \dfrac{}{A_0' \vdash x :: \alpha_1, \emptyset}}{(\texttt{g}\ y\ x)) :: \alpha_5, \{\beta_1 \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5\}}$$

(b)

$$(RA_{PP})\ \frac{(RA_{PP})\ \dfrac{(AxSK)\ \dfrac{}{A_0' \vdash g :: \beta_2, \emptyset}, (AxV)\ \dfrac{}{A_0' \vdash x :: \alpha_1, \emptyset}}{A_0' \vdash (\texttt{g}\ x) :: \alpha_6, \{\beta_2 \doteq \alpha_1 \to \alpha_6\}}, (AxV)\ \dfrac{}{A_0' \vdash y :: \alpha_2, \emptyset}}{(\texttt{g}\ x\ y)) :: \alpha_7, \{\beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}}$$

(c)

$$(RA_{PP})\ \frac{\begin{array}{c}(RA_{PP})\ \dfrac{(AxK)\ \dfrac{}{A_0' \vdash \texttt{Knoten} :: \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8, \emptyset}, (AxK)\ \dfrac{}{A_0' \vdash \texttt{True} :: \texttt{Bool}, \emptyset}}{\begin{array}{c}A_0' \vdash (\texttt{Knoten}\ \texttt{True}) :: \alpha_9,\\ \{\alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \doteq \texttt{Bool} \to \alpha_9\}\end{array}}\\ (b)\end{array}}{\begin{array}{c}A_0' \vdash (\texttt{Knoten}\ \texttt{True}\ (\texttt{g}\ x\ y)) :: \alpha_{10},\\ \{\alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \doteq \texttt{Bool} \to \alpha_9,\\ \alpha_9 \doteq \alpha_7 \to \alpha_{10}, \beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}\end{array}}$$

Die Unifikation ergibt

$$\begin{aligned}\sigma \quad = \quad & \{\alpha_1 \mapsto \alpha_2, \alpha_3 \mapsto \texttt{Baum Bool}, \alpha_4 \mapsto \alpha_2 \to \texttt{Baum Bool},\\ & \alpha_5 \mapsto \texttt{Baum Bool}, \alpha_6 \mapsto \alpha_2 \to \texttt{Baum Bool}, \alpha_8 \mapsto \texttt{Bool},\\ & \alpha_9 \mapsto \texttt{Baum Bool} \to \texttt{Baum Bool} \to \texttt{Baum Bool}, \alpha_7 \mapsto \texttt{Baum Bool},\\ & \alpha_{10} \mapsto \texttt{Baum Bool} \to \texttt{Baum Bool}\}\end{aligned}$$

Die Unifikation ergibt:

$$\begin{aligned}\sigma \quad = \quad & \{\beta_1 \mapsto \alpha_2 \to \alpha_1 \to \texttt{Baum Bool}, \beta_2 \mapsto \alpha_1 \to \alpha_2 \to \texttt{Baum Bool},\\ & \alpha_3 \mapsto \texttt{Baum Bool}, \alpha_4 \mapsto \alpha_1 \to \texttt{Baum Bool}, \alpha_5 \mapsto \texttt{Baum Bool},\\ & \alpha_6 \mapsto \alpha_2 \to \texttt{Baum Bool}, \alpha_7 \mapsto \texttt{Baum Bool}, \alpha_8 \mapsto \texttt{Bool},\\ & \alpha_9 \mapsto \texttt{Baum Bool} \to \texttt{Baum Bool} \to \texttt{Baum Bool}, \alpha_{10} \mapsto \texttt{Baum Bool} \to \texttt{Baum Bool}\}\end{aligned}$$

Das ergibt $\texttt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3) = \alpha_1 \to \alpha_2 \to \texttt{Baum Bool}$ *Daher ist* $A_1 = A \cup \{\texttt{g} :: \forall\alpha, \beta, \alpha \to \beta \to \texttt{Baum Bool}\}$. *Eine weitere Iteration zeigt, dass* A_1 *konsistent ist* (*wir lassen sie weg*).

Insgesamt ergibt das iterative Verfahren $\texttt{g} :: a \to b \to \texttt{Baum Bool}$. *Der iterative Typ ist daher allgemeiner als der Milner-Typ. Im GHC wird der eingeschränktere Typ berechnet:*

```
*Main> let g x y =  Knoten True (g x y) (g y x)
*Main> :t g
g :: t -> t -> Baum Bool
```

Gibt man allerdings den iterativen Typ per Hand vor, dann kann Haskell den Typ verifizieren:

```
*Main> let g :: a -> b -> Baum Bool; g x y =  Knoten True (g x y) (g y x)
*Main> :t g
g :: a -> b -> Baum Bool
```

Das Milner-Verfahren erfüllt die Eigenschaft, dass Milner-getypte Programme niemals dynamisch ungetypt sind. Ebenso gilt das Progress Lemma, d.h. Milner-getypte (geschlossene) Programme sind entweder reduzibel oder WHNFs. Die Type-Preservation Eigenschaft gilt in KFPTSP+seq auch für die Milner-Typisierung. Verwendet man jedoch einen Kalkül mit rekursiven `let`-Ausdrücken, kann es passieren das Milner-getypte Ausdrücke durch Reduktion in Ausdrücke überführt werden, die nicht mehr Milner-getypt sind. Trotzdem sind diese Ausdrücke iterativ getypt (daher kann kein dynamischer Typfehler auftreten).

Das Gegenbeispiel ist:

```
let x = (let y = \u -> z in (y [], y True, seq x True)); z = const z x in x
```

ist Milner-typisierbar. Wenn man eine so genannte $(llet)$-Reduktion durchführt erhält man

```
let x = (y [], y True, seq x True); y = \u -> z; z = const z x in x
```

Dieser Ausdruck ist nicht mehr Milner-typisierbar, da `y` zusammen mit `x` typisiert wird.

Bemerkung 5.5.22. *Die Anmerkungen die unter „Haskells Monomorphism Restriction“ in vorherigen Versionen des Skripts beschrieben waren, sind offenbar in der aktuellen Version von Haskell 7.10.2 nicht mehr gültig. Die Beschränkung ist aufgehoben. Offenbar war es eher ein Implementierungsproblem, das behoben worden ist und kein grundsätzliches.*

5.6 Typisierung unter Typklassen-Beschränkungen

Die iterative und die Milner-Typisierung und auch die benutzte Unifikation der Typen kann man leicht auf Typklassenbeschränkungen erweitern. Die Analyse und die Typisierung der Typklassendefinitionen in Haskell selbst lassen wir außen vor.

Wir betrachten also nur Ausdrücke, wobei Typen von vorkommenden Superkombinatornamen durch Typklassen beschränkt sein können. D.h. alle Typklassenbeschränkungen kommen aus den Annahmen (innerhalb der Typisierung). Typklassen selbst kann man sich als Menge von (Grund-) typen vorstellen.

Um es etwas genauer zu machen, nehmen wir die Sprache KFPTSP und erweitern deren Typsystem:

Definition 5.6.1. *Erweiterung der Kernsprache KFPTSP um Typklassen.*

- *Es gibt eine Menge von Typklassen. Die sind der Einfachheit halber nur als Namen gegeben.*
- *Die Grammatik der Typen wird erweitert zu einem Paar aus* Typklassenconstraint C *und polymorphen Typ* τ*, geschrieben als* C `=>` τ*, wobei ein Typklassenconstraint (oder kurz: Constraint) eine Menge von Ausdrücken der Form* $Cl\ a$ *ist. Hierbei ist* Cl *ein Typklassenname und* a *eine Typvariable.*

  Als Beschränkung geht noch ein, dass alle Typvariablen, die in C *vorkommen, auch in* τ *vorkommen.*

  Wenn die Menge C *der Typklassenconstraints in* C `=>` τ *leer ist, kann man auch einfach nur den Typ hinschreiben.*

- *Es gibt vorgegebene Funktionen (auch Klassenfunktionen) deren Typ schon nichttriviale Typklassenconstraints enthält.*
  (In Haskell gibt es keine Konstruktoren mit Typklassenconstraints.)

□

Man braucht noch Berechnungsmöglichkeiten auf den Bedingungen, die erkennen, ob ein (Grund-) Typ zu einer Typklasse gehört:

Definition 5.6.2. *Jede Typklasse Cl besteht aus einer (meist unendlichen) Menge von Grundtypen: $\texttt{sem}(Cl)$, siehe Definition 5.6.5*

Um die Typisierung vernünftig zu machen, braucht man ein Berechnungsverfahren: Dieses basiert auf einer global vorgegebenen Menge

$$M_{Typklassenaxiome}$$

von Formeln:
aus Fakten und einfachen Implikations-Formeln (analog zu einem Prologprogramm), die in Haskell aus den Klassendefinitionen und Instanzdefinitionen kommen. Folgende Formeln sind möglich:

1. *$Cl\ \tau$ (d.h. $\tau \in Cl$) für Basistypen τ. D.h. für Typkonstruktoren ohne Argumente (nur Namen) ist vorgegeben ob das gilt oder nicht.*
2. *Implikationen der Form $C \implies Cl(TC\ a_1 \ldots a_n)$, wobei $a_1, \ldots, a_n$ verschiedene Typvariablen sind und in C nur Typklassenconstraints der Form $Cl_i\ a_i$ vorkommen.*
   Pro Typkonstruktor TC darf es nur eine solche Implikation geben.
3. *Implikationen der Form $Cl_1\ a \implies Cl_2\ a$. (Hier nehmen wir an, dass das nur für Basistypen benötigt wird.)*

Das algorithmische Problem, ob $Cl\ \tau$ gilt, ist eigentlich dasselbe wie die die Frage, ob ein Baumautomat einen gegebenen Baum akzeptiert oder nicht. Hier gibt es den Unterschied, ob der Baumautomat deterministisch ist oder nicht, und ob er von oben oder von unten den Baum abarbeitet. Zu allgemeine Aussagen dazu, insbesondere Komplexität siehe (Comon et al., 2007). Es gibt in jedem Fall einen polynomiellen Algorithmus der das Problem entscheidet.
In speziellen Fall der Typklassen hat man deterministische Varianten und kann eine einfachen Algorithmus nehmen.

Folgender Algorithmus kann Constraints $Cl\ \tau$ schnell entscheiden:

Definition 5.6.3. *Man nimmt $M_{Typklassenaxiome}$ als gegeben an.*
Man startet mit der Constraint-Menge $\mathcal{C} = \{Cl\ \tau\}$ und vereinfacht die Menge, mit den zwei folgenden Schritten solange, bis die Menge leer ist und somit alle Constraints erfüllt, oder nicht bearbeitbare Constraint bleiben, und somit das Constraint dann falsch ist.

1. *Wähle ein Constraint $Cl\ TC$ aus $\mathcal{C}$: wenn dies gilt, d.h. in $M_{Typklassenaxiome}$ enthalten ist, wobei wir die einfachen Implikation $Cl_1\ a \implies Cl_2\ a$ hierbei mitberücksichtigen, dann entferne das Constraint aus der Menge $\mathcal{C}$.*

2. *Nehme ein Constraint $Cl\ (TC\ \tau_1 \ldots \tau_n)$ aus $\mathcal{C}$ (mit maximaler Größe); wenn es eine Implikation $C_0 \implies TC\ a_1 \ldots a_n$ gibt, dann ersetzt das Constraint in C in $\mathcal{C}$ durch die Menge $\sigma(C_0)$, wobei $\sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$ ist.*

Wenn die Constraint-Menge nicht komplett eliminiert werden kann, ergibt sich insgesamt `False`.

Wenn man die Implikationen vorwärts verwendet, erzeugt man evtl. exponentiell viele Constraints bevor man den richtigen Constraint findet.

Damit kann man Constraints der Form $Cl\ \tau$, wobei τ ein Grundtyp ist, algorithmisch entscheiden:

Satz 5.6.4. *Sei die Menge $M_{Typklassenaxiome}$ gegeben. Bei Eingabe eines Constraints $Cl\ \tau$ kann man in polynomieller Zeit (in der Größe von $M_{Typklassenaxiome}$ und τ) entscheiden, ob $Cl\ \tau$ gilt.*

Beweis. Das Argument ist einfach □

Definition 5.6.5. *Die (Grundtypen-)Semantik eines Typs unter den Constraints kann man so definieren:*

$$sem(C, \tau) = \{\sigma(\tau) \mid \sigma \text{ setzt Grundtypen für Typvariablen ein und für alle Constraints } (TC\ a) \in C : (\sigma(a) \in sem(TC)\}$$

Um ausreichend viele Beispiele zu haben, nehmen wir einfach mal an, dass die Klassenfunktionen zu den Haskelltypklassen wie `Num`, `Ord`, und `Show` schon vorhanden sind.

Als Beispiel die einfache Funktion:
$\lambda x.\lambda y.(x, y, x + y)$.
Auch hier nehmen wir an, dass der Typ von $+$ schon gegeben ist als:

$+ :: \{\texttt{Num}\ a\}$ => $a \to a \to a$.
Dann ergibt sich:
$\lambda x.\lambda y.(x, y, x + y) :: \{\texttt{Num}\ a\}$ => (a, a, a)

5.6.1 Die Typisierung unter Typklassen

Die Änderungen der Unifikation als Teil der Typisierungsalgorithmen sind einfach zu beschreiben:

1. Man startet mit einer Gleichung $s \doteq t$ und eine Menge $\mathcal{C}$ von Typklassenconstraints.
2. Man wendet die Unifikationsregeln auf die Gleichungen an, bis sich am Ende eine Substitution σ ergibt.
3. Man wendet die zwei Schritte des Algorithmus in Def. 5.6.3 auf die Menge solange an, bis sich keine Vereinfachung mehr ergibt. Wenn danach die Constraints der endgültigen Constraintmenge $\mathcal{C}'$ nur noch die Form $(TC\ a)$ haben, dann ist das Verfahren erfolgreich. Ergebnis ist die Substitution σ und das Constraint $\mathcal{C}'$ als $\sigma\mathcal{C}$.

Analog sind die Änderungen bei den Typisierungsverfahren von Ausdrücken, und Superkombinatoren. Sowohl im iterativen als auch im Milnerverfahren. Beim iterativen Verfahren werden nicht nur in jedem Schritt die Typen verfeinert, sondern auch die Constraints.

Beispiel 5.6.6. *Wir nehmen mal an, dass wir die Addition auch auf Paaren definiert haben. In Haskell ist das möglich durch eine instance Erklärung. Ein Basisconstraint ist dann* $(\texttt{Num Int})$. *Eine entsprechende Implikation für Paare ist dann* $\{\texttt{Num}\ a, \texttt{Num}\ b\} \implies \texttt{Num}(a, b)$. *Betrachte die Funktion:*

```
f x = x+(1,2)
```

Die Typisierung ergibt, dass folgendes Unifikationsproblem zu lösen ist: Wenn $x :: a$, *und* $+$ *den Typ* $\texttt{Num}\ b$ => $b \to b \to b$. *Die Gleichungen sind* $a \doteq b, b \doteq (\texttt{Int}, \texttt{Int})$ *und die Constraintmenge ist* $\{\texttt{Num}\ b\}$. *Es ergibt sich* $\sigma = \{a \mapsto (\texttt{Int}, \texttt{Int}), b \mapsto (\texttt{Int}, \texttt{Int})\}$ *als Lösung und die Constraintmenge wird* $\{\texttt{Num}\ (\texttt{Int}, \texttt{Int})\}$. *Aus der Implikation ergibt sich, dass* $\{\texttt{Num Int}, \texttt{Num Int}\}$ *zu lösen ist, was sich aus dem gegebenen Constraint durch das Basisconstraint ergibt und somit ohne Constraint die Lösung* σ *ergibt.*

Wandelt man das Bspl ab durch
`g x y = x+(y,y)`
ergeben sich leicht allgemeinere Typen:

Mit $x :: a, y :: c$ *ergeben sich die Gleichungen* $a \doteq b, b \doteq (c, c)$ *und die Constraintmenge ist* $\{\texttt{Num}\ b\}$. *Es ergibt sich* $\sigma = \{a \mapsto (c, c), b \mapsto (c, c)\}$ *als Lösung und die Constraintmenge wird* $\{\texttt{Num}\ (c, c)\}$. *Aus der Implikation folgt, dass* $\{\texttt{Num}\ c, \texttt{Num}\ c\}$ *sich als Constraint ergibt zusammen mit der Lösung* σ.
Der Typ der Funktion g *ist dann*
$g :: \texttt{Num}\ c \texttt{ => } (c, c) \to c \to (c, c)$.

5.7 Zusammenfassung und Quellennachweis

Wir haben in diesem Kapitel die polymorphe Typisierung für Haskell und für KFPTSP+seq-Ausdrücke erörtert. Typklassen wurden dabei nicht behandelt. Wir haben zwei Verfahren zur polymorphen Typisierung rekursiver Superkombinatoren kennen gelernt und analysiert.

Das Milner-Typisierungsverfahren wurde in (Milner, 1978) und in ähnlicher Form auch schon in (Hindley, 1969) eingeführt. In (Damas & Milner, 1982) wurde die Korrektheit und Vollständigkeit des Verfahrens nachgewiesen. Das iterative Typisierungsverfahren entspricht im Wesentlichen dem Verfahren in (Mycroft, 1984).

6

IO in Haskell und Monadisches Programmieren

6.1 Monadisches Programmieren

Monadisches Programmieren ist (aus Programmiersicht) eine bestimmte Strukturierungsmethode, um *sequentiell* ablaufende Programme in einer funktionalen Programmiersprache zu implementieren. Der Begriff *Monade* stammt aus dem Teilgebiet der Kategorientheorie der Mathematik (Begriffe wie Morphismen, Isomorphismen, ...). Ein Typkonstruktor ist eine Monade, wenn er etwas verpackt und bestimmte Operationen auf dem Datentyp zulässt, wobei die Operationen die sog. *monadischen Gesetze* erfüllen müssen.

In Haskell ist der Begriff der Monade durch eine Typklasse realisiert. D.h. alle Instanzen der Typklasse `Monad` sind Monaden (sofern sie die monadischen Gesetze erfüllen).

Die Typklasse `Monad` ist eine Konstruktorklasse und definiert als

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

  m >> k      = m >>= \_ -> k
  fail s      = error s
```

Es müssen die Operatoren >>= und `return` definiert werden, da für >> und `fail` Default-Implementierungen gegeben sind.

Ein Objekt vom Typ `m a` (wobei `m` und `a` entsprechend instantiiert sind) bezeichnet man als *monadische Aktion*.

Der Operator >>= wird als „bind“ ausgesprochen und verkettet sequentiell zwei monadische Aktionen zu einer. Dabei kann die zweite Aktion das (verpackte) Ergebnis der ersten Aktion verwenden. Die Operation `return` ver-

packt einen beliebigen funktionalen Ausdruck in der Monade. Die Operation >> wird als „then“ bezeichnet und ist ähnlich zum bind, wobei die zweite Aktion das Ergebnis der ersten Aktion nicht verwendet.

Bevor wir auf die monadischen Gesetze eingehen, betrachten wir ein Beispiel für eine Monade: Der Datentyp `Maybe` ist Instanz der Klasse `Monad`. Durch die Monadenimplementierung können sequentiell Berechnungen durchgeführt werden, die entweder ein Ergebnis der Form `Just` x liefern, oder im Falle eines Fehlschlagens `Nothing` liefern. Die Implementierung von >>= sichert dabei zu, dass das Fehlschlagen einer Berechnung innerhalb einer Sequenz auch zum Fehlschlagen der gesamten Sequenz führt. Der Wert `Nothing` wird dann quasi durchgereicht. Die `Monad`-Instanz für `Maybe` ist definiert als:

```
instance Monad Maybe where
    return          = Just
    fail s          = Nothing
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
```

Eine Beispielanwendung ist der Lookup in einer Datenbank. Wir verwenden die Funktion `lookup`, die Listen als Datenbanken verwendet.

Nehmen wir an, wir haben zwei Datenbanken (Tabellen):

- Eine Liste `db`, die Kontonummern und Kontostände beinhaltet, z.B.
  `db = [(101,100),(102,200),(103,-500)]`
- Eine Liste `passdb`, in der passend zur Kontonummer Passwörter gespeichert sind, z.B.
  `passdb = [(101,"KSPW!3"),(102,"Ow23="),(103,"12ko12")]`

Wir möchten eine Funktion `getKontostand` implementieren, die eine Kontonummer und das entsprechende Passwort erwartet und nur dann den aktuellen Kontostand liefert, wenn Kontonummer und Passwort zueinander passen. Mit den monadischen Operationen ist das z.B. wie folgt möglich

```
getKontostand knr pass = lookup knr db >>=
                         \x -> lookup knr passdb >>=
                          \y -> eqMaybe pass y >>
                            return x
eqMaybe a b
 | a == b = Just True
 | otherwise = Nothing
```

Die Funktion `eqMaybe` ist dabei nur eine Hilfsfunktion, die den Gleichheitstest zum für `Maybe`-Typ anpasst. Wenn die beiden Argumente gleich sind, liefert `eqMaybe` als Wert `Just True` zurück, anderenfalls `Nothing`. Die Funktion `getKontostand` führt sequentiell die folgenden Berechnungen durch: Nachschauen in der Datenbank `db`, Nachschauen in der Datenbank `passdb`, Vergleich der Passwörter und Rückgabe des Ergebnisses. Tritt irgendwann dabei `Nothing` auf, so wird insgesamt `Nothing` das Ergebnis sein.

Einige Beispielaufrufe sind:

```
*Main> getKontostand 101 "KSPW!3"
Just 100
*Main> getKontostand 102 "KSPW!3"
Nothing
*Main> getKontostand 10 "KSPW!3"
Nothing
```

Man kann die gleiche Funktionalität auch ohne die monadischen Operatoren implementieren, der Code wird dann allerdings unübersichtlicher, z.B. mit verschachtelten `case`-Ausdrücken:

```
getKontostand knr pass =
 case lookup knr db of
   Nothing -> Nothing
   Just x -> case lookup knr passdb of
               Nothing -> Nothing
               Just y -> case eqMaybe pass y of
                           Nothing -> Nothing
                           Just _  -> Just x
```

Die Verwendung der Monadischen Operatoren ist allerdings auch etwas schwer zu lesen. Deshalb gibt es als syntaktischen Zucker die sogenannte `do`-*Notation*: Eingeleitet wird ein solcher `do`-Block mit dem Schlüsselwort `do`, anschließend folgen monadische Aktionen, wobei als Spezialsyntax `x <- aktion` verwendet werden kann, um auf das Ergebnis der Aktion zuzugreifen. Zudem gibt es spezielle Form von `let`-Ausdrücken, die keinen `in`-Ausdruck haben, also Ausdrücke der Form `let x = e`, diese können in monadischen `do`-Blöcken verwendet werden. Die `do`-Blöcke ähneln der imperativen Programmierung. Die Funktion `getKontostand` kann mit `do` implementiert werden als:

```
getKontostand knr pass =
 do
  x <- lookup knr db
  y <- lookup knr passdb
  eqMaybe pass y
  return x
```

Wie bereits erwähnt, ist die `do`-Notation nur syntaktischer Zucker, sie kann durch die monadischen Operatoren >>= und >> dargestellt werden. Die Übersetzung ist[1]:

```
do { x <- e ; s }   = e >>= \x -> do { s }
do { e; s }         = e >> do { s }
do { e }            = e
do { let binds; s } = let binds in do { s }
```

Verwendet man diese Übersetzung für die Definition von `getKontostand`, so erhält man gerade die ursprüngliche Definition:

```
getKontostand knr pass =
 do
  x <- lookup knr db
  y <- lookup knr passdb
  eqMaybe pass y
  return x
==>
getKontostand knr pass =
  lookup knr db >>=
   \x ->
    do
     y <- lookup knr passdb
     eqMaybe pass y
     return x
==>
getKontostand knr pass =
  lookup knr db >>=
   \x ->
```

[1] Beachte, dass wir hier Semikolon und geschweifte Klammern verwenden, in der Funktion `getKontostand` haben wir dies durch Einrückung vermieden

```
    lookup knr passdb >>=
    \y ->
     do
      eqMaybe pass y
      return x

==>
getKontostand knr pass =
  lookup knr db >>=
   \x ->
    lookup knr passdb >>=
    \y ->
     eqMaybe pass y >>
      do
       return x

==>
getKontostand knr pass =
  lookup knr db >>=
   \x ->
    lookup knr passdb >>=
     \y ->
      eqMaybe pass y >>
       return x
```

6.1.1 Die Monadischen Gesetze

Damit ein Datentyp, der Instanz von `Monad` ist, wirklich eine Monade ist, müssen die folgenden drei Gesetze (d.h. Gleichheiten) gelten:

(1) `return x >>= f` $=$ `f x`

(2) `m >>= return` $=$ `m`

(3) `m1 >>= (\x -> m2 x >>= m3)` $=$ `(m1 >>= m2) >>= m3` wenn $x \notin FV($`m2,m3`$)$

Das erste Gesetz besagt, dass `return` sozusagen links-neutral bzgl. `>>=` ist und nichts anderes tut, als sein Argument in der Monade zu verpacken. Das

zweite Gesetz besagt, dass `return` rechts-neutral bezüglich `>>=` ist und das dritte Gesetz drückt eine eingeschränkte Form der Assoziativität von `>>=` aus.

Wenn die monadischen Gesetze erfüllt sind, dann kann man daraus schließen, dass die durch `>>=` erzwungene Sequentialität wirklich gilt: Die Berechnungen werden sequentiell ausgeführt.

Fügt man für einen Datentypen eine Instanz der Klasse `Monad` hinzu, so sollte man vorher überprüft haben, dass die Monadengesetze für die Implementierung erfüllt sind. Der Compiler kann diese Prüfung nicht durchführen (i.a. ist dieses Problem unentscheidbar).

Wir rechnen die Gesetze für die Instanz von `Maybe` nach:

Das erste Gesetz folgt direkt aus der Definition von `return` und `>>=`:

```
return x >>= f  =  Just x >>= f  =  f x
```

Für das zweite Gesetz `m >>= return  =  m`, ist eine Fallunterscheidung nötig. Wenn `m` zu `Nothing` auswertet, dann:

```
Nothing >>= return  =  Nothing
```

Wenn `m` zu `Just e` auswertet, dann:

```
Just e >>= return  =  Just e
```

Wenn die Auswertung von `m` divergiert (symbolisch dargestellt als $\bot$), dann divergiert auch die Auswertung von $\bot$ `>>= return` (und ist daher gleich zu $\bot$).

Wir betrachten das dritte Gesetz:

```
m1 >>= (\x -> m2 x >>= m3)  =  (m1 >>= m2) >>= m3
```

Wir führen eine Fallunterscheidung durch:

- Wenn `m1` zu `Nothing` auswertet, dann:

```
  (Nothing >>= m2) >>= m3
= Nothing >>= m3
= Nothing
= Nothing >>= (\x -> m2 x >>= m3)
```

- Wenn `m1` zu `Just e` auswertet, dann:

```
    (Just e >>= m2) >>= m3
=   m2 e >>= m3
= (\x -> m2 x >>= m3) e
= Just e >>= (\x -> m2 x >>= m3)
```

- Wenn die Auswertung von `m1` divergiert, dann divergieren sowohl `(m1 >>= m2) >>= m3` als auch `m1 >>= (\x -> m2 x >>= m3)`.

6.1.2 Weitere Monaden

6.1.2.1 Die Listen-Monade

Auch Listen sind Instanzen der Klasse `Monad`. Die Instanzdefinition ist:

```
instance Monad [] where
   m >>= f  = concatMap f m
   return x = [x]
   fail s   = []
```

Die Implementierung der `>>=` und `return`-Operation entspricht dabei gerade den List Comprehensions. Betrachte die List Comprehension `[x*y | x <- [1..10], y <-[1,2]]`. Diese kann als Folge von Monadischen Listen-Aktionen geschrieben werden als

```
do
  x <- [1..10]
  y <- [1,2]
  return (x*y)
```

Beidesmal erhält man die gleiche Liste als Ergebnis. [1,2,2,4,3,6,4,8,5,10,6,12,7,14,8,16,9,18,10,20]

Auch Filter kann man in der do Notation verwenden: Als List-Comprehension: `[x*y | x <- [1..10], y <-[1,2], x*y < 15]`

Das kann man mit der Listenmonade schreiben als:

```
do
  x <- [1..10]
  y <- [1,2]
  if (x*y < 15) then return undefined else fail ""
  return (x*y)
```

Man erhält in beiden Fällen: `[1,2,2,4,3,6,4,8,5,10,6,12,7,14,8,9,10]`.

Die Listenmonade erfüllt die monadische Gesetze:
Wir rechnen das erste Gesetz mal nach, durch Verwendung der Definitiondn und Reduktionen:

```
return x >>= f = concatMap f [x]  = f x
```

Also gilt das erste Monadengesetz.

Das zweite Gesetz, `m >>= return = m` , kann man mit Fallunterscheidung nachweisen:

- Wenn `m` zu `[]` auswertet, dann ergibt sich `concatMap f []`, was `[]` ergibt.
- Wenn `m` nicht konvergiert, dann auch nicht `m >>= return` und umgekehrt.
- Wenn `m` zu `(x:xs)` auswertet, dann ergibt sich `concatMap` λ`y.[y] (x:xs)`, und somit `x: concatMap f xs`. Hier benötigt man eine spezielle Induktion, um zu schließen, dass sich hier genau die Liste selbst wieder ergibt.

Wir rechnen das dritte Gesetz nicht nach, da uns auch hier die Methoden fehlen, um Gleichheiten für beliebige (auch unendlich lange) Listen nachzuweisen.

6.1.2.2 Die StateTransformer-Monade

Der "StateTransformer" ist ein Datentyp, der die Veränderung eines Zustands kapselt. Er ist polymorph über dem eigentlichen Zustand definiert als:

```
data StateTransformer state a = ST (state -> (a,state))
```

Die gekapselte Funktion erhält einen Zustand und liefert als Ergebnis ein Paar bestehend aus einem Zustand und einem Ergebnis (vom Typ `a`).

Schaltet man solche StateTransformer mit den monadischen Operatoren hintereinander so kann man Zustandsbasiert programmieren, ohne den Zustand später wirklich sichtbar zu machen.

Die Instanz von `StateTransformer` für die Klasse `Monad` ist definiert als

```
instance Monad (StateTransformer s) where
 return a        = ST (\s -> (a,s))
 (ST x) >>= f    = ST (\s -> case x s of
                               (a,s' ) ->  case (f a) of
                                             (ST y) -> (y s'))
```

Wir betrachten als Beispiel die Programmierung eines "Taschenrechners", der sofort auswertet, d.h. die Punkt-vor-Strichrechnung nicht beachtet und (der Einfachheit halber) nur die Grundrechenarten beherrscht. Als inneren Zustand des Taschenrechners verwenden wir ein Paar: Die erste Komponente ist eine Funktion, die zweite Komponente eine Zahl. Die erste Komponente enthält das aktuelle Zwischenergebnis und der Operation die noch angewendet werden muss, die zweite Komponente die gerade eingebene Zahl.

Z.B. sei `30+50=` die Eingabe. Die Zustände sind dann (je nach verarbeitetem Zeichen):

Zustand	Resteingabe
`(\x -> x, 0.0)`	`30+50=`
`(\x -> x, 3.0)`	`0+50=`
`(\x -> x, 30.0)`	`+50=`
`(\x -> (+) 30.0 x, 0.0)`	`50=`
`(\x -> (+) 30.0 x, 5.0)`	`0=`
`(\x -> (+) 30.0 x, 50.0)`	`=`
`(\x -> x, 80.0)`	

Wir definieren ein Typsynonym für den Zustand:

```
type CalcState = (Float -> Float, Float)
```

Wir verwenden den `StateTransformer` zum monadischen Programmieren, indem wir für den Zustand `CalcState` verwenden, d.h. wir können als Abkürzung definieren:

```
type CalcTransformer a = StateTransformer CalcState a
```

Der Startzustand ist definiert als

```
startState = (id, 0.0)
```

Nun besteht unsere Aufgabe im Wesentlichen darin, verschiedene `CalcTransformer` zu definieren. Wenn diese nur den Zustand ändern (also keinen zusätzlichen Rückgabewert haben), so verwenden wir das Nulltupel `()`, um dies anzuzeigen, d.h. solche Zustandsveränderer sind vom Typ `CalcTransformer ()`.

Die Funktion `oper` implementiert die Zustandsveränderung für jeden beliebigen binären Operator:

```
oper :: (Float -> Float -> Float) -> CalcTransformer ()
oper op = ST $ \(fn,zahl) -> ((), (op (fn zahl), 0.0))
```

Die Funktion `clear` löscht die letzte Eingabe:

```
clear :: CalcTransformer ()
clear = ST $ \(fn,zahl) -> ((),if zahl == 0.0 then startState else (fn,0.0))
```

Die Funktion `total` berechnet ein Ergebnis

```
total :: CalcTransformer ()
total = ST $ \(fn,zahl) -> ((), (id, fn zahl))
```

Die Funktion `digit` verarbeitet eine Ziffer

```
digit :: Int -> CalcTransformer ()
digit i = ST $ \(fn,zahl) -> ((), (fn, (fromIntegral i) + zahl*10.0))
```

Schließlich definieren wir noch eine Funktion `readResult`, die das aktuelle Ergebnis aus dem Zustand ausliest, der Rückgabewert ist eine `Float`-Zahl

```
readResult:: CalcTransformer Float
readResult = ST $ \(fn,zahl) -> (fn zahl, (fn,zahl))
```

Als nächstes definieren wir die Funktion `calcStep`, die ein Zeichen der Eingabe verarbeiten kann:

```
calcStep :: Char -> CalcTransformer ()
calcStep x
 | isDigit x = digit (fromIntegral $ digitToInt x)

calcStep '+' = oper (+)
calcStep '-' = oper (-)
calcStep '*' = oper (*)
calcStep '/' = oper (/)
calcStep '=' = total
calcStep 'c' = clear
calcStep _   = ST $ \(fn,z) -> ((),(fn,z))
```

Für die Verarbeitung der gesamten Eingabe (als Text) definieren wir die Funktion `calc`. Diese benutzt die monadische `do`-Notation:

```
calc []     = readResult
calc (x:xs) = do
                calcStep x
                calc xs
```

Zum Ausführen des Taschenrechners muss nun die gesamte Aktion auf den Startzustand angewendet werden. Wir definieren dies als

```
runCalc :: CalcTransformer Float -> (Float,CalcState)
runCalc (ST akt) = akt startState
```

Schließlich definieren wir noch eine Hauptfunktion, die aus dem Ergebnis nur den Wert liest:

```
mainCalc xs = fst $ runCalc (calc xs)
```

Nun kann man den Taschenrechner schon ausführen:

```
 *Main> mainCalc "1+2*3"
9.0
*Main> mainCalc "1+2*3="
9.0
*Main> mainCalc "1+2*3c*3"
0.0
*Main> mainCalc "1+2*3c5"
15.0
*Main> mainCalc "1+2*3c5===="
15.0
*Main>
```

Allerdins fehlt dem Taschenrechner noch die Interaktion (z.B. erscheint das =-Zeichen noch eher nutzlos). Wir werden den Taschenrechner später erweitern, widmen uns zunächst aber Programmierung von Ein- und Ausgabe in Haskell.

6.2 Ein- und Ausgabe: Monadisches IO

In einer rein funktionalen Programmiersprache mit verzögerter Auswertung wie Haskell sind Seiteneffekte zunächst verboten. Fügt man Seiteneffekte einfach hinzu (z.B. durch eine „Funktion“ `getZahl`), die beim Aufruf eine Zahl

vom Benutzer abfragt und anschließend mit dieser Zahl weiter auswertet, so erhält man einige unerwünschte Effekte der Sprache, die man im Allgemeinen nicht haben möchte.

- Rein funktionale Programmiersprachen sind *referentiell transparent*, d.h. eine Funktion angewendet auf gleiche Werte, ergibt stets denselben Wert im Ergebnis. Die referentielle Transparenz wird durch eine Funktion wie `getZahl` verletzt, da `getZahl` je nach Ablauf unterschiedliche Werte liefert.

- Ein weiteres Gegenargument gegen das Einführen von primitiven Ein-/ Ausgabefunktionen besteht darin, dass übliche (schöne) mathematische Gleichheiten wie $e + e = 2 * e$ für alle Ausdrücke der Programmiersprache nicht mehr gelten. Setze `getZahl` für e ein, dann fragt $e * e$ zwei verschiedene Werte vom Benutzer ab, während $2 * e$ den Benutzer nur einmal fragt. Würde man also solche Operationen zulassen, so könnte man beim Transformieren innerhalb eines Compilers übliche mathematische Gesetze nur mit Vorsicht anwenden.

- Durch die Einführung von direkten I/O-Aufrufen besteht die Gefahr, dass der Programmierer ein anderes Verhalten vermutet, als sein Programm wirklich hat. Der Programmierer muss die verzögerte Auswertung von Haskell beachten. Betrachte den Ausdruck `length [getZahl, getZahl]`, wobei `length` die Länge einer Liste berechnet als

  ```
  length [] = 0
  length (_:xs) = 1 + length xs
  ```

  Da die Auswertung von `length` die Listenelemente gar nicht anfasst, würde obiger Aufruf, keine `getZahl`-Aufrufe ausführen.

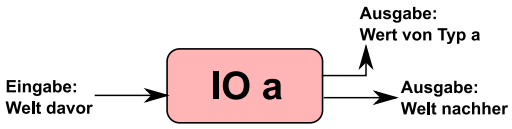
- In reinen funktionalen Programmiersprachen wird oft auf die Festlegung einer genauen Auswertungsreihenfolge verzichtet, um Optimierungen und auch Parallelisierung von Programmen durchzuführen. Z.B. könnte ein Compiler bei der Auswertung von $e_1 + e_2$ zunächst e_2 und danach e_1 auswerten. Werden in den beiden Ausdrücken direkte I/O-Aufrufe benutzt, spielt die Reihenfolge der Auswertung jedoch eine Rolle, da sie die Reihenfolge der I/O-Aufrufe wider spiegelt.

Aus all den genannten Gründen, wurde in Haskell ein anderer Weg gewählt. I/O-Operationen werden mithilfe des so genannten *monadischen I/O* programmiert. Hierbei werden I/O-Aufrufe vom funktionalen Teil gekapselt. Zu Programmierung steht der Datentyp `IO a` zu Verfügung. Ein Wert vom Typ `IO a` stellt jedoch kein Ausführen von Ein- und Ausgabe dar, sondern eine *I/O-Aktion*, die erst beim *Ausführen* (außerhalb der funktionalen Sprache) Ein-/Ausgaben durchführt und anschließend einen Wert vom Typ `a` liefert. Der Datentyp der `IO` ist Instanz der Klasse `Monad`. Wir geben die genau Implementierung der Instanz nicht an. Man kann die monadischen Operatoren verwenden, um aus kleinen IO-Aktionen größere zusammenzusetzen. Die große (durch `main`) definierte I/O-Aktion wird im Grunde dann außerhalb von Haskell ausgeführt (das Haskell-Programm beschreibt ja nur die Aktion). Die kleinsten IO-Aktionen sind primitiv eingebaut.

Im folgenden erläutern wir eine anschauliche Vorstellung der Implementierung von `IO`. In Realität ist die Implementierung leicht anders. Eine I/O-Aktion ist passend zum `StateTransformer` eine Funktion, die als Eingabe einen Zustand erhält und als Ausgabe den veränderten Zustand sowie ein Ergebnis liefert. Da der gesamte Speicher manipuliert werden kann, ist der manipulierte Zustand dabei die ganze Welt (als Vorstellung). D.h. der Typ `IO` kann als Haskell-Typ geschrieben werden:

```
type IO a = Welt -> (a,Welt)
```

Man kann dies auch durch folgende Grafik illustrieren:



Aus Sicht von Haskell sind Objekte vom Typ `IO a` bereits *Werte*, d.h. sie können nicht weiter ausgewertet werden. Dies passt dazu, dass auch andere Funktionen Werte in Haskell sind. Allerdings im Gegensatz zu „normalen“ Funktionen kann Haskell kein Argument vom Typ „Welt“ bereit stellen. (Die Funktion `runCalc` für den `CalcTransformer` ist für `IO` so nicht definierbar). Die Ausführung der Funktion geschieht erst durch das Laufzeitsystem, welche die Welt auf die durch `main` definierte I/O-Aktion anwendet.

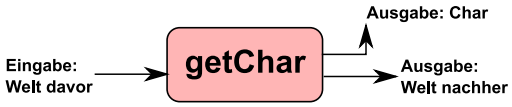
6.2.1 Primitive I/O-Operationen

Wir gehen zunächst von zwei Basisoperationen aus, die Haskell primitiv zur Verfügung stellt. Zum Lesen eines Zeichens vom Benutzer gibt es die Funkti-

on `getChar`:

```
getChar :: IO Char
```

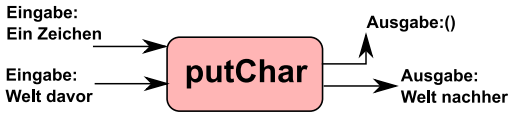
In der Welt-Sichtweise ist `getChar` eine Funktion, die eine Welt erhält und als Ergebnis eine veränderte Welt sowieso ein Zeichen liefert. Man kann dies durch folgendes Bild illustrieren:



Analog dazu gibt es die primitive Funktion `putChar`, die als Eingabe ein Zeichen (und eine Welt) erhält und nur die Welt im Ergebnis verändert. Da alle I/O-Aktionen jedoch noch ein zusätzliches Ergebnis liefern müssen, wird hier der 0-Tupel `()` verwendet.

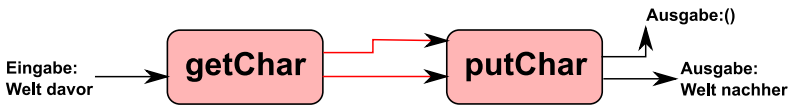
```
putChar :: Char -> IO ()
```

Auch `putChar` lässt sich mit einem Bild illustrieren:



6.2.2 Komposition von I/O-Aktionen

Um aus den primitiven I/O-Aktionen größere Aktionen zu erstellen, werden Kombinatoren benötigt, um I/O-Aktionen miteinander zu verknüpfen. Z.B. könnte man zunächst mit `getChar` ein Zeichen lesen, welches anschließend mit `putChar` ausgegeben werden soll. Im Bild dargestellt möchte man die beiden Aktionen `getChar` und `putChar` wiefolgt sequentiell ausführen und dabei die Ausgabe von `getChar` als Eingabe für `putChar` benutzen (dies gilt sowohl für das Zeichen, aber auch für den Weltzustand):



Genau diese Verknüpfung leistet der monadische Kombinator >>=. Für den `IO`-Typen hat >>= den Typ:

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

D.h. er erhält eine IO-Aktion, die einen Wert vom Typ `a` liefert, und eine Funktion die einen Wert vom Typ `a` verarbeiten kann, indem sie als Ergebnis eine IO-Aktion vom Typ `IO b` erstellt.

Wir können nun die gewünschte IO-Aktion zum Lesen und Asgegeben eines Zeichens mithilfe von >>= definieren:

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

Man kann alternativ die `do`-Notation verwenden, und würde dann programmieren:

```
echo :: IO ()
echo = do
       c <- getChar
       putChar c
```

Der >>-Operator kann zum Verknüpfen von zwei IO-Aktionen verwendet werden, wenn das Ergebnis der ersten Operation irrelevant ist.

Er wird benutzt, um aus zwei I/O-Aktionen die Sequenz beider Aktionen zur erstellen, wobei das Ergebnis der ersten Aktion *nicht* von der zweiten Aktion benutzt wird (die Welt wird allerdings weitergereicht).

Mithilfe der beiden Operatoren kann man z.B. eine IO-Aktion definieren, die ein gelesenes Zeichen zweimal ausgibt:

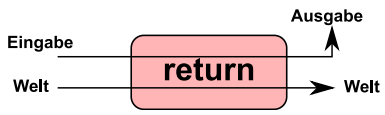
```
echoDup :: IO ()
echoDup = getChar >>= (\x -> putChar x >> putChar x)
```

Alternativ mit der `do`-Notation

```
echoDup :: IO ()
echoDup = do
          x <- getChar
          putChar x
          putChar x
```

Angenommen wir möchten eine IO-Aktion erstellen, die zwei Zeichen liest und diese anschließend als Paar zurück gibt. Dafür muss man das Paar in eine IO-Aktion verpacken. Das liefert die Funktion `return`.

Als Bild kann man sich die `return`-Funktion wie folgt veranschaulichen:



Die gewünschte Operation, die zwei Zeichen liest und diese als Paar zurück liefert kann nun definiert werden:

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>= \x ->
              getChar >>= \y ->
              return (x,y)
```

oder alternativ mit der `do`-Notation:

```
getTwoChars :: IO (Char,Char)
getTwoChars = do
                x <- getChar
                y <- getChar
                return (x,y)
```

Als Fazit zur Implementierung von IO in Haskell kann man sich merken, dass neben den primitiven Operationen wie `getChar` und `putChar`, die Kombinatoren `>>=` und `return` ausreichen, um genügend viele andere Operationen zu definieren.

Wir zeigen noch, wie man eine ganze Zeile Text einlesen kann, indem man `getChar` wiederholt rekursiv aufruft:

```
getLine :: IO [Char]
getLine = do c <- getChar;
             if c == '\n' then
               return []
             else
               do
                cs <- getLine
                return (c:cs)
```

6.2.3 Implementierung der IO-Monade

Passend zum Welt-Modell kann man den `>>=` und den `return`-Operator wie folgt für den `IO`-Typen implementieren:

Zunächst muss der `IO`-Typ extra verpackt werden, damit man eine Klasseninstanz hinzufügen kann (damit entspricht der `IO`-Typ fast dem StateTransformer-Typ):

```
newtype IO a = IO (Welt -> (a,Welt))
instance Monad IO where
 (IO m) >>= k = IO (\s -> case m s of
                           (s',a') -> case (k a) of
                                        (IO k') -> k' s'
 return x = IO (\ s -> (s, x))
```

Ein interessanter Punkt bei der Implementierung ist, dass sie nur dann richtig ist, wenn die call-by-need Auswertung verwendet wird, da anderenfalls die Welt verdoppelt werden könnte (wenn man die (β)-Reduktion und call-by-name Auswertung verwendet).

Intern wird durch den GHC jedoch das Durchreichen der Welt weg optimiert, d.h. es wird nie ein echtes Objekt des Welt-Zustands erzeugt.

6.2.4 Monadische Gesetze und die `IO`-Monade

Es wird oft behauptet, dass die `IO`-Monade die monadischen Gesetze erfüllt. Analysiert man jedoch genau, so stimmt dies nicht ganz. Betrachte z.B. das erste Gesetz

```
return x >>= f  =  f x
```

Die Implementierung von `>>=` für die `IO`-Monade liefert sofort einen Konstruktor `IO`. Deshalb terminiert ein Aufruf der Form `seq (return e1 >>= e2) True` immer (`return e1 >>= e2` ist immer ein Konstruktor!). Allerdings kann man `e1` und `e2` so wählen, dass die Anwendung `e2 e1` nicht terminiert. Der Kontext `seq` $[\cdot]$ `True` unterscheidet dann `return eq >>= e2` und `e2 e1`. Ein Beispiel im Interpreter:

```
> let a = True
> let f = \x -> (undefined::IO Bool)
> seq (return a >>= f) True
True
> seq (f a) True
*** Exception: Prelude.undefined
```

Man kann sich streiten, ob die Implementierung der `IO`-Monade falsch ist, oder ob der obige Gleichheitstest zuviel verlangt. Es gilt ungefähr: Die monadischen Gesetze für die `IO`-Monade gelten in Haskell, wenn man nur Werte betrachtet (also insbesondere keine divergenten Ausdrücke). Trotzdem funktioniert die Sequentialisierung, deshalb werden die kleinen Abweichung von den monadischen Gesetzen in Kauf genommen.

6.2.5 Verzögern innerhalb der IO-Monade

Eine der wichtigsten Eigenschaften des monadischen IOs in Haskell ist, dass es keinen Weg aus einer Monade heraus gibt. D.h. es gibt keine Funktion `f:: IO a -> a`, die aus einem in einer I/O-Aktion verpackten Wert nur diesen Wert extrahiert. Dies erzwingt, dass man `IO` nur innerhalb der Monade programmieren kann und i.A. rein funktionale Teile von der I/O-Programmierung trennen sollte.

Dies ist zumindest in der Theorie so. In der Praxis stimmt obige Behauptung nicht mehr, da alle Haskell-Compiler eine Möglichkeit bieten, die IO-Monade zu „knacken“. Im folgenden Abschnitt werden wir uns mit den Gründen dafür beschäftigen und genauer erläutern, wie das „Knacken“ durchgeführt wird.

Wir betrachten ein Problem beim monadischen Programmieren. Wir schauen uns die Implementierung des `readFile` an, welches den Inhalt einer Datei ausliest. Hierfür werden intern `Handles` benutzt. Diese sind im Grunde „intelligente“ Zeiger auf Dateien. Für `readFile` wird zunächst ein solcher Handle erzeugt (mit `openFile`), anschließend der Inhalt gelesen (mit `leseHandleAus`).

```
-- openFile :: FilePath -> IOMode -> IO Handle
-- hGetChar :: Handle -> IO Char

readFile :: FilePath -> IO String
readFile path =
 do
  handle <- openFile path ReadMode
  inhalt <- leseHandleAus handle
  return inhalt
```

Es fehlt noch die Implementierung von `leseHandleAus`. Diese Funktion soll alle Zeichen vom `Handle` lesen und anschließend diese als Liste zurückgege-

ben und den Handle noch schließen (mit `hClose`). Wir benutzen außerdem die vordefinierten Funktion `hIsEOF :: Handle -> IO Bool`, die testet ob das Dateiende erreicht ist und `hGetChar`, die ein Zeichen vom Handle liest.

Ein erster Versuch führt zur Implementierung:

```
leseHandleAus handle =
  do
    ende <- hIsEOF handle
    if ende then
          do
           hClose handle
           return []
     else do
           c  <- hGetChar handle
           cs <- leseHandleAus handle
           return (c:cs)
```

Diese Implementierung funktioniert, ist allerdings sehr speicherlastig, da das letzte `return` erst ausgeführt wird, nachdem auch der rekursive Aufruf durchgeführt wurde. D.h. wir lesen die gesamte Datei aus, bevor wir irgendetwas zurückgegeben. Dies ist unabhängig davon, ob wir eigentlich nur das erste Zeichen der Datei oder alle Zeichen benutzen wollen. Für eine verzögert auswertende Programmiersprache und zum eleganten Programmieren ist dieses Verhalten nicht gewünscht. Deswegen benutzen wir die Funktion `unsafeInterleaveIO :: IO a -> IO a`, die die strenge Sequentialisierung der IO-Monade aufbricht, d.h. anstatt die IO-Aktion sofort durchzuführen wird beim Aufruf innerhalb eines `do`-Blocks:

```
do
 ...
 ergebnis <- unsafeInterleaveIO aktion
 weitere_Aktionen
```

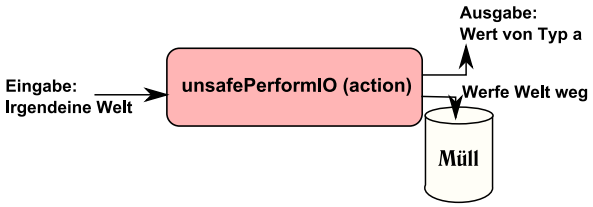
nicht die Aktion `aktion` durchgeführt (berechnet), sondern direkt mit den weiteren Aktionen weitergemacht. Die Aktion `aktion` wird erst dann ausgeführt, wenn der Wert der Variablen `ergebnis` benötigt wird.

Die Implementierung von `unsafeInterleaveIO` verwendet `unsafePerformIO`:

```
unsafeInterleaveIO a = return (unsafePerformIO a)
```

Die monadische Aktion `a` vom Typ `IO a` wird mittels `unsafePerformIO` in einen nicht-monadischen Wert vom Typ `a` konvertiert; mittels `return` wird dieser Wert dann wieder in die `IO`-Monade verpackt.

Die Funktion `unsafePerformIO` „knackt“ also die IO-Monade. Im Welt-Modell kann man sich dies so vorstellen: Es wird irgendeine Welt benutzt, um die IO-Aktion durchzuführen. Anschließend wird die neue Welt nicht weitergereicht, sondern sofort weggeworfen.



Die Implementierung von `leseHandleAus` ändern wir nun ab in:

```
leseHandleAus handle =
 do
  ende <- hIsEOF handle
  if ende then
        do
         hClose handle
         return []
   else do
         c  <- hGetChar handle
         cs <- unsafeInterleaveIO (leseHandleAus handle)
         return (c:cs)
```

Nun liefert `readFile` schon Zeichen, bevor der komplette Inhalt der Datei gelesen wurde. Beim Testen im Interpreter sieht man den Unterschied.

Mit der Version ohne `unsafeInterleaveIO`:

```
*Main> writeFile "LargeFile" (concat [show i | i <- [1..100000]])
*Main> readFile "LargeFile" >>= print . head
'1'
(7.09 secs, 263542820 bytes)
```

Mit Benutzung von `unsafeInterleaveIO`:

```
*Main> writeFile "LargeFile" (concat [show i | i <- [1..100000]])
*Main> readFile "LargeFile" >>= print . head
```

```
'1'
(0.00 secs, 0 bytes)
```

Beachte, dass beide Funktionen `unsafeInterleaveIO` und `unsafePerformIO` nicht vereinbar sind mit monadischem IO, da sie die strenge Sequentialisierung aufbrechen. Eine Rechtfertigung, die Funktionen trotzdem einzusetzen, besteht darin, dass man gut damit Bibliotheksfunktionen oder ähnliches definieren kann. Wichtig dabei ist, dass der Benutzer der Funktion sich bewusst ist, dass deren Verwendung eigentlich verboten ist und dass das Ein- / Ausgabeverhalten nicht mehr sequentiell ist. D.h. sie sollte nur verwendet werden, wenn das verzögerte I/O das Ergebnis nicht beeinträchtigt.

6.2.6 Speicherzellen

Haskell stellt primitive Speicherplätze mit dem (abstrakten) Datentypen `IORef` zur Verfügung. Abstrakt meint hier, dass die Implementierung (die Datenkonstruktoren) nicht sichtbar ist, die Konstruktoren sind in der Implementierung verborgen.

```
data IORef a = (nicht sichtbar)
```

Ein Wert vom Typ `IORef a` stellt eine Speicherzelle dar, die ein Element vom Typ `a` speichert. Es gibt drei primitive Funktionen (bzw. I/O-Aktionen) zum Erstellen und zum Zugriff auf `IORefs`. Die Funktion

```
newIORef :: a -> IO (IORef a)
```

erwartet ein Argument und erstellt anschließend eine IO-Aktion, die bei Ausführung eine Speicherzelle mit dem Argument als Inhalt erstellt.

Die Funktion

```
readIORef :: IORef a -> IO a
```

kann benutzt werden, um den Inhalt einer Speicherzelle (innerhalb der IO-Monade) auszulesen. Analog dazu schreibt die Funktion

```
writeIORef :: a -> IORef a -> IO ()
```

das erste Argument in die Speicherzelle (die als zweites Argument übergeben wird).

6.2.7 Kontrollstrukturen – Schleifen

Wir haben bereits gesehen, dass die monadische Programmierung der imperativen Programmierung ähnelt. In imperativen Sprachen sind Schleifen als Kontrollstrukturen ein wichtiges Konstrukt. Auch in Haskell kann man "monadische" Schleifen programmieren.

Die repeat-until-Schleife kann man wie folgt implementieren:

```
repeatUntil :: (Monad m) => m a -> m Bool -> m ()
repeatUntil koerper bedingung =
  do
    koerper
    b <- bedingung
    if b then return () else repeatUntil koerper bedingung
```

Der Schleifenkörper und die Abbruchbedingung sind dabei selbst monadische Operationen. Beachte die `repeatUntil`-Schleife kann so für alle Monaden verwendet werden.

Wir können sie z.B. für die `IO`-Monade verwenden und ein Programm schreiben, dass die ersten 100 Zahlen ausdruckt.

In einer imperativen Sprache würde man hierfür inetwa schreiben:

```
X := 0;
repeat
  print X;
  X := X+1;
until X > 100
```

In Haskell kann man dies implementieren mit `IORef`s für den Speicherplatz und obiger `repeatUntil`-Funktion als:

```
dieErstenHundertZahlen =
 do
  x <- newIORef 0
  repeatUntil
     (do
       wertVonX <- readIORef x
       print wertVonX
       writeIORef x (wertVonX + 1)
      )
     (readIORef x >>= \x -> return (x > 100))
```

Die Implementierung einer `while`-Schleife ist analog:

```
while :: (Monad m) => m Bool -> m a -> m ()
while  bedingung koerper =
 do
  b <- bedingung
  if b then  do
    koerper
    while  bedingung koerper
   else return ()
```

Das imperative Programm:

```
X := 0;
while X <= 100 do
  print X;
  X := X+1;
```

kann man damit in Haskell programmieren als:

```
dieErstenHundertZahlen' =
  do
   x <- newIORef 0
   while (readIORef x >>= \x -> return (x <= 100))
      (do
        wertVonX <- readIORef x
        print wertVonX
        writeIORef x (wertVonX + 1)
       )
```

6.2.8 Nützliche Monaden-Funktionen

In diesem Abschnitt stellen wir einige weitere nützliche Funktionen auf Monaden vor. Die meisten davon sind in der Bibliothek `Control.Monad` definiert.

Die Funktion `forever` führt eine monadische Aktion unendlich lange wiederholt aus, sie ist definiert als:

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

Die Funktion `when` verhält sich wie ein imperatives `if-then` (also eine bedingte Verzweigung ohne `else`-Zweig). Sie ist definiert als

```
when      :: (Monad m) => Bool -> m () -> m ()
when p s  =  if p then s else return ()
```

Die Funktion `sequence` erwartet eine Liste von monadischen Aktionen und erstellt daraus eine monadische Aktion, die die Aktion aus der Eingabe sequentiell nacheinander ausführt und als Ergebnis die Liste aller Einzelergebnisse liefert. Man kann `sequence` wie folgt definieren:

```
sequence :: (Monad m) => [m a] -> m [a]
sequence []          = return []
sequence (action:as) = do
                         r <- action
                         rs <- sequence as
                         return (r:rs)
```

Analog dazu ist die Funktion `sequence_` definiert, die sämtliche Ergebnisse verwirft:

```
sequence_ :: (Monad m) => [m a] -> m ()
sequence_ []          = return ()
sequence_ (action:as) = do
                         action
                         sequence_ as
```

Die Funktion `mapM` ist der `map`-Ersatz für das monadische Programmieren. Eine Funktion, die aus einem Listenelement eine monadische Aktion macht, wird auf eine Liste angewendet, gleichzeitig wird mit `sequence` eine große Aktion erstellt:

```
mapM            :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as       =  sequence (map f as)
```

Analog dazu ist `mapM_`, wobei das Ergebnis verworfen wird:

```
mapM_           :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as      =  sequence_ (map f as)
```

Zum Beispiel kann man damit ganz einfach die ersten einhundert Zahlen Zeilenweise ausdrucken:

```
*Main> mapM_ print [1..100]
1
2
3
4
5
...
```

6.3 Monad-Transformer

Wir kehren zurück zum Taschenrechner-Beispiel. Die bisherige Implementierung erlaubt keine Interaktion. Wir würden nun gerne IO-Aktion durchführen, um jedes eingegebene Zeichen direkt zu verarbeiten. D.h. eine `main`-Funktion der folgenden Form implementieren:

```
calc = do
          c <- getChar
          if c /= '\n' then do
            calcStep c
            calc
           else return ()
```

Leider funktioniert die Implementierung so nicht, da dort monadische Aktion *zweier verschiedener* Monaden vermischt werden: `getChar` ist eine Aktion der `IO`-Monade während `calcStep c` eine Aktion der `StateTransformer`-Monade ist. Ein Ausweg aus dieser Situation ist es, die `StateTransformer`-Monade zu erweitern, so dass sie `IO`-Aktionen auch verarbeiten kann. Man kann dies gleich allgemein machen: Die `StateTransformer`-Monade wird um eine beliebige andere Monade erweitert. Wir definieren hierfür den Typ `StateTransformerT`:

```
newtype StateTransformerT monad state a = STT (state -> monad (a,state))
```

Im Gegensatz zu `StateTransformer` ist die gekapselte Funktion nun eine monadische Aktion selbst (polymorph über der Variablen `monad`. Solche Datentypen (die Monaden sind, und selbst um eine Monade erweitert sind) nennt man auch "Monad-Transformer".

Die Monaden-Instanz für `StateTransformerT` ist

```
instance Monad m => Monad (StateTransformerT m s) where
 return x = STT $ \s -> return (x,s)
 (STT x) >>= f = STT $ (\s -> do
                              (a,s') <- x s
                              case (f a) of
                                (STT y) ->  (y s'))
```

Für den Taschenrechner definieren wir noch als Abkürzung ein Typsynonym: Der verwendete StateTransformerT bindet nun die `IO`-Monade mit ein:

```
type CalcTransformerT a = StateTransformerT IO CalcState a
```

Das nächste Ziel ist es, die bereits für `CalcTransformer` definierten Funktionen in den neuen Typ `CalcTransformerT` zu "liften": Da sie keine neue Funktionalität haben müssen sie lediglich monadisch verpackt werden. Hierfür kann man allgemein eine Funktion `lift` definieren:

```
lift :: CalcTransformer a -> CalcTransformerT a
lift (ST fn) = STT $ \x -> return (fn x)
```

Mithilfe von `lift` können wir nun die meisten Funktionen für `CalcTransformer` hochziehen zu Funktionen für `CalcTransformerT`:

```
oper' :: (Float -> Float -> Float) -> CalcTransformerT ()
oper' op = lift (oper op)

digit' :: Int -> CalcTransformerT ()
digit' i = lift (digit i)

readResult' :: CalcTransformerT Float
readResult' = lift readResult
```

Die Implementierungen von `clear` und `total` möchten wir jedoch anpassen, da sie nun auch Ausgaben durchführen sollen. Hier können wir nun die `IO`-Monade verwenden:

```
total' =
 STT $ \(fn,zahl) -> do
  let res = ((), (id, fn zahl))
  putStr $ show (fn zahl)
  return res
```

```
clear' =
 STT $ \(fn,zahl) ->
  if zahl == 0.0 then do
    putStr ("\r"  ++ (replicate 100 ' ') ++ "\r")
    return ((),startState)
  else do
    let l = length (show zahl)
    putStr $ (replicate l '\b') ++ (replicate l ' ' ) ++ (replicate l '\b')
    return ((),(fn,0.0))
```

Die Definition von `calcStep` ist analog wie vorher:

```
calcStep' :: Char -> CalcTransformerT ()
calcStep' x
 | isDigit x = digit' (fromIntegral $ digitToInt x)

calcStep' '+' = oper' (+)
calcStep' '-' = oper' (-)
calcStep' '*' = oper' (*)
calcStep' '/' = oper' (/)
calcStep' '=' = total'
calcStep' 'c' = clear'
calcStep' _   = STT $ \(fn,z) -> return ((),(fn,z))
```

Die Hauptschleife findet in der Funktion `calc'` statt, diese liste ein Zeichen, berechnet den Folgezustand und ruft sich anschließend selbst auf. Die verwendete Monade ist die `StateTransformerT`-Monade. Daher kann `getChar` aus der `IO`-Monade nicht direkt verwendet werden. Diese muss erst in die `StateTransformerT`-Monade geliftet werden, hierfür definieren wir allgemein die Funktion `liftIO`: Diese führt eine IO-Aktion aus, und lässt den Zustand der `StateTransformerT`-Monade unverändert:

```
liftIO :: IO a -> CalcTransformerT a
liftIO akt = STT (\s -> do
                          r <- akt
                          return (r,s))

calc' :: CalcTransformerT ()
calc' = do
```

```
          c <- liftIO $ getChar
          if c /= '\n' then do
            calcStep' c
            calc'
           else return ()
```

Das Hauptprogramm ruft nun `calc'` auf und wendet den initialen Zustand an, die ersten beiden Zeilen des Hauptprogramms setzen die Pufferung so, dass jedes Zeichen direkt vom Programm verarbeitet wird:

```
main = do
          hSetBuffering stdin NoBuffering  -- neu
          hSetBuffering stdout NoBuffering -- neu
          runST' $ calc'

runST' (STT s) = s startState
```

Zusammenfassend dienen Monad-Transformer dazu, mehere Monaden zu vereinen. Dabei kann man mit `lift`-Funktionen bestehende Funktionen relativ komfortabel in die neue Monade liften. Beachte, dass wir sowohl Funktionen aus der `StateTransformer`-Monade als auch Funktionen aus der `IO`-Monade in die `StateTransformerT`-Monade geliftet haben.

Übungsaufgabe 6.3.1. *Der vorgestellte Taschenrechner kann als Eingabe nur nichtnegative ganze Zahlen verarbeiten. Erweitern Sie den Taschenrechner, so dass auch Kommazahlen eingegeben werden können.*

6.4 Quellennachweis

Die Darstellung von Haskells monadischen IO richtet sich im Wesentlichen nach (Peyton Jones, 2001). Als weiterführende Literatur sind die Originalarbeiten von Wadler zur Verwendung von Monaden in Haskell (z.B. (Wadler, 1992; Wadler, 1995)) und die theoretische Fundierung von Moggi (Moggi, 1991) sehr empfehlenswert.

Literatur

Ariola, Z. M. & Felleisen, M. (**1997**). The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., & Wadler, P. (**1995**). The call-by-need lambda calculus. In *POPL '95: Proceedings of the 22th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 233–246. ACM Press, New York, NY, USA.

Barendregt, H. P. (**1984**). *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York.

Bird, R. (**1998**). *Introduction to Functional Programming using Haskell*. Prentice Hall PTR, 2 edition.

Bird, R. & Wadler, P. (**1988**). *Introduction to Functional Programming*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, Upper Saddle River, NJ, USA.

Chakravarty, M. & Keller, G. (**2004**). *Einführung in die Programmierung mit Haskell*. Pearson Studium.

Church, A. (**1941**). *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey.

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., & Tommasi, M. (**2007**). Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`. Release October, 12th 2007.

Damas, L. & Milner, R. (**1982**). Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, New York, NY, USA.

Davie, A. J. T. (**1992**). *An introduction to functional programming systems using Haskell*. Cambridge University Press, New York, NY, USA.

Hall, C. V., Hammond, K., Jones, S. L. P., & Wadler, P. (**1996**). Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138.

Hankin, C. (**2004**). *An introduction to lambda calculi for computer scientists*. Number 2 in Texts in Computing. King's College Publications, London, UK.

Henglein, F. (**1993**). Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289.

Hindley, J. R. (**1969**). The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60.

Hoare, C. A. R. (**1969**). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Hughes, J. (**1989**). Why Functional Programming Matters. *Computer Journal*, 32(2):98–107.

Hutton, G. (**2007**). *Programming in Haskell*. Cambridge University Press.

Jones, M. P. (**1995**). A system of constructor classes: Overloading and implicit higher-order polymorphism. *J. Funct. Program.*, 5(1):1–35.

Kfoury, A. J., Tiuryn, J., & Urzyczyn, P. (**1990**a). Ml typability is dexptime-complete. In *CAAP '90: Proceedings of the fifteenth colloquium on CAAP'90*, pages 206–220. Springer-Verlag New York, Inc., New York, NY, USA.

Kfoury, A. J., Tiuryn, J., & Urzyczyn, P. (**1990**b). The undecidability of the semi-unification problem. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 468–476. ACM, New York, NY, USA.

Kfoury, A. J., Tiuryn, J., & Urzyczyn, P. (**1993**). Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311.

Kurt, W. (**2018**). *Get programming with Haskell*. Manning Publications Co., Shelter Island, NY, USA.

Kutzner, A. (**2000**). *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic choice: Operationale Semantik, Programmtransformationen und Anwendungen*. Dissertation, J. W. Goethe-Universität Frankfurt. In german.

Kutzner, A. & Schmidt-Schauß, M. (**1998**). A nondeterministic call-by-need lambda calculus. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 324–335. ACM Press.

Mairson, H. G. (**1990**). Deciding ml typability is complete for deterministic exponential time. In *POPL '90: Proceedings of the 17th ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 382–401. ACM, New York, NY, USA.

Mann, M. (**2005**a). *A Non-Deterministic Call-by-Need Lambda Calculus: Proving Similarity a Precongruence by an Extension of Howe's Method to Sharing*. Dissertation, J. W. Goethe-Universität, Frankfurt.

Mann, M. (**2005**b). Congruence of bisimulation in a non-deterministic call-by-need lambda calculus. *Electronic Notes in Theoretical Computer Science*, 128(1):81–101.

Maraist, J., Odersky, M., & Wadler, P. (**1998**). The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317.

Marlow, S., editor (**2010**). *Haskell 2010 Language Report*.

Milner, R. (**1978**). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Moggi, E. (**1991**). Notions of computation and monads. *Inf. Comput.*, 93(1):55–92.

Morris, J. H., Jr. (**1968**). *Lambda Calculus Models of Programming Languages*. Ph.D. thesis, MIT, Cambridge, MA.

Mycroft, A. (**1984**). Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228. Springer-Verlag, London, UK.

O'Sullivan, B., Goerzen, J., & Stewart, D. (**2008**). *Real World Haskell*. O'Reilly Media, Inc.

Pepper, P. (**1998**). *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. Springer-Lehrbuch. ISBN 3-540-64541-1.

Pepper, P. & Hofstedt, P. (**2006**). *Funktionale Programmierung - Weiterführende Konzepte und Techniken*. Springer-Lehrbuch. ISBN 3-540-20959-X.

Peyton Jones, S., editor (**2003**). *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, New York, NY, USA. `www.haskell.org`.

Peyton Jones, S. L. (**1987**). *The Implementation of Functional Programming Languages* (*Prentice-Hall International Series in Computer Science*). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Peyton Jones, S. L. (**2001**). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In T. Hoare, M. Broy, & R. Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, Amsterdam, The Netherlands.

Pierce, B. C. (**2002**). *Types and programming languages*. MIT Press, Cambridge, MA, USA.

Plotkin, G. D. (**1975**). Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159.

Sabel, D. (**2008**). *Semantics of a Call-by-Need Lambda Calculus with McCarthy's amb for Program Equivalence*. Dissertation, J. W. Goethe-Universität Frankfurt, Institut für Informatik. Fachbereich Informatik und Mathematik.

Sabel, D. & Schmidt-Schauß, M. (**2008**). A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Mathematical Structures in Computer Science*, 18(3):501–553.

Sabel, D., Schmidt-Schauß, M., & Harwath, F. (**2009**). Reasoning about contextual equivalence: From untyped to polymorphically typed calculi. In S. Fischer, E. Maehle, & R. Reischuk, editors, *INFORMATIK 2009, Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 28.9 - 2.10.2009 in Lübeck*, volume 154 of *GI Edition - Lecture Notes in Informatics*, pages 369; 2931–45. (4. Arbeitstagung Programmiersprachen (ATPS)).

Schmidt-Schauß, M. & Machkasova, E. (**2008**). A finite simulation method in a non-deterministic call-by-need lambda-calculus with letrec, constructors and case. In A. Voronkov, editor, *Rewriting Techniques and Applications (RTA-18)*, volume 5117 of *LNCS*, pages 321–335. Springer-Verlag.

Schmidt-Schauß, M., Sabel, D., & Machkasova, E. (**2010**). Simulation in the call-by-need lambda-calculus with letrec. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 295–310. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

Schmidt-Schauß, M., Schütz, M., & Sabel, D. (**2008**). Safety of Nöcker's strictness analysis. *Journal of Functional Programming*, 18(4):503–551.

Thompson, S. (**1999**). *Haskell: The Craft of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Wadler, P. (**1992**). Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493.

Wadler, P. (**1995**). Monads for functional programming. In J. Jeuring & E. Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer.

Wadler, P. & Blott, S. (**1989**). How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76. ACM, New York, NY, USA.