Hochschule RheinMain

Funktionale Programmierung

Sommersemester 2025

Prof. Dr. David Sabel

Theoretische Informatik
Fachbereich DCSM
Hochschule RheinMain
Unter den Eichen 5
65195 Wiesbaden
Email: david.sabel@hs-rm.de

Dieses Skript wird im Laufe der Vorlesung überarbeitet!

Stand: 18. Juni 2025

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 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 und hohe Abstraktion: Funktionen höherer Ordnung (d.h. Funktio-

nen, 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 „liste ist leer“ then 0 else
    „erstes Element von liste“ + summe „liste ohne erstes Element“

produkt liste =
  if „liste ist leer“ then 1 else
    „erstes Element von liste“ * produkt „liste ohne erstes Element“
```

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

```
reduce e f liste =
  if „liste ist leer“ then e else
    f „erstes Element von 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 Funktionskomposition 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.
- Unveränderliche Datenstrukturen: Eine Konsequenz der referentiellen Transparenz ist,

dass Datenstrukturen unveränderlich sind (da Seiteneffekte nicht erlaubt sind.). Daher muss z.B. beim Einfügen eines Elements in einer Liste eine neue Liste erzeugt werden. Dies hat Vor- und Nachteile: Es gilt Persistenz, denn die alte Liste existiert auch noch. Das Erstellen der neuen Liste kann mittels Sharing so implementiert werden, dass gleiche Teile der alten und der neuen Liste nur einmal (gemeinsam) im Speciher gehalten werden. Eine automatische Garbage Collection entfernt nicht mehr verwendete alte Versionen aus dem Speicher. Vorteile bei Verwendung von unveränderlichen Datenstrukturen ist, dass typische Fallen von veränderlichen Strukturen nicht auftreten (wie z.B. eine Veränderung während der Iteration in einer for-Iteration in Java), dass beim Rekursionsrücksprung die alten Zustände noch vorhanden sind und daher nicht wiederhergestellt werden müssen und bei Nebenläufigkeit ist kein explizites Kopieren von Datenstrukturen notwendig, wenn nebenläufige Threads diese Daten gleichzeitig verändern möchten.

- Starke Typisierung: Funktionale Sprachen haben fast immer ein starkes Typsystem. Das Typsystem dient dabei der Aufdeckung von Fehlern durch den Compiler, der Auflösung von Überladung, der Dokumentation von Funktionen und der Suche passender Funktionen. Typen können oft automatisch inferiert werden. Fortgeschrittene Typsysteme dienen der Verifikation (z.B. in Agda).
- 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 (Hug89).

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: Funktionale Programmiersprache, mit einem Typsystem mit sogenannten abhängigen Typen (dependent types). Kommt mit einem automatischen Beweissystem zur Verifikation von Programmen. Hat eine Haskell-ähnliche Syntax aber eine andere Semantik. Die Sprache ist total, d.h. Funktionen müssen terminieren https://wiki.portal.chalmers.se/agda/.

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

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

Lean: (2013 von Leonardo de Moura, Microsoft Research) Strikte funktionale Programmiersprache, die auch als interaktiver Theorembeweiser verwendet werden kann. Hierfür werden abhängige Typen verwendet. Insbesondere kann Lean zur Verifikation verwendet werden, indem Eigenschaften von Funktionen in Lean selbst bewiesen werden können. https://lean-lang.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.

1.4 Weiteres Material

1.4.1 Literatur

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.
Thinking Functionally with Haskell,
Cambridge University Press, 2014
Gutes Buch, deckt einige Themen der Vorlesung ab.

Bird, R.
Introduction to Functional Programming using Haskell.
Prentice Hall PTR, 2 edition, 1998.
Älteres 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

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.

Lipovaca, M.
Learn You a Haskell for Great Good! A Beginner's Guide
No Starch Press, 2011.

Online-Version:http://learnyouahaskell.com/.
umfassende Einführung in Haskell.

Marlow, S.
Parallel and Concurrent Programming in Haskell
O'Reilly, ISBN 9781449335939, 2013.
Online-Version: www.oreilly.com/library/view/parallel-and-concurrent/9781449335939 .
beschreibt paralleles und nebenläufiges Programmieren in Haskell.

O'Sullivan, B., Goerzen, J., & Stewart, D.
Real World Haskell.O'Reilly Media, Inc, 2008.
Online-Version: http://book.realworldhaskell.org/read/
Programmierung in Haskell für Real-World Anwendungen

Allen, C. and Moronuki, J. and Syrek, S.
Haskell Programming from First Principles, Lorepub LLC, ISBN 9781945388033, 2019. Webseite: haskellbook.com/, Umfangreiches und aktuelles Buch zur Haskell-Programmierung in der realen Welt.

Bragilevsky,V
Haskell in Depth, Manning, ISBN 9781617295409, 2021. Aktuelles Buch zur Haskell-Programmierung auch mit vertiefenden Themen.

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.

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.

Snoyman, M.
Developing Web Applications with Haskell and Yesod,
O'Reilly, 2012. ISBN 1449316972.
Online-Version:https://www.yesodweb.com/book.
Programmieren mit dem Web-Framework Yesod.

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 sind auch:

Pierce, B. C.,
Types and programming languages.
MIT Press, Cambridge, MA, USA, 2002. Typen und Typsysteme werden anhand der Programmiersprache ML erläutert.

Maguire, S.,
Thinking with Types – Type-Level Programming in Haskell, publisher = https://leanpub.com/thinking-with-types, 2019. Programmierung auf der Typebene wird anhand von (Erweiterungen des Typsystems von) Haskell vorgestellt.

1.4.2 Webseiten, etc

Es gibt zahlreiche Webseiten zu Haskell. Ein zentrale Übersicht bzw. Anlaufstelle ist das Haskell Wiki: https://wiki.haskell.org/Haskell. Es gibt aber auch spezialisierte Suchmaschinen, z.B. **Hoogle** https://hoogle.haskell.org/. Diese durchsucht die Paket-Dokumentationen. Man kann z.B. auch nach Typen suchen, um Funktionen dieses Typs zu finden.

Bei speziellen Fragen sind die Haskell-Mailinglisten eine gute Anlaufstelle, um Fragen zu stellen:

- haskell-cafe@haskell.org
- beginners@haskell.org
- haskell@haskell.org

Auch auf Code-Probleme spezialisierte Webseiten können Haskell-Programmierer helfen:
http://stackoverflow.com/questions/tagged/haskell

2 Eine Einführung in wichtige Konstrukte der Programmiersprache Haskell

In diesem Kapitel führen wir wesentliche Konstrukte der funktionalen Programmiersprache Haskell ein.

2.1 Einführung und Werkzeuge

Haskell (https://www.haskell.org/) ist eine Effekt-freie rein funktionale Sprache mit verzögerter Auswertung. Sie ist benannt nach dem US-amerikanischen Logiker Haskell B. Curry (1900-1982). Die beiden neuesten Standards der Sprachen sind Haskell 98 und Haskell 2010, danach wurden keine Standards mehr veröffentlicht. Allerdings gibt es sogenannte GHC Proposals https://github.com/ghc-proposals/ghc-proposals die Sprachanpassungen und Erweiterungen anhand des Haskell Compilers GHC festlegen. Die Proposals GHC 2024 und GHC 2021 können als Aktualisierungen des Haskell 2010-Standards verstanden werden.

Es gibt verschiedene Implementierungen von Haskell, wobei der Glasgow Haskell Compiler (GHC) https://www.haskell.org/ghc/ der wichtigste ist. Die Dokumentation des Compilers ist online via https://downloads.haskell.org/ghc/latest/docs/html/users_guide/ verfügbar, die Dokumentation der Standardbibliotheken ist unter http://www.haskell.org/ghc/docs/latest/html/libraries/ zu finden. Die Installation von Haskell und das Setup des Editors ist unter https://www.haskell.org/get-started/ beschrieben.
Neben dem Compiler GHC wird auch der Interpreter GHCi mit installiert. Dort kann man direkt Ausdrücke eingeben und auswerten lassen:

```
$>  ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> 1 + 2
3
ghci> 10 + 3 * 5
25
ghci> 14 `mod` 6
2
```

Mit den Pfeiltasten kann man die vorherigen Eingaben durchblättern. Zur Steuerung des Interpreters stehen Befehle zur Verfügung, welche alle mit einem Doppelpunkt beginnen, z.B. `:?` für die Hilfe. Befehle kann man abkürzen, z.B. kann man `:quit` also auch mit `:q` den Interpreter verlassen. Überlicherweise macht man Programmdefinitionen mit einem gewöhnlichen Texteditor in einer separaten Datei und lädt diese dann in den Interpreter:

```
:l datei.hs -- lade Definition aus Datei datei.hs
:r          -- erneut alle offenen Dateien einlesen
```

Haskell-Quellcodedateien enden mit `.hs` oder mit `.lhs`, wobei letztere Quellcode als *Literate-Haskell-Programm* erwarten. In normalen Haskell-Quellcodedateien (deren Dateiendung `.hs` lautet) werden Kommentare explizit gekennzeichnet: Mit `--␣` wird ein Zeilenkommentar eingeleitet, alles was danach kommt, bis zum Ende der Zeile wird ignoriert. Mit `{-` und `-}` wird ein mehrzeiliger Kommentar geklammert. Alles was zwischen dem öffnenden `{-` und dem schließenden `-}` steht, wird als Kommentar behandelt (und vom Compiler ignoriert).

Bei *Literate-Haskell-Programmen* ist die Idee, dass sämtliche Zeilen in der Datei als Kommentar angesehen werden, solange sie nicht als Code gekennzeichnet werden. Im sogenannten Bird-Stil (benannt nach Richard Bird) werden Zeilen, die mit `>␣` beginnen, als Codezeilen interpretiert. Ein solcher Code-Block muss mit einer Leerzeile am Anfang und am Ende eingeleitet werden (ansonsten wird der Fehler „unlit: Program line next to comment gemeldet“). Ein Beispiel für ein Literate-Haskell-Programm im Bird-Style ist:

```
Hier steht beliebiger Kommentar, der
sich auch über mehrere Zeile erstrecken
kann.

> meineFunktion [] = putStrLn "Hallo Welt"
> meineFunktion (_:rs) = do
>                        putStrLn "Hallo Welt"
>                        meineFunktion rs

Hier steht erneut ein Kommentar.
```

Im LaTeX-Stil werden Code-Blöcke durch `\begin{code}` und `\end{code}` gekennzeichnet, obiges Programm wird dann als

```
Hier steht beliebiger Kommentar, der
sich auch über mehrere Zeile erstrecken
kann.
\begin{code}
meineFunktion [] = putStrLn "Hallo Welt"
meineFunktion (_:rs) = do
                       putStrLn "Hallo Welt"
                       meineFunktion rs
\end{code}
Hier steht erneut ein Kommentar.
```

Haskell beachtet die Groß-/Kleinschreibung und es ist daher nicht egal, ob man etwas klein oder groß schreibt. Ebenso ist Haskell „whitespace“-sensitiv, d.h. insbeondere Veränderungen

an Leerzeichen, Tabulatoren und Zeilenumbruch können Fehler verursachen. Statt Einrückung kann man auch geschweifte Klammern `{ }` und Semikolons `;` verwenden. Es empfiehlt sich Tabulatoren nicht zu verwenden, sondern (am besten durch den Text-Editor) durch Leerzeichen zu ersetzen. Hinweise zur Verwendung von Editoren und IDEs findet man unter `https://wiki.haskell.org/IDEs`.

Zudem gilt in Haskell im Allgemeinen als Daumenregel: Beginnt die nächste Zeile in einer Spalte weiter rechts als die vorherige, dann geht die vorherige Zeile weiter, beginnt die nächste Zeile in der gleichen Spalte wie die vorherige, dann beginnt das nächste Element eines Blocks, und beginnt die Zeile weiter links als die vorherig, dann ist der Block beendet.

2.2 Einführendes zu Typen in Haskell

2.2.1 Syntax und Sprechweisen

Ein Typ ist eine Menge von Werten, z.B. bezeichnet der Typ `Int` meistens die Menge der ganzen Zahlen von -2^{63} bis $2^{63}-1$. In Haskell beginnen Typnamen stets mit einem Großbuchstaben und Typvariablen beginnen stets mit einem Kleinbuchstaben.
Im GHCi kann mit `:type` *Ausdruck* der Typ eines Ausdrucks angezeigt werden

```
ghci> :type 'a'
'a' :: Char
```

Gilt `e :: T`, so sagt man „Ausdruck `e` hat Typ `T`“ oder alternativ „Ausdruck `e` ist vom Typ `T`“ Mit dem Befehl `:set +t` wird der Typ jedes ausgewerteten Ausdrucks angezeigt im GHCi angezeigt, mit `:unset +t` stellt man das wieder ab.

Die Syntax von Typen (ohne Typklassenbeschränkungen, diese werden wir später erläutern) in Haskell kann durch folgende kontextfreie Grammatik beschrieben werden:

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1\ \texttt{->}\ \mathbf{T}_2$$

Dabei steht TV für eine Typvariable (die mit einem Kleinbuchstaben beginnen muss) und TC ist ein Typkonstruktor mit Stelligkeit n (Typkonstruktoren beginnen mit einem Großbuchstaben, oder – wenn sie infix verwendet werden, mit einem Doppelpunkt). Das Konstrukt $\mathbf{T}_1$ `->` $\mathbf{T}_2$ stellt einen Funktionstypen dar, d.h. es ist der Typ von Funktionen die als Eingabe Werte des Typs $\mathbf{T}_1$ bekommt und als Ausgabe Werte des Typs $\mathbf{T}_2$ liefert

Man unterscheidet noch die Sprechweisen für verschiedene Typen. Eingebaute, vordefinierte Typen, nennt man *Basistypen*. Das entspricht in obiger Syntax den nullstelligen Typkonstruktoren. Diese haben keine Typvariablen und sind auch nicht aus anderen Typen zusammengesetzt. Ein Beispiel ist der Typ `Int` für Ganzzahlen beschränkter Größe, oder auch der Typ `Bool` für Wahrheitswerte. Grundtypen oder auch monomorphe Typen sind Typen, die keine Typvariablen enthalten. D.h. jeder Basistyp ist auch ein Grundtyp, aber z.B. sind die Typen `Baum Int` (ein Baum mit Zahlen als Blattmarkierungen), `[(Int,Bool)]` (eine Liste von Paaren bestehend aus

Zahlen und Wahrheitswerten) oder `Int -> Bool` (eine Funktion von `Int` nach `Bool`) ebenfalls Grundtypen, da sie keine Typvariablen enthalten. Ein *polymorpher Typ* darf auch Typvariablen enthalten, z.B. ist `[a]` oder auch `a -> a` ein polymorpher Typ. Einen polymorphen Typ kann man sich wie ein Typschema vorstellen: Er repräsentiert alle Grundtypen, die man erzeugen kann, indem man die Typvariablen durch Grundtypen ersetzt.

2.2.2 Wichtige Basistypen und weitere Typen

2.2.2.1 Arithmetische Operatoren und Zahlen

Haskell verfügt über eingebaute Typen für

- ganze Zahlen beschränkter Größe `Int`, deren Bereich ist maschinenabhängig, er reicht jedoch mindestens von -2^{29} bis $2^{29}-1$. Es wird nicht auf Überläufe geprüft.
- ganze Zahlen beliebiger Länge `Integer`,
- Gleitkommazahlen `Float`,
- Gleitkommazahlen mit doppelter Genauigkeit `Double`,
- 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). Zur Verwendung muss das Modul `Data.Ratio` immportiert werden, denn dort wird der Operator `%` definiert.

Die üblichen Rechenoperationen sind verfügbar für alle diese Zahlen:

- `+` 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 5). 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)`.

2.2.2.2 Weitere wichtige Typen: Bool, Char, String

Neben Zahlen gibt es noch weitere wichtige grundlegende Typen:

`Bool` Boolesche (logische) Wahrheitswerte: `True` und `False`

`Char` Unicode Zeichen, z.B. `'q'`. Diese werden immer in Apostrophen eingeschlossen.

`String` Zeichenketten, z.B. `"Hallo!"`. Diese werden immmer in Anführungszeichen eingeschlossen.

Von diesen drei Typen ist lediglich `Char` wirklich speziell eingebaut, die beiden anderen kann man leicht selbst definieren:

```
type String = [Char]      -- Typsynonym
data Bool = True | False
```

Die Standardbibliothek (die sogenannte Prelude) definiert u.a. folgende Funktionen

- `&&` Konjunktion, logisches Und
- `||` Disjunktion, logisches Oder
- `not` Negation, Verneinung
- `==` Test auf Gleichheit
- `/=` Test auf Ungleichheit

```
ghci> (True || False) && (not True)
False
ghci> True == False
False
ghci> False /= True
True
```

Die üblichen Vergleichsoperationen für Zahlen sind auch 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: Während Funktionsnamen mit einem Kleinbuchstaben oder _ beginnen müssen, bestehen Operatoren nur aus Symbolen, die keine alphanumerischen Zeichen sind. Operatoren werden standardmäßig infix verwendet, während Funktionen standardmäßig präfix verwendet werden. Man kann in Haskell 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`

2.3 Funktionen und Ausdrücke

Definition 2.3.1 (Funktion)**.** *Eine* partielle Funktion $f : A \to B$ *ordnet einer Teilmenge* $A' \subset A$ *einen Wert aus B zu, und ist ansonsten undefiniert.*

Wir bezeichnen A als Quellbereich*,* A' *als* Definitionsbereich *und B als* Wertebereich*. Für* totale *Funktionen gilt* $A = A'$.

Im allgemeinen möchte man möglichst totale Funktionen in Haskell verwenden, wenn dies nicht möglich ist, sollten die undefinierten Fälle duch `error` abgefangen werden.

2.3.1 Funktionen und Funktionstypen

In Haskell werden benannte Funktionen durch Gleichungen definiert:

```
double1 x = x + x            -- Funktion mit 1 Argument
fun x y z = x + y * double1 z         -- mit 3 Argumenten
add       = \x y -> x + y    -- Anonyme Fkt. 2 Argumente

twice f x = f x x            -- Higher-order function
double2 y = twice (+) y
double3   = twice (+)        -- Partielle Applikation
```

In einer Haskell-Datei müssen alle Top-Level Definitionen in der gleichen Spalte stehen. Die Funktionsanwendung geschieht durch Leerzeichen (z.B. wendet `foo 1 2 3` die Funktion `foo` auf die Argumente `1`, `2` und `3` an). Die Anwendung ist dabei links-assoziativ, z.B. entspricht `foo 1 2 3` vollgeklammert dem Ausdruck `(((foo 1) 2) 3)` Durch Klammerung kann ein Infix-Operator wie + als Präfixoperator verwendet werden, z.B. entspricht `(+) 1 2` dem Ausdruck `1 + 2`. Umgekehrt kann ein Präfix-Operator auch infix verwendet werden, indem der Operator in Back-Ticks eingasst wird. Z.B. entspricht `1 ‘add‘ 2` dem Ausdruck `add 1 2`.

Selbst-definierte infix-Operatoren sind möglich, diese dürfen jedoch nicht mit einem Buchstaben oder einer Zahl beginnen. Z.B. können wir definieren

```
x !!! y = x*x + y*y
```

Funktionen bilden einen Argumenttyp auf einen Ergebnistyp ab, wobei sogenanntes Currying verwendet wird: Statt mehrstellige Funktionen $f : (A_1 \times \cdots \times A_n) \to B$ zu verwenden, übergibt man die Parameter $A_1, \ldots, A_n$ einzeln als $f : A_1 \to A_2 \to \cdots \to A_n \to B$. D.h. f wird wie eine einstellige Funktion gesehen, die ein Argument vom Typ A_1 erhält und eine *Funktion* liefert, die noch $n-1$ Argumente erwartet und ein Ergebnis vom Typ B liefert. Z.B.

```
foo :: Int -> Int -> Int
foo x y  = x+2*y
```

Die Klammerkonvention ist, dass Funktionstypen implizit rechtsgeklammert sind. Z.B. ist `Int -> Int -> Int` äquivalent zu `Int -> (Int -> Int)` und *verschieden* zu `(Int -> Int) -> Int`.

Durch die Verwendung von Currying, darf man in Haskell partiell anwenden. Z.B. ist die partielle Anwendung `(foo 1)` eine Funktion des Typs `Int -> Int`. Funktionen sind also normale Werte in einer funktionalen Sprache.

Allgemein ist eine Funktionsdefinition in Haskell von der Form

funktionsName `::` *Typ*$_1$ `->` *Typ*$_2$ `->` `...` `->` *Typ*$_n$ `->` *Ergebnistyp*
funktionsName *var*$_1$ *var*$_2$ ... *var*$_n$ `=` *expr*

Dabei beginnt der Funktionsname mit einem Kleinbuchstaben. Die Typdeklaration in der ersten Zeile optional, sie hilft jedoch oft: Einerseits dient sie der Dokumentation des Quellcodes und andererseits, erhält man oft bessere Fehlermeldungen des Compilers, wenn man den erwarteten Typ vorgibt. Die zweite Zeile obiger Funktionsdefinition ist die eigentliche Definition, wobei $var_1, var_2, \ldots, var_n$ die formalen Parameter sind und *expr* der Funktionsrumpf.

Anstelle von Variablen können auch Patterns verwendet werden. Patterns sind Daten (wie z.B. `True`) aber auch Teile von Daten, wobei unbekannte Teile Variablen sind (wir sehen noch viele Patterns später)- Die Wildcard _ ist ei Pattern, das immer passt.

Für Funktionen können mehrere Definitionszeilen angegeben werden (was z.B. sinnvoll ist, um mit Patterns eine Fallunterscheidung durchzuführen), und mithilfe von *Guards* können Bedingungen geprüft werden, anhand derer sich der jeweils gültige Funktionsrumpf bestimmt. Die allgemeinere Syntax für Funktionsdefinitionen ist daher

funktionsName $pat_1 \ldots pat_n$
    | $guard_1$ = e_1
    | ...
    | $guard_n$ = e_n

Hierbei sind $guard_1$ bis $guard_n$ Boolesche Ausdrücke, die die Variablen der Patterns pat_1,...,pat_n benutzen dürfen. Die Guards werden von oben nach unten ausgewertet. Liefert ein Guard `True`, so wird die entsprechende rechte Seite e_i als Resultat übernommen. Der Guard `otherwise` ist nur ein Synonym für `True`d.h.

```
otherwise = True
```

ist vordefiniert.

Die Abarbeitung einzelner Zeilen geschieht von oben nach unten. Entsprechend bestimmt das erste passende Pattern bzw. der erste Guard, der zu `True` auswertet, den Wert der Funktion.

Z.B. definiert

```
even 0 = True
even 1 = False
even x = if x < 0 then even (abs x) else even (x-2)
```

eine Funktion, die für Ganzzahlen prüft, ob diese gerade sind. Die Definition

```
even' x = even' (x-2)
even' 0 = True
even' 1 = False
```

ist nicht korrekt: Die Funktion `even'` terminiert nie, da das Pattern der ersten Zeile (die Variable `x`) immer passt. D.h. die zweite Zeile wird niemals erreicht. Hingegen ist die Variante

```
even'' 0 = True
even'' 1 = False
even'' x
 | x > 1 = even'' (x-2)
even'' x = even'' (abs x)
```

korrekt: Wenn `x` passt, aber der Guard `x > 1` falsch ist, wird das nächste Pattern probiert. Dies ist die Wildcard _, die immer passt.

2.3.2 Anonyme Funktionen

Es ist oft praktisch, einmal verwendeten Funktionen keinen besonderen Namen zu geben. Aus dem Lambda-Kalkül (Alonzo Church, 1936), der mathematischen Grundlage der funktionalen

Programmierung, nehmen wir daher die λ-Notation für anonyme Funktionen: $\lambda x_1, \ldots, x_n.e$ definiert eine anonyme Funktion (auch *Abstraktion* genannt), deren formale Parameter $x_1, \ldots, x_n$ sind und deren Rumpf e ist. Z.B.:

Identitätsfunktion	$\lambda x.x$
Nachfolgerfunktion	$\lambda x.x + 1$
Wurzelfunktion	$\lambda y.\sqrt{y}$
Dreistelliges Plus	$\lambda x, y, z.x + y + z$

In Haskell schreiben wir anstelle von λ einen Backslash \, anstelle des . den Pfeil ->, und mehrere formale Parameter werden durch Leerzeichen anstelle der Kommata getrennt. D.h. die Beispiele werden in Haskell geschrieben als:

```
\x -> x
\x -> x + 1
\y -> sqrt y
\x y z -> x + y + z
```

2.3.3 Ausdrücke

Ausdrücke, wie z.B. auf den rechten Seiten von Funktionsdefinitionen verwendet werden, werden in Haskell durch Anwendung von Funktionen auf Argumente (geschrieben durch Leerzeichen), sowie durch die folgenden Syntaxkonstrukte gebildet:

- `if-then-else`-Ausdrücke der Form

  `if` *Ausdruck* `then` *Ausdruck* `else` *Ausdruck*
- `case`-Ausdrücke zum Zerlegen von Daten, von der Form

  `case` *Ausdruck* `of` *Alternativen*

  Wir erläutern `case`-Ausdrücke in Abschnitt 2.4.5 genauer.
- `let`-Ausdrücke der Form

  `let` *Bindungen* `in` *Ausdruck*

  wobei *Bindungen* Bindungen der Form *Funktionsname* $par_1 \ldots par_n$ = *Ausdruck* sind (die einzelnen Bindungen sind durch Einrückung, oder alternativ Semikolons, getrennt). Die Bindungen in `let`-Ausdrücken sind rekursiv, d.h. der Gültigkeitsbereich der Funktionsnamen sind alle rechten Seiten der Bindungen sowie der `in`-Ausdruck.
- `do`-Notation: Diese erläutern wir später, da sie an dieser Stelle nicht von zentraler Bedeutung ist.
- `seq`-Ausdrücke sind von der Form `seq` e_1 e_2. Die Semantik ist, dass `seq` e_1 e_2 gleich zu e_2 ist, falls die Auswertung von e_1 terminiert. Operational wird daher in `seq` e_1 e_2 zunächst e_1 ausgewertet und bei Erfolg dann e_2.

- Datenkonstruktoren, Listen, List-Comprehensions: Auch diese werden wir an entsprechender Stelle erläutern

2.3.3.1 `where`-Klauseln

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

```
summe 1 = 1
summe n = n+sumrek
 where
  sumrek = summe (n-1)
```

Allerdings ist zu beachten, dass (`let ... in` *e*) einen Ausdruck darstellt, während *e* `where` ... kein Ausdruck ist. Das `where` ist gültig für die Funktionsdefinition kann und daher für alle Guards wirken. 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 einer `where`-Klausel 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 in der `where`-Klausel gebunden ist. Oft ist die Verwendung von `let`- oder `where` jedoch auch Geschmackssache.

2.4 Algebraische Datentypen in Haskell

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

2.4.1 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 = Konstante$_1$ | Konstante$_2$ | ... | Konstante$_N$

Ein Beispiel für einen selbst-definierten Datentyp ist der Typ Wochentag:

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

Bemerkung 2.4.1. *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: error:
    • No instance for (Show Wochentag) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
```

Mit Show-Instanz kann der Interpreter die Werte anzeigen:

```
*Main> Montag
Montag
```

Die Werte eines Summentyps können mit mit einem `case`-Ausdruck abgefragt werden, und eine entsprechende Fallunterscheidung kann damit durchgeführt werden. 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 auch, eine default-Alternative im `case` zu verwenden. Dabei wird anstelle des Patterns 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
```

Haskell bietet auch die Möglichkeit (verschachtelte) Patterns 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.

2.4.2 Produkttypen

Produkttypen fassen verschiedene Werte zu einem neuen Typ zusammen. Die bekanntesten Produkttypen sind Paare und mehrstellige Tupel.

Definition 2.4.2 (Kartesisches Produkt)**.** *Sind $A_1, \ldots A_n$ Mengen, so ist das kartesische Produkt definiert als* $A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) \mid a_i \in A_i \text{ für } i = 1 \ldots n\}$

Die Elemente von $A_1 \times \cdots \times A_n$ heißen allgemein *n-Tupel*, spezieller auch Paare, Tripel, Quadrupel, ...
In Haskell schreiben wir Tupelausdrücke und Tupeltypen mit runden Klammern und Kommas. $\mathbb{Z} \times \mathbb{Z}$ wird zu `(Int,Int)`.

```
ghci> :t (True, 'a', 7)
(True, 'a', 7) :: (Bool, Char, Int)

ghci> :t (4.5,"Hi!")
(4.5,"Hi!") :: (Double, String)
```

Das 0-Tupel ist ebenfalls in Haskell erlaubt: `() :: ()`. Der Typ `()` wird als *Unit*-Typ bezeichnet, und hat nur den einzigen Wert `()`. Aus dem Kontext wird fast immer klar, ob mit `()` der Typ oder der Wert gemeint ist.

Mit `data` können auch Produkttypen definiert werden (die keine Tupelsyntax verwenden). Die Syntax hierfür ist

`data` Typname = KonstruktorName Typ_1 Typ_2 ... Typ_N

Ein Beispiel ist der folgende 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) -> Student name' vorname mnr
```

Alternativ mit Patterns 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
```

Als weiteres Beispiel definieren wir einen Datentyp für Früchte:

```
data Frucht = Apfel (Int,Double)
            | Birne Int Double
            | Banane Int Int Double
```

Allgemein kann man einen solchen Typ definieren als

`data Typ =` Typ_1 `|` ... `|` Typ_n

wobei Typ_1,...,Typ_n selbst komplexe Typen sind, z.B. Produkttypen. Man nennt diese Klasse von Typen (aufgrund der mit | getrennten Alternativen) oft auch *Summentypen*.

2.4.2.1 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
```

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

```
nameMitA Student{name = 'A':xs} = True
nameMitA _ = False
```

Diese Definition ist äquivalent zur Definition

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

Die Record-Syntax kann auch benutzt werden, um „Updates“ durch zu führen[1]. 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}
```

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

2.4.3 Parametrisierte Datentypen

Haskell bietet die Möglichkeit, Datentypen mit (polymorphen) Parametern zu versehen. Allgemein ist die Syntax der Datentypdeklaration:

`data` *Typname* $par_1 \ldots par_m$ = *Konstruktor*$_1$ $arg_{1,1} \ldots arg_{1,i_1}$ | ... | *Konstruktor*$_n$ $arg_{n,1} \ldots arg_{n,i_n}$
`deriving` $(class_1, \ldots, class_l)$

Der Typname muss mit einem Großbuchstaben beginnen, die Typparameter sind optional, ebenso ob es mehrere Alternativen gibt, Konstruktoren müssen ebenfalls mit einem Großbuchstaben beginnen und erwarten als Argumente eine beliebige Anzahl von Argumenten, wobei ein Argument ein bekannter Typ oder einer der Typparameter ist. Die optionale `deriving`-Klausel mit einer Liste von Typklassen, erlaubt es automatisch Instanzen des Typs für Typklassen (siehe Abschnitt 5) erstellen zu lassen.

2.4.3.1 Der Datentyp Maybe

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. können wir eine Funktion für die oben definierten Früchte programmieren, die nur ein sinnvolles Ergebnis bei Bananen liefert, aber anderes Obst mit `Nothing` als Ergebnis ebenfalls sinnvoll behandelt.

```
getKruemmung :: Frucht -> Maybe Double
getKruemmung (Banane _ _ k)  = Just k
getKruemmung _               = Nothing
```

Weitere (vordefinierte) Funktionen für den `Maybe`-Typen sind:

```
isJust         :: Maybe a -> Bool
isJust Nothing = False
isJust _       = True

fromMaybe :: a -> Maybe a -> a
fromMaybe standardWert Nothing  = standardWert
fromMaybe      _       (Just x) = x

catMaybes :: [Maybe a] -> [a]
catMaybes ls = [x | Just x <- ls]
```

2.4.3.2 Der Datentyp Either

Polymorphe Datentypen können auch mehrere Typparameter haben. Der Datentyp `Either` ist ein wichtiges Beispiel:

```
data Either a b  =  Left a | Right b
```

Zum Beispiel kann `Either a String` anstelle von `Maybe a` für die Rückgabe von Fehlermeldungen verwendet werden, ohne die Berechnung abzubrechen. Betrachte z.B.:

```
getKruemmung' :: Frucht -> Either String Double
getKruemmung' (Banane _ _ k)  = Right k
getKruemmung' _               = Left "Eingabe ist keine Banane."

myDiv :: Double -> Double -> Either String Double
myDiv x 0 = Left "Error: Division by 0"
myDiv x y = Right $ x / y
```

Im Grunde ist `Either` ist der Typ für disjunkte Vereinigungen:

$$A_1 \dot\cup A_2 = \left\{(i,a) \;\middle|\; a \in A_i\right\}$$

Zum Beispiel ist $\{\diamond,\heartsuit\}\dot\cup\{\heartsuit,\clubsuit,\spadesuit\} = \{(1,\diamond),(1,\heartsuit),(2,\heartsuit),(2,\clubsuit),(2,\spadesuit)\}$.

Für endliche Mengen gilt $|A_1 \dot\cup A_2| = |A_1| + |A_2|$.

Das Modul `Data.Either` definiert neben dem Datentyp noch einige hilfreiche Funktionen, u.a.:

```
isRight :: Either a b -> Bool
isRight (Left  _) = False
isRight (Right _) = True

lefts   :: [Either a b] -> [a]
lefts x = [a | Left a <- x]

partitionEithers :: [Either a b] -> ([a],[b])
partitionEithers []      = ([],[])
partitionEithers (h : t)
   | Left  l <- h = (l:ls,   rs)
   | Right r <- h = (  ls, r:rs)
  where (ls,rs) = partitionEithers t
```

2.4.4 Rekursive Datentypen

Es ist in Haskell auch möglich, rekursive Datentypen zu definieren. Das prominenteste Beispiel hierfür sind Listen. Rekursive Datentypen zeichnen sich dadurch aus, dass der zu definierende Typ selbst 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.

```
[1,2,3] :: [Int]
[1,2,2,3,3,3] :: [Int]
["Hello","World","!"] :: [[Char]]
```

Eine Liste kann sogar ganz leer sein, geschrieben `[]`.

Während in einer Liste die Anzahl der Elemente variabel ist, muss der Typ der Elemente der gleich sein. Im Gegensatz dazu ist bei Tupeln die Anzahl der Elemente fest und bekannt, aber die Typen der verschiedenen Komponenten dürfen verschieden sein.

Man kann Listen und Tupel man beliebig ineinander verschachteln:

```
[(1,'a'),(2,'z'),(-4,'w')] :: [(Integer, Char)]
[[1,2,3],[],[4]] :: [[Integer]]
(4.5,[(True,'a',[5,7],())]) :: (Double, [(Bool, Char, [Integer], ())])
```

Beachte, dass `[]` und `[[]]` und `[[],[]]` und `[[[]]]` und `[[],[[]]]` alles verschiedene Werte sind.

Haskell erlaubt verschachtelte Patterns, z.B. kann man definieren

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

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

2.4.5 Case-Ausdrücke

Die Syntax von `case`-Ausdrücke zum Zerlegen von Daten war bereits definiert als

`case` *Ausdruck* `of` *Alternativen*

Dabei sind *Alternativen* `case`-Alternativen der Form *Pattern* `->` *Ausdruck* oder *Pattern Match*. In *Patterns* sind auch verschachtelte Patterns erlaubt (z.B. ist `((x,y):xs)`) ein solches verschachteltes Pattern, dass für eine nicht-leere Liste von Paaren erfolgreich gematcht werden kann.

Match bezeichnet eine Folge der Form

| *Guard*$_1$ -> *Ausdruck*$_1$
| ...
| *Guard*$_n$ -> *Ausdruck*$_n$

Wenn das Pattern in *Patterns Match* passt, dann bestimmt sich der Wert des `case`-Ausdrucks durch den ersten zu `True` auswertenden Guard. Falls kein Guard zu `True` auswertet, wird diese `case`-Alternative verworfen. Bei der Auswertung des Guards können zudem neue Namen gebunden werden, die im Ausdruck *Ausdruck*$_i$ verwendet werden können.

Ein Beispiel ist

```
myFun frucht p personendb =
  case frucht of
    Banane l h kruemmung
      | Just x <- lookup p personendb -> Right x
    _  -> Left "Fehler"
```

In diesem Fall wird der Eintrag `x` für `p` nur dann zurück gegeben, wenn die übergebene Frucht eine Banane war und `(p,x)` in der Liste `personendb` vorhanden ist. In allen anderen Fällen wird `Left "Fehler"` zurück gegeben.

Z.B.

```
*Main> myFun (Banane 1 2 3) 1 [(1,"Horst")]
Right "Horst"
*Main> myFun (Apfel (1,2)) 1 [(1,"Horst")]
Left "Fehler"
*Main> myFun (Banane 1 2 3) 10 [(1,"Horst")]
Left "Fehler"
```

Schließlich kann auch eine Default-Alternative der Form `x` -> *Ausdruck* angegeben werden. Diese match immer. Ein `case`-Ausdruck der Form `case` *Ausdruck*$_1$ `of x` -> *Ausdruck*$_2$ hat nur eine Default-Alternative und wird im GHC ersetzt durch `let x`=Ausdruck$_1$ `in` Ausdruck$_2$. Eine Konsequenz daraus ist: Wenn `x` nicht in Ausdruck$_2$ verwendet wird, dann wird Ausdruck$_1$ nicht ausgewertet.

2.4.6 Typdefinitionen mit data, type und 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 = filter nameMitA
```

Für die Ausführung von Programmen ist es unerheblich, ob das Typsynonym oder der ursprüngliche Typ verwendet wird (der Compiler unterscheidet diese nicht). GHC/GHCi kann Typabkürzungen in Fehlermeldungen ignorieren, d.h. GHCi gibt dann `[Char]` anstelle von `String` aus.

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 scheint aus Programmiersicht 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. Der GHC tut dies auch tatsächlich, indem die zusätzlich Verpackung bei mit `newtype` definiertem Typ stets wegwirft. Man bemerkt diesen Unterschied, wenn man folgendes probiert

```
*Main> case undefined of {St _ -> 1}
1
*Main> case undefined of {St'' _ -> 1}
*** Exception: Prelude.undefined
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/Err.hs:79:14 in base:GHC.Err
  undefined, called at <interactive>:12:1 in interactive:Ghci2
```

Im ersten Fall eliminiert der Haskell-Compiler die Verpackung `ST` ... und der Code verhält sich genauso wie `case undefined { _ -> 1 }`: Durch das Wildcard-Pattern `_` wird `undefined` ignoriert und direkt 1 zurückgelieft. Im zweiten Fall prüft die Auswertung, ob der Ausdruck `undefined` von der Form `ST''` *x* ist, und daher direkt `undefined` auswertet.

Eine andere Frage ist, warum man `newtype` anstelle von `type` 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.

2.5 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.

2.5.1 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

$$\texttt{module}\ \textit{Modulname}\,\texttt{(}\textit{Exportliste}\texttt{)}\ \ \texttt{where}\quad \left.\begin{array}{l}\textit{Modulimporte,}\\ \textit{Datentypdefinitionen,}\\ \textit{Funktionsdefinitionen, }\ldots\end{array}\right\} \mathit{Modulrumpf}$$

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[2].

[2] Bei hierarchischen Modulen muss der Dateipfad dem Modulnamen entsprechen, siehe Abschnitt 2.5.4)

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:

Beispiel 2.5.1. *Das Modul `Spiel` sei definiert als:*

```
module Spiel where
data Ergebnis = Sieg | Niederlage | Unentschieden
berechneErgebnis a b
 | a > b = Sieg
 | a < b = Niederlage
 | otherwise = Unentschieden
istSieg Sieg = True
istSieg _    = False
istNiederlage Niederlage = True
istNiederlage _          = False
```

2.5.2 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 2.5.1 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 2.5.1 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 2.5.1 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 2.5.1 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`.

2.5.3 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.

```
ghci> :l C.hs
[1 of 3] Compiling B                ( B.hs, interpreted )
[2 of 3] Compiling A                ( A.hs, interpreted )
```

```
[3 of 3] Compiling C                ( C.hs, interpreted )

C.hs:4:5: error:
    Ambiguous occurrence ‘f’
    It could refer to either ‘A.f’,
                             imported from ‘A’ at C.hs:2:1-8
                             (and originally defined at A.hs:2:1)
                          or ‘B.f’,
                             imported from ‘B’ at C.hs:3:1-8
                             (and originally defined at B.hs:2:1)
```

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
```

führt zu

```
ghci> :l C

C.hs:3:5: error:
    • Variable not in scope: f :: Integer -> Integer -> t
    • Perhaps you meant ‘A.f’ (imported from A)
Failed, modules loaded: A.
```

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`

2.5.4 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[3]“ sind mithilfe der hierarchischen Modulstruktur aufgebaut, z.B. sind Funktionen, die auf Listen operieren, im Modul `Data.List` definiert.

[3] siehe `http://www.haskell.org/ghc/docs/latest/html/libraries`

3 Programmieren mit rekursiven Datentypen: Listen und Bäume

3.1 Listenverarbeitung in Haskell

In diesem Abschnitt erläutern wir die Verarbeitung von Listen und einige prominente Operationen auf Listen. Die hier angegebenen Typen sind spezieller als sie in den aktuellen Bibliotheken verwendet werden: Im Zuge einer Umstrukturierung wurden viele Funktionen, die nur für Listen verfügbar waren, derart verallgemeinert, dass man sie für alle Typen verwenden kann, die „faltbar“ sind (Listen sind inbesondere solche Typen). In Haskell's Typsystem wird dies über Typklassenconstraints (an die Typklasse `Foldable`) ausgedrückt (Typklassen behandeln wir genau in Abschnitt 5). Zur Verständlichkeit geben wir in diesem Abschnitt die spezielleren Typen an, die sich nur auf Listen beziehen.

3.1.1 Listen von Zahlen

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

- `[`*start*`..`*end*`]` erzeugt die Liste der Zahlen von *start* bis *end*, z.B. ergibt `[10..15]` die Liste `[10,11,12,13,14,15]`.
- `[`*start*`..]` erzeugt die unendliche Liste ab dem Wert *start*, z.B. erzeugt `[1..]` die Liste aller natürlichen Zahlen.
- `[`*start*`,`*next*`..`*end*`]` erzeugt die Liste $[start, start+\delta, start+2*\delta, \ldots, start+m\cdot\delta]$, wobei $\delta = next - start$ und solange Vielfache von δ dazuaddiert werden, bis der Endwert erreicht ist (d.h. er wurde überschritten, falls δ positiv, und unterschritten falls δ negativ ist).
- Analog dazu erzeugt `[`*start*`,`*next*`..]` die unendlich lange Liste mit der Schrittweite *next* – *start*.

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 = fromThenDelta start 1

fromThen :: Integer -> Integer -> [Integer]
fromThen start next = fromThenDelta start (next-start)
```

```
fromTo :: Integer -> Integer -> [Integer]
fromTo start end = fromThenDeltaTo start 1 end

fromThenTo :: Integer -> Integer -> Integer  -> [Integer]
fromThenTo start next end  = fromThenDeltaTo start (next-start) end

fromThenDelta :: Integer -> Integer  -> [Integer]
fromThenDelta start delta = start:(fromThenDelta (start+delta) delta)

fromThenDeltaTo :: Integer -> Integer -> Integer  -> [Integer]
fromThenDeltaTo start delta end = go start
 where
    go val
     | test val     = []
     | otherwise    = val:(go (val+delta))
    test val
     | delta >= 0 = val > end
     | otherwise  = val < end
```

3.1.2 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.

Didaktisch lässt sich mit dieser Repräsentation von Strings gut arbeiten, allerdings sind diese sehr langsam und für RealWorld-Anwendungen eher nicht geeignet. Hierfür bietet es sich an mit effizienteren Strukturen wie `ByteString`s zu arbeiten (diese findet man im Modul `Data.ByteString`).

3.1.3 Standard-Listenfunktionen

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

3.1.3.1 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).

3.1.3.2 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
_      !! i
  | i < 0   = error "negative index"
[]     !! _ = 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
```

3.1.3.3 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]
ghci>  map (2^) [1..10]
[2,4,8,16,32,64,128,256,512,1024]
*Main> :m + Data.Char
*Main Data.Char> map toUpper "Informatik"
"INFORMATIK"
```

3.1.3.4 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 Data.Char> filter isAlpha "2020 Informatik 2020"
"Informatik"
```

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

```
remove :: (a -> Bool) -> [a] -> [a]
remove p xs = filter (not . p) xs
```

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

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

3.1.3.5 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
```

3.1.3.6 Reverse

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

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

Da die Laufzeit von ++ linear in der linken Liste ist, folgt, dass der Aufwand von `reverseSlow` 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.
```

3.1.3.7 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"
```

3.1.3.8 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 n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs

drop :: Int -> [a] -> [a]
drop n xs     | n <= 0 =  xs
drop _ []              =  []
drop n (_:xs)          =  drop (n-1) xs

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

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

Einige Beispielaufrufe:

```
ghci> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
ghci>  drop 5 "Informatik"
"matik"
ghci>  takeWhile (> 5) [5,6,7,3,6,7,8]
[]
```

```
ghci> takeWhile (> 5) [7,6,7,3,6,7,8]
[7,6,7]
ghci> dropWhile (< 10) [1..20]
[10,11,12,13,14,15,16,17,18,19,20]
```

3.1.3.9 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 (x:xs) (y:ys) = (x,y) : (zip xs ys)
zip _      _      = []


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

Beispielaufrufe:

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

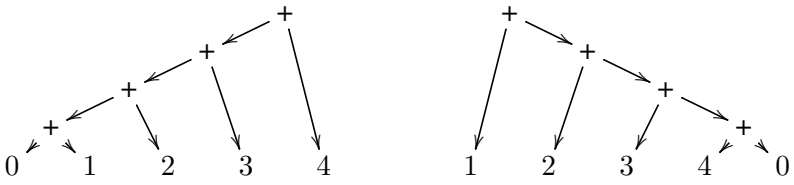
```
vectorAdd :: (Num a) => [a] -> [a] -> [a]
vectorAdd = zipWith (+)
```

3.1.3.10 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 der foldl-Funktionen für Listen sind:

```
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)
```

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 rechtsneutral 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 nicht erfolgreich ausgeweretet werden kann, terminiert die gesamte Auswertung des `foldl'`-Ausrufs nicht[1].

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]`.

[1] 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.

3.1.3.11 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`[2] und
- `head (scanr f e xs) = foldr f e xs`.

Die Definitionen sind:

```
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]
```

3.1.3.12 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:

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

```
partitionSlow 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 (an die Klasse `Ord`), denn es können nur Elemente sortiert werden, die auch verglichen werden können.

3.1.3.13 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 $\perp$ ist (Listen der Form $a_1 : \ldots : a_{n-1} : \perp$) 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 3.1.1. *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 Vielfachen von 3,5,7 so zu mischen, dass Elemente in aufsteigender Reihenfolge und nicht doppelt erscheinen.

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

3.1.3.14 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
```

3.1.3.15 Mengenoperationen

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 :: (a -> Bool) -> [a] -> Bool
  any _ []     = False
  any p (x:xs)
   | (p x)     = True
   | otherwise = any p xs

  all :: (a -> Bool) -> [a] -> Bool
  all _ []     = True
  all p (x:xs)
   | (p x)     = all p xs
   | otherwise = False
  ```

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

```
delete :: (Eq a) => a -> [a] -> [a]
delete _ [] = []
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 (\x -> any (== x) ys) xs
  ```

Einige Beispielaufrufe sind:

```
*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] [9,9,6,4,4,3,1]
[1,2,3,4,4,9,9,6]
*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]
```

3.1.4 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 (intensionale) Mengenschreibweise – so genannten ZF-Ausdrücke[3] – zu benutzen. Ein Beispiel ist

$$\left\{x^2 \,\middle|\, x \in \{1,2,\ldots,10\} \text{ und } x \text{ ist ungerade}\right\} = \left\{1,9,25,49,81\right\}$$

wird gelesen als "Menge aller x^2, so dass gilt: x ist aus der Menge $\{1,2\ldots,10\}$ und x ist ungerade."
Haskell bietet diese Notation ganz analog für Listen:

```
[ x^2 | x <- [1..10], odd x ]  ==  [1,9,25,49,81]
```

List Comprehensions haben die folgende Syntax:

`[Expr | qual`$_1$`,...,qual`$_n$`]`

wobei der *Rumpf* `Expr` ein Ausdruck ist (dessen freie Variablen durch `qual`$_1$`,.....,qual`$_n$ 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 x`$_1$`=e`$_1$`,...,x`$_n$`=e`$_n$ (ohne `in`-Ausdruck!) ist.

Es können beliebig viele Generatoren, Guards und Deklarationen benutzt werden, die lokale Deklarationen können "weiter rechts" verwendet werden und die Verschachtelung von List-Comprehensions ist ebenfalls erlaubt.

Wir betrachten einige einfache Beispiele:

- `[x | x <- [1..]]` erzeugt die Liste der natürlichen Zahlen
- `[ (wert,name) | wert <-[1..3], name <- ['a'..'b']]` erzeugt das kartesische Produkt `[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]`. Die Reihenfolge der Generatoren bestimmt die Reihenfolge der Werte in der Ergebnisliste, d.h. insbesondere erzeugt `[ (wert,name) | name <- ['a'..'b'], wert <-[1..3]]` die Liste `[(1,'a'),(2,'a'),(3,'a'),(1,'b'),(2,'b'),(3,'b')]`.
- `[(x,y) | x <- [1..], y <- [1..]]` erzeugt das kartesische Produkt der natürlichen Zahlen, ein Aufruf:

  ```
  ghci> 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.

[3] nach der Zermelo-Fränkel Mengenlehre

- `[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]]]` erzeugt als Ergebnis die Liste `[1,1,1,2,2,2,3,3,3]`.
- Eine alternative Definition für `map` mit List Comprehensions ist `map f xs = [f x | x <- xs]`.
- Eine alternative Definition für `filter` mit List Comprehensions ist `filter p xs = [x | x <- xs, p x]`.
- 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 3.1.2. `[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.

3.2 Bäume

Bäume können wie Listen durch rekursive Datentypen definiert werden. Wir betrachten verschiedene Varianten.

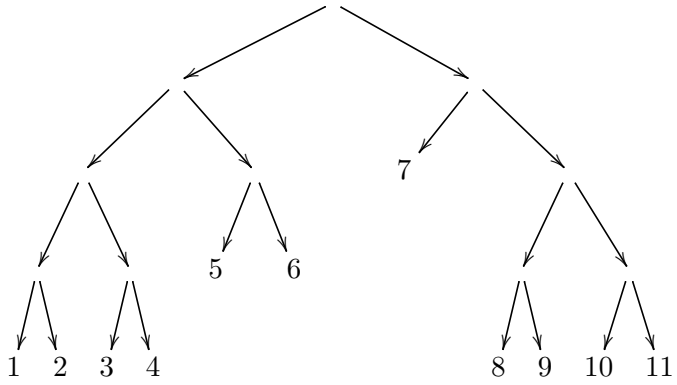
3.2.1 Binäre Bäume

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
```

3.2.2 Bäume mit höherem Grad

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.

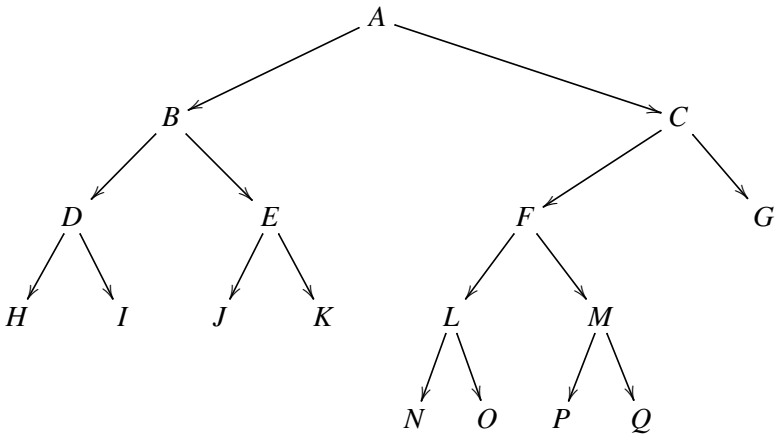
Übungsaufgabe 3.2.1. *Definiere Funktionen* `nRand`, `nFold`, `nMap` *für n-äre Bäume.*

3.2.3 Bäume durchlaufen

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), 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)

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

```
levelorderSchlecht :: BinBaum t -> [t]
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 :: BinBaum t -> [t]
levelorder b = loForest [b]
 where
  loForest xs                = map root xs ++ loForest (concatMap 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"
```

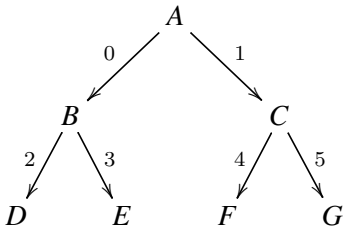
Übungsaufgabe 3.2.2. *Geben Sie einen Datentyp in Haskell für n-äre Bäume mit Markierung aller Knoten an. Implementieren Sie die Berechnung aller Knotenmarkierungen als Liste in pre-order, in-order, post-order und level-order Reihenfolge.*

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,BiKn¸oten 'b' (True,BiBlatt 'd') (False,BiBlatt 'e'))
 (False,BiKnoten 'c' (True,BiBlatt 'f') (False,BiBlatt 'g'))
```

3.2.4 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):

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

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:

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

4 Auswertung von Programmen in Haskell

In diesem Kapitel betrachten wir recht informell die Auswertung von Haskell-Ausdrücken und wie diese geschickt für die Programmierung eingesetzt werden kann.

4.1 Verzögerte Auswertung

Haskell verwendet die verzögerte Auswertung. Synonym werden die Begriffe Bedarfsauswertung, nicht-strikte Auswertung mit Sharing oder lazy evaluation und call-by-need evaluation im Englischen verwendet. Anstatt dies ganz formal zu definieren (z.B. kann man dies mathematisch korrekt mit einer Kernsprache erweitert um rekursive let-Ausdrücke erfassen, siehe z.B. (AF97; SSSS08)), betrachten wir nur die Kernpunkte.

Die beiden wesentlichen Ideen der verzögerten Auswertung sind, das einerseits Unterausdrücke nur dann ausgewertet werden, wenn sie für das Ergebnis benötigt werden und andererseits, dass mehrfache Auswertung desselben Ausdrucks vermieden wird. Für den ersten Punkt, kann man z.B. die Auswertung von `squareAndIgnore (6+9) (1+2+3+4+5)` betrachten, wobei `square` definiert sei als:

```
squareAndIgnore x y = x*x
```

Die nicht-strikte Auswertung berechnet den Wert durch die folgenden Schritte

```
squareAndIgnore (6+9) (1+2+3+4+5)
        → (6+9)*(6+9) → 15*(6+9) → 15*15 → 15*15 → 225.
```

Die meisten imperativen und auch andere funktionale Programmiersprachen verwenden eine strikte Auswertung und würden beide Argumente `(6+9)` und `(1+2+3+4+5)` auswerten *bevor* sie die Argumente in die Funktionsdefinition einsetzen: Hier zeigt sich schon ein Vorteil der nicht-strikten Auswertung: Da der Wert von `(1+2+3+4+5)` nicht benötigt wird, wird er auch nicht berechnet.

Genauso zeigt das Beispiel jedoch die mehrfache Auswertung von `(6+9)`, da `x` mehrfach in der Funktion `squareAndIgnore` verwendet wird. Die verzögerte Auswertung kombiniert deshalb die nicht-strikte Auswertung mit Sharing:

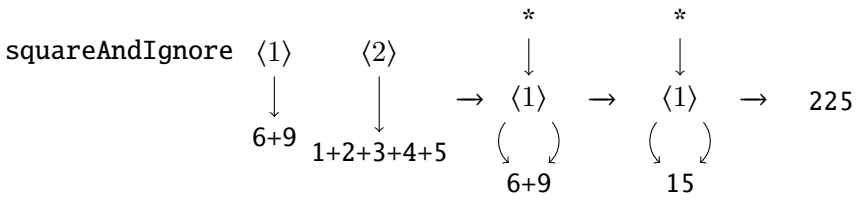
Die Idee im GHC ist, dass Ausdrücke in Haskell als Graphen repräsentiert werden. Ein unausgewerteter (Unter-)Ausdruck ist dann ein Verweis auf einen Teilgraphen und wird als *Thunk* bezeichnet. Bei der ersten Verwendung des Verweises wird der Redex ausgewertet und durch

seinen Ergebniswert ersetzt. Weitere Verwendungen des Verweises liefern sofort diesen Ergebniswert.

Die verzögerte Auswertung sieht die Unterausdrücke `6+9` und `1+2+3+4+5` als Thunks, und daher wird `6+9` nur einmal ausgewertet. Wir deuten das an, indem wir den Verweis als Speicheraddressen in der Form $\langle 1 \rangle$ und $\langle 2 \rangle$ darstellen und die Abbildung von Speicheraddresse auf Ausdruck nebenbei notieren. Beachte, die Unterausdrücke, die Werte sind (wie `6` und `9`) stellen wir ohne Verweis darauf dar:

(`squareAndIgnore` $\langle 1 \rangle$ $\langle 2 \rangle$, $\{\langle 1 \rangle \mapsto$ `6+9`, $\langle 2 \rangle \mapsto$ `1+2+3+4+5` $\})$
$\rightarrow$ ($\langle 1 \rangle$ * $\langle 1 \rangle$, $\{\langle 1 \rangle \mapsto$ `6+9`, $\langle 2 \rangle \mapsto$ `1+2+3+4+5` $\})$
$\rightarrow$ (15 * 15, $\{\langle 1 \rangle \mapsto$ `15`, $\langle 2 \rangle \mapsto$ `1+2+3+4+5` $\})$
$\rightarrow$ (225, $\{\langle 1 \rangle \mapsto$ `15`$\}$)

Eine alternative Darstellung ist die Verwendung von Graphen:



Wir betrachten als Beispiel ein Programm mit (eigentlich verbotenen, puren) Seiteneffekten, um uns die Unterschiede von nicht-strikter, strikter und verzögerter Auswertung zu veranschaulichen:

```
import Debug.Trace             -- von Verwendung wird abgeraten!!
-- trace :: String -> a -> a   -- Seiteneffekt: Textausgabe

foo :: Int -> Int -> Int -> Int
foo x y z = y + y + z

z = foo (trace "first"  1)
        (trace "second" 2)
        (trace "third"  3)
```

Die Auswertung von `z` in strikter Reihenfolge würde zu `"first" "second" "third" 7` führen, da erst die Argumente sequentiell von links nach rechts ausgewertet werden, und anschließend die erhaltenen *Werte* in den Rumpf von `foo` eingesetzt werden. In nicht-strikter Reihenfolge werden die unausgewerteten Argumente zunächst in den Rumpf eingesetzt (was `((trace "second" 2) + (trace "second" 1)) + (trace "third" 3)` ergibt. Danach werden die Argumente der +-Operationen ausgewertet, was zur Ausgabe `"second" "second" "third" 7` führt. Die verzögerte Auswertung setzt Thunks für die Argumente ein, und wertet dann die entsprechenden Redexe nur einmal aus. Das führt zur Ausgabe `"second" "third" 7`.

Als weiteres Beispiel für die verzögerte Auswertung betrachten wir:

```
foo x y z  = if x<0 then abs x else x+y
```

Wir erläutern die Auswertung einer Anwendung `foo` e_1 e_2 e_3: Die Ausdrücke e_1, e_2 und e_3 werden nicht angefasst, sondern in den Graphen (Heap) eingefügt und `x`, `y` und `z` sind Verweise auf die Ausdrücke e_1, e_2 und e_3. Im Anschluss beginnt die Auswertung des Rumpfs von `foo`. Die Auswertung des `if`-Ausdrucks erfordert ein Auswerten von `x<0` und dieses wiederum ein Auswerten des Arguments `x`. Daher wird e_1 ausgewertet und durch das Resultat ersetzt. Falls `x<0` wahr ist, wird der Wert von `abs x` zurückgegeben; weder e_2 noch e_3 werden ausgewertet. Falls `x<0` falsch ist, wird der Wert von `x+y` zurückgegeben; dies erfordert die Auswertung von e_2. Daher wird e_3 in keinem Fall ausgewertet. Der Ausdruck `foo 1 2 (1 ‘div‘ 0)` ist daher wohldefiniert.

Betrachte als weiteres Beispiel das Programm:

```
g x y =   let z = x ‘div‘ y
          in if x > y * y then x else z
t2 = g 5 0
```

Wertet man `t2` aus, so erhält man `5`. Da der Wert von `z` nicht benötigt wird, kommt es nicht zur Division durch Null. D.h. nicht nur Funktionsargumente werden verzögert ausgewertet, sondern beliebige Teilausdrücke. Sequentialität der Auswertung (erst das, dann das) ist daher nicht gegeben, was manchmal verwirrend sein kann.

4.1.1 Potentiell unendliche Datenstrukturen

Zu beachten ist, dass die verzögerte Auswertung Daten und Funktionen nicht ganz auswertet, sondern nur soweit, wie sie benötigt werden. Dadurch ermöglicht sie den Umgang mit „unendlichen“ Datenstrukturen:

```
ones  =  1 : ones       -- 'unendliche' Liste von 1en
twos  =  map (1+) ones  -- 'unendliche' Liste von 2en
nums  = iterate (1+) 0  -- Liste der natürlichen Zahlen

iterate :: (a -> a) -> a -> [a]
iterate f x  =   x : iterate f (f x)
```

In endlicher Zeit und mit begrenztem Speicher kann man sich natürlich nur endliche Teile davon anschauen. Es wird allerdings immer nur soviel von der Datenstruktur ausgewertet wie benötigt wird:

```
> take 10 nums
[0,1,2,3,4,5,6,7,8,9]
```

4.2 Programmieren mit Let-Ausdrücken in Haskell

Wir betrachten wie `let`-Ausdrücke in Haskell aufgebaut sind und demonstrieren einige Programmiertechniken im Bezug zu diesen.

4.2.1 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[1]. Ein einfaches Beispiel ist:

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

[1] 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.

4.2.2 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 Patterns in die einzelnen Komponenten zerlegen:

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

4.2.3 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 Aufruf 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)
```

4.3 Thunks im GHCi

Im GHCi ist es (teilweise) möglich, den Speicher zu inspizieren und daher zu beobachten, welche Teile einer Datenstruktur durch die Auswertung ausgewertet wurden und welche nicht. Die beiden Kommandos hierzu sind `:print` und `:sprint` (die zweite Variante liefert eine

vereinfachte Ansicht im Vergleich zur ersten). Der Aufruf ist `:sprint [<name> ...]`. Im Anschluss werden die Speicherzustände der definierten Namen in `<name> ...` angezeigt. Dabei werden unausgewertete Thunks durch _ gekennzeichnet.
Wir betrachten ein Beispiel:

```
Prelude> let x = map even [1..22]
Prelude> let y = (map odd [10..20],x)
Prelude> :sprint x y
x = _
y = (_,_)
Prelude> take 3 $ drop 3 x
[True,False,True]
Prelude> :sprint x y
x = _ : _ : _ : True : False : True : _
y = (_,_ : _ : _ : True : False : True : _)
```

Zunächst ist `x` unausgewertet, während für `y` schon klar ist, dass es ein Paar ist, dessen beide Argumente allerdings unausgewertet sind. Der Aufruf `take 3 $ drop 3 x` wertet einen Teil der Liste, auf die `x` zeigt, aus: Die ersten sechs Elemente werden angefordert, allerdings werden durch `drop 3`, die ersten drei Elemente selbst nicht ausgewertet. Die folgenden drei Elemente werden komplett ausgewertet, da sie im GHCi ausgedruckt werden.
Im Gegensatz zu `:sprint` zeigt `:print` für jeden Thunk eine Referenz und den jeweiligen Typ an:

```
Prelude> let x = map even [1..22]
Prelude> let y = (map odd [10..20],x)
Prelude> :print x y
x = (_t1::[Bool])
y = ((_t2::[Bool]),(_t3::[Bool]))
Prelude> take 3 $ drop 3 x
[True,False,True]
Prelude> :print x y
x = (_t4::Bool) : (_t5::Bool) : (_t6::Bool) : True : False : True :
    (_t7::[Bool])
y = ((_t8::[Bool]),(_t9::Bool) : (_t10::Bool) : (_t11::Bool) :
                   True : False : True : (_t12::[Bool]))
```

Man beachte, dass das Anzeigen nur für monomorphe Typen funktioniert, z.B. ergibt

```
Prelude> let z =  map (*2) [1..6]::[Int]
Prelude> take 3 $ drop 3 z
[8,10,12]
Prelude> :sprint z
z = _ : _ : _ : 8 : 10 : 12 : _
Prelude> let z =  map (*2) [1..6]
Prelude> take 3 $ drop 3 z
```

```
[8,10,12]
Prelude> :sprint z
z = _
Prelude> :print z
z = (_t30::(Num b, Enum b) => [b])
```

Die Überladung im zweiten Fall führt dazu, dass z immer noch ein Thunk ist (den Grund verstehen wir an dieser Stelle nicht: Es liegt an der Auflösung der Typklassen, die dazu führt, dass z eine Funktion ist, und daher nicht weiter ausgewertet wird.). Mit `:force` kann man die Auswertung von Thunks außerhalb der Reihenfolge erzwingen:

```
Prelude> let x = map even [1..22]
Prelude> :sprint x
x = _
Prelude> take 3 $ drop 3 x
[True,False,True]
Prelude> :print x
x = (_t1::Bool) : (_t2::Bool) : (_t3::Bool) : True : False : True :
    (_t4::[Bool])
Prelude> :force _t2
_t2 = True
Prelude> :print x
x = (_t5::Bool) : True : (_t6::Bool) : True : False : True :
    (_t7::[Bool])
```

Die verzögerte Auswetung ermöglicht die Trennung von Daten und Kontrollfluss, betrachte z.B.

```
factors :: Integral a => a -> [a]
factors n = filter (\m -> n `mod` m == 0) [2 .. (n - 1)]

isPrime :: Integral a => a -> Bool
isPrime n = n > 1 && null (factors n)
```

Die Funktion `factors` berechnet alle Faktoren einer Zahl. Die Berechnung von `isPrime` bricht sofort ab, sobald der erste Faktor gefunden wurde, trotz der Verwendung von `factors` werden also nicht alle Faktoren berechnet!

Als weiteres Beispiel, betrachte das Sieb des Erathostenes zur Berechnung der unendlichen Liste aller Primzahlen:

```
primes :: [Integer]
primes   =   sieve [2..]

sieve :: [Integer] -> [Integer]
sieve (p:xs) = p : sieve (xs `minus` [p,p+p..])
```

```
minus xs@(x:xt) ys@(y:yt) = case compare x y of
  LT -> x : minus xt ys
  EQ ->     minus xt yt
  GT ->     minus xs yt
```

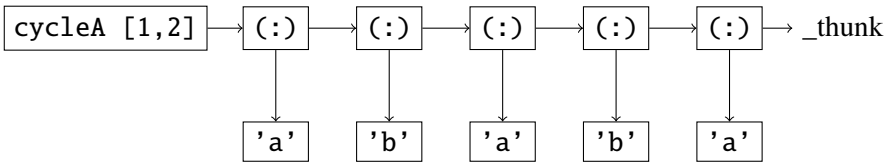
Wir müssen uns nur um die Daten kümmern, also *wie* wir die Primzahlen berechnen. Die Kontrolle über die Anzahl der benötigten Primzahlen erfolgt später. Zu Effizienz-Betrachtungen siehe http://www.haskell.org/haskellwiki/Prime_numbers.

Je nach Programmierung kann, die Verwendung von unendlichen Datenstrukturen, den Speicherverbrauch oder auch das Laufzeitverhalten von Programmen beeinflussen. Betrachte als Beispiel

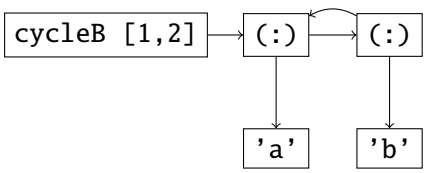
```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys

cycleA xs = xs ++ cycleA xs
cycleB xs = xs' where xs' = xs++xs'
```

`cycleA l` verbraucht potenziell unendlich viel Speicher, bzw. genauer so viel Speicherzellen, wie Elemente von `cycleA l` gelesen werden. Für eine Liste `l` mit Länge n verbraucht `cycleB` jedoch nur maximal n Speicherzellen.

Z.B. führt die Call-by-Need-Auswertung von `cycleA [1,2]` nach Anforderung einiger Elemente zu folgender Graphstruktur:



während die Auswertung von `cycleB [1,2]` eine zyklische Struktur erzeugt:



Wir haben bereits den `seq`-Operator und den `$!`-Operator gesehen, um strikte Auswertung von Unterausdrücken zu erzwingen. Dabei wird stets nur bis zur WHNF ausgewertet, was z.B. bedeutet, dass ein Thunk, der zu einer Liste auswertet, nur soweit ausgewertet wird, bis klar ist, ob die Liste leer ist oder nicht.

Will man tiefer auswerten, so kann man das Modul `Control.DeepSeq` verwenden, um die Kontrolle über die Auswertungsreihenfolge. zu übernehmen.

Eine weitere Alternative, um Striktheit zu erzwingen, bietet die Spracherweiterung `BangPatterns`. Damit ist `($!)` tatsächlich definiert:

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

Patterns, welche mit `!` beginnen, müssen immer zuerst ausgewertet werden:

```
stack exec -- ghci -XBangPatterns
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> let foo (x,!y) = [x,y]
Prelude> drop 1 $ foo (undefined,2)
[2]
Prelude> take 1 $ foo (2,undefined)
*** Exception: Prelude.undefined
```

4.3.1 Strikte Datentypen

Der GHC erlaubt auch die Deklaration von strikten Datentypen durch Annotation mit dem Ausrufezeichen:

```
data FaulesPaar   = FP  Integer  Bool deriving Show
data StriktesPaar = SP !Integer !Bool deriving Show
```

Den Unterschied sieht man z.B. folgendermaßen:

```
Prelude>  let (FP u v) = FP undefined True in v
True
Prelude> let (SP u v) = SP undefined True in v
*** Exception: Prelude.undefined
```

Der Konstruktor `SP` wird hier immer mit ausgewerteten Argumenten abgespeichert, die Argumente von `FP` können dagegen thunks sein.

Die Spracherweiterung `StrictData` macht alle Datentypen strikt; lazy Argumente erhält man dann mit einer Tilde:

```
stack exec -- ghci -XStrictData
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> data Paar = P Integer Bool
Prelude> let (P a b) = P undefined True in b
*** Exception: Prelude.undefined
Prelude> data FaulesPaar = FP ~Integer ~Bool
Prelude> let (FP a b) = FP undefined True in b
True
```

4.3.2 Laziness

Man kann in Haskell auch lazy Pattern-Matches erzwingen, indem man das Pattern mit ~ versieht. Betrachte z.B.

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> let bar (x,y) = 42
Prelude> bar undefined
*** Exception: Prelude.undefined
Prelude> let baz ~(x,y) = 42
Prelude> baz undefined
42
```

Beachte, dass solche Lazy-Patterns immer matchen und daher *irrefutable* sind. D.h. man sollte diese Patterns immer nur als letzte Möglichkeit verwenden. Z.B. ist die folgende Implementierung der Listenlängen-Funktion falsch:

```
mylength ~[] = 0
mylength (_:xs) = 1 + mylength xs
```

Da das Lazy-Pattern `~[]` immer matcht, liefert die Funktion für jede Liste 0. Beachte, dass Patterns auf linken Seiten von `let`-Ausdrücken ebenfalls stets lazy sind. Die Verwendung der Lazy-Patterns kann man sich so vorstellen: Aus

```
quu ~(x,y) = x+y
```

wird

```
quu p = let (x,y) = p in x+y
```

was wiederum übersetzt wird in

```
quu p = let x = case p of (x,_) -> x
            y = case p of (_,y) -> y
        in (x+y)
```

oder kürzer geschrieben:

```
quu p = (fst p) +(snd p)
```

Ein Beispiel aus dem Haskell-Wiki, für welches Lazy-Patterns nützlich sind, ist die Funktion `splitAt`, die eine Liste an einer Position aufspaltet und die beiden Teillisten liefert:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n ls
  | n <= 0       = ([], ls)
```

```
splitAt _ []     = ([], [])
splitAt n (y:ys) =
    case splitAt (n-1) ys of
      ~(prefix, suffix) -> (y : prefix, suffix)
```

Durch den Lazy-Pattern-Match steht der erste Teil des Ergebnis-Paares sofort zur Verfügung. Z.B. liefert

```
head $ fst $ splitAt 10000000 (repeat 'a')
```

sofort `'a'` zurück. Bei Verwendung eines normalen Pattern Matches, dauert dies viel länger (da erst 10 Millionen rekursive Aufrufe ausgewertet werden).

Wir fassen nochmal die wesentlichen Erkenntnisse zur verzögerten Auswertung in Haskell zusammen: Alle effizienten Implementierungen von Haskell verwenden diese Auswertung, deren wesentliche Idee ist, Unterausdrücke nur höchstens einmal auszuwerten.

Es gibt Annotationen, die es erlauben es Striktheit zu beeinflussen: Ein striktes Pattern ist `!p` und ein lazy Pattern ist `~p`. Verzögerte Auswertung erlaubt das natürliche Hantieren mit potentiell unendlichen Datenstrukturen Dadurch wird eine gute Modularisierung durch Trennung von Daten und Kontrollfluss erlaubt.

4.4 Quellennachweis und weiterführende Literatur

Die wesentliche Teile dieses Kapitels stammen aus Lehrmaterial von Steffen Jost (Jos19).

5 Haskells Typklassensystem

5.1 Ad hoc und parametrischer Polymorphismus

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 sich die Implementierungen für verschiedene Typen völlig anders verhalten können, also auch im Allgemeinen verschieden 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[1]. `==` 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*.

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

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

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 **T** ein polymorpher Typ (ohne Typklassen) ist und **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. Die Klasseninstanz ist 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:

[1] Das Core Libraries Committee, welches die wesentlichen Haskell-Bibliotheken standardisiert, hat kürzlich beschlossen, `/=` aus der Klasse `Eq` herauszunehmen und außerhalb der Klasse als `x /= y = not (x == y)` zu definieren. Damit wird `/=` zu einer normalen Funktion, die keine Klassenmethode mehr ist (siehe `https://github.com/haskell/core-libraries-committee/issues/3`

```
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
```

Datentypen, die mit `data` definiert werden, können zu Instanzen von Typklassen gemacht werden. Typsynonyme, die mit `type` definiert werden, dürfen nicht zu Instanzen von Typklassen gemacht werden. Man erhält dann eine Fehlermeldung. Der Grund liegt auf der Hand: Da Typsynonyme genau wie die orginalen Typen behandelt werden, man aber für jedes Synonym eine eigene Instanz definieren könnte, wüsste der Compiler nicht, welche Instanz er nun konkret wählen muss. Daher gibt es da `newtype`-Konstrukt: Mit `newtype` definierte Typsynonyme dürfen als Klasseninstanzen verwendet werden.

```
newtype Wahrheitswert = W Bool
instance Eq Wahrheitswert where
   (W True) == (W True) = True
   (W False) == (W False) = True
   _ == _ = False
```

Durch die Verwendung von mit `newtype`-definierten Synonymen kann man so auch mehrere Typklasseninstanzen für (im Grunde) denselben Typ angeben:

```
newtype VerkehrteWelt = VW Bool
instance Eq VerkehrteWelt where
```

```
(VW True) == (VW False) = True
(VW False) == (VW True) = True
_ == _ = False
```

5.2 Vererbung und Mehrfachvererbung

Bei Klassendefinitionen können eine oder mehrere Oberklassen angegeben werden. Die Syntax ist

```
class (Oberklasse_1 a, ..., Oberklasse_N a) => Klassenname a where
 ...
```

Die Schreibweise ist etwas ungewöhnlich, da man => nicht als logische Implikation auffassen darf. Vielmehr ist die Semantik folgendermaßen festgelegt: Ein Typ kann nur dann Instanz der Klasse *Klassenname* sein, wenn für ihn bereits Instanzen des Typs für die Klassen $Oberklasse_1$, ..., $Oberklasse_N$ 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.

5.3 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 (*Klasse*$_1$ a$_1$, ..., *Klasse*$_N$ a$_N$) => *Klasse Typ* where
 ...

Hierbei müssen die Typvariablen a$_1$, ..., a$_N$ alle im polymorphen Typ *Typ* vorkommen. 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> Blatt 1 == Blatt 2
False
*Main> Blatt 1 == Blatt 1
True
*Main> Blatt (\x ->x) == Blatt (\y -> y)

<interactive>:23:1: error:
    • No instance for (Eq (t0 -> t0)) arising from a use of ‘==’
        (maybe you haven’t applied a function to enough arguments?)
    • In the expression: Blatt (\ x -> x) == Blatt (\ y -> y)
      In an equation for ‘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 5.3.1. *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 für a
 Right x == Right y = x == y -- benutzt Eq-Instanz für b
 _       == _       = False
```

Eine analoge Problemstellung ergibt sich bei der `Eq`-Instanz für `BinBaumMitKM`.

Übungsaufgabe 5.3.2. *Definiere eine* `Eq`*-Instanz für* `BinBaumMitKM`*, wobei Bäume nur dann gleich sind, wenn ihre Knoten- und Kantenmarkierungen identisch sind.*

5.4 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[2]:

```
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>
```

[2] Die Funktion showChar ist bereits vordefiniert, sie ist vom Typ Char -> ShowS

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 5.4.1. *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> let x = read "10" in  seq x True

<interactive>:7:27: error:
    • Ambiguous type variable ‘t0’ arising from a use of ‘x’
      prevents the constraint ‘(Read t0)’ from being solved.
      Probable fix: use a type annotation to specify what ‘t0’ should be.
      ....

*Main> let x = (read "10")::Integer in  seq x True
True
```

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`[3].

[3] Mit dem Schlüsselwort `default` kann man das Defaulting-Verhalten auf Modulebene anpassen, siehe (Pey03, Abschnitt 4.3.4)

5.5 Die Klassen Num und Enum

Die Klasse Num stellt Operatoren für Zahlen zur Verfügung. Sie ist in aktuell definiert als:

```
class  Num a  where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
```

Beachte: In der früheren Definition (z.B. im Haskell 2010 Standard) war Num Unterklasse von Eq und Show.) 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]
```

5.6 Kinds

Wir führen in diesem Abschnitt sogenannte *Kinds* (engl. für Sorte) ein. Diese kann man sich Vorstellen als „Typen über Typen'. Dabei ist das für den Haskell-Standard definierte Kind-System sehr einfach. Die Syntax von Kinds κ ist

$$\kappa := * \mid \kappa \longrightarrow \kappa$$

D.h. es gibt nur den Basiskind * und Funktionskind. Ein normaler Haskell-Typ hat den Kind *. Das kann man auch im GHCi testen:

```
> :kind Bool
Bool :: *
> :kind (Bool -> [Int] -> Char)
(Bool -> [Int] -> Char) :: *
```

Typkonstruktoren TC mit Stelligkeit n haben den Kind

$$\underbrace{\texttt{* -> } \cdots \texttt{ -> *}}_{n \text{ Sterne}} \texttt{ -> *}$$

Dies kann man lesen als: Der Typkonstruktor TC erwartet n Argumente, die selbst Typen (und daher vom Kind *) sind. Wendet man TC auf solche Argumente an, so erhält man einen Typ. Z.B. hat der Typkonstruktor `Maybe` die Stelligkeit 1 und daher den Kind `* -> *`, Der Typkonstruktor `Either` erwartet zwei Typen als Argumente und hat daher den Kind `* -> * -> *`. Beides kann man im GHCi testen:

```
Prelude> :kind Maybe
Maybe :: * -> *
Prelude> :kind Either
Either :: * -> * -> *
Prelude> :kind Maybe Bool
Maybe Bool :: *
Prelude> :kind Either Bool
```

```
Either Bool :: * -> *
Prelude> :kind Either Bool Char
Either Bool Char :: *
```

Genau wie `Maybe` ist der Typkonstruktor für Listen einstellig, in Haskell-Syntax schreiben wir `[]` für diesen Konstruktor und meistens `[a]` für die Anwendung auf einen Typ `a` (man darf auch `[] a`) schreiben:

```
Prelude> :kind []
[] :: * -> *
Prelude> :kind [Bool]
[Bool] :: *
Prelude> :kind [] Bool
[] Bool :: *
```

Auch der Funktionspfeil `->` verhält sich wie ein Typkonstruktor, der zwei Typen als Argumente verlangt, er hat daher den Kind `* -> * -> *`:

```
Prelude> :kind (->)
(->) :: * -> * -> *
```

Tatsächlich gibt es noch weitere Kinds in Haskell, die auch im GHCi verfügbar sind. Die Typklassenbeschränkungen, wie `Show Bool` sind vom Kind `Constraint`, und `Show` selbst hat dann den Kind `* -> Constraint`, denn as erwartet als Argument einen Typ, um dann zu einem Constraint zu werden.

```
Prelude> :kind (Show Bool)
(Show Bool) :: Constraint
Prelude> :kind Show
Show :: * -> Constraint
```

5.7 Konstruktorklassen

Die bisher vorgestellten Typklassen abstrahieren über einen Typ, genauer: Die Variable in der Klassendefinition steht für einen Typ (sie ist vom Kind `*`). Man kann dies erweitern und hier auch Typkonstruktoren zulassen, über die abstrahiert wird. Konstruktorklassen ermöglichen genau dies (die abstrahierte Variable kann einen anderen Kind haben). Bei binären Bäumen ist `BBaum a` der Typ und `BBaum` der Typkonstruktor.

5.7.1 Die Klasse Functor

Die Klasse `Functor` ist eine Konstruktorklasse, d.h. sie abstrahiert über einen Typkonstruktor vom Kind `* -> *`.

```
Prelude> :kind Functor
Functor :: (* -> *) -> Constraint
```

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
```

In `Data.Functor` wird ein Synonym für `fmap` definiert:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

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:

```
... error:
    • Expecting one fewer argument to ‘BBaum a’
      Expected kind ‘* -> *’, but ‘BBaum a’ has kind ‘*’
    • In the first argument of ‘Functor’, namely ‘BBaum a’
      In the instance declaration for ‘Functor (BBaum a)’
```

Wie aus der Fehlermeldung hervorgeht, verwendet Haskell zum Prüfen der richtigen Instanzdefinition die Kinds. Man kann 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 5.7.1. *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.*

5.8 Auswahl weiterer vordefinierter Typklassen

Abbildung 5.1 zeigt die Klassenhierarchie der vordefinierten Haskell-Typklassen, sowie für welche Datentypen Instanzen der entsprechenden Klassen vordefiniert sind. Die gezeigt Hierarchie entspricht dem Haskell 2010 Report. Die Hierarchie der aktuell im GHC verwendeten Prelude[4] ist in Abbildung 5.2 gezeigt.

Die Änderungen lassen sich kurz wie folgt zusammenfassen: Neben den neuen Klassen `Semigroup`, `Monoid`, `Foldable`, `Traversable` wurden die Klasse `Applicative` als Unterklasse von `Functor` eingefügt, und `Monad` zu einer Unterklasse von `Applicative` gemacht. Die Klasse `MonadFail` wurde als Unterklasse von `Monad` eingefügt. Die Klassenbeziehung zwischen `Num` und `Eq` und `Show` wurden außerdem geändert.

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. Mit der Compileroption `-ddump-deriv` kann man sich die automatisch erzeugten Instanzen anzeigen lassen:

[4] Es wurde hier die Version 4.13 der base-Bibliothek als Vorlage verwendet. Diese wird mit dem GHC Version 8.8.3 geliefert, die Dokumentation der Prelude ist unter https://downloads.haskell.org/ghc/8.8.3/docs/html/libraries/base-4.13.0.0/Prelude.html zu finden.

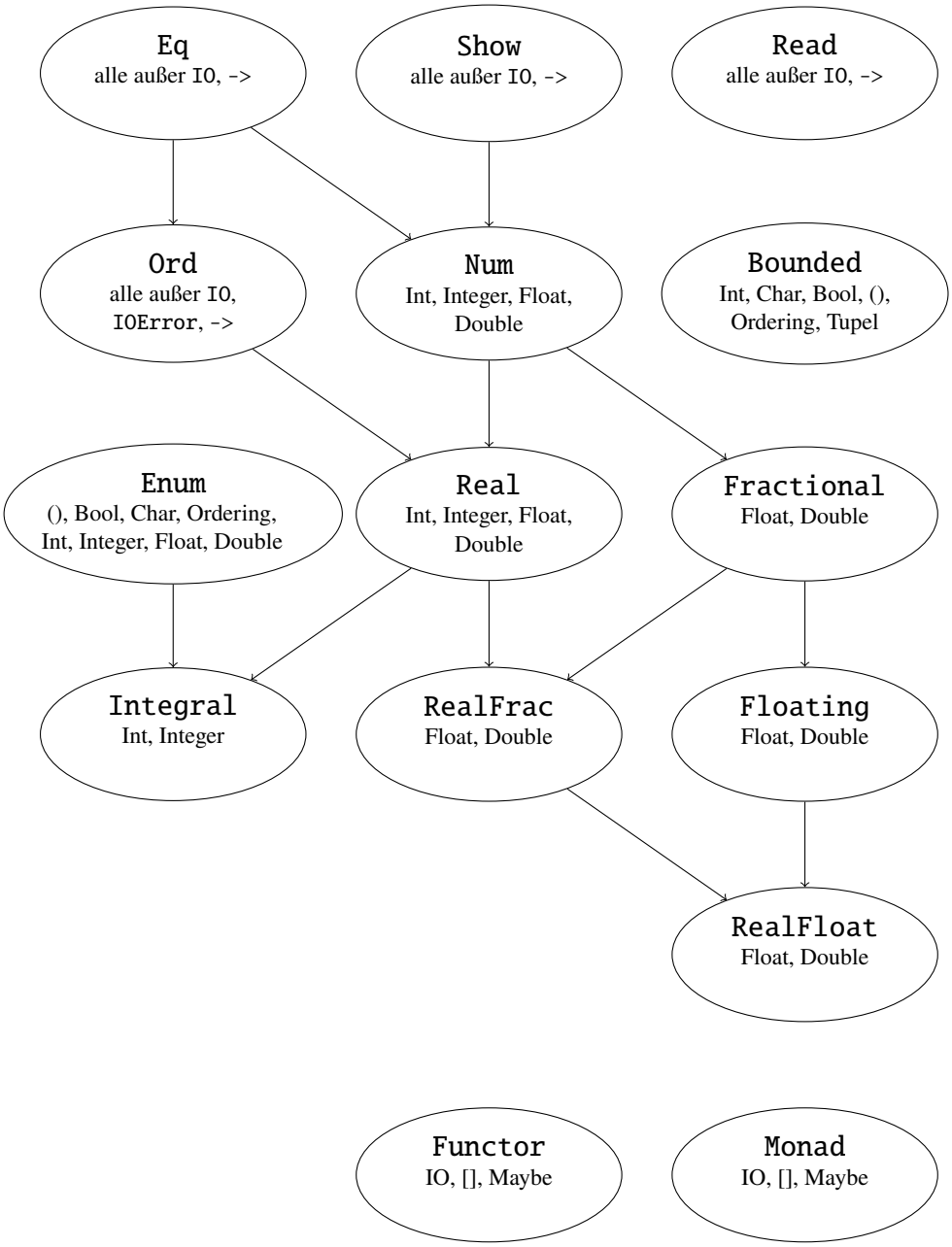


Abbildung 5.1: Typklassenhierarchie der vordefinierten Haskell-Typklassen und Instanzen der vordefinierten Typen entsprechend des Haskell Language Report 2010

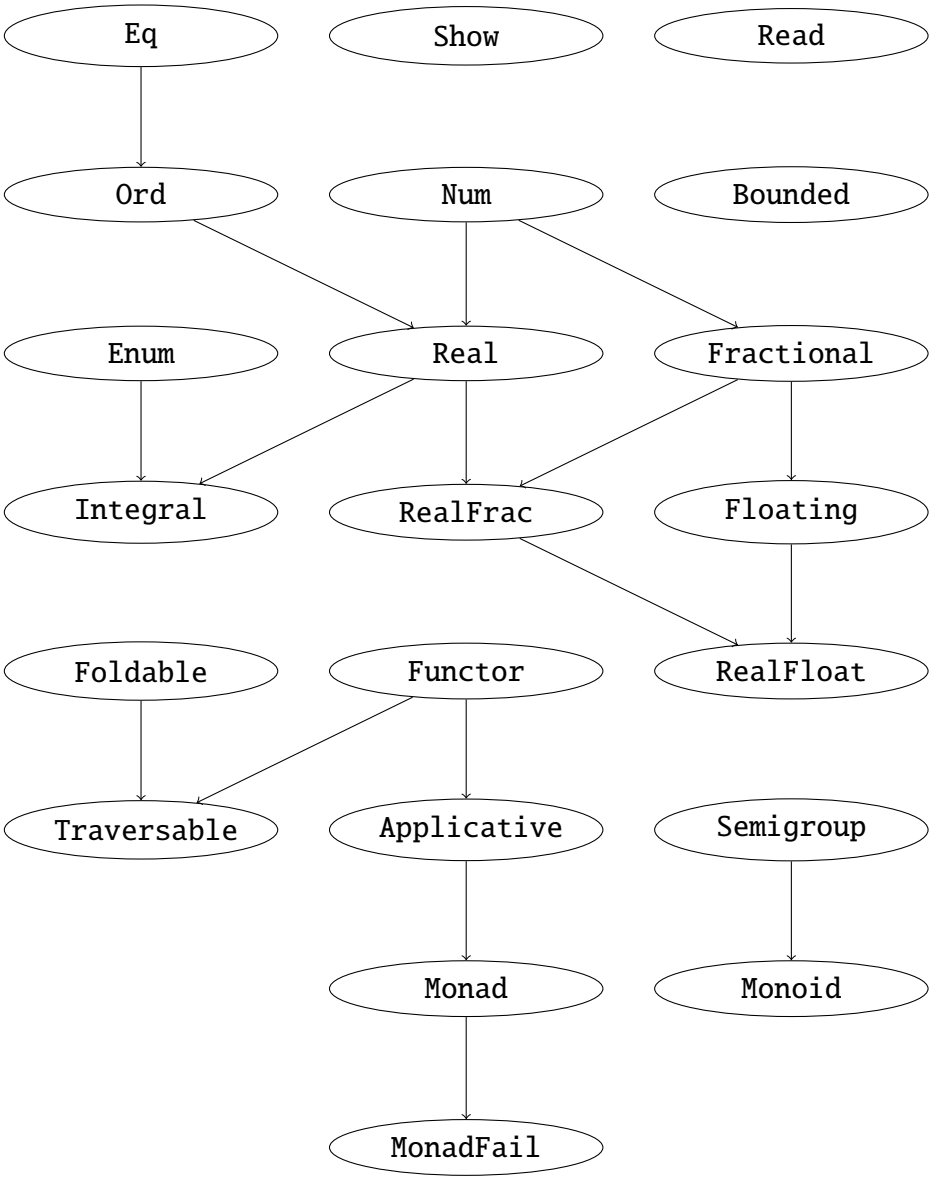


Abbildung 5.2: Typklassenhierarchie der vordefinierten Haskell-Typklassen entsprechend der aktuellen GHC-Prelude (Version: base-4.13.0.0)

```
Prelude> :set -ddump-deriv
Prelude> data WW = Wahr | Falsch deriving (Eq)

==================== Derived instances ====================
Derived class instances:
  instance GHC.Classes.Eq Ghci1.WW where
    (GHC.Classes.==) (Ghci1.Wahr) (Ghci1.Wahr) = GHC.Types.True
    (GHC.Classes.==) (Ghci1.Falsch) (Ghci1.Falsch) = GHC.Types.True
    (GHC.Classes.==) _ _ = GHC.Types.False
```

5.8.1 Monoide und Halbgruppen

Wir betrachten die relativ neuen Typklassen `Monoid` und `Semigroup`, die für Monoide und Halbgruppen eingeführt wurden.

Eine Halbgruppe ist eine Menge mit einer binären Verknüpfung, die assoziativ ist. Die Klasse `Semigroup` definiert hierfür eine Typklasse:

```
class Semigroup a where
  (<>) :: a -> a -> a             -- should be associative
  stimes  :: Integral n => n -> a -> a  -- fails for n<1
  sconcat :: NonEmpty a -> a
```

Es genügt, die Operation <> zu definieren, die assoziativ sein muss (damit es sich um eine Halbgruppe handelt), d.h. `x <> (y <> z) == (x <> y) <> z` für alle `x`, `y` und `z` gelten. Die Funktion `stimes` n a erzeugt

$$\underbrace{a \mathbin{<>} a \mathbin{<>} \ldots \mathbin{<>} a}_{n \text{ viele } a\text{s}}.$$

Die Funktion `sconcat` konkateniert eine nicht-leere Liste von Werten, indem sie diese mit dem <>-Operator verknüpft (`NonEmpty` ist ein Datentyp zur Repräsentation nicht-leerer Listen, definiert als `data NonEmpty a = a :| [a]`.

Eine Beispiel für eine Instanz von `Semigroup` ist die Minimum-Operation auf jedem Typ, der Instanz der Klasse `Ord` ist:

```
newtype Min a = Min a
getMin :: Min a -> a
getMin (Min x) = x

instance Ord a => Semigroup (Min a) where
  Min a <> Min b = Min (min a b)
```

Einige Beispielaufrufe:

```
*> getMin (Min 7 <> Min 8 <> Min 10 <> Min 3)
3
> getMin $ sconcat $ (Min 4):| (map Min [1,2,3,4,10,-1,2,3])
-1
```

Ein Monoid ist eine Menge mit einer binären Verknüpfung $\otimes$, die assoziativ ist, sowie einem neutralen Element 1, sodass $1 \otimes m = m = m \otimes 1$ für alle m in der Menge. Daher erweitert ein Monoid, eine Halbgruppe um das neutrale Element. Dieser Zusammenhang wird in Haskells Typklasse `Monoid` repräsentiert:

```
class Semigroup a => Monoid a where
  mempty  :: a            -- neutrales Element
  mappend :: a -> a -> a  -- Verknuepfung
  mappend = (<>)
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Beachte: `Monoid` ist erst ab GHC 8.4.x eine Unterklasse von `Semigroup`. Vorher waren beide Typklassen unabhängig voneinander und die Typklasse `Monoid` forderte zusätzlich, dass `mappend` assoziativ ist. Wenn man bei Monoid-Instanzen `import Data.Semigroup` und explizit `mappend = (<>)` hinschreibt, dann geht es mit beiden Versionen.

Wir betrachten als Beispiel Listen, denn diese bilden mit dem append-Operator ++ Monoide, wobei die leere Liste das neutrale Element ist:

```
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>)   = (++)
instance Monoid [a] where
  -- mempty :: [a]
  mempty  = []
```

Dass die geforderten Gesetze gelten, kann man nachrechnen, wobei man für endliche Listen Induktion verwenden kann, für unendliche Listen andere Methoden braucht (z.B. Co-Induktion). Beispiele:

```
> [1,2,3] <> mempty <> [4,5,6]
[1,2,3,4,5,6]

> Just [1,2,3]  <>  Just [4,5,6]
Just [1,2,3,4,5,6]

> Nothing  <>  Just [4,5,6]
Just [4,5,6]
```

```
> ([1,2,3],"abc") <> ([4,5,6],"def")
([1,2,3,4,5,6],"abcdef")
```

Die Beispiele verwenden schon `Monoid`-Instanzen für `Maybe` und Paare. Wir geben die Instanzen an:

```
instance Semigroup a => Semigroup (Maybe a) where
     Nothing <> my      = my
     mx      <> Nothing = mx
     Just x  <> Just y  = Just (x <> y)

instance Semigroup a => Monoid (Maybe a) where
     mempty = Nothing
```

Wiederum kann man nachrechnen, dass die Gesetze gelten. Interessanterweise genügt es, für die `Monoid`-Instanz zu fordern, dass der verpackte Typ eine Halbgruppe ist – neutrale Elemente werden auf diesem Typ nicht benötigt.
Für die Paar-Instanz ist dies anders, dort wird auf die neutralen Elemente der Komponenten zurückgegriffen.

```
instance (Semigroup a, Semigroup b) =>
   Semigroup (a, b) where
   (a,b) <> (a',b') = (a<>a', b<>b')

instance (Monoid a, Monoid b) => Monoid (a,b) where
   mempty = (mempty, mempty)
```

Auch Zahlen bilden Monoide:

- Additives Monoid $(+, 0)$: Es gilt $(x + y) + z = x + (y + z)$
- Multiplikatives Monoid $(\cdot, 1)$: Es gilt $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

D.h. man kann zwei Instanzen angeben. In der Haskell-Standardbibliothek wurde entschieden, dass man sich explizit entscheiden muss, welches Monoid man meint: Realisiert wird das mit Hilfe von `newtypes`:

```
newtype Sum a     = Sum     { getSum :: a }
newtype Product a = Product { getProduct :: a }

instance Num a => Semigroup (Sum a)     where
     Sum     x <> Sum     y = Sum     (x+y)

instance Num a => Semigroup (Product a) where
    Product x <> Product y = Product (x*y)
```

```
instance Num a => Monoid (Sum a)     where mempty = Sum 0

instance Num a => Monoid (Product a) where mempty = Product 1
```

```
> mconcat  $ map Sum [1..10]
Sum {getSum = 55}
> mconcat  $ map Product [1..10]
Product {getProduct = 3628800}
```

Auch Funktionen bilden mit der Operation der Funktionskomposition ein Monoid.

```
newtype Endo a = Endo { appEndo :: a -> a }
instance Semigroup (Endo a} where
  Endo f <> Endo g = Endo (f . g)
instance Monoid (Endo a) where
  mempty  = Endo id
```

Beispiel:

```
> let f= mconcat $ map Endo [(4+),(10*),succ,max 1,\n -> n*n+1]
> appEndo f 1
34
```

In der Mathematik ist ein Endomorphismus eine 1-stellige-Abbildung in sich selbst (also eine Funktion mit Haskell-Typ `a -> a`).

5.8.2 Die Klasse Foldable

Die Typklasse `Foldable` ist eine Konstruktorklasse und steht für jene Typen, die sich zusammenfalten lassen:

```
class Foldable t where          {-# MINIMAL foldMap | foldr #-}
    fold    :: Monoid m => t m -> m
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap' :: Monoid m => (a -> m) -> t a -> m
    foldr   :: (a -> b -> b) -> b -> t a -> b
    foldr' :: (a -> b -> b) -> b -> t a -> b
    foldl   :: (b -> a -> b) -> b -> t a -> b
    foldl' :: (b -> a -> b) -> b -> t a -> b
    toList  :: t a -> [a]
    null    :: t a -> Bool
    length  :: t a -> Int
```

```
elem    :: Eq a => a -> t a -> Bool
sum, product :: Num a => t a -> a
foldr1 :: (a -> a -> a) -> t a -> a
foldl1 :: (a -> a -> a) -> t a -> a
```

Es müssen dabei die Gesetze gelten

- Identität: `fold == foldMap id`
- Funktoren-Komposition: `foldMap f . fmap g == foldMap (f . g)`
- Gesetze für `foldMap` und `foldr`/`foldl` foldr f z t == foldMap

Die Instanz für Listen ist offensichtlich, daher geben wir sie nicht an.

5.9 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 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“ → „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 ==
overloadededeq :: EqDict a -> a -> a -> Bool
overloadededeq 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 `overloadededeq` 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, an denen (==) 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]
```

Die Übersetzung ist noch nicht komplett ausgeführt, 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 }
```

5.10 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 Turingmächtig 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.e

5.11 Quellennachweise und weitere Literatur

Haskells Typklassensystem wurde eingeführt in (WB89). Die Standardklassen sind im Haskell Report dokumentiert. In (WB89) ist auch die Auflösung der Überladung zu finden. Tiefergehend formalisiert wurden Haskells Typklassen in (HHJW96). Konstruktorklassen wurden in (Jon95) vorgeschlagen.

6 Applikative Funktoren, Monaden und Ein- und Ausgabe in Haskell

Monadisches Programmieren ist (aus Programmiersicht) eine Strukturierungsmethode, um Berechnungen zu komponieren. Sie werden insbesondere verwendet, um *sequentiell* ablaufende Programme in einer funktionalen Programmiersprache zu implementieren. In Haskell wird diese Technik verwendet, um sequentielle Ein- und Ausgabe zu programmieren.

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 sogenannten *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 seit der Version 7.10 des Glasgow Haskell Compilers eine Unterklasse von `Applicative`[1] Daher gehen wir zunächst auf `Applicative` ein, bevor wir die Klasse `Monad` betrachtet.

6.1 Applikative Funktoren

Die Klasse `Functor` haben wir bereits eingeführt. Sie ist eine Konstruktorklasse für solche Datentypen über deren „Inhalt“ man „mappen“ kann. Mathematisch ist ein Funktor eine strukturerhaltende Abbildung. Aus Programmiersicht passt das, denn die Klassenfunktion `fmap` verändert zwar den Inhalt, aber nicht die Verpackung (d.h. die Struktur). Zur Erinnerung wiederholen wir die Definition der Konstruktorklasse `Functor`, wobei wir die aktuelle Definition in den Haskell-Basisbibliotheken verwenden, die neben `fmap` noch die Klassenfunktion `(<$)` definiert. Wir verwenden zudem die neuere Schreibweise, die den Kind der Klassenvariablen `f` explizit angibt:

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  (<$) = fmap . const
  {-# MINIMAL fmap #-}
```

[1] Applikative Funktoren wurden erst spät identifiziert, siehe insbesondere (MP08).

In *a* `<$` *m* wird dabei der Inhalt in *m* durch *a* ersetzt. Z.B. wertet `42 <$ [1,2,3,4]` zu `[42,42,42,42]` aus.

Zur Erinnerung geben wir auch noch die `Functor`-Instanz für `Maybe` an:

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

Angenommen wir wollen für den Typ `Maybe` eine Operation `add` definieren, die zwei `Just`-Werte addiert, und `Nothing` liefert, sofern eines der beiden Argumente selbst `Nothing` ist. Z.B. soll `add (Just 5) (Just 10)` als Ergebnis `Just 15` liefern. Dann können wir dies direkt definieren durch Pattern-Matching, Auspacken, Abfragen aller Fälle und anschließendem Verpacken:

```
add (Just a) (Just b) = (Just a+b)
add _ _ = Nothing
```

Allerdings kann man Funktionen dieser Form eleganter (und generischer) definieren, wenn der Datentyp ein sogenannter *applikativer Funktor* ist. Die wesentliche Eigenschaft dabei ist, dass man eine mehrstellige Funktion auf mehrere im Datentyp verpackte Argumente anwenden kann, *ohne* dass man einerseits explizit auspacken muss und andererseits eine sequentielle Auswertung der Argumente erzwingt. Die `Functor`-Instanz selbst hilft dabei noch nicht viel. Ein Versuch ist:

```
add a1 a2 = (fmap (+) a1) <*> a2
```

Allerdings fehlt dafür der Operator `<*>`, der für `Maybe` den Typ `Maybe (a -> b) -> Maybe a -> Maybe b` hat. Erst stellt die „geliftete“ Anwendung einer Funktion auf ein Argument für den Typ `Maybe` dar.

Die Konstruktorklasse `Applicative` stellt uns diese Anwendung zur Verfügung:

```
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
  {-# MINIMAL pure, ((<*>) | liftA2) #-}

  (<*>) :: f (a -> b) -> f a -> f b
  (<*>) = liftA2 id
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

```
liftA2 f x = (<*>) (fmap f x)

(*>) :: f a -> f b -> f b
a1 *> a2 = (id <$ a1) <*> a2

(<*) :: f a -> f b -> f a
(<*) = liftA2 const
```

Die Operation `pure` verpackt ein beliebiges Objekt in den Datentyp, <*> ist die sequentielle Anwendung, und *> und <* verwerfen das linke bzw. rechte Ergebnis. Die Funktion `liftA2` lifted eine binäre Funktion.
Die Funktion `liftA2` macht schon alles was wir im obigen Beispiel erreichen wollten:

```
add = liftA2 (+)
```

Wenn wir eine ternäre Addition liften wollen, hilft uns dies noch nicht. Aber es gibt auch

```
liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

in der Bibliothek `Control.Applicative`, sodass wir schreiben können

```
add3 = liftA3 (\x y z -> x + y + z)
```

oder direkt mit der sequentiellen Applikation <*> und `fmap`:

```
fmap (\x y z -> x + y + z)  (Just 10) <*> (Just 20) <*> (Just 30)
```

Die Semantik ist, dass die Argumente sequentiell angewendet werden uns anschließend addiert wird (innerhalb des `Maybe`-Typs) und das Ergebnis ist in diesem Fall `(Just 60)`.
Mit dem Synonym <$> für `fmap` können wir äquivalent schreiben:

```
(\x y z -> x + y + z)  <$> (Just 10) <*> (Just 20) <*> (Just 30)
```

Man kann auch komplett auf `fmap` oder <$> verzichten, indem man die pure Funktion explizit mit `pure` in die Struktur hebt, d.h. ebenso äquivalent ist:

```
pure (\x y z -> x + y + z)  <*> (Just 10) <*> (Just 20) <*> (Just 30)
```

Allgemein kann man mit einem Applikativen Funktor daher Aufrufe der Form

$$f \texttt{ <\$> } arg_1 \texttt{ <*> } arg_2 \texttt{ ... <*> } arg_n$$

machen, wobei f eine n-stellige Funktion ist, die Argumente $arg_1, \ldots, arg_n$sequentiell angewendet werden und anschließend die Funktion auf ihren Inhalt angewendet wird.
Durch Verwendung von `pure` kann allgemein der Aufruf auch als

$$\texttt{pure } f \texttt{ <*> } arg_1 \texttt{ <*> } arg_2 \texttt{ ... <*> } arg_n$$

ausgedrückt werden.

6.1.1 Gesetze und die Maybe-Instanz für Applicative

Wir geben nun die `Maybe`-Instanz für `Applicative` an:

```
instance Applicative Maybe where
  pure a = Just a
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something
```

Für jede Instanz von `Applicative` sollten die folgenden Gesetze (Gleichungen) gelten, die man nachprüfen sollte, bevor man eine entsprechende Instanz definiert:

- Identität: `pure id <*> v = v`
- Komposition: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- Homomorphismus: `pure f <*> pure x = pure (f x)`
- Austausch: `u <*> pure y = pure (\x -> x y) <*> u`

Beispiel 6.1.1. *Wir rechnen die Gesetze für die* `Maybe`*-Instanz nach:*

1. *Identität: Es gilt*

   `pure id <*> v = Just id <*> v = fmap id v = v`

   wobei die letzte Umformung aus dem 1. Gesetz für Funktoren folgt.
2. *Komposition: Wir vereinfachen zunächst:*

   `pure (.) <*> u <*> v <*> w = Just (.) <*> u <*> v <*> w = fmap (.) u <*> v <*> w.`

   Nun unterscheide die Fälle:

   a) `u = Nothing`. *Dann gilt*

   ```
   fmap (.) Nothing <*> v <*> w
   = Nothing <*> v <*> w
   = Nothing <*> w
   = Nothing
   = Nothing <*> (v <*> w)
   ```

   b) `u = Just u'`. *Dann gilt zunächst*

   ```
   fmap (.) (Just u') <*> v <*> w
   = (Just ((.) u')) <*> v <*> w
   = fmap ((.) u') v <*> w
   ```

   Nun unterscheide erneut:

   i. `v = Nothing`. *Dann gilt*

   ```
   fmap ((.) u') Nothing <*> w
   = Nothing <*> w
   = Nothing
   = fmap u' Nothing
   = fmap u' (Nothing <*> w)
   = (Just u') <*> (Nothing <*> w)
   ```

ii. `v = Just v'`. *Dann gilt*

```
fmap ((.) u') (Just v') <*> w
= Just (u' . v') <*> w
= fmap (u' . v') w
=† fmap u' (fmap v' w)
= fmap u' (Just v' <*> w)
= Just u' <*> (Just v' <*> w)
```

wobei die mit † markierte Umformung aus dem 2.Gesetz für Funktoren folgt.

Beachte, dass wir auch die Fälle $u = \bot$ und $v = \bot$ betrachten müssten, um alle Ausdrücke abzudecken (nämlich auch die nichtterminierenden). Wir verzichten hier darauf.

3. *Homomorphismus: Es gilt*

```
pure f <*> pure x = Just f <*> pure x = fmap f (pure x)= fmap f (Just x)
                                                       = Just (f x) = pure (f x).
```

4. *Austausch: Wir betrachten zwei Fälle:*

   a) `u = Nothing`. *Dann gilt*

```
Nothing <*> pure y = Nothing = fmap (\x -> x y) Nothing
                             = Just (\x -> x y) <*> Nothing
                             = pure (\x -> x y) <*> Nothing
```

   b) `u = Just u'`. *Dann gilt*

```
(Just u') <*> pure y
= fmap u' (pure y)
= fmap u' (Just y)
= Just (u' y)
= Just ((\x -> x y) u')
= fmap (\x -> x y) (Just u')
= Just (\x -> x y) <*> (Just u')
= pure (\x -> x y) <*> (Just u')
```

Im einführenden Beispiel zur Addition hatte die übergebene Funktion für beide Argumente den gleichen Typ, aber dies muss nicht so sein. Es kann (wie die allgemeinen Typen schon zeigen) eine beliebige Funktion übergeben werden, daher kann man z.B. wie folgt verknüpfen

```
*> (\x y z -> (x > 1) && (isLower y) || (length z > 5))
                                        <$> Just 5 <*> Just 'c' <*> Just [1,2,3]
Just True
```

Wir betrachten ein Beispiel, welches angelehnt an ein Beispiel aus (Sno17) ist. Implementiere eine Funktion

```
response :: String -> String -> Maybe String
response yearOfBirth yearToday = ...
```

die ein Geburtsjahr und das heutige Jahr als Texteingabe bekommen und daraus das heutige Alter berechnen. Wenn die beiden Eingaben als Zahlen erkannt werden können, dann berechne die Ausgabe, wenn einer der Strings keine Zahl ist, dann liefere `Nothing`.

Wir verwenden `readMaybe :: Read a => String -> Maybe a` aus der Bibliothek `Text.Read`, um `response` zu implementieren:

```
response :: String -> String -> Maybe String
response yearOfBirth yearToday =
 case readMaybe yearOfBirth of
   Just byear -> case readMaybe yearToday of
                   Just tyear ->  (Just $ show $ calculateAge  byear tyear)
                   Nothing -> Nothing
   Nothing -> Nothing

calculateAge :: Int -> Int -> Int
calculateAge byear tyear =  tyear - byear
```

Diese Implementierung funktioniert zwar, sie ist jedoch unübersichtlich und komplex: Ständig wird das durch `Maybe` verpackte Ergebnis ausgepackt und wieder eingepackt.

Durch Verwendung der sequentiellen Applikation <*> und `fmap` kann die gleiche Funktionalität funktionaler geschrieben werden als:

```
response yearOfBirth yearToday =
     show <$> ((calculateAge) <$> (readMaybe yearOfBirth) <*> (readMaybe yearToday))

calculateAge :: Int -> Int -> Int
calculateAge byear tyear =  tyear - byear
```

Bemerkung 6.1.2. *Wenn wir allerdings in Abhängigkeit der Werte andere Funktionen aufrufen möchten, d.h. der Kontrollfluss vom aktuellen Kontext abhängt, dann funktioniert das nicht mehr mit Applikativen Funktoren (aber mit Monaden). Ein Beispiel ist: Nehme an, dass die* `response`*-Parameter vertauscht wurden, wenn* `yearOfBirth` *größer als* `yearToday` *ist, und schreibe ein Programm, das diesen Fehler korrigiert, indem es* `calculateAge` *mit umgekehrten Argumenten aufruft. Dies kann so nicht mit Applikativen Funktoren implementiert werden. (Ein Trick wäre hier, die Funktion* `calculateAge` *direkt abzuändern, aber das war in der Problembeschreibung nicht erlaubt).*

Wir betrachten im Folgenden weitere Instanzen der Klasse `Applicative`.

6.1.2 Listen als Instanz der Klasse Applicative

6.1.2.1 Die Standardinstanz für Listen – Nichtdeterministische Berechnung

Die in der Standardbibliothek enthaltene Instanz (definiert im Modul `Control.Applicative`) von `Applicative` für Listen ist die folgende:

```
instance Applicative [] where
    pure x         = [x]
    fs <*> xs      = [f x | f <- fs, x <- xs]
    liftA2 f xs ys = [f x y | x <- xs, y <- ys]
    xs *> ys       = [y | _ <- xs, y <- ys]
```

Der `pure`-Operator verpackt ein Element in eine Liste. Die sequentielle Anwendung (hier mit dem Typ `(<*>) :: [a -> b] -> [a] -> [b]`[2]) kombiniert alle Möglichkeiten der Anwendung von Funktionen aus `f` aus `fs` auf Elemente `x` aus `xs`. Z.B.

```
*Main> (+) <$> [1,2,3] <*> [4,5,6]
[5,6,7,6,7,8,7,8,9]
*Main> (++) <$> ["A","B"] <*> ["c","d","e"]
["Ac","Ad","Ae","Bc","Bd","Be"]
*Main> [(*4),(+6),(^2)] <*> [1,2]
[4,8,7,8,1,4]
*Main> [(*),(+)] <*> [1,2] <*> [3,4]
[3,4,6,8,4,5,5,6]
```

Die zugrundeliegende Vorstellung dieser Implementierung ist, dass `[a]` die Liste der möglichen (nicht-deterministisch berechneten) Werte für `a` darstellt.

6.1.2.2 Zip-Listen Instanz

In der vorherigen Instanz wurden für zwei Listen `fs` $= [f_1, \ldots, f_n]$ und `xs` $= [x_1, \ldots, x_m]$ alle Kombinationen $(f_i\, x_j)$ erzeugt. Man kann Listen aber auch durch *zippen* zu einer gültigen Instanz von `Applicative` machen. Dabei werden nur f_i und x_i jeweils zu einem Paar verschmolzen, d.h. die Semantik dieser Instanz ist dann verschieden zu der vorherigen. Wir verwenden dazu erneut einen `newtype` (hier in Verbindung mit der Record-Syntax);

```
newtype ZipList a = ZipList {getZipList :: [a]} deriving Show

instance Applicative ZipList where
  pure x                    = ZipList $ repeat x
  ZipList fs <*> ZipList xs = ZipList $ zipWith (\f x -> f x) fs xs
```

[2] Mit der Spracherweiterung `TypeApplications` kann man sich diesen Typ im GHCi anzeigen lassen, indem man `:type (<*>) @ []` eingibt: Man wendet den Typkonstruktor `[]` dabei auf den allgemeinen Typ von <*> an, und erhält den spezialisierten Typen. Analog erhält man durch Eingabe von `:type (<*>) @ Maybe` im GHCi den Typ `Maybe (a -> b) -> Maybe a -> Maybe b`.

Einige Beispielaufrufe sind:

```
*> (*) <$>  [1,2,3] <*>  [1,10,100,1000]
[1,10,100,1000,2,20,200,2000,3,30,300,3000]
*> (*) <$> ZipList [1,2,3] <*> ZipList [1,10,100,1000]
ZipList [1,20,300]
*> (,,) <$> ZipList[1,2] <*> ZipList[3,4] <*> ZipList[5,6]
ZipList [(1,3,5),(2,4,6)]
```

Beachte, dass `pure` für diese Instanz eine unendliche Liste des gleichen Elements erzeugt. Der Grund ist, dass eine solche erzeugte Liste, dann einen Wert an jeder Position produzieren kann. Durch das Verwenden von `zipWith` bestimmt zudem die kürzere Liste die Länge der Ergebnisliste.

Betrachte z.B. das Gesetz `pure id <*> v = v`, welches für alle Instanzen von `Applicative` gelten muss.

Würde man `pure x = ZipList [x]` definieren, so wäre das Gesetz z.B. für `v = [1,2]` verletzt, während es mit der unendlichen Liste gilt.

Mit der `ZipList`-Instanz kann man leicht ein `zipN` simulieren, welches aus n Listen eine Liste von n-Tupeln erstellt (beachte: `zipN` ist in Haskell nicht definierbar, da ihm kein Typ zugeordnet werden kann). Beispiele sind:

```
*Main> getZipList $ (,) <$> ZipList [1..10] <*> ZipList [11..20]
[(1,11),(2,12),(3,13),(4,14),(5,15),(6,16),(7,17),(8,18),(9,19),(10,20)]
*Main> getZipList $  (,,,) <$> ZipList "Hund"
                           <*> ZipList "Maus"
                            <*> ZipList [1,2,3,4]
                             <*> ZipList [False,True,True,False]
[('H','M',1,False),('u','a',2,True),('n','u',3,True),('d','s',4,False)]
```

6.1.3 Applicative-Instanz für Funktionen

Auch Funktionen können zu Instanzen von `Functor` und `Applicative` gemacht werden: Die `Functor`-Instanz für Funktionen ist die Funktionskomposition:

```
instance Functor ((->) e) where
  fmap :: (a -> b) -> (e -> a) -> (e -> b)
  fmap = (.)
```

Die `Applicative`-Instanz ist:

```
  instance Applicative ((->) e) where
    pure x  = \_ -> x
    f <*> g = \e -> f e (g e)
```

Der Typ `((->) e)` hat den erwarteten Kind `* -> *`, da ein Funktionstyp zwei Argumente hat und hier nur ein Argument angegeben wird. Die Instanziierung der Typen ergibt:

```
pure  :: a -> (e -> a)
<*>   :: (e -> (a -> b)) -> (e -> a) -> (e -> b)
```

Diese beiden Funktionen sind übrigens im Lambda-Kalkül auch als die Kombinatoren $\mathbb{K}$ und $\mathbb{S}$ bekannt. Die Instanz für `(->) e` erlaubt uns z.B. eine Vereinfachung des folgenden Codes für eine Berechnung mit einer Umgebung. Betrachte zunächst den Code mit einer Umgebung, der einfache arithmetische Ausdrücke auswertet:

```
import Control.Applicative

data Exp v = Var v | Val Int | Neg (Exp v)
           | Add (Exp v ) (Exp v )
type Env v = [(v,Int)]

fetch :: (Eq v) => v -> Env v -> Int  -- Variable nachschlagen
fetch x env
 | Just val <- lookup x env = val
 | otherwise                = error "Variable not found"

eval  :: Exp String -> Env String -> Int
eval  (Var x)   env = fetch x env
eval  (Val i)   env = i
eval  (Neg p)   env = negate  $  eval  p env
eval  (Add p q) env = (+) (eval  p env)  (eval  q env)
```

Man kann argumentieren, dass es störend ist, dass die Umgebung `env` überall durchgeschleift und erwähnt werden muss. Dank der `(->) e`-Instanz für `Applicative` geht es jedoch auch ohne dies zu tun (die Umgebung `env` wird dabei größtenteils versteckt):

```
eval  :: Exp String -> Env String -> Int
eval  (Var x)       = fetch x
eval  (Val i)       = pure i
eval  (Neg p)       = negate <$> (eval  p)
eval  (Add p q)     = (+) <$> (eval  p) <*>  (eval  q)

fetch :: (Eq v) => v -> Env v -> Int  -- Variable nachschlagen
fetch x = maybe (error "Variable not found") id <$> (lookup x)
```

6.1.4 Die Traversable-Klasse

Liften des Cons-Operators `(:)` für Listen ergibt die folgende nützliche Definition aus dem Modul `Data.Traversable`:

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA []     = pure []
sequenceA (x:xs) = liftA2 (:) x (sequenceA xs)
                -- = (:) <$> x <*> sequenceA xs
```

Dadurch wird eine Liste von verpackten Werten zu einer verpackten Liste von Werten:

```
*> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
*> (:) <$> Just 3 <*> ((:) <$> Just 2 <*> ((:) <$> Just 1 <*> pure []))
Just [3,2,1]
*> sequenceA [Just 3, Nothing, Just 1]
Nothing
*> sequenceA [(+3),(+2),(+1)] 3
[6,5,4]
*> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
```

Im Modul `Data.Traversable` findet sich folgende Definition der Typklasse `Traversable`

```
class (Functor t, Foldable t) => Traversable t where
  {-# MINIMAL traverse | sequenceA #-}
  traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequence  :: Monad m => t (m a) -> m (t a)
  mapM      :: Monad m => (a -> m b) -> t a -> m (t b)
```

Diese Typklasse steht für Typen, die linear durchlaufen werden können. Instanzen müssen die folgenden Gesetze erfüllen (siehe Dokumentation des Moduls `Data.Traversable`):

- Natürliche Transformation: `t . sequenceA` = `sequenceA . fmap t`
- Identität: `sequenceA . fmap Identity` = `Identity`
- Komposition: `sequenceA . fmap Compose` = `Compose . fmap sequenceA . sequenceA`

6.1.5 Paare als Applikative Funktoren

Wir haben bereits eine `Functor`-Instanz für `Either a` gesehen. Analog kann man eine `Functor`-Instanz für Paare angeben, welche die Funktion nur auf die zweite Komponente anwendet:

```
instance Functor ((,) a) where
   fmap f (x,y) = (x, f y)
```

Will man nun aus Paaren einen applikativen Funktor machen, so kann man fast analog verfahren (und die erste Komponente nahezu ignorieren), d.h. `(x,(+3)) <*> (y,5)` soll `(z,8)` ergeben. Wie soll nun jedoch aus `x` und `y` welches `z` berechnet werden? Hier wird eine `Monoid`-Instanz der ersten Komponenten gefordert, um `x` und `y` mit der binären Verknüpfung `mappend` zu verknüpfen:

```
instance Monoid a => Applicative ((,) a) where
   pure x = (mempty, x)
   (u, f) <*> (v, x) =
   (u `mappend` v, f x)
```

Ein Beispiel für die Verwendung ist:

```
*> (\x y z -> x + y + z) <$> (Product 3, (9)) <*> (Product 2, 5) <*> (Product 3, 10)
(Product {getProduct = 18},24)
```

6.2 Monaden

Wir betrachten im Folgenden die Klasse `Applicative` nicht weiter, sondern wenden uns der Klasse `Monad` zu[3], wir werden aber den Unterschied zwischen `Applicative` und `Monad` zunächst erläutern (dieser dient auch der Motivation zur Einführung der Klasse `Monad`).

Beachte, dass es keine allgemeine Möglichkeit gibt, die Struktur eines Applikativen Funktors wieder zu verlassen, d.h. es gibt keine allgemeine Funktion

```
unpure :: Applicative f => f a -> a
```

Der Grund dafür ist offensichtlich, wenn man es für einzelne Instanzen versucht:

```
unpure :: Maybe a -> a
unpure (Just x) = x
unpure Nothing  = undefined  -- Wie ein a erzeugen ???

unpure :: [a] -> a
unpure (x:_)    = x
unpure []       = undefined  -- Wie ein a erzeugen ???

unpure :: (r -> a) -> a
unpure f        = undefined  -- Wie ein a erzeugen ???
```

[3] Wir müssen jedoch stets auch Instanzen für `Functor` und `Applicative` definieren, um Instanzen der Klasse `Monad` zu bilden.

Daher verbleibt man im Allgemeinen in der Struktur / Verpackung.

Ähnlich verhält es sich mit Monaden, diese bieten aber immerhin die Möglichkeit, während der Berechnung in die Struktur zu schauen und anhand des damit verpackten Wertes die Berechnung zu steuern. Wir betrachten zunächst die Definition der Klasse `Monad`:

```
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  {-# MINIMAL (>>=) #-}
  m >> k      = m >>= \_ -> k
  return      = pure
```

Als Unterklasse gibt es in der aktuellen Klassenhierarchie die Klasse `MonadFail` (in früheren Definitionen war die zugehörige `fail`-Operation in die Klasse `Monad` integriert:

```
class Monad m => MonadFail (m :: * -> *) where
  fail :: String -> m a
  {-# MINIMAL fail #-}
```

Für Instanzen von `Monad` muss der Operator >>= definiert werden, da für `return` und >> Default-Implementierungen gegeben sind.

Die Operation `fail` bietet einen Fehlerausgang mit Fehlerstring.

Ein Objekt vom Typ `m a` (wobei `m` und `a` entsprechend instantiiert sind und `m` eine Instanz der Klasse `Monad` ist) bezeichnet man als *monadische Aktion*.

Der Operator >>= wird „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` verpackt 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 durchgereicht. Die `Monad`-Instanz für `Maybe` ist definiert als:

```
instance Monad Maybe where
    return         = Just
    Nothing  >>= f = Nothing
```

```
    (Just x) >>= f = f x
instance MonadFail Maybe where
    fail _         = Nothing
```

Wir haben bereits die `Applicative`-Instanz für `Maybe` verwendet, um die Funktion `response` zu definieren, die für zwei Eingabestrings ein Alter berechnet, sofern sich beide Strings in Jahreszahlen konvertieren lassen.

Im Grunde lassen sich hier die gleichen Bemerkungen machen: Das Programm

```
response :: String -> String -> Maybe String
response yearOfBirth yearToday =
 case readMaybe yearOfBirth of
   Just byear -> case readMaybe yearToday of
                   Just tyear -> Just $ show $ calculateAge  byear tyear
                   Nothing -> Nothing
   Nothing -> Nothing

calculateAge :: Int -> Int -> Int
calculateAge byear tyear =  tyear - byear
```

kann durch Verwendung der Monaden-Instanz für `Maybe` kürzer und eleganter formuliert werden als:

```
responseM :: String -> String -> Maybe String
responseM yearOfBirth yearToday =
 (readMaybe yearOfBirth) >>=
     (\byear -> readMaybe yearToday >>=
       \tyear -> return $ show $ calculateAge byear tyear)

calculateAge :: Int -> Int -> Int
calculateAge byear tyear =  tyear - byear
```

Die selbe Funktionalität lieferte bereits die `Applicative`-Instanz, die in diesem Fall noch kürzer und „funktionaler“ ist. Wollen wir das Programm jedoch nun wie in Bemerkung 6.1.2 erwähnt, so anpassen, dass die Argumente an `calculateAge` in umgekehrt geordneter Reihenfolge übergeben werden, so war dies mit der `Applicative`-Instanz nicht möglich, aber mit der `Monad`-Instanz ist das einfach: Wir können die Jahres-Werte nämlich inspizieren:

```
responseM' :: String -> String -> Maybe String
responseM' yearOfBirth yearToday =
 (readMaybe yearOfBirth) >>=
     (\byear -> readMaybe yearToday >>=
       \tyear -> return $ show $
```

```
                  (if byear >= tyear
                   then calculateAge byear tyear
                   else calculateAge tyear byear
                  ))
calculateAge :: Int -> Int -> Int
calculateAge byear tyear =  tyear - byear
```

Ein allgemeineres Beispiel ist das folgende: Implementiere eine Funktion

`applicativeIf :: Applicative f => f Bool -> f a -> f a -> f a`,

sodass `applicativeIf aCond aThen aElse` je nachdem, ob das verpackte `aCond` ein `True` oder ein `False` enthält den Berechnungszweig `aThen` oder `aElse` einschlägt.

Der folgende Versuch sieht scheinbar richtig aus:

```
applicativeIf :: Applicative f => f Bool -> f a -> f a -> f a
applicativeIf aCond aThen aElse = ifte <$> aCond <*> aThen <*> aElse
 where ifte a b c = if a then b else c
```

Die folgenden Tests zeigen jedoch, dass „die Berechnung“ von `aCond` nur bedingten Einfluss auf die weitere Berechnung nehmen kann:

```
*> applicativeIf (Just True) (Just 1) (Just 2)
Just 1
*> applicativeIf (Just False) (Just 1) (Just 2)
Just 2
*> applicativeIf (Just True) (Just 1) Nothing
Nothing
*> applicativeIf (Just False) Nothing (Just 2)
Nothing
```

Während die ersten beiden Ergebnisse wie erwartet sind, hätten wir für die letzten beiden Aufrufe nicht `Nothing` als Ergebnis gewollt.

Mit der Monaden-Instanz ist die verlangte Funktionalität leicht zu implementieren, da wir direkten Zugriff auf den in `aCond` verpackten Booleschen Wert haben:

```
monadicIf :: Monad m => m Bool -> m b -> m b -> m b
monadicIf aCond aThen aElse = aCond >>= \r -> if r then aThen else aElse
```

Diese Implementierung verhält sich, wie es gewünscht war:

```
*> monadicIf  (Just True) (Just 1) (Just 2)
Just 1
*> monadicIf  (Just False) (Just 1) (Just 2)
Just 2
*> monadicIf  (Just True) (Just 1) Nothing
Just 1
*> monadicIf  (Just False) Nothing (Just 2)
Just 2
```

6.2.1 Monaden sind Applikative Funktoren

Jede Monade ist ein Applikativer Funktor, denn es gilt sowohl

```
fmap f mx   ==   mx >>= return . f
```

als auch

```
pure        ==   return
mf <*> mx   ==   mf >>= (\f -> mx >>= return . f)
```

Mithilfe dieser Gleichungen kann man für jede Monaden-Instanz auch `Functor` und `Applicative`-Instanzen erstellen. Die Umkehrung gilt nicht, denn nicht alle Applikativen Funktoren sind auch Monaden. Z.B. ist `ZipList` keine Monade.

6.2.2 Do-Notation

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 eine 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. Verwendet man Patterns in `pat <- aktion` oder `let pat = e`, so muss der entsprechende Typkonstruktor auch eine Instanz der Klasse `MonadFail` zur Verfügung stellen, denn falls der Match fehl schlägt, so wird die `fail`-Funktion aus `MonadFail` aufgerufen.

Die `do`-Blöcke ähneln der imperativen Programmierung. Die Funktion `response` kann mit `do` implementiert werden als:

```
responseM'' :: String -> String -> Maybe String
responseM'' yearOfBirth yearToday =
 do
    byear <- readMaybe yearOfBirth
    tyear <- readMaybe yearToday
    return $ show
      (if byear >= tyear
        then calculateAge byear tyear
        else calculateAge tyear byear)
```

Die `do`-Notation ist nur syntaktischer Zucker, sie kann durch die monadischen Operatoren `>>=` und `>>` dargestellt werden. Die Übersetzung ist[4]:

[4] Beachte, dass wir hier Semikolon und geschweifte Klammern verwenden, in der Funktion `response` haben wir dies durch Einrückung vermieden

```
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 `responseM"`, so erhält man gerade die ursprüngliche Definition:

```
responseM'' yearOfBirth yearToday =
 do
    byear <- readMaybe yearOfBirth
    tyear <- readMaybe yearToday
    return $ show
      (if byear >= tyear
        then calculateAge byear tyear
        else calculateAge tyear byear)

==>
responseM'' yearOfBirth yearToday =
    (readMaybe yearOfBirth) >>= \byear ->
     do
      tyear <- readMaybe yearToday
      return $ show
       (if byear >= tyear
         then calculateAge byear tyear
         else calculateAge tyear byear)

==>
responseM'' yearOfBirth yearToday =
    (readMaybe yearOfBirth) >>= \byear ->
      (readMaybe yearToday) >>= \tyear ->
       do
        return $ show
         (if byear >= tyear
           then calculateAge byear tyear
           else calculateAge tyear byear)
==>
responseM'' yearOfBirth yearToday =
    (readMaybe yearOfBirth) >>= \byear ->
      (readMaybe yearToday) >>= \tyear ->
        return $ show
         (if byear >= tyear
           then calculateAge byear tyear
           else calculateAge tyear byear)
```

Oft können anstelle der monadischen do-Notation, die Operationen des Applikativen Funktors verwendet werden. Z.B. sieht man häufig

```
do
 f <- mf
 x <- mx
 return (f x)
```

Die Verwendung der Monade ist hier unnötig, da `mf <*> mx` dasselbe liefert, kürzer und prägnanter ist. Insbesondere wird durch letztere Notation auch ausgedrückt, dass die Effekte der Ausführung von `mf` und `mx` sich nicht gegenseitig beeinflussen können, während dies bei der monadischen Variante mit `do` nicht zugesichert wird.
Mit der Spracherweiterung `ApplicativeDo` kann die `do`-Notation auch für applikative Funktoren verwendet werden. Haskell versucht dann innerhalb der `do`-Blöcke – falls möglich – auf Applicative zurück zu greifen[5]:

```
Prelude> :set -XApplicativeDo
Prelude> let fun mf mx = do {f <- mf; x <- mx; return $ f x}
Prelude> :type fun
fun :: Applicative f => f (t -> b) -> f t -> f b
Prelude> :unset -XApplicativeDo
Prelude> let fun mf mx = do {f <- mf; x <- mx; return $ f x}
Prelude> :type fun
fun :: Monad m => m (t -> b) -> m t -> m b
```

Betrachte als weiteres Beispiel, den Code:

```
do
   a <- ma
   b <- mb
   c <- (mc a)
   d <- (md b)
   return (c,d)
```

In diesem Fall hängen die Berechnungen

```
   a <- ma
   b <- mb
```

und

```
   c <- (mc a)
   d <- (md b)
```

untereinander jeweils nicht voneinander ab und können daher mit den Operatoren von `Applicative` ausgedrückt werden, während jedoch die beiden Gruppen sehr wohl voneinander abhängen (da `a` und `b` verwendet werden).
Eine mögliche Übersetzung zum Wegkodieren der do-Notation ist daher

[5] mehr Details zur Übersetzung und zum Hintergrund sind in (MPJKM16) zu finden

```
((,) <$> ma <*> mb) >>= \(a,b) -> (,) <$> (mc a) <*> (md b)
```

Dabei werden `ma` und `mb` durch die `Applicative`-Instanz mit <*> verkettet und genauso werden `(mc a)` und `(md b)` verkettet. Die beiden Blöcke müssen jedoch monadisch (in diesem Fall mit >>= verbunden). Der GHC verwendet statt >>= die in `Control.Monad` definierte Funktion

```
join :: Monad m => m (m a) -> m a
join mma = mma >>= \ma -> ma
```

und übersetzt den Block (mit der ApplicativeDo-Erweiterung) in

```
join ((\a b -> (,) <$> (mc a) <*> (md b)) <$> ma <*> mb)
```

welche äquivalent aber etwas effizienter als die vorherige ist, da kein Paar erstellt wird, dass im Anschluss direkt wieder zerlegt wird.

6.2.3 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`$)$

Der erste Gesetz besagt, dass `return` links-neutral bezüglich >>= 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.

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 $\perp$), dann divergiert auch die Auswertung von $\perp$ `>>= return` (und ist daher gleich zu $\perp$).

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)`.

Für Instanzen der Klasse `MonadFail` muss das Gesetz

```
fail s >>= f  = fail s
```

gelten.

6.2.4 Die Listen-Monade

Auch Listen sind Instanzen der Klasse `Monad` (und `MonadFail`). Die Instanzdefinitionen sind:

```
instance Monad [] where
   m >>= f  = concatMap f m
   return x = [x]

instance MonadFail [] where
   fail s   = []
```

Die bind-Operation hat dabei den Typ `[a] -> (a -> [b]) -> [b]` und in `m >>= f` wird die Funktion `f` auf alle Elemente der Liste `m` angewendet. Da `f` selbst eine Liste von Ergebnissen liefert werden diese konkateniert. Die Implementierung der >>= und `return`-Operation entspricht dem nichtdeterministischen Ausprobieren aller Möglichkeiten, wobei das Resultat die Liste aller Möglichkeiten darstellt. Die `fail`-Operation führt zu keinem Resultat, welches entsprechend durch die leere Liste dargestellt wird. Die Monaden-Instanz entspricht dabei 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)
```

Beides mal 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 ein 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 () 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 diese nicht nach, machen aber auch hier die Bemerkung, dass man diese für für beliebige (auch unendlich lange) Listen nachweisen muss.

Eine Verfeinerung von Monaden wird durch die Klasse `MonadPlus` definiert. Dies sind solche Monaden, die einen Auswahloperator mit Fehlerausgang bereitstellt, d.h. sie stellen ein Monoid für Monaden zur Verfügung:

```
class (Alternative m, Monad m) => MonadPlus (m :: * -> *) where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Dabei ist `Alternative` ein Monoid für einen applikativen Functor:

```
class Applicative f => Alternative (f :: * -> *) where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

```
  some :: f a -> f [a]
  many :: f a -> f [a]
```

Die Instanzen für Listen sind:

```
instance MonadPlus [] where
   mzero = []
   mplus = (++)


instance Alternative [] where
    empty = []
    (<|>) = (++)
```

Die Gesetze und weitere Informationen zu beiden Klassen können der Dokumentation zu den Modulen `Control.Monad` und `Control.Applicative` entnommen werden. Mithilfe von `MonadPlus` kann man z.B. generisch die Funktion `guard` definieren:

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

Damit kann man die List Comprehensions noch eleganter mit der `do`-Notation ausdrücken:

```
do
  x <- [1..10]
  y <- [1,2]
  guard (x*y < 15)
  return (x*y)
```

6.2.5 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.

Nützliche Operatoren für Monaden sind:

- `(=<<) :: Monad m => (a -> m b) -> m a -> m b`: Entspricht `(flip (>>=))`
- `(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c`: Links-nach-rechts monadische Komposition, `(f >=> g) x` entspricht `do {y <- f x; g y}`
- `(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c`: Rechst-nach-links monadische Komposition, `(f <=< g) x` entspricht `do {y <- g x; f y}`

Die Funktion `replicateM` führt Aktionen mehrfach aus, der Typ ist

```
replicateM :: Applicative m => Int -> m a -> m [a]
```

Ein Aufruf `replicateM n act` führt die Aktion `act` `n`-Mal hintereinander aus und verknüpft die Ergebnisse in einer Liste.

Die Funktion `forever` führt eine monadische Aktion unendlich lange wiederholt aus, sie ist definiert als:

```
forever   :: (Applicative f) => f a -> f b
forever a = let a' = a *> a' in a'
```

Die Funktion `when` verhält sich wie ein imperatives `if-then` (also eine bedingte Verzweigung ohne `else`-Zweig). Sie ist definiert als

```
when      :: (Applicative m) => Bool -> m () -> m ()
when p s  =  if p then s else return ()
```

Analog dazu ist `unless`:

```
unless      :: (Applicative m) => Bool -> m () -> m ()
unless p s  =  if p then return () else s
```

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)
```

In `Control.Monad` ist `sequence` allgemeiner definiert mit dem Typ

```
sequence :: (Traversable t, Monad m) => t (m a) -> m (t a)
```

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)
```

Beachte, dass auch hier, die Funktionen in den Bibliotheken etwas verallgemeinert sind, und anstelle der Listen eine Instanz von `Traversable` verwenden.
Zum Beispiel kann man damit ganz einfach die ersten einhundert Zahlen Zeilenweise ausdrucken:

```
*Main> mapM_ print [1..100]
1
2
3
4
5
...
```

Die Funktionen `forM` und `forM_` kann man als Ersatz für for-Schleifen sehen:

```
forM :: Monad m => [a] -> (a -> m b) -> m [b]
forM = flip mapM
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
forM_ = flip mapM_
```

Damit kann man z.B. schreiben:

```
import Control.Monad
main = do
  namen <- forM [1,2,3,4] (\i ->
                do
                 putStrLn $ "Gib den " ++ show i ++ ". Namen ein!"
                 getLine
               )
  putStrLn "Die Namen sind"
  forM_ [1,2,3,4] (\i -> putStrLn $ show i ++ "." ++ (namen!!(i-1)))
```

Eine Verallgemeinerung der `filter`-Funktion ist

```
mfilter :: MonadPlus m => (a -> Bool) -> m a -> m a
```

Dabei werden in `mfilter p act` die in `act` verpackten Werte x herausgefiltert, für die `p` x wahr ist. Im Ergebnis werden diese Werte wieder verpackt.

Ähnlich aber doch verschieden ist

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

Hiermit wird ein monadischer Filter auf eine Liste angewendet.

Betrachte zum Beispiel die Berechnung der Potenzmenge, also aller Teilmengen einer Menge:

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True, False]) xs
```

Da die Listenmonade verwendet wird, ist die Berechnung wie eine nichtdeterministische Berechnung: Für jedes Element werden alle Möglichkeiten erzeugt: Das Element ist in der Ausgabe oder es ist nicht in der Ausgabe. Betrachte z.B.

```
*> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

Die Funktion

```
foldM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b
```

ist analog zur `fold`-Funktion, aber die Ergebnisse werden in der Monade verpackt.

Die Funktion

```
zipWithM :: Applicative m => (a -> b -> m c) -> [a] -> [b] -> m [c]
```

ist analog zu `zipWith`, aber die Ergebnisse werden in der Struktur verpackt.

Die Funktion `msum` „summiert“ die Ergebnisse (als Verallgemeinerung von `concat`). Sie kann definiert werden durch:

```
msum :: (Foldable t, MonadPlus m) => t (m a) -> m a
msum = foldr mplus mzero
```

Die Funktion `join` wickelt verschachtelte Monaden aus:

```
join :: Monad m => m (m a) -> m a
join mma = mma >>= id
```

6.3 Die Zustandsmonade

Will man zustandsbasiert programmieren, so kann man dies in einer funktionalen Programmiersprache wie Haskell mithilfe einer Monade bewerkstelligen. Der wesentliche Trick besteht dabei darin, dass man im Datentyp *nicht* den Zustand selbst, sondern den *Effekt* des Verändern eines Zustands darstellt, und diesen dann zur Monaden-Instanz macht.

In mancher Literatur heißen diese Datentypen dementsprechend StateTransformer, wir benennen sie jedoch (wie die Standardbibliotheken) nur als Zustand d.h. `State`[6].

Als Beispiel werden wir die Programmierung eines (einfachen) Taschenrechners betrachten, der die Eingabe als String von links nach rechts liest und keine Punkt-vor-Strich-Rechnung beachtet.

Allgemein kann ein Zustandveränderer als Funktion vom Typ `s -> (a,s)` aufgefasst werden, wobei `s` der Zustand ist (der verändert wird) und `a` ein Ergebnis der Aktion ist. Dies kann durch den folgenden Typ in Haskell ausgedrückt werden:

```
newtype State s a = State (s -> (a,s))
```

Für unseren Taschenrechner werden wir für den Zustand (d.h. den Parameter `s`) ein Paar benutzen, das aus einer Funktion und einer Zahl besteht (beides für Fließkommazahlen). Wir definieren daher die Abkürzungen:

```
type InternalCalcState = (Double -> Double, Double)
type CalcState a = State InternalCalcState a
```

Wir benötigen primitive Operationen, die auf dem Taschenrechner operieren und z.B. bei Eingabe von `'+'` das Paar entsprechend verändern. Die Monadeninstanz liefert uns dann die sequentielle Komposition dieser Operationen. Wir machen es jedoch etwas generischer und definieren allgemeine primitive Operationen für `State`: `put`, um etwas in den inneren Zustand zu schreiben (das erzeugte zusätzliche Ergebnis ist dabei `()`) und `get`, um den inneren Zustand auszulesen.

```
put :: s -> State s ()
put x = State $ \_ -> ((),x)

get :: State s s
get = State $ \s -> (s,s)
```

Die Instanz der Klasse Monad kann nun wie folgt definiert werden:

```
instance Monad (State s) where
  -- return :: a -> State s a
  return x  = State $ \s -> (x,s)
```

[6] Der Grund ist, dass wir später sogenannte Monaden-Transformer einführen, und dann nicht von StateTransformer-Transformern sprechen müssen.

```
  --  (>>=) :: State s a -> (a -> State s b) -> State s b
  State x >>= f =
        State $ \s0 -> let (val_x, s1)  = x s0
                           (State cont) = f val_x
                       in  cont s1
```

Die Instanzen für `Functor` und `Applicative` lassen sich dabei direkt programmieren mit

```
instance Functor (State s) where
  fmap f mx  = mx >>= return . f

instance Applicative (State s) where
  pure       = return
  mf <*> mx  = mf >>= (\f -> mx >>= return . f)
```

Damit kann man z.B. eine Funktion `modify` programmieren, die es erlaubt den inneren Zustand zu verändern.

```
modify :: (s -> s) -> State s ()
modify f = do
            a <- get   -- aktuellen Zustand lesen
            put (f a)  -- veränderten Zustand schreiben
```

Es fehlt noch eine Funktion, um diese Zustandsveränderer nun wirklich laufen zu lassen. Diese muss den Zustandsveränderer auf ein initialen Zustand anwenden:

```
runState :: State s a -> s -> (a,s)
runState (State x) i = x i
```

Beachte: Bei der Programmierung mit den Zustandsveränderern sieht man die Zustände nicht explizit (es sei denn man fordert sie explizit z.B. mit `get` an).

Wir können z.B. ein Programm schreiben, das einen `Int`-Zähler mehrfach erhöht:

```
run = runState prg 0
 where prg = do  modify (+1)
                 modify (+1)
                 modify (+1)
                 modify (+1)
                 get
```

Wie erwartet liefert das Programm `(4,4)` als Rückgabe.

Für den Taschenrechner ist der gespeicherte Zustand ein Paar. Die erste Komponente ist eine Funktion, die zweite Komponente eine Zahl. Die erste Komponente enthält das aktuelle Zwischenergebnis und die Operation die noch angewendet werden muss, die zweite Komponente die momentan eingegebene 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)`	

Für den Taschenrechner können wir den Startzustand definieren als

```
start :: InternalCalcState
start = (id, 0.0)
```

Nun besteht unsere Aufgabe im Wesentlichen darin, verschiedene `CalcState`-Aktionen zu definieren.

Die Funktion `oper` implementiert die Zustandsveränderung für jeden beliebigen binären Operator des Taschenrechners: Dabei wird die aktuelle Funktion auf die Zahl angewendet und darauf wird der Operator angewendet. Die Zahl wird schließlich auf 0 gesetzt. Mit der vordefinierten `modify`-Funktion geht das einfach:

```
oper :: (Double -> Double -> Double) -> CalcState ()
oper op = modify (\ (fn,num) -> (op (fn num), 0))
```

Die Funktion `clear` löscht die letzte Eingabe (wenn die Zahl 0 ist, wird die Funktion gelöscht, ansonsten die Zahl)

```
clear :: CalcState ()
clear =  modify (\ (fn,num) -> if num == 0 then (id,0.0) else (fn,0.0))
```

Die Funktion `total` berechnet das Ergebnis

```
total :: CalcState ()
total = do (fn,num) <- get
           put (id, fn num)
```

Die Funktion `digit` verarbeitet eine Ziffer

```
digit :: Int -> CalcState ()
digit i = do
           (fn,num) <- get
           put (fn,num*10 + (fromIntegral i) ) -- Ziffern verschieben und Ziffer hi
```

Schließlich definieren wir noch eine Funktion `readResult`, die das aktuelle Ergebnis aus dem Zustand ausliest, der Rückgabewert ist eine `Double`-Zahl

```
readResult:: CalcState Double
readResult = do (fn,num) <- get
                return (fn num)
```

Als nächstes definieren wir die Funktion `calcStep`, die ein Zeichen der Eingabe verarbeiten kann:

```
calcStep :: Char -> CalcState ()
calcStep x
 | isDigit x = digit (fromIntegral $ digitToInt x)

calcStep '+' = oper (+)
calcStep '-' = oper (-)
calcStep '*' = oper (*)
calcStep '/' = oper (/)
calcStep '=' = total
calcStep 'c' = clear
calcStep _   = return () -- nichts machen
```

Für die Verarbeitung der gesamten Eingabe (als Text) definieren wir die Funktion `calc`. Diese benutzt die monadische `do`-Notation:

```
calc xs = do
            mapM_ calcStep xs
            readResult
```

Zum Ausführen des Taschenrechners muss nun die gesamte Aktion auf den Startzustand angewendet werden. Wir definieren dies als

```
runCalc :: CalcState Double -> (Double,InternalCalcState)
runCalc act = runState act start
```

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>
```

Allerdings fehlt dem Taschenrechner noch die Interaktion (z.B. erscheint das =-Zeichen noch eher nutzlos). Wir werden den Taschenrechner später erweitern, widmen uns im nächsten Abschnitt zunächst aber der Programmierung von Ein- und Ausgabe in Haskell.

Zuvor erwähnen wir noch, dass der hier verwendete Typ `State` durch die Bibliothek `Control.Monad.Trans.State` grundsätzlich in der gleichen Weise zur Verfügung gestellt wird (und daher nicht neu implementiert werden muss). Die Interna sind dabei leicht verschieden, wir werden darauf später zurück kommen. Schließlich gibt es noch spezielle Varianten, wenn der Zustand nur gelesen, aber nicht verändert werden soll: Dies wir durch den Typ `Reader r a` in der Bibliothek `Control.Monad.Trans.Reader` bewerkstelligt (an die Stelle von `get` tritt die Funktion `ask :: Reader r r`. Ein kleines Beispiel ist

```
*> runReader (do {a <- ask; b <-ask; return (a,b)}) 10
(10,10)
```

Als größerer Beispiel betrachte diese Variante der Auswertung von arithmetischen Ausdrücken mit Umgebung:

```
import Control.Monad.Trans.Reader

data Exp v = Var v | Val Int | Neg (Exp v) | Add (Exp v ) (Exp v )
type Env v = [(v,Int)]


fetch :: String -> Reader (Env String) Int
fetch x = do
          env  <- ask
          case lookup x env of
           Nothing ->  error "Variable not found"
           Just val -> return val

eval :: Exp String -> Reader (Env String) Int
eval  (Var x)  = fetch x
eval  (Val i)  = return i
eval  (Neg p)  = do  r <- eval p
                     return (negate r)
eval  (Add p q) = do e1 <- eval p
                     e2 <- eval q
                     return (e1+e2)
evaluate e env = runReader (eval e) env
```

Ein Beispielaufruf ist:

```
*Main> evaluate (Add (Val 1) (Var "x")) [("x",5)]
6
```

Ebenso gibt es eine Variante der Zustandsmonade, wenn der Zustand nur beschrieben, aber nie gelesen wird: Dies wird durch den Typ `Writer` in `Control.Monad.Trans.Writer` implementiert.

6.4 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
  ```

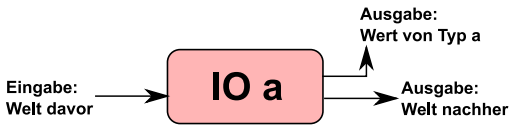
  Da die Auswertung von `length` die Listenelemente gar nicht anfasst, würde obiger Aufruf, keine `getZahl`-Aufrufe ausführen.
- In reinen funktionalen Programmiersprachen wird oft auf die Festlegung einer genauen Auswertungsreihenfolge verzichtet, um Optimierungen und auch Parallelisierung von Programmen 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` zur 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 genaue 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 `State` 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:



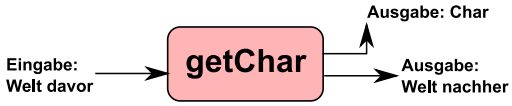
Aus Sicht von Haskell sind Objekte vom Typ `IO a` bereits *Werte*, d.h. sie können nicht weiter ausgewertet werden. Dies passt dazu, dass auch andere Funktionen Werte in Haskell sind. Allerdings im Gegensatz zu „normalen“ Funktionen kann Haskell kein Argument vom Typ „Welt“ bereit stellen. (Die Funktion `runState` für die `State`-Monade 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.

6.4.1 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 Funktion `getChar`:

```
getChar :: IO Char
```

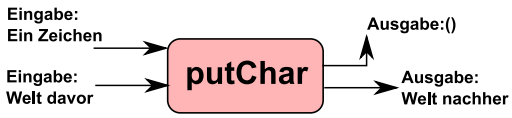
In der Welt-Sichtweise ist `getChar` eine Funktion, die eine Welt erhält und als Ergebnis eine veränderte Welt sowieso ein Zeichen liefert. Man kann dies durch folgendes Bild illustrieren:



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.

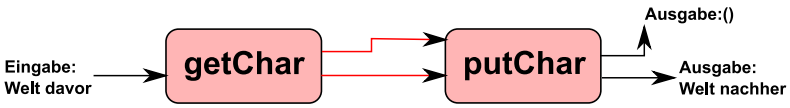
```
putChar :: Char -> IO ()
```

Auch `putChar` lässt sich mit einem Bild illustrieren:



6.4.2 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` wie folgt 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 Ausgeben 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:

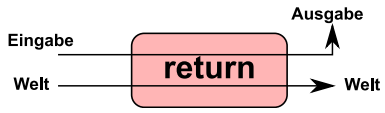
```
echoDup :: IO ()
echoDup = getChar >>= \x -> (putChar x >> putChar x)
```

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)
```

6.4.3 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 State-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.

6.4.4 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` [·] `True` unterscheidet dann `return e1 >>= 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 Abweichungen von den monadischen Gesetzen in Kauf genommen.

6.4.5 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 `unIO :: 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 zurückgegeben 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:

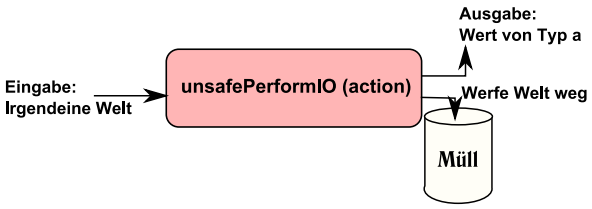
```
do
 ...
 ergebnis <- unsafeInterleaveIO aktion
 weitere_Aktionen
```

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.

6.4.6 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 `IORef`s. 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).

6.5 Monad-Transformer

Wir kehren zurück zu den Zustandsmonaden und 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:

```
main = do c <- getChar
          if c /= '\n' then do calcStep c
                               main
           else return ()
```

Leider funktioniert die Implementierung so nicht, da dort monadische Aktionen *zweier verschiedener* Monaden vermischt werden: `getChar` ist eine Aktion der `IO`-Monade während `calcStep c` eine Aktion der `State`-Monade ist. Ein Ausweg aus dieser Situation ist es, die `State`-Monade zu erweitern, so dass sie `IO`-Aktionen auch verarbeiten kann. Man kann dies gleich allgemein machen: Die `State`-Monade wird um eine beliebige andere Monade erweitert. Wir definieren hierfür den Typ `StateT`:

```
newtype StateT state monad a = StateT (state -> monad (a,state))
```

Im Gegensatz zu `State` 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 `StateT` ist[7]:

```
instance Monad m => Monad (StateT s m) where
    return x = StateT $ \s -> return (x, s)
    (StateT x) >>= f  = StateT $ \s ->  do
                                          (a,s') <- x s
                                          case (f a) of
                                             (StateT y) -> (y s')
```

Entsprechend sind `get`, `put`, `modify` und `runStateT` definiert als

```
put :: Monad m => s -> StateT s m ()
put x = StateT $ \_ -> return ((),x)

get :: Monad m => StateT s m s
get = StateT $ \s -> return (s,s)

modify :: Monad m => (s -> s) -> StateT s m ()
modify f = do
            a <- get
            put (f a)

runStateT :: StateT s m a -> s -> m (a, s)
runStateT (StateT x) s = x s
```

[7] Die Instanzen für `Functor` und `Applicative` erläutern wir an dieser Stelle nicht.

Tatsächlich ist der in `Control.Monad.Trans.State` definierte Typ für `State` nur ein Synonym:

```
type State s a = StateT s Identity a
```

D.h. es wurde der State-Transformer verwendet, wobei die innere Monade der Typ `Identity` ist. Die Identity-Monade tut im Grunde nichts:

```
newtype Identity a = Identity a
instance Monad Identity where
 return x = Identity x
 (Identity m) >>= f  = (f m)
instance Applicative Identity where
 pure = return
 (Identity f) <*> (Identity x) = Identity (f x)
instance Functor Identity where
 fmap f (Identity x) = Identity (f x)
```

Die Funktion `runState` kann dann mit den neuen Typen definiert werden als

```
runState :: State s a -> s -> (a,s)
runState f s = case runStateT f s of
                 Identity r -> r
```

Für den Taschenrechner definieren wir noch als Abkürzung ein Typsynonym: Der verwendete Typ `StateT` bindet nun die `IO`-Monade mit ein:

```
type CalcStateIO a = StateT InternalCalcState IO a
```

Das nächste Ziel ist es, die bereits für `CalcState` definierten Funktionen für den neuen Typ `CalcStateIO` zu verwenden: Dies ist für Funktionen, die nicht geändert werden, einfach: Wir brauchen nichts zu tun (außer die Typen zu ändern), da sie auch für die `StateT`-Monade direkt funktionieren. Um in Aktionen der inneren Monade (in unserem Fall, die `IO`-Monade) auszuführen, kann man eine allgemeine Funktion zum „liften“ definieren, die bereits durch die Typklasse `MonadTrans` (im Modul `Control.Monad.Trans.Class`) spezifiziert wird:

```
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

Für `StateT`sieht die Instanz folgendermaßen aus:

```
instance MonadTrans (StateT s) where
  lift m = StateT $ \s -> do
                            a <- m
                            return (a,s)
```

Die erstellte Aktion führt zunächst die Operation in der Monade `m` aus, und verpackt dann das erhaltene Ergebnis in die `StateT`-Monade.

Die Implementierungen von `clear` und `total` möchten wir anpassen, da sie nun auch Ausgaben durchführen sollen. Hier können wir nun die `IO`-Monade verwenden:

```
total' :: CalcStateIO ()
total' =
 do (fn,num) <- get
    lift $ putStr $ show (fn num)
    put (id,fn num)

clear' :: CalcStateIO ()
clear' =
 do (fn,num) <- get
    if num == 0.0 then do
       lift $ putStr ("\r"  ++ (replicate 100 ' ') ++ "\r")
       put start
     else do
       let l = length (show num)
       lift $ putStr $ (replicate l '\b') ++ (replicate l ' ' ) ++ (replicate l '\b
       put (fn,0.0)
```

Die Definition von `calcStep` ist wie vorher:

```
calcStep' :: Char -> CalcStateIO ()
calcStep' x
 | isDigit x = digit (fromIntegral $ digitToInt x)

calcStep' '+' = oper (+)
calcStep' '-' = oper (-)
calcStep' '*' = oper (*)
calcStep' '/' = oper (/)
calcStep' '=' = total'
calcStep' 'c' = clear'
calcStep' _   = return ()
```

Die Hauptschleife findet in der Funktion `calc'` statt. Diese liest ein Zeichen, berechnet den Folgezustand und ruft sich anschließend selbst auf. Die verwendete Monade ist die `StateT`-Monade. Daher kann `getChar` aus der `IO`-Monade nicht direkt verwendet werden. Diese muss erst in die `StateT`-Monade geliftet werden.

```
calc' :: CalcStateIO ()
calc' = do
```

```
          c <- lift $ 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
          runStateT calc' start
```

Zusammenfassend dienen Monad-Transformer dazu, mehrere Monaden zu vereinen. Dabei kann man mit `lift`-Funktionen bestehende Funktionen relativ komfortabel in die neue Monade liften.

Übungsaufgabe 6.5.1. *Der vorgestellte Taschenrechner kann als Eingabe nur nicht-negative ganze Zahlen verarbeiten. Erweitern Sie den Taschenrechner, so dass auch Kommazahlen eingegeben werden können.*

6.6 Weitere Anwendungen von und Bemerkungen zu Monaden

Im Taschenrechnerbeispiel haben wir zwei Monaden miteinander vereint. Das selbe Prinzip kann jedoch verwendet werden, um beliebige Monaden miteinander zu vereinen. Man erhält dann einen Stack von Monaden.

Wir demonstrieren dies an einem Beispiel: Angenommen, wir möchten arithmetische Ausdrücke mit Umgebung auswerten und dabei die Umgebung mit einem `Reader` verwalten, ein Loggen der lookups mit einem `Writer` verwalten und schließlich noch Ein- und Ausgaben machen, dann kann dies durch Verschachtelten der Monaden implementiert werden:

```
fetch :: String -> ReaderT (Env String) (WriterT [String] IO) Int
fetch x = do
          env  <- ask
          case lookup x env of
           Nothing ->  do
                         lift $ tell [("not found " ++ x)]
                         return 0
           Just val -> do lift $ tell [("found " ++ x)]
                         return val
```

```
eval :: Exp String -> ReaderT (Env String) (WriterT [String] IO) Int
eval  (Var x)  = fetch x
eval  (Val i)  = return i
eval  (Neg p)  = do
                   r <- eval p
                   lift $ lift $ putStrLn "computed a negation"
                   return (negate r)
eval  (Add p q) = do
                   e1 <- eval p
                   e2 <- eval q
                   lift $ lift $ putStrLn "computed a sum"
                   return (e1+e2)
```

Ein Aufruf ist

```
*Main> runWriterT (runReaderT  (eval (Add (Var "x") (Neg (Var "y"))))  [("x",1)])
computed a negation
computed a sum
(1,["found x","not found y"])
```

Die Anzahl der `lift`-Operationen hängt von der Tiefe der entsprechenden Monade ab. Eine kleine Abhilfe bietet `liftIO :: MonadIO m => IO a -> m a` aus `Control.Monad.Trans`, welche die `IO`-Monade beliebig hoch liften kann (wenn alle Monaden Instanzen von `MonadIO` sind). Eine andere Abhilfe bietet das `monad-tf`-Paket, welches Typ-Familien verwendet und damit automatisch die passende `ask` oder `tell`-Operation ermitteln kann, ohne dass man explizite `lift`s verwendet.

6.6.1 ST-Monade

Als Alternative zur Zustandsmonade wird in `Control.Monad.ST` und `Data.STRef`die Typen `ST` und `STRef` zur Verfügung gestellt, die echte, funktionale Speicherplätze bietet (im Grunde analog zu `IORef`s, aber in der `ST`-Monade anstelle der `IO`-Monade). Das Interface dazu ist

```
data ST s a                                  -- Instanz von Monad

runST :: (forall s. ST s a) -> a
newSTRef    :: a -> ST s (STRef s a)         -- Allokation
readSTRef   :: STRef s a -> ST s a           -- Lesen
writeSTRef  :: STRef s a -> a -> ST s ()     -- Schreiben
modifySTRef :: STRef s a -> (a -> a) -> ST s ()
```

Das `forall` im Typ von `runST` ist ein Trick, der verhindert, dass eine Referenz als Ergebnis zurückgeben wird und in einem anderen `runST` verwendet werden kann. Das Typsystem stellt sicher, dass jedes `runST` eigene, unabhängige Referenzen hat.

Ein Beispiel zur Verwendung der `ST`-Monade ist:

```
demo :: String
demo = runST $ do
        i <- newSTRef 1
        b <- newSTRef True
        action 2 i b

action :: Int -> STRef s Int ->
     STRef s Bool -> ST s String
action c i b = do
  x <- readSTRef i
  writeSTRef i $ x + 10*c
  y <- readSTRef i
  writeSTRef b $ x > y
  writeSTRef i $ x + 100*c
  inc i
  inc i
  z <- readSTRef i
  return $ show [x,y,z]

inc :: Num a => STRef s a -> ST s ()
inc r = modifySTRef r (+1)
```

Im Programm werden zuerst je eine Referenz auf ein `Int` und ein `Bool` erstellt.

Danach können Referenzen beliebig gelesen und geschrieben werden; sind aber niemals leer.

Ausführung ergibt:

```
*> demo
"[1,21,203]"
```

Beachte: In `demo` würde `return i` am Ende zu Typfehler führen. In `action` wäre das aber erlaubt.

Der Grund ist der `forall`-Typ. Z.B. ist ebenso falsch (compiliert nicht):

```
fehlerdemo = let x = runST $ do x <- newSTRef 42
                                return x
             in      runST $ do y <- readSTRef x
                                return y
```

Wegen `runST :: (forall s. ST s a)->a` kann der Typ der Referenz nicht in einem anderen `runST` verwendet werden. Somit wird sichergestellt, dass verschiedene Berechnungen mit Zustand voneinander unabhängig sein müssen.

6.6.2 Fehlermonade

In `Control.Monad.Trans.Except` wird die Fehlermonade `Except` (bzw. `ExceptT`) definiert. Neben dem Werfen von Ausnahmen ist auch das Abfangen und die Behandlung von Fehlern damit möglich. Das Interface dazu ist:

```
type Except e a = ExceptT e Identity a

runExceptT :: ExceptT e m a -> m (Either e a)

throwE :: Monad m => e -> ExceptT e m a
catchE :: Monad m => ExceptT e1 m a
                     -> (e1 -> ExceptT e2 m a)
                     -> ExceptT e2 m a
```

Es handelt sich wieder um einen Monaden-Transformer: Für `ExceptT e m a` ist `e` der Typ der Ausnahmen, `m` die innere Monade und `a` der Typ des funktionalen Ergebnis der Berechnung.

Die Grundidee ist die der `Either e`-Monade, welche wiederum ganz ähnlich wie bei der Maybe-Monade funktioniert, nur das anstatt `Nothing` nun eine `String`-Fehlermeldung transportiert wird:

```
data Either a b = Left a | Right a
instance Monad (Either a) where
      return x       = Right x
      Left  e >>= _  = Left e
      Right x >>= mf = mf x

instance Monad m => Monad (ExceptT e m) where
      return = ExceptT . return . Right
      mx >>= mf = ExceptT $ do
        x <- runExceptT mx
        case x of
          Left  e -> return $ Left e
          Right y -> runExceptT $ mf y
```

Wir verzichten auf die Implementierungsdetails von `throwE` und `catchE`. Beachte, dass Ausnahmebehandlung mit `IO`, nicht die `Except`-Monade verwendet, sondern sich auf eingebaute Mechanismen stützt.

6.6.3 Beispiel: Werte mit Wahrscheinlichkeiten

Wir betrachten ein Beispiel aus (Lip11). Wir modellieren mehrere Ergebniswerte mit Wahrscheinlichkeiten.

```
import Data.Ratio


newtype Prob a = Prob [(a,Rational)] deriving Show
getProb :: Prob a -> [(a,Rational)]
getProb (Prob l) = l

ex1= Prob[("Blue",1%2),("Red",1%4),("Green",1%4)]
```

Dies modelliert, dass das Ergebnis zu $\frac{1}{2}$ =50% Blau ist, zu $\frac{1}{4}$ = 25% Grün, usw. Das die Summe 100% ergibt wird hier nicht modelliert.

Eine Frage, die wir uns stellen könnten: Ist `Prob` eine Monade, und wie machen wir dies zur Monade? Eine Functor-Instanz ist offensichtlich

```
instance Functor Prob where
  fmap f (Prob xs) = Prob $ map (\(x,p) -> (f x,p)) xs
```

Die Applicative-Instanz machen wir generisch mit der Monaden-Instanz:

```
instance Applicative Prob where
  pure = return
  mf <*> mx = mf >>= (mx >>=).(return.)
```

Jetzt können wir damit die `Monad`-Instanz deklarieren:

```
import Data.Ratio


instance Monad Prob where
  return x = Prob [(x,1%1)]        -- 1 Antwort, 100%
  m >>= f  = vereinigen (fmap f m) --
instance MonadFail Prob where
  fail _   = Prob []

vereinigen :: Prob (Prob a) -> Prob a
vereinigen (Prob xs) = Prob $ concat $ map multAll xs
  where
    multAll (Prob inxs,p) = map (\ (x,r) -> (x,p*r)) inxs
```

Die Idee für >>= kommt so zustande: Da `m >>= f` immer gleich zu `join (fmap f m)` ist, können wir uns fragen, wie man `join` direkt implementieren würde. In diesem Fall hat man eine Liste von Elementen mit Wahrscheinlichkeiten, wobei die Elemente selbst wieder solche Listen sind. Wie man diese Liste glättet ist mathematisch klar: Multipliziere die die innere und äußere Wahrscheinlichkeit jeweils.
Als Beispiel betrachte:

```
ex1= Prob[("Blue",1%2),("Red",1%4),("Green",1%4)]
ex2= Prob[(("Bright "++),1%5),(("Dark "++),2%5),(id,2%5)]

*> ex2 <*> ex1
  Prob[("Bright Blue",1%10), ("Bright Red",1%20)
      ,("Bright Green",1%20), ("Dark Blue",1%5)
      ,("Dark Red",1%10), ("Dark Green",1%10)
      ,("Blue",1%5), ("Red",1%10), ("Green",1%10)]
```

D.h. wenn wir zufällig eine Farbe wählen mit den gegebenen Wahrscheinlichkeiten und dann zufällig zu 20% die Farbe aufhellen oder zu 40% abdunkeln, dann ist das Ergebnis z.B. mit 5% Wahrscheinlichkeit hell rot.

6.7 Quellennachweis

Die Darstellung der Monadischen Programmierung stammt im Wesentlichen aus (Jos19) und (Lip11). Das Taschenrechnerbeispiel ist schon älter (siehe z.B. (SS09)) wurde aber an die aktuellen Bibliotheken angepasst. Die Darstellung von Haskells monadischen IO richtet sich im Wesentlichen nach (Pey01). Als weiterführende Literatur sind die Originalarbeiten von Wadler zur Verwendung von Monaden in Haskell (z.B. (Wad92; Wad95)) und die theoretische Fundierung von Moggi (Mog91) sehr empfehlenswert.

7 Parallelität, Ausnahmen und Nebenläufigkeit in Haskell

7.1 Einleitung

Das Ziel beim *parallelen Rechnen* ist die *schnelle* Ausführung von Programmen durch die gleichzeitige Verwendung mehrerer Prozessoren. Ein Problem für viele Ansätze des parallelen Rechnens ist, dass diese oft sehr systemnah und schwierig zu handhaben sind. Parallele funktionale Programmiersprachen bieten zum Teil einen abstrakteren Ansatz, so dass nur wenige Programmanpassungen zur parallelen Berechnung gemacht werden müssen. Dabei muss gesagt werden, dass die bisherigen Ansätze zur automatischen Parallelisierung nicht erfolgreich waren, so dass man händisch eingreifen muss und meist durch Testen und Ausführen überprüfen muss, ob die Parallelisierung erfolgreich ist.

Im Unterschied zur Parallelität bedeutet *Nebenläufigkeit (Concurrency)*, dass mehrere Berechnungen nichtdeterministisch ausgeführt werden, d.h. die Prozesse oder Threads laufen nichtdeterministisch abwechselnd und interagieren miteinander. Dieser Ablauf muss daher nicht notwendigerweise parallel sein, sondern kann auch abwechselnd auf einem Prozessorkern durchgeführt werden. Das Ziel nebenläufiger Berechnungen kann zwar auch Beschleunigung sein, aber oft ist der Zweck die quasi gleichzeitige Abarbeitung mehrerer Aufgaben, wie Tastatureingaben, Mausklicks, Anfragen an einen Webserver usw.

Wir werden in diesem Kapitel einen Überblick über verschiedene Möglichkeiten geben, in Haskell (bzw. mit Erweiterungen) parallel und nebenläufig zu programmieren.

7.2 Grundlegendes

Parallele Berechnung ist im Allgemeinen schwierig. Oft beginnt man mit einem sequentiellen Algorithmus und versucht diesen zu parallelisieren. Hierbei muss die Berechnung in sinnvolle unabhängige Einheiten eingeteilt werden, die dann parallel ausgewertet werden können. Schließlich muss unter Umständen das Gesamtergebnis aus den Teilergebnissen zusammengesetzt werden.

Die Beurteilung der Effizienz erfolgt oft durch Vergleich der Beschleunigung relativ zur Berechnung mit einem Prozessor. Die maximale Beschleunigung lässt sich mit *Amdahls Gesetz* ausdrücken als

$$\text{maximale Beschleunigung} \leq \frac{1}{(1-P) + P/N}$$

wobei N die Anzahl Kerne und P der parallelisierbare Anteil des Programms ist

Z.B. wenn 80% des Programms parallelisierbar sind, dann ist die maximal mögliche Beschleunigung 5, selbst wenn beliebig viele Prozessoren zur Verfügung stehen.

Als *Thread* bezeichnen wir einen Ausführungsstrang eines Programms, welcher sequentiell abläuft. Ein Programm oder Prozess kann aus mehreren Threads bestehen. Als *Core* bezeichnet man einen Prozessorkern, welcher jeweils einen Thread abarbeiten kann. Üblicherweise gibt es mehr Threads als Prozessorkerne. Das Betriebssystem (oder die Laufzeitumgebung der Programmiersprache) kümmert sich darum, alle Threads auf die Prozessorkerne zu verteilen. Für dieses sogenannte Scheduling gibt es verschiedene Strategien. Meist werden Threads regelmäßig unterbrochen, um alle Threads scheinbar gleichzeitig auszuführen.

In Haskell wird ein (virtueller) Kern als *Haskell Execution Context (HEC)* bezeichnet (auf diesen werden die Haskell-eigenen Threads ausgeführt, die Anzahl der HECs kann jedoch größer sein als die Anzahl der echten Cores).

Zur Koordination der parallelen Ausführung muss das sequentielle Programm in unabhängig parallel berechenbare Teile partitioniert werden. Daraus ergibt sich, wie viele Threads es gibt, und was ein einzelner Thread macht. Sofern Abhängigkeiten zwischen Threads bestehen, müssen die Threads *synchronisiert* werden, d.h. sie müssen aufeinander warten oder auch miteinander kommunizieren (z.B. über gemeinsamen Speicher), um Daten auszutauschen. Als *Mapping* bezeichnet man die Zuordnung der Threads zu Prozessoren und als *Scheduling* die Auswahl der laufenden Threads auf einem Prozessor.

Bei parallelen Berechnungen mit mehreren Threads können u.a. folgende Probleme auftreten:

- *Race-Condition*: Verschiedene Threads können sich durch Seiteneffekte gegenseitig beeinflussen. Da manchmal ein Thread schneller als ein anderer abgehandelt wird und die Möglichkeiten der Verzahnung immens sind, ist das Gesamtergebnis der Berechnung nicht vorhersagbar. Das Gesamtergebnis hängt daher von der konkreten nichtdeterministischen Ausführung des Programms ab. Oft sind Race-Conditions nicht gewünscht und werden dann als Programmierfehler angesehen.
- *Deadlock*: Ein Thread wartet auf das Ergebnis eines anderen Threads, welcher direkt oder indirekt selbst auf das Ergebnis des ersten Threads wartet. Die Berechnung kommt somit zum erliegen.

Diese Probleme lassen sich in *nebenläufigen* Programmen nicht generell vermeiden. In rein funktionalen *parallelen* Programmen können diese Probleme aber von vornherein eliminiert werden.

Rein funktionale Programmiersprachen haben keine Seiteneffekte und sind *referentiell transparent*, daher ist die Auswertungsreihenfolge (nahezu) beliebig, d.h. parallele Berechnung verändert das Ergebnis nicht und kann für rein funktionale Ausdrücke (auch spekulativ) durchgeführt werden, wenn sichergestellt ist, dass das Terminierungsverhalten des Gesamtprogramms nicht verändert wird. Betrachte z.B. den Ausdruck `const 1 bot`, wobei `const` und `bot` definiert sind als

```
const x y = x
bot = bot
```

Wird in diesem Fall das Argument `bot` parallel ausgewertet und die Gesamtauswertung endet erst, wenn alle parallelen Auswertungen beendet sind, dann ändert sich das Terminierungsverhalten gegenüber dem sequentiellen Programm. Denn `const 1 bot` wertet sequentiell in einem Schritt zu 1 aus.

7.2.1 Multithreading durch den GHC

Die Anzahl der verwendbaren Kerne ist im GHC einstellbar. Hierbei gbit es verschiedene Möglichkeit:

- Die Anzahl wird statisch während dem Kompilieren festgelegt, indem (neben dem Flag `-threaded`) das Flag `-with-rtsopts="-N`*number*`"` verwendet wird, wobei *number* die Anzahl der verwendbaren Kerne angibt.
- Die Anzahl wird als Kommandozeilenparameter zur Ausführung festgelegt, indem das Programm mit den Flags `-threaded` und `-rtsopts` kompiliert wird und dann bei der Ausführung mit den Optionen `+RTS -N`*number* gestartet wird
- Die Anzahl wird dynamisch (d.h. während des Ablaufs) im Programm festgelegt. Hierfür kann die durch das Modul `Control.Concurrent` zur Verfügung gestellte Funktion `setNumCapabilities :: Int -> IO ()`verwendet werden. Das Programm muss dabei ebenfalls mit `-threaded` compiliert werden, um die parallele Ausführung zu ermöglichen.

7.2.2 Messen und Analysieren des Ressourcenverhaltens

7.2.2.1 Profiling

Der GHC erlaubt *Profiling*, d.h. zur Laufzeit wird protokolliert, was das Programm wie lange macht. Dafür ist es ein spezielles Kompilieren mit den Optionen `-prof -fprof-auto -rtsopts` nötig (es empfiehlt sich auch zusätzlich `-O2` zu verwenden, damit das Programm optimiert wird) . D.h. das Kompilieren hat die Form:

```
> ghc MyProg.hs -O2 -prof -fprof-auto -rtsopts
```

Anschließend wird das Programm mit der RTS-Option `-p` ausgeführt:

```
> ./MyProg +RTS -p
```

Die Ausführung erstellt die Datei `MyProg.prof`, in der man sehen kann wie viel Zeit bei der Auswertung der einzelnen Funktionen verwendet wurde. Damit das funktioniert, müssen die benutzten Module der externen Bibliotheken ebenfalls mit Profiling-Unterstützung installiert sein. z.B. mit

```
> cabal install mein-modul -p
```

bzw. bei der Verwendung von `stack` mit

```
> stack build --profile
> stack run --profile -- Main +RTS -p
```

Für das Profiling sind viele Optionen verfügbar. Ohne `-fprof-auto` werden z.B. nur komplette Module abgerechnet. Mit `+RTS -h` wird Speicher-Profiling angeschaltet.

7.2.2.2 Statistikausgabe des Runtime-Systems

Eine einfachere Möglichkeit ohne Profiling bietet die Option `-s` für das Laufzeitsystem, d.h.

```
> ghc -O -rtsopts MyProg.hs
> ./MyProg +RTS -s
```

Nach Ablauf des Programms wird eine Statistik zu Laufzeit und Platzbedarf gedruckt. Auch der Zeitaufwand für Garbage Collection und Eckdaten zur parallelen Auswertung werden ausgedruckt. Man kann mit `-s`*Dateiname* diese Ausgabe auch in eine Datei schreiben lassen.

7.2.2.3 ThreadScope

Speziell für das Profiling von parallel/nebenläufig ausgeführten Haskell Programmen gibt es den *ThreadScope*-Profiler (wiki.haskell.org/ThreadScope).
Die Verwendung ist wie folgt

```
> ghc MyPrg.hs -threaded -eventlog -rtsopts
> ./MyPrg +RTS -N4 -l
> threadscope MyPrg.eventlog
```

D.h. zunächst wird das Programm mit den Optionen `-threaded`, `-eventlog` und `-rtsopts` compiliert. Danach wird das Programm ausgeführt, wobei die Option `-l` das Event-Logging anschaltet. Schließlich visualisiert ThreadScope visualisiert dann das Event-Log.

Abbildung 7.1 zeigt eine Ansicht von ThreadScope. Die Ansicht zeigt, dass anfangs zwei Kerne genutzt wurden (HEC1, HEC2), während der dritte Kern (HEC0) komplett ungenutzt blieb. Arbeitsphasen sind dabei in Grün dargestellt, Garbage-Collection des Laufzeitsystems in Orange.

7.3 Semi-explizite Parallelität: Glasgow parallel Haskell

Glasgow parallel Haskell (GpH) wurde bereits 1996 entwickelt und kann durch das Modul `Control.Parallel` geladen werden. Es erweitert Haskell durch zwei Primitive:

```
par :: a -> b -> b
pseq :: a -> b -> b
```

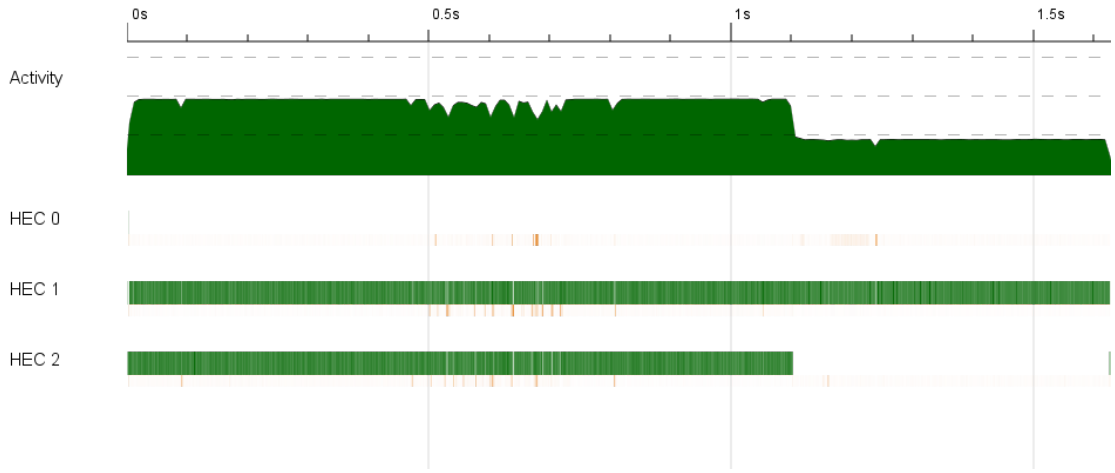


Abbildung 7.1: Ansicht von ThreadScope

Der Ausdruck `x ‘par‘ e` erlaubt die parallele Auswertung von `x` und `e`. Genauer gibt dies dem Laufzeitsystem nur „einen Hinweis“, dass `x` parallel zu `e` ausgewertet werden kann. Die Auswertung von `x` wird nicht erzwungen, sondern es wird ein sogenannter *Spark* für `x` erzeugt. Sparks werden durch das Laufzeitsystem verwaltet und bis zur WHNF ausgewertet, falls ein Prozessor frei ist.

Der Ausdruck `x ‘pseq‘ e` definiert die sequentielle Auswertung von `x` und `e`, wobei `x` zur Weak Head Normal Form ausgewertet. Im Unterschied zu `seq` garantiert der Kompiler bei `pseq` die sequentielle Auswertung, während bei `seq` Optimierungen möglich sind, die eine andere Reihenfolge (z.B. `e` vor `x`) erlauben.

Als Beispiel betrachten wir die rekursive und exponentielle Berechnung der Fibonacci-Zahlen:

```
import Control.Parallel
import System.Environment(getArgs)

fib :: Integer -> Integer
fib n | n < 2     = 1
      | otherwise = fib (n-1) + fib (n-2)

main = do
  (inp1:inp2:_) <- getArgs
  let x = fib (read inp1)
  let y = fib (read inp2)
  let s = x ‘par‘ y ‘pseq‘ x+y
  print s
```

Compilieren und Ausführen:

```
> stack ghc -- fib1.hs -O  -threaded -rtsopts
./fib1 41 41 +RTS -N1 -s  2>&1 | sed -n '/Total/p'
  Total   time   17.953s  ( 17.991s elapsed)
> ./fib1 41 41 +RTS -N2 -s  2>&1 | sed -n '/Total/p'
  Total   time   20.591s  ( 10.341s elapsed)
```

Mit 2 Kernen wird nur noch ungefähr die Hälfte der Zeit benötigt, allerdings ist die Summe der Rechenzeit beider Kerne gestiegen. Dies liegt daran, dass die Verwaltung der Sparks ebenfalls Zeit kostet.

Wenn wir mehr parallelisieren:

```
import Control.Parallel
import System.Environment(getArgs)

pfib :: Integer -> Integer
pfib n | n < 2     = 1
       | otherwise = let fn1 = pfib (n-1)
                         fn2 = pfib (n-2)
          in fn1 `par` fn2 `pseq` fn1 + fn2

main = do
  (inp1:inp2:_) <- getArgs
  let x = pfib (read inp1)
  let y = pfib (read inp2)
  let s = x `par` y `pseq` x+y
  print s
```

zeigt die Ausführung:

```
> ./fib2 41 41 +RTS -N1 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 535828591 (0 converted, 234141903 overflowed, 0 dud, 301642185 GC'd, 44503 fizzled)
  Total   time   21.857s  ( 21.921s elapsed)
>  ./fib2 41 41 +RTS -N2 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 535893347 (48 converted, 226501882 overflowed, 0 dud, 309088071 GC'd, 303346 fizzled)
  Total   time   26.268s  ( 13.191s elapsed)
>  ./fib2 41 41 +RTS -N3 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 535954678 (97 converted, 223541630 overflowed, 0 dud, 311958286 GC'd, 454665 fizzled)
  Total   time   30.573s  ( 10.221s elapsed)
>  ./fib2 41 41 +RTS -N4 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 536058575 (178 converted, 217787349 overflowed, 0 dud, 317527048 GC'd, 744000 fizzled)
  Total   time   33.098s  (  8.301s elapsed)
>  ./fib2 41 41 +RTS -N5 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 536152118 (293 converted, 159965617 overflowed, 0 dud, 375147985 GC'd, 1038223 fizzled)
  Total   time   41.048s  (  8.351s elapsed)
>  ./fib2 41 41 +RTS -N6 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 536205153 (352 converted, 109976066 overflowed, 0 dud, 424988603 GC'd, 1240132 fizzled)
  Total   time   49.862s  (  8.362s elapsed)
>  ./fib2 41 41 +RTS -N7 -s  2>&1 | sed -n '/Total\|SPARKS/p'
  SPARKS: 538082452 (1675 converted, 116996337 overflowed, 0 dud, 417045295 GC'd, 4039145 fizzled)
```

```
  Total   time   61.974s  (  9.061s elapsed)
>
```

Es zeigt sich, dass sich die Laufzeit nur unwesentlich verbessert oder sogar verschlechtert. Der Grund ist, dass zuviel Overhead durch zu viele Sparks entsteht.

Ein *Spark* steht für eine *mögliche* parallele Berechnung. Zur Laufzeit gibt es mehrere Möglichkeiten für einen Spark:

- Konvertiert *(converted)*: Der Spark wurde ausgerechnet.
- Abgeschnitten *(pruned)*: Der Spark wurde nicht ausgerechnet.
  - *Dud*: Der Spark war schon ein ausgerechneter Wert.
  - *Fizzled*: Inzwischen durch anderen Thread berechnet.
  - *Garbage Collected*: Keine Referenz zum Spark vorhanden, d.h. Wert wird gar nicht mehr benötigt.
  - *Overflowed*: Der Spark wurde gleich verworfen, da der Ringpuffer der Sparks momentan voll ist.

Idealerweise sollten deutlich mehr Sparks konvertiert als abgeschnitten werden, was in obigem Beispiel nicht der Fall ist. Das Problem ist eine gute Granularität für die Erzeugung der Sparks zu finden.

Ein Spark ist sehr billig (deutlich einfacher als ein Thread), da der Spark-Pool lediglich ein Ringpuffer von Thunks ist. Es ist akzeptabel mehr Sparks einzuführen als letztendlich ausgeführt werden, da die Anzahl der verfügbaren Kerne ja variieren kann. Dennoch ist es schädlich, wenn zu viele Sparks angelegt werden (z.B. wenn die Ausführung des Sparks schneller geht als das Anlegen des Sparks selbst).

Die Granularität, also die Größe der einzelnen Aufgaben/Threads/Sparks sollte

- nicht zu klein sein, damit sich die parallele Behandlung gegenüber dem erhöhten Verwaltungsaufwand auch lohnt,
- nicht zu groß sein, damit genügend Aufgaben für alle verfügbaren Kerne bereit stehen.

7.3.1 Eval-Monade

Ein weiteres Problem für die parallele Auswertung kann durch die verzögerte Auswertung entstehen:

```
genlist :: Integer -> Integer -> [[Integer]]
genlist _ 0 = []
genlist x n = let f = fib x
                  h = (:) f []
                  t = genlist x (n-1)
              in h `par` t `pseq` h : t
main = do
```

```
  let l = genlist 38 6
  mapM_ print l
```

Die Liste wird zwar parallel zusammengebaut, aber Fibonacci-Berechnungen finden erst sequentiell bei bei der Bildschirmausgabe statt, da die parallele Auswertung nur bis zur WHNF erfolgte.

Ein echten Speedup erhalten wir, wenn wir die Auswertung von `f` statt `h` parallelisieren:

```
genlist :: Integer -> Integer -> [[Integer]]
genlist _ 0 = []
genlist x n = let f = fib x
                  h = (:) f []
                  t = genlist x (n-1)
              in f `par` t `pseq` h : t  -- parallel
main = do
  let l = genlist 38 6
  mapM_ print l
```

Allgemein wird aufgrund der verzögerten Auswertung schnell unklar, wann der wesentliche Teil einer Berechnung ausgeführt wird. Dies beeinflusst letztendlich die Granularität. In einer Sprache mit verzögerter Auswertung muss sich der Programmierer daher explizit um den *Auswertegrad* und die *Auswertungsreihenfolge* der Daten kümmern, um die Granularität korrekt abschätzen zu können. Daher sind die Primitive `par` und `pseq` in der Praxis schlecht brauchbar.

Eine Lösungsmöglichkeit bietet das Modul `Control.Parallel.Strategies`. Dort werden *Strategien* zur Koordination der parallelen Berechnung angeboten, um diese von der eigentlichen Berechnung zu trennen. Die Strategie legt Auswertegrad und -reihenfolge fest.

Die Abhängigkeiten verschiedener paralleler Berechnungen werden durch eine Monade explizit ausgedrückt.

Die Komposition verschiedener paralleler Berechnungen erfolgt mit den üblichen monadischen Kombinatoren, z.B. `join :: Monad m => m (m a) -> m a` und `sequence :: Monad m => [m a] -> m [a]`.

Das Modul `Control.Parallel.Strategies` definiert die Monade `Eval` zur Erzeugung von Sparks:

```
runEval :: Eval a -> a
rpar :: a -> Eval a -- kreiert einen Spark
rseq :: a -> Eval a -- wartet auf das Ergebnis
```

Die zusammengesetzten Aktionen der `Eval`-Monade drücken die Abhängigkeiten zwischen parallelen Berechnungen aus. Z.B. erzwingt `m >>= f` die Berechnung von `m` vor `f`. Die Komposition der Aktionen erfolgt durch monadische Kombinatoren. Die Monade ist „rein“, hat also keine Seiteneffekte.

Betrachte z.B. die folgende Implementierung der parallelen `fib`-Berechnung:

```
import Control.Parallel
import Control.Parallel.Strategies

fib :: Integer -> Integer
fib n | n < 2     = 1
      | otherwise = fib (n-1) + fib (n-2)

main = do
  let myfib = fib
  let s = runEval $ do
            x <- rpar $ myfib 40
            y <- rseq $ myfib 38
            return $ (x+y)
  print s
```

Auch die oben gezeigte Listenerzeugung kann parallelisiert werden:

```
import Control.Parallel
import Control.Parallel.Strategies

fib :: Integer -> Integer
fib n | n < 2     = 1
      | otherwise = fib (n-1) + fib (n-2)

genlist :: Integer -> Integer -> Eval [[Integer]]
genlist _ 0 = return []
genlist x n = do f <- rpar $ fib x
                 let h = [f]
                 t <- genlist x (n-1)
                 return $ h : t
main = do
  let l = runEval $ genlist 38 6
  mapM_ print l
```

Die Implementierung der `Eval`-Monade ist sehr einfach gehalten:

```
data Eval a = Done a

instance Monad Eval where
  -- return :: a -> Eval a
  return x = Done x
  -- (>>=) :: Eval a -> (a -> Eval b) -> Eval b
  Done x >>= k = k x
```

```
runEval :: Eval a -> a
runEval (Done a) = a

rpar :: a -> Eval a
rpar x = x `par` return x

rseq :: a -> Eval a
rseq x = x `pseq` return x
```

7.3.2 Auswertungsstrategien

Folgende Typabkürzung erlaubt im Zusammenhang mit der Monade eine schöne Vereinfachung:

```
type Strategy a = a -> Eval a
```

Anwendung einer Auswertungsstrategie erfolgt mit `using`:

```
using :: a -> Strategy a -> a
using x  s = runEval (s x)
```

Z.B. kann damit ein sequentieller Haskell-Ausdruck einfach parallelisiert werden, indem man statt

```
foo1 x y = someexpr
```

den Code

```
foo1 x y = someexpr `using` someParallelStrategy
```

schreibt.

Einfache Strategien zur Kontrolle von Auswertegrad, Auswertungsreihenfolge und Parallelität sind:

- `r0` führt keine Auswertung durch
- `rpar` führt die Auswertung parallel durch
- `rseq` führt eine Auswertung zur WHNF durch (default)
- `rdeepseq` führt eine komplette Auswertung durch

Die Strategien sind definiert als

```
r0 :: Strategy a
r0 x = Done x

rpar :: Strategy a
rpar x = x `par` Done x

rseq :: Strategy a
rseq x = x `pseq` Done x
```

Den Code für `rdeepseq` zeigen wir gleich. Wir können auch eigene Strategien definieren, z.B. für die Auswertung eines Paares:

```
evalPair :: Strategy a -> Strategy b -> Strategy (a,b)
evalPair sa sb (a,b) = do a' <- sa a
                          b' <- sb b
                          return (a',b')

parEvalPair :: Strategy (a,b)
parEvalPair = evalPair rpar rpar

main = do
  (inp1:inp2:_) <- getArgs
  let x = fib (read inp1)
  let y = fib (read inp2)
  let p = (x, y) `using` parEvalPair
  print $ fst p + snd p  -- p muss verwendet werden!
```

Beachte, dass `evalPair` die Strategien zur Auswertung des ersten und des zweiten Paares erhält. Alternativ können wir für evalPair auch die `Applicative`-Instanz verwenden:

```
evalTuple2 :: Strategy a -> Strategy b -> Strategy (a,b)
evalTuple2 sa sb (a,b) = (,) <$> sa a <*> sb b
```

Auswertungsstrategien können auch kombiniert werden:

Die *Komposition* zweier Strategien kann definiert werden durch:

```
dot :: Strategy a -> Strategy a -> Strategy a
s2 `dot` s1 = s2 . runEval . s1
```

Wir betrachten ein Beispiel zur Verdeutlichung der Strategien und der Komposition. Die Funktionen `evalList2` und `evalList3` werten das zweite bzw. dritte Element einer Liste aus und geben anschließend die Liste zurück:

```
evalList2 (x:y:ys) = do
                     y' <- rseq y
                     return (x:y':ys)
evalList2 xs = return xs

evalList3 (x:y:z:zs) = do
                       z' <- rseq z
                       return (x:y:z':zs)
evalList3 xs = return xs
```

Die folgenden Aufrufe zeigen die Auswertung:

```
*Main> let xs = [(1+1),(2+2),(3+3),(4+4)]::[Int]
*Main> :sprint xs
xs = [_,_,_,_]
*Main> take 1 xs
[2]
*Main> :sprint xs
xs = [2,_,_,_]
*Main> take 1 (xs `using` evalList3)
[2]
*Main> :sprint xs
xs = [2,_,6,_]
*Main> take 1 (xs `using` evalList2)
[2]
*Main> :sprint xs
xs = [2,4,6,_]
*Main> let ys = [(1+1),(2+2),(3+3),(4+4)]::[Int]
*Main> take 1 (xs `using` evalList2 `dot` evalList3)
[2]
*Main> :sprint xs
xs = [2,4,6,_]
*Main>
```

Weitere Strategien können selbst programmiert werden, z.B. die parallele Anwendung auf Paare:

```
parPair' :: Strategy a -> Strategy b -> Strategy (a,b)
parPair' strat1 strat2 =
  evalPair (rpar `dot` strat1) (rpar `dot` strat2)
```

Die sequentielle Anwendung auf Listen, wendet eine gegebene Strategie auf alle Listenelemente an

```
evalList :: Strategy a -> Strategy [a]
     --  :: (a -> Eval a) -> ([a] -> Eval [a])
```

```
evalList s [] = return []
evalList s (x:xs) = (:) <$> s x <*> evalList s xs
```

Betrachte z.B.

```
*Main> let ys = [(1+1),(2+2),(3+3),(4+4)]::[Int]
*Main> length (ys `using` (evalList r0))
4
*Main> :sprint ys
ys = [_,_,_,_]
*Main> length (ys `using` (evalList rseq))
4
*Main> :sprint ys
ys = [2,4,6,8]
```

Wir können die Listenelemente auch parallel auswerten:

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

Ein Beispielaufruf dazu ist:

```
main = do
  (inp1:_) <- getArgs
  let ys = [fib (read inp1),fib (read inp1)]
  let ys' = ys `using` (parList rseq)
  print ys'
```

In `ys'` werden alle Listenelemente parallel bis zur WHNF ausgewertet.

Wir können damit ein paralleles `map` definieren:

```
parMap :: (a -> b) -> [a] -> [b]
parMap f xs = map f xs `using` parList rseq
```

7.3.3 NFData

Die Strategien `r0`, `rseq` und `rdeepseq` regulieren den Auswertegrad eines Ausdrucks. Die Strategie `rdeepseq` wertet vollständig zur Normal-Form (NF) aus; was mithilfe der Klasse `NFData` erreicht wird:

```
class NFData a where
  rnf :: a -> ()
  rnf x = x `seq` ()
```

Für Basistypen reicht die Default-Implementierung. Weitere Instanzen definiert das Modul `Control.Parallel.Strategies` Eigene Instanzen sind leicht zu definieren:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
instance NFData a => NFData (Tree a) where
    rnf Leaf = ()
    rnf (Node l a r) = rnf l `seq` rnf a `seq` rnf r
```

Eine generische `NFData`-Instanz für Listen ist:

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs
```

Während `seq` das erste Argument zu WHNF auswertet, wertet `deepseq` sein erstes Argument zur Normalform, also vollständig, aus:

```
deepseq :: NFData a => a -> b -> b
deepseq a b = rnf a `seq` b
```

Die `rdeepseq`-Auswertungsstrategie wird oft verwendet:

```
rdeepseq :: NFData a => Strategy a
rdeepseq x = x `deepseq` Done x
```

Ein Beispiel im GHCi zeigt die tiefe Auswertung;

```
*Main> let ys = [replicate 2 (1+1), replicate 3 (2*2), replicate 4 (3-3)]::[[Int]]
*Main> length (ys `using` rdeepseq)
3
*Main> :sprint ys
ys = [[2,2],[4,4,4],[0,0,0,0]]
```

7.3.4 Zusammenfassung zu GpH

Das zu GpH zugehörige Paket ist `parallel`. Es erlaubt die parallele Auswertung von Teilausdrücken, d.h. von Thunks. Dabei wird die Parallelität vom Laufzeitsystem verwaltet. Daher ist der Overhead geringer als in anderen Ansätzen. Der Programmierer entscheidet über die Auswertungsreihenfolge und der Auswertegrad. Komponierbare Auswertungsstrategien erfassen Auswertungsreihenfolge und Auswertegrad. Das schöne an der Bibliothek ist, dass sequentieller funktionaler Code mit wenigen Änderungen parallelisierbar ist.

Die Korrektheit des Code bleibt unverändert, sofern ein Wert geliefert wird, aber bei Verwendung von zu strikten Strategien kann sich das Terminierungsverhalten ändern, was zu falschen Programmen führen kann.

7.4 Par-Monade

Eine Alternative zur GpH ist die Verwendung der `Par`-Monade, die durch das Paket `monad-par` definiert wird.

Die Vorteile dieses Ansatzes ist, dass das Nachdenken über die verzögerte Auswertung entfällt, die Monade sich um das globale Scheduling kümmert und die Berechnung deterministisch bleibt (es kommt stets der selbe Wert heraus). Die Nachteile des Ansatzes sind, dass ein deutlich teurerer Overhead im Vergleich zu GpH-Sparks in Kauf genommen werden muss. Der Ansatz erlaubt Parallelität, aber keine Nebenläufigkeit, aber es können Deadlocks auftreten. Schließlich darf innerhalb der `Par`-Aktionen kein IO geschehen (sonst wäre der Determinismus verletzt). Es gibt eine nichtdeterministische Variante `Control.Monad.Par.IO`, welche die parallele Ausführung in der IO-Monade ausführt und daher auch IO erlaubt. Diese Berechnungen sind nichtdeterministisch. Wir beschränken uns im folgenden Abschnitt auf die deterministische Variante `Control.Monad.Par`

7.4.1 Explizite Synchronisation

Die Synchronisation erfolgt explizit durch den Programmierer, indem er sogenannte `IVar`-Referenzen benutzt. Diese sind durch die folgenden Operationen verfügbar:

```
new     :: Par (IVar a)
```

erzeugt eine Referenz auf einen leerem Speicherplatz eines konkreten Typs

```
put     :: NFData a => IVar a -> a -> Par ()
```

beschreibt den Speicherplatz. Dies kann nur einmal ausgeführt werden. Ein zweiter Schreibversuch führt zu einer Ausnahme!

```
get     :: IVar a -> Par a
```

liest den Speicherplatz bzw. wartet, bis ein Wert vorliegt,

Durch zyklisches Warten können dabei Deadlocks auftreten. Zudem dürfen `IVar`-Referenzen keinesfalls zwischen verschiedenen `Par`-Monaden herumgereicht werden.

Eine grobe Struktur zur Verwendung der `IVar`s in der `Par`-Monade ist:

```
example :: a -> Par (b,c)
example x = do
    vb <- new  -- vb :: IVar b
    vc <- new  -- vc :: IVar c
    --  Parallele Berechnungen werden nun gestartet
    --  und befüllen vb and vc mit Aufruf an put
```

```
    rb <- get vb    -- rb :: b
    rc <- get vc    -- rc :: c
    return (rb,rc)
```

Die `new`-Befehle erzeugen leere Speicherplätze, deren Namen `vb` und `vc` an die parallelen Berechnungen übergeben werden.

Innerhalb der parallelen Berechnungen erfolgen die Aufrufe `put vb somevalue` und `put vc othervalue`, welche die leeren Speicherplätze unabhängig voneinander füllen.

Die `get`-Befehle *synchronisieren*, denn mit dem Auslesen wird gewartet, bis der Speicherplatz gefüllt ist.

7.4.2 Fork

Mit `fork :: Par () -> Par ()` wird eine übergebene Berechnung parallel zum aktuellen Thread gestartet. Der Typ `Par ()` besagt, dass parallele Berechnungen kein Ergebnis haben. Also müssen Ergebnisse als *Seiteneffekte* in der `Par`-Monade übergeben werden – durch Beschreiben von `IVar`-Variablen, der einzige erlaubte Seiteneffekt in der `Par`-Monade.

Ein Beispiel zur Verwendung von `fork` ist:

```
example :: a -> Par (b,c)
example x = do
    vb <- new                   -- vb :: IVar b
    vc <- new                   -- vc :: IVar c
    fork $ put vb $ taskB x     -- taskB :: a -> b
    fork $ put vc $ taskC x     -- taskC :: a -> c
    rb <- get vb                -- rb == taskB x
    rc <- get vc                -- rc == taskC x
    return (rb,rc)
```

Da nächste Beispiel:

```
example2 :: Par d
example2 = do
  va <- new; vb <- new; vc <- new; vd <- new
  fork $ put va taskA                          -- A
  fork $ do ra <- get va; put vb $ taskB ra    -- B
  fork $ do ra <- get va; put vc $ taskC ra    -- C
  fork $ do rb <- get vb
            rc <- get vc
            put vd $ taskD rb rc -- D
  get vd
```

verwendet implizite Abhängigkeiten: Sowohl `taskB` als auch `taskC` benötigen das Ergebnis von `taskA`, und `taskD` benötigt die Ergebnisse von `taskB` und `taskC`. Die Reihenfolge der `fork`-Anweisungen ist dabei im Grunde irrelvant, da die Abhängigkeiten bereits implizit durch Lesen und Schreiben der `IVar`-Variablen ausgedrückt werden, d.h. wir können genauso schreiben:

```
example2 :: Par d
example2 = do
  va <- new; vb <- new; vc <- new; vd <- new
  fork $ do ra <- get va; put vc $ taskC ra     -- C
  fork $ do ra <- get va; put vb $ taskB ra     -- B
  fork $ do rb <- get vb                        -- D
            rc <- get vc
            put vd $ taskD rb rc
  fork $ put va taskA    -- A
  get vd
```

Das folgende Beispiel zeigt wie zyklische Abhängigkeit zu einer Verklemmung (Deadlock) führen:

```
deadlockExample :: Par (b,c)
deadlockExample = do
  vb <- new
  vc <- new
  fork $ do rc <- get vc; put vb $ task1 rc
  fork $ do rb <- get vb; put vc $ task2 rb
  get vb
  get vc
  return (vb,vc)
```

Hier wird `task1` erst ausgeführt, wenn `vc` gefüllt ist und `task2` wird erst ausgeführt, wenn `vb` gefüllt ist. Beide Threads warten also zuerst darauf, dass der andere fertig wird.

Die `Par`-Monade unterstützt die Operationen:

```
runPar   :: Par a  ->    a
runParIO :: Par a  -> IO a

fork     :: Par () -> Par ()

new      :: Par (IVar a)
get      :: IVar a -> Par a
put      :: NFData a => IVar a -> a -> Par ()
put_     :: IVar a -> a -> Par ()
```

Dabei führt `runPar` die Monade aus, `fork` startet eine parallele Auswertung, `new` erzeugt eine `IVar`, `put` erzwingt die volle Auswertung seines Argumentes und füllt die `IVar`, `put_` wertet nur bis zur WHNF aus und füllt die `IVar`, und `get` wartet bis der Wert verfügbar ist.

Die Operation `spawn :: NFData a => Par a -> Par (IVar a)` ist ähnlich zu `fork`, aber sie liefert eine `IVar` zurück, über die auf das Ergebnis der parallelen Berechnung zurückgegriffen werden kann. Die Implementierung ist einfach:

```
spawn p = do
   r <- new
   fork (p >>= put r)
   return r
```

Z.B. kann ein paralleles `map` damit definiert werden als:

```
parMap :: NFData b => (a ->     b) -> [a] -> Par [b]
parMap f xs = do ibs <- mapM (spawn . return . f) xs
                 mapM get ibs
```

7.4.3 Zusammenfassung Par-Monade

Die `Par`-Monade wird ausschließlich zur Beschleunigung der Berechnung durch paralleles Auswerten verwendet. In der vorgestellten Variante ist die Berechnung mit der `Par`-Monade deterministisch, d.h. sie liefert immer das gleiche Resultat. IO-Operationen sind innerhalb `Par`-Monade nicht erlaubt. Im Vergleich zu GpH ist der interne Verwaltungsaufwand größer, daher sollten die parallel ausgeführten Berechnungseinheiten alle wesentlich größere Berechnungen durchführen und benötigt werden, denn parallele Einheiten werden immer vollständig ausgewertet. Die `Par`-Monade kann jederzeit verwendet werden, ein Durchschleifen des Monaden-Typ ist nicht notwendig. Parallelität wird nur innerhalb eines `runPar` verwendet, mehrere `runPar` untereinander werden immer sequentiell ausgewertet.

7.5 Ausnahmen

Manchmal können Berechnungen fehlschlagen oder sind undurchführbar. Wir haben bereits gesehen, wie wir solche Fälle mithilfe der Typen `Maybe a` und `Either a b` behandeln können, z.B.

```
safeHead :: [a] -> Maybe a
safeHead   []  = Nothing
safeHead (h:_) = Just h
```

Prinzipiell ist dies die beste Methode, um mit Ausnahmen umzugehen, denn man erkennt direkt am Typ einer Funktion, dass Ausnahmen eintreten können! Nachteilig bei diesem Vorgehen ist

jedoch, dass einerseits die Behandlung einer Ausnahme manchmal nur an einer ganz anderen Stelle im Programm durchgeführt werden kann und dass der gesamte Code auf den `Maybe`-Typ umgestellt werden muss.

Eine Abhilfe haben wir bereits gesehen indem wir `Maybe` und `Either a` als Monaden auffassten, wodurch das Herumreichen eines Fehlers implizit wird.

Mit einer speziell zugeschnittenen Monade kann man so auch verschiedene mögliche Ausnahmen beschreiben und behandeln:

```
data Ausnahme = Bauchschmerzen | Kopfweh | Zahnschmerzen Int
type Gesundheit a = Either Ausnahme a   --besser newtype verwenden
-- instance Monad Gesundheit            --hier automatisch über Either

catch :: Gesundheit a -> (Ausnahme -> Gesundheit a) -> Gesundheit a
catch tat abhilfe = case tat of
  (Left ausnahme) -> abhilfe ausnahme
  anderes         -> anderes

hilfe Kopfweh = Right nimmAspirin -- Fehlerbehandlung
hilfe anderes = Left  anderes -- andere Leiden nicht abgefangen
```

Ein Nachteil ist, dass die Auswertung dadurch manchmal strikter und sequentieller als ohne Monade ist.

Wenn wir nach diesem Schema verfahren, dann kennzeichnet die Monade also alle möglichen Ausnahmen von vornherein. Dennoch gibt es in Haskell auch noch implizit auftretende Ausnahmen, welche wir leider nicht am Typ erkennen können:

```
> head []
*** Exception: Prelude.head: empty list
> undefined
*** Exception: Prelude.undefined
> error "Pfui Deife!"
*** Exception: Pfui Deife!
```

Insbesondere zeigt der Typ der `error` Funktion dies nicht an:

```
> :t error
error :: String -> a
```

Der Vorteil ist, dass `error` überall verwendet werden kann.

7.5.1 Ausnahmen im GHC

Solche Ausnahmen sind durch die Klasse `Exception` aus dem Modul `Control.Exception` erfasst:

```
class (Typeable e, Show e) => Exception e where ...
  toException   :: Exception e -> SomeException
  fromException :: SomeException -> Maybe e
```

Die eingebaute Funktion `throw` erlaubt es, eine Ausnahme mit einer beliebigen Instanz der Typklasse `Exception` an einer beliebigen Stelle im Code zu werfen:

```
  throw   :: Exception e => e ->    a
  throwIO :: Exception e => e -> IO a
```

Alle Instanzen der Typklasse `Exception` bilden eine erweiterbare Hierarchie, an dessen Wurzel der Typ `SomeException` steht.

Der Typ `SomeException` ist definiert als

```
data SomeException = forall e. Exception e => SomeException e
```

Der `forall`-Quantor ist eigentlich eine existentielle Quantifizierung, der fordert, dass in `SomeException` s, der Ausdruck *s* von einem Typ sein muss, der Instanz der Klasse `Exception` ist. Durch den Quantor wird der Typ aber nicht nach außen gegeben.

7.5.2 Eigene Ausnahmen definieren

```
  import Data.Typeable
  import Control.Exception

  data Ausnahme = Bauchschmerzen | Kopfweh
                | Zahnschmerzen Int
    deriving (Typeable, Show)

  instance Exception Ausnahme
    -- Defaults reichen aus, also kein 'where'
```

Diese Deklarationen erweitern die Hierarchie der Ausnahmen mit Datentypen zur Beschreibung einer speziellen Art von Ausnahmen.

```
> throw (Zahnschmerzen 3)
*** Exception: Zahnschmerzen 3
```

Auch die Standarbibliothek definiert Ausnahmen nach diesem Muster:

```
newtype ErrorCall = ErrorCall String
    deriving (Typeable)

instance Show ErrorCall where
  showsPrec _ (ErrorCall err) = showString err

instance Exception ErrorCall -- default

error :: String -> a
error s = throw (ErrorCall s)
```

Ausnutzen können wir die Hierarchie bei der Ausnahmebehandlung. Ausnahmen können nur innerhalb der IO-Monade behandelt werden:

```
module Control.Exception where
  catch  :: Exception e => IO a -> (e -> IO a) -> IO a
  handle :: Exception e => (e -> IO a) -> IO a -> IO a
  handle = flip catch
```

Der *Typ* legt fest, welche Art von Exceptions gefangen werden:

```
*> catch (throw Kopfweh) (\e -> print (e :: Ausnahme))
Kopfweh
*> catch (throw $ ErrorCall "Bad") (\e -> print (e :: Ausnahme))
*** Exception: Bad
*> catch (throw $ ErrorCall "Bad") (\e -> print (e :: SomeException))
Bad
*> catch (throw Kopfweh) (\e -> print (e :: SomeException))
Kopfweh
```

Es ist nicht ratsam, blind alle möglichen Ausnahmen zu fangen, wie hier in diesem Beispiel:

```
handle (\e -> print (e :: SomeException)) (...)
```

Man sollte nur Ausnahmen fangen, welche man sinnvoll behandeln kann. Jede ungefangene Ausnahme ist ein Fehler. Ein Fehler führt immer zum Abbruch des Programms, d.h. wir trennen zwischen Exceptions und Errors:

- Exception: Ausnahme, welche speziell behandelt werden muss
- Error: Programmierfehler, d.h. der Programmierer hat nicht alle Fälle korrekt durchdacht, auch Endlosschleifen, etc.

Soll z.B. eine Datei geöffnet werden, doch die Datei existiert nicht, dann kann diese Ausnahme durch Programmierer abgefangen worden sein (Exception), oder der Fall wurde (un-)bewusst vergessen (Error).

Wesentliche Funktionen zur Ausnahmebehandlung sind:

```
catch  :: Exception e => IO a -> (e -> IO a) -> IO a
-- Zur regulären Ausnahmebehandlung:
try :: Exception e => IO a -> IO (Either e a)
-- Zum Aufräumen im Fehlerfall:
onException :: IO a -> IO b -> IO a
finally     :: IO a -> IO b -> IO a
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

- `try` eignet zur gewöhnliche Ausnahmebehandlung besser als `catch`, da die Ausnahme zum greifbaren Datum wird
- `onException` führt bei Ausnahme noch Aufräumaktion aus
- `finally` führt die Aufräumaktion immer aus
- `bracket open close work` führt `close` immer aus

`onException`, `finally`, `bracket` reichen eine eventuelle Ausnahme immer nach außen weiter.

7.5.3 Zusammenfassung

Ausnahmen können überall geworfen werden, aber Ausnahmen können nur in der IO-Monade abgefangen werden. Ausnahme-Typen bilden eine Hierarchie, welche um benutzerdefinierbare Ausnahmen erweitert werden können. Man sollte Ausnahmen nur gezielt abfangen um konkrete Probleme zu beheben und notwendige Aufräumaktionen durchzuführen. Ausnahme-Behandlung sollte man eher sparsam einsetzen, da dies schnell zu undurchsichtigem Code führt, besser `Maybe` oder `Either` oder ähnliche Typen verwenden

7.6 Nebenläufigkeit

Zur nebenläufigen Programmierung betrachten wir zwei Ansätze in Haskell: Concurrent Haskell und STM Haskell.

7.6.1 Concurrent Haskell

Explizite Parallelität bietet das Modul `Control.Concurrent` mit Bibliotheksfunktionen zur Erzeugung und Kontrolle von nebenläufigen Berechnungen, sogenannten IO-Threads in der IO-Monade.

Ein IO-Thread wird nebenläufig zum Hauptthread abgearbeitet. Auch wenn nur ein Prozessorkern verfügbar ist, soll die Illusion entstehen, als würden beide Threads gleichzeitig agieren. IO-Threads sind explizite Objekte, die im Code kontrolliert werden.

```
myThreadId :: IO ThreadId
forkIO     :: IO () -> IO ThreadId
```

Die Primitive `forkIO` erzeugt einen Thread, der über eine `ThreadId` identifiziert werden kann.

Betrachte als einfaches Beispiel das folgende Programm:

```
import Control.Concurrent
import Control.Monad
import System.IO

main = do hSetBuffering stdout NoBuffering
          forkIO (replicateM_ 2000 (putChar 'A'))
          replicateM_ 2000 (putChar 'B')
```

Im Beispiel wird zuerst der Ausgabepuffer abgeschaltet, damit man genau sieht, welcher Thread wann agiert. Der eine Thread schreibt 2000 mal `A`, der andere völlig unabhängig davon 2000 mal `B`. Die Verzahnung der Ausführung ist nichtdeterministisch, GHC bemüht sich jedoch, alle Threads insofern gleichmässig auszführen, dass faires Scheduling verwendet wird (jeder Thread kommt irgenwann einmal dran).

```
BBBBABABABABABABABABABABABABABABABABABAABABABABA...
```

Nebenläufig erzeugte Threads werden beendet, sobald der Haupthread endet, z.B. gibt es keine Garantie, dass im folgenden Programm, alle Ergebnisse berechnet und ausgedruckt werden:

```
import Control.Concurrent

fib :: Integer -> Integer
fib n | n < 2     = 1
      | otherwise = fib (n-1) + fib (n-2)

showFib s n = putStrLn $ s ++ (show $ fib n)

main = do putStrLn "Creating Threads."
          forkIO $ showFib "A" 42
          forkIO $ showFib "B" 38
          forkIO $ showFib "C" 40
          putStrLn "Done."
```

Es fehlt daher noch die Möglichkeit zur Synchronisation von Threads. Concurrent Haskell bietet dafür als Basisprimitive sogenannte MVars an. Eine `MVar a` ist ein veränderlicher Speicherplatz, der leer oder gefüllt sein kann. Der Zugriff erfolgt in der IO-Monade. Grundlegende Operationen sind

```
newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a ->      IO a
putMVar      :: MVar a -> a -> IO ()
```

Dabei gilt:

	MVar leer	MVar besetzt
takeMVar	blockiert	liest Inhalt & leert MVar
putMVar	Setzt MVar	blockiert

Wartende Threads (die auf das Befüllen oder Leeren einer MVar warten) werden dabei durch eine FIFO-Warteschlange an der MVar aufgereit, wodurch Fairness garantiert wird.

Obiges Synchronisationsproblem kann mit MVars gelöst werden:

```
import Control.Concurrent

fib :: Integer -> Integer
fib n | n < 2     = 1
      | otherwise = fib (n-1) + fib (n-2)

showFib s n = putStrLn $ s ++ (show $ fib n)

main = do putStrLn "Creating Threads."
          syncA <- newEmptyMVar
          syncB <- newEmptyMVar
          syncC <- newEmptyMVar
          forkIO $ showFib "A" 42 >> putMVar syncA ()
          forkIO $ showFib "B" 38 >> putMVar syncB ()
          forkIO $ showFib "C" 40 >> putMVar syncC ()
          mapM_ takeMVar [syncA, syncB, syncC]
          putStrLn "Done."
```

Mit MVars kann z.B. der exklusive Zugriff programmiert werden, indem der entsprechende Bereich durch `takeMVar` und `putMVar` geschützt wird. Z.B. kann damit ein atomarer Zähler implementiert werden:

```
type Counter = MVar Integer
newCounter :: IO (Counter)
```

```
newCounter i = newMVar i

increaseCounter counter = do
 r <- takeMVar counter
 putMVar counter (r+1)
```

Verwendet man MVars falsch, so kann man Deadlocks programmieren, z.B

```
import Control.Concurrent

main = do
   a <- newEmptyMVar
   b <- newEmptyMVar
   forkIO (takeMVar a >>= putMVar b)
   takeMVar b >>= putMVar a
```

Die Ausführung führt in diesem Fall zum Fehler:

```
thread blocked indefinitely in an MVar operation
```

Eine MVar kann z.B. auch zum nichtdeterministischen Mischen zweier Listen verwendet werden (in Erweiterung des Beispiels kann man damit auch sogenannte Erzeuger-Verbraucher-Probleme lösen). Die beiden schreibenden Threads sind dabei die Erzeuger, der lesende Thread der Verbraucher.

```
mergeByMVar :: [a] -> [a] -> IO [a]
mergeByMVar xs ys =
 do
  mvar <- newEmptyMVar
  forkIO (mapM_ (putMVar mvar) xs)
  forkIO (mapM_ (putMVar mvar) ys)
  (mapM (\_ -> takeMVar mvar) (xs++ys))
```

Es kann nachteilig sein, dass nur ein Platz zu Verfügung steht, wenn die Erzeuger z.B. unterschiedlich schnell sind. Für MVars gilt, dass diese im Gegensatz zu IVars mehrfach beschrieben werden können und dass `putMVar` nicht-strikt im zweiten Argument ist, d.h. Ausdrücke werden nicht ausgewertet, bevor sie in die MVar geschrieben werden. Z.B. parallelisiert folgender Code nicht:

```
main = do i1:i2:_ <- getArgs
          result1 <- newEmptyMVar
          result2 <- newEmptyMVar
          forkIO $ putMVar result1 (fib (read i1))
          forkIO $ putMVar result2 (fib (read i2))
```

```
          r1 <- takeMVar result1
          r2 <- takeMVar result2
          print $ r1 + r2
```

Eine Abhilfe ist es, die Auswertung mit `$!` zu erzwingen:

```
main = do i1:i2:_ <- getArgs
          result1 <- newEmptyMVar
          result2 <- newEmptyMVar
          forkIO $ putMVar result1 $! (fib (read i1))
          forkIO $ putMVar result2 $! (fib (read i2))
          r1 <- takeMVar result1
          r2 <- takeMVar result2
          print $ r1 + r2
```

Da MVars für beliebige Typen deklariert werden können, ist es relativ leicht möglich, Kommunikationskanäle mit unbegrenztem Puffer (FIFO-Warteschlangen) zu erstellen. Diese gibt es aber auch schon fertig in `Control.Concurrent.Chan`:

```
newChan   :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan  :: Chan a -> IO a
```

Schreiben blockiert nie. Ist der Channel leer, so blockiert ein Leseversuch, bis der Channel wieder gefüllt wird.

Neben den Basisprimitiven gibt es die Operationen `readMVar :: MVar a -> IO a` zum Lesen `swapMVar :: MVar a -> a -> IO a` zum Austauschen des Werts und `modifyMVar_:: MVar a -> (a -> IO a) -> IO ()` zum Anwenden einer Funktion. Alle drei Operationen blockieren bei einer leeren MVar. Nichtblockierende Operationen sind `newMVar :: a -> IO (MVar a)` zum Erzeugen einer gefüllten MVar, `isEmptyMVar :: MVar a -> IO Bool` zum Testen, ob eine MVar leer ist, sowie `tryTakeMVar :: MVar a -> IO (Maybe a)` und `tryPutMVar :: MVar a -> a -> IO Bool`.

7.6.2 Asynchrone Ausnahmen

Manchmal ist es notwendig, laufende Threads zu unterbrechen, z.B. wenn ein Benutzer „Abbrechen“ in einem Fensterthread klickt, oder ein Timeout für eine IO-Operationen auftritt, etc.

Eine Möglichkeit dies Umzusetzen ist sogenanntes Polling: Der Thread fragt regelmässig eine MVar ab, ob eine Unterbrechung vorliegt. Dies erfordert jedoch Disziplin und Sorgfalt bei der Programmierung, aber auch erhöhte Last.

Eine andere Möglichkeit sind asynchrone Ausnahmen, diese sind in Concurrent Haskell verfügbar über

```
throwTo :: Exception e => ThreadId -> e -> IO ()
```

Ein Aufruf von `throwTo` wirft eine Exception in einem anderen Thread. Das Modul `Control.Concurrent.Async` stellt eine relative einfache Schnittstelle als Hilfe bereit.

```
data Async a = Async ThreadId (MVar Either SomeException a))

async :: IO a -> IO (Async a)
async action = do m <- newEmptyMVar
                  t <- forkIO (do r <- try action; putMVar m r)
                  return (Async t m)

cancel :: Async a -> IO ()
cancel (Async t _var) = throwTo t ThreadKilled

waitCatch :: Async a -> IO (Either SomeException a)
waitCatch (Async _ var) = readMVar var

wait :: Async a -> IO a
wait a = do r <- waitCatch a
            case r of Left e  -> throwIO e
                      Right a -> return a
```

Die Operation `async` ist ein bequemer Ersatz für `forkIO`. Die `ThreadId` ist im Rückgabewert gespeichert, ebenso wird das Ergebnis oder die Ausnahme des neu eröffneten Threads nach dessen Beendigung in einer frischen `MVar` aufgesammelt.

Die Funktion `cancel` ermöglicht es den zugehörigen Thread abzubrechen, da der Datentyp `Async` die `ThreadId` speichert

Die Funktion `waitCatch` wartet, bis der abgezweigte Async-Thread beendet ist – entweder mit einer Ausnahme oder mit einem Ergebnis.

Die Funktion `wait` wartet auf das Ende und Ergebnis eines Threads. Eine Ausnahme wird durch erneutes Werfen einfach nach aussen weitergereicht.

Ein schönes Beispiel zur Verwendung ist das folgende: Es werden verschiedene Threads gestartet, um Webseiten zu laden, ein weiterer Thread wartet auf die Tastatureingabe `'q'`, und unterbricht ggf. die anderen Threads:

```
timeDownload :: String -> IO ()
timeDownload url = do
  (page, time) <- timeit $ getURL url
```

```
  printf "downloaded: %s (%d bytes, %.2fs)\n"
         url (B.length page) time

main = do
  as <- mapM (async . timeDownload) sites
  forkIO $ do                              -- Thread wartet auf q
     hSetBuffering stdin NoBuffering
     forever $ do c <- getChar
                  when (c == 'q') $ mapM_ cancel as
  rs <- mapM waitCatch as
  printf "%d/%d succeeded\n" (length (rights rs)) (length rs)
```

Ein Problem beim Werfen von asynchronen Ausnahmen, zeigt die folgende Implementierung von `modifyMVar_`:

```
myModifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
myModifyMVar_ m fkt = do
  a <- takeMVar m                                -- <1>
  r <- fkt a `catch` \e -> do putMVar m a        -- <2>
                              throw e            -- <3>
  putMVar m r                                    -- <4>
```

Wird die Funktion durch einen anderen Thread unterbrochen zwischen <1> und <2> oder auch zwischen <2> und <4>, so bleibt die MVar leer. Dadurch wird ein inkonsistenter Zustand erzeugt, der z.B. zu Deadlocks in anderen Threads führen kann, wenn „mv nie leer“ als Invariante angenommen wurde.

Daher werden Primitive benötigt, welche Threads vom Abbrechen abschirmen können. Dies leistet das sogeannte *Masking*. Die Funktion

```
mask :: ((forall a. IO a -> IO a) -> IO b) -> IO b
```

unterdrückt Ausnahmen temporär. Dies kann in `modifyMVar_` wie folgt eingesetzt werden:

```
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
modifyMVar_ m fkt = mask $ \restore -> do
  a <- takeMVar m
  r <- restore (fkt a) `catch` \e -> do putMVar m a
                                        throw e
  putMVar m r
```

Eine maskierte Funktion kann nicht unterbrochen werden. Für die Ausführung von `(fkt a)` wird die Möglichkeit für Unterbrechungen temporär mit `restore` wiederhergestellt. Blockierte Aktionen (z.B. `takeMVar` und `putMVar`) können trotzdem unterbrochen werden!

Masking wird an verschiedenen Stellen eingesetzt, z.B. auch in der bereits erwähnten `bracket` Funktion:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after thing = mask $ \restore -> do
    a <- before
    r <- restore (thing a) `onException` after a
    _ <- after a
    return r
```

Damit wird sichergestellt, dass die Aufräumaktionen in jedem Fall durchgeführt werden.

Das vorgestellte `async` war fehlerhaft gewesen, da im Falle einer Unterbrechung nicht sichergestellt war, dass die MVar gesetzt wird:

- nach dem `forkIO`, aber noch vor dem `try`,
- nach dem `try`, aber vor `putMVar`.

Auch hier hilft maskieren:

```
data Async a = Async ThreadId (MVar Either SomeException a))

async :: IO a -> IO (Async a)
async action = do
  m <- newEmptyMVar
  t <- mask $ \restore -> forkIO (do r <- try (restore action);
                                     putMVar m r)
  return (Async t m)
```

Da `forkIO` immer die aktuelle Maskierung übernimmt, sind nun alle Lücken geschlossen! Da dieses Muster häufiger auftritt, gibt es `forkFinally`:

```
async :: IO a -> IO (Async a)
async action = do
  m <- newEmptyMVar
  t <- forkFinally action (putMVar m)
  return (Async t m)

forkFinally :: IO a -> (Either SomeException a -> IO ())
                                                  -> IO ThreadId
forkFinally action fun =
  mask $ \restore ->
    forkIO (do r <- try (restore action); fun r)
```

7.7 Software Transactional Memory

7.7.1 Grundlagen

Ein Problem bei der Verwendung sperrenbasierter Programmierung wie z.B. MVars es ermöglichen, ist das explizite Setzen und Entfernen der Sperren wie auch die fehlende Kompositionalität der Programmierung.

Betrachte als Beispiel ein durch eine MVar modelliertes Bankkonto mit Operationen zum Abbuchen und zubuchen.

```
newtype Account = Account (MVar Int)

deposit  :: Account -> Int -> IO ()
deposit (Account a) n = modifyMVar_ a (\x -> return $ x+n)

withdraw :: Account -> Int -> IO ()
withdraw a n = deposit a (negate n)
```

Will man diesen Code wiederverwenden, um eine Aktion zum Überweisen zwischen zwei Konten zu programmieren, so muss man die gesamte Synchronisation neu programmieren: Beide Konten sperren, dann ändern, dann entsperren. Trotzdem ist diese Lösung

```
transfer :: Int -> Account -> Account -> IO ()
transfer n (Account from) (Account to) = do
  bal_from <- takeMVar from
  bal_to   <- takeMVar to
  putMVar from (bal_from - n)
  putMVar to   (bal_to   + n)
```

nicht korrekt, da Deadlocks auftreten können bei der verzahnten Ausführung von `transfer n1 acc1 acc2` und `transaction n2 acc2 acc1`.

In der Tat gibt es viele Nachteile bei der sperrenbasierten Programmierung:

- Bei zu wenigen Locks drohen Race Conditions und verletzte Invarianten.
- Bei zu vielen Locks wird das Programm sequentialisiert oder sie verursachen sogar Deadlocks.
- Die Reihenfolge in der Locks gesetzt werden ist unklar bzw. muss global festgelegt werden.
- Die Platzierung der Locks ist oft unklar.
- Die Ausnahmebehandlung kann schwierig sein, da alle Locks in einen konsistenten Zustand gebracht werden müssen.

Eine Abhilfe ist die Idee des Transactional Memory. Dabei greifen verschiedene Threads auf gemeinsamem Speicher durch Transaktionen zu. Eine *Transaktion* ist ein Bündel von Speicherzugriffen eines Threads, welche atomar ausgeführt werden. Zwischenzustände werden für andere Threads unsichtbar.

Transaktionen werden überwacht nebenläufig ausgeführt; bei einem Konflikt von Speicherzugriffen wird die Transaktion abgebrochen, alle bisherigen Effekte rückgängig gemacht und später wiederholt.

Das Rollback abgebrochener Transaktionen ist in der funktionalen Welt recht einfach. Es ist implementiert im Modul `Control.Concurrent.STM`. Die Monade `STM` bündelt Blöcke von Zugriffen auf gemeinsame Variablen (`TVar`) zu Transaktionen zusammen. Wie in jeder Monade können diese Blöcke miteinander kombiniert werden. `STM`-Transaktionen können innerhalb der `IO`-Monade scheinbar atomar und *ohne Blockierung* (lock-free) ausgeführt werden. Probleme mit unpassenden Unterbrechungen sind von vornherein ausgeschlossen. Innerhalb der `STM`-Monade ist kein `liftIO` möglich. Dadurch sind alle Seiteneffekte bekannt und eine Berechnung kann bei Konflikten im Zugriff auf `TVar`s einfach zurückgespult werden.

7.7.2 TVar

Die Schnittstelle zu Software Transactional Memory ist neben der STM-Monade die transaktionale Variable, deren Operationen in der STM-Monade angesiedelt sind:

```
newtype STM a = ... -- Eine Monade innerhalb von IO
data TVar a         -- Speicherzelle für Transaktionen

newTVar    :: a                  -> STM (TVar a)
readTVar   :: TVar a             -> STM a
writeTVar  :: TVar a -> a        -> STM ()
modifyTVar :: TVar a -> (a -> a) -> STM ()
swapTVar   :: TVar a -> a        -> STM a
```

Eine `TVar` ist wie ein Speicherplatz und weder `readTVar` noch `writeTVar` können jemals blockieren. Die Operation `writeTVar` überschreibt immer. Eine `TVar` wird in der Regel in verschiedenen `STM`-Aktionen verwendet. Zugriffe auf `TVar`s sind nur innerhalb der `STM`-Monade möglich. Mehrere `TVar` Zugriffe werden monadisch zu einer Aktion der `STM`-Monade zusammengefasst (hier „Transaktion“ genannt), aber noch nicht ausgeführt.

Sofortiges *atomares* Ausführen einer Transaktion in der `IO`-Monade wird durch `atomically :: STM a -> IO a` durchgeführt. Intern werden Lese- und Schreibzugriffe auf TVars in einem Log festgehalten. Wenn die Transaktion beendet ist, wird das Log geprüft. Bei Konflikten wird ein „Roll-back“ durchgeführt, indem das Log verworfen und die Transaktion neu gestartet wird. Bei keinem Konflikt werden die Schreibzugriffe auf dem echten Speicher ausgeführt (mit vorübergehendem Sperren der TVars).

Betrachte das Beispiel:

```
-- in Thread 1:
atomically $ do
    v <- readTVar acc
    writeTVar acc (v + 1)

-- in Thread 2:
atomically $ do
    v <- readTVar acc
    writeTVar acc (v - 3)
```

Hier kann nichts schief gehen, da sichergestellt wird, dass Lese- und Schreibzugriffe immer komplett oder gar nicht durchgeführt werden. Wird z.B. Thread 1 vor dem Schreiben unterbrochen und Thread 2 verändert die `TVar` zwischenzeitlich, dann fängt Thread 1 mit der atomaren `STM`-Transaktion einfach wieder von vorne an.

Die Bankkonto-Aktion zum Überweisen lässt sich programmieren als:

```
transfer3 :: Int -> TVar Int -> TVar Int -> STM ()
transfer3 n from to = do
  bal_from <- readTVar from
  bal_to   <- readTVar to
  writeTVar from (bal_from - n)
  writeTVar to   (bal_to + n)
```

Falls während einer Unterbrechung die betroffenen `TVar`s verändert wurden, dann wird die gesamte Aktion später wiederholt.

Auch mit STM kann man blockieren. Dies wird durch die Funktion `retry :: STM a` bewerkstelligt: Sie löst ein explizites Roll-back aus. Der Thread wird angehalten, bis sich mindestens eine der bis dahin *gelesenen* `TVar`s verändert hat.

Dies ist hilfreich wenn im Programm erkannt wird, dass eine Transaktion nicht erfolgreich abgeschlossen werden kann.

Z.B. Abheben von einem leeren Konto

```
withdrawP :: TVar Int -> Int -> STM ()
withdrawP acc n = do
  bal <- readTVar acc
  if bal < n
    then retry
    else writeTVar acc (bal - n)
```

Neben der sequentiellen Komposition mit >>= aus der Monade, stellt STM-Haskell die Komposition mit `orElse :: STM a -> STM a -> STM a` als alternative Auswahl zwischen zwei

STM-Transaktionen zur Verfügung: Schlägt die erste Transaktion fehl mit `retry`, so wird die zweite Transaktion ausgeführt. Wenn beide Transaktionen fehlschlagen, schlägt auch die gesamte Transaktion fehl und wird komplett wiederholt. Damit kann man z.B. einen Transfer von zwei Konto-Alternativen bewerkstelligen

```
transEither :: Int -> TVar Int
                   -> TVar Int -> TVar Int ->  STM ()
transEither n from1 from2 to3 = do
    (withdrawP from1 n  `orElse`  withdrawP from2 n)
    deposit acc3 n
```

Das folgende Beispiel zeigt die Implementierung von MVars mithilfe von STM-Haskell:

```
newtype TMVar a = TMVar (TVar (Maybe a))
newEmptyTMVar :: STM (TMVar a)
newEmptyTMVar = TMVar <$> newTVar Nothing

takeTMVar :: TMVar a -> STM a
takeTMVar (TMVar t) = do m <- readTVar t
                         case m of
                           Nothing -> retry -- block
                           Just a  -> do writeTVar t Nothing
                                         return a
putTMVar :: TMVar a -> a -> STM ()
putTMVar (TMVar t) a = do m <- readTVar t
                          case m of
                            Nothing -> writeTVar t (Just a)
                            Just _  -> retry -- block
```

7.7.3 Ausnahmenbehandlung in STM

Ausnahmen brechen eine STM-Aktion einfach ab. Dank Roll-back ist dies immer unproblematisch. Dennoch ist es möglich eine Ausnahmebehandlung durchzuführen:

```
throwSTM :: Exception e => e -> STM a
catchSTM :: Exception e => STM a -> (e -> STM a) -> STM a
```

Diese funktionieren wie in der IO-Monade, mit dem Unterschied, dass auch bei `catchSTM` alle Seiteneffekte des ersten Arguments verworfen werden, bevor der Handler ausgeführt wird.

Mögliche Probleme bei Verwendung der STM-Monade, sind dass die STM-Aktionen sollten nicht zu groß werden, da ggf. alles wiederholt werden muss. Im Gegensatz zu `MVar`-Code gibt es

außerdem kein faires Scheduling: es werden immer alle auf eine `TVar` wartenden Threads geweckt da unbekannt ist, welche Konditionen genau zum Blockieren geführt haben. Kürzere `STM`-Aktionen können deshalb schnell erfolgreich abschließen und dadurch längere `STM`-Aktionen auf gleicher `TVar` zum ewigen Roll-back zwingen

Allgemein ist STM-Code langsamer als `MVar`-Code, z.B. hat `readTVar` Laufzeit linear in der Anzahl der Zugriffe, da bei jedem Zugriff das `TVar`-Transaktionen Log der Aktion geprüft werden muss

7.8 Quellen

Diese Kapitel verwendet Material aus (Jos19) und ist ähnlich aufgebaut. Primär ist zu allen Themen des Kapitels das Buch (Mar13) empfehlenswert. STM Haskell wird ausführlich in (PJ07) beschrieben.

8 Template Haskell

8.1 Einleitung

Template Haskell dient zur Meta-Programmierung von Haskell-Programmcode. Meta-Programmierung bedeutet, dass man keine Quelltext-Programme der Programmiersprache schreibt, sondern Programme, die selbst wieder Quelltext-Programme erzeugen. Ein klassisches Beispiel für Meta-Programmierung ist der C-Präprozessor, mit dem man Quelltext einfügen kann, Makros definieren kann und bedingt Programme compilieren kann, z.B. wird in

```
#define VERSION ...
 ...
#if VERSION >= 3
  print "NEUESTE VERSION"
#else
  print "ALTE VERSION"
#endif
```

je nachdem, welche Version bei `define VERSION` definiert wird, ein Quelltext erzeugt, der das Kommando `print "NEUESTE VERSION"` oder `print "ALTE VERSION"` enthält. Der C-Präprozessor kann auch mit dem GHC verwendet werden (für Haskell-Code). Betrachte z.B. den folgenden Code (in der Datei `TestCPP.hs`):

```
main =
 do
#ifdef NEWVERSION
   print "Neue Version"
#else
   print "Alte Version"
#endif
   print "Fertig!"
```

Die Anweisung `#ifdef` fragt, ob ein Makro definiert ist. Da obiger Quelltext das Makro nicht definiert, wird nach Compilation und Ausführung `"Alte Version"` gedruckt:

```
*> stack exec -- ghc -cpp  TestCPP.hs
*> ./TestCPP
"Alte Version"
"Fertig!"
```

Definiert man das Symbol (dies geht auch mit der Compiler-Option `-D`), so kann man das Verhalten ändern:

```
*> stack exec -- ghc -cpp -DNEWVERSION TestCPP.hs
*> ./TestCPP
"Neue Version"
"Fertig!"
```

Die Möglichkeiten von *Template Haskell* gehen jedoch über diesen Mechanismus hinaus: Denn Template Haskell verwendet Haskell, um Haskell-Code zu erzeugen.

Template Haskell kann als Code-Generator eingesetzt werden und erlaubt es dabei dem Benutzer aus einem Meta-Programm viele verschiedene Haskell-Quellcode-Programme zu erzeugen. Benötigt wird dafür eine algorithmische Beschreibung, wie die Haskell-Programme erzeugt werden. Das Meta-Programm implementiert diesen Algorithmus.

Eine Anwendung von Template Haskell besteht darin, oft wiederholenden Code (d.h. Code, der einem einfach Schema folgt) automatisch zu erzeugen, anstatt ihn immer wieder schreiben zu müssen.

Template Haskell wird z.B. auch verwendet, um automatisch Typklasseninstanzen zu generieren.

Eine anderes Anwendungsgebiet sind sogenannte eingebettete domänenspezifische Sprachen. Template Haskell erlaubt es dabei, Programme einer anderen Programmiersprache (oder Beschreibungsprache, z.B. HTML) in Haskell-Code direkt einzubetten.

Wir werden all diese Möglichkeiten mit kleinen Beispielen betrachten. Wir beginnen damit, wie man die Benutzung von Template Haskell ermöglicht: Template Haskell wird mit dem Pragma `{-# LANGUAGE TemplateHaskell #-}` im Quellcode aktiviert, bzw. mit `:set -XTemplateHaskell` im GHCi angeschaltet.

Wir erläutern noch die Sprechweisen von Meta-Programmen und Objekt-Programmen: Die durch Template Haskell erzeugten Meta-Programme werden zur *Compilezeit* ausgeführt und erzeugen dabei normale Haskell-Programme (manchmal als Objekt-Programme bezeichnet) als Resultat dieser Ausführungen.

8.2 Template Haskell als Code-Generator

Als einführendes Beispiel betrachten wir Erweiterungen der Funktion `curry :: ((a,b) -> c) -> a -> b -> c`. Die Funktion `curry` nimmt eine auf Paaren operierende Funktion und erstellt daraus eine Funktion, welche die beiden Argumente nacheinander nimmt. Die Implementierung in Haskell ist einfach:

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

Wir können analoge Varianten für 3-Tupel, 4-Tupel, . . . und entsprechend mehrstellige Funktionen implementieren:

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
curry3 f x y z = f (x,y,z)

curry4 :: ((a,b,c,d) -> e) -> a -> b -> c -> d -> e
curry4 f w x y z = f (w,x,y,z)
```

Allerdings ist es zeitraubend und auch fehlerbehaftet all diese Varianten per Hand zu implementieren. Eine Funktion `curryN`, die eine auf N-Tupel operierende Funktion erhält und daraus die Funktion erstellt, welche die N Elemente nacheinander empfängt, kann jedoch in Haskell *nicht* definiert werden, da sie nicht typisierbar ist (selbst dann nicht, wenn wir ihr zusätzlich die Zahl N als Parameter übergeben). Einen Ausweg bietet jedoch Template Haskell: Wir können ein Meta-Programm implementieren, welches uns die passende `curryX`-Funktion (für passendes `X`) automatisch erzeugt:

Genauer implementieren wir eine Meta-Funktion `curryN :: Int -> Q Exp`, welche uns für eine Zahl $n \geq 1$, den Quellcode der passenden curry-Funktion (für n-stellige Funktionen) erzeugt:

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad
import Language.Haskell.TH

curryN :: Int -> Q Exp
curryN n = do
  f  <- newName "f"
  xs <- replicateM n (newName "x")
  let args = map VarP (f:xs)
      ntup = TupE (map (Just . VarE) xs)
  return $ LamE args (AppE (VarE f) ntup)
```

Die Meta-Funktion `curryN` n erzeugt im Grunde die Lambda-Abstraktion `\f x1 ... xn -> f (x1,...,xn)`, wobei diese allerdings nicht als ausführbares Haskell-Programm, sondern als Syntaxbaum (vom Typ `Exp`) in der sogenannten `Q`-Monade erzeugt wird.

Die Q-Monade (als Kurzform für „Quotation-Monade“) dient der Codeerzeugung für Template Haskell. Sie kümmert sich um alle Seiteneffekte der Code-Generierung, wie die Generierung frischer Namen. In obigem Code haben wir die Operation `newName :: String -> Q Name` der Q-Monade verwendet, um frische Namen für die Parameter zu erzeugen. Die Q-Monade sichert dabei zu, dass die Namen wirklich frisch sind.

Mit

```
runQ :: Quasi m => Q a -> m a
```

können wir die Q-Monade ablaufen lassen und damit den Ausdruck als Syntaxbaum erzeugen. Instanzen von `Quasi` sind `Q` und `IO` wobei die `IO`-Instanz im Wesentlichen zum Ausdrucken des Ausdrucks (und damit dem Debugging) verwendet wird.
Z.B ergibt

```
*Main> runQ (curryN 2)
LamE [VarP f_0,VarP x_1,VarP x_2] (AppE (VarE f_0) (TupE [Just (VarE x_1),Just (VarE x_2)])
*Main> runQ (curryN 4)
LamE [VarP f_3,VarP x_4,VarP x_5,VarP x_6,VarP x_7]
     (AppE (VarE f_3) (TupE [Just (VarE x_4),Just (VarE x_5),Just (VarE x_6),Just (VarE x_7
```

Die Syntax für die abstrakten Syntaxbäume für Haskell-Code sind im Modul `Language.Haskell.TH` durch zahlreiche Typen definiert. U.a. `Dec` für Deklarationen, `Clause` für Klauseln, `Exp` für Ausdrücke, `Pat` für Patterns, `Lit` für Literale (wie Zahlkonstanten usw.), `Type` für Typen. Ein Ausschnitt ist:

```
data Dec  = FunD Name [Clause]
          | ValD Pat Body [Dec]
          | DataD Cxt Name [TyVarBndr] (Maybe Kind) [Con] [DerivClause]
          | NewtypeD Cxt Name [TyVarBndr] (Maybe Kind) Con [DerivClause]
          | ...

data Clause = Clause [Pat] Body [Dec]

data Body = NormalB Exp
          | GuardedB [(Guard,Exp)]

data Exp = VarE Name
         | LitE Lit
         | ConE Name
         | ParensE Exp
         | LamE [Pat] Exp
         | AppE Exp Exp
         | CaseE Exp [Match]
         | ...

data Pat = VarP Name
          | LitP Lit
          | ConP Name [Pat]
          | ParensP Pat
          | WildP
          | TupP [Pat]
          | List [Pat]
```

```
          | ...

data Lit = IntegerL Integer | CharL Char | StringL String | ...

data Type = VarT Name | ConT Name | ArrowT | TupleT Int | ...
```

Die exakte Definition ist in der Bibliothek nachzulesen und diese wird mit Syntax-Änderungen durch neue GHC-Versionen auch aktualisiert, da sie den gesamten Sprachumfang von Haskell umfasst. Die meisten syntaktischen Konstrukte ergeben sich analog zu den Datentypen, wie wir sie für den Lambda-Kalkül oder Erweiterungen davon gesehen haben. Obwohl es andere Möglichkeiten gibt, abstrakte Syntaxbäume zu erzeugen (wir werden später noch einige sehen), muss man des Öfteren die Dokumentation konsultieren, um den syntaktischen Aufbau nachzuvollziehen.

Für Aktionen der Q-Monade, die Ausdrücke, Deklarationen, Typen oder Patterns liefern, gibt es Typsynonyme, wie `ExpQ`, `DecsQ`, `TypeQ` und `PatQ`, die definiert sind als

```
type ExpQ  = Q Exp
type DecsQ = Q [Dec]
type TypeQ = Q Type
type PatQ  = Q Pat
```

Bisher haben wir Meta-Programme geschrieben und mit `runQ` ausgeführt, allerdings liefert uns diese Ausführung nur einen abstrakten Syntaxbaum des Quellcodes und nicht den Quellcode selbst.

Um Meta-Programme zur Compilezeit auszuführen und echten Haskell-Code (anstelle eines abstrakten Syntaxbaums) zu erzeugen, kann man den splice-Operator `$(...)` verwenden. Dabei wird `...` durch das Meta-Programm ersetzt, z.B. erzeugt `$(curryN 3)` die Abstraktion `\f x1 x2 x3 -> f x1 x2 x3` als Haskell-Programm und wir können z.B. aufrufen

```
*Main> :set -XTemplateHaskell
*Main> let fst3 (a,b,c) = a in $(curryN 3) fst3 1 2 3
1
```

D.h. die erzeugte Funktion kann anstelle von ganz normalem Haskell-Code verwendet werden. Mit dem Compiler-Flag `-ddump-splices` kann man die erzeugte Funktion sogar anzeigen lassen:

```
*Main> :set -ddump-splices
*Main> let fst3 (a,b,c) = a in $(curryN 3) fst3 1 2 3
<interactive>:4:27-34: Splicing expression
    curryN 3
  ======>
    \ f_a59H x_a59I x_a59J x_a59K -> f_a59H (x_a59I, x_a59J, x_a59K)
1
```

Im Allgemeinen kann der splice-Operator für jede Aktion der Q-Monade verwendet werden, sodass die Berechnung in der Monade durchgeführt wird und das erhaltene Programm als Quelltext eingefügt wird. Um Typsicherheit herzustellen, werden die Meta-Programme selbst durch den Typchecker geprüft. Dies muss geschehen, bevor sie verwendet werden. Deshalb darf der splice-Operator für ein Meta-Programm *P* nicht im gleichen Modul verwendet werden, indem *P* definiert wird.

Betrachte z.B. den folgenden *ungültigen* Code

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad
import Language.Haskell.TH

curryN :: Int -> Q Exp
curryN n = do
  f  <- newName "f"
  xs <- replicateM n (newName "x")
  let args = map VarP (f:xs)
      ntup = TupE (map (Just . VarE) xs)
  return $ LamE args (AppE (VarE f) ntup)

main = print $(curryN 3) fst3 1 2 3
fst3 (a,b,c) = a
```

Das Compilieren geht schief und endet mit der Fehlermeldung:

```
  error:
    • GHC stage restriction:
        ‘curryN’ is used in a top-level splice, quasi-quote, or annotation,
        and must be imported, not defined locally
    • In the untyped splice: $(curryN 3)
   |
13 | main = print $(curryN 3) fst3 1 2 3
```

Eventuell möchten wir dem Anwender auch gar nicht die Meta-Programme selbst in die Hand geben, sondern z.B. einfach ein Modul erstellen, welches die ersten zehn curry Funktionen erzeugt: Wir können hierfür eine Meta-Funktion implementieren, welche Deklarationen erzeugt.

```
{-# LANGUAGE TemplateHaskell #-}
module GenCurry where
...
generateCurries :: Int -> DecsQ
generateCurries n = forM [1..n] mkCurryDec
  where mkCurryDec ith = do
```

```
        cury <- curryN ith
        let name = mkName $ "curry" ++ show ith
        return $ FunD name [Clause [] (NormalB cury) []]
```

und diese anschließend in unserem Modul für den Anwender erzeugen:

```
{-# LANGUAGE TemplateHaskell #-}
module Curry where
import GenCurry
$(generateCurries 10)
```

Inspizieren des Moduls `Curry` mit `:browse` im GHCi zeigt, dass die 10 Funktionen vorhanden sind:

```
 *Curry> :browse Curry
curry1 :: (t1 -> t2) -> t1 -> t2
curry2 :: ((a, b) -> t) -> a -> b -> t
curry3 :: ((a, b, c) -> t) -> a -> b -> c -> t
curry4 :: ((a, b, c, d) -> t) -> a -> b -> c -> d -> t
curry5 :: ((a, b, c, d, e) -> t) -> a -> b -> c -> d -> e -> t
curry6 ::
  ((a, b, c, d, e, f) -> t) -> a -> b -> c -> d -> e -> f -> t
curry7 ::
  ((a, b, c, d, e, f, g) -> t)
  -> a -> b -> c -> d -> e -> f -> g -> t
curry8 ::
  ((a, b, c, d, e, f, g, h) -> t)
  -> a -> b -> c -> d -> e -> f -> g -> h -> t
curry9 ::
  ((a, b, c, d, e, f, g, h, i) -> t)
  -> a -> b -> c -> d -> e -> f -> g -> h -> i -> t
curry10 ::
  ((a, b, c, d, e, f, g, h, i, j) -> t)
  -> a -> b -> c -> d -> e -> f -> g -> h -> i -> j -> t
*Curry>
```

Beachte, dass wir `mkName :: String -> Name` verwendet haben, um nun feste und keine frischen Namen für die Namen der Funktionen `curry1`, . . . , `curry10` zu erzeugen!

Die Auswertung von Meta-Programmen zur Compilezeit und das Verwenden des splice-Operators zum Einsetzen des erzeugten Codes in normalen Haskell-Code ist das erste wichtige Konzept von Template Haskell. Die beiden anderen wichtigen Punkte sind die algebraischen Datentypen für Quelltext und die Quotation-Monade Q.

Die in Template Haskell erzeugten Objekte-Programme werden in der Quotation-Monade Q erzeugt. Die Monade wird durch den splice-Operator ausgeführt. Bisher haben wir die Q-Monade nur verwendet, um frische Namen zu erzeugen. Das wesentliche andere Feature, welches uns

die Q-Monade zur Verfügung stellt, ist die sogenannte Reification, die es erlaubt Compilezeit-Information während der Objekt-Code-Erzeugung zu verwenden. Wir erläutern die Reification erst später in Abschnitt 8.3.

D.h. die Kernfunktionalität von Template Haskell ist das Erzeugen von Objekt-Programmen mit `$(...)`, wobei diese durch Ausführung von Q-monadischen Programmen mit den algebraischen Datentypen als Rückgabe erzeugt werden. Das Erzeugen der abstrakten Syntaxbäume mit direkten Mitteln (d.h. Verwendung der oben vorgestellten Datentypen und ihrer Konstruktoren) ist ziemlich aufwendig und schwierig. Händisch erzeugt man oft falsche Syntaxbäume, die zu Typfehlern führen. Als Abhilfe dazu bietet Template Haskell zwei Funktionalitäten:

- Spezielle Funktionen zur Erzeugung der Syntaxbäume, die mehr oder weniger als Ersatz für die direkte Verwendung der Konstruktoren dienen.
- Sogenannte Quotation-Klammern (sogenannte Oxford-Klammern, der Form `[| ... |]`), welche Syntaxbäume anhand von Quellcode erzeugen.

Die Funktionen zur Syntaxerzeugung korrespondieren direkt zu den Konstruktoren der algebraischen Datentypen `Exp`, `Pat`, `Dec` und `Type`, aber sie verstecken zum Teil die monadische Erzeugung. Z.B. kann obiges `generateCurries`-Programm mit den Funktionen zur Syntaxerzeugung etwas lesbarer geschrieben werden als:

```
generateCurries :: Int -> Q [Dec]
generateCurries n = forM [1..n] mkCurryDec
  where mkCurryDec ith = funD name [clause [] (normalB (curryN ith)) []]
          where name = mkName $ "curry" ++ show ith
```

Der Unterschied zwischen den Funktionen und den Konstruktoren ist ihr Typ:

```
FunD    :: Name -> [Clause] -> Dec      funD    :: Name  -> [Q Clause] -> Q Dec
Clause  :: [Pat] -> Body -> Clause      clause  :: [Q Pat] -> Q Body  -> Q Clause
NormalB :: Exp -> Body                  normalB :: Q Exp -> Q Body
```

D.h. die Funktionen sind schon in der Q-Monade angesiedelt und ersparen dem Benutzer daher das explizite Ver-, Aus- und Umpacken in der Monade.

Quotation-Klammern sind eine weitere Möglichkeit, um abstrakte Syntaxbäume zu erzeugen. Dabei werden Objekt-Programme in Syntaxbäume konvertiert! Dies geschieht, indem man die normalen Haskell-Programme mit Oxford-Klammern `[| ... |]` umschließt. Betrachte z.B. die Erzeugung der Identitätsfunktion, einmal auf direktem Weg durch Erstellen des Syntaxbaums und einmal mit den Quotation-Klammern:

```
generateId :: Q Exp
generateId = do
  x <- newName "x"
  lamE [varP x] (varE x)
```

```
generateId' :: Q Exp
generateId' = [| \x -> x |]
```

Beide Varianten erzeugen (bis auf Benennung der gebundenen Variablen) das selbe Programm:

```
*Main> runQ generateId
LamE [VarP x_0] (VarE x_0)
*Main> runQ generateId'
LamE [VarP x_1] (VarE x_1)
```

Quotation-Klammern umschließen regulären Haskell-Code und entnehmen dabei die Fragmente des entsprechenden Objekt-Programms innerhalb der Q-Monade. Es gibt verschiedene Quotation-Klammern je nach Ergebnis-Typ: `[e| ... |]` für Haskell-Ausdrücke, `[p| ... |]` für Patterns, `[d| ... |]` für Deklarationen und `[t| ... |]` für Typen. Die Schreibweise `[| ... |]` ist eine Abkürzung für `[e| ... |]`.
Wir demonstrieren die Verwendung im Interpreter: Dabei wird zunächst mit den Quotation-Klammern eine `ExpQ`-Aktion erzeugt, welche anschließend (zum Anzeigen) mit `runQ` ausgeführt wird.

```
*Main> runQ [| \x -> x |]
LamE [VarP x_1] (VarE x_1)

*Main> runQ [| \x y z-> (x y z) |]
LamE [VarP x_2,VarP y_3,VarP z_4] (AppE (AppE (VarE x_2) (VarE y_3)) (VarE z_4))

*Main> runQ [| case y of { [] -> True; _:_ -> False } |]
CaseE
 (UnboundVarE y)
   [Match (ConP GHC.Types.[] []) (NormalB (ConE GHC.Types.True)) [],
    Match (InfixP WildP GHC.Types.: WildP) (NormalB (ConE GHC.Types.False)) []]
```

Da die Quotation-Klammern Haskell-Objekt-Code in die Q-Monade hieven, sind sie dual zum splice-Operator `$` zu sehen, welcher Q-Aktionen in Haskell-Objekt-Code konvertiert (durch Auswertung der Meta-Programme). Entsprechend können Quotation-Klammern als Umkehrung des splice-Operators gesehen werden. Es gilt `$([| e |])` = `e` für alle Ausdrücke `e` (und analoges für gilt für Deklaration und Typen).
Als Zusatz für Identifier kann man auf die internen Namen (repräsentiert durch den Typ `name`) für Identifier mit einer Spezialsyntax zugreifen. Die Syntax ist 'Identifier und z.B. erhält man mit `'generateId` einen entsprechenden Namen für den generateId-Identifier:

```
*Main> :t  'generateId
'generateId :: Name
```

Auch auf in der Prelude definierte Funktionsnamen kann so zugegriffen werden, z.B. mittels `'map`. Dieser Zugriff ist zwar äquivalent zu `mkName "map"`, aber diesem vorzuziehen, da durch die Referenzierung mittel Apostroph geprüft wird, dass der Name existiert.
Mit `''Identifier` kann auf Namen der Typen zugegriffen werden. Einge Beispiele dazu sind:

```
*Main> (ConE 'True)::Exp
ConE GHC.Types.True

*Main> (AppE (AppE (ConE '(:)) (ConE 'True)) (ConE '[]) )::Exp
AppE (AppE (ConE GHC.Types.:) (ConE GHC.Types.True)) (ConE GHC.Types.[])

*Main> (AppE (VarE 'map) (VarE 'even))
AppE (VarE GHC.Base.map) (VarE GHC.Real.even)

*Main> (ConT ''Bool)
ConT GHC.Types.Bool

*Main> AppT (ConT ''[]) (ConT ''Bool)
AppT (ConT GHC.Types.[]) (ConT GHC.Types.Bool)
```

Oxford-Klammern und der Splice-Operator können auch beliebig vermischt und verschachtelt werden, wir betrachten einige Beispiele im Interpreter:

```
*Main> :set -XTemplateHaskell
*Main> let f = [| \x ->  1 + x |]
*Main> runQ f
LamE [VarP x_0] (InfixE (Just (LitE (IntegerL 1))) (VarE GHC.Num.+) (Just (VarE x_0)))
*Main> $(f) 6
7
*Main> let g = [| \x -> 1 + $(f) x|]
*Main> runQ g
LamE [VarP x_1]
  (InfixE (Just (LitE (IntegerL 1)))
          (VarE GHC.Num.+)
          (Just (AppE (LamE
                        [VarP x_2]
                        (InfixE
                            (Just (LitE (IntegerL 1)))
                            (VarE GHC.Num.+)
                            (Just (VarE x_2))))
                      (VarE x_1))))
*Main> $f 3
4
*Main> $(g) 6
8
>$f $ $g 10
13
*Main> $([| $f $ $g $ $f  $( [| $(f) 3 * ($g) 3 |]) |])
24
```

Beachte, dass man den splice-Operator ohne Klammern verwenden kann, wenn man nur einen Identifier splicen möchte. Beachte aber auch, dass keine Leerzeichen eingefügt werden, denn sonst

verwendet man den Operator $ für die schwach bindende Anwendung. In obigen Beispielen sind verschiedene Schreibweisen und Mixturen zu sehen.

Als weiteres größeres Beispiel betrachten wird die Erzeugung einer Meta-Funktion `genMapN :: Int -> Q Dec`, welche generisch `map`-Funktionen erzeugt: `$(genMapN 1)` erzeugt die übliche `map`-Funktion auf Listen, und `$(genMapN 2)` erzeugt eine binäre `map`-Funktion vom Typ `(a -> b -> c) -> [a] -> [b] -> [c]` usw.

```
genMapN :: Int -> Q Dec
genMapN n
  | n >= 1    = funD name [cl1, cl2]
  | otherwise = fail "genMapN: argument n may not be <= 0."
  where
    name = mkName $ "map" ++ show n
    cl1  = do f  <- newName "f"
              xs <- replicateM n (newName "x")
              ys <- replicateM n (newName "ys")
              let argPatts  = varP f : consPatts
                  consPatts = [ [p| $(varP x) : $(varP ys) |]
                              | (x,ys) <- xs `zip` ys ]
                  apply     = foldl (\ g x -> [| $g $(varE x) |])
                  first     = apply (varE f) xs
                  rest      = apply (varE name) (f:ys)
              clause argPatts (normalB [| $first : $rest |]) []
    cl2  = clause (replicate (n+1) wildP) (normalB (conE '[])) []
```

Die Implementierung ist ähnlich zu `curryN` (aber erzeugt eine Deklaration und keinen Ausdruck). Z.B. erzeugt `$(genMapN 3)` die folgende Funktion zur Compilezeit:

```
map3 f (x:xs) (y:ys) (z:zs) = f x y z : map3 f xs ys zs
map3 _ _      _      _      = []
```

Wir lassen uns mit dem Compiler-Flag `-ddump-splices` die erzeugten Funktionen anzeigen: Sei `genMapN` im Modul `GenMap` definiert und das Modul `Map` von der Form:

```
 {-# LANGUAGE TemplateHaskell #-}
module Map where
import GenMap
$((\x -> [x]) <$> (genMapN 3))
```

Dann können wir das Erzeugen beim Laden des Moduls beobachten (Das zusätzliche `(\x -> [x]) <$> ...` macht aus einer `Q Dec` ein `Q [Dec]`):

```
[1 of 2] Compiling GenMap           ( GenMap.hs, interpreted )
```

```
[2 of 2] Compiling Map              ( Map.hs, interpreted )
Map.hs:4:3-29: Splicing declarations
    (\ x_a7EN -> [x_a7EN]) <$> (genMapN 3)
  ======>
    map3
      f_a7Fq
      (x_a7Fr : ys_a7Fu)
      (x_a7Fs : ys_a7Fv)
      (x_a7Ft : ys_a7Fw)
      = (((f_a7Fq x_a7Fr) x_a7Fs) x_a7Ft
           : (((map3 f_a7Fq) ys_a7Fu) ys_a7Fv) ys_a7Fw)
    map3 _ _ _ _ = []
Ok, two modules loaded.
```

Die Definition von `genMapN` verwendet die besprochenen Möglichkeiten, abstrakte Syntaxbäume zu erzeugen. Dabei werden Quotation-Klammern verwendet und der splice-Operator wird an verschiedenen Stellen verwendet (z.B. in der Hilfsfunktion `apply`, welche den Rumpf der Funktion erstellt). Zudem werden Identifier-Quotes (nämlich `'[]`) verwendet, um den Objekt-Programm-Namen zu erzeugen, der zu Haskells Listen-Konstruktor `[]` gehört. Außerdem zeigt das Beispiel, wie alle drei APIs (direkte Konstruktoren der Syntaxbäume mit den Datentypen und der Q-Monade, Konstruktionsfunktionen und Quotation-Klammern) zum Erzeugen von Template Haskell-Objekt-Code verzahnt werden können.

Schließlich zeigt das Beispiel, wie sich Bindungsbereiche auf Objekt-Programme erweitern: Wie in normalem Haskell-Code zählt der innerste Binder und es werden statische Bindungsbereiche verwendet (d.h. Binder, so wie sie im Quellcode zu lesen sind). Quotation-Klammern und splice-Operationen ändern dies nicht, aber sie können ein Objekt-Programm in den Bindungsbereich bringen. Betrachte z.B.

```
x :: Int
x = 42

static :: Q Exp
static = [| x |]

plus42 :: Int -> Int
plus42 x = $static + x
```

Das Vorkommen von `x` in `static` bezieht sich auf die globale Bindung `x = 42`. Das Splicen von `static` in der letzten Zeile ändert dies nicht, obwohl dort ein neuer Bindungsbereich für `x` vorhanden ist. Die einzige Ausnahmen zu statischen Bindungsbereichen sind die mit `mkName :: String -> Name` generierten Namen. Diese werden dynamisch behandelt und können daher durch Splicen eingefangen werden. Betrachte das Beispiel:

```
x :: Int
```

```
x = 42

dynamic :: Q Exp
dynamic = VarE (mkName "x")

times2 :: Int -> Int
times2 x = $dynamic + x
```

In diesem Fall wird der Name `x` in `times2` eingesetzt und anschließend an das `x` gebunden, welches ihm an nächsten ist. Daher ist das durch `$dynamic` erzeugte `x` durch das `x` an der Stelle `times2 x` gebunden und nicht durch das globale `x`.

8.3 Reification

Ein weiteres wichtiges Merkmal von Template Haskell ist Programm-Reifikation. Reifikation erlaubt es Meta-Programmen zur Compile-Zeit, Informationen über andere Programmteile abzufragen. D.h. Meta-Programme können andere Programmteile inspizieren und sich dabei Fragen wie „Welchen Typ hat diese Variable?“, „Welche Instanzen hat diese Typklasse?“ oder „Was sind die Datenkonstruktoren dieses Datentyps?“ beantworten lassen. Dadurch kann oft wiederkehrender Code oft automatisch statt händisch erzeugt werden. Das Paradebeispiel dazu ist die automatische Erzeugung von Typklasseninstanzen (wie es z.B. mit dem Schlüsselwort `deriving` bereits gemacht wurde). Wir betrachten hierfür als Beispiel die automatische Erzeugung von `Functor`-Instanzen:

Betrachte z.B. die polymorphen Datentypen

```
data Result e a = Err e | Ok a
data List     a = Nil | Cons a (List a)
data Tree     a = Leaf a | Node (Tree a) a (Tree a)
```

Die Functor-Instanzen hierfür sind einfach zu implementieren:

```
instance Functor (Result e) where
 fmap f (Err e) = Err e
 fmap f (Ok a)  = Ok (f a)

instance Functor List where
 fmap f Nil          = Nil
 fmap f (Cons a xs) = Cons (f a) (fmap f xs)

instance Functor Tree  where
 fmap f (Leaf a) = Leaf (f a)
 fmap f (Node l a r) = Node (fmap f l) (f a) (fmap f r)
```

Das Schema ist immer das gleiche. Daher kann man die `Functor`-Instanz algorithmisch erstellen: Um für einen Typ `T a` den Typkonstruktor `T` zu einer `Functor`-Instanz zu machen, muss man die Funktion `fmap :: (a -> b) -> T a -> T b` implementieren. Aus dem Typ und den Funktorgesetzen folgt, dass man in Werten von Typ `T a` alle Werte vom Typ `a` durch den Werte vom Typ `b` ersetzen muss, indem man die gegebene Funktion auf den Wert anwendet. Durch Rekursion muss man entsprechend auch die noch verpackten Werte vom Typ `T a` zu Werten vom Typ `T b` machen.

Diese Idee kann man verwenden, um die folgende Meta-Funktion `deriveFunctor :: Name -> Q [Dec]` zu implementieren, welche für einen gegebenen Typkonstruktornamen, eine Functor-Instanz erzeugt. Dabei werden die Hilfsfunktionen `genFmap`, `genFmapClause` und `newField` verwendet. Wir zeigen zunächst den vollständigen Code, bevor wir ihn erläutern:

```
{-# LANGUAGE TemplateHaskell #-}
module GenFunctor where

import Control.Monad
import Language.Haskell.TH
import Language.Haskell.TH.Syntax

data Deriving = Deriving { tyCon :: Name, tyVar :: Name }

deriveFunctor :: Name -> Q [Dec]
deriveFunctor ty
  = do (TyConI tyCon) <- reify ty
       (tyConName, tyVars, cs) <- case tyCon of
         DataD _ nm tyVars _ cs _ -> return (nm, tyVars, cs)
         NewtypeD _ nm tyVars _ c _ -> return (nm, tyVars, [c])
         _ -> fail "deriveFunctor: tyCon may not be a type synonym."
       let (KindedTV tyVar StarT) = last tyVars
           instanceType           = conT ''Functor `appT`
             (foldl apply (conT tyConName) (init tyVars))
       putQ $ Deriving tyConName tyVar
       sequence [instanceD (return []) instanceType [genFmap cs]]
  where
    apply t (PlainTV name)    = appT t (varT name)
    apply t (KindedTV name _) = appT t (varT name)


genFmap :: [Con] -> Q Dec
genFmap cs  = do
               -- erzeuge eine Definitionszeile pro Konstruktor in cs
```

```
              funD 'fmap (map genFmapClause cs)

genFmapClause :: Con -> Q Clause
genFmapClause c@(NormalC name fieldTypes)
  = do f          <- newName "f"
       fieldNames <- replicateM (length fieldTypes) (newName "x")
       let pats = varP f:[conP name (map varP fieldNames)]
           body = normalB $ appsE $ conE name : map (newField f) (zip fieldNames fieldTypes)
       clause pats body []

newField :: Name -> (Name, StrictType) -> Q Exp
newField f (x, (_, fieldType))
  = do Just (Deriving typeCon typeVar) <- getQ
       case fieldType of
         -- Fall: x ist vom Typ a, dann ersetze durch (f x)
         VarT typeVar'
            | typeVar' == typeVar -> [| $(varE f) $(varE x) |]
         -- Fall: x ist vom Typ T a, dann ersetze durch (fmap f x)
         (AppT ty (VarT typeVar'))
          | leftmost ty == (ConT typeCon) && typeVar' == typeVar ->
                                [| fmap $(varE f) $(varE x) |]
         -- Ansonsten: behalte x
         _ -> [| $(varE x) |]

leftmost :: Type -> Type
leftmost (AppT ty1 _) = leftmost ty1
leftmost ty           = ty
```

Z.B. erzeugt `$(deriveFunctor ''Tree)` den folgenden Code:

```
instance Functor Tree where
  fmap f (Leaf x)     = Leaf (f x)
  fmap f (Node l x r) = Node (fmap f l) (f x) (fmap f r)
```

Auch dies kann man verifizieren:

```
 stack exec -- ghci -ddump-splices FunEx.hs
GHCi, version 8.8.3: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling GenFunctor       ( GenFunctor.hs, interpreted )
[2 of 2] Compiling FunEx            ( FunEx.hs, interpreted )
FunEx.hs:6:3-22: Splicing declarations
    deriveFunctor ''Tree
  ======>
    instance Functor Tree where
```

```
fmap f_a6le (Leaf x_a6lf) = Leaf (f_a6le x_a6lf)
fmap f_a6lg (Node x_a6lh x_a6li x_a6lj)
  = ((Node ((fmap f_a6lg) x_a6lh)) (f_a6lg x_a6li))
      ((fmap f_a6lg) x_a6lj)
```

Hierbei hat `FunEx.hs` als Inhalt:

```
{-# LANGUAGE TemplateHaskell #-}
module FunEx where
import GenFunctor

data Tree     a = Leaf a | Node (Tree a) a (Tree a)
$(deriveFunctor ''Tree)
```

Die Meta-Funktion `deriveFunctor` ruft zunächst die Funktion `reify :: Name -> Q Info` für den Typkonstruktornamen auf, um die Datentyp-Definition des zugehörigen Typs zu erhalten. Verwendung von `reify` liefert auch, ob der Typ mit `data` oder `newtype` konstruiert wurde, welche Konstruktoren verwendet wurden und wie diese genau aussehen. Zunächst extrahiert `deriveFunctor` mithilfe von `reify` den Typkonstruktornamen `tyConName`, die deklarierten Typvariablen `tyVars` und die verwendeten Konstruktoren `cs`. Dann berechnet `deriveFunctor` die rechteste Typvariable `tyVar` und speichert sie zusammen mit dem Typkonstruktornamen `tyConName` im Zustand der Q-Monade. Diese Information wird später in `newField` benötigt und extrahiert. Dann erzeugt `deriveFunctor` die `fmap`-Definition mithilfe von `genFmap`. Dabei wird für jeden Konstruktor aus `cs` eine Zeile der `fmap`-Funktion mittels `genFmapClause` erzeugt. Diese wendet `newField` für jedes Argument des Konstruktors an, und wendet die Funktion `f :: a -> b` auf Konstruktorargumente an, die vom Typ `a` sind, und wendet `fmap f` auf Argumente vom Typ `T a` an. Sie lässt alle anderen Konstruktorargumente unverändert.

8.4 Quasi-Quotes

Wir haben bereits die Oxford-Klammern als Quotation-Klammern gesehen, wobei diese für verschiedene Typen verwendet werden können, z.B. `[e| ... |]` für die Erzeugung von Ausdrücken vom Typ `Exp` usw. Mit der Erweiterung `QuasiQuoter` kann man das gleiche Prinzip für die eigenen Datentypen verwenden. Der große Vorteil ist: Man kann dadurch andere Sprachen in ihrer Syntax direkt im Quelltext einbinden. Z.B. bietet die shakespeare-Bibliothek einen Quasi-Quoter `[hamlet| ... ]` mit dem HTML-artiger Code direkt in einen Datentyp, der HTML-Dokumente darstellt, konvertiert werden kann. In (Has20) wird ein Quasi-Quoter `[regex| ... ]` für reguläre Ausdrücke vorgestellt und implementiert. Wir halten es hier kürzer und demonstrieren das Vorgehen anhand eines Minimalbeispiels.

Angenommen wir haben eine Sprache, die nur aus den Worten 0 und 1 besteht, so kann diese z.B. durch den Datentyp

```
data Simple = Zero | One
```

repräsentiert werden. Wir möchten nun ermöglichen, dass der Benutzer im Quellcode `[simple| 0 |]` oder `[simple| 1 |]` eingeben kann, was direkt in einen Syntaxbaum vom Typ `Simple` konvertiert wird, wobei das Ergebnis im Maybe-Typ zurückgegeben wird, um Fehler abzufangen. Wie gesagt, das Beispiel ist sehr künstlich und minimal, üblicherweise würde man statt `0` und `1` komplizierte Quelltexte einer komplizierten Sprache erwarten.

Wir zeigen zunächst den Quellcode und erläutern ihn im Anschluss:

```
{-# LANGUAGE QuasiQuotes, TemplateHaskell #-}
module Simple where
import Language.Haskell.TH.Quote
import Language.Haskell.TH
import Language.Haskell.TH.Syntax
import Data.Char(isSpace)

data Simple = Zero | One
 deriving Show

compile :: String -> Q Exp
compile = lift . parseSimple

parseSimple xs =
    let removeBlanks = [a | a <- xs, not $ isSpace a]
    in case removeBlanks of
            "0" -> Just Zero
            "1" -> Just One
            _  ->  Nothing

instance Lift Simple where
  lift Zero = conE 'Zero
  lift One  = conE 'One
  liftTyped = unsafeTExpCoerce . lift -- liftTyped muss impementiert werden
                                      -- (in neuer Versionen)


simple :: QuasiQuoter
simple = QuasiQuoter {
    quoteExp  = compile
  , quotePat  = notHandled "patterns"
  , quoteType = notHandled "types"
  , quoteDec  = notHandled "declarations"
```

```
  }
  where notHandled things = error $ things ++ " are not handled by the simple quasi
```

Wir können nun wie versprochen Quasiquotes verwenden:

```
$> stack exec -- ghci Simple.hs
GHCi, version 8.8.3: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Simple           ( Simple.hs, interpreted )
Ok, one module loaded.
*Simple> :set -XTemplateHaskell -XQuasiQuotes
*Simple> [simple| 0 |]
Just Zero
*Simple> [simple| 1 |]
Just One
*Simple> [simple| 2 |]
Nothing
*Simple> [simple| 0      |]
Just Zero
```

Zur Erläuterung des Quellcodes: Die wesentliche Funktion ist die Definition von `simple :: QuasiQuoter`. Hier werden die Funktionen definiert, der Typ `QuasiQuoter` ist definiert als:

```
data QuasiQuoter = QuasiQuoter {
    -- | Quasi-quoter for expressions, invoked by quotes like lhs = $[q|...]
    quoteExp  :: String -> Q Exp,
    -- | Quasi-quoter for patterns, invoked by quotes like f $[q|...] = rhs
    quotePat  :: String -> Q Pat,
    -- | Quasi-quoter for types, invoked by quotes like f :: $[q|...]
    quoteType :: String -> Q Type,
    -- | Quasi-quoter for declarations, invoked by top-level quotes
    quoteDec  :: String -> Q [Dec]
    }
```

Für den QuasiQuoter für `Simple` ist nur der Fall für Ausdrücke interessant, da die anderen Fälle für unseren Typ `Simple` sowieso nicht vorkommen. D.h. wir definieren die Funktion `quoteExp` als `compile :: String -> Q Exp`. Die `compile`-Funktion setzt sich als Komposition zweier Funktionen zusammen: Dem Parser `parseSimple :: String -> Maybe Simple`, welcher einen `String` in den Typ `Simple` parst oder `Nothing` ergibt, falls das Parsen fehlschlägt, und der Funktion `lift`, die über die Typklasse `Lift` aus `Language.Haskell.TH.Syntax` zur Verfügung steht. Diese ist definiert als

```
class Lift t where
  lift :: t -> Q Exp
  liftTyped :: Quote m => t -> Code m t
```

Sie implementiert die Abbildung des Typs in einen Ausdruck (als Q-monadische Aktion). Die `Lift`-Instanz von Simple ist einfach, sie setzt den Syntaxbaum direkt zusammen.

Anstelle der beiden Funktionen `parseSimple` und `lift` hätten wir `compile` auch direkt definieren können als:

```
compile :: String -> Q Exp
compile xs =
    let removeBlanks = [a | a <- xs, not $ isSpace a]
    in case removeBlanks of
            "0" -> appE (conE 'Just) (conE 'Zero)
            "1" -> appE (conE 'Just) (conE 'One)
            _  -> conE 'Nothing
```

aber üblicherweise steht der Parser schon zur Verfügung bzw. sollte gekapselt werden von der Erzeugung des Haskell-Syntaxbaums.

8.5 Zusammenfassung und Quellen

Wir haben die wesentlichen Konstrukte von Template Haskell betrachtet. Dieses Kapitel orientiert sich im Wesentlichen an (Has20), wobei teilweise die Syntax an die aktuelle GHC-Version angepasst wurden und Beispiele eingefügt wurden. Auch das Material in (Jos19) wurde verwendet, um die Darstellung anzureichern.

9 Lean – Funktionales Programmieren und Verifikation

9.1 Einleitung

Die funktionale Programmiersprache Lean ist eine strikte funktionale Programmiersprache, die sogenannte abhängige Typen (dependent types) verwendet und die Verifikation von Programmeigenschaften mithilfe des Typsystems erlaubt.

Ziel dieses Kapitels ist es, einen kurzen Überblick über die Syntax von Lean zu geben, um auch in dieser Sprache funktionale Programme schreiben zu können. Auf die Verifikation mit Lean wird nicht tiefergehend eingegangen.

Lean wurde 2013 von Leonardo de Moura (Microsoft Research) in Version 0.1 veröffentlicht, wobei Lean damals noch aus C++-Bibliotheken bestand. In 2017 wurde Lean 3 veröffentlicht und von vielen Mathematikern verwendet. Die Lean-Community implementierte die sogenannte Mathlib, die am Ende von Lean 3 aus über 1 Million Zeilen formalisierter Mathematik bestand. In 2023 wurde Lean 4 veröffentlicht. Wesentliche Neuerung war, das der Großteil von Lean 4 selbst in Lean 4 implementiert ist. Mathlib wurde nach Lean 4 exportiert und besteht mittlerweile aus über 1,5 Millionen Zeilen Code.

9.1.1 Installation

Hinweise zur Installation von Lean für verschiede Betriebssysteme sind unter

`https://leanprover-community.github.io/get_started.html`

bzw. in der offiziellen Dokumentation unter

`https://docs.lean-lang.org/lean4/doc/quickstart.html`

zu finden.

9.2 Einführung in Lean

9.2.1 Auswerten

Ein Ausdruck `Expr` kann durch die Quelltextanweisung

```
#eval Expr
```

ausgewertet werden. Das Ergebnis wird (bei Verwendung von Visual Studio Code) im VS Code-Tab „Lean Infoview“ gezeigt. Z.B. ergibt die Anweisung `#eval 6*6` wie erwartet den Wert 36.

Lean verwendet die strikte Auswertung, d.h. beim Auswerten einer Funktionsanwendung $f\ a_1\ \ldots\ a_n$ werden erst die Argumente a_1 bis a_n ausgewertet, bevor sie in den Rumpf der Definition von f eingesetzt werden.

9.2.2 Typen

Mit der Anweisung

`#check Expr`

kann der Typ eines Ausdrucks berechnet und angezeigt werden. Dabei wird Typinferenz verwendet, d.h. Lean versucht für nicht angegebene Typen einen eigenen Typen herzuleiten. Z.B. ergibt

`#check 6 * 6`

als Ergebnis `6 * 6 : Nat`. Der Typ `Nat` steht für natürliche Zahlen inklusive der 0. Man kann Typen angeben mit der Syntax `Expr : Type`. Z.B. können wir schreiben:

`#check (6 * 6 : Int)`

um dem Produkt den Typ `Int` (für beliebige Ganzzahlen) zu zuweisen.

Funktionstypen werden mit den Funktionspfeil `->` oder dem entsprechenden Unicode-Symbol für einen Pfeil geschrieben.

9.2.3 Funktionen definieren

Funktionen werden mit dem Schlüsselwort `def` definiert und haben die Form

`def` *funktionsName Argument*$_1$ ... *Argument*$_n$ `:=` *Rumpf*

Z.B. können wir eine Funktion zum Verdoppeln einer Zahl definieren:

```
def verdopple n := 2*n
```

Oft müssen wir jedoch einige Typen angeben, damit Lean die Definition akzeptiert, z.B. akzeptiert Lean die folgende Funktionsdefinition nicht:

```
def verdopple' n := n+n
```

Der gemeldete Fehler ist

```
typeclass instance problem is stuck, it is often due to metavariables
HAdd ?m.512 ?m.512 (?m.501 n)
```

Das Problem ist, dass Lean nicht weiß, welchen Typ er für `n` verwenden soll, da + überladen ist.
Die Lösung ist es, den Typ anzugeben. Interessanterweise funktioniert die Definition

```
def verdopple'' n := 0+n+n
```

da Lean für 0 den Typ `Nat` annimmt und anschließend „weiß“, welche Implementierung von + verwendet werden muss.
Der Typ einer Funktion wird durch Annotation der Argumente mit einem Typ und einer zusätzlichen Annotation für den Ergebnistyp angegeben. Die Syntax ist dann

`def` *funktionsName* `(`*Argument*$_1$ `:` *Typ*$_1$`)` ... `(`*Argument*$_n$ `:` *Typ*$_n$`)` `:` *Ergebnistyp* `:=` *Rumpf*

Z.B. können wir definieren:

```
def verdopple' (n : Nat) : Nat := n+n
```

Lean zeigt dann als den Typ `Nat -> Nat` als Typ von `verdopple'` an.
Will man den Typ ähnlich wie in Haskell aufschreiben, so kann man eine Abstraktion verwenden (eine Abstraktion $\lambda x.s$ wird in Lean als `fun x => s` oder alternativ λ`x => s` geschrieben).
Z.B. können wir schreiben:

```
def verdopple''' : Nat -> Nat := fun n => n+n
```

Da Lean als Beweisassistent fungiert, müssen Funktionen in Lean *total* und *terminierend* sein. Kann die Terminierung nicht automatisch nachgewiesen werden, so meldet Lean ein Fehler.
Z.B. wird die Definition

```
def nichtterminierend (n : Nat) : Nat := nichtterminierend n
```

nicht akzeptiert, sondern liefert den Fehler „fail to show termination for nichtterminierend ...“.
Die Terminierung kann händisch nachgewiesen werden (für `nichtterminierend` unmöglich) oder die Funktion kann als explizit als partiell definiert werden. Dies geht mit dem Schlüsselwort `partial`. Z.B. wird

```
partial def nichtterminierend (n : Nat) : Nat := nichtterminierend n
```

vom Lean-Interpreter akzeptiert.
Lean verwendet abhängige Typen (dependent types), d.h. das Typen von Werten abhängen können. Z.B. kann man in Lean eine Funktion definieren, die je nach Wahrheitswert der Eingabe komplett verschiedene Typen liefert:

```
def jenachdem (w:Bool) : if w then Nat else Bool :=
 match w with
   | true => (0:Nat)
   | false => false
```

Das wesentliche hierbei ist, dass der *Typ* von `jenachdem` vom Wert von `w` abhängt.

9.2.4 Namensbereiche: Namespaces

Lean verwendet für die Strukturierung sogenannte „namespaces“, also Namensbereiche. Dadurch können gleichnamige Funktionen wie z.B. `map` für unterschiedliche Typen verwendet werden, indem man sie z.B. im Namensbereich `List` für Listen und im Namensbereich für `Array` für Felder definiert. Der Zugriff erfolgt dann über `List.map` bzw. `Array.map`.

Will man direkten Zugriff, so kann man mit `open` *Namensbereich*, den Namensbereich öffnen. Z.B.

```
#eval List.map (fun x => x + x) [1,2,3] -- ergibt [2,4,6]
#eval map (fun x => x + x) [1,2,3] -- ergibt Fehler: unknown identifier 'map'
open List
#eval map (fun x => x + x) [1,2,3] -- ergibt [2,4,6]
```

Durch die Namensbereiche ist auch die sogenannte Punkt-Notation für Funktionen erlaubt: Anstelle von *Namensbereich*.f *Argument* darf auch *Argument*.f geschrieben werden, wenn der Typ von *Argument* im *Namensbereich* definiert wurde. Z.B. darf man daher schreiben (auch ohne vorheriges `open`)

```
#eval [1,2,3].map (fun x => x + x) -- ergibt [2,4,6]
```

Man kann Funktionen zu einem bestenden Namenspace hinzufügen. Z.B. kann man Funktionen mit Namensbereich definieren. Z.B.

```
def Nat.verdoppele (n: Nat) : Nat := n+n
```

Damit ist `verdoppele` im Namensbereich `Nat` definiert und man z.B. schreiben:

```
#eval (5:Nat).verdoppele -- ergibt 10
```

9.2.5 Eingebaute und vordefinierte Typen

9.2.5.1 Zahlen

Lean kennt Basistypen für natürliche Zahlen (`Nat`), für Ganzzahlen (`Int`) für endliche natürliche Zahlen (der Typ `Fin n` enthält alle natürlichen Zahlen kleiner als `n`, die meisten Operation werden automatisch modulo `n` durchgeführt). `ISize` für Ganzzahlen beschränkter Länge (Architekturabhängig, 32bit oder 64bit), `USize` für natürliche Zahlen beschränkter Länge (Architekturabhängig, 32bit oder 64bit), Bit-Vektoren beschränkter Länge (`BitVec`). Fließkommazahlen (`Float32` unter Verwendung von 32 Bits und `Float` mit doppelter Genauigkeit, d.h. 64 Bits).

9.2.5.2 Zeichen und Zeichenketten

Zeichen (vom Typ `Char`) werden wie in Haskell in einfache Anführungsstriche geschrieben, Zeichenketten (vom Typ `String`) werden in doppelte Anführungsstriche geschrieben. Auch wenn sich die Zeichenketten, wie Listen von Zeichen verhalten, wird intern eine andere / effizientere Repräsentation verwendet.

9.2.5.3 Typen mit einem und keinem Wert

Der Typ `Unit` enthält nur das Nulltupel `()`. Der Typ `Empty` enhält keine Werte.

9.2.5.4 Wahrheitswerte

Wahrheitswerte sind `false` und `true` welche zum Typ `Bool` gehören. Die Operatoren `&&`, `||` und `^^`, können infix für die Konjunktion, Disjunktion und das Exklusive Oder verwendet werden. Die Negation wird durch den Präfixoperator `!` ausgedrückt.

9.2.5.5 Optionale Werte

Der Typ `Option` mit Konstruktoren `none` und `some` ist analog zum `Maybe`-Typ in Haskell zu sehen.

9.2.5.6 Produkttypen: Paare und Tupel

Paare werden wie in Haskell mit runden Klammern geschrieben und die Selektoren `fst` und `snd` sind verfügbar. Der Typ ist $(\alpha_1 \times \alpha_2)$ oder alternativ `Prod` α_1 α_2. Sie stellen Produkttypen dar, da sie zwei Typen miteinander kombinieren. Auch n-stellige Tupel sind verfügbar, Objekte werden werden als `(`e1,...,en,`)` geschrieben, der Typ ist $(\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n)$. Intern werden n-Tupel durch geschachtelte Paare dargestellt, wobei die Schachtelung rechts-assoziativ ist. Auch `fst` und !`snd` sind daher verfügbar, z.B. ergibt `#eval Prod.fst (1,2,3)` als Ergebnis `1` und `#eval Prod.snd (1,2,3)` als Ergebnis das Paar `(2,3)`.

9.2.5.7 Summentypen

Summentypen vereinigen zwei Typen disjunkt. Lean kennt hierzu eine explizite Syntax, die analog zum `Either`-Typen in Haskell zu sehen ist: Wenn α und β Typen sind, dann ist der Typ $\alpha \oplus \beta$ die disjunkte Vereinigung der beiden Typen. Die Konstruktoren sind `inl` und `inr`.

9.2.5.8 Listen

Homogene Listen sind in Lean als Typ `List a` vorhanden, wobei `a` der Inhaltstyp der Elemente ist. Der Konstruktor `nil` für die leere Liste kann genau wie in Haskell als `[]` geschrieben

werden. Der Konstruktor `cons` wird infix als zwei Doppelpunkte geschrieben (z.B. repräsentiert `1::2::3::[]` die Liste der Zahlen 1,2 und 3). Auch die übliche Syntax, die Elemente mit Kommas trennt und die Liste mit `[` und `]` umschließt, ist erlaubt. Mit `List.cons` und `List.nil` kann aber auch auf die beiden Konstruktoren zugegriffen werden.

Listen sind als *induktiver Datentyp* definiert, inetwa wie die folgende Definition

```
inductive List (a:Type) where
  | nil: List a
  | cons: a -> List a -> List a
```

Das entspricht in Haskell den rekursiv definierten Datentypen, wobei in Lean keine unendlichen Listen erlaubt sind. Daher scheitert z.B. der Versuch eine Funktion wie `repeat` zu definieren, da deren Terminierung (weder automatisch noch manuell gezeigt werden kann). Die üblichen Listenfunktionen sind bereits implememtiert.

Wir zeigen einige Beispielaufrufe:

```
def mylist1 : List Nat := [1,2,3,4,5,6]
def mylist2 : List Nat := List.replicate 5 3
def mylist3 : List (List Nat) := [mylist1,mylist1,mylist2]

                                        -- ergibt
#eval mylist1                           -- [1,2,3,4,5,6]
#eval List.length mylist1               -- 6
#eval List.take 4 mylist1               -- [1,2,3,4]
#eval mylist1.take 4                    -- [1,2,3,4]
#eval List.drop 4 mylist1               -- [5,6]
#eval mylist1.get (2: Fin 6)            -- 3
#eval List.append mylist1 mylist2       -- [1,2,3,4,5,6,3,3,3,3,3]
#eval List.concat mylist1 50            -- [1,2,3,4,5,6,50]
#eval mylist1.reverse                   -- [6,5,4,3,2,1]
#eval mylist1.foldl (fun x y => x + y) 0 -- 21
#eval mylist1.foldr (fun x y => x + y) 0 -- 21
#eval mylist1.map (fun x => 2*x)        -- [2,4,6,8,10,12]
#eval mylist2.flatMap (fun x => [2*x])  -- [6,6,6,6,6]
#eval mylist1.filter (fun x => x > 5)   -- [6]
#eval mylist3.flatten                   -- [1,2,3,4,5,6,1,2,3,4,5,6,3,3,3,3,3]
```

Die Definition eines induktiven Datentyps erzeugt einen Namespace für den Typnamen und die Konstruktoren werden darunter definiert.

9.2.5.9 Arrays

Arrays sind mit dem Typ `Array a` verfügbar, wobei `a` der Typ der Elemente ist. Ein Array mit den Elementen `e1`, `e2`, ..., `en` kann mit `#[ e1, e2, ... en ]` erzeugt werden. Wenn `ar` ein

Array ist, dann kann mit `ar[`i`]` auf das $i+1$-te Element zugriffen werden, mit `ar[`i`:`j`]` kann auf das Unterarray von Index i bis j zugriffen werden (wobei `Subarray a` ein eigener Typ ist).

9.2.5.10 Subtypen

Subtypen erlauben es, einen bestehenden Typen mit einem Prädikat einzuschränken. Wenn p ein Prädikat ist und a ein Typ, dann ist die Syntax für den Subtyp, der nur Objekte vom Typ a enthält, die das Prädikat erfüllen: `{ n : `a` // `p` n }`. Wenn a inferiert werden soll, kann man auch schreiben `{ n // `p` n }`.

Z.B. kann man damit ein Vorgängerfunktion für natürliche Zahlen ohne 0 definieren:

```
def pred (x : {n : Nat // n > 0 }) : Nat := x - 1
```

Elemente eines Subtypen sind Paare bestehend aus einem Element des Typs und einem Beweis, dass das Element das Prädikat erfüllt. Will man `pred` auf ein Element anwenden, so muss man nachweisen, dass die Zahl vom passenden Subtyp ist:

```
def ten : { n : Nat // n > 0 } := ⟨10, by simp⟩
```

Der Beweis `by simp` ist ein Aufruf des eingebauten Simplifiers, der es automatisch schafft, die geforderte Eigenschaft zu zeigen.

Anschließend funktioniert

```
#check pred ten
#eval pred ten
```

und wir erhalten die erwarteten Ergebnisse. Ein Aufruf mit 0 ist gar nicht möglich, da wir `0 : { n : Nat // n > 0 }` nicht nachweisen können.

Schließlich sehen wir, dass wir Subtypen zum Basistypen automatisch konvertieren können, denn

```
#check (Nat.succ ten)
```

liefert ebenfalls `Nat` als Ergebnistyp. Analoges ist in der Definition von `pred` passiert: Die Subtraktion mit `- 1` in der Definition funktioniert, da `x` automatisch vom Subtypen `{ n : Nat // n > 0 }` zu seinem Basistypen `Nat` konvertiert wird und anschließend die Subtraktion auf `Nat` verwendet wird.

9.2.5.11 Verzögerte Auswertung

Um verzögerte Auswertung (oder *lazy evaluation* im Englischen) zu ermöglichen, kann man in strikten Sprachen Ausdrücke als Abstraktionen verpacken, d.h. anstelle von e verwendet

man $\lambda x.e$, wobei x nicht in e vorkommt. Will man die Auswertung von e erzwingen, wendet man die Abstraktion auf irgendeinen Wert (z.B. `()`) an. Lean bietet hierfür ein angenehmere Methode: Der Typ `Thunk a` (wobei `a` der Typ des verpackten Ausdruck ist). Mit `Thunk.mk e` wird die Abstraktion `fun (x:Unit) => e` verpackt. Mit `Thunk.get` kann auf den Ausdruck `e` zugegriffen werden. Lean kann automatisch den Thunk aufgrund des Typen Thunk erstellen, z.B. können wir schreiben

```
#eval Thunk.get ((factorial 100) : Thunk Nat)
```

was die Fakultät von 100 auswertet (wenn verb!factorial! entsprechend definiert ist).

Ein Beispiel für die Verwendung der verzögerten Auswertung ist

```
def leftIfGreaterZero (x:Nat) (y:Nat) : Nat :=
   if x > 0 then x else y

def lazyLeftIfGreaterZero (x:Nat) (y:Thunk Nat) : Nat :=
   if x > 0 then x else y.get
```

Währen `leftIfGreaterZero` immer beide Argumente auswertet, wird in `lazyLeftIfGreaterZero` die Auwertung des zweiten Arguments `y` verzögert. Die beiden Aufrufe

```
#eval lazyLeftIfGreaterZero 10 ((factorial 200000):Thunk Nat)
#eval leftIfGreaterZero 10 (factorial 200000)
```

zeigen den Unterschied: Die erste Auswertung geht schnell, während die zweite länger dauert.

9.3 Verifikation von Programmen mit Lean

Mit Lean kann man nicht nur funktionale Programme schreiben, sondern man kann die Korrektheit der Programme auch mit Lean verifizieren, d.h. Lean ist auch ein Beweiser. Genauer handelt es sich um einen interaktiven Theorembeweiser, d.h. man geht Schrittweise vor und gibt die einzelnen Beweissschritte ein, Lean unterstützt und prüft die Schritte. Dabei wendet man in jedem Schritt sogenannte Taktiken an (eine gute Übersicht über die verschiedenen Taktiken findet man unter `https://www.ma.imperial.ac.uk/~buzzard/xena/formalising-mathematics-2024/Part_C/Part_C.html`). Es gibt auch Taktiken, die vieles automatisch beweisen können (z.B. versucht Lean bei jeder Funktionsdefinition deren Terminierung automatisch nachzuweisen).

Wir geben hier nur einige Beispiele zur Verifikation und überlassen die theoretischen Grundlagen und die tiefergehende Beweisführung anderen Spezialveranstaltungen.

9.3.1 Einige Gesetze auf Listen

Gleichungen für Listenverarbeitende Funktionen sind Standardbeispiele zur Verifikation funktionaler Programme. Da in Lean die Listen endlich (und induktiv definiert) sind, kann strukturelle Induktion über den Aufbau der Liste verwendet werden. In Sprachen wie Haskell mit potentiell unendlich langen Listen sind andere Methode nötig.

Wir betrachten als Beispiel zwei Gesetze auf Listen. Wir verwenden jedoch einen eigenen Listen-Typ und eigene Funktionen, damit keine bewiesenen Gesetze aus den Standardbibliotheken angewendet werden können.

Wir definieren daher

```
import Mathlib

-- Eigener ListenTyp

inductive MyList (a:Type) where
  | myNil: MyList a
  | myCons: a -> MyList a -> MyList a

-- namespace MyList öffnen
open MyList
```

Nun definieren wir zwei Funktionen:

```
-- append-Funktion für MyList-Listen

def app (xs: MyList a) (ys: MyList a) : MyList a :=
 match xs with
  | myNil => ys
  | myCons a as => myCons a (app as ys)

-- reverse-Funktion für MyList-Listen

def rev (xs:MyList a) : MyList a :=
   match xs with
     | myNil => myNil
     | myCons a as => app (rev as) (myCons a myNil)
```

Als erstes zeigen wir die Assoziativität von append:

```
theorem app_associative:
          app (app xs ys) zs = app xs (app ys zs) := by
          ...
```

Dies formulieren ein Theorem namens `app_associative` als Namen erhält:

Im Infoview sehen wir das aktuelle Beweisziel. Der gesamte Beweis kann mit einer Induktion über die erste Liste geführt werden. Die entsprechende Taktik ist `induction`

```
theorem app_associative:
          app (app xs ys) zs = app xs (app ys zs) := by
    induction xs with
    ...
```

Anschließend müssen alle Fälle bewiesen werden, die zum Typ von `xs` passen. Das `xs` eine Liste ist, gibt es den Fall `xs = myNil` und `xs = myCons a as`. Für den zweiten Fall, verwenden wir die Induktionshypothese. Deshalb wir sie als zusätzlicher Namen `ih` aufgeführt:

```
theorem app_associative:
          app (app xs ys) zs = app xs (app ys zs) := by
    induction xs with
    | myNil          => rw [app]
                        rw [app]
    | myCons a as ih => rw [app]
                        rw [app]
                        rw [ih]
                        rw [app]
```

Die Taktik `rw[h]` wendet bereits bewiesene Gleichungen auf das Ziel an. Der Name `rw` stammt von rewrite, denn das Ziel wird durch rewriting mit der bekannten Gleichung ersetzt. Lean beschwert sich, wenn die Gleichung nicht angewendet werden kann. Wir verwenden hier `rw`, um die Definition von `app` anzuwenden, und einmal, um die Induktionshypothese anzuwenden.

Unser zweites Beispiel ist, die Gleichung `rev (rev xs) = xs` zu zeigen, denn zweimaliges Umdrehen einer endlichen Liste gibt die Liste selbst zurück[1]

Die Gleichung kann nicht direkt bewiesen werden, sondern erfordert zwei Hilfslemmas:

```
-- Hilfslemmas

lemma app_nil : app xs myNil = xs :=  by
  induction xs with
    | myNil          => rw[app]
    | myCons a as ih => rw[app]
                        rw[ih]
```

[1] In Haskell wäre die Gleichung nicht nur aufgrund unendlicher Listen falsch, auch für `xs=1:undefined` stimmt die Gleichung nicht in jedem Kontext: Z.B. `head (1:undefined) = 1` aber `head (rev (rev 1:undefined)) = undefined`.

```
lemma rev_append : rev (app xs ys) = app (rev ys) (rev xs)
   := by induction xs with
       | myNil          => rw[rev]
                           rw[app]
                           rw[app_nil]
       | myCons a as ih => rw[app]
                           rw[rev]
                           rw[ih]
                           rw[app_associative]
                           rw[rev]

-- reverse zweimal anwenden ist die Identität
theorem rev_rev: rev (rev xs) = xs
    := by induction xs with
  | myNil          => rw[rev]
                      rw[rev]
  | myCons a as ih => rw[rev]
                      rw[rev_append]
                      rw[ih]
                      rw[rev]
                      rw[rev]
                      rw[app]
                      rw[app]
                      rw[app]
```

Das Beispiel zeigt, wie vorher bewiesene Lemmas und Theorem wiederverwendet werden.

9.4 Abhängige Typen

Lean erlaubt die Verwendung von abhängigen Typen (dependent types). Mit der Funktion `jenachdem` haben wir bereits ein Beispiel für abhängige Typen gesehen. Ein Standardbeispiel für abhängige Typen sind Vektoren (also Listen fester Länge), deren Typ die Länge des Vektors speichert. Viele Lean-Tutorials drücken einen solchen Typ für Vektoren als Subtyp von Listen aus, der zusätzlich die Länge des Vektors speichert. Das hat den Vorteil, dass man Listenfunktionen für Vektoren verwenden kann (siehe z.B. (Ren25)). Die direkte Definition mit eigenem induktiven Datentyp ist jedoch leichter zu verstehen, daher geben wir hier eine direkte Definition und nennen den Typ `Vect`:

```
inductive Vect (a : Type) : Nat -> Type where
  | nil : Vect a 0
  | cons : a -> Vect a n -> Vect a (n + 1)
```

Die Zahl im Typ kodiert dabei die Länge des Vektors. Die Vektor-Addition kann dann implementiert werden als:

```
def Vect.add (x : Vect Nat n) (y: Vect Nat n) : Vect Nat n :=
  match x with
  | Vect.nil => Vect.nil
  | Vect.cons a as =>
        match y with
          | Vect.cons b bs => Vect.cons (a + b) (Vect.add as bs)
```

Die Definition macht deutlich, dass man nur gleich lange Vektoren addieren kann und einen genauso langen Vektor als Ergebnis erhält.

Wendet man die Definition auf unterschiedliche lange Vektoren an so erhält man einen Typfehler:

```
#check (Vect.cons 1 Vect.nil).add (Vect.cons 1 (Vect.cons 2 Vect.nil)) -- Typfehler
```

Mithilfe der abhängigen Typen kann man leicht `head` und `tail` für nicht-leere Vektoren definieen

```
def Vect.head (x : Vect a (n+1)) : a :=
   match x with
   | Vect.cons a _ => a

def Vect.tail (x : Vect a (n+1)) : Vect a n :=
   match x with
   | Vect.cons _ as => as
```

Fälle für `nil` sind nicht möglich, da $n + 1 > 0$ gilt. Die Awendung auf leere Vektoren (d.h. Vektoren vom Typ `Vect a 0`) ist ungetypt und daher nicht möglich.

Einige Beispiele:

```
def emptyVect : Vect Nat 0 := Vect.nil
def myVect1 := Vect.cons 100 (Vect.cons 200 (Vect.cons 300 Vect.nil))
#eval myVect1.head     -- ergibt 100
#eval myVect1.tail     -- ergibt Vect.cons 200 (Vect.cons 300 (Vect.nil))
#eval emptyVect.head   -- Typfehler
```

Will man eine Funktion `replicate` implementieren, die ein gegebenen Element n mal repliziert, d.h. einen Vektor der Länge n erstellt, der ein und dasselbe Element erhält, so ist die möglich:

```
def Vect.replicate (n:Nat) (x:a) : (Vect a n) :=
  match n with
   | 0 => Vect.nil
   | Nat.succ m =>  Vect.cons x (Vect.replicate m x)
```

Auch hier wird deutlich, wie abhängige Typen funktionieren: Der Typ von `replicate` ist Abhängig vom Wert der Eingabe n.

Will man die `append`-Funktion für Vektoren definieren, so addieren sich die Längen der beiden Vektoren. Dies kann folgendermaßen in Lean bewerkstelligt werden:

```
def Vect.append (xs:Vect a m) (ys:Vect a n) : (Vect a (n+m)) :=
 match xs with
   | Vect.nil => ys
   | Vect.cons z zs => Vect.cons z (zs.append ys)
```

Damit sind die Vektorlängen der Operationen bereits zur Compilezeit vorhersagbar. Benutzt man dieselbe Funktionsdefintion aber verwendet als Ergebnistyp den Typ `Vect a (m+n)`, so scheitert die Typisierung. Intenr liegt dies daran, dass der Typcheck in Lean die Gleichheit $m + n = n + m$ nicht herleiten kann (oder genauer gesagt: will). Der Grund hierfür ist, dass der Typcheck anderenfalls nicht terminiert. Man kann dem Typcheck jedoch helfen: Folgendes funktioniert:

```
def Vect.append (xs:Vect a m) (ys:Vect a n) : (Vect a (n+m)) ...

def Vect.append2 (xs:Vect a m) (ys:Vect a n) : (Vect a (m+n)) :=
 Nat.add_comm m n  ▸  xs.append ys
```

Dert Teil `Nat.add_comm m n  ▸` sagt dem Lean-System, dass er ein bewiesenes Gesetz (hier die Kommutativität der Addition für natürliche Zahlen) beim Typcheck verwenden soll. D.h. hier werden Logik und Programmierung vermischt. Ähnliches ist nötig, wenn man die `reverse`-Funktion definiert:

```
def Vect.reverse (xs:Vect a n) : (Vect a n) :=
 match n, xs with
   | 0, Vect.nil => Vect.nil
   | u+1, Vect.cons y ys =>
      Nat.add_comm 1 u ▸ ys.append (Vect.cons y Vect.nil)
```

Das Beispiel zeigt auch, dass man mit dem `match`-Konstrukt mehrere Terme auf einmal matchen und Zerlegen kann (hier die Zahl `n` und der Vektor verb!xs!)

Als abschließendes Beispiel betrachten wir den sicheren Zugriff auf ein Vektorelement per Index. Normalerweise, hat man Ausnahmefälle zu beachten, für die der Index nicht gültig ist (z.B. größer als die Anzahl der Elemente ist). Mit abhängigen Typen und dem Typ `Fin` lassen sich diese Fälle auf Typebene ausschließen:

```
def Vect.get (x: Vect Nat n) (i:Fin n) : Nat :=
 match i, x with
  | ⟨0,_⟩, Vect.cons y ys => y
  | ⟨j+1,h⟩, Vect.cons _ ys => ys.get ⟨j,Nat.le_of_succ_le_succ h⟩
```

Wir erinnern daran, dass ein Element vom Typ `Fin n` ein Paar `⟨x,h⟩` ist, wobei `x` eine natürliche Zahl vom Typ `Nat` ist und `h` der Beweis, dass `x < n` gilt. In der Definition von `get` wird dieser Beweis mit `match` extrahiert und anschließend verwendet, um den Beweis für `j = i-1` zu erbringen (mithilfe des Theorems `le_of_succ_le_succ`, welches zeigt, dass $n+1 \leq m+1 \implies n \leq m$ für alle Zahlen n, m vom Typ `Nat` gilt.

Wir zeigen einige Beispiele:

```
def myVect2 := Vect.replicate 5 2
def four: Fin 5 := ⟨ 4, by simp ⟩
                        -- Ausgabe:
#check myVect1          -- myVect1 : Vect Nat (0 + 1 + 1 + 1)
#check myVect2          -- Vect Nat 5
#eval myVect2.get four  -- 2
#check myVect1.get four -- appication type mismatch ...
```

9.5 Quellennachweis

Einführungen in die funktionale Programmierung mit Lean finden sich in (Chr23) und (Ren25). Die Sprachreferenz zu Lean ist (Lea25).

Literatur

Ariola, Zena M. ; Felleisen, Matthias: The call-by-need lambda calculus. In: *Journal of Functional Programming* 7 (1997), Nr. 3, S. 265–301. – ISSN 0956–7968

Allen, C. ; Moronuki, J. ; Syrek, S.: *Haskell Programming from First Principles*. Lorepub LLC, 2019 `https://haskellbook.com/`. – ISBN 9781945388033

Barendregt, H. P.: *The Lambda Calculus. Its Syntax and Semantics*. Amsterdam, New York : North-Holland, 1984

Bird, Richard: *Introduction to Functional Programming using Haskell*. 2. Prentice Hall PTR, 1998. – 448 S. – ISBN 0134843460

Bird, Richard: *Thinking Functionally with Haskell*. Cambridge University Press, 2014. – ISBN 1107452643

Bragilevsky, Vitaly: *Haskell in Depth*. Manning, 2021. – ISBN 9781617295409

Bird, Richard ; Wadler, Philip: *Introduction to Functional Programming*. Upper Saddle River, NJ, USA : Prentice-Hall International, 1988 (Prentice-Hall International Series in Computer Science)

Christiansen, David T.: *Functional Programming in Lean*. https://lean-lang.org/functional_programming_in_lean/, 2023

Chakravarty, Manuel ; Keller, Gabriele: *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004

Davie, Antony J. T.: *An introduction to functional programming systems using Haskell*. New York, NY, USA : Cambridge University Press, 1992. – ISBN 0–521–25830–8

Hankin, Chris: *An introduction to lambda calculi for computer scientists*. London, UK : King's College Publications, 2004 (Texts in Computing 2)

HaskellWiki: *A practical Template Haskell Tutorial*. `https://wiki.haskell.org/A_practical_Template_Haskell_Tutorial`, 2020

Hall, Cordelia V. ; Hammond, Kevin ; Jones, Simon L. P. ; Wadler, Philip: Type Classes in Haskell. In: *ACM Trans. Program. Lang. Syst.* 18 (1996), Nr. 2, S. 109–138

Hughes, J.: Why Functional Programming Matters. In: *Computer Journal* 32 (1989), Nr. 2, S. 98–107

Hutton, Graham: *Programming in Haskell*. Cambridge University Press, 2007

Jones, Mark P.: A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. In: *J. Funct. Program.* 5 (1995), Nr. 1, S. 1–35

Jost, Steffen: *Material zur Vorlesung Fortgeschrittene Funktionale Programmie-*

rung im WS 2018/19 an der LMU München. https://www.tcs.ifi.lmu.de/lehre/ws-2018-19/fun/, 2019

Lean Focused Research Organization: *The Lean Language Reference*. https://lean-lang.org/doc/reference/latest, 2025

Lipovaca, Miran: *Learn You a Haskell for Great Good! A Beginner?s Guide*. 1st. USA : No Starch Press, 2011. – ISBN 1593272839

Maguire, Sandy: *Thinking with Types – Type-Level Programming in Haskell*. https://leanpub.com/thinking-with-types, 2019. – https://leanpub.com/thinking-with-types

Marlow, Simon: *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013

Moggi, Eugenio: Notions of computation and monads. In: *Inf. Comput.* 93 (1991), Nr. 1, S. 55–92. http://dx.doi.org/http://dx.doi.org/10.1016/0890-5401(91)90052-4. – DOI http://dx.doi.org/10.1016/0890–5401(91)90052–4. – ISSN 0890–5401

McBride, Conor ; Paterson, Ross: Applicative programming with effects. In: *Journal of Functional Programming* 18 (2008), Nr. 1, S. 1–13. http://dx.doi.org/10.1017/S0956796807006326. – DOI 10.1017/S0956796807006326. – ISSN 0956–7968

Marlow, Simon ; Peyton Jones, Simon ; Kmett, Edward ; Mokhov, Andrey: Desugaring Haskell's Do-Notation into Applicative Operations. In: *Proceedings of the 9th International Symposium on Haskell*. New York, NY, USA : Association for Computing Machinery, 2016 (Haskell 2016). – ISBN 9781450344340, 92–104

O'Sullivan, Bryan ; Goerzen, John ; Stewart, Don: *Real World Haskell*. O'Reilly Media, Inc., 2008

Pepper, Peter: *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. Springer-Lehrbuch, 1998. – ISBN 3-540-64541-1

Peyton Jones, Simon L.: *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1987. – ISBN 013453333X

Peyton Jones, Simon L.: Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Hoare, Tony (Hrsg.) ; Broy, Manfred (Hrsg.) ; Steinbruggen, Ralf (Hrsg.): *Engineering theories of software construction*. Amsterdam, The Netherlands : IOS Press, 2001, S. 47–96

Peyton Jones, Simon (Hrsg.): *Haskell 98 language and libraries: the Revised Report*. New York, NY, USA : Cambridge University Press, 2003. – www.haskell.org

Pepper, Peter ; Hofstedt, Petra: *Funktionale Programmierung - Weiterführende Konzepte und Techniken*. Springer-Lehrbuch, 2006. – ISBN 3-540-20959-X

Pierce, Benjamin C.: *Types and programming languages*. Cambridge, MA, USA : MIT Press, 2002. – ISBN 0–262–16209–1

Peyton-Jones, Simon: Beautiful concurrency. In: Oram, Andy (Hrsg.) ; Wilson, Greg (Hrsg.): *Beautiful code*. O'Reilly, January 2007

Renz, Burkhardt: *Funktionale Programmierung in Lean Funktionale Programmierung in Lean. Vorlesungsskript TH Mittelhessen*. https://esb-dev.github.io/fpl.html, 2025

Snoyman, Michael: *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, Inc., 2012. – ISBN 1449316972. – Online verfügbar (angepasst an aktuelle Versionen) unter `https://www.yesodweb.com/book-1.6`

Snoyman, Michael: *Functors, Applicatives, and Monads*. `http://www.snoyman.com/blog/2017/01/functors-applicatives-and-monads`, 2017

Schmidt-Schauß, Manfred: *Skript zur Vorlesung „Einführung in die Funktionale Programmierung", WS 2009/10*. Institut für Informatik. Fachbereich Informatik und Mathematik : http://www.ki.informatik.uni-frankfurt.de/WS2009/EFP, 2009

Schmidt-Schauß, Manfred ; Schütz, Marko ; Sabel, David: Safety of Nöcker's Strictness Analysis. In: *Journal of Functional Programming* 18 (2008), Nr. 4, S. 503–551

Thompson, Simon: *Haskell: The Craft of Functional Programming*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999. – ISBN 0201342758

Wadler, Philip: Comprehending Monads. In: *Mathematical Structures in Computer Science* 2 (1992), Nr. 4, S. 461–493

Wadler, Philip: Monads for Functional Programming. In: Jeuring, Johan (Hrsg.) ; Meijer, Erik (Hrsg.): *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text* Bd. 925, Springer, 1995 (Lecture Notes in Computer Science), S. 24–52

Wadler, P. ; Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1989 (POPL '89). – ISBN 0–89791–294–2, 60–76