Fortgeschrittene Funktionale Programmierung

Wintersemester 2021/22

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67
80538 München
Email: david.sabel@ifi.lmu.de

Stand: 4. Februar 2022

Dank: Dieses Skript basiert unter anderem auf einem Skript von Manfred Schmidt-Schauß (Goethe-Universität Frankfurt) zur Funktionalen Programmierung und dem Material von Steffen Jost zur gleichnamigen Veranstaltung aus dem Wintersemester 2018/19 an der LMU München. Beiden danke ich für die Überlassung des Materials und die Genehmigung dieses hier zu verwenden.

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

  `summe` *liste* =
  if „*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 (Hughes, 1989).

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

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. Inhalte

Wir behandeln vertiefende Themen der Funktionalen Programmierung und Funktionaler Programmiersprachen. Einerseits werden theoretische Grundlagen behandelt (z.B. Syntax, Semantik und Typisierung von (Kern-)sprachen funktionaler Programmiersprachen) und andererseits werden fortgeschrittene Techniken der funktionaler Programmierung behandelt (z.B. Behandlung von I/O und Effekten, Nebenläufige und Parallele Programme, Testen und Verifikation, Entwicklung ereignisgesteuerter Anwendungen wie Webapplikationen und graphischer Benutzeroberflächen).

1.5. Weiteres Material

1.5.1. 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.

1.5.2. 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 von Haskell

In diesem Kapitel werden wir die verschiedenen Konstrukte der funktionalen Programmiersprache Haskell erörtern.

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. 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. Wir empfehlen eine Installation mittels `stack` (siehe http://haskellstack.org). Neben dem Übersetzen in ausführbaren Code, kann der GHC auch als Interpreter verwendet werden (GHCi). Dort kann man direkt Ausdrücke eingeben und auswerten lassen:

```
> stack exec -- ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> 1+2
3
Prelude> 10 * (2 + 4)
60
Prelude> 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 separate 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-Programm* 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 Allgmeinen 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

```
Prelude> :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) in Haskell ist

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

wobei TV für eine Typvariable steht (die mit einem Kleinbuchstaben beginnen muss) und TC ein Typkonstruktor mit Stelligkeit n ist (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. jedes 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.

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 4). 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

```
Prelude> (True || False) && (not True)
False
Prelude> True == False
False
Prelude> 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. Algebraische Datentypen in Haskell

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

2.3.1. 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 `|` $\ldots$ `|` 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 2.3.1. *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) Pattern links in einer Funktionsdefinition zu benutzen und die einzelnen Fälle durch mehrere Definitionsgleichungen abzuarbeiten z.B.

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

Hierbei ist _ eine namenslose Variable („Wildcard“), die wie eine Variable wirkt aber rechts nicht benutzt werden kann.

2.3.2. Produkttypen

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

Definition 2.3.2 (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)`.

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

Prelude> :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 Pattern auf der linken Seite der Funktionsdefinition:

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

Für Produkttypen bietet Haskell noch die so genannte *Record*-Syntax, die wir im nächsten Abschnitt behandeln werden.
Zuvor sei noch erwähnt, dass man Aufzählungstypen und Produkttypen verbinden kann, z.B. kann man einen Datentyp für 3D-Objekte definieren:

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

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*.

2.3.2.1. 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!

2.3.3. Parametrisierte Datentypen

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

```
data Typname par_1 ... par_m = Konstruktor_1 arg_{1,1}... arg_{1,i_1} | ... | Konstruktor_n arg_{n,1} ... arg_{n,i_n}
   deriving (class_1, ..., 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 4) erstellen zu lassen.

2.3.3.1. 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]
```

2.3.3.2. 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:

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

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

2.3.4. 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 Pattern, 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. 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 4). Zur Verständlichkeit geben wir in diesem Abschnitt die spezielleren Typen an, die sich nur auf Listen beziehen.

2.4.1. 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
```

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

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

Hierbei sind `guard`$_1$ bis `guard`$_n$ Boolesche Ausdrücke, die die Variablen der Pattern `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. Im Beispiel oben haben wir den Guard `otherwise` verwendet, dieser ist nur ein Synonym für `True`, d.h.

```
otherwise = True
```

ist vordefiniert.

Ebenso haben wir in obigem Programm das `where`-Konstrukt von Haskell verwendet. Damit können nachgestellte Bindungen (oder auch Funktionen) definiert werden. Im Unterschied zu `let`...`in`... formt `where` keinen Ausdruck, sondern ist ein syntaktisches Konstrukt, dass sich auf die aktuelle Funktions-Definition einschließlich der aktuellen Pattern bezieht. In obigem Fall könnte auch ein `let`-Ausdruck verwendet werden:

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

```
    test val
     | delta >= 0 = val > end
     | otherwise  = val < end
 in go start
```

Wir werden später nochmal genauer auf `let`-Ausdrücke und `where`-Klauseln eingehen.

2.4.2. 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`).

2.4.3. Standard-Listenfunktionen

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

2.4.3.1. 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).

2.4.3.2. 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
```

2.4.3.3. 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]
Prelude>  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"
```

2.4.3.4. 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)
```

2.4.3.5. 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
```

2.4.3.6. 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.
```

2.4.3.7. 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,^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"
```

2.4.3.8. 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:

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

2.4.3.9. 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:

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

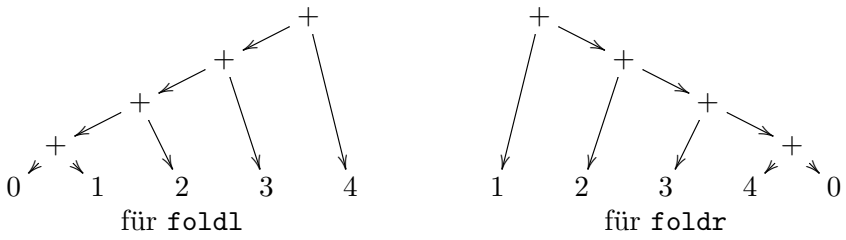
```
vectorAdd :: (Num a) => [a] -> [a] -> [a]
vectorAdd = zipWith (+)
```

2.4.3.10. 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 rechts-neutral ist), allerdings wäre dies ineffizienter, da die Laufzeit von `++` linear in der linken Liste ist.

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

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

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

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

Diese Variante verbraucht nur konstanten Platz.

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

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

Die zum Falten genutzte Funktion bildet konstant auf das zweite Argument ab. Der `foldl`-Aufruf ergibt daher zunächst `(\x y -> y) ((\x y -> y) 0 undefined) 1`. Anschließend wertet die verzögerte Auswertung den Ausdruck direkt zu `1` aus. Der `foldl'`-Aufruf wertet jedoch durch die Erzwingung mit `seq` zunächst das Zwischenergebnis `((\x y -> y) 0 undefined)` aus. Da dieser Ausdruck jedoch nicht erfolgreich ausgeweretet werden kann, terminiert die gesamte Auswertung des `foldl'`-Ausrufs nicht[2].

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]`.

2.4.3.11. 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`[3] 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`:

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

```
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]
```

2.4.3.12. 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:

```
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.

2.4.3.13. Listen als Ströme

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

Ströme in Ströme verwandeln und bei den `take` n (f *list*) für jede Eingabeliste *list* und jedes n terminiert, nennt man *strom-produzierend*.

Wir sehen das etwas weniger restriktiv und akzeptieren auch solche Funktionen, die nur für fast alle Eingabeströme diese Eigenschaft haben.

Von den vorgestellten Funktionen eignen sich zur Stromverarbeitung z.B. `map`, `filter`, `zip` und `zipWith` (zur Verarbeitung mehrerer Ströme), `take`, `drop`, `takeWhile`, `dropWhile`. Für Strings gibt es die nützlichen Funktionen `words :: String -> [String]` (Zerlegen einer Zeichenkette in eine Liste von Wörtern), `lines :: String -> [String]` (Zerlegen einer Zeichenkette in eine Liste der Zeilen), `unlines :: [String] -> String` (einzelne Zeilen in einer Liste zu einem String zusammenfügen (mit Zeilenumbrüchen)).

Übungsaufgabe 2.4.1. *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,..
```

2.4.3.14. 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
```

2.4.3.15. 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]
```

2.4.4. 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[4] – 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:

[4] nach der Zermelo-Fränkel Mengenlehre

```
[ 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:
  ```
  Prelude> take 10 [(x,y) | x <- [1..], y <- [1..]]
  [(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)]
  ```
- `[x | x <- [1..], odd x]` erzeugt die Liste aller ungeraden natürlichen Zahlen.
- `[x*x | x <- [1..]]` erzeugt die Liste aller Quadratzahlen.
- `[(x,y)| x <- [1..], let y = x*x]` erzeugt die Liste aller Paare (Zahl,Quadrat der Zahl).
- `[a | (a,_,_,_) <- [(x,x,y,y) | x <- [1..3], y <- [1..3]]]` 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 2.4.2. `[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.

2.5. Bäume

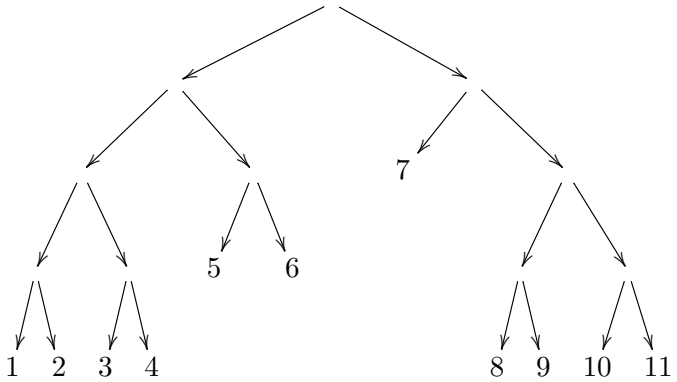
Bäume können wie Listen durch rekursive Datentypen definiert werden. Wir betrachten verschiedene Varianten.

2.5.1. 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
```

2.5.2. 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.

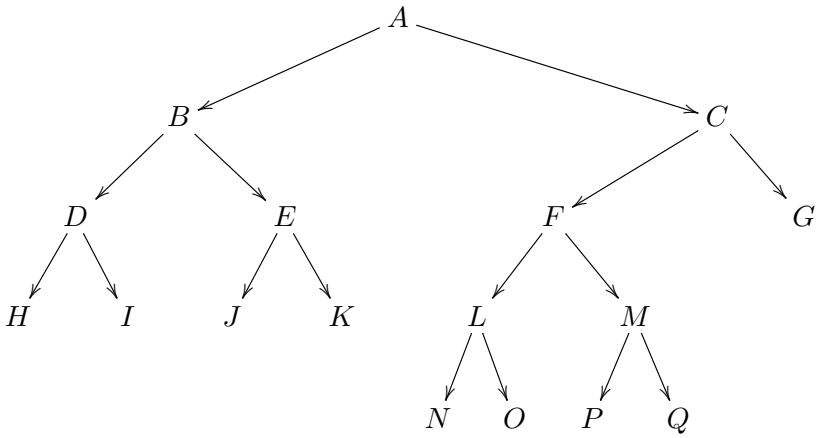
Übungsaufgabe 2.5.1. *Definiere Funktionen* `nRand`, `nFold`, `nMap` *für n-äre Bäume.*

2.5.3. 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), in-order (linker Teilbaum, Wurzel, rechter Teilbaum), post-order (linker Teilbaum, rechter Teilbaum, Wurzel), level-order (Knoten stufenweise, wie bei Breitensuche) berechnen:

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

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

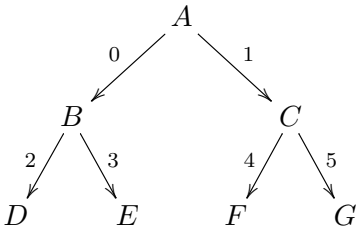
Übungsaufgabe 2.5.2. *Geben Sie einen Datentyp in Haskell für n-äre Bäume mit Markierung aller Knoten an. Implementieren Sie die Berechnung aller Knotenmarkierungen als Liste in pre-order, in-order, post-order und level-order Reihenfolge.*

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'))
```

2.5.4. 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):

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

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

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

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

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

Anstelle der Präfix-Konstruktoren `Plus` und `Mult` verwendet diese Variante die Infix-Konstruktoren `:+:` und `:*:`. Der arithmetische Ausdruck $(3 + 4) * (5 + (6 + 7))$ wird dann als Objekt vom Typ `ArEx` dargestellt durch: `((Zahl 3) :+: (Zahl 4)) :*: ((Zahl 5) :+: ((Zahl 6) :+: (Zahl 7)))`.
Man kann nun z.B. einen Interpreter für arithmetische Ausdrücke sehr leicht implementieren, indem man die einzelnen Fälle (d.h. verschiedenen Konstruktoren) durch Patter-Matching abarbeitet:

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

Z.B. ergibt:

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

2.6. 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.

2.7. Funktionen und Ausdrücke

2.7.1. Funktionen und Funktionstypen

Funktionen werden 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
```

Man kann auch Infix-Datenkonstruktoren definieren. Diese müssen mit einem `:` beginnen. Z.B. führt die folgende Definition die Konstruktoren `:||` und `:&&` zur Repräsentation von Konjunktion und Disjunktion ein:

```
data Formula = Val Bool | Formula :|| Formula | Formula :&& Formula | Not Formula
```

Definition 2.7.1 (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* Zielbereich*. 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.
Alle Funktionen bilden einen Argumenttyp auf einen Ergebnistyp ab, allerdings dürfen diese beiden Typen algebraische Datentypen oder auch Funktionstypen sein:

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

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

In Haskell darf man partiell anwenden. Z.B. ist die partielle Anwendung — `(bar 1)` eine Funktion des Typs `Int -> (Int,Int)`

Funktionen sind also normale Werte in einer funktionalen Sprache.

Eine Funktionsdefinition in Haskell ist 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, ..., var_n die formalen Parameter sind und *expr* der Funktionsrumpf.

Anstelle von Variablen können auch Pattern verwendet werden, es können mehrere Definitionszeilen angegeben werden (was z.B. sinnvoll ist, um mit Pattern eine Fallunterscheidung durchzuführen), und mithilfe von Guards können Bedingungen geprüft werden, anhand derer sich der jeweils gültige Funktionsrumpf bestimmt. 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

```
second (x:y:xs) = y
second _ = error "not enough elements"
```

die Funktion, die das zweite Element einer Liste zurückgibt, sofern dieses existiert und anderenfalls einen Fehler erzeugt. Vertauschen der Zeilen, d.h.

```
second' _ = error "not enough elements"
second' (x:y:xs) = y
```

ist nicht korrekt: Die Funktion `second'` wird immer mit einem Fehler enden, da das Pattern der ersten Zeile (die Wildcard `_`) immer passt. D.h. die zweite Zeile wird niemals erreicht. Hingegen ist die Variante

```
second'' []       = error "not enough elements"
second'' [x]      =  error "not enough elements"
second'' (x:y:xs) = y
```

korrekt, da die beiden ersten Zeilen nur passende Pattern für leere und einelementige Listen liefern.

2.7.2. 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
```

2.7.3. 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*

Wobei *Alternativen* `case`-Alternativen der Form *Pattern* `->` *Ausdruck* oder *Pattern Match* sind. In *Pattern* sind auch verschachtelte Pattern 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 *Pattern 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.

- `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.

- Datenkonstruktoren, Listen, List-Comprehensions

2.7.4. Programmieren mit Let-Ausdrücken in Haskell

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

2.7.4.1. Lokale Funktionsdefinitionen mit `let`

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

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

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

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

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

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

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

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

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.

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

2.7.4.2. Pattern-Matching mit `let`

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

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

2.7.4.3. Memoization

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

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

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

```
-- Fibonacci mit Memoization

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

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

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

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

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

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

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

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

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

2.7.4.4. `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:

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

Allerdings ist zu beachten, dass (`let` ... `in` e) einen Ausdruck darstellt, während e `where`... kein Ausdruck ist. 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.

2.8. 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.

2.8.1. 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

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

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[6].

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:

[6] Bei hierarchischen Modulen muss der Dateipfad dem Modulnamen entsprechen, siehe Abschnitt 2.8.4)

Beispiel 2.8.1. *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
```

2.8.2. 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 2.8.1 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 2.8.1 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 2.8.1 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 Kons¸truktor `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 2.8.1 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`.

2.8.3. Modulimport

Die exportierten Definitionen eines Moduls können mittels der `import`-Anweisung in ein anderes Modul importiert werden. Diese steht am Anfang des Modulrumpfs. In einfacher Form geschieht dies durch

`import` *Modulname*

Durch diese Anweisung werden sämtliche Einträge der Exportliste vom Modul mit dem Namen *Modulname* importiert, d.h. sichtbar und verwendbar.

Will man nicht alle exportierten Namen in ein anderes Modul importieren, so ist dies auf folgende Weisen möglich:

Explizites Auflisten der zu importierenden Einträge: Die importierten Namen werden in Klammern geschrieben aufgelistet. Die Einträge werden hier genauso geschrieben wie in der Exportliste.

Z.B. importiert das Modul

```
module Game where
import Spiel(berechneErgebnis, Ergebnis(..))
...
```

nur die Funktion `berechneErgebnis` und den Datentyp `Ergebnis` mit seinen Konstruktoren, nicht jedoch die Funktionen `istSieg` und `istNiederlage`.

Explizites Ausschließen einzelner Einträge: Einträge können vom Import ausgeschlossen werden, indem man das Schlüsselwort `hiding` gefolgt von einer Liste der ausgeschlossenen Einträge benutzt.

Den gleichen Effekt wie beim expliziten Auflisten können wir auch im Beispiel durch Ausschließen der Funktionen `istSieg` und `istNiederlage` erzielen:

```
module Game where
import Spiel hiding(istSieg,istNiederlage)
...
```

Die importierten Funktionen sind sowohl mit ihrem (unqualifizierten) Namen ansprechbar, als auch mit ihrem qualifizierten Namen: *Modulname*`.`*unqualifizierter Name*, manchmal ist es notwendig den qualifizierten Namen zu verwenden, z.B.

```
module A(f) where
f a b = a + b

module B(f) where
```

```
f a b = a * b

module C where
import A
import B
g = f 1 2 + f 3 4 -- funktioniert nicht
```

führt zu einem Namenskonflikt, da `f` mehrfach (in Modul `A` und `B`) definiert wird.

```
Prelude> :l C.hs
[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

```
Prelude> :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`

2.8.4. 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[7]“ sind mithilfe der hierarchischen Modulstruktur aufgebaut, z.B. sind Funktionen, die auf Listen operieren, im Modul `Data.List` definiert.

2.9. Quellennachweise und weiterführende Literatur

Die vorgestellte Übersicht in Haskell ist in den meisten Haskell-Büchern zu finden (siehe Kapitel 1). Formal definiert sind sie im Haskell-Report ((Peyton Jones, 2003) bzw. (Marlow, 2010)). Die Dokumentation der Standard-Implementierungen findet man unter http://www.haskell.org/ghc/docs/latest/html/libraries, wobei von besonderer Bedeutung die Bibliothek `Data.List` ist. Das hierachische Modulsystem ist seit Haskell 2010 Teil des Standards.

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

3

Semantik funktionaler Programmiersprachen: Funktionale Kernsprachen

In diesem Kapitel werden verschiedene Kernsprachen (für Haskell) erörtert. Wir gehen dabei modular vor, d.h. wir beginnen mit einfachen Sprachen und erweitern diese schrittweise. Am Ende des Kapitels in Abschnitt 3.8 gehen wir auch auf die in Haskell verwendete Call-by-Need-Auswertung ein, aber verzichten dabei auf die formale Semantik, sondern machen dies eher informell, um eine Vorstellung dieser Auswertung zu vermitteln. Kernsprachen werden auch oft als *Kalküle* bezeichnet. Sie bestehen im Wesentlichen aus der Syntax, die festlegt welche Ausdrücke in der Sprache gebildet werden dürfen und der Semantik, die angibt, welche Bedeutung die Ausdrücke haben. Diese Kalküle kann man als abgespeckte Programmiersprachen auffassen. „Normale“ Programmiersprachen werden oft während des Compilierens in solche einfacheren Sprachen übersetzt, da diese besser überschaubar sind und sie u.a. besser mathematisch handhabbar sind. Mithilfe einer Semantik kann man (mathematisch korrekte) Aussagen über Programme und deren Eigenschaften nachweisen. Das Gebiet der „Formalen Semantiken“ unterscheidet im Wesentlichen drei Ansätze:

- Eine *axiomatische Semantik* beschreibt Eigenschaften von Programmen mithilfe logischer Axiome bzw. Schlussregeln. Weitere Eigenschaften von Programmen können dann mithilfe von logischen Schlussregeln hergeleitet werden. Im Allgemeinen werden von axiomatischen Semantiken nicht alle Eigenschaften, sondern nur einzelne Eigenschaften der Programme erfasst. Ein prominentes Beispiel für eine axiomatische Semantik ist der Hoare-Kalkül (Hoare, 1969).
- *Denotationale Semantiken* verwenden mathematische Räume um die Bedeutung von Programmen festzulegen. Eine *semantische Funktion* bildet Programme in den entsprechenden mathematischen Raum ab. Für funktionale Programmiersprachen werden als mathematische Räume oft „Domains“, d.h. partiell geordnete Mengen, verwendet. Denotationale Semantiken sind mathematisch elegant, allerdings für umfangreichere Sprachen oft schwer zu definieren.
- Eine *operationale Semantik* legt die Bedeutung von Programmen fest, indem sie definiert wie Programme *ausgewertet* werden. Es gibt verschiedene Möglichkeiten operationale Semantiken zu definieren: *Zustandsübergangssysteme* geben an, wie sich der Zustand der Maschine (des Speichers) beim Auswerten verändert, *Abstrakte Maschinen* geben ein Maschinenmodell zur Auswertung von Programmen an und *Ersetzungssysteme* definieren, wie der Wert von Programmen durch Term-Ersetzungen „ausgerechnet“ werden kann. Man kann operationale Semantik noch in *big-step* und *small-step*-Semantiken unterscheiden. Während small-step-Semantiken die Auswertung in kleinen Schritten festlegen, d.h. Programme werden schrittweise ausgewertet, legen big-step-Semantik diese Auswertung in größeren (meist einem) Schritt fest. Oft erlauben big-step-Semantiken Freiheit bei der Implementierung eines darauf basierenden Interpreters, während bei small-step-Semantiken

meist ein Interpreter direkt aus den Regeln ablesbar ist.

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

3.1. Der Lambda-Kalkül

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

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

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

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

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

Konvention 3.1.1. *Um Klammern zu sparen legen wir die folgenden Assoziativitäten und Prioritäten fest: Die Anwendung ist links-assoziativ, d.h. $s\ t\ r$ entspricht $((s\ t)\ r)$ und* nicht *$(s\ (t\ r))$. Der Rumpf einer Abstraktion erstreckt sich soweit wie möglich, z.B. entspricht $\lambda x.s\ t$ dem Ausdruck $\lambda x.(s\ t)$ und* nicht *dem Ausdruck $((\lambda x.s)\ t)$. Als weitere Abkürzung verwenden wir die Schreibweise $\lambda x_1, \ldots, x_n.t$ für die geschachtelten Abstraktionen $\lambda x_1.(\lambda x_2. \ldots (\lambda x_n.t) \ldots)$.*

Beispiel 3.1.2. *Einige prominente Ausdrücke des Lambda-Kalküls sind:*

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

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

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

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

Beispiel 3.1.3. *Sei s der Ausdruck $(\lambda x.\lambda y.\lambda w.(x\ y\ z))\ x$. Dann ist gilt $FV(s) = \{x, z\}$ und $BV(s) = \{x, y, w\}$.*

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

Beispiel 3.1.4. *Sei s der Ausdruck $(\lambda x.\lambda y.\lambda w.(x\ y\ z))\ x$. Wir markieren die Vorkommen der Variablen (nicht an den Bindern): $(\lambda x.\lambda y.\lambda w.(\underbrace{x}_{1}\ \underbrace{y}_{2}\ \underbrace{z}_{3}))\ \underbrace{x}_{4}$. Das Vorkommen von x markiert mit 1 und das Vorkommen von y markiert mit 2 sind gebundene Vorkommen von Variablen. Das Vorkommen von z markiert mit 3 und das Vorkommen von x markiert mit 4 sind freie Vorkommen von Variablen.*

Übungsaufgabe 3.1.5. *Sei s der Ausdruck $(\lambda y.(y\ x))\ (\lambda x.(x\ y))\ (\lambda z.(z\ x\ y))$. Berechne die freien und gebundenen Variablen von s. Welche der Vorkommen von x, y, z sind frei, welche sind gebunden?*

Definition 3.1.6. *Um die operationale Semantik des Lambda-Kalküls zu definieren, benötigen wir den Begriff der* Substitution*: Wir schreiben $s[t/x]$ für den Ausdruck, der entsteht, indem alle freien Vorkommen der Variable x in s durch den Ausdruck t ersetzt werden. Um Namenskonflikte bei dieser Ersetzung zu vermeiden, nehmen wir an, dass $BV(s) \cap FV(t) = \emptyset$ gilt. Unter diesen Annahmen kann man die Substitution formal durch die folgenden Gleichungen definieren:*

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

[1] Beachte, dass wir eigentlich noch keinen Gleichheitsbegriff für den Lambda-Kalkül definiert haben. Syntaktische Gleichheit gilt für $Y\ f$ und $f\ (Y\ f)$ nicht. In einem späteren Abschnitt werden wir einen Gleichheitsbegriff einführen.

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

Übungsaufgabe 3.1.7. *Sei $s = (x\ z)\ (\lambda y.x)$ und $t = \lambda w.(w\ w)$. Welchen Ausdruck erhält man für $s[t/x]$?*

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

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

Hierbei seien die Produktionen für **Expr** wie zuvor definiert.

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

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

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

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

Konvention 3.1.8 (Distinct Variable Convention)**.** *In einem Ausdruck haben alle gebundenen Variablen unterschiedliche Namen die Namen gebundener Variablen sind stets verschieden von Namen freier Variablen.*

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

Beispiel 3.1.9. *Betrachte den Ausdruck $(y\ (\lambda y.((\lambda x.(x\ x))\ (x\ y))))$. Er erfüllt die DVC nicht, da x und y sowohl frei als auch gebunden vorkommen. Benennt man die gebundenen Variablen mit frischen Namen um (das sind alles α-Umbenennungen), erhält man*

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

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

Übungsaufgabe 3.1.10. *Sei $s = ((\lambda x.(x\ \lambda y.(x\ z)\ (y\ x)))\ (\lambda z.y))$. Führe α-Umbenennungen für s durch, so dass der entstehende Ausdruck die DVC erfüllt.*

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

Definition 3.1.11. *Wenn $BV(s) \cap FV(t) = \emptyset$, dann definiert man $s[t/x]$ wie in Definition 3.1.6. Wenn die Bedingung nicht erfüllt ist, dann sei $s' =_\alpha s$ eine Umbenennung von s, so dass $BV(s') \cap FV(t) = \emptyset$. Ein solches s' gibt es! Dann definiere $s[t/x]$ als $s'[t/x]$ entsprechend Definition 3.1.6.*

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

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

Wenn $r_1 \xrightarrow{\beta} r_2$, so sagt man auch r_1 *reduziert unmittelbar* zu r_2.

Beispiel 3.1.12. *Den Ausdruck* $(\lambda x.x)\ (\lambda y.y)$ *kann man* β*-reduzieren:*

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

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

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

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

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

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

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

3.1.1. Call-by-Name-Reduktion

Die *call-by-name Auswertung* sucht immer den am weitesten oben und am weitesten links stehenden Redex. Formal kann man die Call-by-Name-Reduktion mithilfe von Reduktionskontexten definieren.

Definition 3.1.13. *Reduktionskontexte* R *werden durch die folgende Grammatik mit Startsymbol* **RCtxt** *generiert (mit den Produktionen für* **Expr** *wie zuvor):*

$$\textbf{RCtxt} ::= [\cdot] \mid (\textbf{RCtxt}\ \textbf{Expr})$$

Wenn $r_1 \xrightarrow{\beta} r_2$ *und* R *ein Reduktionskontext ist, dann ist* $R[r_1] \xrightarrow{name} R[r_2]$ *ein* Call-by-Name Reduktionsschritt.

Übungsaufgabe 3.1.14. *Sei* $s = (\lambda w.w)\ (\lambda x.x)\ ((\lambda y.((\lambda u.y)\ y))\ (\lambda z.z))$*. Gebe alle Reduktionskontexte* R *und Ausdrücke* t *an für die gilt:* $R[t] = s$*. Führe einen Call-by-Name-Reduktionsschritt für* s *aus.*

Bemerkung 3.1.15. *Wenn man geschlossene Ausdrücke mittels Call-By-Name-Reduktion reduziert, dann kann man im Prinzip auf die Vorsichtsmaßnahme der Variablen-Umbenennung verzichten, denn eine Einfangen von freien Variablen ist dann nicht möglich. Aber: wenn man innerhalb eines Ausdrucks eine Nicht-Call-by-Name-Reduktion macht, dann braucht man i.a. die Vorsichtsmaßnahmen der Variablenumbenennung, um eine korrekte Beta-Reduktion durchzuführen.*

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

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

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

- s_1 ist eine Abstraktion $\lambda x.s_1'$: Falls $n \geq 2$ dann reduziere s wie folgt: $(\lambda x.s_1')\ s_2\ \ldots s_n \xrightarrow{name} (s_1'[s_2/x]\ \ldots s_n)$. Falls $n = 1$, dann ist keine Reduktion möglich, da der Gesamtausdruck eine Abstraktion ist.
- s_1 ist eine Variable. Dann ist keine Reduktion möglich: eine freie Variable wurde entdeckt. Dieser Fall kommt bei geschlossenen Ausdrücken nicht vor.

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

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

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

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

Da dieser erste Auswertungsschritt mit einer Abstraktion endet, ist die Call-by-Name-Reduktion erfolgreich beendet.

Übungsaufgabe 3.1.16. *Werte die folgenden Ausdrücke mit Call-by-Name-Auswertung aus:*

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

Eine Eigenschaft der Call-by-Name-Reduktion ist, dass diese stets *deterministisch* ist, d.h. für einen Ausdruck s gibt es höchstens einen Ausdruck t, so dass $s \xrightarrow{name} t$. Es gibt auch Ausdrücke für die keine Reduktion möglich ist. Dies sind zum Einen Ausdrücke, bei denen die Auswertung auf eine freie Variable stößt (z.B. ist $(x\ (\lambda y.y))$ nicht reduzibel). Zum Anderen sind dies Abstraktionen, die auch als FWHNFs (functional weak head normal forms, funktionale schwache Kopfnormalformen) bezeichnet werden. D.h. sobald wir eine FWHNF mithilfe von $\xrightarrow{name}$-Reduktionen erreicht haben, ist die Auswertung beendet. Ausdrücke, die man so in eine Abstraktion überführen kann, *konvergieren* (oder alternativ *terminieren*). Wir definieren dies formal, wobei $\xrightarrow{name,+}$ die transitive Hülle von $\xrightarrow{name}$ und $\xrightarrow{name,*}$ die reflexiv-transitive Hülle

von $\xrightarrow{name}$ sei[2]

Definition 3.1.17. *Ein Ausdruck s (call-by-name)* konvergiert *genau dann, wenn es eine Folge von call-by-name Reduktionen gibt, die s in eine Abstraktion überführt, wir schreiben dann $s\Downarrow$. D.h. $s\Downarrow$ gdw. $\exists$Abstraktion $v : s \xrightarrow{name,*} v$. Falls s nicht konvergiert, so schreiben wir $s\Uparrow$ und sagen s* divergiert.

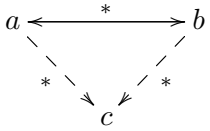
Haskell hat die call-by-name Auswertung als semantische Grundlage, wobei Implementierungen die verbesserte Strategie der call-by-need Auswertung verwenden, die Doppelauswertungen von Argumenten vermeidet (siehe Abschnitt 3.1.4). Der Lambda-Kalkül selbst ist jedoch syntaktisch zu eingeschränkt, um als Kernsprache für Haskell zu dienen, wir werden später erweiterte Lambda-Kalküle betrachten, die sich besser als Kernsprachen für Haskell eignen. Für die Call-by-Name-Reduktion gilt die folgende Eigenschaft:

Satz 3.1.18. *[Standardisierungseigenschaft der Call-by-Name-Auswertung] Sei s ein Lambda-Ausdruck. Wir nehmen an, dass s mit beliebigen β-Reduktionen (an beliebigen Positionen) in eine Abstraktion v überführt werden. Dann gilt $s\Downarrow$.*

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

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

Theorem 3.1.19 (Church-Rosser Theorem)**.** *Für den Lambda-Kalkül gilt (bzgl. der Gleichheit $=_\alpha$:)*
Wenn $a \overset{*}{\leftrightarrow} b$, dann existiert c, so dass $a \xrightarrow{*} c$ und $b \xrightarrow{*} c$



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

3.1.2. Implementierung eines Interpreters für den ungetypen Lambda-Kalkül

Wir betrachten eine Implementierung des ungetypten Lambda-Kalküls. Als Datentyp für Lambda-Kalkül-Ausdrücke kann man definieren:

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

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

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

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

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

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

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

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

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

```
-- substitute freshvars expr1 expr2 var
-- f"uhrt die Ersetzung expr1[expr2/var] durch
substitute  :: (Eq b) => [b] -> LExp b -> LExp b -> b -> (LExp b, [b])

substitute freshvars (Var v) expr2 var
```

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

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

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

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

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

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

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

-- Andere Faelle: Keine Reduktion m"oglich:
tryNameBeta _ vars = Nothing
```

Schließlich kann man die Call-by-Name-Auswertung implementieren, indem man `tryNameBeta` solange anwendet bis keine Reduktion mehr möglich ist:

```
tryName e vars = case tryNameBeta e vars of
                   Nothing -> e
                   Just (e',vars') -> tryName e' vars'
```

Die Hauptfunktion ruft nun `tryName` mit einer Liste von frischen Variablennamen auf und benennt zur Sicherheit die Eingabe vor der ersten Reduktion um:

```
reduceName e = let (e',v') = rename e fresh
               in  tryName e' v'
      where fresh = ["x_" ++ show i | i <- [1..]]
```

Seien

```
example_id  = (Lambda "v" (Var "v"))
example_k   = (Lambda "x" (Lambda "y" (Var "x")))
```

dann betrachte die folgenden Beispiele:

```
*Main> reduceName example_id
Lambda "x_1" (Var "x_1")
*Main> reduceName (App example_id example_id)
Lambda "x_3" (Var "x_3")
*Main> reduceName (App example_k example_id)
Lambda "x_2" (Lambda "x_4" (Var "x_4"))
```

3.1.3. Call-by-Value-Reduktion

Wir betrachten eine weitere wichtige Auswertungsstrategie. Die *Call-by-value Auswertung* oder *strikte Auswertung*) verlangt, dass eine β-Reduktion nur dann durchgeführt werden darf, wenn das Argument eine Abstraktion ist. Wir definieren hierfür die β_{value}-Reduktion als:

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

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

Definition 3.1.20. *Call-by-value Reduktionskontexte E werden durch die folgende Grammatik mit Startsymbol* **ECtxt** *generiert (mit den Produktionen für* **Expr** *wie zuvor):*

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

Wenn $r_1 \xrightarrow{\beta_{value}} r_2$ und E ein Call-by-Value-Reduktionskontext ist, dann ist $E[r_1] \xrightarrow{value} E[r_2]$ eine Call-by-Value-Reduktion.

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

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

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

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

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

Die erste Reduktion erfolgt daher im Argument:

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

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

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

Nun hat man eine Abstraktion erhalten, und die Auswertung in Call-by-Value-Reihenfolge damit erfolgreich beendet.

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

Definition 3.1.21. *Ein Ausdruck s* konvergiert in Call-by-Value-Reihenfolge *genau dann, wenn es eine Folge von Call-by-Value-Reduktionen gibt, die s in eine Abstraktion überführt, wir schreiben dann $s{\Downarrow_{value}}$. D.h. $s{\Downarrow_{value}}$ gdw. $\exists$Abstraktion $v : s \xrightarrow{value,*} v$. Falls s nicht konvergiert, so schreiben wir $s{\Uparrow_{value}}$ und sagen s* divergiert in Call-by-Value-Reihenfolge.

Satz 3.1.18 zeigt sofort: $s{\Downarrow_{value}} \implies s{\Downarrow}$. Die Umkehrung gilt nicht, da es Ausdrücke gibt, die in Call-by-Value-Reihenfolge divergieren, bei Auswertung in Call-by-Name-Reihenfolge jedoch konvergieren:

Beispiel 3.1.22. *Betrachte den Ausdruck $\Omega := (\lambda x.x\ x)\ (\lambda x.x\ x)$. Es lässt sich leicht nachprüfen, dass $\Omega \xrightarrow{name} \Omega$ als auch $\Omega \xrightarrow{value} \Omega$. Daraus folgt sofort, dass $\Omega{\Uparrow}$ und $\Omega{\Uparrow_{value}}$. Betrachte nun den Ausdruck $t := ((\lambda x.(\lambda y.y))\ \Omega)$. Dann gilt $t \xrightarrow{name} \lambda y.y$, d.h. $t{\Downarrow}$. Da die Auswertung in Call-by-Value-Reihenfolge jedoch zunächst das Argument Ω auswerten muss, gilt $t{\Uparrow_{value}}$.*

Trotz dieser eher schlechten Eigenschaft der Auswertung in Call-by-Value-Reihenfolge, spielt sie eine wichtige Rolle für Programmiersprachen. Die Reihenfolge der Auswertung ist bei der Call-by-Value-Reihenfolge statisch und modular festgelegt: Wenn man z.B. eine Funktion f auf geschlossenen Ausdrücke s_1, s_2, s_3 anwendet, dann wird zunächst s_1 in Call-by-Value-Reihenfolge ausgewertet, dann s_2, dann s_3 und danach wird f auf die entsprechenden Werte angewendet. Im Gegensatz zur Call-by-Value-Reihenfolge: wenn man die Call-by-Name-Reihenfolge der Auswertung von s_i kennt, kann die Reihenfolge der Auswertung von $(f\ s_1\ s_2\ s_3)$ diese Auswertung z.B. so sein: zunächst wird s_1 ausgewertet, dann abhängig vom Ergebnis s_3 und dann s_2 oder evtl. sogar s_2 überhaupt nicht. Bei Sharing bzw call-by-need sind die Reihenfolgen noch schwerer vorherzusagen.

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

3.1.4. Call-by-Need-Auswertung

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

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

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

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

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

Statt der (β)-Reduktion werden die folgenden Reduktionsregeln verwendet, genauer jede der folgenden Reduktionen ist ein *Call-by-Need-Auswertungsschritt*, den wir mit $\xrightarrow{need}$ notieren[5]:

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

Hierbei ist R_{need} ein durch $\mathbf{R}_{need}$ erzeugter Reduktionskontext und LR ist durch **LR** erzeugt. Wir erläutern die Regeln: Anstelle der einsetzenden (β)-Reduktion wird die (*lbeta*)-Reduktion verwendet, die das Argument nicht einsetzt, sondern mit einer neuen `let`-Bindung „speichert“. Die (*cp*)-Reduktion kopiert dann eine solche Bindung, falls sie benötigt wird, allerdings muss vorher die rechte Seite der Bindung zu einer Abstraktion ausgewertet werden. Die Regeln (*llet*) und (*lapp*) dienen dazu, die `let`-Verschachtelungen zu justieren.

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

[3] Beachte: In Haskell sind allgemeinere `let`-Ausdrücke erlaubt, die u.A. rekursiv sein dürfen.
[4] Der Name **LR** spiegelt das wider, er steht für „let rechts“
[5] Die Regeln gehen dabei davon aus, dass Ausdrücke die DVC erfüllen!

(2) und Regel (3) anwendbar sind, immer Regel (2) angewendet wird.

$$
\begin{array}{ll}
(1) & (\texttt{let}\ x = s\ \texttt{in}\ t)^{\star} \Rightarrow (\texttt{let}\ x = s\ \texttt{in}\ t^{\star}) \\
(2) & (\texttt{let}\ x = C_1[y^{\diamond}]\ \texttt{in}\ C_2[x^{\circledcirc}]) \Rightarrow (\texttt{let}\ x = C_1[y^{\diamond}]\ \texttt{in}\ C_2[x]) \\
(3) & (\texttt{let}\ x = s\ \texttt{in}\ C[x^{\star \vee \diamond}]) \Rightarrow (\texttt{let}\ x = s^{\diamond}\ \texttt{in}\ C[x^{\circledcirc}]) \\
(4) & (s\ t)^{\star \vee \diamond} \Rightarrow (s^{\diamond}\ t)
\end{array}
$$

Zur Erläuterung: Regel (1) springt in den `in`-Ausdruck eines `let`-Ausdrucks. Die Markierung $\star$ wird zu $\diamond$ nachdem einmal in die Funktionsposition einer Anwendung bzw. in eine `let`-Bindung gesprungen wurde. Hierdurch wird verhindert, dass anschließend wieder `let`-Ausdrücke mit Regel (1) gesprungen werden kann. Regel (3) führt die Markierung $\circledcirc$ ein, um das Kopierziel einer evtl. (*cp*)-Reduktion zu markieren. Regel (2) verschiebt das Kopierziel, wenn eine `let`-Bindung der Form $x = C_1[y]$ gefunden wurde: In diesem Fall wird zunächst in die Bindung kopiert (y in $C_1[y]$ ersetzt).

Nach Abschluss des Markierungsalgorithmus werden die Reduktionsregeln (falls möglich) wie folgt angewendet (wir geben hier nur den Redex an, ein äußerer Kontext ist durchaus möglich):

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

Wir betrachten ein Beispiel.

Beispiel 3.1.23. *Die Call-by-Need-Auswertung (mit Markierungsalgorithmus) des Ausdrucks* $\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)\ x)$ *ist:*

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

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

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

Definition 3.1.24. *Ein Ausdruck s* call-by-need konvergiert *(geschrieben als $s{\Downarrow_{need}}$), gdw. er mit einer Folge von $\xrightarrow{need}$-Reduktionen in eine FWHNF überführt werden kann, d.h. $s{\Downarrow_{need}} \iff \exists$ FWHNF $v : s \xrightarrow{need,*} v$*

Man kann zeigen, dass sich die Call-by-Name-Reduktion und die Call-by-Need-Auswertung bezüglich der Konvergenz gleich verhalten, d.h.

Satz 3.1.25. *Sei s ein (`let`-freier) Ausdruck, dann gilt $s{\Downarrow} \iff s{\Downarrow_{need}}$.*

3.1.5. Gleichheit von Programmen

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

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

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

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

Definition 3.1.26 (Kontextuelle Approximation und Gleichheit)**.** *Für den call-by-name Lambda-Kalkül definieren wir die* Kontextuelle Approximation $\leq_c$ *und die* Kontextuelle Gleichheit $\sim_c$ *als:*

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

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

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

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

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

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

Satz 3.1.27. $\sim_c$ *und* $\sim_{c,value}$ *sind* Kongruenzen*, d.h. sie sind Äquivalenzrelationen und kompatibel mit Kontexten, d.h.* $s \sim t \implies C[s] \sim C[t]$.

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

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

ist, da man in diesem Fall nur einen Kontext angeben muss, der Ausdrücke bezüglich ihrer Konvergenz unterscheidet.

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

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

Mit dem Begriff der Kontextuellen Gleichheit werden wir uns im Rahmen dieser Veranstaltung nur wenig beschäftigen. In der Veranstaltung „M-CEFP Programmtransformationen und Induktion in funktionalen Programmen“ wird sich hiermit tiefer gehend beschäftigt.

3.1.6. Kernsprachen von Haskell

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

Bemerkung 3.1.28. *Zunächst stellt sich die Frage, warum sich der Lambda-Kalkül nicht sonderlich als Kernsprache für Haskell eignet. Hierfür gibt es mehrere Gründe:*

- *Der Lambda-Kalkül verfügt nicht über Daten: Will man Daten wie z.B. Zahlen, aber auch kompliziertere Strukturen wie beispielsweise Listen darstellen, so muss man dies durch Funktionen tun. Dies ist zwar möglich (was im Allgemeinen als Church-Kodierung bezeichnet wird), aber man kann Funktionen dann nicht von Daten unterscheiden und außerdem ist diese Methode eher schwer zu überblicken.*
- *Rekursive Funktionen: Man kann zwar im Lambda-Kalkül durch so genannte Fixpunktkombinatoren Rekursion darstellen, allerdings ist auch dies sehr unübersichtlich und die Kodierung aufwändig. Mit dem Fixpunktkombinator $Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$ (für ihn gilt $Y\ f \sim_c f\ (Y\ f)$) kann man z.B. die rekursive Fakultätsfunktion nicht-rekursiv definieren:*

  $$fak = Y\ (\lambda f.\lambda x.\texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (f\ (x-1)))$$

  Als Randbemerkung erwähnen wir, dass es im Lambda-Kalkül unendlich viele Fixpunktkombinatoren gibt, einer davon ist der folgende (siehe (Klop, 2007)):

  $$\begin{array}{lcl} Y_{Klop} & = & LLLLLLLLLLLLLLLLLLLLLLLLLL \\ L & = & \lambda a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,s,t,u,v,w,x,y,z,r. \\ & & \qquad r\,(t\,h\,i\,s\,i\,s\,a\,f\,i\,x\,e\,d\,p\,o\,i\,n\,t\,c\,o\,m\,b\,i\,n\,a\,t\,o\,r) \end{array}$$

- *Typisierung: Haskell ist eine polymorph getypte Sprache, der bisher vorgestellte Lambda-Kalkül war ungetypt.*
- *`seq`: Der in Haskell verfügbare Operator `seq` lässt sich nicht im reinen Lambda-Kalkül kodieren.*

3.2. Die Kernsprache KFPT

3.2.1. Syntax von KFPT

Als erste Erweiterung beheben wir den ersten Kritikpunkt aus Bemerkung 3.1.28 und fügen *Datenkonstruktoren* und `case`-Ausdrücke zum Lambda-Kalkül hinzu. Wir nehmen an, es gibt eine Menge von Typen, wobei jeder Typ einen Namen hat, z.B. `Bool`, `List`, ... Zu jedem Typ gibt es eine (endliche) Menge von Datenkonstruktoren, die wir im Allgemeinen mit c_i darstellen. Konkret hat z.B. `Bool` die Datenkonstruktoren `True` und `False`, und zum Typ `List` gehören die Datenkonstruktoren `Nil` und `Cons` (die in Haskell-Notation durch `[]` und `:` (infix) dargestellt werden[6]). Jeder Datenkonstruktor hat eine feste Stelligkeit $\mathrm{ar}(c_i) \in \mathbb{N}_0$ (ar kürzt dabei das englische Wort „arity“ ab). Z.B. ist $\mathrm{ar}(\texttt{True}) = \mathrm{ar}(\texttt{False}) = 0$ und $\mathrm{ar}(\texttt{Nil}) = 0$ und $\mathrm{ar}(\texttt{Cons}) = 2$. Wir fordern stets, dass Datenkonstruktoren in Ausdrücken nur *gesättigt* vorkommen, d.h. es sind stets $\mathrm{ar}(c_i)$ viele Argumente für den Konstruktor c_i angegeben[7].

Definition 3.2.1. *Die Syntax der* Kernsprache KFPT *wird durch folgende Grammatik gebildet, wobei V bzw. V_i Variablen sind:*

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

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

Das „T“ in KFPT steht für geTyptes `case`. Allerdings ist KFPT ansonsten ungetypt und kann daher auch als „schwach getypt“ bezeichnet werden. KFPT verfügt genau wie der Lambda-Kalkül über Variablen, Abstraktionen und Anwendungen, erweitert ihn jedoch um `case`-Ausdrücke (auch Fallunterscheidungen genannt) und Konstruktoranwendungen. Wir vergleichen KFPT-`case`-Ausdrücke mit Haskells `case`-Ausdrücken: Die Syntax für case-Ausdrücke in Haskell ist ähnlich, wobei $\to$ durch `->` ersetzt ist. In Haskell haben `case`-Ausdrücke keine Typmarkierung (die durch Alternativen abgedeckten Konstruktoren müssen jedoch alle vom gleichem Typ stammen). In Haskell ist es zudem nicht notwendig alle Konstruktoren eines Typs abzudecken. Man erhält dann u.U. einen Fehler zur Laufzeit, wenn keine Alternative vorhanden ist. Z.B.

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

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

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

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

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

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

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

aber das Konstrukt

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

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

$$\begin{aligned}&\texttt{case}_{\texttt{List}}\ \texttt{Nil}\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil};\\ &\qquad (\texttt{Cons}\ x\ z) \to\ \texttt{case}_{\texttt{List}}\ z\ \texttt{of}\ \{\texttt{Nil} \to \Omega; (\texttt{Cons}\ y\ ys) \to (\texttt{Cons}\ y\ \texttt{Nil})\}\end{aligned}$$

Diese Übersetzung zeigt auch, wie man nicht vorhandene `case`-Alternativen behandeln kann: Man fügt diese hinzu, wobei die rechte Seite der Alternative auf einen nicht terminierenden Ausdruck, z.B. Ω, abgebildet wird. Im weiteren benutzen wir $\bot$ (gesprochen als „bot“) als Repräsentanten für einen geschlossenen nicht terminierenden Ausdruck. Als weitere Abkürzung benutzen wir für `case`-Ausdrücke die Notation $(\texttt{case}_{Typ}\ s\ \texttt{of}\ Alts)$, wobei $Alts$ für irgendwelche – syntaktisch korrekte – `case`-Alternativen steht.

Wir geben einige KFPT-Beispielausdrücke an:

Beispiel 3.2.2. *Eine Funktion, die eine Liste erwartet, und das erste Element der Liste liefert, kann man in* KFPT *darstellen als:*

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

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

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

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

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

Beispiel 3.2.3. *In Haskell gibt es* `if-then-else`*-Ausdrücke der Form* `if` e `then` s `else` t*. Solche Ausdrücke gibt es in* KFPT *nicht, sie können jedoch durch den folgenden* `case`*-Ausdruck vollständig simuliert werden:*

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

Beispiel 3.2.4. *Paare können in* KFPT *durch einen Typ* `Paar`*, der einen zweistelligen Konstruktor* `Paar` *besitzt, dargestellt werden. Z.B. kann das Paar* (`True`,`False`) *durch* (`Paar True False`) *in* KFPT *dargestellt werden. Projektionsfunktionen fst und snd, die das erste bzw. das zweite Element eines Paares liefern, können definiert werden als Abstraktionen:*

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

Analog kann man diese Definition auf mehrstellige Tupel erweitern.

In Haskell sind Paare und Tupel bereits vorhanden (eingebaut), die Schreibweise in Haskell ist $(a_1, \ldots, a_n)$*. Beachte: In Haskell gibt es mehrstellige Tupel, aber auch 0-stellige Tupel (dargestellt als* `()`*), allerdings gibt es* keine *einstelligen Tupel*[8]*.*

3.2.2. Freie und gebundene Variablen in KFPT

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

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

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

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

Beispiel 3.2.5. *Für den* KFPT*-Ausdruck*

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

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

gilt

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

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

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

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

3.2.3. Operationale Semantik für KFPT

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

Definition 3.2.6. *Die Reduktionsregeln (β) und (case) sind in KFPT definiert als:*

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

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

c zum entsprechenden Typ gehört).

Beispiel 3.2.7. *Der Ausdruck*

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

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

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

Wenn $r_1 \to r_2$ mit einer (β)- oder (case)-Reduktion, dann sagt man r_1 *reduziert unmittelbar* zu r_2. Kontexte sind auch in KFPT Ausdrücke, die an einer Stelle ein Loch $[\cdot]$ haben, sie sind durch die folgende Grammatik mit Startsymbol **Ctxt** definiert:

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

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

Im folgenden werden wir die Call-by-Name-Reduktion für KFPT definieren. Wir führen keine weiteren Strategien (wie bspw. die strikte Auswertung) ein, diese können jedoch leicht aus den für den Lambda-Kalkül definierten Strategien abgeleitet werden.

Zur Definition einer eindeutigen Reduktionsstrategie verwenden wir (Call-by-Name-)Reduktionskontexte:

Definition 3.2.8. *Reduktionskontexte R in* KFPT *werden durch die folgende Grammatik mit Startsymbol* **RCtxt** *erzeugt:*

$$\textbf{RCtxt} ::= [\cdot] \mid (\textbf{RCtxt}\ \textbf{Expr}) \mid (\texttt{case}_{Typ}\ \textbf{RCtxt}\ \texttt{of}\ \mathit{Alts})$$

Die Call-by-Name-Reduktion ist in KFPT definiert als:

Definition 3.2.9. *Seien s und t Ausdrücke, so dass s unmittelbar zu t reduziert, dann ist $R[s] \to R[t]$ für jeden Reduktionskontext R eine* Call-by-Name-Reduktion. *Wir notieren Call-by-Name-Reduktionen mit $\xrightarrow{name}$.*

Um kenntlich zu machen, welche Reduktionsregel verwendet wurde, benutzen wir auch die Notation $\xrightarrow{name,\beta}$ bzw. $\xrightarrow{name,case}$. Des Weiteren bezeichne $\xrightarrow{name,+}$ die transitive, und $\xrightarrow{name,*}$ die reflexiv-transitive Hülle von $\xrightarrow{name}$.

Beispiel 3.2.10. *Die Reduktion $(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) \to (\lambda y.y)\ (\lambda z.z)$ ist eine Call-by-Name-Reduktion. Die Reduktion $(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) \to (\lambda x.x)\ (\lambda z.z)$ ist hingegen* keine *Call-by-Name-Reduktion, da der Kontext $(\lambda x.x)\ [\cdot]$* kein *Reduktionskontext ist.*

Die Auswertung in Call-by-Name-Reihenfolge besteht aus einer Folge von Call-by-Name-Reduktionen. Sie ist erfolgreich beendet sobald eine WHNF (schwache Kopfnormalform) erreicht ist. Wir definieren verschiedene Arten von Normalformen:

Definition 3.2.11. *Ein* KFPT-*Ausdruck s ist eine*

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

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

Definition 3.2.12 (Konvergenz, Terminierung)**.** *Ein* KFPT-*Ausdruck s* konvergiert *(oder* terminiert*, notiert als $s{\Downarrow}$) genau dann, wenn er mit Call-by-Name-Reduktionen in eine WHNF überführt werden kann, d.h.*

$$s{\Downarrow} \iff \exists\ \mathit{WHNF}\ t : s \xrightarrow{name,*} t$$

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

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

3.2.4. Dynamische Typisierung

Die Call-by-Name-Reduktion kann unter Umständen stoppen (d.h. es ist keine $\xrightarrow{name}$-Reduktion mehr anwendbar), *ohne* dass eine WHNF erreicht wurde. In diesem Fall wurde entweder eine freie Variable an Reduktionsposition gefunden, d.h. der Ausdruck ist von der Form $R[x]$, wobei R ein Reduktionskontext ist, oder es wurde ein Typfehler beim Auswerten gefunden. Da der Typfehler erst beim Auswerten entdeckt wird (Erinnerung: KFPT ist nur schwach getypt), spricht man von einem *dynamischen Typfehler*. Auf dieser Beobachtung aufbauend, kann man dynamische Typregeln für KFPT definieren:

Definition 3.2.13 (Dynamische Typregeln für KFPT)**.** *Sei s ein* KFPT-*Ausdruck. Wir sagen s ist* direkt dynamisch ungetypt*, falls s von einer der folgenden Formen ist (hierbei ist R ein beliebiger Reduktionskontext):*

- $R[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ Alts]$ *und* c *ist* nicht *vom Typ* T
- $R[\texttt{case}_T\ \lambda x.t\ \texttt{of}\ Alts]$.
- $R[(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ t]$

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

$$s \text{ ist dynamisch ungetypt} \iff \exists t : s \xrightarrow{name,*} t \wedge t \text{ ist direkt dynamisch ungetypt}$$

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

Satz 3.2.14. *Ein* geschlossener KFPT-*Ausdruck* s *ist irreduzibel (bzgl. der Call-by-Name-Reduktion) genau dann, wenn eine der folgenden Bedingungen auf ihn zutrifft:*

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

Beachte, dass obiger Satz nur über geschlossene Ausdrücke spricht. Ebenfalls sei erwähnt, dass nicht jeder geschlossene divergente Ausdruck auch dynamisch ungetypt ist: Betrachte z.B. $\Omega := (\lambda x.x)\ (\lambda x.x)$, dieser Ausdruck divergiert, da gilt $\Omega \xrightarrow{name} \Omega \xrightarrow{name} \Omega \ldots$, aber Ω ist nicht dynamisch ungetypt.

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

3.2.5. Suche nach dem Call-by-Name-Redex mit einem Markierungsalgorithmus

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

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

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

Nach Ausführen des Markierungsalgorithmus können folgende Fälle auftreten:

- Die Markierung ist an einer Abstraktion. Dann gibt es drei Fälle:
  - s selbst ist die markierte Abstraktion. Dann wurde eine WHNF erreicht, es ist keine Call-by-Name-Reduktion möglich.

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

  - Der direkte Oberterm der Abstraktion ist eine Anwendung, d.h. s ist von der Form $C[(\lambda x.s')^\star\ t]$. Dann reduziere $C[(\lambda x.s')\ t] \xrightarrow{name,\beta} C[s'[t/x]]$.
  - Der direkte Oberterm der Abstraktion ist ein `case`-Ausdruck, d.h. s ist von der Form $C[\texttt{case}_{Typ}\ (\lambda x.s')^\star\ \texttt{of}\ Alts]$. Dann ist keine Call-by-Name-Reduktion möglich. s ist direkt dynamisch ungetypt.
- Die Markierung ist an einer Konstruktoranwendung. Dann gibt es die Fälle:
  - s selbst ist die markierte Konstruktoranwendung. Dann wurde eine WHNF erreicht, es ist keine Call-by-Name-Reduktion möglich.
  - Der direkte Oberterm der Konstruktoranwendung ist ein `case`-Ausdruck, d.h. s ist von der Form $C[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ Alts]$. Es gibt zwei Fälle:
    * c ist vom Typ T, d.h. es gibt eine passende Alternative für c, dann reduziere: $C[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ \{\ldots;(c\ x_1\ \ldots\ x_n) \to t\ ;\ldots\}] \xrightarrow{name,case} C[t[s_1/x_1,\ldots,s_n/x_n]]$.
    * c ist nicht vom Typ T. Dann ist keine Call-by-Name-Reduktion möglich für s. s ist direkt dynamisch ungetypt.
  - Der direkte Oberterm der Konstruktoranwendung ist eine Anwendung, d.h. s ist von der Form $C[(c\ s_1\ \ldots\ s_n)^\star\ t]$. In diesem Fall ist keine Call-by-Name-Reduktion möglich für s. Der Ausdruck s ist direkt dynamisch ungetypt.
- Die Markierung ist an einer Variablen, d.h. s ist von der Form $C[x^\star]$. Dann ist keine Call-by-Name-Reduktion möglich, da eine freie Variable entdeckt wurde. Der Ausdruck s ist keine WHNF, aber auch nicht direkt dynamisch ungetypt.

Beispiel 3.2.15. *Die Auswertung in Call-by-Name-Reihenfolge für den Ausdruck*

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

wobei sämtliche Suchschritte zum Finden des Call-by-Name-Redexes mit angegeben sind, ist:

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

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

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

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

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

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

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

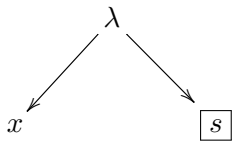
$$\Rightarrow(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^\star\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})$$

$$\begin{aligned}
&\xrightarrow{name,case}(((\lambda u, v.v)\ (\lambda w.w))\ \texttt{True})^\star\\
&\Rightarrow(((\lambda u, v.v)\ (\lambda w.w))^\star\ \texttt{True})\\
&\Rightarrow(((\lambda u, v.v)^\star\ (\lambda w.w))\ \texttt{True})\\
&\xrightarrow{name,\beta}((\lambda v.v)\ \texttt{True})^\star\\
&\Rightarrow((\lambda v.v)^\star\ \texttt{True})\\
&\xrightarrow{name,\beta}\texttt{True}
\end{aligned}$$

3.2.6. Darstellung von Termen als Termgraphen

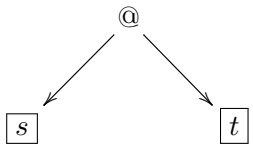
Wir erläutern kurz die Darstellung von KFPT-Ausdrücken als Baum. Jeder Knoten eines solchen Baumes stellt ein syntaktisches Konstrukt des Ausdrucks dar (von der Wurzel beginnend).

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



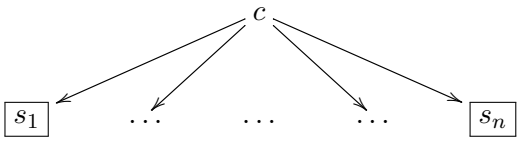
  wobei $\boxed{s}$ der Baum für s ist.
- Applikationen werden durch einen mit „@“ markierten Knoten dargestellt, der zwei Kinder für den Ausdruck an Funktionsposition und dem Argument besitzt. D.h. $(s\ t)$ wird

dargestellt durch



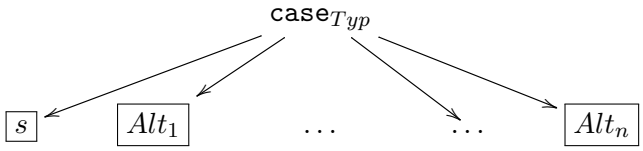
wobei $\boxed{s}$ und $\boxed{t}$ die Bäume für s und t sind.

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



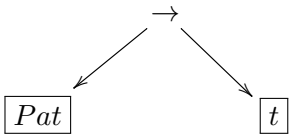
wobei $\boxed{s_i}$ die Bäume für s_i sind.

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



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

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

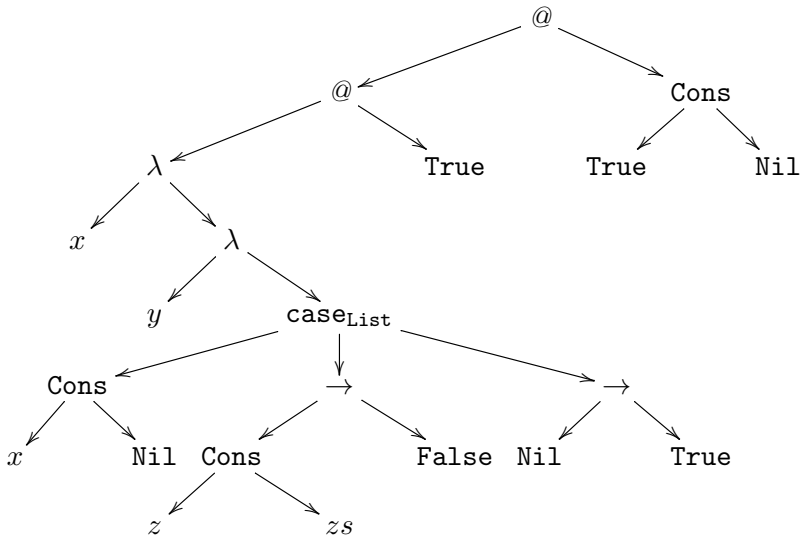


dargestellt, wobei $\boxed{Pat}$ der Baum für das Pattern und $\boxed{t}$ der Baum für t ist.

Beispiel 3.2.16. *Der Baum für den Ausdruck*

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

ist:



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

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

Der direkte Oberterm (bzw. Oberbaum) ist dann der Call-by-Name-Redex, falls eine Call-by-Name-Reduktion möglich ist.

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

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

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

$$\begin{array}{ll}\xrightarrow{name,\beta} & \texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})\ \texttt{of}\ \{(\texttt{Cons}\ z\ zs) \to \texttt{False}; \texttt{Nil} \to \texttt{True}\}\\ \xrightarrow{name,case} & \texttt{False}\end{array}$$

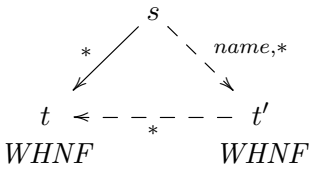
3.2.7. Eigenschaften der Call-by-Name-Reduktion

Für die Call-by-Name-Reduktion gilt:

- Die Call-by-Name-Reduktion ist deterministisch, d.h. für jeden KFPT-Ausdruck s gibt es höchstens einen Ausdruck t mit $s \xrightarrow{name} t$.
- Eine WHNF ist irreduzibel bezüglich der Call-by-Name-Reduktion.

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

Theorem 3.2.17 (Standardisierung)**.** *Sei s ein* KFPT*-Ausdruck. Wenn $s \xrightarrow{*} t$ mit beliebigen (β)- und (case)-Reduktionen (in beliebigem Kontext angewendet), wobei t eine WHNF ist, dann existiert eine WHNF t', so dass $s \xrightarrow{name,*} t'$ und $t' \xrightarrow{*} t$ (wobei das modulo alpha-Gleichheit gemeint ist). D.h. das folgende Diagramm gilt, wobei die gestrichelten Pfeile Existenz-quantifizierte Reduktionen sind:*



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

3.3. Die Kernsprache KFPTS

Als nächste Erweiterung von KFPT betrachten wir die Sprache KFPTS. Das „S“ steht hierbei für Superkombinatoren. Superkombinatoren sind Namen (bzw. Konstanten), die (rekursive) Funktionen bezeichnen. Mit KFPTS beheben wir den zweiten Kritikpunkt aus Bemerkung 3.1.28.

3.3.1. Syntax

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

Definition 3.3.1. *Ausdrücke der Sprache* KFPTS *werden durch die folgende Grammatik gebildet, wobei die Bedingungen an* `case`*-Ausdrücke genau wie in* KFPT *gelten müssen:*

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

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

Zu jedem verwendeten Superkombinator muss es genau eine Superkombinatordefinition geben:

Definition 3.3.2. *Eine* Superkombinatordefinition *ist eine Gleichung der Form:*

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

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

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

Definition 3.3.3. *Ein* KFPTS-Programm *besteht aus:*

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

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

3.3.2. Auswertung von KFPTS-Ausdrücken

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

$$\textbf{RCtxt} ::= [\cdot] \mid (\textbf{RCtxt Expr}) \mid \texttt{case}_{Typ}\ \textbf{RCtxt}\ \texttt{of}\ \mathit{Alts}$$

Die Reduktionsregeln werden erweitert um eine neue Regel.

Definition 3.3.4. *Die Reduktionsregeln (β), (case) und (SK-β) sind in* KFPTS *definiert als:*

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

Die Call-by-Name-Reduktion in KFPTS wendet nun eine der drei Regeln in einem Reduktionskontext an:

Definition 3.3.5. *Wenn $r_1 \to r_2$ mit einer (β)-, (case)- oder (SK-β)-Reduktion, dann ist $R[r_1] \to R[r_2]$ für jeden Reduktionskontext eine (*KFPTS*)-Call-by-Name-Reduktion. Wir notieren dies mit $R[r_1] \xrightarrow{name} R[r_2]$.*

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

Für die dynamischen Typregeln fügen wir hinzu:

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

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

- Der mit $\star$ markierte Unterausdruck ist ein Superkombinator, d.h. der markierte Ausdruck ist von der Form $C[SK^\star]$. Es gibt nun drei Unterfälle:

- $C = C'[[\cdot]\ s_1\ \ldots\ s_n]$ und $\mathrm{ar}(SK) = n$. Dann wende die (SK-β)-Reduktion an: $C'[SK\ s_1\ \ldots\ s_n] \xrightarrow{name,SK-\beta} C'[e[s_1/x_1, \ldots, s_n/x_n]]$ (wenn $SK\ x_1\ \ldots\ x_n = e$ die Definition von SK ist).
- $C = [[\cdot]\ s_1\ \ldots\ s_m]$ und $\mathrm{ar}(SK) > m$. Dann ist der Ausdruck eine WHNF.
- $C = C'[\texttt{case}_T\ [\cdot]\ s_1\ \ldots\ s_m\ \texttt{of}\ Alts]$ und $\mathrm{ar}(SK) > m$. Dann ist der Ausdruck direkt dynamisch ungetypt.

Beispiel 3.3.6. *Die Superkombinatoren map und not seien definiert als:*

$$\begin{aligned} map\ f\ xs &= \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{Cons}\ (f\ y)\ (map\ f\ ys)\} \\ not\ x &= \texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to \texttt{False}; \texttt{False} \to \texttt{True}\} \end{aligned}$$

Die Auswertung des Ausdrucks map not (Cons True (Cons False Nil)) *in* KFPTS-*Call-by-Name-Reihenfolge ist:*

$$\begin{aligned} & map\ not\ (\texttt{Cons True}\ (\texttt{Cons False Nil})) \\ \xrightarrow{name,SK-\beta}\ & \texttt{case}_{\texttt{List}}\ (\texttt{Cons True}\ (\texttt{Cons False Nil}))\ \texttt{of}\ \{ \\ & \quad \texttt{Nil} \to \texttt{Nil}; \\ & \quad (\texttt{Cons}\ y\ ys) \to \texttt{Cons}\ (not\ y)\ (map\ not\ ys)\} \\ \xrightarrow{name,case}\ & \texttt{Cons}\ (not\ \texttt{True})\ (map\ not\ (\texttt{Cons False Nil})) \end{aligned}$$

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

3.4. Erweiterung um `seq`

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

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

Operational kann man dies so darstellen: Die Auswertung eines Ausdrucks (`seq` a b) wertet zunächst a zur WHNF aus, danach erhält man `seq` v b wobei v eine WHNF ist. Nun darf man (`seq` v b) $\to b$ reduzieren. Ein analoger Operator ist der Operator `strict` (in Haskell infix geschrieben als `!$`). Angewendet auf eine Funktion, macht `strict` die Funktion strikt in ihrem ersten Argument[10], d.h. `strict` erzwingt die Auswertung des Arguments bevor die Definitions-

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

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

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

einsetzung bzw. β-Reduktion durchgeführt werden kann. `seq` und `strict` sind äquivalent, da jeder der beiden Operatoren mit dem jeweilig anderen kodiert werden kann[11]:

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

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

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

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

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

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

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

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

$$\texttt{seq}\ v\ t \to t, \text{ wenn } v \text{ eine Abstraktion oder eine Konstruktoranwendung ist}$$

3.5. KFPTSP: Polymorphe Typen

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

[11] In Haskell-Notation:

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

Definition 3.5.1. *Die Syntax von polymorphen Typen kann durch die folgende Grammatik beschrieben werden:*

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

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

Einige Beispiel-Ausdrücke mit Typen sind:

$$\begin{array}{lcl} \texttt{True} & :: & \texttt{Bool} \\ \texttt{False} & :: & \texttt{Bool} \\ \texttt{not} & :: & \texttt{Bool} \to \texttt{Bool} \\ \texttt{map} & :: & (a \to b) \to [a] \to [b] \\ (\lambda x.x) & :: & (a \to a) \end{array}$$

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

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

d.h. um die Konsequenz herleiten zu dürfen, muss die Voraussetzung erfüllt sein. Die vereinfachten Regeln sind:

- Für die Anwendung:
$$\frac{s :: T_1 \to T_2, t :: T_1}{(s\ t) :: T_2}$$
- Instanziierung
$$\frac{s :: T}{s :: T'} \quad \text{wenn } T' = \sigma(T), \text{ wobei } \sigma \text{ eine Typsubstitution ist, die Typen für Typvariablen ersetzt.}$$
- Für `case`-Ausdrücke:
$$\frac{s :: T_1, \quad \forall i : Pat_i :: T_1, \quad \forall i : t_i :: T_2}{(\texttt{case}_T\ s\ \texttt{of}\ \{Pat_1 \to t_1; \ldots; Pat_n \to t_n\}) :: T_2}$$

Beispiel 3.5.2. *Die booleschen Operatoren* ***and*** *und* ***or*** *kann man in* KFPTS *als Abstraktionen definieren:*

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

Den Ausdruck ***and*** `True False` *kann man nun typisieren durch zweimaliges Anwenden der Anwendungsregel:*

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

Beispiel 3.5.3. *Der Ausdruck*

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

lässt sich mit obigen Regeln folgendermaßen typisieren:

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

Beispiel 3.5.4. *Der Operator* `seq` *hat den Typ* $a \to b \to b$*, da er sein erstes Argument im Ergebnis ignoriert.*

Beispiel 3.5.5. *Hat man die Typen von* ***`map`*** *und* ***`not`*** *bereits gegeben, so kann man den Typ der Anwendung (****`map not`****) mit obigen Regeln berechnen als:*

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

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

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

3.6. KFPTSP+seq als Kernsprache für Haskell

Wir argumentieren informell, warum KFPTSP+seq als Kernsprache (des funktionalen Anteils) von Haskell angesehen werden kann, da andere Haskell-Konstrukte in dies Sprache „verlustfrei“ übersetzt werden können. Umgekehrt bedeutet dies, wir können die Haskell-Konstrukte benutzen, da wir durch die Übersetzung in KFPTSP+seq deren Semantik kennen.

3.6.1. `let`-Ausdrücke und `where`-Klauseln

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

3.6.2. Darstellung von Zahlen in KFPTSP+seq

Zahlen mit endlichem Wertebereich (z.B. `Int`) können direkt in KFPTSP+seq dargestellt werden, indem ein entsprechender Typ mit (0-stelligen) Konstruktoren hinzugefügt wird. Zahlen unbeschränkter Länge können auf diese Weise nicht dargestellt werden, da wir stets angenommen haben, dass die Menge der Konstruktoren zu einem Typ endlich ist.

Natürliche Zahlen unbeschränkter Länge können z.B. durch Listen von Bits dargestellt werden. Ein etwas anderer Ansatz ist die sogenannte *Peano*-Kodierung[12]:

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

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

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

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

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

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

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

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

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

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

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

Die Multiplikation kann analog definiert werden durch

[12] nach dem italienischen Mathematiker Giuseppe Peano benannt

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

Vergleichsoperationen können auf Peano-Zahlen wie folgt implementiert werden:

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

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

3.6.3. Datentypen

Datentypen können Typen in KFPTSP+seq dargestellt werden (mit den entsprechenden Stelligkeiten der Konstruktoren). Auch wenn KFPTSP+seq keine Record-Syntax unterstützt, so ist jedoch einsichtig, dass diese nur syntaktischer Zucker ist (die Zugriffsfunktionen müssen automatisch erzeugt werden), und daher eine Übersetzung in KFPTSP+seq unproblematisch.

3.6.4. if-then-else-Ausdrücke

Diese können in `case`-Ausdrücke übersetzt werden: Aus `if` e_1 `then` e_2 `else` e_3 wird $\texttt{case}_{Bool}\ e_1\ \texttt{of}\ \{\texttt{True} \to e_2; \texttt{False} \to e_3\}$.

3.6.5. `case`-Ausdrücke

Die `case`-Ausdrücke in Haskell bieten durch ihre verschachtelten Pattern und mögliche default-Alternativen mehr Möglichkeiten als die `case`-Ausdrücke von KFPTSP+seq. Zudem müssen in Haskell nicht alle möglichen Pattern abgedeckt werden. Trotzdem ist eine Übersetzung möglich:

- Verschachtelte Pattern werden in verschachtelte `case`-Ausdrücke mit einfachen Pattern übersetzt.
- Fehlende Pattern werden hinzugefügt, wobei die entsprechenden `case`-Alternativen auf einen nichttermineren Ausdruck $\bot$ abgebildet werden.
- Default-Alternativen werden durch alle nicht abgedeckten Pattern ersetzt.

Betrachte

```
case ys of
  (x1:(x2:(x3:(x4:xs)))) -> Just x4
  _ ->  Nothing
```

Für die Übersetzung in KFPTSP+seq entseht der folgende Ausdruck (hier in Haskell-Syntax, die Darstellung in KFPTSP+seq ist analog):

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

3.6.6. Funktionsdefinitionen

Definition mit Pattern können in Superkombinatordefinitionen und `case`-Ausdrücke übersetzt werden. Etwaige Guards und Pattern Guards müssen entsprechend des Datenflusses in if-then-else-Ausdrücke (und dann wiederum in `case`-Ausdrücke) übersetzt werden.
Guards lassen sich in KFPTSP+seq übersetzen, indem man verschachtelte `if-then-else`-Ausdrücke benutzt (und diese durch $\texttt{case}_{\texttt{Bool}}$-Ausdrücke darstellt). Etwa in der Form

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

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

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

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

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

Als weiteres Beispiel betrachte

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

Die passende Übersetzung (noch in Haskell-Syntax) ist:

```
myFun fruct p personendb = case frucht of
                             Banane l h kruemmung -> case lookup p personendb of
                                                       Just x -> Right x
                                                       Nothing -> Left "Fehler"
                             Apfel x1 -> Left "Fehler"
                             Birne x2 x3 -> Left "Fehler"
```

3.7. Übersicht über die Kernsprachen

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

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

3.8. Lazy Evaluation in Haskell

Wir führen keine Call-by-Need-Variante der Kernsprachen ein, da diese eher kompliziert sind (z.B. kann man diese mathematisch korrekt mit einer Kernsprache erweitert um rekursive let-Ausdrücke erfassen, siehe z.B. (Ariola & Felleisen, 1997; Schmidt-Schauß et al., 2008)). Stattdessen beschreiben wir aus eher praktischer Sicht, wie die effiziente Implementierung im Haskell-Compiler funktioniert und wie die Auswertung im GHCi beobachtet werden kann.

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.

Betrachte z.B. den Ausdruck `square (10+5)` wobei

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

Die Call-By-Name-Auswertung ist

$$\texttt{square (10+5)} \xrightarrow{name} \texttt{(10+5)*(10+5)} \xrightarrow{name} \texttt{15*(10+5)} \xrightarrow{name} \texttt{15*15} \xrightarrow{name} \texttt{225}.$$

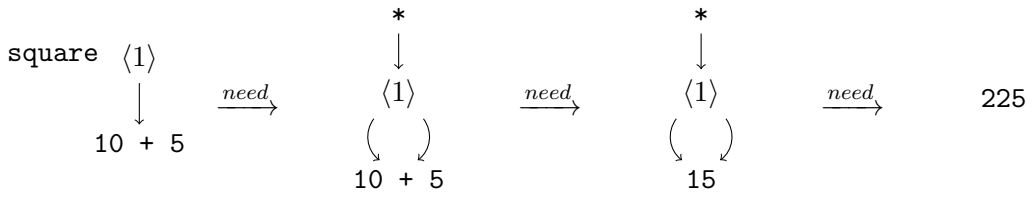
Bei Auswertung in Call-by-Value-Reihenfolge wird das Argument `10+5` vor der Definitionseinsetzung ausgewertet:

$$\texttt{square (10+5)} \xrightarrow{value} \texttt{square 15} \xrightarrow{value} \texttt{15*15} \xrightarrow{value} 225.$$

Die Call-by-Need-Auswertung sieht den Unterausdruck `10+5` als Thunk, und daher wird `10+5` nur einmal ausgewertet. Wir deuten das an, indem wir den Verweis als Speicheraddresse in der Form $\langle 1 \rangle$ darstellen und die Abbildung von Speicheraddresse auf Ausdruck nebenbei notieren. Beachte, die Unterausdrücke, die Werte sind (wie `10` und `5`) stellen wir ohne Verweis darauf dar:

$$(\texttt{square}\ \langle 1 \rangle, \{\langle 1 \rangle \mapsto \texttt{10+5}\}) \xrightarrow{need} (\langle 1 \rangle * \langle 1 \rangle, \{\langle 1 \rangle \mapsto \texttt{10+5}\}) \xrightarrow{need} (15 * 15, \{\langle 1 \rangle \mapsto \texttt{15}\}) \xrightarrow{need} (225, \{\langle 1 \rangle \mapsto \texttt{15}\}).$$

Eine alternative Darstellung ist die Verwendung von Graphen:



Wir betrachten als Beispiel ein Programm mit (eigentlich verbotenen, puren) Seiteneffekten, um uns die Unterschiede von Call-by-name, Call-by-value und Call-by-Need-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 Call-by-Value-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 Call-by-Name-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 Call-by-Need-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.

3.8.1. Potentiell unendliche Datenstrukturen

Lazy Evaluation ermöglicht „unendliche“ 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)

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

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:

```
nums = iterate (1+) 0

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

3.8.2. 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. Wenn wir verstanden haben, wie die Typklassen aufgelöst werden (siehe Kapitel 4), dann sehen wir, dass `z` eigentlich 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])
```

Lazy Evaluation 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
```

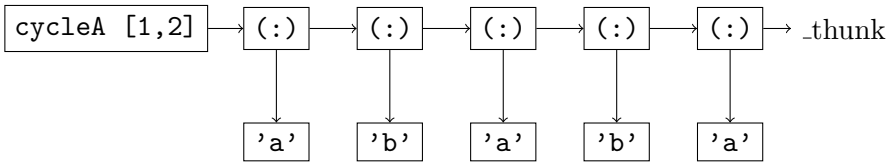
Wir müssen uns nur um die Daten kümmern, also *wie* wir die Primzahlen berechnen. Die Kontrolle über die Anzahl der benötigten Primzahlen erfolgt später. Zu Effizienz-Betrachtungen siehe http://www.haskell.org/haskellwiki/Prime_numbers.

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

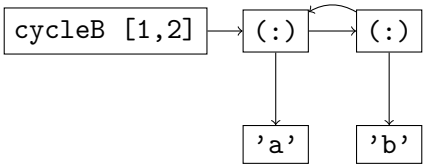
```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys

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
```

3.8.3. 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
```

3.8.4. 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 Call-by-Need-Auswertung in Haskell zusammen: Alle effizienten Implementierungen von Haskell verwenden die Call-by-Need-Auswertung, deren wesentliche Idee ist, Unterausdrücke nur höchstens einmal auszuwerten. Will man die Semantik der Haskell-Programme betrachten, ohne auf die Effizienz zu achten, kann man auch die Call-by-Name-Semantik (wie wir sie z.B. für die Kernsprache KFPTSP+seq gesehen haben) verwenden (da Call-by-Need und Call-by-Name-Auswertung *semantisch* äquivalent sind).

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

3.9. Quellennachweis und weiterführende Literatur

Zum Lambda-Kalkül gibt es viele Quellen. Ein nicht ganz leicht verständliches Standardwerk ist (Barendregt, 1984). Ein gute Einführung ist in (Hankin, 2004) zu finden. Call-by-need Lambda-Kalküle mit `let` sind in (Ariola et al., 1995; Ariola & Felleisen, 1997) und (Maraist et al., 1998) beschrieben. Nichtdeterministische Erweiterungen sind in (Kutzner & Schmidt-Schauß, 1998; Kutzner, 2000; Mann, 2005b; Mann, 2005a) zu finden. Die vorgestellte call-by-need Reduktion orientiert sich dabei an (Mann, 2005a). Call-by-need Kalküle mit rekursivem `let` wurden u.a. in (Schmidt-Schauß et al., 2008; Schmidt-Schauß et al., 2010) behandelt. Der Begriff der kontextuellen Äquivalenz geht auf (Morris, 1968) zurück und wurde im Rahmen des Lambda-Kalküls insbesondere von (Plotkin, 1975) untersucht. Die „KFP“-Kernsprachen wurden im Wesentlichen aus (Schmidt-Schauß, 2009) entnommen. Abschnitt 3.8 stammt im Wesentlichen aus Lehrmaterial von Steffen Jost (Jost, 2019).

4

Haskells Typklassensystem

4.1. 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].

[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`

`==` 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

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

gebildet. Zur Abbildung der Typklassenbedingungen könnte man erweiterte Typen $\mathbf{T_e}$ wie folgt definieren:

$$\mathbf{T_e} ::= \mathbf{T} \mid \mathbf{Kon}\ \texttt{=>}\ \mathbf{T}$$

wobei **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:

```
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
```

4.2. 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.

4.3. 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 $a_1, \ldots, a_N$ 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 4.3.1. *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 4.3.2. *Definiere eine* Eq*-Instanz für* BinBaumMitKM*, wobei Bäume nur dann gleich sind, wenn ihre Knoten- und Kantenmarkierungen identisch sind.*

4.4. 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>
```

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:

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

```
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 4.4.1. *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].

4.5. 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:

[3] Mit dem Schlüsselwort `default` kann man das Defaulting-Verhalten auf Modulebene anpassen, siehe (Peyton Jones, 2003, Abschnitt 4.3.4)

```
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]
```

4.6. 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

$$\kappa := * \mid \kappa \to \kappa$$

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
```

4.7. 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.

4.7.1. 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 4.7.1. *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.*

4.8. Auswahl weiterer vordefinierter Typklassen

Abbildung 4.1 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 4.2 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.

[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.

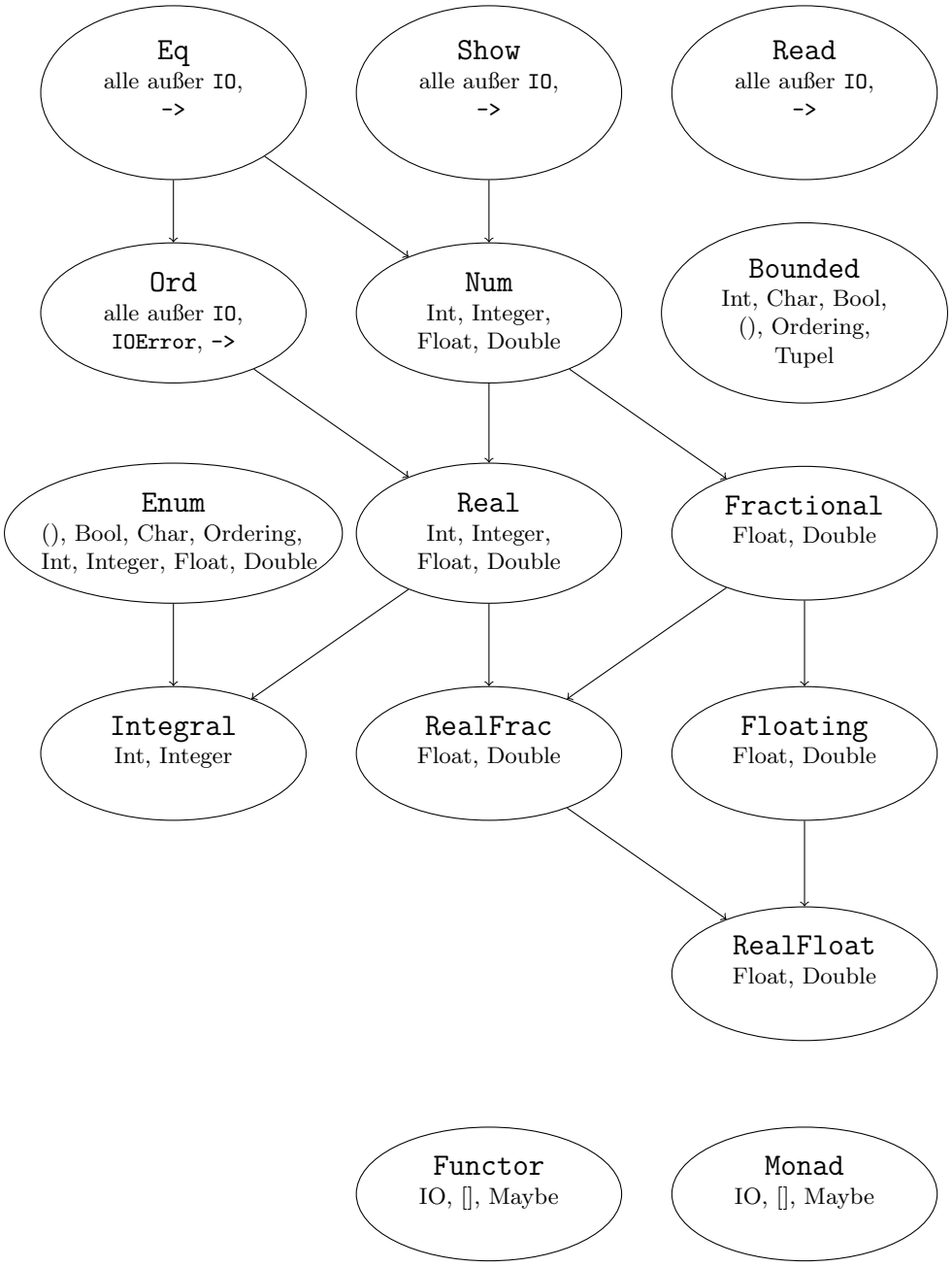


Abbildung 4.1.: Typklassenhierarchie der vordefinierten Haskell-Typklassen und Instanzen der vordefinierten Typen entsprechend des Haskell Language Report 2010

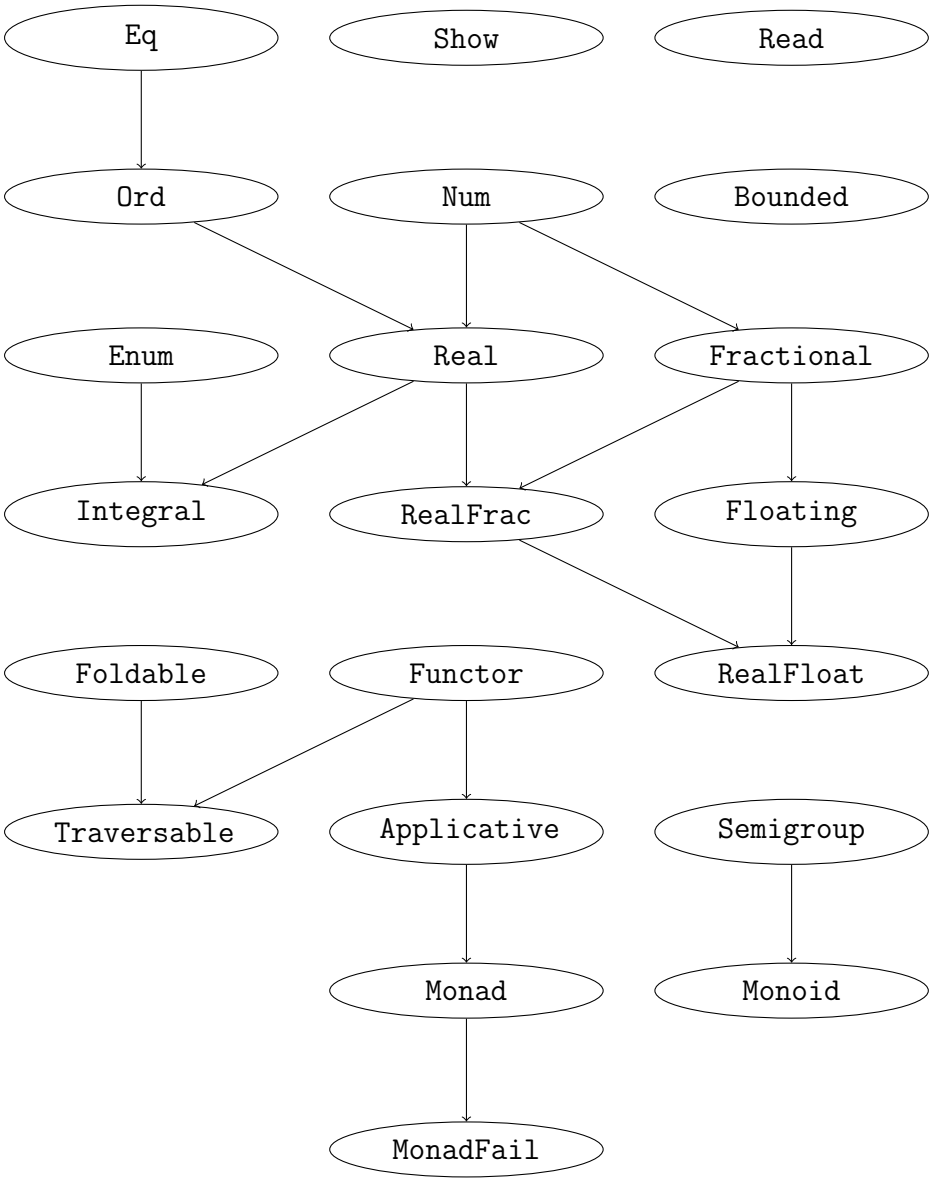


Abbildung 4.2.: Typklassenhierarchie der vordefinierten Haskell-Typklassen entsprechend der aktuellen GHC-Prelude (Version: base-4.13.0.0)

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:

```
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
```

4.8.1. 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

$$\underbrace{a \texttt{ <> } a \texttt{ <> } \ldots \texttt{ <> } a}_{n \text{ viele } as}.$$

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`).

4.8.2. 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.

4.9. 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 ==
overloadedeq :: EqDict a -> a -> a -> Bool
overloadedeq dict a b = eqEq dict a b

-- Ersatz f"ur /=
overloadedneq :: EqDict a -> a -> a -> Bool
overloadedneq dict a b = eqNeq dict a b
```

Beachte, wie sich der Typ verändert hat: Aus der Klassenbeschränkung `Eq a` in `(==) :: Eq a => a -> a -> Bool` wurde ein zusätzlicher Parameter: Das Dictionary für `Eq`.

Jetzt kann man überladene Funktionen entsprechend anpassen und überall, wo `(==)` (mit allgemeinem Typ) stand, einen zusätzlichen Dictionary-Parameter einfügen und `overloadedeq` verwenden. Z.B. wird die Funktion `elem` wie folgt modifiziert: Aus

```
elem :: (Eq a) => a -> [a] -> Bool
elem e []    = False
elem e (x:xs)
  | e == x   = True
  | otherwise = elem e xs
```

wird

```
elemEq :: EqDict a -> a -> [a] -> Bool
elemEq dict e []    = False
elemEq dict e (x:xs)
  | (eqEq dict) e  x   = True
  | otherwise          = elemEq dict e xs
```

Es gibt jedoch stellen im Programm, 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 }
```

4.10. 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

4.11. Quellennachweise und weitere Literatur

Haskells Typklassensystem wurde eingeführt in (Wadler & Blott, 1989). Die Standardklassen sind im Haskell Report dokumentiert. In (Wadler & Blott, 1989) ist auch die Auflösung der Überladung zu finden. Tiefergehend formalisiert wurden Haskells Typklassen in (Hall et al., 1996). Konstruktorklassen wurden in (Jones, 1995) vorgeschlagen.

5

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.

5.1. 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 #-}
```

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:

[1] Applikative Funktoren wurden erst spät identifiziert, siehe insbesondere (McBride & Paterson, 2008).

```
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.

5.1.1. 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 5.1.1. *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 (Snoyman, 2017) 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 5.1.2. *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`.

5.1.2. Listen als Instanz der Klasse Applicative

5.1.2.1. 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.

5.1.2.2. Zip-Listen Instanz

In der vorherigen Instanz wurden für zwei Listen $\texttt{fs} = [f_1, \ldots, f_n]$ und $\texttt{xs} = [x_1, \ldots, x_m]$ 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);

[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`.

```
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
```

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)]
```

5.1.3. 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)
```

5.1.4. 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`

5.1.5. 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)
```

5.2. 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 ???
```

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*.

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

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 5.1.2 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
```

5.2.1. 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.

5.2.2. 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]:

```
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:

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

```
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 Applica-

tive 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

```
((,) <$> 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.

[5] mehr Details zur Übersetzung und zum Hintergrund sind in (Marlow et al., 2016) zu finden

5.2.3. 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 $x \notin FV(\texttt{m2}, \texttt{m3})$

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.

5.2.4. 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)
```

5.2.5. 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
```

5.3. 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:

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

```
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)
  --  (>>=) :: 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 hinzu
```

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.

5.4. 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
  ```

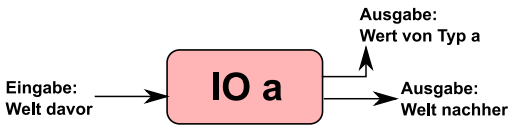
  Da die Auswertung von `length` die Listenelemente gar nicht anfasst, würde obiger Aufruf, keine `getZahl`-Aufrufe ausführen.
- In reinen funktionalen Programmiersprachen wird oft auf die Festlegung einer genauen Auswertungsreihenfolge verzichtet, um Optimierungen und auch Parallelisierung von Programmen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



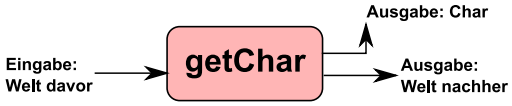
Aus Sicht von Haskell sind Objekte vom Typ `IO a` bereits *Werte*, d.h. sie können nicht weiter ausgewertet werden. Dies passt dazu, dass auch andere Funktionen Werte in Haskell sind. Allerdings im Gegensatz zu „normalen“ Funktionen kann Haskell kein Argument vom Typ „Welt“ bereit stellen. (Die Funktion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

5.4.1. 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
```

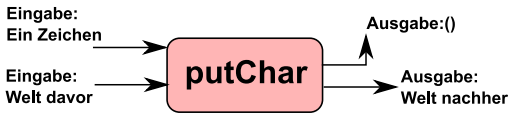
In der Welt-Sichtweise ist `getChar` eine Funktion, die eine Welt erhält und als Ergebnis eine veränderte Welt sowieso ein Zeichen liefert. Man kann dies durch folgendes Bild illustrieren:



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.

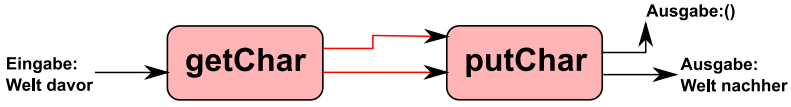
```
putChar :: Char -> IO ()
```

Auch `putChar` lässt sich mit einem Bild illustrieren:



5.4.2. 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:

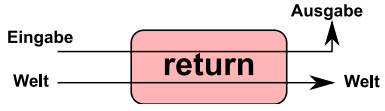
```
echoDup :: IO ()
echoDup = getChar >>= \x -> (putChar x >> putChar x)
```

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)
```

5.4.3. 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.

5.4.4. 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.

5.4.5. 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 `Handle`s 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
```

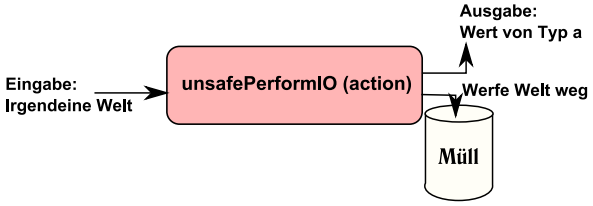
nicht die Aktion `aktion` durchgeführt (berechnet), sondern direkt mit den weiteren Aktionen weitergemacht. Die Aktion `aktion` wird erst dann ausgeführt, wenn der Wert der Variablen `ergebnis` benötigt wird.

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.

5.4.6. 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).

5.5. 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

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

```
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
```

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

5.6. 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.

5.6.1. 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.

5.6.2. 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.

5.6.3. Beispiel: Werte mit Wahrscheinlichkeiten

Wir betrachten ein Beispiel aus (Lipovaca, 2011). 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.

5.7. Quellennachweis

Die Darstellung der Monadischen Programmierung stammt im Wesentlichen aus (Jost, 2019) und (Lipovaca, 2011). Das Taschenrechnerbeispiel ist schon älter (siehe z.B. (Schmidt-Schauß, 2009)) wurde aber an die aktuellen Bibliotheken angepasst. Die Darstellung von Haskells monadischen IO richtet sich im Wesentlichen nach (Peyton Jones, 2001). Als weiterführende Literatur sind die Originalarbeiten von Wadler zur Verwendung von Monaden in Haskell (z.B. (Wadler, 1992; Wadler, 1995)) und die theoretische Fundierung von Moggi (Moggi, 1991) sehr empfehlenswert.

6 Typisierung

In diesem Kapitel stellen wir zwei Verfahren zur polymorphen Typisierung von Haskell bzw. KFPTS+seq Programmen vor. Wir beschränken uns dabei auf die Typisierung ohne Typklassen. Zunächst motivieren wir, warum Typsysteme sinnvoll sind und warum der Begriff „dynamisch ungetypt“ nicht ausreicht. Anschließend führen wir schrittweise Grundbegriffe und notwendige Methoden zur Typisierung ein (u.a. die Unifikation von Typen). Nachdem wir die Typisierung von KFPTS+seq-*Ausdrücken* eingeführt haben, stellen wir zwei Verfahren zur Typisierung von (rekursiven) Superkombinatoren vor: Das iterative Verfahren berechnet allgemeinste polymorphe Typen, aber die zugrundeliegende Aufgabe ist nicht entscheidbar. Das Milner-Verfahren wird in Haskell verwendet, berechnet eingeschränktere Typen als das iterative Verfahren, ist aber terminierend, und somit ein Entscheidungsverfahren.

6.1. Motivation

KFPTS+seq-Programme sind nicht typisiert, daher können bei der Auswertung dynamische Typfehler zur Laufzeit auftreten. Solche dynamischen Typfehler sind Programmierfehler, d.h. i.A. rechnet der Programmierer nicht damit, dass sein Programm einen Typfehler hat. Ein starkes statisches Typsystem unterstützt daher den Programmierer, Fehler im Programm möglichst früh (also vor der Laufzeit) zu erkennen. Typen können jedoch auch den Charakter einer Dokumentation übernehmen: Oft kann man am Typ einer Funktion schon ausmachen, welche Funktionalität durch die entsprechende Funktion implementiert wird und wie die Funktion verwendet werden kann. Daher ergeben sich die folgenden Grundanforderungen für die Typisierung von Programmiersprachen:

- Die Typisierung sollte zur Compilezeit entschieden werden.
- Korrekt getypte Programme erzeugen keine Typfehler zur Laufzeit.

Dies sind Mindestanforderungen an ein starkes statisches Typsystem. Gleichzeitig gibt es jedoch weitere wünschenswerte Eigenschaften

- Das Typsystem sollte beim Programmieren möglichst wenig einschränken, d.h. sinnvolle Programme sollten möglichst nicht nur aufgrund des Typsystems als falsche (ungetypte) Programme zurück gewiesen werden.
- Idealerweise muss der Programmierer die Typen nicht (oder nur selten) selbst vorgeben. Selbst Typen anzugeben ist oft mühsam. In vielen Fällen kann dies jedoch auch sinnvoll sein. Trotzdem ist es wünschenswert, wenn der Compiler (der Typchecker) die Typen des Programms selbst berechnen kann (diesen Vorgang nennt man *Typinferenz*). Auch hier erwartet man, dass der Compiler möglichst die allgemeinsten Typen berechnet.

Es gibt einige Typsysteme, die diese beiden wünschenswerten Eigenschaften nicht erfüllen, z.B. gibt es den *einfach getypten* (simply-typed) Lambda-Kalkül: Für ihn ist der Typcheck zur Compilezeit entscheidbar, aber die Menge der getypten Ausdrücke ist nicht mehr Turing-mächtig,

da alle korrekt getypten Programme terminieren (das kann man beweisen!). Der einfach getypte Lambda-Kalkül schränkt daher den Programmierer sehr stark ein. In Haskell-Implementierungen gibt es Erweiterungen des Typsystems (nicht im Haskell-Standard), die vollständige Typinferenz nicht mehr zulassen. Der Programmierer muss dann Typen vorgeben. Es gibt auch Erweiterungen bzw. Typsysteme, die nicht entscheidbar sind, dann kann es passieren, dass der Typcheck (also der Compiler) nicht terminiert.

Ein erster Ansatz für ein allgemeines Typsystem für KFPTSP+seq ist das folgende:

> Ein KFPTSP+seq-Programm ist korrekt getypt, wenn es keine dynamischen Typfehler zur Laufzeit erzeugt.

Dies schränkt die Menge der korrekt getypten Programme, auf solche Programme ein, die nicht dynamisch ungetypt sind.

Leider ist dieser Begriff eher ungeeignet, da die Frage, ob ein beliebiges KFPTS-Programm dynamisch ungetypt ist, unentscheidbar ist. Wir skizzieren den Beweis:

Sei `tmEncode` eine KFPTS+seq-Funktion, die sich wie eine universelle Turingmaschine verhält, d.h. sie erhält eine Turingmaschinenbeschreibung und eine Eingabe für die Turingmaschine und simuliert anschließend die Turingmaschine. Sie liefert `True`, falls die Turingmaschine auf dieser Eingabe anhält und akzeptiert. Wir nehmen an, dass `tmEncode` angewendet auf passende Argumente niemals dynamisch ungetypt ist[1].

Für eine Turingmaschinenbeschreibung b und eine Eingabe e für diese Turingmaschine sei der Ausdruck s definiert durch:

$$
\begin{aligned}
s := \quad & \texttt{if tmEncode } b\ e \\
& \quad \texttt{then case}_{\text{Bool}}\ \texttt{Nil of } \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\} \\
& \quad \texttt{else case}_{\text{Bool}}\ \texttt{Nil of } \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\}
\end{aligned}
$$

Da (`tmEncode` b e) nicht dynamisch ungetypt ist, gilt: s ist genau dann dynamisch ungetypt, wenn die Auswertung von (`tmEncode` b e) mit `True` endet. Umgekehrt bedeutet dies: Wenn wir entscheiden könnten, ob s dynamisch ungetypt ist, dann können wir auch das Halteproblem für Turingmaschinen entscheiden, was bekanntlich nicht möglich ist. Daher folgt:

Satz 6.1.1. *Die dynamische Typisierung von* KFPTS+seq*-Programmen ist unentscheidbar.*

Wir werden daher Typsysteme betrachten, die die Programmierung weiter einschränken, d.h. für die beiden von uns im Folgenden betrachteten Typsysteme gilt:

- Ein korrekt getypter Ausdruck ist nicht dynamisch ungetypt (sonst wäre das Typsystem nicht viel wert)
- Es gibt Ausdrücke, die nicht dynamisch ungetypt sind, aber trotzdem nicht korrekt getypt sind.

[1] Diese Annahme scheint zunächst etwas unrealistisch, da wir nachweisen möchten, dass diese Frage unentscheidbar ist. Aber: Wir zeigen, dass die Frage, ob ein beliebiger KFPTS+seq-Ausdruck dynamisch ungetypt ist, unentscheidbar ist. `tmEncode` ist glücklicherweise Milner-Typisierbar (das folgt, da `tmEncode` in Haskell typisierbar ist), und für Milner-getypte KFPTS-Programme gilt: Solche Programme sind niemals dynamisch ungetypt (was wir in einem späteren Abschnitt auch zeigen werden). Im Anhang ist zur Vollständigkeit eine Haskell-Implementierung für `tmEncode` angegeben.

6.2. Typen: Sprechweisen, Notationen und Unifikation

In diesem Abschnitt führen wir einige Sprechweisen und Notationen für Typen ein. Wir wiederholen zunächst die Syntax von polymorphen Typen (ohne Typklassenconstraints):

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

wobei TV für eine Typvariable steht und TC ein Typkonstruktor ist[2].

Zur Erinnerung: Ein *Basistyp* ist ein Typ der Form TC, wobei TC ein nullstelliger Typkonstruktor ist. Ein *Grundtyp* (oder alternativ *monomorpher Typ*) ist ein Typ, der keine Typvariablen enthält.

Beispiel 6.2.1. *Die Typen* `Int`, `Bool` *und* `Char` *sind Basistypen. Die Typen* `[Int]` *und* `Char -> Int` *sind keine Basistypen aber Grundtypen. Die Typen* `[a]` *und* `a -> a` *sind weder Basistypen noch Grundtypen.*

Wir verwenden als neue Notation auch *all-quantifizierte Typen*. Sei τ ein polymorpher Typ mit Vorkommen der Typvariablen $\alpha_1, \ldots, \alpha_n$, dann ist $\forall\alpha_1, \ldots, \alpha_n.\tau$ der *all-quantifizierte Typ* für τ. Man kann polymorphe Typen als allquantifiziert ansehen, da die Typvariablen für beliebige Typen stehen können. Wir benutzen die Quantor-Syntax, um (im Typisierungsverfahren) zwischen Typen zu unterscheiden die „kopiert“ (also umbenannt) werden dürfen und Typen für die wir im Typisierungsverfahren keine Umbenennung erlauben. Die Reihenfolge der allquantifizierten Typvariablen am Quantor spielt keine Rolle. Deshalb verwenden wir auch die Notation $\forall\mathcal{X}.\tau$, wobei $\mathcal{X}$ eine Menge von Typvariablen ist.

Definition 6.2.2. *Eine Typsubstitution ist eine Abbildung* $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ *einer endlichen Menge von Typvariablen auf Typen. Sei* σ *eine Typsubstitution. Die homomorphe Erweiterung* σ_E *von* σ *ist die Erweiterung der Typsubstitution als Abbildung von Typen auf Typen, die definiert ist durch:*

$$\begin{aligned}
\sigma_E(TV) &:= \sigma(TV),\ \textit{falls}\ \sigma\ \textit{die Variable}\ TV\ \textit{abbildet}\\
\sigma_E(TV) &:= TV,\ \textit{falls}\ \sigma\ \textit{die Variable}\ TV\ \textit{nicht abbildet}\\
\sigma_E(TC\ T_1\ \ldots\ T_n) &:= TC\ \sigma_E(T_1)\ \ldots\ \sigma_E(T_n)\\
\sigma_E(T_1 \to T_2) &:= \sigma_E(T_1) \to \sigma_E(T_2)
\end{aligned}$$

Wir unterscheiden im Folgenden nicht zwischen σ *und der Erweiterung* σ_E.

Eine Typsubstitution ist eine Grundsubstitution für einen Typ τ *genau dann, wenn* σ *auf Grundtypen abbildet und alle Typvariablen, die in* τ *vorkommen, durch* σ *auf Grundtypen abgebildet werden.*

Mit Grundsubstitutionen kann man eine Semantik für polymorphe Typen angeben, die auf monomorphe Typen zurückführt. Sei τ ein polymorpher Typ, dann ist die Grundtypensemantik $\text{sem}(\tau)$ von τ die Menge aller möglichen Grundtypen, die durch Substitution, angewendet auf τ, erzeugt werden können, d.h.

$$\texttt{sem}(\tau) := \{\sigma(\tau) \mid \sigma \text{ ist Grundsubstitution für } \tau\}$$

[2] Es sei nochmals daran erinnert, dass der Listentyp in Haskell als `[a]` dargestellt wird, der Typkonstruktor ist hierbei `[]`. Ebenso wird für Tupel-Typen eine spezielle Syntax verwendet: `(,...,)` ist der Typkonstruktor, z.B. ist `(a,b)` der Typ für Paare. Wir verwenden diese Syntax auch im Folgenden

Das entspricht gerade der Vorstellung von *schematischen* Typen: Ein polymorpher Typ beschreibt das Schema einer Menge von Grundtypen.

Wir haben bereits in Kapitel 3 (Abschnitt 3.5) einfache Typregeln eingeführt. Betrachten wir erneut die Typregel für die Anwendung

$$\frac{s :: T_1 \to T_2, t :: T_1}{(s\ t) :: T_2}$$

Diese Regel verlangt dass der Argumenttyp von s schon passend zum Typ von t ist. Betrachtet man z.B. die Anwendung (`map not`) unter der Annahme, dass wir die (allgemeinsten) Haskell-Typen von `map` und `not` schon kennen:

```
map :: (a -> b) -> [a] -> [b]
not :: Bool -> Bool
```

Damit die Anwendungsregel verwendbar ist, müssen wir vorher den Typ von `map` instanziieren (d.h. die passende Substitution ist $\sigma = \{\texttt{a} \mapsto \texttt{Bool}, \texttt{b} \mapsto \texttt{Bool}\}$). Hier haben wir die passende Substitution durch Hinschauen geraten. Ein allgemeines (und automatisierbares) Verfahren zum „Gleichmachen“ von Typen (und i.A. auch anderen Termen) ist die *Unifikation*. Diese werden wir nun formal definieren:

Definition 6.2.3. *Ein* Unifikationsproblem *auf Typen ist gegeben durch eine Menge E von Gleichungen der Form $\tau_1 \doteq \tau_2$, wobei τ_1 und τ_2 polymorphe Typen sind. Eine Lösung eines Unifikationsproblem auf Typen ist eine Substitution σ (bezeichnet als* Unifikator*), so dass $\sigma(\tau_1) = \sigma(\tau_2)$ für alle Gleichungen $\tau_1 \doteq \tau_2$ des Problems. Eine* allgemeinste Lösung *(allgemeinster Unifikator, mgu = most general unifier) von E ist ein Unifikator σ, so dass gilt: Für jeden anderen Unifikator ρ gibt es eine Substitution γ so dass $\rho(x) = \gamma \circ \sigma(x)$ für all $x \in FV(E)$.*

Man kann zeigen, dass allgemeinste Unifikatoren eindeutig bis auf Umbenennung von Variablen sind.

Beispiel 6.2.4. *Für das Unifikationsproblem $\{\alpha \doteq \beta\}$ sind $\sigma = \{\alpha \mapsto \beta\}$ und $\sigma' = \{\beta \mapsto \alpha\}$ allgemeinste Unifikatoren. Die Substitution $\sigma'' = \{\alpha \mapsto \texttt{Bool}, \beta \mapsto \texttt{Bool}\}$ ist ein Unifikator aber kein allgemeinster.*

Der folgende Unifikationsalgorithmus berechnet einen allgemeinsten Unifikator, falls dieser existiert. Die Datenstruktur ist eine Multimenge von zu lösenden Gleichungen. (Multimengen sind analog zu Mengen, aber mehrfaches Vorkommen von Elementen ist erlaubt.) Wir verwenden dabei E für solche (Multi-)Mengen und $E \cup E'$ stellt die disjunkte Vereinigung zweier Multi-Mengen dar. Die Notation $E[\tau/\alpha]$ bedeutet, dass in allen Gleichungen in E die Typvariable α durch τ ersetzt wird (d.h. auf alle linken und rechten Seiten der Gleichungen wird die Substitution $\{\alpha \mapsto \tau\}$ angewendet). Der Algorithmus startet mit dem zum Unifikationsproblem passenden Gleichungssystem und wendet anschließend die folgenden Schlussregeln (nichtdeterministisch) solange an, bis

- keine Regel mehr anwendbar ist,
- oder Fail auftritt.

$$\text{(Fail1)}\ \frac{E \cup \{(TC_1\ \tau_1\ \ldots\ \tau_n) \doteq (TC_2\ \tau'_1\ \ldots\ \tau'_m)\}}{\text{Fail}}\ \text{wenn } TC_1 \neq TC_2$$

$$\text{(Fail2)}\ \frac{E \cup \{(TC_1\ \tau_1\ \ldots\ \tau_n) \doteq (\tau'_1 \to \tau'_2)\}}{\text{Fail}} \qquad \text{(Fail3)}\ \frac{E \cup \{(\tau'_1 \to \tau'_2) \doteq (TC_1\ \tau_1\ \ldots\ \tau_n)\}}{\text{Fail}}$$

$$\text{(Decompose1)}\ \frac{E \cup \{TC\ \tau_1\ \ldots\ \tau_n \doteq TC\ \tau'_1\ \ldots\ \tau'_n\}}{E \cup \{\tau_1 \doteq \tau'_1, \ldots, \tau_n \doteq \tau'_n\}} \qquad \text{(Decompose2)}\ \frac{E \cup \{\tau_1 \to \tau_2 \doteq \tau'_1 \to \tau'_2\}}{E \cup \{\tau_1 \doteq \tau'_1, \tau_2 \doteq \tau'_2\}}$$

$$\text{(Orient)}\ \frac{E \cup \{\tau_1 \doteq \alpha\}}{E \cup \{\alpha \doteq \tau_1\}}\ \text{wenn } \tau_1 \text{ keine Typvariable und } \alpha \text{ Typvariable} \qquad \text{(Elim)}\ \frac{E \cup \{\alpha \doteq \alpha\}}{E}\ \text{wobei } \alpha \text{ Typvariable}$$

$$\text{(Solve)}\ \frac{E \cup \{\alpha \doteq \tau\}}{E[\tau/\alpha] \cup \{\alpha \doteq \tau\}}\ \text{wenn Typvariable } \alpha \text{ nicht in } \tau \text{ vorkommt, aber } \alpha \text{ kommt in } E \text{ vor} \qquad \text{(OccursCheck)}\ \frac{E \cup \{\alpha \doteq \tau\}}{\text{Fail}}\ \text{wenn } \tau \neq \alpha \text{ und Typvariable } \alpha \text{ kommt in } \tau \text{ vor}$$

Man kann zeigen, dass der Algorithmus die folgenden Eigenschaften erfüllt (wir verzichten auf einen Beweis, dieser kann in der Literatur zu Unifikation von Termen erster Ordnung gefunden werden, siehe z.B. (Baader & Nipkow, 1998)):

- Der Algorithmus endet mit Fail gdw. es keinen Unifikator für die Eingabe gibt.
- Der Algorithmus endet erfolgreich gdw. es einen Unifikator für die Eingabe gibt. Das Gleichungssystem E ist dann von der Form $\{\alpha_1 \doteq \tau_1, \ldots, \alpha_n \doteq \tau_n\}$, wobei α_i paarweise verschiedene Typvariablen sind und kein α_i in irgendeinem τ_j vorkommt. Der Unifikator kann dann abgelesen werden als $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$.
- Liefert der Algorithmus einen Unifikator, dann ist es ein allgemeinster Unifikator.
- Man braucht keine alternativen Regelanwendungen auszuprobieren! Der Algorithmus kann deterministisch implementiert werden.
- Der Algorithmus terminiert für jedes Unifikationsproblem auf Typen[3].

[3] Für den interessierten Leser skizzieren wir den Beweis: Sei E ein Unifikationsproblem und

- $Var(E)$ die Anzahl an *ungelösten* Typvariablen in E, wobei eine Variable α gelöst ist, falls sie nur einmal in E in einer Gleichung auf der linken Seite vorkommt (d.h. $E = E' \cup \{\alpha \doteq \tau\}$ wobei α weder in E' noch in τ vorkommt).
- $Size(E)$ ist die Summe alle Typtermgrößen aller Typen der rechten und linken Seiten in E, wobei die Typtermgröße `ttg` eines Typen definiert ist als: $\texttt{ttg}(TV) = 1$, $\texttt{ttg}(TC\ T_1\ \ldots\ T_n) = 1 + \sum_{i=1}^{n} \texttt{ttg}(T_i)$ und $\texttt{ttg}(T_1 \to T_2) = 1 + \texttt{ttg}(T_1) + \texttt{ttg}(T_2)$
- $OEq(E)$ ist die Anzahl der ungerichteten Gleichungen in E, wobei eine Gleichung gerichtet ist, falls sie von der Form $\alpha \doteq \tau$ ist, wobei α eine Typvariable ist.
- $M(E) = (Var(E), Size(E), OEq(E))$, wobei $M(\text{Fail}) := (-1, -1, -1)$.

Für den Terminierungsbeweis zeigen wir, dass für jede Regel $\frac{E}{E'}$ gilt: $M(E') <_{lex} M(E)$, wobei $<_{lex}$ die lexikographische Ordnung auf 3-Tupeln sei. Vorher muss man noch verifizieren, dass das Maß M wohl-fundiert ist, was jedoch einsichtig ist. Wir gehen die Regeln durch: Die Fail-Regeln und die Regel (OccursCheck) verkleinern das Maß stets. Die Decompose-Regeln vergrößern $Var(.)$ nicht, verkleinern aber stets $Size(.)$ (und vergrößern evtl. $OEq(.)$ was aber nicht stört aufgrund der lexikographischen Ordnung). Die (Orient)-Regel

- Die Typen in der Resultat-Substitution können exponentiell groß werden.
- der Unifikationsalgorithmus kann so implementiert werden, dass er Zeit $O(n * \log n)$ benötigt. Man muss Sharing dazu beachten; und dazu eine andere Solve-Regel benutzen.
- Das Unifikationsproblem (d.h. die Frage, ob eine (Multi-)Menge von Typgleichungen unifizierbar ist) ist P-complete. D.h. man kann im wesentlichen alle PTIME-Probleme als Unifikationsproblem darstellen:
  Interpretation ist: Unifikation ist nicht effizient parallelisierbar.

Beispiel 6.2.5. *Das Unifikationsproblem* $\{(a \to b) \doteq \texttt{Bool} \to \texttt{Bool}\}$ *wird durch den Unifikationsalgorithmus in einem Schritt gelöst:*

$$(\textsc{Decompose2})\ \frac{\{(a \to b) \doteq \texttt{Bool} \to \texttt{Bool}\}}{\{a \doteq \texttt{Bool}, b \doteq \texttt{Bool}\}}$$

Beispiel 6.2.6. *Betrachte das Unifikationsproblem* $\{a \to [a] \doteq \texttt{Bool} \to c, [d] \doteq c\}$. *Ein Ablauf des Unifikationsalgorithmus ist:*

$$(\textsc{Decompose1})\ \frac{(\textsc{Solve})\ \dfrac{(\textsc{Solve})\ \dfrac{(\textsc{Orient})\ \dfrac{(\textsc{Decompose2})\ \dfrac{\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}}{\{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\}}}{\{[d] \doteq c, a \doteq \texttt{Bool}, c \doteq [a]\}}}{\{[d] \doteq [a], a \doteq \texttt{Bool}, c \doteq [a]\}}}{\{[d] \doteq [\texttt{Bool}], a \doteq \texttt{Bool}, c \doteq [\texttt{Bool}]\}}$$

Der Unifikator ist $\{d \mapsto \texttt{Bool}, a \mapsto \texttt{Bool}, c \mapsto [\texttt{Bool}]\}$.

Beispiel 6.2.7. *Das Unifikationsproblem* $\{a \doteq [b], b \doteq [a]\}$ *hat keine Lösung:*

$$(\textsc{OccursCheck})\ \frac{(\textsc{Solve})\ \dfrac{\{a \doteq [b], b \doteq [a]\}}{\{a \doteq [[a]], b \doteq [a]\}}}{\mathit{Fail}}$$

Das Unifikationsproblem $\{a \to [b] \doteq a \to c \to d\}$ *hat ebenfalls keine Lösung:*

$$(\textsc{Fail2})\ \frac{(\textsc{Elim})\ \dfrac{(\textsc{Decompose2})\ \dfrac{\{a \to [b] \doteq a \to c \to d\}}{\{a \doteq a, [b] \doteq c \to d\}}}{\{[b] \doteq c \to d\}}}{\mathit{Fail}}$$

vergrößert $\mathit{Var}(.)$ nicht, lässt $\mathit{Size}(.)$ unverändert, aber verkleinert $\mathit{OEq}(.)$. Die (Elim)-Regel vergrößert $\mathit{Var}(.)$ nicht, aber verkleinert $\mathit{Size}(.)$. Die (Solve)-Regel verkleinert $\mathit{Var}(.)$ (und vergrößert $\mathit{Size}(.)$).

Übungsaufgabe 6.2.8. *Löse die Unifikationsprobleme:*

- $\{\texttt{BBaum}\ a \doteq \texttt{BBaum}\ (b \to c), [b] \doteq [[c]], d \to e \to f = \texttt{Bool} \to c\}$
- $\{a \to g \to (b \to c) \to (d \to e) \doteq a \to f, ([b] \to [c] \to [d] \to ([e] \to [g])) \to h \doteq f \to h\}$

Wir verwenden ab jetzt die Unifikation in den Beispielen ohne den Rechenweg anzugeben.

6.3. Polymorphe Typisierung für KFPTSP+seq-Ausdrücke

Wir betrachten zunächst die Typisierung von Ausdrücken, später werden wir auf die Typisierung von Superkombinatoren eingehen. Zunächst nehmen wir an, Superkombinatoren seien getypt.

Da wir nun wissen, wie Typen unifiziert werden, können wir die Regel für die Anwendung für polymorphe Typen allgemeiner formulieren:

$$\frac{s :: \tau_1, t :: \tau_2}{(s\ t) :: \sigma(\alpha)} \quad \begin{array}{l}\text{wenn } \sigma \text{ allgemeinster Unifikator für } \tau_1 \doteq \\ \tau_2 \to \alpha \text{ ist und } \alpha \text{ neue Typvariable ist.}\end{array}$$

Allerdings reicht dieses Vorgehen noch nicht aus, um Ausdrücke mit Bindern zu typisieren. Betrachte z.B. die Typisierung einer Abstraktion $\lambda x.s$. Will man $\lambda x.s$ typisieren, so muss man den Rumpf s typisieren, und dann einen entsprechenden Funktionstypen konstruieren. Z.B. für $\lambda x.\texttt{True}$ kann man `True` mit `Bool` typisieren, und dann die Abstraktion mit $\alpha \to \texttt{True}$ typisieren. Dieser Fall ist aber zu einfach, denn x kann im Rumpf s vorkommen und alle Vorkommen von x müssen gleich typisiert werden. Deswegen ist die bessere Regel von der Form:

$$\frac{\text{Typisierung von } s \text{ unter der Annahme } x \text{ hat Typ } \tau \text{ ergibt } s :: \tau'}{\lambda x.s :: \tau \to \tau'}$$

Allerdings muss man nun den richtigen Typ τ für x erraten, was (zumindest algorithmisch) wenig sinnvoll ist. Deshalb gibt man zunächst einen allgemeinsten Typ für x vor und berechnet den passenden Typen durch Unifikation. Um dies effizient zu implementieren (und nicht zu oft zu unifizieren), kann man alle Typisierungsschritte durchführen und sich dabei zu unifizierende Gleichungen merken und erst ganz am Ende eine Unifikation durchführen. Genau dieses Verfahren wenden wir an. Die Herleitungsregeln werden daher verändert: Neben einem Typ leiten wir Gleichungssysteme her (die wir mit E notieren). Die obige Typregel für Abstraktionen zeigt auch, dass wir uns Annahmen merken müssen, was zu einer weiteren Notationserweiterung führt. Die Notation ist

$$\Gamma \vdash s :: \tau, E.$$

Sie bedeutet: Unter der Annahme Γ (die eine Menge von Annahmen darstellt) kann für den Ausdruck s der Typ τ und die Typgleichungen E hergeleitet werden.

Am Anfang bestehen die Annahmen aus schon bekannten Typen: Dies sind Typen für Superkombinatoren und für die Konstruktoren. Wir verwenden hierbei all-quantifizierte polymorphe Typen, um zu verdeutlichen, dass diese Typen umbenannt werden dürfen. Z.B. können wir für `map` in die Annahme einfügen: $\texttt{map} :: \forall a, b.(a \to b) \to [a] \to [b]$. Konstruktoranwendungen werden in den folgenden Typisierungsregeln wie Anwendungen auf eine Konstante behandelt, z.B. wird `Cons True Nil` wie ((`Cons True`) `Nil`) behandelt. Die Typannahme für `Cons` ist z.B. $\texttt{Cons} :: \forall a.a \to [a] \to [a]$.

Wir geben nun Regeln zur Typisierung von KFPTSP+seq-Ausdrücken an, wobei auftretende Superkombinatoren alle als bereits typisiert angenommen werden (und daher in den Annahmen enthalten sind):

Axiom für Variablen:

$$(\text{AxV})\ \frac{}{\Gamma \cup \{x :: \tau\} \vdash x :: \tau, \emptyset}$$

Axiom für Konstruktoren:

$$(\text{AxK})\ \frac{}{\Gamma \cup \{c :: \forall \alpha_1 \ldots \alpha_n.\tau\} \vdash c :: \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n], \emptyset}\ \text{wobei } \beta_i \text{ neue Typvariablen sind}$$

Axiom für Superkombinatoren:

$$(\text{AxSK})\ \frac{}{\Gamma \cup \{SK :: \forall \alpha_1 \ldots \alpha_n.\tau\} \vdash SK :: \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n], \emptyset}\ \text{wobei } \beta_i \text{ neue Typvariablen sind}$$

Regel für Anwendungen:

$$(\text{RApp})\ \frac{\Gamma \vdash s :: \tau_1, E_1 \quad \text{und} \quad \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (s\ t) :: \alpha, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha\}}\ \text{wobei } \alpha \text{ neue Typvariable ist}$$

Regel für `seq`:

$$(\text{RSeq})\ \frac{\Gamma \vdash s :: \tau_1, E_1 \quad \text{und} \quad \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (\texttt{seq}\ s\ t) :: \tau_2, E_1 \cup E_2}$$

Regel für Abstraktionen:

$$(\text{RAbs})\ \frac{\Gamma \cup \{x :: \alpha\} \vdash s :: \tau, E}{\Gamma \vdash \lambda x.s :: \alpha \to \tau, E}\ \text{wobei } \alpha \text{ neue Typvariable ist}$$

Regel für `case`:

$$(\text{RCase})\ \frac{\begin{array}{l}\Gamma \vdash s :: \tau, E \\ \text{für alle } i = 1, \ldots, m\text{:} \\ \quad \Gamma \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,\text{ar}(c_i)} :: \alpha_{i,\text{ar}(c_i)}\} \vdash (c_i\ x_{i,1}\ \ldots\ x_{i,\text{ar}(ci)}) :: \tau_i, E_i \\ \text{für alle } i = 1, \ldots, m\text{:} \\ \quad \Gamma \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,\text{ar}(c_i)} :: \alpha_{i,\text{ar}(c_i)}\} \vdash t_i :: \tau_i', E_i'\end{array}}{\Gamma \vdash \left(\begin{array}{l}\texttt{case}_{Typ}\ s\ \texttt{of}\{ \\ \quad (c_1\ x_{1,1}\ \ldots\ x_{1,\text{ar}(c_1)}) \to t_1; \\ \quad \ldots; \\ \quad (c_m\ x_{m,1}\ \ldots\ x_{m,\text{ar}(c_m)}) \to t_m\}\end{array}\right) :: \alpha, E'}$$

$$\text{wobei } E' = E \cup \bigcup_{i=1}^{m} E_i \cup \bigcup_{i=1}^{m} E_i' \cup \bigcup_{i=1}^{m} \{\tau \doteq \tau_i\} \cup \bigcup_{i=1}^{m} \{\alpha \doteq \tau_i'\}$$

und $\alpha_{i,j}, \alpha$ neue Typvariablen sind

Damit lässt sich der Typ eines Ausdrucks s wie folgt berechnen:

Algorithmus 6.3.1 (Typisierung von KFPTSP+seq-Ausdrücken)**.** *Sei s ein geschlossener*

KFPTSP+seq-*Ausdruck, wobei die Typen für alle in s benutzten Superkombinatoren und Konstruktoren bekannt sind (d.h. schon berechnet wurden)*

1. *Starte mit Anfangsannahme Γ, die Typen für die Konstruktoren und die Superkombinatoren enthält.*
2. *Leite $\Gamma \vdash s :: \tau, E$ mit den Typisierungsregeln her.*
3. *Löse E mit Unifikation.*
4. *Wenn die Unifikation mit Fail endet, ist s nicht typisierbar; Andernfalls: Sei σ ein allgemeinster Unifikator von E, dann gilt $s :: \sigma(\tau)$.*

Als Optimierung kann man zu obigen Herleitungsregeln hinzufügen:

Typberechnung:

$$(\text{RUnif})\ \frac{\Gamma \vdash s :: \tau, E}{\Gamma \vdash s :: \sigma(\tau), E_\sigma}$$

wobei E_σ das gelöste Gleichungssystem zu E ist
und σ der ablesbare Unifikator ist

Damit kann man jederzeit zwischendrin schon einmal unifizieren.

Definition 6.3.2. *Ein (*KFPTSP+seq*) Ausdruck s ist* wohl-getypt*, wenn er sich mit obigem Verfahren typisieren lässt.*

Wir betrachten einige Beispiele:

Beispiel 6.3.3. *Betrachte die Konstruktoranwendung* `Cons True Nil`*. In der Anfangsannahme sind die Typen für* `Cons`*,* `Nil` *und* `True` *enthalten:* $\Gamma_0 = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} :: \forall a.[a], \texttt{True} :: \texttt{Bool}\}$.

Zur Typisierung von `Cons True Nil` *fängt man von unten nach oben an, den Herleitungsbaum aufzubauen, d.h. der erste Schritt ist:*

$$(\text{RApp})\ \frac{\Gamma_0 \vdash (\texttt{Cons True}) :: \tau_1, E_1, \quad \Gamma_0 \vdash \texttt{Nil} :: \tau_2, E_2}{\Gamma_0 \vdash \texttt{Cons True Nil} :: \alpha_4, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha_4\}}$$

Die genauen Typen τ_1, τ_2 und die Gleichungssysteme E_1, E_2 erhält man erst beim nach oben Steigen im Herleitungsbaum. Insgesamt muss man zweimal die Anwendungsregel und dreimal das Axiom für Konstanten anwenden und erhält dann den vollständigen Herleitungsbaum:

$$\cfrac{\cfrac{(\text{AxK})\ \overline{\Gamma_0 \vdash \texttt{Cons} :: \alpha_1 \to [\alpha_1] \to [\alpha_1], \emptyset}, \quad (\text{AxK})\ \overline{\Gamma_0 \vdash \texttt{True} :: \texttt{Bool}, \emptyset}}{(\text{RApp})\ \Gamma_0 \vdash (\texttt{Cons True}) :: \alpha_2, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \texttt{Bool} \to \alpha_2\}}, \quad (\text{AxK})\ \overline{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_3], \emptyset}}{(\text{RApp})\ \Gamma_0 \vdash (\texttt{Cons True Nil}) :: \alpha_4, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \texttt{Bool} \to \alpha_2, \alpha_2 \doteq [\alpha_3] \to \alpha_4\}}$$

Um den Typ von (`Cons True Nil`) *zu berechnen, muss man noch unifizieren. Das ergibt:*

$$\sigma = \{\alpha_1 \mapsto \texttt{Bool}, \alpha_2 \mapsto [\texttt{Bool}] \to [\texttt{Bool}], \alpha_3 \mapsto \texttt{Bool}, \alpha_4 \mapsto [\texttt{Bool}]\}$$

und daher $(\texttt{Cons True Nil}) :: \sigma(\alpha_4) = [\texttt{Bool}]$.

Beispiel 6.3.4. *Der Ausdruck* $\Omega := (\lambda x.(x\ x))\ (\lambda y.(y\ y))$ *ist* nicht *wohl-getypt: Die Anfangsannahme ist leer, da keine Konstruktoren oder Superkombinatoren vorkommen.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}(\text{AxV}), \dfrac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}(\text{AxV})}{\{x :: \alpha_2\} \vdash (x\ x) :: \alpha_4, \{\alpha_2 \doteq \alpha_2 \to \alpha_4\}}(\text{RApp})}{\emptyset \vdash (\lambda x.(x\ x)) :: \alpha_2 \to \alpha_4, \{\alpha_2 \doteq \alpha_2 \to \alpha_4\}}(\text{RAbs}), \dfrac{\dfrac{\dfrac{}{\{y :: \alpha_3\} \vdash y :: \alpha_3, \emptyset}(\text{AxV}), \dfrac{}{\{y :: \alpha_3\} \vdash y :: \alpha_3, \emptyset}(\text{AxV})}{\{y :: \alpha_3\} \vdash (y\ y) :: \alpha_5, \{\alpha_3 \doteq \alpha_3 \to \alpha_5\}}(\text{RApp})}{\emptyset \vdash (\lambda y.(y\ y)) :: \alpha_3 \to \alpha_5, \{\alpha_3 \doteq \alpha_3 \to \alpha_5\}}(\text{RAbs})}{\emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, \{\alpha_2 \doteq \alpha_2 \to \alpha_4, \alpha_3 \doteq \alpha_3 \to \alpha_5, \alpha_2 \to \alpha_4 \doteq (\alpha_3 \to \alpha_5) \to \alpha_1\}}(\text{RApp})$$

Die Unifikation schlägt jedoch fehl:

$$(\text{OccursCheck})\ \frac{\{\alpha_2 \doteq \alpha_2 \to \alpha_4, \ldots\}}{\mathit{Fail}}$$

Übungsaufgabe 6.3.5. *Zeige, dass der Fixpunktkombinator*

$$Y := \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

nicht typisierbar ist.

Beispiel 6.3.6. *Seien* `map` *und* `length` *bereits typisierte Superkombinatoren. Wir typisieren den Ausdruck*

$$t := \lambda xs.\texttt{case}_{\mathit{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{map length}\ ys\}$$

Als Anfangsannahme benutzen wir:

$$\begin{aligned}\Gamma_0 = \{&\texttt{map} :: \forall a, b.(a \to b) \to [a] \to [b],\\ &\texttt{length} :: \forall a.[a] \to \texttt{Int},\\ &\texttt{Nil} :: \forall a.[a]\\ &\texttt{Cons} :: \forall a.a \to [a] \to [a]\}\end{aligned}$$

Zur Typisierung fängt man im Herleitungsbaum von unten an, und arbeitet sich dann nach oben, d.h. die unterste Regel ist (RAbs)*, da* t *eine Abstraktion ist:*

$$(\text{RAbs})\ \frac{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash \texttt{case}_{\mathit{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to 1; (\texttt{Cons}\ y\ ys) \to \texttt{map length}\ ys\} :: \tau, E}{\Gamma_0 \vdash \alpha_1 \to \tau, E}$$

Zu diesem Zeitpunkt kennt man den Typ τ *und das Gleichungssystem* E *noch nicht. Diese ergeben sich beim Berechnen. Der nächste Schritt ist nun die Voraussetzung zu zeigen, wofür man die* (RCase)*-Regel anwenden muss usw.*

Aus Platzgründen geben wir den Herleitungsbaum und seine Beschriftung getrennt an:

$$\dfrac{\dfrac{\dfrac{}{B_3}(\text{AxV}), \dfrac{}{B_4}(\text{AxK}), \dfrac{\dfrac{\dfrac{}{B_8}(\text{AxK}), \dfrac{}{B_9}(\text{AxV})}{B_6}(\text{RApp}), \dfrac{}{B_7}(\text{AxV})}{B_5}(\text{RApp}), \dfrac{}{B_{10}}(\text{AxK}), \dfrac{\dfrac{\dfrac{}{B_{14}}(\text{AxSK}), \dfrac{}{B_{15}}(\text{AxSK})}{B_{12}}(\text{RApp}), \dfrac{}{B_{13}}(\text{AxV})}{B_{11}}(\text{RApp})}{B_2}(\text{RCase})}{B_1}(\text{RAbs})$$

Die Beschriftungen sind dabei:

$$
\begin{aligned}
B_1 =& \Gamma_0 \vdash t :: \alpha_1 \to \alpha_{13},\\
&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\
&(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\
&\alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12},\}\\
B_2 =& \Gamma_0 \cup \{xs :: \alpha_1\} \vdash\\
&\quad \texttt{case}_{List}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{map length}\ ys\} :: \alpha_{13},\\
&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\
&(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\
&\alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12},\}\\
B_3 =& \Gamma_0 \cup \{xs :: \alpha_1\} \vdash\ xs :: \alpha_1, \emptyset\\
B_4 =& \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: [\alpha_2], \emptyset\\
B_5 =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{Cons}\ y\ ys) :: \alpha_7,\\
&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7\}\\
B_6 =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{Cons}\ y) :: \alpha_6,\\
&\quad \{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6\}\\
B_7 =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset\\
B_8 =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset\\
B_9 =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash y :: \alpha_3, \emptyset\\
B_{10} =& \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: [\alpha_{14}], \emptyset\\
B_{11} =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{map length})\ ys :: \alpha_{12},\\
&\quad \{(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12}\}\\
B_{12} =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{map length}) :: \alpha_{11},\\
&\quad \{(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}\}\\
B_{13} =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset\\
B_{14} =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{map} :: (\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9], \emptyset\\
B_{15} =& \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{length} :: [\alpha_{10}] \to \texttt{Int}, \emptyset
\end{aligned}
$$

Um den Typ von t zu berechnen müssen wir das Gleichungssystem

$$
\begin{aligned}
&\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\
&(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\
&\alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}\}
\end{aligned}
$$

noch unifizieren. Man erhält dann als Lösung die Substitution:

$$\begin{aligned}\sigma \;=\; \{&\alpha_1 \mapsto [[\alpha_{10}]], \alpha_2 \mapsto [\alpha_{10}], \alpha_3 \mapsto [\alpha_{10}], \alpha_4 \mapsto [[\alpha_{10}]], \alpha_5 \mapsto [\alpha_{10}],\\ &\alpha_6 \mapsto [[\alpha_{10}]] \to [[\alpha_{10}]], \alpha_7 \mapsto [[\alpha_{10}]], \alpha_8 \mapsto [\alpha_{10}], \alpha_9 \mapsto \texttt{Int},\\ &\alpha_{11} \mapsto [[\alpha_{10}]] \to [\texttt{Int}], \alpha_{12} \mapsto [\texttt{Int}], \alpha_{13} \mapsto [\texttt{Int}], \alpha_{14} \mapsto \texttt{Int}\}\end{aligned}$$

Damit erhält man $t :: \sigma(\alpha_1 \to \alpha_{13}) = [[\alpha_{10}]] \to [\texttt{Int}]$.

Beispiel 6.3.7. *Die Funktion (bzw. der Superkombinator)* `const` *ist definiert als*

```
const :: a -> b -> a
const x y = x
```

Der Ausdruck $\lambda x.$`const` $(x$ `True`$)$ $(x$ `'A'`$)$ *ist nicht wohl-getypt: Sei* $\Gamma_0 = \{$`const` $:: \forall a, b.a \to b \to a,$ `True` $::$ `Bool`, `'A'` $::$ `Char`$\}$.

Sei $\Gamma_1 = \Gamma_0 \cup \{x :: \alpha_1\}$. *Die Typisierung ergibt:*

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \texttt{const} :: \alpha_2 \to \alpha_3 \to \alpha_2, \emptyset}\,(\text{AxSK}) \quad \dfrac{\dfrac{}{\Gamma_1 \vdash x :: \alpha_1}\,(\text{AxV}) \quad \dfrac{}{\Gamma_1 \vdash \texttt{True} :: \texttt{Bool}}\,(\text{AxK})}{\Gamma_1 \vdash (x\ \texttt{True}) :: \alpha_4, E_1}\,(\text{RApp})}{\Gamma_1 \vdash \texttt{const}\ (x\ \texttt{True}) :: \alpha_5, E_2}\,(\text{RApp}) \quad , \quad \dfrac{\dfrac{}{\Gamma_1 \vdash x :: \alpha_1}\,(\text{AxV}) \quad \dfrac{}{\Gamma_1 \vdash \texttt{'A'} :: \texttt{Char}}\,(\text{AxK})}{\Gamma_1 \vdash (x\ \texttt{'A'}) :: \alpha_6, E_3}\,(\text{RApp})}{\Gamma_1 \vdash \texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'}) :: \alpha_7, E_4}\,(\text{RApp})$$
$$\overline{\Gamma_0 \vdash \lambda x.\texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'}) :: \alpha_1 \to \alpha_7, E_4}\,(\text{RAbs})$$

wobei:

$$\begin{aligned}E_1 &= \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4\}\\ E_2 &= \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4, \alpha_2 \to \alpha_3 \to \alpha_2 \doteq \alpha_4 \to \alpha_5\}\\ E_3 &= \{\alpha_1 \doteq \texttt{Char} \to \alpha_6\}\\ E_4 &= \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4, \alpha_2 \to \alpha_3 \to \alpha_2 \doteq \alpha_4 \to \alpha_5, \alpha_1 \doteq \texttt{Char} \to \alpha_6, \alpha_5 \doteq \alpha_6 \to \alpha_7\}\end{aligned}$$

Die Unifikation schlägt fehl, da die beiden Gleichungen für α_1 *nicht zueinander passen.*

Auch in Haskell erhält man den entsprechenden Typfehler:

```
Main> \x -> const (x True)  (x 'A')

<interactive>:1:23:
Couldn't match expected type `Char' against inferred type `Bool'
      Expected type: Char -> b
      Inferred type: Bool -> a
 In the second argument of `const', namely `(x 'A')'
 In the expression: const (x True) (x 'A')
```

Das letzte Beispiel verdeutlicht, dass das Typisierungsverfahren Lambda-gebundene Variablen monomorph behandelt: Für eine Abstraktion $\lambda x.s$ darf x im Rumpf nur mit dem gleichen Typ verwendet werden. Beachte: Für den Ausdruck $\lambda x.$`const` $(x$ `True`$)$ $(x$ `'A'`$)$ gibt es durchaus sinnvolle Argumente, z.B. eine Funktion die ihr Argument ignoriert und konstant auf einen Wert abbildet. Da Lambda-gebundene Variablen nur monomorph typisiert werden dürfen (und Patternvariablen in `case`-Alternativen genauso) spricht man auch von `let`-*Polymorphismus*, da nur `let`-gebundene Variablen (die in KFPTSP+seq nur als Superkombinatoren auftreten) wirklich polymorph typisiert werden.

Würde man versuchen, das Typsystem dahingehend anzupassen, dass obiger Ausdruck typisierbar ist, so müsste der Typ für die Variable x in etwa aussagen: Lasse nur Ausdrücke zu, die Funktionen sind, die für beliebigen Argumenttypen den gleichen Ergebnistyp liefern. Unser Typsystem kann eine solche Bedingung nicht ausdrücken, da sie nicht zu unserer Mengensemantik `sem`(.) passt. Man spricht hierbei auch von *prädikativem* Polymorphismus, da Typvariablen nur für Grundtypen stehen, aber nicht selbst für polymorphe Typen (wie dies in so genanntem imprädikativen Polymorphismus der Fall ist).

In erweiterten Typsystemen (mit imprädikativem Polymorphismus) kann man obigen Ausdruck typisieren als

```
(\x -> const (x True) (x 'A'))::(forall b.(forall a. a -> b) -> b)
```

Beachte, dass der innere Allquantor in unserer Typsyntax nicht erlaubt ist. Die Typisierung des Ausdrucks funktioniert auch im `ghci`, wenn man Typsystemerweiterungen anschaltet. Beachte: Lässt man solche Erweiterungen zu, erhält der Programmierer mehr Freiheit, die Typinferenz ist im Allgemeinen jedoch nicht mehr möglich, d.h. der Programmierer muss Typen vorgeben.

6.4. Typisierung von nicht-rekursiven Superkombinatoren

Bisher haben wir nur KFPTSP+seq-Ausdrücke typisiert und uns nicht um die Typisierung von Superkombinatoren gekümmert. Wir betrachten in diesem Abschnitt den einfachsten Fall: Wir betrachten Superkombinatoren, die sich selbst nicht im Rumpf aufrufen und auch nicht verschränkt rekursiv sind.

Sei $\mathcal{SK}$ eine Menge von Superkombinatoren. Für $SK_i, SK_j \in \mathcal{SK}$ sei $SK_i \preceq SK_j$ genau dann, wenn SK_j den Superkombinator SK_i im Rumpf benutzt. Sei $\preceq^+$ der transitive Abschluss von $\preceq$ ($\preceq^*$ der reflexiv-transitive Abschluss).

Ein Superkombinator SK_i ist direkt rekursiv, wenn $SK_i \preceq SK_i$ gilt. Er ist rekursiv, wenn $SK_i \preceq^+ SK_i$ gilt.

Eine Teilmenge $\{SK_1, \ldots, SK_m\} \subseteq \mathcal{SK}$ von Superkombinatoren ist verschränkt rekursiv, wenn $SK_i \preceq^+ SK_j$ für alle $i, j \in \{1, \ldots, m\}$ gilt.

Nicht-rekursive Superkombinatoren kann man analog zu Abstraktionen typisieren (andere im Rumpf des Superkombinators benutzte Superkombinatoren müssen bereits getypt sein, damit deren Typ in der Annahme verfügbar ist). Wir benutzen die Notation $\Gamma \vdash_T SK :: \tau$, wenn man für den Superkombinator SK den Typ τ unter der Annahme Γ herleiten kann[4]. Die Regel für nicht rekursive Superkombinatoren ist:

$$\text{(RSK1)}\ \frac{\Gamma \cup \{x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SK :: \forall \mathcal{X}.\sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)}$$

wenn σ Lösung von E, $SK\ x_1\ \ldots\ x_n = s$ die Definition von SK, SK nicht rekursiv ist und $\mathcal{X}$ die Typvariablen in $\sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)$ sind

Beispiel 6.4.1. *Wir betrachten die Typisierung der Komposition. Er ist definiert als*

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

[4] Im Gegensatz zur Notation $\Gamma \vdash s :: \tau, E$ hat die Notation $\Gamma \vdash_T SK :: \tau$ kein Gleichungssystem als Komponente

Die Anfangsannahme Γ_0 *ist leer, da* `(.)` *im Rumpf keine Konstruktoren und keine anderen Superkombinatoren benutzt. Sei* $\Gamma_1 = \{f :: \alpha_1, g :: \alpha_2, x :: \alpha_3\}$*. Die Typherleitung für* `(.)` *ist:*

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash f :: \alpha_1, \emptyset}\,(\text{AxV}) \quad \dfrac{\dfrac{}{\Gamma_1 \vdash g :: \alpha_2, \emptyset}\,(\text{AxV}) \quad \dfrac{}{\Gamma_1 \vdash x :: \alpha_3, \emptyset}\,(\text{AxV})}{\Gamma_1 \vdash (g\ x) :: \alpha_5, \{\alpha_2 \doteq \alpha_3 \to \alpha_5\}}\,(\text{RApp})}{\Gamma_1 \vdash (f\ (g\ x)) :: \alpha_4, \{\alpha_2 \doteq \alpha_3 \to \alpha_5, \alpha_1 = \alpha_5 \to \alpha_4\}}\,(\text{RApp})}{\emptyset \vdash_T \texttt{(.)} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4) = (\alpha_5 \to \alpha_4) \to (\alpha_3 \to \alpha_5) \to \alpha_3 \to \alpha_4}\,(\text{RSK1})$$

Hierbei ist $\sigma = \{\alpha_2 \mapsto \alpha_3 \to \alpha_5, \alpha_1 \mapsto \alpha_5 \to \alpha_4\}$*. Den erhaltenen Typ darf man nun allquantifizieren (und umbenennen) und anschließend zur Typisierung von Superkombinatoren benutzen, die* `(.)` *verwenden. Z.B.*

$$\texttt{(.)} :: \forall a, b, c.(a \to b) \to (c \to a) \to c \to b$$

6.5. Typisierung rekursiver Superkombinatoren

Für die Typisierung rekursiver Superkombinatoren, bzw. einer Gruppe von verschränkt rekursiven Superkombinatoren ergibt sich zunächst das Problem, dass man beim Typisieren des Rumpfs einer entsprechenden Superkombinatordefinition den Typ des gerade eben zu typisierenden Superkombinators nicht kennt. Deshalb fängt man mit den allgemeinsten Annahmen an und versucht diese so einzuschränken, bis man eine so genannte *konsistente Annahme* erreicht hat. Wir werden zwei Verfahren erörtern.

6.5.1. Das iterative Typisierungsverfahren

Das iterative Typisierungsverfahren fängt mit allgemeinsten Annahmen an, und iteriert den Typcheck (und berechnet in jedem Iterationsschritt eine neue Annahme) so lange bis die Annahme konsistent ist, d.h. aus der Annahme mittels Typisierung genau die gleiche Annahme folgt. Die Regel für die Typisierung eines Superkombinators ist analog zur Regel für nicht-rekursive Superkombinatoren:

$$(\text{SKRek})\ \frac{\Gamma \cup \{x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SK :: \sigma(\alpha_1 \to \ldots \alpha_n \to \tau)}\ \begin{array}{l}\text{wenn } SK\ x_1\ \ldots\ x_n = s \text{ die Definition von } SK,\\ \sigma \text{ Lösung von } E\end{array}$$

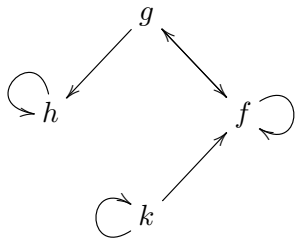
Der Unterschied liegt darin, dass die Annahme Γ bereits einen Typ für SK (bzw. alle zugehörigen verschränkt rekursiven Superkombinatoren) enthält (siehe Algorithmus 6.5.1).

Global gesehen wird man für eine Menge von Superkombinatordefinitionen zunächst eine Abhängigkeitsanalyse durchführen, die im Aufrufgraph die starken Zusammenhangskomponenten berechnet. Genauer: Sei $\simeq$ die Äquivalenzrelation passend zu $\preceq^*$, dann sind die starken Zusammenhangskomponenten gerade die Äquivalenzklassen zu $\simeq$. Anschließend fängt man mit den kleinsten Elementen bzgl. der Ordnung auf $\simeq$ an (diese hängen nicht von anderen Superkombinatoren ab) und arbeitet sich entsprechend der Abhängigkeiten voran. Wichtig ist, dass man stets eine Gruppe von verschränkt rekursiven Superkombinatoren (d.h. alle Superkombinatoren einer Äquivalenzklasse zu $\simeq$) während der Typisierung gemeinsam behandeln muss.

Betrachte z.B. die Superkombinatoren:

```
f x y = if x<=1 then y else f (x-y) (y + g x)
g x   = if x==0 then (f 1 x) + (h 2) else 10
h x   = if x==1 then 0 else h (x-1)
k x y = if x==1 then y else k (x-1) (y+(f x y))
```

Der Aufrufgraph ist



Die Äquivalenzklassen (mit Ordnung) sind $\{h\} \preceq^+ \{f, g\} \preceq^+ \{k\}$. Man typisiert daher zuerst h alleine, dann f und g zusammen und schließlich zuletzt k.

Das *iterative Typisierungsverfahren* geht wie folgt vor:

Algorithmus 6.5.1 (Iterativer Typisierungsalgorithmus)**.** *Als Eingabe erhält man eine Menge von verschränkt rekursiven Superkombinatoren $SK_1, \ldots, SK_m$ wobei alle anderen in den Definitionen verwendeten Superkombinatoren bereits typisiert sind.*

1. *Die Anfangsannahme Γ besteht aus den Typen der Konstruktoren und den Typen der bereits bekannten Superkombinatoren.*
2. $\Gamma_0 := \Gamma \cup \{SK_1 :: \forall\alpha_1.\alpha_1, \ldots, SK_m :: \forall\alpha_m.\alpha_m\}$ *und* $j = 0$.
3. *Verwende für jeden Superkombinator SK_i (mit $i = 1, \ldots, m$) die Regel* (SKRek) *und Annahme Γ_j, um SK_i zu typisieren.*
4. *Wenn die m Typisierungen erfolgreich waren, d.h.*

$$\Gamma_j \vdash_T SK_1 :: \tau_1, \ldots, \Gamma_j \vdash_T SK_m :: \tau_m.$$

   Dann allquantifiziere die Typen τ_i indem alle Typvariablen quantifiziert werden. O.B.d.A ergebe dies $SK_1 :: \forall\mathcal{X}_1.\tau_1, \ldots, SK_m :: \forall\mathcal{X}_m.\tau_m$. Sei

$$\Gamma_{j+1} := \Gamma \cup \{SK_1 :: \forall\mathcal{X}_1.\tau_1, \ldots, SK_m :: \forall\mathcal{X}_m.\tau_m\}$$

5. *Wenn $\Gamma_j \neq \Gamma_{j+1}$ (wobei Gleichheit auf Typen die Gleichheit bis auf Umbenennung meint), dann gehe mit $j := j + 1$ zu Schritt* (3)*. Anderenfalls ($\Gamma_j = \Gamma_{j+1}$) war Γ_j konsistent, die Typen der SK_i sind entsprechend in Γ_j zu finden.*

Sollte irgendwann ein Fail in der Unifikation auftreten, dann sind $SK_1, \ldots, SK_m$ nicht typisierbar.

Beachte: Die Annahme am Anfang in Schritt (2) ist die *allgemeinste* die man treffen kann, denn der polymorphe Typ $\forall a.a$ steht für jeden beliebigen Typen ($\texttt{sem}(a)$ ist die Menge aller Grundtypen).

Folgende Eigenschaften gelten für den Algorithmus (wir verzichten auf einen Beweis):

- Die berechneten Typen pro Iterationsschritt sind eindeutig bis auf Umbenennung. Daher folgt: Wenn das Verfahren terminiert, sind die berechneten Typen der Superkombinatoren eindeutig.

- In jedem Schritt werden die neu berechneten Typ spezieller (oder bleiben unverändert). D.h. bzgl. der Grundtypensemantik ist das Verfahren monoton (die Mengen von Grundtypen werden immer kleiner).
- Wenn das Verfahren nicht terminiert (endlos läuft), dann gibt es keinen polymorphen Typ für die zu typisierenden Superkombinatoren. Das folgt, da die Typen dann immer spezieller werden (und daher die Mengen der Grundtypensemantik immer kleiner werden) und da man mit den allgemeinsten Annahmen angefangen hat. Man muss noch ausschließen, dass der Fixpunkt bzgl der Grundtypensemantik nicht die leere Menge ist: Jeder Iterationsschritt berechnet ja einen polymorphen Typ τ, den man syntaktisch hinschreiben kann. Es ist leicht nachzuweisen, dass man bei syntaktisch gleich großen Typen nur eine endliche Kette bekommen kann. Also muss eine unendliche Iteration dann irgendwann einen syntaktisch größeren polymorphen Typ erzeugen. Bei nichtterminierendem Iterationsverfahren werden die Typen also syntaktisch immer größer, also kann es keinen Grundtyp geben der in allen polymorphen Typen, der von der Iteration erzeugt wird, enthalten ist.
- Das iterative Verfahren berechnet einen *größten Fixpunkt* (bzgl. der Grundtypensemantik): Man startet mit der Menge aller Grundtypen und schränkt die Menge solange durch iteriertes Typisieren ein, bis sich die Menge nicht mehr verkleinert. D.h. es wird der *allgemeinste* polymorphe Typ berechnet.

6.5.2. Beispiele für die iterative Typisierung und Eigenschaften des Verfahrens

Wir betrachten einige Beispiele, wobei wir meistens nur einen rekursiven Superkombinator (und keine verschränkt rekursiven) betrachten.

Beispiel 6.5.2. *Wir typisieren die Funktion* `length`*, die definiert ist als:*

$$\texttt{length}\ xs = \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ ys\}$$

Die Anfangsannahme für die Konstruktoren und $(+)$ *ist:* $\Gamma = \{\texttt{Nil} :: \forall a.[a], (:) :: \forall a.a \to [a] \to [a], 0 :: \texttt{Int}, 1 :: \texttt{Int}, (+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}\}$*. Der iterative Typisierungsalgorithmus nimmt am Anfang (für* $j = 0$*) den allgemeinsten Typ für* `length` *an, d.h.* $\Gamma_0 = \Gamma \cup \{\texttt{length} :: \forall\alpha.\alpha\}$ *und verwendet nun die* (SKRek)*-Regel um* `length` *im ersten Iterationsschritt zu typisieren:*

$$\dfrac{\dfrac{\begin{array}{ll}(a) & \Gamma_0 \cup \{xs :: \alpha_1\} \vdash xs :: \tau_1, E_1\\ (b) & \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: \tau_2, E_2\\ (c) & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (y : ys) :: \tau_3, E_3\\ (d) & \Gamma_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \tau_4, E_4\\ (e) & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}\} \vdash (1 + \texttt{length}\ ys) :: \tau_5, E_5\end{array}}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash (\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ xs\}) :: \alpha_3,\ E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha_3 \doteq \tau_4, \alpha_3 \doteq \tau_5\}}\ (\text{RCase})}{\Gamma_0 \vdash_T \texttt{length} :: \sigma(\alpha_1 \to \alpha_3)}\ (\text{SKRek})$$

wobei σ *Lösung von* $E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha_3 \doteq \tau_4, \alpha_3 \doteq \tau_5\}$

Zur besseren Darstellung zeigen wir die Voraussetzungen (a) *bis* (e) *getrennt (die Typen* $\tau_1, \ldots, \tau_5$ *und die Gleichungen* $E_1, \ldots, E_5$ *werden dabei berechnet):*

(a): $\dfrac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash xs :: \alpha_1, \emptyset}\ (\text{AxV})$
D.h $\tau_1 = \alpha_1$ *und* $E_1 = \emptyset$

(b): $\dfrac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash \texttt{Nil} :: [\alpha_6], \emptyset}\ (\text{AxK})$
D.h. $\tau_2 = [\alpha_6]$ *und* $E_2 = \emptyset$

(c)
$$\dfrac{\dfrac{\dfrac{}{\Gamma_0' \vdash (:) :: \alpha_9 \to [\alpha_9] \to [\alpha_9], \emptyset}\ (\text{AxK}) \quad \dfrac{}{\Gamma_0' \vdash y :: \alpha_4, \emptyset}\ (\text{AxV})}{\Gamma_0' \vdash ((:)\ y) :: \alpha_8, \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8\}}\ (\text{RApp}) \quad \dfrac{}{\Gamma_0' \vdash ys :: \alpha_5, \emptyset}\ (\text{AxV})}{\Gamma_0' \vdash (y : ys) :: \alpha_7, \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}}\ (\text{RApp})$$
wobei $\Gamma_0 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$
D.h. $\tau_3 = \alpha_7$ *und* $E_3 = \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}$

(d) $\dfrac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \texttt{Int}, \emptyset}\ (\text{AxK})$
D.h. $\tau_4 = \texttt{Int}$ *und* $E_4 = \emptyset$

(e)
$$\dfrac{\dfrac{\dfrac{}{\Gamma_0' \vdash (+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}, \emptyset}\ (\text{AxK}) \quad \dfrac{}{\Gamma_0' \vdash 1 :: \texttt{Int}, \emptyset}\ (\text{AxK})}{\Gamma_0' \vdash ((+)\ 1) :: \alpha_{11}, \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}\}}\ (\text{RApp}) \quad \dfrac{\dfrac{}{\Gamma_0' \vdash \texttt{length} :: \alpha_{13}, \emptyset}\ (\text{AxSK}) \quad \dfrac{}{\Gamma_0' \vdash (ys) :: \alpha_5, \emptyset}\ (\text{AxV})}{\Gamma_0' \vdash (\texttt{length}\ ys) :: \alpha_{12}, \{\alpha_{13} \doteq \alpha_5 \to \alpha_{12}\}}\ (\text{RApp})}{\Gamma_0' \vdash (1 + \texttt{length}\ ys) :: \alpha_{10}, \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}}\ (\text{RApp})$$
wobei $\Gamma_0' = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$
D.h. $\tau_5 = \alpha_{10}$ *und*
$E_5 = \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}$

Setzt man die erhaltenen Werte ein, ergibt das für die Anwendung der (SKRek)*-Regel:*

$$\dfrac{\begin{array}{l}\Gamma_0 \cup \{xs :: \alpha_1\} \vdash (\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ xs\}) :: \alpha_3, \\ \quad \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\} \\ \cup \{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\} \\ \cup \{\alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \texttt{Int}, \alpha_3 \doteq \alpha_{10}\}\end{array}}{\begin{array}{c}\Gamma_0 \vdash_T \texttt{length} :: \sigma(\alpha_1 \to \alpha_3) \\ \textit{wobei } \sigma \textit{ Lösung von} \\ \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7, \\ \texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}, \\ \alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \texttt{Int}, \alpha_3 \doteq \alpha_{10}\}\end{array}}\ (\text{SKRek})$$

Die Unifikation ergibt als Unifikator

$$\{\alpha_1 \mapsto [\alpha_9], \alpha_3 \mapsto \texttt{Int}, \alpha_4 \mapsto \alpha_9, \alpha_5 \mapsto [\alpha_9], \alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto [\alpha_9], \alpha_8 \mapsto [\alpha_9] \to [\alpha_9],$$
$$\alpha_{10} \mapsto \texttt{Int}, \alpha_{11} \mapsto \texttt{Int} \to \texttt{Int}, \alpha_{12} \mapsto \texttt{Int}, \alpha_{13} \mapsto [\alpha_9] \to \texttt{Int}\}$$

Der neue berechnete Typ von `length` *ist daher* $\sigma(\alpha_1 \to \alpha_3\} = [\alpha_9] \to \texttt{Int}$, *und* $\Gamma_1 = \Gamma \cup \{\texttt{length} :: \forall\alpha.[\alpha] \to \texttt{Int}\}$. *Da* $\Gamma_0 \neq \Gamma_1$ *muss man mit* Γ_1 *erneut iterieren.*

Wir führen die erneute Typisierung nicht komplett auf: Die einzige Stelle, wo sich etwas ändert ist im Teil (e): statt

$$(\text{AxSK})\ \frac{}{\Gamma_0' \vdash \texttt{length} :: \alpha_{13}, \emptyset}$$

erhält man

$$(\text{AxSK})\ \frac{}{\Gamma_1' \vdash \texttt{length} :: [\alpha_{13}] \to \texttt{Int}, \emptyset}$$

Natürlich muss man die Gleichungssysteme entsprechend anpassen und Γ_0 *durch* Γ_1 *ersetzen. Die Verwendung der* (SKREK)*-Regel ergibt:*

$$(\text{SKREK})\ \frac{\begin{array}{l}\Gamma_1 \cup \{xs :: \alpha_1\} \vdash (\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to 0; (y : ys) \to 1 + \texttt{length}\ xs\}) :: \alpha_3,\\ \quad \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7\}\\ \cup\{\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, : [\alpha_{13}] \to \texttt{Int} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10}\}\\ \cup\{\alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \texttt{Int}, \alpha_3 \doteq \alpha_{10}\}\end{array}}{\begin{array}{c}\Gamma_1 \vdash_T \texttt{length} :: \sigma(\alpha_1 \to \alpha_3)\\ \textit{wobei } \sigma \textit{ Lösung von}\\ \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7,\\ \texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, [\alpha_{13}] \to \texttt{Int} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10},\\ \alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \texttt{Int}, \alpha_3 \doteq \alpha_{10}\}\end{array}}$$

Die Unifikation ergibt für σ*:*

$$\{\alpha_1 \mapsto [\alpha_9], \alpha_3 \mapsto \texttt{Int}, \alpha_4 \mapsto \alpha_9, \alpha_5 \mapsto [\alpha_9], \alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto [\alpha_9], \alpha_8 \mapsto [\alpha_9] \to [\alpha_9],$$
$$\alpha_{10} \mapsto \texttt{Int}, \alpha_{11} \mapsto \texttt{Int} \to \texttt{Int}, \alpha_{12} \mapsto \texttt{Int}, \alpha_{12} \mapsto \texttt{Int}, \alpha_{13} \mapsto \alpha_9\}$$

Daher ergibt dies $\texttt{length} :: \sigma(\alpha_1 \to \alpha_3) = [\alpha_9] \to \texttt{Int}$ *und* $\Gamma_2 = \Gamma \cup \{\texttt{length} :: \forall\alpha.[\alpha] \to \texttt{Int}\}$. *Da* $\Gamma_1 = \Gamma_2$ *ist* Γ_1 *eine konsistente Annahme und* $[\alpha] \to \texttt{Int}$ *der iterative Typ von* `length`.

6.5.2.1. Das iterative Verfahren ist allgemeiner als Haskell

Beispiel 6.5.3. *Wir typisieren den Superkombinator* `g` *mit der Definition*

```
g x = 1 : (g (g 'c'))
```

Die Anfangsannahme ist $\Gamma = \{1 :: \texttt{Int}, \texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$. *Der iterative Typisierungsalgorithmus startet mit* $\Gamma_0 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha\}$ *und typisiert* `g` *mit der* (SKREK)*-Regel (sei* $\Gamma_0' = \Gamma_0 \cup \{x :: \alpha_1\}$*):*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_0' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,(\text{AxK}) \quad \dfrac{}{\Gamma_0' \vdash 1 :: \texttt{Int}, \emptyset}\,(\text{AxK})}{\Gamma_0' \vdash (\texttt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3}\,(\text{RApp}) \quad , \quad \dfrac{\dfrac{}{\Gamma_0' \vdash \texttt{g} :: \alpha_6, \emptyset}\,(\text{AxSK}) \quad \dfrac{\dfrac{}{\Gamma_0' \vdash \texttt{g} :: \alpha_8, \emptyset}\,(\text{AxSK}) \quad \dfrac{}{\Gamma_0' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\,(\text{AxK})}{\Gamma_0' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7\}}\,(\text{RApp})}{\Gamma_0' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4\}}\,(\text{RApp})}{\Gamma_0' \vdash \texttt{Cons}\ 1\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,(\text{SKRek})$$
$$\Gamma_0 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = \alpha_1 \to [\texttt{Int}]$$

wobei $\sigma = \{\alpha_2 \mapsto [\texttt{Int}], \alpha_3 \mapsto [\texttt{Int}] \to [\texttt{Int}], \alpha_4 \mapsto [\texttt{Int}], \alpha_5 \mapsto \texttt{Int}, \alpha_6 \mapsto \alpha_7 \to [\texttt{Int}], \alpha_8 \mapsto \texttt{Char} \to \alpha_7\}$ *die Lösung von* $\{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$ *ist.*

D.h. $\Gamma_1 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha \to [\texttt{Int}]\}$. *Die nächste Iteration (wir lassen sie an dieser Stelle weg) zeigt, dass* Γ_1 *konsistent ist. Beachte: Für die Funktion* g *kann Haskell keinen Typ herleiten:*

```
Prelude> let g x = 1:(g(g 'c'))

<interactive>:1:13:
Couldn't match expected type `[t]' against inferred type `Char'
      Expected type: Char -> [t]
      Inferred type: Char -> Char
    In the second argument of `(:)', namely `(g (g 'c'))'
    In the expression: 1 : (g (g 'c'))
```

Aber: Haskell kann den Typ verifizieren, wenn man ihn angibt:

```
let g::a -> [Int]; g x = 1:(g(g 'c'))
Prelude> :t g
g :: a -> [Int]
```

Das liegt daran, dass Haskell dann keine Typinferenz mehr durchführt, sondern nur den Rumpf wie einen Ausdruck mit bekanntem Typ für den Superkombinator g *behandelt.*

6.5.2.2. Das iterative Verfahren benötigt mehrere Iterationen

Beispiel 6.5.4. *Wir typisieren den Superkombinator* g *mit der Definition*

```
g x = x : (g (g 'c'))
```

Die Anfangsannahme ist $\Gamma = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$. *Der iterative Typisierungsalgorithmus startet mit* $\Gamma_0 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha\}$ *und typisiert* g *mit der* (SKRek)*-Regel (sei* $\Gamma_0' = \Gamma_0 \cup \{x :: \alpha_1\}$*):*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_0' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,(\text{AxK}) \quad \dfrac{}{\Gamma_0' \vdash x :: \alpha_1, \emptyset}\,(\text{AxV})}{\Gamma_0' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}\,(\text{RApp}) \quad , \quad \dfrac{\dfrac{}{\Gamma_0' \vdash \texttt{g} :: \alpha_6, \emptyset}\,(\text{AxSK}) \quad \dfrac{\dfrac{}{\Gamma_0' \vdash \texttt{g} :: \alpha_8, \emptyset}\,(\text{AxSK}) \quad \dfrac{}{\Gamma_0' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\,(\text{AxK})}{\Gamma_0' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7\}}\,(\text{RApp})}{\Gamma_0' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4\}}\,(\text{RApp})}{\Gamma_0' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,(\text{SKRek})$$
$$\Gamma_0 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = \alpha_5 \to [\alpha_5]$$

wobei $\sigma = \{\alpha_1 \mapsto \alpha_5, \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto [\alpha_5], \alpha_6 \mapsto \alpha_7 \to [\alpha_5], \alpha_8 \mapsto \texttt{Char} \to \alpha_7\}$ *die Lösung von* $\{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$ *ist.*

D.h. $\Gamma_1 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha \to [\alpha]\}$. *Da* $\Gamma_0 \neq \Gamma_1$ *muss eine weitere Iteration durchgeführt werden: Sei* $\Gamma_1' = \Gamma_1 \cup \{x :: \alpha_1\}$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,(\text{AxK}) \quad \dfrac{}{\Gamma_1' \vdash x :: \alpha_1, \emptyset}\,(\text{AxV})}{\Gamma_1' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}\,(\text{RApp}) \quad \dfrac{\dfrac{}{\Gamma_1' \vdash \texttt{g} :: \alpha_6 \to [\alpha_6], \emptyset}\,(\text{AxSK}) \quad \dfrac{\dfrac{}{\Gamma_1' \vdash \texttt{g} :: \alpha_8 \to [\alpha_8], \emptyset}\,(\text{AxSK}) \quad \dfrac{}{\Gamma_1' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\,(\text{AxK})}{\Gamma_1' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7\}}\,(\text{RApp})}{\Gamma_1' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4\}}\,(\text{RApp})}{\Gamma_1' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,(\text{SKRek})$$

$$\Gamma_1 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = [\texttt{Char}] \to [[\texttt{Char}]]$$

wobei $\sigma = \{\alpha_1 \mapsto [\texttt{Char}], \alpha_2 \mapsto [[\texttt{Char}]], \alpha_3 \mapsto [[\texttt{Char}]] \to [[\texttt{Char}]], \alpha_4 \mapsto [[\texttt{Char}]], \alpha_5 \mapsto [\texttt{Char}], \alpha_6 \mapsto [\texttt{Char}], \alpha_7 \mapsto [\texttt{Char}], \alpha_8 \mapsto \texttt{Char}\}$

die Lösung von $\{\alpha_8 \to [\alpha_8] \doteq \texttt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$ *ist.*

Daher ist $\Gamma_2 = \Gamma \cup \{\texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]]\}$. *Da* $\Gamma_1 \neq \Gamma_2$ *muss eine weitere Iteration durchgeführt werden: Sei* $\Gamma_2' = \Gamma_2 \cup \{x :: \alpha_1\}$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_2' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,(\text{AxK}) \quad \dfrac{}{\Gamma_2' \vdash x :: \alpha_1, \emptyset}\,(\text{AxV})}{\Gamma_2' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}\,(\text{RApp}) \quad \dfrac{\dfrac{}{\Gamma_2' \vdash \texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]], \emptyset}\,(\text{AxSK}) \quad \dfrac{\dfrac{}{\Gamma_2' \vdash \texttt{g} :: [\texttt{Char}] \to [[\texttt{Char}]], \emptyset}\,(\text{AxSK}) \quad \dfrac{}{\Gamma_2' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\,(\text{AxK})}{\Gamma_2' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_7, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7\}}\,(\text{RApp})}{\Gamma_2' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4\}}\,(\text{RApp})}{\Gamma_2' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,(\text{SKRek})$$

$$\Gamma_2 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2)$$

wobei σ *die Lösung von*

$\{[\texttt{Char}] \to [[\texttt{Char}]] \doteq \texttt{Char} \to \alpha_7, [\texttt{Char}] \to [[\texttt{Char}]] \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$ *ist.*

Aber die Unifikation des letzten Gleichungssystem schlägt fehl. Daher ist `g` *nicht typisierbar.*

Satz 6.5.5. *Das iterative Typisierungsverfahren benötigt unter Umständen mehrere Iterationen, bis ein Ergebnis (untypisiert / konsistente Annahme) gefunden wurde.*

Beweis. Beispiel 6.5.4 zeigt, dass mehrere Iterationen benötigt werden, um Ungetyptheit festzustellen. Es gibt auch Beispiele, die zeigen, dass mehrere Iterationen nötig sind, um eine konsistente Annahme zu finden (siehe Übungsaufgabe 6.5.6). □

Übungsaufgabe 6.5.6. *Typisiere den Superkombinator* `fix` *mit dem iterativen Typisierungsverfahren, Die Definition von* `fix` *ist:*

```
fix f = f (fix f)
```

Zeige, dass der iterative Typisierungsalgorithmus mehrere Iterationen benötigt, bis eine konsistente Annahme gefunden wird.

6.5.2.3. Das iterative Verfahren muss nicht terminieren

Beispiel 6.5.7. *Seien* `f` *und* `g` *Superkombinatoren mit den Definitionen:*

```
f = [g]
g = [f]
```

Es gilt $\texttt{f} \simeq \texttt{g}$*, d.h. das iterative Verfahren typisiert die beiden Superkombinatoren gemeinsam. Die Anfangsannahme ist* $\Gamma = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} : \forall a.a\}$*. Die iterative Typisierung startet mit* $\Gamma_0 = \Gamma \cup \{\texttt{f} :: \forall\alpha.\alpha, \texttt{g} :: \forall\alpha.\alpha\}$ *und wendet die* (SKRek)*-Regel für* `f` *und* `g` *an:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_0 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxK}) \quad \dfrac{}{\Gamma_0 \vdash \mathtt{g} :: \alpha_5}\,(\text{AxSK})}{\Gamma_0 \vdash (\texttt{Cons}\ \mathtt{g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3\}}\,(\text{RApp}) \quad \dfrac{}{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxK})}{\Gamma_0 \vdash [\mathtt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_0 \vdash_T \mathtt{f} :: \sigma(\alpha_1) = [\alpha_5]}\,(\text{SKRek})$$

$\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto \alpha_5\}$ *ist Lösung von* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_0 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxK}) \quad \dfrac{}{\Gamma_0 \vdash \mathtt{f} :: \alpha_5}\,(\text{AxSK})}{\Gamma_0 \vdash (\texttt{Cons}\ \mathtt{f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3\}}\,(\text{RApp}) \quad \dfrac{}{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxK})}{\Gamma_0 \vdash [\mathtt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_0 \vdash_T \mathtt{g} :: \sigma(\alpha_1) = [\alpha_5]}\,(\text{SKRek})$$

$\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto \alpha_5\}$ *ist Lösung von* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

Daher ist $\Gamma_1 = \Gamma \cup \{\mathtt{f} :: \forall a.[a], \mathtt{g} :: \forall a.[a]\}$. *Da* $\Gamma_1 \neq \Gamma_0$ *muss man weiter iterieren.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxK}) \quad \dfrac{}{\Gamma_1 \vdash \mathtt{g} :: [\alpha_5]}\,(\text{AxSK})}{\Gamma_1 \vdash (\texttt{Cons}\ \mathtt{g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3\}}\,(\text{RApp}) \quad \dfrac{}{\Gamma_1 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxK})}{\Gamma_1 \vdash [\mathtt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_1 \vdash_T \mathtt{f} :: \sigma(\alpha_1) = [[\alpha_5]]}\,(\text{SKRek})$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \to [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$ *ist Lösung von* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxK}) \quad \dfrac{}{\Gamma_1 \vdash \mathtt{f} :: [\alpha_5]}\,(\text{AxSK})}{\Gamma_1 \vdash (\texttt{Cons}\ \mathtt{f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3\}}\,(\text{RApp}) \quad \dfrac{}{\Gamma_1 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxK})}{\Gamma_1 \vdash [\mathtt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_1 \vdash_T \mathtt{g} :: \sigma(\alpha_1) = [[\alpha_5]]}\,(\text{SKRek})$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \to [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$ *ist Lösung von* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

Daher ist $\Gamma_2 = \Gamma \cup \{\mathtt{f} :: \forall a.[[a]], \mathtt{g} :: \forall a.[[a]]\}$. *Da* $\Gamma_2 \neq \Gamma_1$ *muss man weiter iterieren. Man kann vermuten, dass dieses Verfahren nicht endet. Wir beweisen es: Sei* $[a]^i$ *die i-fache Listenverschachtelung. Durch Induktion zeigen wir, dass* $\Gamma_i = \Gamma \cup \{\mathtt{f} :: \forall a.[a]^i, \mathtt{g} :: \forall a.[a]^i\}$. *Die Basis haben wir bereits gezeigt, der Induktionsschritt ist:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_i \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxK}) \quad \dfrac{}{\Gamma_i \vdash \mathtt{g} :: [\alpha_5]^i}\,(\text{AxSK})}{\Gamma_i \vdash (\texttt{Cons}\ \mathtt{g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3\}}\,(\text{RApp}) \quad \dfrac{}{\Gamma_i \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxK})}{\Gamma_i \vdash [\mathtt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_i \vdash_T \mathtt{f} :: \sigma(\alpha_1) = [[\alpha_5]^i]}\,(\text{SKRek})$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \to [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$ *ist Lösung von* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_i \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxK}) \quad \dfrac{}{\Gamma_i \vdash \texttt{f} :: [\alpha_5]^i}\,(\text{AxSK})}{\Gamma_i \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3\}}\,(\text{RApp}) \quad \dfrac{}{\Gamma_i \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxK})}{\Gamma_i \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_i \vdash_T \texttt{g} :: \sigma(\alpha_1) = [[\alpha_5]^i]}\,(\text{SKRek})$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \to [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$ *ist Lösung von* $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

D.h. $\Gamma_{i+1} = \Gamma \cup \{\texttt{f} :: \forall a.[a]^{i+1}, \texttt{g} :: \forall a.[a]^{i+1}\}$.

Satz 6.5.8. *Das iterative Typisierungsverfahren terminiert nicht immer.*

Beweis. Siehe Beispiel 6.5.7. □

Man kann zeigen, dass die iterative Typisierung im Allgemeinen unentscheidbar ist.

Theorem 6.5.9. *Die iterative Typisierung ist unentscheidbar.*

Beweis. Wir führen den Beweis nicht durch. Er folgt aus der Unentscheidbarkeit der so genannten Semi-Unifikation von First-Order Termen. Die entsprechenden Beweise sind in (Kfoury et al., 1990b; Kfoury et al., 1993; Henglein, 1993) zu finden. □

Abschließend zeigen wir, dass die so genannte „Type safety“ für das iterative Verfahren gilt. Diese besteht aus zwei Eigenschaften: Zum einen, dass die Typisierung unter der Reduktion erhalten bleibt (sogenannte „Type Preservation“) und dass getypte geschlossene Ausdrücke reduzibel sind, solange sie keine WHNF sind (das wird i.A. als „Progress Lemma“ bezeichnet).

Lemma 6.5.10. *Sei s ein direkt dynamisch ungetypter* KFPTS+seq*-Ausdruck. Dann kann das iterative Typsystem keinen Typ für s herleiten.*

Beweis. Wenn s direkt dynamisch ungetypt ist, trifft einer der folgenden Fälle zu:

- $s = R[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ \mathit{Alts}]$ und c ist nicht vom Typ T. Wenn die Typisierung den `case`-Ausdruck typisieren will, wird sie Gleichungen hinzufügen, so dass der Typ von $(c\ s_1\ \ldots\ s_n)$ gleich zum Typ der Pattern in den `case`-Alternativen gemacht wird. Da die Typen der Pattern aber zu Typ T gehören und c nicht, wird die Unifikation scheitern.
- $s = R[\texttt{case}_T\ \lambda x.t\ \texttt{of}\ \mathit{Alts}]$: Bei der Typisierung des `case`-Ausdrucks und von $\lambda x.t$ werden für $\lambda x.t$ Gleichungen erzeugt, die Implizieren, dass $\lambda x.t$ einen Funktionstyp erhält. Dieser Typ wird mit den Typen der Pattern in den `case`-Alternativen unifiziert, was scheitern muss.
- $R[(c\ s_1\ \ldots\ s_{\text{ar}(c)})\ t]$: Die Typisierung typisiert die Anwendung $((c\ s_1\ \ldots\ s_{\text{ar}(c)})\ t)$ wie verschachtelte Anwendung $(((c\ s_1)\ \ldots)\ s_{\text{ar}(c)})\ t)$. Hierbei werden Gleichungen hinzugefügt, die implizieren, dass c höchstens $\text{ar}(c)$ Argumente verarbeiten kann. Da die Anwendung ein weiteres Argument hat, wird die Unifikation scheitern.

□

Lemma 6.5.11 (Type Preservation)**.** *Sei s ein wohl-getypter, geschlossener* KFPTS+seq*-Ausdruck (wobei auch die verwendeten Superkombinatoren wohlgetypt sind) und $s \xrightarrow{name} s'$. Dann ist s' wohl-getypt.*

Beweis. Hierfür muss man die einzelnen Fälle einer (β)-, ($SK-\beta$)- und (case)-Reduktion durchgehen. Für die Typherleitung von s kann man aus der Typherleitung einen Typ für jeden Unterterm von s ablesen. Bei der Reduktion werden diese Typen einfach mitkopiert. Man kann dann leicht nachvollziehen, dass eine Typherleitung immer noch möglich ist. □

Genauer: Für einen Grundtyp τ: $t : \tau$ vor der Reduktion $t \to t'$, dann auch $t'\tau$ danach.
D.h. polymorphe Typen können nach Reduktion auch allgemeiner werden.

Aus den letzten beiden Lemmas folgt:

Satz 6.5.12. *Sei s ein wohlgetypter, geschlossener* KFPTSP+seq*-Ausdruck. Dann ist s nicht dynamisch ungetypt.*

Lemma 6.5.13 (Progress Lemma)**.** *Sei s ein wohl-getypter, geschlossener* KFPTS+seq*-Ausdruck. Dann gilt:*

- *s ist eine WHNF, oder*
- *s ist call-by-name-reduzibel, d.h. $s \xrightarrow{name} s'$ für ein s'.*

Beweis. Betrachtet man die Fälle, wann ein geschlossener KFPTS+seq-Ausdruck irreduzibel ist, so erhält man: s ist eine WHNF oder s ist direkt-dynamisch ungetypt. Daher folgt das Lemma. □

Theorem 6.5.14. *Die iterative Typisierung für* KFPTSP+seq *erfüllt die „Type-safety“-Eigenschaft.*

6.5.2.4. Erzwingen der Terminierung: Milner Schritt

Will man die Terminierung des iterativen Typisierungsverfahren nach einigen Schritten erzwingen, so kann man einen so genannten Milner-Schritt durchführen. Beachte: Im nächsten Abschnitt betrachten wir das Milner-Typisierungsverfahren. Der Unterschied zwischen dem iterativen Typisierungsverfahren mit Milner-Schritt und dem Milner-Typisierungsverfahren liegt darin, dass man beim iterativen Verfahren den Milner-Schritt erst nach einigen Iterationen (wenn man nicht mehr weitermachen will) durchführt, das Milner-Verfahren jedoch sofort einen Milner-Schritt durchführt (und deswegen gar nicht iteriert).

Definition 6.5.15 (Milner-Schritt)**.** *Sei $SK_1, \ldots, SK_m$ eine Gruppe verschränkt rekursiver Superkombinatoren und $\Gamma_i \vdash_T SK_1 :: \tau_1, \ldots, \Gamma_i \vdash_T SK_m :: \tau_m$ seien die durch die i. Iteration hergeleiteten Typen für $SK_1, \ldots, SK_m$.*

Führe einen Milner-Schritt wie folgt durch:

Typisiere alle Superkombinatoren $SK_1, \ldots, SK_m$ auf einmal, wobei die Typannahme ist $\Gamma_M = \Gamma \cup \{SK_1 :: \tau_1, \ldots, SK_m :: \tau_m\}$ (wobei Γ die Annahmen für Konstruktoren und bereits typisierte Superkombinatoren enthält). Verwende dafür die folgende Regel:

$$\text{(SKRekM)}\ \frac{\begin{array}{c}\text{für } i=1,\ldots,m:\\ \Gamma_M \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i', E_i\end{array}}{\Gamma_M \vdash_T \text{für } i=1,\ldots,m\ SK_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i')}$$

$$\text{wenn } \sigma \text{ Lösung von } E_1 \cup \ldots \cup E_m \cup \bigcup_{i=1}^{m} \{\tau_i \doteq \alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i'\}$$

$$\begin{array}{rcl} \text{und}\ \ SK_1\ x_{1,1}\ \ldots\ x_{1,n_1} & = & s_1 \\ \ldots & & \\ SK_m\ x_{m,1}\ \ldots\ x_{m,n_m} & = & s_m \end{array}$$

$$\text{die Definitionen von } SK_1, \ldots, SK_m \text{ sind}$$

Als zusätzliche Regel muss im Typisierungsverfahren hinzugefügt werden:

$$\text{(AxSK2)}\ \frac{}{\Gamma \cup \{SK :: \tau\} \vdash SK :: \tau}$$

$$\text{wenn } \tau \text{ nicht allquantifiziert ist}$$

Wir erläutern den Milner-Schritt: In der Annahme Γ_M werden die zu typisierenden Superkombinatoren *nicht* mit allquantifiziertem Typ verwendet, sondern ohne Allquantor. Die Regel (AxSK2) wird dann benutzt, um einen solchen Typ aus der Annahme herauszuholen. Dabei findet *keine* Umbenennung statt. D.h. es wird nur ein fester Typ für jedes Auftreten eines Superkombinators verwendet und keine Kopien gemacht im Gegensatz zum iterativen Verfahren. Außerdem werden durch (SKRekM)-Regel alle Gleichungssysteme vereint und gemeinsam unifiziert und es werden *neue Gleichungen* hinzugefügt: Für jeden Superkombinator werden die angenommenen Typen mit den hergeleiteten Typen unifiziert.

Daraus folgt: Die neue Annahme, die man durch die (SKRekM)-Regel herleiten kann ist *stets konsistent*, d.h. man braucht keine weitere Iteration durch zu führen.

Beachte, dass im Unterschied zu (SKRek) nun alle Superkombinatoren einer verschränkt rekursiven Gruppe in der Regel (SKRekM) gleichzeitig erfasst werden müssen, um die Unifikation über alle Gleichungssysteme zu ermöglichen.

Der Nachteil bei diesem Verfahren ist zum Einen, dass man nicht weiß, wann man den Milner-Schritt durchführen sollte und zum Anderen, dass eingeschränktere Typen hergeleitet werden können, als beim iterativen Verfahren (das sehen wir später). Es kann auch durchaus passieren, dass man durch weitere Iteration einen Typ herleiten hätte können, der Milner-Schritt jedoch einen Fehler bei der Unifikation fest stellt und den Superkombinator als ungetypt zurückweist.

6.5.3. Das Milner-Typisierungsverfahren

Das Typisierungsverfahren von Milner geht eigentlich auf drei Autoren zurück: Robin Milner, Luis Damas und Roger Hindley. Je nach Laune findet man daher auch die Bezeichnungen Hindley-Milner oder Damas-Milner Typisierung usw. Wir verwenden „Milner-Typisierung“. Das Verfahren lässt sich einfach beschreiben: Anstelle des iterativen Typisierungsverfahrens mit Milner-Schritt zur Terminierung, wird sofort ein Milner-Schritt durchgeführt:

Algorithmus 6.5.16 (Milner-Typisierungsverfahren)**.** *Seien $SK_1, \ldots, SK_m$ alle Superkombinatoren einer Äquivalenzklasse bzgl. $\simeq$ und seien die Superkombinatoren die kleiner bzgl. dem strikten Anteil der Ordnung $\preceq$ als SK_i sind bereits typisiert.*

1. *In die Typisierungsannahme* Γ *werden die allquantifizierten Typen der bereits typisierten Superkombinatoren und der Konstruktoren eingefügt.*
2. *Typisiere* $SK_1, \ldots, SK_m$ *mit der Regel* (MSKRek)*:*

$$\text{(MSKRek)}\ \frac{\begin{array}{c}\text{für } i = 1, \ldots, m\text{:} \\ \Gamma \cup \{SK_1 :: \beta_1, \ldots, SK_m :: \beta_m\} \\ \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i, E_i\end{array}}{\Gamma \vdash_T \text{für } i = 1, \ldots, m\ SK_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i)}$$

$$\text{wenn } \sigma \text{ Lösung von } E_1 \cup \ldots \cup E_m \cup \bigcup_{i=1}^{m} \{\beta_i \doteq \alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i\}$$

$$\text{und} \quad \begin{array}{lcl} SK_1\ x_{1,1}\ \ldots\ x_{1,n_1} & = & s_1 \\ \ldots & & \\ SK_m\ x_{m,1}\ \ldots\ x_{m,n_m} & = & s_m \end{array}$$

$$\text{die Definitionen von } SK_1, \ldots, SK_m \text{ sind}$$

Schlägt die Unifikation fehl, so sind $SK_1, \ldots, SK_m$ *nicht Milner-typisierbar, anderenfalls sind die Milner-Typen von* $SK_1, \ldots, SK_m$ *genau die Typen die man durch die Regel* (MSKRek) *erhalten hat. Beachte: Die Regel* (AxSK2) *wird bei der Typherleitung benötigt.*

Um die Regel zu Milner-Typisierung verständlicher zu machen, schreiben wir sie noch einmal für einen einzelnen Superkombinator auf (d.h. wir gehen davon aus, dass dieser Superkombinator rekursiv aber nicht verschränkt rekursiv mit anderen Superkombinatoren ist).

$$\text{(MSKRek1)}\ \frac{\Gamma \cup \{SK :: \beta, x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SK :: \sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)}$$

$$\text{wenn } \sigma \text{ Lösung von } E \cup \{\beta \doteq \alpha_1 \to \ldots \to \alpha_n \to \tau\}$$

$$\text{und } SK\ x_1\ \ldots\ x_n = s \text{ die Definition von } SK \text{ ist}$$

Wir beschreiben die Regel: Die Typ-Annahme für den Superkombinator ist β, also der allgemeinste Typ, aber dieser ist nicht (im Gegensatz zum iterativen Verfahren) allquantifiziert. Die Folge ist, dass bei der Herleitung alle Vorkommen von SK im Rumpf s mit dem gleichen Typ β (und nicht umbenannten Kopien) typisiert werden (im iterativen Verfahren wird die (AxSK)-Regel benutzt, während im Milner-Verfahren die (AxSK2)-Regel benutzt wird). Als weitere Veränderung (gegenüber dem iterativen Verfahren) wird bei der Unifikation die Gleichung $\beta \doteq \alpha_1 \to \ldots \to \alpha_n \to \tau$ hinzugefügt: Diese erzwingt, dass der angenommene Typ (β) gleich zum hergeleiteten Typ ist.

Für das Milner-Typisierungsverfahren gelten die folgenden Eigenschaften:

- Das Verfahren terminiert.
- Das Verfahren liefert eindeutige Typen (bis auf Umbenennung von Variablen)
- Die Milner-Typisierung ist entscheidbar.
- Das Problem, ob ein Ausdruck Milner-Typisierbar ist, ist DEXPTIME-vollständig (siehe (Mairson, 1990; Kfoury et al., 1990a)).
- Das Verfahren liefert u.U. eingeschränktere Typen als das iterative Verfahren. Insbesondere kann ein Ausdruck iterativ typisierbar, aber nicht Milner-typisierbar sein.

Aufgrund der Entscheidbarkeit wird das Milner-Typverfahren in Haskell und auch anderen funktionalen Programmiersprachen z.B. ML eingesetzt.

Beispiel 6.5.17. *Man benötigt manchmal exponentiell viele Typvariablen:*

```
(let x0 = \z->z in
  (let x1 = (x0,x0) in
    (let x2 = (x1,x1) in
      (let x3 = (x2,x2) in
         (let x4 = (x3,x3) in
            (let x5 = (x4,x4) in
               (let x6 = (x5,x5) in x6)))))
```

Die Anzahl der Typvariablen im Typ ist 2^6*. Verallgemeinert man das Beispiel, dann sind es* 2^n*.*

Wir betrachten einige Beispiele zur Milner-Typisierung:

Beispiel 6.5.18. *Wir typisieren den Superkombinator* `map` *mit dem Milner-Verfahren. Die Definition ist*

```
map f xs = case xs of {
             [] -> []
             (y:ys) -> (f y):(map f ys)
           }
```

Die Annahme für die Konstruktoren ist $\Gamma_0 = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} :: \forall a.[a]\}$.

Sei $\Gamma = \Gamma_0 \cup \{\texttt{map} :: \beta, f :: \alpha_1, xs :: \alpha_2\}$ *und* $\Gamma' = \Gamma' \cup \{y : \alpha_3, ys :: \alpha_4\}$.

Die Typisierung beginnend mit der (MSKRek)*-Regel ist:*

$$\text{(MSKRek1)}\ \frac{\begin{array}{ll}(a) & \Gamma' \vdash xs :: \tau_1, E_1\\ (b) & \Gamma' \vdash \texttt{Nil} :: \tau_2, E_2\\ (c) & \Gamma' \vdash (\texttt{Cons}\ y\ ys) :: \tau_3, E_3\\ (d) & \Gamma' \vdash \texttt{Nil} :: \tau_4, E_4\\ (e) & \Gamma' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \tau_5, E_5\end{array}}{\begin{array}{c}\text{(RCase)}\ \dfrac{}{\Gamma' \vdash \texttt{case}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; \texttt{Cons}\ y\ ys \to \texttt{Cons}\ y\ (\texttt{map}\ f\ ys)\} :: \alpha, E}\\ \Gamma \vdash_T \texttt{map} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha)\\ \textit{wenn } \sigma \textit{ Lösung von } E \cup \{\beta \doteq \alpha_1 \to \alpha_2 \to \alpha\}\end{array}}$$

wobei $E = E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha \doteq \tau_4, \alpha \doteq \tau_5\}$*. Die Typen* $\tau_1, \ldots, \tau_5$ *und die Gleichungen* $E_1, \ldots, E_5$ *ergeben sich beim Herleiten der Voraussetzungen* $(a), \ldots, (e)$*. Wir machen das zur Übersichtlichkeit getrennt:*

(a) $\dfrac{}{\Gamma' \vdash xs :: \alpha_2, \emptyset}\ (\text{AxV})$

D.h. $\tau_1 = \alpha_2$ *und* $E_1 = \emptyset$.

(b) $\dfrac{}{\Gamma' \vdash \texttt{Nil} :: [\alpha_5], \emptyset}\ (\text{AxK})$

D.h. $\tau_2 = [\alpha_5] und E_2 = \emptyset$

(c)
$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \texttt{Cons} :: \alpha_6 \to [\alpha_6] \to [\alpha_6], \emptyset}\ (\text{AxK}) \quad \dfrac{}{\Gamma' \vdash y :: \alpha_3, \emptyset}\ (\text{AxV})}{\Gamma' \vdash (\texttt{Cons}\ y) :: \alpha_7, \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7\}}\ (\text{RApp}) \quad \dfrac{}{\Gamma' \vdash ys :: \alpha_4, \emptyset}\ (\text{AxV})}{\Gamma' \vdash (\texttt{Cons}\ y\ ys) :: \alpha_8, \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8\}}\ (\text{RApp})$$

D.h. $\tau_3 = \alpha_8$ *und* $E_3 = \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8\}$

(d) $\dfrac{}{\Gamma' \vdash \texttt{Nil} :: [\alpha_9], \emptyset}\ (\text{AxK})$

D.h. $\tau_4 = [\alpha_9]$ *und* $E_4 = \emptyset$.

(e)

$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \texttt{Cons} :: \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}], \emptyset}\ (\text{AxK}) \quad \dfrac{\dfrac{}{\Gamma' \vdash f :: \alpha_1, \emptyset}\ (\text{AxV}) \quad \dfrac{}{\Gamma' \vdash y :: \alpha_3, \emptyset}\ (\text{AxV})}{\Gamma' \vdash (f\ y) :: \alpha_{15}, \{\alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}\ (\text{RApp})}{\Gamma' \vdash (\texttt{Cons}\ (f\ y)) :: \alpha_{11}, \{\alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}\ (\text{RApp}) \quad \dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \texttt{map} :: \beta, \emptyset}\ (\text{AxSK2}) \quad \dfrac{}{\Gamma' \vdash f :: \alpha_1, \emptyset}\ (\text{AxV})}{\Gamma' \vdash (\texttt{map}\ f) :: \alpha_{12}, \{\beta \doteq \alpha_1 \to \alpha_{12}\}}\ (\text{RApp}) \quad \dfrac{}{\Gamma' \vdash ys :: \alpha_4, \emptyset}\ (\text{AxV})}{\Gamma' \vdash (\texttt{map}\ f\ ys) :: \alpha_{13}, \{\beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}}\ (\text{RApp})}{\Gamma' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \alpha_{14}, \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}, \beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}}\ (\text{RApp})$$

D.h. $\tau_5 = \alpha_{14}$ *und*
$E_5 = \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15},$
$\beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}$

Insgesamt müssen wir das Gleichungssystem $E \cup \{\beta \doteq \alpha_1 \to \alpha_2 \to \alpha\}$ *durch Unifikation lösen, d.h.*

$$\begin{aligned}\{&\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8, \alpha_{11} \doteq \alpha_{13} \to \alpha_{14},\\ &\alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}, \beta \doteq \alpha_1 \to \alpha_{12},\\ &\alpha_{12} \doteq \alpha_4 \to \alpha_{13}, \alpha_2 \doteq [\alpha_5], \alpha_2 \doteq \alpha_8, \alpha \doteq \alpha_9, \alpha \doteq \alpha_{14}\}\end{aligned}$$

Die Unifikation ergibt

$$\begin{aligned}\sigma = \{&\alpha \mapsto [\alpha_{10}], \alpha_1 \mapsto \alpha_6 \to \alpha_{10}, \alpha_2 \mapsto [\alpha_6], \alpha_3 \mapsto \alpha_6, \alpha_4 \mapsto [\alpha_6], \alpha_5 \mapsto \alpha_6,\\ &\alpha_7 \mapsto [\alpha_6] \to [\alpha_6], \alpha_8 \mapsto [\alpha_6], \alpha_9 \mapsto [\alpha_{10}], \alpha_{11} \mapsto [\alpha_{10}] \to [\alpha_{10}],\\ &\alpha_{12} \mapsto [\alpha_6] \to [\alpha_{10}], \alpha_{13} \mapsto [\alpha_{10}], \alpha_{14} \mapsto [\alpha_{10}], \alpha_{15} \mapsto \alpha_{10},\\ &\beta \mapsto (\alpha_6 \to \alpha_{10}) \to [\alpha_6] \to [\alpha_{10}],\end{aligned}$$

D.h. $map :: \sigma(\alpha_1 \to \alpha_2 \to \alpha) = (\alpha_6 \to \alpha_{10}) \to [\alpha_6] \to [\alpha_{10}]$.

Beispiel 6.5.19. *Wir betrachten Beispiel 6.5.4 erneut und führen die Milner Typisierung dafür durch. Der Superkombinator* `g` *sei definiert als*

```
g x = x : (g (g 'c'))
```

Die Anfangsannahme ist $\Gamma = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$. *Sei* $\Gamma' = \Gamma \cup \{x :: \alpha, \texttt{g} :: \beta\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,{\scriptstyle(\text{AxK})}, \dfrac{}{\Gamma' \vdash x :: \alpha, \emptyset}\,{\scriptstyle(\text{AxV})}}{\Gamma' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3}\,{\scriptstyle(\text{RApp})}, \dfrac{\dfrac{}{\Gamma' \vdash \texttt{g} :: \beta, \emptyset}\,{\scriptstyle(\text{AxSK2})}}{\Gamma' \vdash (\texttt{g}\ \texttt{'c'}) :: \alpha_4, \{\beta \doteq \texttt{Char} \to \alpha_4\}}\,{\scriptstyle(\text{RApp})}}{\Gamma' \vdash \texttt{Cons}\ x\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,{\scriptstyle(\text{RApp})}}{\Gamma \vdash_T \texttt{g} :: \sigma(\alpha \to \alpha_2)}\,{\scriptstyle(\text{MSKRek})}$$

wobei σ die Lösung von $\{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2, \beta \doteq \alpha \to \alpha_2\}$ ist.

Die Unifikation schlägt jedoch fehl, da `Char` *mit einer Liste unifiziert werden soll. D.h.* `g` *ist nicht Milner-typisierbar.*

Beispiel 6.5.20. *Wir betrachten die Funktion* `g` *aus Beispiel 6.5.3 erneut:*

```
g x = 1 : (g (g 'c'))
```

Sei $\Gamma' = \Gamma \cup \{x :: \alpha, \texttt{g} :: \beta\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\,{\scriptstyle(\text{AxK})}, \dfrac{}{\Gamma' \vdash 1 :: \texttt{Int}, \emptyset}\,{\scriptstyle(\text{AxK})}}{\Gamma' \vdash (\texttt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3}\,{\scriptstyle(\text{RApp})}, \dfrac{\dfrac{}{\Gamma' \vdash \texttt{g} :: \beta, \emptyset}\,{\scriptstyle(\text{AxSK2})}}{\Gamma' \vdash (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_4, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4\}}\,{\scriptstyle(\text{RApp})}}{\Gamma' \vdash \texttt{Cons}\ 1\ (\texttt{g}\ (\texttt{g}\ \texttt{'c'})) :: \alpha_2, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}\,{\scriptstyle(\text{RApp})}}{\Gamma \vdash_T \texttt{g} :: \sigma(\alpha \to \alpha_2)}\,{\scriptstyle(\text{SKRek})}$$

wobei σ die Lösung von $\{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2, \beta \doteq \alpha \to \alpha_2\}$ ist.

Die Unifikation schlägt fehl, da $[\alpha_5] \doteq$ `Char` *unifiziert werden sollen. Beachte* `g` *ist iterativ typisierbar.*

Als abschließendes Beispiel betrachten wir eine Funktion, die sowohl iterativ als auch Milner-typisierbar ist, aber der iterative Typ ist echt allgemeiner als der Milner-Typ:

Beispiel 6.5.21. *Sei der Datentyp* `Baum` *definiert als*

```
data Baum a = Leer | Knoten a (Baum a) (Baum a)
```

Die Typen für die Konstruktoren sind `Leer` $:: \forall a.$`Baum` a *und* `Knoten` $:: \forall a.a \to$ `Baum a` $\to$ `Baum a` $\to$ `Baum a`

Wir typisieren die Funktion `g`*, die definiert ist als*

```
g x y = Knoten True (g x y) (g y x)
```

zunächst mit dem Milner-Verfahren: Beinhalte Γ die Annahmen für die Konstruktoren. Sei $\Gamma' = \Gamma \cup \{x :: \alpha_1, y :: \alpha_2, \texttt{g} :: \beta\}$. Wir zerlegen die Herleitungsbäume zur Übersichtlichkeit:

$$\text{(MSKRek)}\ \frac{\text{(RApp)}\ \dfrac{(c),(a)}{\begin{array}{c}\Gamma' \vdash (\texttt{Knoten True}\ (\texttt{g}\ x\ y)\ (\texttt{g}\ y\ x)) :: \alpha_3,\\ \{\alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \doteq \texttt{Bool} \to \alpha_9,\\ \beta \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10},\\ \beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3\}\end{array}}}{\begin{array}{c}\Gamma \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3)\\ \textit{wobei } \sigma \textit{ Lösung von}\\ \{\alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \doteq \texttt{Bool} \to \alpha_9,\\ \beta \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10},\\ \beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3 \beta = \alpha_1 \to \alpha_2 \to \alpha_3\}\end{array}}$$

(a)

$$\text{(RApp)}\ \frac{\text{(RApp)}\ \dfrac{\text{(AxSK2)}\ \dfrac{}{\Gamma' \vdash g :: \beta, \emptyset},\ \text{(AxV)}\ \dfrac{}{\Gamma' \vdash y :: \alpha_2, \emptyset}}{\Gamma' \vdash (\texttt{g}\ y) :: \alpha_4, \{\beta \doteq \alpha_2 \to \alpha_4\}},\ \text{(AxV)}\ \dfrac{}{\Gamma' \vdash x :: \alpha_1, \emptyset}}{(\texttt{g}\ y\ x)) :: \alpha_5, \{\beta \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5\}}$$

(b)

$$\text{(RApp)}\ \frac{\text{(RApp)}\ \dfrac{\text{(AxSK2)}\ \dfrac{}{\Gamma' \vdash g :: \beta, \emptyset},\ \text{(AxV)}\ \dfrac{}{\Gamma' \vdash x :: \alpha_1, \emptyset}}{\Gamma' \vdash (\texttt{g}\ x) :: \alpha_6, \{\beta \doteq \alpha_1 \to \alpha_6\}},\ \text{(AxV)}\ \dfrac{}{\Gamma' \vdash y :: \alpha_2, \emptyset}}{(\texttt{g}\ x\ y)) :: \alpha_7, \{\beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}}$$

(c)

$$\text{(RApp)}\ \frac{\text{(RApp)}\ \dfrac{\text{(AxK)}\ \dfrac{}{\Gamma' \vdash \texttt{Knoten} :: \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8, \emptyset},\ \text{(AxK)}\ \dfrac{}{\Gamma' \vdash \texttt{True} :: \texttt{Bool}, \emptyset}}{\Gamma' \vdash (\texttt{Knoten True}) :: \alpha_9, \quad \{\alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \doteq \texttt{Bool} \to \alpha_9\}},\ (b)}{\begin{array}{c}\Gamma' \vdash (\texttt{Knoten True}\ (\texttt{g}\ x\ y)) :: \alpha_{10},\\ \{\alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \to \texttt{Baum}\ \alpha_8 \doteq \texttt{Bool} \to \alpha_9,\\ \alpha_9 \doteq \alpha_7 \to \alpha_{10}, \beta \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}\end{array}}$$

Die Unifikation ergibt

$$\begin{array}{rcl}\sigma & = & \{\alpha_1 \mapsto \alpha_2, \alpha_3 \mapsto \texttt{Baum Bool}, \alpha_4 \mapsto \alpha_2 \to \texttt{Baum Bool},\\ & & \alpha_5 \mapsto \texttt{Baum Bool}, \alpha_6 \mapsto \alpha_2 \to \texttt{Baum Bool}, \alpha_8 \mapsto \texttt{Bool},\\ & & \alpha_9 \mapsto \texttt{Baum Bool} \to \texttt{Baum Bool} \to \texttt{Baum Bool}, \alpha_7 \mapsto \texttt{Baum Bool},\\ & & \alpha_{10} \mapsto \texttt{Baum Bool} \to \texttt{Baum Bool}\}\end{array}$$

D.h. $\Gamma \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3) = \alpha_2 \to \alpha_2 \to \texttt{Baum Bool}$.

Daher ergibt der Milner-Typcheck $\texttt{g} :: a \to a \to \texttt{Baum Bool}$.

Als nächstes betrachten wir die iterative Typisierung von `g`*: Sei* $\Gamma_0 = \Gamma \cup \{\mathtt{g} :: \forall\alpha.\alpha\}$*, die erste Iteration für* Γ_0 *ergibt, wobei* $\Gamma_0' = \Gamma_0 \cup \{x :: \alpha_1, y :: \alpha_2\}$*:*

$$\text{(SKRek)}\ \frac{\begin{array}{c}\text{(RApp)}\ \dfrac{(c),(a)}{\Gamma_0' \vdash (\mathtt{Knoten\ True}\ (\mathtt{g}\ x\ y)\ (\mathtt{g}\ y\ x)) :: \alpha_3,\\ \{\alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9,\\ \beta_1 \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10},\\ \beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3\}\end{array}}{\begin{array}{c}\Gamma_0 \vdash_T \mathtt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3)\\ \textit{wobei } \sigma \textit{ Lösung von}\\ \{\alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9,\\ \beta_1 \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5, \alpha_9 \doteq \alpha_7 \to \alpha_{10},\\ \beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7, \alpha_{10} \doteq \alpha_5 \to \alpha_3\}\end{array}}$$

(a)

$$\text{(RApp)}\ \frac{\text{(RApp)}\ \dfrac{\text{(AxSK)}\ \dfrac{}{\Gamma_0' \vdash g :: \beta_1, \emptyset},\ \text{(AxV)}\ \dfrac{}{\Gamma_0' \vdash y :: \alpha_2, \emptyset}}{\Gamma_0' \vdash (\mathtt{g}\ y) :: \alpha_4, \{\beta_1 \doteq \alpha_2 \to \alpha_4\}},\ \text{(AxV)}\ \dfrac{}{\Gamma_0' \vdash x :: \alpha_1, \emptyset}}{(\mathtt{g}\ y\ x)) :: \alpha_5, \{\beta_1 \doteq \alpha_2 \to \alpha_4, \alpha_4 \doteq \alpha_1 \to \alpha_5\}}$$

(b)

$$\text{(RApp)}\ \frac{\text{(RApp)}\ \dfrac{\text{(AxSK)}\ \dfrac{}{\Gamma_0' \vdash g :: \beta_2, \emptyset},\ \text{(AxV)}\ \dfrac{}{\Gamma_0' \vdash x :: \alpha_1, \emptyset}}{\Gamma_0' \vdash (\mathtt{g}\ x) :: \alpha_6, \{\beta_2 \doteq \alpha_1 \to \alpha_6\}},\ \text{(AxV)}\ \dfrac{}{\Gamma_0' \vdash y :: \alpha_2, \emptyset}}{(\mathtt{g}\ x\ y)) :: \alpha_7, \{\beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}}$$

(c)

$$\text{(RApp)}\ \frac{\begin{array}{c}\text{(RApp)}\ \dfrac{\text{(AxK)}\ \dfrac{}{\Gamma_0' \vdash \mathtt{Knoten} :: \alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8, \emptyset},\ \text{(AxK)}\ \dfrac{}{\Gamma_0' \vdash \mathtt{True} :: \mathtt{Bool}, \emptyset}}{\Gamma_0' \vdash (\mathtt{Knoten\ True}) :: \alpha_9, \quad \{\alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9\}},\\ (b)\end{array}}{\begin{array}{c}\Gamma_0' \vdash (\mathtt{Knoten\ True}\ (\mathtt{g}\ x\ y)) :: \alpha_{10},\\ \{\alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \to \mathtt{Baum}\ \alpha_8 \doteq \mathtt{Bool} \to \alpha_9,\\ \alpha_9 \doteq \alpha_7 \to \alpha_{10}, \beta_2 \doteq \alpha_1 \to \alpha_6, \alpha_6 \doteq \alpha_2 \to \alpha_7\}\end{array}}$$

Die Unifikation ergibt

$$\begin{array}{rcl}\sigma & = & \{\alpha_1 \mapsto \alpha_2, \alpha_3 \mapsto \mathtt{Baum\ Bool}, \alpha_4 \mapsto \alpha_2 \to \mathtt{Baum\ Bool},\\ & & \alpha_5 \mapsto \mathtt{Baum\ Bool}, \alpha_6 \mapsto \alpha_2 \to \mathtt{Baum\ Bool}, \alpha_8 \mapsto \mathtt{Bool},\\ & & \alpha_9 \mapsto \mathtt{Baum\ Bool} \to \mathtt{Baum\ Bool} \to \mathtt{Baum\ Bool}, \alpha_7 \mapsto \mathtt{Baum\ Bool},\\ & & \alpha_{10} \mapsto \mathtt{Baum\ Bool} \to \mathtt{Baum\ Bool}\}\end{array}$$

Die Unifikation ergibt:

$$\begin{aligned}\sigma \quad = \quad & \{\beta_1 \mapsto \alpha_2 \to \alpha_1 \to \texttt{Baum Bool}, \beta_2 \mapsto \alpha_1 \to \alpha_2 \to \texttt{Baum Bool},\\ & \alpha_3 \mapsto \texttt{Baum Bool}, \alpha_4 \mapsto \alpha_1 \to \texttt{Baum Bool}, \alpha_5 \mapsto \texttt{Baum Bool},\\ & \alpha_6 \mapsto \alpha_2 \to \texttt{Baum Bool}, \alpha_7 \mapsto \texttt{Baum Bool}, \alpha_8 \mapsto \texttt{Bool},\\ & \alpha_9 \mapsto \texttt{Baum Bool} \to \texttt{Baum Bool} \to \texttt{Baum Bool}, \alpha_{10} \mapsto \texttt{Baum Bool} \to \texttt{Baum Bool}\}\end{aligned}$$

Das ergibt $\texttt{g} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha_3) = \alpha_1 \to \alpha_2 \to \texttt{Baum Bool}$ *Daher ist* $\Gamma_1 = \Gamma \cup \{\texttt{g} :: \forall \alpha, \beta, \alpha \to \beta \to \texttt{Baum Bool}\}$. *Eine weitere Iteration zeigt, dass* Γ_1 *konsistent ist (wir lassen sie weg). Insgesamt ergibt das iterative Verfahren* $\texttt{g} :: a \to b \to \texttt{Baum Bool}$. *Der iterative Typ ist daher allgemeiner als der Milner-Typ. Im GHC wird der eingeschränktere Typ berechnet:*

```
*Main> let g x y =  Knoten True (g x y) (g y x)
*Main> :t g
g :: t -> t -> Baum Bool
```

Gibt man allerdings den iterativen Typ per Hand vor, dann kann Haskell den Typ verifizieren:

```
*Main> let g :: a -> b -> Baum Bool; g x y =  Knoten True (g x y) (g y x)
*Main> :t g
g :: a -> b -> Baum Bool
```

Das Milner-Verfahren erfüllt die Eigenschaft, dass Milner-getypte Programme niemals dynamisch ungetypt sind. Ebenso gilt das Progress Lemma, d.h. Milner-getypte (geschlossene) Programme sind entweder reduzibel oder WHNFs. Die Type-Preservation Eigenschaft gilt in KFPTSP+seq für die Milner-Typisierung auch. Verwendet man jedoch einen Kalkül mit rekursiven `let`-Ausdrücken, kann es passieren das Milner-getypte Ausdrücke durch Reduktion in Ausdrücke überführt werden, die nicht mehr Milner-getypt sind. Trotzdem sind diese Ausdrücke iterativ getypt (daher kann kein dynamischer Typfehler auftreten).
Das Gegenbeispiel ist:

```
let x = (let y = \u -> z in (y [], y True, seq x True)); z = const z x in x
```

ist Milner-typisierbar. Wenn man eine so genannte (*llet*)-Reduktion durchführt erhält man

```
let x = (y [], y True, seq x True); y = \u -> z; z = const z x in x
```

Dieser Ausdruck ist nicht mehr Milner-typisierbar, da `y` zusammen mit `x` typisiert wird.

6.6. Haskells Monomorphism Restriction

In Haskells Typsystem gibt es eine weitere Beschränkung, die jedoch nur im Zusammenhang mit Typklassen auftritt. Wir haben Typklassen im Milner- bzw. iterativen Typcheck bisher nicht betrachtet und machen dies auch nicht. Die „Monomorphism Restriction“ in Haskell führt dazu, dass manchmal Superkombinatoren einen spezielleren Typ erhalten, als man denkt: Eine Funktion könnte den Typ `Class a => ... a ...` erhalten, aber aufgrund der Beschränkung erhält sie einen Instanztyp. Die Instanz ist dabei der default-Typ. Man kann die Monomorphism Restriction umgehen, indem man den Typ explizit angibt. Aus dem Haskell-Wiki sind die folgenden Beispiele:

```
-- This is allowed
f1 x = show x

-- This is not allowed
f2 = \x -> show x

-- ...but this is allowed
f3 :: (Show a) => a -> String
f3 = \x -> show x

-- This is not allowed
f4 = show

-- ...but this is allowed
f5 :: (Show a) => a -> String
f5 = show
```

Hierbei bedeutet "not allowed", dass der GHC einen eingeschränkten Typ herleitet. Z.B. wird `f2` dann typisiert als

```
Prelude> let f2 = \x -> show x
Prelude> :t f2
```

Wann die "Monomorphism Restriction" genau zutrifft, ist eine eher technische Definition die im Haskell Report nachgelesen werden kann. Man kann sich merken: Um die Beschränkung zu umgehen, ist es oft nötig einer Funktionsdefinition mehr Argumente zu geben (vergleiche `f4` und `f1`) oder explizit einen Typ anzugeben.

Die Monomorphism Restriction ist aktuell im GHCi als Standardeinstellung ausgeschaltet, aber im GHC eingeschaltet.

6.7. Zusammenfassung und Quellennachweis

Wir haben in diesem Kapitel die polymorphe Typisierung für Haskell und für KFPTSP+seq-Ausdrücke erörtert. Typklassen wurden dabei nicht behandelt. Wir haben zwei Verfahren zur polymorphen Typisierung rekursiver Superkombinatoren kennen gelernt und analysiert. Der Inhalt dieses Kapitel stammt im Wesentlichen aus dem Skript von (Schmidt-Schauß, 2009).

Das Milner-Typisierungsverfahren wurde in (Milner, 1978) und in ähnlicher Form auch schon in (Hindley, 1969) eingeführt. In (Damas & Milner, 1982) wurde die Korrektheit und Vollständigkeit des Verfahrens nachgewiesen. Das iterative Typisierungsverfahren entspricht im Wesentlichen dem Verfahren in (Mycroft, 1984).

7 Template Haskell

7.1. 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.

7.2. 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 7.3.

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`.

7.3. 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.

7.4. 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 (HaskellWiki, 2020) 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 quasiquoter."
```

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.

7.5. Zusammenfassung und Quellen

Wir haben die wesentlichen Konstrukte von Template Haskell betrachtet. Dieses Kapitel orientiert sich im Wesentlichen an (HaskellWiki, 2020), wobei teilweise die Syntax an die aktuelle GHC-Version angepasst wurden und Beispiele eingefügt wurden. Auch das Material in (Jost, 2019) wurde verwendet, um die Darstellung anzureichern.

8

Funktionale Referenzen (Linsen)

8.1. Einleitung

Bei der Programmierung entsteht öfters das Problem, einen kleinen Teil einer großen Datenstruktur abzuändern.

Wenn wir keine echten Referenzen (d.h. veränderbare, benannte Speicherplätze) innerhalb einer Monade, wie z.B. `STRef`, verwenden wollen, bleibt in einer rein funktionalen Sprache nur das Kopieren der gesamten Datenstruktur von der Wurzel bis zum geänderten Element. Dabei können unveränderte Teile übernommen werden, da Aliasing in der funktionalen Welt unproblematisch ist.

Im jetzigen Abschnitt geht es jedoch weniger um die Effizienz einer solchen Änderung, sondern viel mehr darum, wie man solche Änderungen einfach und elegant ausdrücken und programmieren kann:

Der hier verfolgte Ansatz ist der einer *funktionalen Referenz* auf das zu ändernde Element. Wir betrachten hierfür das Konzept der Linsen. Diese Konzept wird nicht nur in Haskell verwendet, sondern auch für andere Programmiersprachen und Paradigmen, die abstrakte Datentypen verwenden. Z.B. gibt es auch Linsen-Bibliotheken für Javascript.

Wir betrachten als Beispiel die folgenden Datentypen, die als Record-Typen modelliert sind:

```
data Wealth  = Wealth { gold:: Int, diamonds :: Int }
instance Show Wealth where
  show (Wealth g d) = "("++show g++"g,"++show d++"d)"
data Person  = Person { name :: String, wealth :: Wealth }
instance Show Person where
 show p = name p ++ show (wealth p)
data Faction = Faction{ faction :: String, members :: [Person]}
instance Show Faction where
 show p = faction p ++ show (members p)
```

Eine Person, hat daher einen Namen und einen Wohlstand, wobei Wohlstand sich aus einer Anzahl an Goldstücken und einer Anzahl an Diamanten zusammensetzt. Eine Fraktion hat einen Namen und eine Liste von Personen als Mitglieder.
Z.B. können wir konkret definieren:

```
cersei = Person "Cersei" $ Wealth 222 22
tyrion = Person "Tyrion" tw1
tw1    = Wealth 100 1
lannisters = Faction "Lannisters" [cersei,tyrion]
```

Nehmen wir an, Tyrion gibt 7 Goldstücke aus, dann benötigen wir eine Funktion der Form:

```
payGold  :: Int -> Person -> Person
payGold  m p = let w  = wealth p
                   w' = w { gold=(gold w) - m}
               in  p { wealth=w' }


> payGold  7 tyrion
Tyrion(93g,1d)
```

Eine alternative Variante ohne Verwendung der Record-Syntax ist:

```
payGold' :: Int -> Person -> Person
payGold' m (Person n (Wealth g d)) =
  let w = (Wealth (g-m) d)
  in Person n w


> payGold  7 tyrion
Tyrion(93g,1d)
```

Beide Varianten haben jedoch den Nachteil, dass die Datenstruktur händisch bis zu der Stelle ausgepackt wird, die verändert werden soll, um im Anschluss wieder zusammengepackt zu werden.

Mit funktionalen Referenzen (auch Linsen genannt) geht es bequemer:

```
*> over (wealth.gold) (subtract 7) tyrion
Tyrion(93g,1d)
```

Hier steuern wir mit `(wealth.gold)` die zu ändernde Position an und sagen mit `(substract 7)`, was wir ändern möchten.

Wenn wir mit `lannisters` starten, um unsere Änderung vorzunehmen, sieht der Code nicht viel anders aus:

```
*> over (members.(ix 1).wealth.gold) (subtract 7) lannisters
Lannisters[Cersei(222g,22d),Tyrion(93g,1d)]
```

Wenn alle Mitglieder `7` Stücke Gold zahlen sollen, können wir aufrufen:

```
*> over (members.traverse.wealth.gold) (subtract 7) lannisters
Lannisters[Cersei(215g,22d),Tyrion(93g,1d)]
```

Für Linsen gab und gibt es viele verschiedene Versuche und Ansätze diese zu implementieren. Ein Durchbruch gelang Edward Kmett 2012 mit dem Package `lens`[1].

Damit der oben gezeigte Code funktioniert, ist folgendes notwendig: Man muss das Modul `Control.Lens` importieren, man muss den Feldern einen Unterstrich voranstellen und kann dann die Linsen automatisch erzeugen lassen mit Template Haskell, indem man `makeLenses ''Wealth`, `makeLenses ''Person` und `makeLenses ''Faction` im Anschluss an die Datentypdefinition aufruft. Also insgesamt ergibt dies als Quellcode:

[1] `https://hackage.haskell.org/package/lens`

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens
data Wealth  = Wealth { _gold:: Int, _diamonds :: Int }
data Person  = Person { _name :: String, _wealth :: Wealth }
data Faction = Faction{ _faction :: String, _members :: [Person]}

makeLenses ''Wealth
makeLenses ''Person
makeLenses ''Faction

instance Show Wealth where
  show (Wealth g d) = "("++show g++"g,"++show d++"d)"
instance Show Person where
 show p = _name p ++ show (_wealth p)
instance Show Faction where
 show p = _faction p ++ show (_members p)
```

Dadurch werden für die drei Typen `Wealth`, `Person` und `Faction` viele verschiedene Linsen erstellt. Wir können diese mit mit `-ddump-splices` ansehen:

```
*> :set -ddump-splices
*> :l L1.hs
 L1.hs:7:1-19: Splicing declarations
    makeLenses ''Wealth
  ======>
    diamonds :: Lens' Wealth Int
    diamonds f_a84k (Wealth x1_a84l x2_a84m)
      = (fmap (\ y1_a84n -> (Wealth x1_a84l) y1_a84n)) (f_a84k x2_a84m)
    {-# INLINE diamonds #-}
    gold :: Lens' Wealth Int
    gold f_a84o (Wealth x1_a84p x2_a84q)
      = (fmap (\ y1_a84r -> (Wealth y1_a84r) x2_a84q)) (f_a84o x1_a84p)
    {-# INLINE gold #-}
L1.hs:8:1-19: Splicing declarations
    makeLenses ''Person
  ======>
    name :: Lens' Person String
    name f_a859 (Person x1_a85a x2_a85b)
      = (fmap (\ y1_a85c -> (Person y1_a85c) x2_a85b)) (f_a859 x1_a85a)
    {-# INLINE name #-}
    wealth :: Lens' Person Wealth
    wealth f_a85d (Person x1_a85e x2_a85f)
      = (fmap (\ y1_a85g -> (Person x1_a85e) y1_a85g)) (f_a85d x2_a85f)
    {-# INLINE wealth #-}
L1.hs:9:1-20: Splicing declarations
    makeLenses ''Faction
  ======>
    faction :: Lens' Faction String
    faction f_a85Y (Faction x1_a85Z x2_a860)
      = (fmap (\ y1_a861 -> (Faction y1_a861) x2_a860)) (f_a85Y x1_a85Z)
    {-# INLINE faction #-}
    members :: Lens' Faction [Person]
    members f_a862 (Faction x1_a863 x2_a864)
```

```
      = (fmap (\ y1_a865 -> (Faction x1_a863) y1_a865)) (f_a862 x2_a864)
    {-# INLINE members #-}
Ok, one module loaded.
```

Das Voranstellen des Unterstrichs ist notwendig für das automatische Erstellen der Linsen mit Template Haskell, denn andere Felder werden ignoriert (für sie wird keine Linse erstellt). Dies soll zunächst einen Eindruck über die Kernfunktionalität der Linsen vermitteln. Wir haben den theoretischen Hintergrund noch nicht erläutert und verstehen daher z.B. die automatisch erzeugten Linsen noch nicht.

8.1.1. Geschichte der Linsen

Die Idee der Linsen ist im Grunde, dass sie ein Paar von put- und get-Funktionen oder auch view- und update-Funktionen darstellen. Diese Idee stammt aus der Datenbankforschung Ende der 1970er Jahre (siehe z.B. (Bancilhon & Spyratos, 1981)). Der Begriff „Linse“ taucht in Arbeiten von Benjamin Pierce im Jahr 2005 auf (Bohannon et al., 2005; Bohannon et al., 2006). 2009 beschrieb Twan van Laarhoven in seinem Blog (van Laarhoven, 2009) einen Typ, welcher jetzt `Lens'` genannt wird.

Im Jahr 2012 erkannte Russell O'Connor (O'Connor, 2012) die Verbindung zwischen seinen Multi-Linsen und den van-Laarhoven-Linsen, was zu Traversals führte.

2012 formulierte Edward Kmett die benötigten Gesetze und schrieb die inzwischen populärste Linsen Bibliothek für Haskell (Kmett, 2020) (Es gibt viel andere Bibliotheken mit anderen Ansätzen).

8.2. Implementierung von Linsen

8.2.1. Grundsätzliche Idee von Linsen

Die Grundidee ist, dass eine Linse ein get/set-Paar (bzw. view/update-Paar), d.h. für unser Beispiel können wir schreiben

```
*> view wealth tyrion
(100g,1d)

*> set wealth (Wealth 0 0) tyrion
Tyrion(0g,0d)
```

Alternativ wird auch gerne eine Schreibweise genommen bei der das Funktionsargument vorne steht.

```
*> tyrion & view wealth
(100g,1d)

*> tyrion & set wealth (Wealth 0 0)
Tyrion(0g,0d)
```

Bei dem Operator `(&)` handelt es sich um `flip ($)`. Er wird im Modul `Data.Function` definiert.

```
(&) :: a -> (a -> b) -> b
x & f = f x
infixl 1 &
```

Prinzipiell kann man daher Linsen als Paare bestehend aus getter und setter modellieren. Z.B. könnte man eine Linsenimplementierung verwenden, die genau das macht:

```
data LensPair s a = LensPair { view :: s -> a
                             , set  :: a -> s -> s }
```

Beachte, dass die Typvariable `s` die große allgemeine Struktur repräsentiert und Typvariable `a` den Ausschnitt, der angeschaut oder verändert wird. Dementsprechend liefert `view` für die große Struktur den Ausschnitt und ist daher eine Funktion vom Typ `s -> a`, `set` ist vom Typ `a -> s -> s`, da sie einen neuen Wert für den Ausschnitt bekommt und dann damit eine Funktion wird, die die große Struktur verändert.
Für unser Beispiel könnten wir dann definieren:

```
lensPersonName   :: LensPair Person String
lensPersonName   = LensPair name   (\x s-> s{name=x})
lensPersonWealth :: LensPair Person Wealth
lensPersonWealth = LensPair wealth (\x s-> s{wealth=x})
```

Ein Aufruf dazu:

```
*> set lensPersonWealth (Wealth 9 1) tyrion
Tyrion(9g,1d)
```

Der `over`-Operator kann nun definiert werden als

```
over :: LensPair s a -> (a -> a) -> s -> s
over lens f s = set lens (f $ view lens s) s
```

Er erhält eine Linse und eine Funktion, die den Ausschnitt verändert. Entsprechend berechnet `over` den Ausschnitt mit `view lens s` , wendet die Funktion darauf an und setzt anschließend mit `set` den neuen Wert für den Ausschnitt ein.

Diese Modellierung wird jedoch nicht verwendet, da sie sich als unkomfortabel erweist, wenn man Linsen komponieren möchte, z.B. wenn man das Gold einer Person fokussieren möchte. Daher wird ein anderer Ansatz verwendet, der Rank-N-Typen verwendet.

8.2.2. Van Laarhoven-Linsen

Die von van Laarhoven vorgeschlagene Implementierung ist

```
{-# LANGUAGE RankNTypes #-}

type Lens' s a = forall f. Functor f => (a -> f a) -> (s -> f s)
```

Zur Erinnerung: Die Spracherweiterung `RankNTypes` erlaubt Typen, die ∀-Quantoren auch tiefer im Typ verwenden können (sie können im linken Argument des Typpfeils vorkommen). Ein Beispiel ist die Funktion

```
foo x = x x
```

Diese ist mit der Milner-Typisierung nicht typisierbar. Aber sie kann mit dem Rank-2-Typ `foo :: (forall a . a -> a) -> (b -> b)` typisiert werden (wenn dieser Typ vorgegeben wird!). Genauso kann sie aber auch mit dem Rank-2-Typ `foo':: (forall a . a) -> (b -> b)` typisiert werden (wir verwenden `foo` und `foo'`, um die beiden Typen zu unterscheiden). Insbesondere hat `foo` keinen allgemeinsten Rank-N-Typen. Beachte, dass `foo'` als erstes Argument ein Objekt erfordert, dass jeden Typ hat, daher kann `foo'` nur mit $\bot$-Ausdrücken (dies sind nichtterminierende Ausdrücke, wie z.B. `undefined`) aufgerufen werden. Für `foo` ist gefordert, dass das erste Argument eine Funktion ist, die für alle Typen `a` den Typ `a -> a` hat, d.h. z.B. dürfen wir `id` als erstes Argument verwenden, aber *nicht* `toUpper :: Char -> Char`, da der Typ von `toUpper` zu speziell ist.
Für die van Laarhoven-Linsen

```
type Lens' s a = forall f. Functor f => (a -> f a) -> (s -> f s)
```

gilt daher, dass sie Funktionen sind, die für alle Funktoren `f` funktionieren müssen.
Die nächste Frage ist, wo sich genau `view` und `set` in obiger Modellierung verstecken. Diese werden wie folgt definiert:

```
view :: Lens' s a -> s -> a
view l = getConst . l Const

set :: Lens' s a -> a -> s -> s
set l a = over l (const a)

over :: Lens' s a -> (a -> a) -> s -> s
over l m = runIdentity . l (Identity . m)
```

Die Typen von `view`, `set` und `over` passen zu den Typen, wie wir sie oben in der Paar-Implementierung erläutert haben. Bevor wir die Definitionen erklären, führen wir die Datentypen `Identity a` und `Const a b` ein. Die `Identity`-Monade kennen wir bereits, sie ist auch ein applikativer Funktor und kann daher verwendet werden. Sie macht im Grunde nichts und ist implementiert als:

```
newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
  fmap :: (a -> b) -> Identity a -> Identity b
  fmap f (Identity x) = Identity $ f x

instance Applicative Identity where
  pure :: a -> Identity a
  pure = Identity
  (<*>) :: Identity (a->b) -> Identity a -> Identity b
  Identity f <*> Identity x = Identity $ f x

instance Monad Identity where
  (>>=) :: Identity a -> (a -> Identity b) -> Identity b
  Identity x >>= mf = mf x
```

Der bind-Operator `>>=` entspricht dabei im Wesentlichen `flip ($)` – vom newtype-Konstruktor abgesehen – und `(<$>)` entspricht `($)` – vom newtype-Konstruktor abgesehen.
Der `Const`-Typ ist definiert als

```
newtype Const a b = Const {getConst :: a}
```

`Const a b` ist ein applikativer Funktor, den man sich wie einen Container über `b` vorstellen kann, welcher keinen Wert vom Typ `b` enthält, dafür als Kontext einen Wert des Typs `a` hat. Den Typ `b` bezeichnet man auch als *Phantom-Typen*, da Werte von `Const a b` nie `b`-Werte enthalten.
Der Typ `Const a b` ist keine Monade, aber nützlich als Instanz der Klassen `Foldable` und `Traversable`.

```
newtype Const a b = Const {getConst :: a}

instance Functor (Const m) where
    fmap :: (a -> b) -> Const m a -> Const m b
    fmap _ (Const x) = Const x

instance Monoid m => Applicative (Const m) where
    pure :: a -> Const m a
    pure = const $ Const mempty
    (<*>) :: Const m (a->b) -> Const m a -> Const m b
    Const x <*> Const y = Const $ x `mappend` y

instance Foldable (Const m) where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap _ _ = mempty

instance Traversable (Const m) where
    traverse :: Applicative f => (a->f b)-> t a-> f(t b)
    traverse _ (Const m) = pure $ Const m
```

Z.B. kann man dank `Const` aus jeder `Traversable`-Instanz ein `Foldable`-Instanz ableiten, denn damit kann man anhand von `traverse` die `foldMap`-Funktion definieren:

```
foldMapDefault::(Traversable t,Monoid c) => (a -> c) -> t a -> c
foldMapDefault f = getConst . traverse (Const . f)
```

Für die van Laarhoven-Linse

```
type Lens' s a = forall f. Functor f => (a -> f a) -> (s -> f s)
```

und die Implementierung von

```
view :: Lens' s a -> s -> a
view l = getConst . l Const
```

wird der Funktor `Const a` im Typ `Lens'` verwendet, d.h. `l` wird mit dem Typ

```
l :: (a -> Const a a) -> (s -> Const a s)
```

verwendet. Dementsprechend liefert `(l Const) :: s -> Const a s`. Wendet man dies auf eine Struktur `s` an, so erhält man `Const a s`. Die Struktur ist daher eigentlich schon gelöscht, denn ihr Typ existiert zwar noch, aber nur im Teil, der Phantomtyp ist. Als Wert steht hier `Const x` mit `x` vom Typ `a`. D.h. mit `getConst :: Const a b -> a` erhält man dieses `x` vom Typ `a`. Insgesamt zeigt dies (auf Typebene), dass `view l` eine Funktion von `s` nach `a` ist.
Für die Definition

```
over :: Lens' s a -> (a -> a) -> s -> s
over l m = runIdentity . l (Identity . m)
```

wird der Funktor `Identity` im Typ `Lens'` verwendet, d.h. `l` wird mit dem Typ `l :: (a -> Identity a) -> (s -> Identity s)` verwendet. Für eine Funktion `m :: a -> a`, die den Ausschnitt der Struktur `s` ändern soll, gilt `(Identity . m)::(a -> Identity a)`, sodass `(l (Identity . m)):: s -> Identity s` gilt. Schließlich kann nach Anwenden der Struktur mit `runIdentity` auf das Ergebnis vom Typ `s` zugegriffen werden.
Die Funktion

```
set :: Lens' s a -> a -> s -> s
set l a = over l (const a)
```

verwendet `over` und damit ebenfalls den Funktor `Identity` im Typ `Lens'`. Durch die Verwendung von `over` wird eine Funktion benötigt, die den Ausschnitt verändert. Hier wird die Funktion `const a` verwendet, die konstant den Wert `a` zurück liefert.

8.2.3. Definition eigener Linsen

Wir betrachten erneut unser Beispiel:

```
data Wealth  = Wealth { gold, diamonds :: Int }
data Person  = Person { name :: String, wealth :: Wealth }
```

Die van-Laarhoven-Linsen für unsere Datentypen sind:

```
-- forall f. Functor f => (Int-> f Int)-> Wealth-> f Wealth
lWgold :: Lens' Wealth Int
lWgold fun (Wealth g d) = (\g' -> Wealth g' d)  <$> (fun g)

lWdiamonds :: Lens' Wealth Int
lWdiamonds fun (Wealth g d) = (\d' -> Wealth g d') <$> (fun d)

-- forall f.Functor f => (String-> f String)-> Person-> f Person
lPname :: Lens' Person String
lPname  fun (Person n w) = (\n' -> Person n' w) <$> fun n

-- forall f.Functor f => (Wealth-> f Wealth)-> Person-> f Person
lPwealth :: Lens' Person Wealth
lPwealth  fun (Person n w) = (\w' -> Person n w') <$> fun w
```

In allen Fällen ist die Idee dieselbe: Die Funktion zum Aktualisieren des Ausschnitts wird auf das entsprechende Feld angewendet, anschließend wird die unveränderte Struktur herum aufgebaut, indem diese mit `fmap` aus dem Funktor „auf den geänderten Wert gemappt wird“. Man sieht, dass sämtliche Funktionen gleich aufgebaut sind, und sich daher algorithmisch erzeugen lassen (daher die Verwendung von Template Haskell hierfür). Das Paket `lens` stellt Template Haskell-Metafunktionen zur Verfügung, um diese Linsen automatisch zu erzeugen: Eine Linse für jedes Record-Feld, welches mit Unterstrich beginnt. Die Linse heißt wie das Feld ohne Unterstrich: Wir zeigen hierfür ein Beispiel:

```
data Wealth  = Wealth { _gold, _diamonds :: Int }
data Person  = Person { _name :: String, _wealth :: Wealth }
data Faction = Faction{ _faction :: String, _members :: [Person]}
makeLenses ''Wealth
$(makeLenses (mkName "Person"))
makeLenses ''Faction
```

Wir können uns nun die Typen anzeigen lassen:

```
> :type gold
gold :: Functor f => (Int -> f Int) -> Wealth -> f Wealth
> :type wealth.gold
wealth.gold :: Functor f => (Int -> f Int) -> Person -> f Person
```

Dies ist die Standard-Option, eigene Benennungen sind wählbar mit der Funktion `makeLensesWith :: LensRules -> Name -> DecsQ`.
Wir betrachten erneut die Implementierung von `view`, um sie zu verstehen:

```
Const    :: a -> Const a b
getConst :: Const a b -> a
view:: (forall f. Functor f => ((a -> f a) -> (s -> f s))) -> s -> a
view l = getConst . l Const
```

Für unser Beispiel betrachten wir dazu zusätzlich:

```
data Person  = Person { name :: String, wealth :: Wealth }
tyrion = Person "Tyrion" tw1    -- für ein tw1::Wealth

lPname :: Functor f => (String -> f String) -> Person -> f Person
lPname fun (Person n w) = (\n' -> Person n' w) <$> fun n
```

Wir werten `view lPname tyrion` per Hand aus, um die Definition von `view` nachzuvollziehen:

```
view lPname tyrion
= (getConst . lPname Const) tyrion
= getConst (lPname Const tyrion)
= getConst (lPname Const (Person "Tyrion" tw1))
= getConst (\n' -> Person n' tw1) <$> (Const "Tyrion")
= getConst (Const "Tyrion")
= "Tyrion"
```

Analog können wir die Implementierung von `over` an einem Beispiel nachvollziehen:

```
Identity    :: a -> Identity a
runIdentity :: Identity a -> a
over::(forall f.Functor f => ((a -> f a) -> (s -> f s))) -> (a -> a) -> s-> s
over l m = runIdentity . l (Identity . m)

data Person  = Person { name :: String, wealth :: Wealth }
tyrion = Person "Tyrion" tw1    -- für ein tw1::Wealth

lPname :: Functor f=> (String-> f String)-> Person-> f Person
lPname fun (Person n w) = (\n' -> Person n' w) <$> fun n
```

Wir berechnen `over lPname reverse tyrion` per Hand:

```
over lPname reverse tyrion
= (runIdentity . lPname (Identity . reverse)) tyrion
= runIdentity (lPname (Identity . reverse) tyrion)
= runIdentity (lPname (Identity . reverse) (Person "Tyrion" tw1)
= runIdentity (\n ' -> Person n' tw1) <$> ((Identity . reverse) "Tyrion")
= runIdentity (\n ' -> Person n' tw1) <$> ((Identity (reverse "Tyrion"))
= runIdentity (Identity (Person (reverse "Tyrion") tw1)
= Person (reverse "Tyrion") tw1
```

Schließlich betrachten wir noch ein Beispiel für `set`

```
set :: Lens' s a -> a -> s -> s
set l a = over l (const a)
```

Wir deuten die Berechnung von `set lPname "TYRION" (Person "Tyrion" tw1)` an:

```
set lPname "TYRION" (Person "Tyrion" tw1)
= over lPname (const "TYRION") (Person "Tyrion" tw1)
= ...
= runIdentity (Identity (Person (const "TYRION" "Tyrion") tw1))
= Person (const "TYRION" "Tyrion") tw1
```

Die Komposition von van-Laarhoven-Linsen entspricht umgedrehter Funktionskomposition:

```
type Lens' s a =
  forall f. Functor f => (a -> f a) -> (s -> f s)

compose :: Lens' b c -> Lens' a b -> Lens' a c
compose r s = s . r

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

Einsetzen des Typsynonyms ergibt:

```
compose :: forall a b c f. Functor f
        => ((c -> f c) -> (b -> f b))
        -> ((b -> f b) -> (a -> f a))
        -> ((c -> f c) -> (a -> f a))
```

Anstelle von `compose` wird jedoch meist die Funktionskomposition verwendet. D.h. man schreibt z.B.

```
*> tyrion & view (lPwealth.lWgold)
100
*> :type (lPwealth.lWgold)
  (lPwealth.lWgold) :: Functor f =>  (Int -> f Int) -> Person -> f Person
```

Dies wirkt wie eine verdrehte Reihenfolge, denn in der funktionalen Welt ist die äußere Funktion normalerweise auf der rechten Seite des Punktes, z.B. ergibt `negate.length $ [1..3]` als Ergebnis `-3`.
Die Komposition von Linsen erinnert dagegen an die Accessor-Verkettung in objektorientierten Sprachen wie etwa Java. Mit Hilfe des Infix-Synonym `(^.)` für `view` sieht das auch in Haskell so aus:

```
*> tyrion^.wealth.gold
  100
```

Eine andere Sichtweise (und damit teilweise Rechtfertigung der umgedrehten Komposition) ist, dass die Komposition von Linsen wie eine Komposition von fmap-Funktionen ist. Betrachte z.B.

```
fmapMaybe :: (a -> b) -> Maybe a -> Maybe b
fmapList  :: (a -> b) -> [a] -> [b]
fmapPair  :: (a -> b) -> (c,a) -> (c,b)
```

Dann ergibt `(fmapMaybe . fmapList . fmapPair)` ein `fmap` für den Typ `Maybe [(c,a)]`, d.h.

```
(fmapMaybe . fmapList . fmapPair) :: (a -> b) -> Maybe [(c, a)] -> Maybe [(c, b)]
```

und daher werden die Typen genau wie bei der Komposition von Linsen von außen nach innen durchlaufen.

8.2.4. Infix-Operatoren

Das `lens`-Paket definiert mehr als 100 Infix Operatoren, z.B.

```
view == (^.) :: s -> Lens' s a -> a
set  == (.~) :: Lens' s a -> a -> s -> s
over == (%~) :: Lens' s a -> (a -> a) -> s -> s
```

Die Benennung hält sich an folgende Konventionen: Operatoren, die

- `^` enthalten, sind Getter-ähnliche Operatoren
- `~` enthalten, sind Setter-ähnliche
- `.` enthalten, sind grundlegende Operatoren
- `%` enthalten, sind Operatoren mit Funktionen als Parameter
- `=` enthalten, sind Setter-Varianten für die `State`-Monade

Es gibt viele Operatoren für kleine Bequemlichkeiten, z.B.

```
(&&~) :: ASetter' s Bool -> Bool -> s -> s
l &&~ n = over l (&& n)
(<>~) :: Monoid a => ASetter' s a -> a -> s -> s
l <>~ m = over l (`mappend` m)
```

8.3. Polymorphe Linsen und weitere Optiken

8.3.1. Polymorphe Linsen

Bis jetzt haben wir folgende Typen:

```
type Lens'    s a = forall f. Functor f =>
                            (a -> f a)       -> s -> f s
```

Ein `Getter` verwendet für den Functor, den `Const`-Functor (wie bei `view`) und ein `Setter` verwendet (wie bei `over` und `set`), die `Identity` Monade. Daher ist im `lens`-Paket definiert:

```
type AGetter' s a = forall r.(a -> Const r  a)-> s -> Const r s
type ASetter' s a =          (a -> Identity b)-> s -> Identity s
```

Für polymorphe Datentypen wie z.B.

```
data Person nty wty = Person { name::nty, wealth::wty }
```

die polymorph über dem Typ für die Felder `name` und `wealth` sind, wird noch folgende Verallgemeinerung der Typen benötigt:

```
type Lens'   s   a   = Lens s s a a
type Lens    s t a b = forall f. Functor f =>
                       (a -> f        b) -> (s -> f        t)
type Setter  s t a b = (a -> Identity b) -> (s -> Identity t)
```

Die Idee dabei ist, dass nun ein setter, eine Funktion von $a \to b$ verwendet, um einen Ausschnitt vom Typ a einer großen Struktur vom Typ s durch einen Ausschnitt vom Typ b zu ersetzen, wobei sich der Typ der Struktur von s auf t ändert. Die Implementierung ändert sich dabei nicht, z.B. können wir auch für die polymorphe Version von `Person` definieren:

```
lPname :: Lens (Person n1 w) (Person n2 w) n1 n2
lPname  fun (Person n w) = (\n' -> Person n' w) <$> fun n
```

8.3.1.1. Linsen-Gesetze

Benjamin Pierce formulierte bereits die folgenden Gesetze für „very well behaved lenses“:

1. Man bekommt zurück, was man hineintut: `view l (set l v s) == v`
2. Das vorhandene neu zu setzen ändert nichts: `set l (view l s) s == s`
3. Nur das letzte Setzen zählt: `set l v2 (set l v1 s) == set l v2 s`

Edward Kmett betrachtete die Konsequenzen aus diesen Gesetzen und formulierte Varianten für weitere Optiken, welche wir jetzt noch betrachten werden.

8.3.2. Traversal

Wenn wir statt `Functor` im `Lens`-Type `Applicative` verwenden, dann erhalten wir eine *schwächere* `Traversal`-Optik:

```
type Traversal s t a b = forall f. Applicative f =>
                                   (a -> f b) -> (s -> f t)
type Lens      s t a b = forall f. Functor     f =>
                                   (a -> f b) -> (s -> f t)
```

Beachte, dass jede `Lens` auch ein `Traversal` ist und nicht umgekehrt! Dies mag auf dem ersten Blick merkwürdig erscheinen, wenn man den Typ aber umgangssprachlich vorliest, wird es klar:

`Traversal` ist eine Funktion, welche mit jedem beliebigen Typen `f` der Klasse `Applicative` umgehen können muss.

`Lens` ist eine Funktion, welche mit jedem beliebigen Typen `f` der Klasse `Functor` umgehen können muss.

Da `Applicative` Unterklasse von `Functor` ist, muss `Lens` mehr Typen beherrschen, und ist daher allgemeiner.

Während eine Linse immer genau ein Element fokussiert, hat eine `Traversal`-Optik möglicherweise mehrere Elemente im Focus.
Die Funktion `traverse` aus der Klasse `Traversable` hat bereits einen zu van-Laarhoven-Linsen kompatiblen Typ:

```
traverse :: (Traversable t, Applicative f) => (a -> f b) ->  t a -> f (t b)
```

Betrachte als Beispiel den bekannten Typ:

```
data Faction = Faction{ faction::String, members::[Person] }
```

und die folgenden Aufrufe:

```
> over' (lFmembers.traverse.lPwealth.lWgold) (*10) lannisters
Lannisters[Cersei(2220g,22d),Tyrion(1000g,1d)]
```

Wir nutzen hier aus, dass Listen `Traversable` sind. Dieses Beispiel nutzt selbst-definierte van-Laarhoven-Linsen. Hier ergibt sich ein Problem, dass wir `over` einen zu allgemeinen Typ gegeben hatten, daher verwenden wir `over' :: Traversal s s a a -> (a -> a) -> s -> s`. Das `lens`-Paket definiert zur Umgehung solcher Probleme zahlreiche Klassen und Typsynonyme. Mit den automatisch erzeugten Linsen können wir daher schreiben:

```
*Main> over (members.traverse.wealth.gold) (*10) lannisters
Lannisters[Cersei(2220g,22d),Tyrion(1000g,1d)]
```

8.3.3. Linsen-Hierarchie

Abbildung 8.1 zeigt die Linsen-Hierarchie. Dabei entspricht ein Pfeil →, der „ist-auch-ein“-Beziehung, wobei für Typen mit zwei statt vier Parametern stets gilt `s=t` und `a=b`. Da wird `foldMap` mit `traverse` implementieren können (wie vorher beschrieben mit `foldMapDefault`) folgt z.B. die Traversal→Fold-Beziehung. Ein genaueres Bild mit allen Funktionen ist auf den Seiten `lens`-Pakets zu finden (`https://github.com/ekmett/lens/`).

8.3.4. Prismen

Linsen fokussieren immer genau ein inneres Element. Prismen fokussieren dagegen ein oder kein Element.

```
type Prism s t a b= forall p f. (Choice p, Applicative f)=>
  p a (f b) -> p s (f t)
```

Linsen arbeiten mit Produkt-Typen wie Paaren oder Records; Prismen kümmern sich um Summen-Typen wie `Maybe` oder `Either`. Ein Prisma kann also Pattern-Matching kodieren, z.B. ist `_Just` ein Prisma und kann z.B. verwendet werden, um in einem `Maybe`-Objekt einen `Just`-Wert neu zu setzen, aber einen `Nothing`-Wert unverändert zu lassen.

```
*> set _Just 7 (Just "Sieben")
  Just 7
*> set _Just 7 Nothing
  Nothing
```

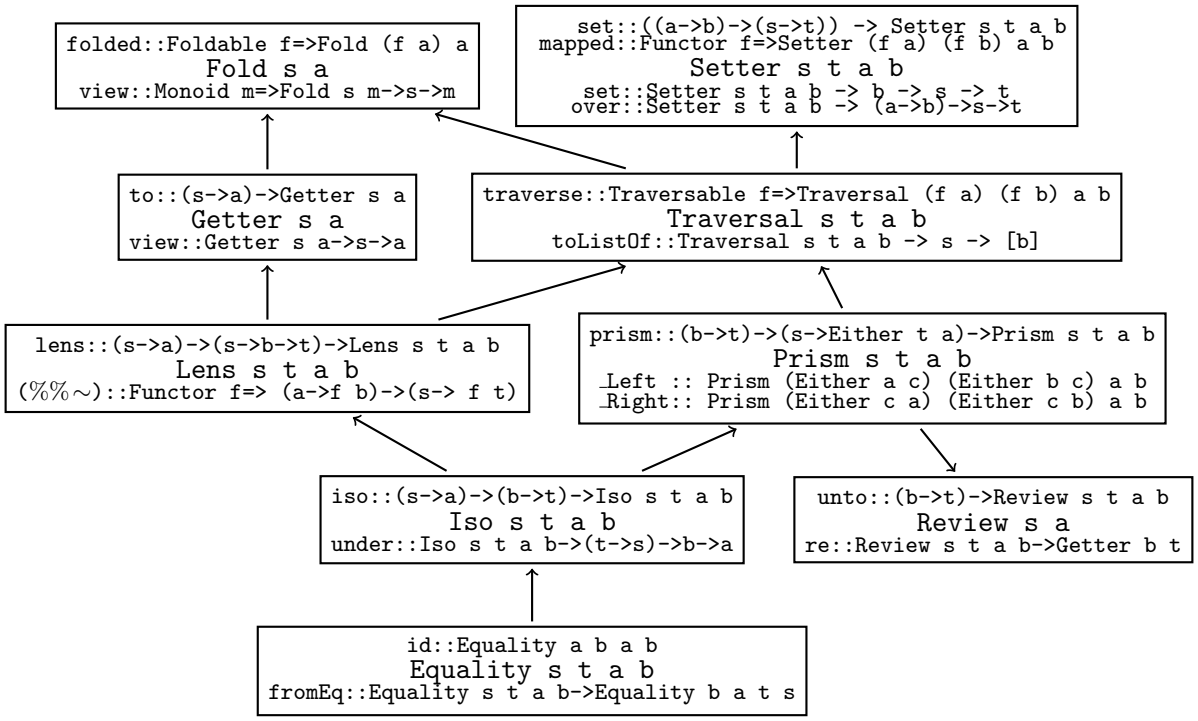


Abbildung 8.1.: Linsen-Hierachie

Analog kann man `over` und z.B. das Prisma `_Right` für den Either-Typ verwenden:

```
*> over _Right (3*) (Right 7)
Right 21
*> over _Right (3*) (Left 7)
Left 7
```

Prismen sind aber keine Getter, da ja möglicherweise kein Fokus existiert. Daher ist ein Aufruf `view _Left (Left 10)` nicht zulässig, denn man wüsste nicht was `view _Left (Right 20)` liefern sollte. Stattdessen gibt es ein `preview`, welches mit `Maybe` verpackte Werte liefert

```
*> preview _Left (Left 42)
Just 42
*> preview _Left (Right 42)
Nothing
*> preview _Just (Just 10)
Just 10
*> preview _Just Nothing
Nothing
```

Außerdem lassen sich Prismen invertieren:

```
> review _Left 69
Left 69
```

Die folgenden Gesetze müssen für Prismen gelten:

1. Eine Vorschau auf eine Rückschau gelingt immer mit dem gleichem Ergebnis:
   `preview p (review p x) == Just x`
2. Eine Rückschau auf eine *erfolgreiche* Vorschau sollte ebenfalls gelingen:
   `review p <$> preview p z == Just z`

8.3.5. Iso

Die Optik `Iso` ist für Isomorphismen und erlaubt es zwei Typen jederzeit ineinander zu überführen. Nützlich ist dies z.B. für `newtype`-Wrapper, oder `Maybe a` und `Either () a` oder `(a,b)` und `(b,a)`, etc. Für einen Isomorphismus braucht man zwei Funktionen `fw :: s -> a` und `rw :: a -> s`, so dass gilt:

```
fw . rw == id
rw . fw == id
```

Damit können wir eine `Iso`-Optik erzeugen: `iso :: (s -> a) -> (b -> t) -> Iso s t a b`. Wir ersparen uns den genauen Typ von `Iso` zu erläutern und betrachten einige Beispiele: Z.B. ist `swappedPair` ein `iso`:

```
swappedPair :: Iso' (a,b) (b,a)
swappedPair = iso (\(a,b) -> (b,a)) (\(b,a) -> (a,b))
```

Da Isos auch Prismen sind, können wir `review` und `preview` für diese verwenden.

```
*> preview swappedPair (1,True)
Just (True,1)
*> review swappedPair (1,True)
(True,1)
*> set (swappedPair._2) 5 (1,True)
(5,True)
```

Beachte, dass Optiken zwischen Haskell-Datentypen und z.B. deren XML- und/oder JSON-Repräsentationen, wie z.B. `Data.Aeson.Lens._JSON` Prismen sind, da das Parsen von XML/JSON fehlschlagen kann!

8.3.6. Nützliche vordefinierte Optiken

Wir listen zum Abschluss einige vordefinierte Optiken auf, die oft verwendet werden:
Linsen

```
(_1) :: Field1 s t a b => Lens s t a b -- Projektionen
(_2) :: Field2 s t a b => Lens s t a b -- Projektionen
```

Traversals

```
firstOf      :: Traversal' s a -> Maybe a -- == preview
ix:: Index m -> Traversal' m (IxValue m)
lastOf       :: Traversal' s a -> Maybe a
```

Prismen

```
_Nothing :: Prism' (Maybe a)          ()
_Just    :: Prism  (Maybe a) (Maybe b) a b
_Left    :: Prism (Either a c) (Either b c) a b
_Right   :: Prism (Either c a) (Either c b) a b
_Cons    :: Prism  [a] [b] (a,[a]) (b,[b])
```

8.4. Zusammenfassung und Quellen

Wir haben gesehen, wie funktionale Referenzen, auch bekannt als Linsen, den Zugriff auf tiefer liegende Teile von abstrakten Datenstrukturen ermöglichen.

Linsen sind Werte, d.h. Referenzen in Datenstrukturen können übergeben, manipuliert und mit `(.)` zusammengesetzt werden. In `Lens s t a b` ist `s` der äußere Datentyp `s`, und `a` der innere Datentyp.

Man kann `a` zu `b` verändern. Dann ist `t` der Datentyp, den man erhält, wenn man in `s` den Typ `a` durch `b` ersetzt.

Rank-N-Typen und Typklassen ermöglichen etwas, was sich sehr ähnlich wie Subtyping verhält, z.B. jede Linse ist ein Getter, jedes Prisma ein Traversal,... Verschiedene Implementierungen von Linsen sind möglich/erhältlich, auch für andere Programmiersprachen.

Die Vorteile des `lens`-Pakets sind, dass es sehr nützlich ist, um kurzen Code für große Änderungen zu schreiben. Z.B. macht es in Teilen Template Haskell überflüssig, man kann mit Linsen typsicher beliebige Tupel fokussieren:

```
> view _3 ('a','b','c')
'c'
> view _3 ('a','b','c','d','e','f','g')
'c'
> set _3 'z' ('a','b','c','d','e','f','g','h')
('a','b','z','d','e','f','g','h')
> :type _3
_3 ::(Field3 s t a b, Functor f)=> (a -> f b)-> s -> f t
```

Der Trick liegt hier in der Klasse `Field3`, eine Klasse für alle Tupel-artigen Container, welche ein drittes Element haben.

Ein Nachteil der `lens`-Bibliothek ist ihre Komplexität. So wirken oft selbst einfache Beschreibungen einschüchternd oder verwenden Kategorientheorie. Es gibt Verwirrung durch eine unüberschaubare Flut von Klassen, Typen, Funktionen und Operatoren. Die Typen sind sehr komplexe Typen und verwenden zahlreiche Typsynonyme und liefern hoffnungslose Typfehler. Schließlich dauert die Installation des Pakets dauert lange, aufgrund vieler Abhängigkeiten

Es gibt einige alternative Linsen-Bibliotheken, welche sich darauf konzentrieren, möglichst leichtgewichtig zu sein, z.B. `fclabels`.

Die wesentlichen Quellen dieses Abschnitts sind (Jost, 2019) die `lens`-Bibliothek (Kmett, 2020) und das Buch (Penner, 2020).

9

Parallelität, Ausnahmen und Nebenläufigkeit in Haskell

9.1. 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.

9.2. 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.

9.2.1. 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. Hiefür 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.

9.2.2. Messen und Analysieren des Ressourcenverhaltens

9.2.2.1. 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.

9.2.2.2. 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.

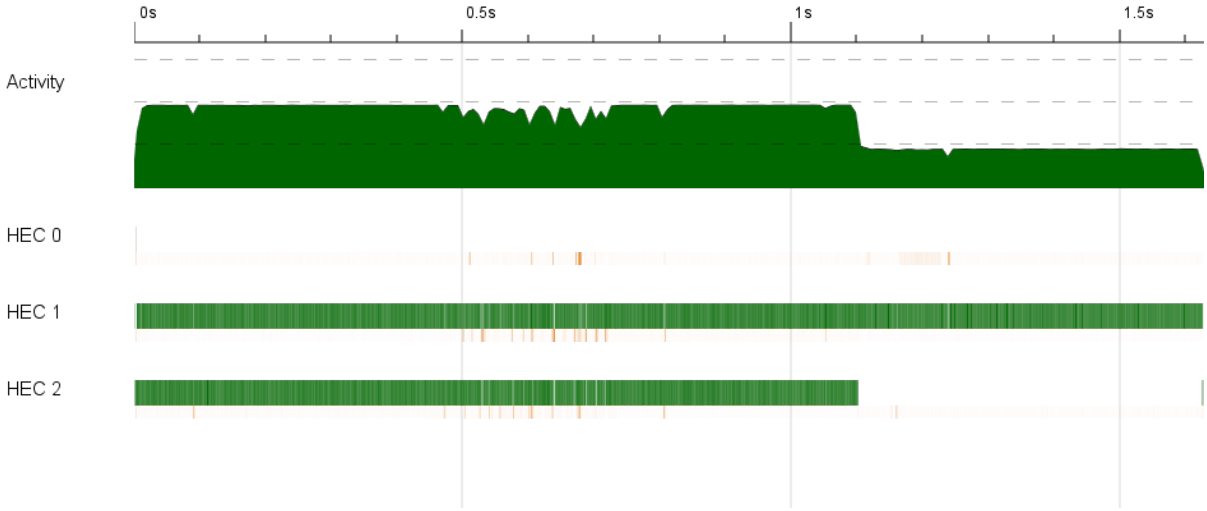


Abbildung 9.1.: Ansicht von ThreadScope

9.2.2.3. 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 9.1 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.

9.3. 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
```

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.

9.3.1. 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
```

9.3.2. 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
```

9.3.3. 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]]
```

9.3.4. 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.

9.4. 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`

9.4.1. 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.

9.4.2. 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
```

9.4.3. 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.

9.5. 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.

9.5.1. 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 Source
  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.

9.5.2. 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.

9.5.3. 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

9.6. Nebenläufigkeit

Zur nebenläufigen Programmierung betrachten wir zwei Ansätze in Haskell: Concurrent Haskell und STM Haskell.

9.6.1. 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. Nicht-blockierende 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`.

9.6.2. 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)
```

9.7. Software Transactional Memory

9.7.1. 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.

9.7.2. 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 TVars 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 TVars 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* TVars 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
```

9.7.3. 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

9.8. Quellen

Diese Kapitel verwendet Material aus (Jost, 2019) und ist ähnlich aufgebaut. Primär ist zu allen Themen des Kapitels das Buch (Marlow, 2013) empfehlenswert. STM Haskell wird ausführlich in (Peyton-Jones, 2007) beschrieben.

10 Spracherweiterungen von Haskell

In diesem Kapitel betrachten wir einige Spracherweiterungen, die im GHC verfügbar sind und welche oft verwendet werden. Dies dient auch der Vorbereitung des Kapitels zum Webframework Yesod, da dieses einige der Erweiterungen verwendet. Wir werden nicht auf alle Details der Erweiterungen eingehen, sondern führen diese eher anhand von Beispielen ein.

10.1. Multi-Parameter Typklassen

Die Erweiterung `MultiParamTypeClasses` erlaubt Typklassen mit mehreren Parametern. Ein Beispiel ist die folgende Klasse `VarMonad` für solche Monaden, die Variablen (Speicherplätze) zur Verfügung stellen:

```
class Monad m => VarMonad m v where
  newRef   :: a -> m (v a)
  readRef  :: v a -> m a
  writeRef :: v a -> a -> m ()
```

Z.B. kann man eine Funktion schreiben, welche den Inhalt einer Speicherzelle um 1 erhöht und für alle Instanzen von `VarMonad` funktioniert:

```
addOne  :: (VarMonad m v, Num a) => v a -> m ()
addOne v = do x <- readRef v
              writeRef v $ x+1
```

Instanzen für `VarMonad` können z.B. sein:

```
instance VarMonad IO IORef
  where
    newRef   = newIORef
    readRef  = readIORef
    writeRef = writeIORef

instance VarMonad STM TVar
  where
    newRef       = newTVar
    readRef      = readTVar
    writeRef     = writeTVar

instance VarMonad IO MVar
  where
    newRef   = newMVar
    readRef  = readMVar
    writeRef m v = swapMVar m v >> return ()
```

Die Funktion `addOne` kann mit `IORef`, `TVar` oder `MVar` arbeiten. Das folgende Programm ist z.B. möglich:

```
main = do
  ioRef <- newIORef 3; mvar <- newMVar 5; stmRf <- newTVarIO 7
  addOne ioRef
  addOne mvar
  atomically $ addOne stmRf
  print =<< readRef ioRef     -- Ausgabe: "4"
  print =<< readRef mvar      -- Ausgabe: "6"
  print =<< readTVarIO stmRf  -- Ausgabe: "8"
```

Wenn man Typklassen als eine Menge von Typen auffasst (all jene, die eine Instanz definieren), so kann man Multi-Parameter-Typklassen als Relation auf Typen auffassen.

Probleme dabei sind, dass die Typ-Inferenz deutlicher komplizierter und teilweise unklar ist. In obiger Klasse `VarMonad` hängt der zweite Parameter `v` von `m` ab, z.B. geht die Kombination `IO` und `TVar` nicht. Dieser Zusammenhang wird jedoch nur durch Multiparameter-Typklassen noch nicht richtig erkennbar. Eine Lösung sind sogenannte *Functional Dependencies*, die mit der Erweiterung `FunctionalDependencies` verfügbar sind. Diese erlauben es, bei der Typklassendeklaration die Funktionale Abhängigkeit `m -> v` festzulegen. Diese besagt, dass der Typ von `v` eindeutig durch den Typ von `m` bestimmt wird.
Verwendet man diese Definition:

```
class Monad m => VarMonad m v | m -> v where
  newRef   :: a -> m (v a)
  readRef  :: v a -> m a
  writeRef :: v a -> a -> m ()
```

so kann man nicht mehr beide Instanzen

```
instance VarMonad IO  IORef   where ...
instance VarMonad IO  MVar    where ...
```

angeben, da dies die Funktionale Abhängigkeit verletzt. Eine Möglichkeit ist es, umgekehrt zu fordern, dass der Typ von `m` von `v` abhängt.

```
class Monad m => VarMonad m v | v -> m where
  newRef   :: a -> m (v a)
  readRef  :: v a -> m a
  writeRef :: v a -> a -> m ()
```

Dann können beide Instanzen definiert werden. Allgemein kann man mehrere funktionale Abhängigkeiten definieren. Betrachte die Klassen

```
class A a b
class B a b | a -> b
class C a b | a -> b, b -> a
```

Klasse `A` repräsentiert beliebige binäre Relationen, während Klasse `B` partielle Funktionen definiert, da die Abhängigkeit `a -> b` Rechtseindeutigkeit impliziert. Die Klasse `C` fordert Links- und Rechtseindeutigkeit. Beispiele dazu sind:

```
-- alles erlaubt:
instance A Char Bool
instance A Char Int
instance A Int Bool

instance B Char Bool
instance B Char Int -- verboten wegen a -> b und Char -> Bool schon definiert
instance B Int Bool -- erlaubt

instance C Char Bool
instance C Char Int -- verboten wegen a -> b (und Char -> Bool schon definiert)
instance C Int Bool -- verboten wegen b -> a (und Bool -> Char schon definiert)
```

Funktionale Abhängigkeiten können auch mehrere Parameter umfassen, z.B.

```
class D a b c d e f | a -> b c, d e -> f
```

besagt, dass der Parameter `a`, die Parameter `b` und `c` eindeutig bestimmt, und dass die Kombination der beiden Parameter `d` und `e` den Parameter `f` eindeutig bestimmen. Beispiele hierzu sind

```
instance D Bool Int Char Float Double Int
instance D Bool Int Char Float Double Integer     -- nicht erlaubt wegen d e -> f
instance D Bool Int Char Float Float Int          -- erlaubt
instance D Bool Integer Char Float Double Integer -- nicht erlaubt wegen a -> b c
instance D Bool Int Integer Float Double Integer  -- nicht erlaubt wegen a -> b c
```

10.2. Typfamilien

Wir betrachten Typfamilien, die mit der Erweiterung `TypeFamilies` verfügbar sind. Während Typklassen dem Überladen von Funktionen dienen, dienen Typfamilien dem Überladen von Datentypen. Dabei darf (wie bei Funktionen und Typklassen) die Implementierung des Datentyps je nach Instanz *unterschiedlich* sein. Typfamilien sind eine sehr mächtige, ausdrucksstarke Erweiterung des Typsystems, welche bereits Funktionen auf Typ-Ebene gestatten.
Wir behandeln das Thema nicht in voller Tiefe und beschränken uns im Wesentlichen auf Beispiele.

```
class Mutation m where
  type Ref m  :: * -> *
  newRef      :: a -> m (Ref m a)
  readRef     :: Ref m a -> m a
  writeRef    :: Ref m a -> a -> m ()

instance Mutation STM where
  type Ref STM = TVar
  newRef       = newTVar
  readRef      = readTVar
  writeRef     = writeTVar
```

`Ref m` berechnet den Typ Referenz. Jetzt können wir aber nur noch eine Instanz für `IO` angeben (entweder `IORef` oder `MVar`).

```
instance Mutation IO where
  type Ref IO  = IORef
  newRef       = newIORef
  readRef      = readIORef
  writeRef     = writeIORef
```

Als weiteres Beispiel zu Typfamilien betrachten wir die folgende Klasse `Add`, welche den Summentyp als Typfamilie `SumTy` definiert:

```
{-# LANGUAGE TypeFamilies #-}
class Add a b where
  type SumTy a b
  add :: a -> b -> SumTy a b
```

`SumTy` ist eine Funktion auf Typebene, die zwei Typen erhält und dafür einen dritten Typ berechnet. Die Idee ist, dass die Summe zweier unterschiedlicher Typen für Zahlen noch berechnet werden kann, wenn das Ergebnis denselben oder einen allgemeineren Typ haben darf. Die Klasse `Add` erlaubt es, die `add`-Funktion entsprechend zu überladen. Z.B. können wir als Instanz definieren:

```
instance Add Integer Double where
  type SumTy Integer Double = Double
  add x y = fromIntegral x + y
```

oder wenn beide Eingaben vom gleichen, numerischen Typ sind:

```
instance (Num a) => Add a a where
  type SumTy a a = a
  add x y = x + y
```

Auch eine Addition einer Zahl zu einer Liste ist denkbar (durch elementweises Addieren):

```
instance (Add Integer a) => Add Integer [a] where
  type SumTy Integer [a] = [SumTy Integer a]
  add x y = map (add x) y
```

10.2.1. Top-level Typfamilien

Wir betrachteten bis jetzt nur mit Klassen *assoziierte Typfamilien*. Dies ist jedoch keine Notwendigkeit, Typfamilien können auch außerhalb von Klassen auftreten, und selbständig definiert sein, wie z.B.:´

```
type family G a where
  G Int = Bool
  G a   = Char
```

Hier haben wir eine Funktion `G` auf Typ-Ebene, welche den Typ `Int` auf den Typ `Bool` abbildet und alle anderen Typen auf `Char`.

Das obige Beispiel ist eine *geschlossene* Deklaration. *Offene*, d.h. erweiterbare Deklarationen sind ebenfalls möglich:

```
type family F a b :: * -> *
type instance F Int Bool = Maybe
type instance F Int Int = Either ()
```

Hier wird eine Funktion `F` auf Typ-Ebene definiert, die zwei Typen erwartet und in diesem Fall einen Typkonstruktor (vom Kind `* -> *`) liefert. Für `Int` und `Bool` gibt `F Maybe` zurück, Für `Int` und `Int` gibt `F  Either ()` zurück. Die Deklaration ist offen, da weitere Fälle hinzugefügt werden können. Tatsächlich werden durch die Deklaration Typsynonyme definiert (`F Int Bool` ist Synonym zu `Maybe` usw.). Z.B. können wir definieren:

```
fun :: F Int Bool Bool
fun = Just True
```

10.2.2. Datentyp-Familien

Neben Typfamilien für Typsynonyme gibt es auch Typfamilien für Datentypen (passend zu den `data` und `newtype`-Deklarationen). Dies können wiederum entweder assoziiert mit Typklassen oder auf Top-Level auftreten. Als einfaches Beispiel betrachten wir einen Typ für strikte (d.h. voll ausgewertete Listen). Eine Liste mit Elementtyp `Char` kann wie üblich repräsentiert werden. Eine Liste mit Elementtyp `()` kann effizient als eine eine Zahl (die Länge der Liste) repräsentiert werden. Daher unterscheiden sich beide Implementierungen völlig und dies kann nicht mit normalen parametrisierten Datentypen implementiert werden.
Mit Typfamilien kann dies bewerkstelligt werden durch

```
-- Deklaration der Datentyp-Familie
data family XList a
-- Instanz für Char:
data instance XList Char = XCons !Char !(XList Char) | XNil
-- Instanz für ()
data instance XList () = XListUnit !Int
```

Die so definierten Typen können nicht immer allgemein verwendet werden, d.h. insbesondere ist der folgende Code nicht möglich:

```
foo :: XList a -> Int -- ERROR
foo XNil          = 0
foo (XCons _ t)   = 1 + foo t
foo (XListUnit n) = n
```

Der Grund liegt darin, dass die Familie `XList` offen ist und daher neue Fälle nach hinzufügen weiterer Instanzen auftreten können. Mit instanziierten Typen und unterschiedlichen Funktionsnamen ist eine Definition jedoch möglich als:

```
foo1 :: XList Char -> Int
foo1 XNil  = 0
foo1 (XCons _ t) = 1 + foo1 t
foo2 :: XList () -> Int
foo2 (XListUnit n) = n
```

Eine Alternative zu dieser Definition sind GADTs, die wir nun betrachten.

10.3. Generalised Algebraic Datatypes

Wir verdeutlichen ein Problem mit herkömmlichen Datentypen. Nehmen wir an, wir haben einen Typ `Expr` der sowohl Zahlen, als auch Boolesche Werte darstellen soll:

```
data Expr = ConstI Int          -- integer constants
          | ConstB Bool         -- boolean constants
          | Or  Expr Expr       -- logic disjunction
          | Add Expr Expr       -- add two expressions
          | Odd Expr            -- convert int to bool
          | If  Expr Expr Expr -- conditional
```

Z.B. ist `(Or (ConstB True) (ConstB False))` ein gültiger Ausdruck, aber auch `If (Odd (Add (ConstI 1) (ConstI 0))) (ConstI 0) (ConstI 1)`. Der Typ erlaubt aber auch „typfalsche Ausdrücke“ wie `Or (ConstI 1) (ConstB True)`. Würden wir eine Auswertefunktion schreiben für `Expr`, so ist unklar, welchen Ergebnistyp diese hätte, da sowohl `Bool` als auch `Int` in Frage kämen.

Eine mögliche Abhilfe wäre `eval :: Expr -> Either Bool Int`, aber es löst nicht das Problem, das der Datentyp keine Typsicherheit garantiert und Ausdrücke wie `eval $ Or (ConstB False) (ConstI 69)` erlaubt wären.

Ein Lösung des Problems wäre es, zwei getrennte Typen für Boolesche und arithmetische Ausdrücke zu verwenden:

```
data BExpr = ConstB Bool | Or BExpr BExpr | Odd IExpr | IfB BExpr BExpr BExpr
data IExpr = ConstI Int  | Add IExpr IExpr | IfI BExpr IExpr IExpr

evalB :: BExpr -> Bool
evalB (ConstB c) = c
evalB (Or a b)   = (evalB a) || (evalB b)
evalB (Odd i)    = odd (evalI i)
evalB (IfB c t e) | evalB c   = evalB t
                  | otherwise = evalB e
evalI :: IExpr -> Int
evalI (ConstI c) = c
evalI (Add a b)  = (evalI a) +  (evalI b)
evalI (IfI c t e) | evalB c   = evalI t
                  | otherwise = evalI e
```

Diese Lösung funktioniert, allerdings gibt es nun keine generische `eval`-Funktion, wir müssen die passende Funktion aufrufen, und der Code für die Auswertung ist verteilt und wechselseitig rekursiv.

Ein weiterer Ansatz ist die Verwendung von Typ-Familien und einer Typklasse für die `eval`-Funktion. Die Idee entspricht dabei im Wesentlichen der vorherigen, mit dem Unterschied, dass der `Expr`-Typ und die `eval`-Funktion beide überladen werden:

```
class ExprClass a where
  data Expr a :: *
  eval :: Expr a -> a

instance ExprClass Bool  where
```

```
  data Expr Bool  = ConstB Bool | Or  (Expr Bool) (Expr Bool)
                  | Odd (Expr Int) | IfB (Expr Bool) (Expr Bool) (Expr Bool)
  eval (ConstB c) = c
  eval (Or a b)   = (eval a) || (eval b)
  eval (Odd i)    = odd (eval i)
  eval (IfB c t e) | eval c = eval t
                   | otherwise = eval e
instance ExprClass Int where
  data Expr Int = ConstI Int  | Add (Expr Int)  (Expr Int)
                | IfI (Expr Bool) (Expr Int) (Expr Int)
  eval (ConstI c) = c
  eval (Add a b)  = (eval a) + (eval b)
  eval (IfI c t e) | eval c    = eval t
                   | otherwise = eval e
```

Der `eval`-Code ist allerdings immer noch verteilt und der Code für `If` ist nicht generisch, da es Wiederholungen für `IfI` und `IfB` gibt.

Wir erinnern daran, dass Datenkonstruktoren wie Funktionen aufgefasst werden können, die jedoch nichts berechnen, sondern ein Speicherobjekt erzeugen. Entsprechend haben die Konstruktoren Funktionstypen. Zum Beispiel haben im ursprünglichen Typ `Expr`

```
data Expr = ConstI Int         -- integer constants
          | ConstB Bool        -- boolean constants
          | Or  Expr Expr      -- logic disjunction
          | Add Expr Expr      -- add two expressions
          | Odd Expr           -- convert int to bool
          | If  Expr Expr Expr -- conditional
```

die Konstruktoren die folgenden Typen:

```
ConstI :: Int -> Expr
ConstB :: Bool -> Expr
Or     :: Expr -> Expr -> Expr
Add    :: Expr -> Expr -> Expr
Odd    :: Expr -> Expr
If     :: Expr -> Expr -> Expr -> Expr
```

Mit der Erweiterung `GADTs` (*Generalised Algebraic Datatypes*) wird dieses Prinzip genutzt und erweitert bzw. verallgemeinert, indem Konstruktoren explizit durch ihren Typ beschrieben werden und der *Ergebnistyp* darf eine *beliebige Instanz* des deklarierten Typs sein.

Komplett analog zu obigem Typen dürfen wir mit der Erweiterung schreiben:

```
{-# LANGUAGE GADTs #-}
data Expr where
  ConstI :: Int -> Expr
  ConstB :: Bool -> Expr
  Or     :: Expr -> Expr -> Expr
  Add    :: Expr -> Expr -> Expr
  Odd    :: Expr -> Expr
  If     :: Expr -> Expr -> Expr -> Expr
```

Dies ändert noch nichts. Der Trick besteht nun darin, den `Expr`-Typ mit einem Typparameter zu parametrisieren, d.h. `Expr a` zu definieren, sodass `Expr Bool` Ausdrücke mit Booleschem Wert und `Expr Int` Ausdrücke mit Int-Wert darstellen. Gleichzeitig dürfen wir für die Ergebnistypen Instanzen von `Expr a` (also insbesondere `Expr Bool` und `Expr Int`) verwenden und damit festlegen, welche Ausdrücke wozu gehören. Das ergibt:

```
{-# LANGUAGE GADTs #-}
data Expr a where
  ConstI :: Int -> Expr Int
  ConstB :: Bool -> Expr Bool
  Or     :: Expr Bool -> Expr Bool -> Expr Bool
  Add    :: Expr Int -> Expr Int -> Expr Int
  Odd    :: Expr Int -> Expr Bool
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Nun sind Ausdrücke, die den Typcheck fehlerfrei durchlaufen auch immer als Boolesche bzw. arithmetische Ausdrücke korrekt getypt, denn z.B. wird `Or (ConstB False) (ConstI 69)` vom Haskell-eigenen Typcheck als nicht korrekt getypt abgewiesen (denn `Or` erwartet beide Argument vom Typ `Expr Bool`, aber `ConstI 69` ist vom Typ `Expr Int`). Ebenso können wir `eval :: Expr a -> a` nun typsicher definieren, wir können Pattern-Matching hierfür wie gewohnt verwenden:

```
eval :: Expr a -> a
eval (ConstB c) = c
eval (ConstI c) = c
eval (Or  a b)  = (eval a) || (eval b)
eval (Add a b)  = (eval a) +  (eval b)
eval (If c t e) | eval c    = eval t
                | otherwise = eval e
eval (Odd e)    = odd (eval e)
```

Im Gegensatz zu Typ-Familien ist diese Definition nicht erweiterbar, d.h. alle Definitionen müssen am gleichen Ort sein. Die `deriving`-Syntax ist in GADT-Syntax nicht möglich, es gibt jedoch die Erweiterung `StandaloneDeriving`, die eine zusätzliche `deriving`-Klausel außerhalb der Datentyp-Definition erlaubt:

```
{-# LANGUAGE GADTs StandaloneDeriving #-}
data Expr a where
  ConstI :: Int -> Expr Int
  ConstB :: Bool -> Expr Bool
  Or     :: Expr Bool -> Expr Bool -> Expr Bool
  Add    :: Expr Int -> Expr Int -> Expr Int
  Odd    :: Expr Int -> Expr Bool
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a
deriving instance (Show a) => Show (Expr a)
deriving instance (Eq a) => Eq (Expr a)
```

Die Anwendung von GADTs liegt vorallem in *typsicheren domänenspezifischen Sprachen*, wie wir es eben vorgeführt haben.

10.4. Typfamilien und GADTs

Das Standardbeispiel für eigenständige Typ-Familien ist durch Vektoren motiviert, wobei wir hier unter einem *Vektor* eine Liste mit *statisch* bekannter Länge interpretieren, sodass Vektoren verschiedener Größe verschiedene Typen haben. Mit GADTs können wir einen solchen Typ umsetzen durch:

```
{-# LANGUAGE GADTs #-}

data Zero
data Succ n
data Vec size a where -- GADT Syntax
  VecZ :: Vec Zero a
  VecS :: a -> Vec size a -> Vec (Succ size) a
```

Einen Vektor der Länge 2 erkennt man nun am Typ:

```
v2 :: Vec (Succ (Succ Zero)) Int
v2 = VecS 1 $ VecS 2 $ VecZ
```

Die folgende Funktion fügt ein Element an aufsteigend geordneter Stelle dem Vektor hinzu. Der Typ zeigt, dass die Länge des Vektor um eins ansteigt:

```
insertVec :: (Ord a) => a -> Vec n a -> Vec (Succ n) a
insertVec a     VecZ                  = VecS a VecZ
insertVec a bv@(VecS b v) | a <= b    = VecS a bv
                          | otherwise = VecS b $ insertVec a v
```

Z.B. können wir aufrufen:

```
*> :t insertVec 10 v2
insertVec 10 v2 :: Vec (Succ (Succ (Succ Zero))) Int
```

Mit dem gezeigten Code können wir vom Compiler überprüfen lassen, dass Sortieren eines Vektors dessen Größe unverändert lässt:

```
isortVec :: (Ord a) => Vec n a -> Vec n a
isortVec VecZ       = VecZ
isortVec (VecS a v) = insertVec a $ isortVec v
```

Doch welchen Typ hat das Aneinanderhängen zweier Vektoren? Programmiert werden kann es durch

```
appendVec  VecZ      v = v
appendVec (VecS a w) v = VecS a (appendVec w v)
```

Allerdings kann Haskell keinen Typ inferieren. Ohne weiteres Zutun können wir den Typ auch nicht angeben, denn er ist von der Form `appendVec :: Vec n a -> Vec m a -> Vec n+m a`, wobei es das `n+m` nicht auf Typebene gibt. Mithilfe einer Typfamilie können wir dies jedoch bewerkstelligen, indem wir damit die Addition auf *Typebene* als Typfamilie definieren:

```
{-# LANGUAGE GADTs TypeFamilies #-}
appendVec :: Vec n a -> Vec m a -> Vec (Plus n m) a
appendVec  VecZ      v = v
appendVec (VecS a w) v = VecS a (appendVec w v)
type family Plus m n :: *
type instance Plus Zero n  = n
type instance Plus (Succ m) n = Succ (Plus m n)
```

Dies geht auch mit der geschlossenen Variante der Typfamilie:.

```
{-# LANGUAGE GADTs TypeFamilies #-}
appendVec :: Vec n a -> Vec m a -> Vec (Plus n m) a
appendVec  VecZ      v = v
appendVec (VecS a w) v = VecS a (appendVec w v)
type family Plus m n :: * where
 Plus Zero n  = n
 Plus (Succ m) n = Succ (Plus m n)
```

10.5. DataKinds

Ein Problem im obigen Beispiel ist, dass es keinen Zusammenhang zwischen `Zero` und `Succ` gibt, und z.B. auch der Typ `Vec Bool String` erlaubt ist, obwohl er keinen Sinn macht. Das eigentliche Problem ist, dass der Kind von `Vec` zu allgemein ist. Dieser ist `* -> * -> *` und lässt damit jeden Typen als ersten Parameter zu. Wir möchten diesen jedoch gerne so einschränken, dass dort nur (aus `Zero` und `Succ` aufgebaute) natürliche Zahlen stehen. Um dies zu bewerkstelligen, muss man in das Kind-System eingreifen. Haskell bietet mit der Erweiterung `DataKinds` eine Möglichkeit dies zu tun: Am Ende hat `Vec` den Kind `Nat -> * -> *` wobei `Nat` ein neuer Kind ist, der genau die Typen umfasst, die aus `Zero` und `Succ` aufgebaut sind. Eine (im GHC *nicht* eingebaute!) Möglichkeit wäre es, neue Syntax für Kinddefinitionen zu erlauben, z.B. von der Form:

```
datakind Nat = Zero | Succ Nat
```

Dieser imaginäre Code würde einen neuen Kind `Nat` erzeugen, der aus den Typkonstruktoren `Zero` (vom Kind `Nat`) und `Succ` vom Kind `Nat -> Nat` aufgebaut ist. Dies würde auch das Problem der Unabhängigkeit der Typen `Zero` und `Succ` beheben, da diese Typen nun über ihren Kind `Nat` miteinander verbunden sind. Tatsächlich sieht obige imaginäre Kind-Definition analog zu einer Datentyp-Definition aus, nur eben eine Abstraktionsstufe höher: Eine Data-Anweisung definiert einen Typ mit seinen Datenkonstruktoren in der Form

```
data      Typ = Datenkonstruktor_1 arg_1_1...arg_1_m_1
              | ...
              | Datenkonstruktor_n arg_n_1...arg_n_m_n
```

Eine imaginäre Data-Kind-Anweisung definiert einen Kind mit seinen Typkonstruktoren in der Form

```
datakind Kind = Typkonstruktor_1 arg_1_1 ... arg_1_m_1
              | ...
              | Typkonstruktor_n arg_n_1 arg_n_m_n
```

In Realität gibt es die `datakind`-Anweisung nicht, sondern aus jeder Datentyp-Definition mit `data` wird intern bei Verwendung der Erweiterung `DataKinds` ein Kind gleichen Namens wie der Typ und Typkonstruktoren gleichen Namens wie die Datenkonstruktoren erzeugt, wobei den Typkonstruktornamen ein Hochkomma im Namen vorangestellt werden kann (manchmal ist dies zur Eindeutigkeit notwendig).
Z.B. erzeugt

```
{-# LANGUAGE DataKinds #-}
data Bool = True | False
```

neben dem Typ `Bool` auch den Kind `Bool` und neben den Datenkonstruktoren `True` und `False` auch die Typen `True` und `False`, die auch als `'True` und `'False` geschrieben werden können.
Beachte, dass es keine weitere Beziehung zwischen dem Typ `Bool` und dem Kind `Bool` gibt.
Für das Vektorenbeispiel können wir alle drei Erweiterungen nun verwenden und es programmieren als:

```
{-# LANGUAGE GADTs TypeFamilies DataKinds #-}
data Nat = Zero | Succ Nat
data Vec :: Nat -> * -> * where -- GADT Syntax
  VecZ :: Vec 'Zero a
  VecS :: a -> Vec size a -> Vec ('Succ size) a

appendVec :: Vec n a -> Vec m a -> Vec (Plus n m) a
appendVec  VecZ      v = v
appendVec (VecS a w) v = VecS a (appendVec w v)
type family Plus m n :: Nat where
 Plus 'Zero n  = n
 Plus ('Succ m) n = 'Succ (Plus m n)
```

Beachte, dass es nun den neuen Kind `Nat` gibt und dass `Zero` und `Succ` sowohl als Daten- wie auch als Typkonstruktoren vorkommen:

```
*> :t Succ
Succ :: Nat -> Nat
*> :k Succ
Succ :: Nat -> Nat
*> :k 'Succ
'Succ :: Nat -> Nat
```

Da es nun unterschiedliche Kinds gibt, wird nochmal ein Typsystem über Kinds erzeugt. Diese „Typen“ werden Sorte genannt und z.B. sind Kind `*` und Kind `Nat` von der Sorte `BOX`. Alle Kinds haben die gleiche Sorte `BOX` (es wird hier nicht weiter unterschieden in Funktionen und Nicht-Funktionen). Ein Problem bei diesem Ansatz ist noch, dass man teilweise Funktionen wie `Plus` zwei Mal definieren muss, nämlich einmal auf Termebene und einmal auf Typebene. Auch hierfür gibt es Ansätze wie Dependent Types oder das singletons-Paket. Wir erläutern diese an dieser Stelle nicht und verweisen auf die entsprechende Literatur.

10.6. Existentielle Typen

Existentielle Typen dienen im Wesentlichen dazu Typparameter in einer Datentypdefinition „verschwinden“ zu lassen. Betrachte z.B. den polymorphen Datentyp

```
data TaskList tType dType = TaskList {tasks    :: [tType]
                                     ,deadline :: dType}
```

der Aufgaben als Liste von einzelnen Aufgaben polymorph über dem Parameter `tType` repräsentiert und zusätzlich noch ein Datum speichert zu dem die Arbeiten erledigt sein sollen. Da wir den Datumstyp nicht wissen, wird auch dieser polymorph verwendet mit dem Typparameter `dType` ist.

Eventuell sind wir gar nicht genau am Typ für die `deadline` interessiert, da wir diese nur anzeigen lassen möchten. Mit existentiellen Typen (die in Wirklichkeit durch Allquantifizierung ausgedrückt werden), können wir schreiben:

```
{-# LANGUAGE ExistentialQuantification #-}
data TaskList tType = forall dType. Show dType => TaskList { tasks    :: [tType]
                                                           , deadline :: dType  }
```

Der Vorteil hierbei ist, dass der Typparameter `dType` nun verschwindet und wir den Parameter nicht überall (instanziert) verwenden müssen. Tatsächlich legt obige Quantifizierung fest, dass wir jeden Typ für `dType` verwenden können, der Instanz der Klasse `Show` ist. Tatsächlich können wir nun auch nichts anderes mit dem Feldeintrag `deadline` machen, als es anzeigen zu lassen:

```
showDate :: TaskList a -> String
showDate (TaskList _ dl) = show dl
```

Z.B. führt

```
fun (TaskList _ dl) = dl
```

zum Fehler. Verwendet man die existentielle Quantifizierung ohne Typklassenconstraint, z.B.

```
data TaskList tType = forall dType. TaskList {tasks :: [tType],deadline  :: dType}
```

so bedeutet dies, dass man jeden beliebigen Typ für `dType` verwenden kann, und umgekehrt auch, dass man mit dem `deadline`-Feld nichts mehr anfangen kann (außer es zu übergeben), da man an den Typ nicht mehr heran kommt.

Eine Verwendung existentieller Typen ist das Erstellen heterogener Datenstrukturen. Z.B. kann man definieren

```
data Object = forall a. Show a => Obj a
```

und nun eine Liste von Objekten (eigentlich unterschiedlichen Typs) erzeugen:

```
*> let list = [Obj 1, Obj True, Obj 'A']
*> :t list
list :: [Object]
```

Mit den Listenelementen kann man nun nichst anderes machen als sie anzeigen zu lassen:

```
*> map (\(Obj s) -> show s) list
["1","True","'A'"]
```

Wir erläutern, wo sich die *existentielle* Quantifizierung bei den existentiell quantifizierten Typen versteckt. Der Konstruktor `Obj` hat den Typ `Obj :: Show a => a -> Object` was äquivalent zu `Obj :: forall a. Show a => a -> Object` ist. Dies jedoch ebenso äquivalent zu `Obj :: (exists a. Show a => a) -> Object`, da der erste Typ sagt, für jeden Typ `a` der Instanz von `Show` ist, kann daraus ein `Object` erstellt werden. Der zweite Typ sagt: sobald man einen Typ `a` hat, der Instanz von `Show` ist, kann man ein `Object` erstellen. Das Schlüsselwort `exists` gibt es nicht in Haskell, da die existentielle Quantifizierung mit dem All-Quantor ausgedrückt wird. Auch mit Prädikatenlogik lässt sich obige Umformung beobachten, denn es gilt die Äquivalenz $\forall x.(P(x) \to Q) \equiv (\exists x.P(x)) \to Q$ (falls x nicht in Q vorkommt).

10.7. View Patterns und Pattern Synonyms

Es ist gute Praxis, Datentypen (in Modulen) abstrakt zu halten, z.B. um später gefahrlos die Repräsentation zu ändern. Betrachte z.B. das Modul `Data.Map`, welches den *abstrakten Datentyp* `Map k v` zur Verfügung stellt, aber die Implementierung jedoch versteckt. Ausschließlich bereitgestellte Funktionen sind als Schnittstelle verwendbar. Ein Nachteil ist, dass wir kein Pattern Matching für den Typ `Map k v` einsetzen können, bzw. dies nur indirekt machen können, nach Konvertierung:

```
getMinKeyVal :: Map k v -> Maybe (k,v)
getMinKeyVal m = case (toAscList m) of (h:_) -> Just h
                                       []    -> Nothing
```

Anders ausgedrückt, kann man sagen: Eine *View-Funktion* wie z.B. `toAscList :: Map k v -> [(k,v)]` erlaubt matching gegen eine *Ansicht* des Typs `Map k v`.

Die Spracherweiterung *View Patterns* bietet syntaktischen Zucker, um View-Funktionen innerhalb von Pattern Matches einzusetzen:

```
{-# LANGUAGE ViewPatterns #-}
getMinKeyVal :: Map k v -> Maybe (k,v)
getMinKeyVal (toAscList -> (h:_)) = Just h
getMinKeyVal _                    = Nothing
```

Die Spracherweiterung `ViewPatterns` erlaubt allgemein die Verwendung von Patterns der Form `(f -> p)` wobei

- `f` ist eine Funktion, welche auf das zu matchende Argument angewandt wird,
- Das Ergebnis wird mit Pattern `p` gematched, falls möglich,
- Dieses Pattern schlägt fehl, wenn der Match mit dem *Ergebnis* der Funktionsanwendung nicht gelingt

Wenn die View-Funktion eine Ausnahme wirft, dann wird nicht einfach der nächste Fall geprüft, sondern mit der Ausnahme abgebrochen. Betrachte z.B.

```
foo (head -> h) = Just h
foo _           = Nothing
```

Dann schlägt `foo []` fehl, da `head []` eine Ausnahme wirft.

Wenn die View-Funktion keine Ausnahme wirft, aber der der anschliessende Pattern-Match fehl schlägt, dann wird einfach der nächste Fall geprüft. Z.B.

```
demo :: Int -> Int -> Int -> Int
demo x (even -> True) z = x+z
demo x y  (even-> True) = x+y


Prelude> demo 1 3 2
4
Prelude> demo 1 2 3
4
```

Auch Verschachtelungen von View Patterns sind möglich und zulässig:

```
minmax :: [Int] ->  (Int,Int)
minmax (sort -> mi:(reverse -> mx:_)) = (mi,mx)
minmax _ = error "minmax argument size > 1 expected"
```

10.7.1. View Patterns vs Pattern Guards

Eine noch allgemeinere Alternative bieten die bereits behandelten Pattern-Guards, da hier beliebige Ausdrücke gematched werden können.

```
getMinKeyValPG :: Map k v -> Maybe (k,v)
getMinKeyValPG m | (h:_) <- toAscList m = Just h
                 | otherwise            = Nothing
```

Pattern-Guards lassen sich jedoch nicht derart verschachteln, weshalb man hier Zwischenvariablen einführen muss:

```
minmaxPG :: [Int] ->  (Int,Int)
minmaxPG l | mi:laux <- sort l
           , mx:_    <- reverse laux = (mi,mx)
           | otherwise = error "Argument too small"
```

10.7.2. Pattern-Synonyme

Eine weitere Erweiterung hinsichtlich Patterns sind sogenannte Pattern-Synonyme. Diese dienen dazu, neue Abkürzungen für verschachtelte komplizierte Pattern zu definieren.

Betrachte z.B. den folgenden Datentyp für arithmetische Ausdrücke, der binäre Operationen durch `Oper op e1 e2` darstellt, wobei `op` ein Operator des Aufzählungstypen `BinOp` ist:

```
data ArithEx = Oper Binop ArithEx ArithEx | Number Int  deriving(Show)
data Binop = AddOp | SubtractOp | TimesOp deriving(Show)
```

Erzeugt man z.B. den Ausdruck $(1 + 2) * (4 - 3)$ so schreibt man

```
example1 = Oper TimesOp
             (Oper AddOp (Number 1) (Number 2))
             (Oper SubtractOp (Number 4) (Number 3))
```

Um es schöner zu schreiben, kann man Kombinatoren definieren und verwenden:

```
myAd d x y = Oper AddOp x y
mySub  x y = Oper SubtractOp x y
myMult x y = Oper TimesOp x y

example2 = myMult
             (myAdd (Number 1) (Number 2))
             (mySub (Number 4) (Number 3))
```

Das Verwenden der Kombinatoren funktioniert zwar bei der Konstruktion, aber nicht beim Zerlegen der Datenstruktur. Als Beispiel betrachte, die Funktion zum Auswerten:

```
exec (Oper AddOp x y)      = (exec x) + (exec y)
exec (Oper TimesOp x y)    = (exec x) * (exec y)
exec (Oper SubtractOp x y) = (exec x) - (exec y)
exec (Number x)            = x
```

Hier helfen die Kombinatoren nicht. Mit der Erweiterung `PatternSynonyms` können wir neue Pattern (als neue Namen für Altbekanntes) definieren:

```
{-# LANGUAGE PatternSynonyms #-}
pattern MyAdd  x y  = Oper AddOp x y
pattern MySub  x y  = Oper SubtractOp x y
pattern MyMult x y  = Oper TimesOp x y
```

und damit `exec` schöner aufschreiben:

```
exec' (MyAdd x y)  = (exec' x) + (exec' y)
exec' (MyMult x y) = (exec' x) * (exec' y)
exec' (MySub x y)  = (exec' x) - (exec' y)
exec' (Number x)   = x
```

Durch obige Pattern-Definition können `MyAdd`, `MyMult` und `MySub` auch wie Konstruktoren verwendet werden:

```
example3 = MyMult
             (MyAdd (Number 1) (Number 2))
             (MySub (Number 4) (Number 3))
```

Sogenannte *unidirektionale Pattern* erlauben dies nicht: Hierbei muss anstelle des Gleichheitszeichen `=` ein Linkspfeil `<-` in der Definition verwendet werden.
Solche Pattern sind sinnvoll, für Pattern nicht invertibel sind. Z.B. können wir definieren

```
pattern Fuenftes x <- (_:_:_:_:x:_)

getFuenftes :: [a] -> Maybe a
getFuenftes (Fuenftes x) = Just x
getFuenftes _            = Nothing
```

Damit berechnet `getFuenftes "ABCDEFG"` als Ergebnis `'E'`, aber man kann offensichtlich aus `Fuenftes 'E'` keine eindeutige Liste erzeugen, die `'E'` als fünftes Element hat.

10.8. Überladene Strings

In Haskell kann der Ausdruck `"Burp"` kann nur Typ `String` haben, wobei `String` ein Typsynonym für `[Char]` ist. Der Typ `[Char]` ist leicht verständlich, aber sehr ineffizient. Bessere Alternativen bieten z.B. die Module `Data.ByteString` für 8-Bit-Arrays und `Data.Text` für Unicode-Strings. Ein Nachteil dabei ist, dass jeder String im Quellcode erst umständlich in den anderen Typ konvertiert werden:

```
pack   :: String -> Text
unpack :: Text   -> String
```

Außerdem tauchen zur Laufzeit wieder gewöhnliche Strings im Speicher auf. Literale wie `9` können jedoch verschiedene Typen haben, z.B. `Int` oder `Double`. Die Spracherweiterung `OverloadedStrings` erlaubt das Gleiche auch für String-Literale.

Voraussetzung hierfür ist, dass der gewünschte Typ Instanz der Klasse `IsString` aus Modul `Data.String` ist:

```
class IsString a where
   fromString :: String -> a
```

String-Literale haben dann den Typ `(IsString a) => a`, d.h. Konvertierung erfolgt implizit. Dies kann auch im GHCi beobachtet werden:

```
Prelude> :t "Hallo"
"Hallo" :: [Char]
Prelude> :set -XOverloadedStrings
Prelude> :t "Hallo"
"Hallo" :: Data.String.IsString p => p
```

Nachteile bei der Verwendung sind, dass manchmal Typannotationen notwendig sind, falls mehrere Instanzen der Klasse `IsString` in Frage kommen. Zudem können Fehlermeldungen komplizierter aussehen. Ein künstliches (nicht sinnvolles) Beispiel ist:

```
ghci -XOverloadedStrings
*> :module + Data.String
*> instance IsString Bool where fromString "True" = True;
                                fromString   _    = False
*> instance IsString Int  where fromString s = length s
*> fromString "True"
"True"
*> fromString "True" :: Bool
True
*> fromString "True" :: Int
4
```

Der Typparameter taucht nur im Ergebnis auf: `fromString :: String -> a`, d.h. der benötigte Ergebnistyp bestimmt den verwendeten Code (analog bei `read` aus Klasse `Read`)

Ein größeres Beispiel mit eigenen Strings zeigt der folgende Code.

```
{-# LANGUAGE OverloadedStrings #-}
import Data.String
newtype MyString = MyString String   deriving (Eq)
```

```
instance IsString MyString where
    fromString = MyString
instance Show MyString where
    show (MyString s) = "<" ++ s ++ ">"

greet :: MyString -> MyString
greet (MyString "hello") = MyString "hallo"
greet "fool" = "welt"     -- 2x automatische Konvertierung,
greet other  = other      -- also auch im Pattern-Match!

main = do
  print $ greet "hello"   -- automatische Konvertierung!
  print $ greet "fool"    -- automatische Konvertierung!
```

Die Ausgabe von `main` ist:

```
main
<hallo>
<welt>
```

10.8.1. Quellen

Dieses Kapitel orientiert sich am Material aus (Jost, 2019) und verwendet Beispiele aus (Kiselyov et al., 2010; Wadler, 1987; Yorgey et al., 2012) sowie der GHC-Dokumentation und dem Haskell-Wiki. Als weiterführende Literatur zur Programmierung auf Typebene sei das Buch (Maguire, 2019) empfohlen.

11

Anwendungsprogrammierung

In diesem Kapitel betrachten wir zwei wichtige Zweige der Anwendungsprogrammierung, nämlich in Abschnitt 11.1 die Webprogrammierung und in Abschnitt 11.2 die GUI-Programmierung mit der Sprache Haskell. Im Abschnitt 11.1.6 betrachten wir außerdem die Verwendung von Datenbanken mit Haskell, was auch für andere Anwendungen (unabhängig von Web- und GUI-Programmierung) von Bedeutung sein kann.

11.1. Webapplikationen mit Yesod

Eine Übersicht über verschiedene Webframe-Frameworks für Haskell findet man im Haskell-Wiki unter `https://wiki.haskell.org/Web/Frameworks`. Wir werden das Framework Yesod betrachten, welches z.B. für die Entwicklung von uni2work (`https://uni2work.ifi.lmu.de`) verwendet wird. Wir werden dabei nicht alle Details des sehr umfangreichen Frameworks erläutern, sondern einen groben Überblick geben. Für Details sei z.B. auf (Snoyman, 2012) verwiesen.

Zur Entwicklung von Webapplikationen bietet sich die Verwendung eines Frameworks an, welches alle grundlegenden Funktionalitäten durch Bibliotheken und vordefinierte Programmgerüste bereitgestellt. Wesentliche Eigenschaften des Yesod-Frameworks sind: Typ-sichere URLs, Templating und domänenspezifische Sprachen (DSLs), d.h. viele modulare Einzelteile, eine integrierte Datenbankanbindung über die Pakete `persist` und `conduit`, REST-Architektur, was für *REpresentational State Transfer* steht und zustandslos ist, d.h. die gleiche URL bezeichnet stets die gleiche Webseite.

Die Installation und das Starten sind unter `https://www.yesodweb.com/page/quickstart` beschrieben, wir fassen dies hier zusammen: Mit `stack new` *Projektname TemplateName* kann ein vorgefertigtes Projekt angelegt werden (dies wird als sogenanntes Scaffolding (d.h. Grundgerüst) bezeichnet). Mit `stack templates` kann man die verfügbaren Templates anzeigen lassen. Für ein `yesod`-Projekt gibt es verschiedene Templates, wir verwenden `yesod-sqlite`:

```
> stack new mein-projekt yesodweb/sqlite
```

Anschließend kann man in das Verzeichnis wechseln und dann `yesod-bin` installieren, um das `yesod`-Programm selbst zu installieren. Im Anschluss daran kann das Projekt compiliert werden und schließlich kann man `yesod` im Entwicklungsmodus starten. Zusammengefasst muss man die folgenden Befehle in einem Terminal ausführen:

```
> cd mein-projekt
> stack build yesod-bin --install-ghc
> stack build
> stack exec -- yesod devel
```

Danach sollte ein lokaler Development-Webserver laufen, so dass man eine Webseite über die URL `http://localhost:3000/` abrufen kann.

Die vorgefertigten Templates nutzen viele nicht-zwingende, aber sinnvolle Voreinstellungen, z.B. sind viele Dinge in mehreren Dateien getrennt, was nicht unbedingt notwendig ist. Wir werden im Weiteren nicht genau darauf eingehen, wo das Scaffolding, welche Informationen anlegt, sondern verweisen hierfür auf `https://www.yesodweb.com/book/scaffolding-and-the-site-template`. Die Development-Version des Webservers (ausgeführt durch `yesod devel`) überwacht Quellcode-Änderungen, sodass bei Änderungen der Development Webserver neu kompiliert und gestartet wird. Compiler-Optionen und Paket-Abhängigkeiten können in `package.yaml` eingestellt werden.

Ein einfaches Beispiel ist das folgende „Hello-Yesod“-Programm:

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, TemplateHaskell, QuasiQuotes #-}
module Main where
import Yesod

data MyApp = MkApp
instance Yesod MyApp

mkYesod "MyApp" [parseRoutes|
  / HomeR GET
  |]

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  setTitle "HelloWorld"
  toWidget [whamlet|
      <h2>Hello Yesod!
      Some text that
      is <i>displayed</i> here.
    |]

main :: IO ()
main = warp 3000 MkApp
```

Dieses kann direkt compiliert werden (nachdem das `Yesod`-Paket z.B. mit `stack install yesod` installiert wurde) und gestartet werden. Es zeigt eine einfache Webseite unter `http://localhost:3000` an. Das Programm zeigt, dass die vier Spracherweiterungen `OverloadedStrings`, `TypeFamilies`, `TemplateHaskell`, `QuasiQuotes` verwendet werden. Template-Haskell und QuasiQuotes werden für das eingebettete Verwenden von HTML, CSS und JavaScript eingesetzt, aber auch zum Parsen der Routen.

11.1.1. Shakespearean Templates

Sogenannte *Shakespearean Templates* erlauben es mit den Mechanismen von Template Haskell HTML, CSS und JavaScript in Yesod zu manipulieren Dazu werden spezialisierte Quasi-Quoter `[|...|]` bereitgestellt:

Sprache	Quasiquoter	Dateiendung
HTML	`hamlet`	.hamlet
CSS	`cassius`	.cassius
CSS	`lucius`	.lucius
JavaScript	`julius`	.julius
I18n interpolierter Text	`stext/ltext`	

Es gibt auch spezialisierte Varianten `shamlet`, `whamlet`, ... Diese unterscheiden sich meist nur durch den Ergebnistyp. Das Beispiel

```
[hamlet|
  <h2>Hello Yesod!
  Some text that is <i>displayed</i> here.
|]
```

erzeugt eine Haskell-interne Darstellung von HTML, aus einer vereinfachten HTML-Syntax. Das Scaffolding lädt die Templates aus Dateien (anstatt sie mit dem QuasiQuoter in den Quelltext einzubinden), dafür wird die Meta-Funktion `widgetFile` zum Laden solcher Beschreibungen aus Dateien verwendet, indem man diese durch Splicen mit dem Dateinnamen als `$(widgetFile` *Filename*`)` aufruft. Entsprechend dem Splicen für Haskell-Code gibt es in den Templates einen Mechanismus zur *Interpolation*:

- `#{ }` für die Interpolation von Variablen im Gültigkeitsbereich (dies geschieht „escaped“, so dass bei beliebigen Typen (die Instanz von `ToHtml` sein müssen) die Funktion `toHtml` automatisch angewendet wird). Man kann innerhalb dieser Variablen-Interpolation auch Haskell-Funktionen anwenden, Strings und Zahlen verwenden und auch qualifizierte Modulnamen verwenden.
- `@{ }` für die typsichere URL-Interpolation, z.B. `@{HomeR}`.
- `^{ }` für die Template-Einbettung, fügt ein Template gleichen Typs ein.
- `_{ }` für internationalisierten Text, fügt eine Übersetzung ein.

Damit können Templates dynamisch erzeugt werden:

```
[whamlet|
  Value of fib22 is #{show (fib 22)}
|]
```

Der Ergebnistyp einer Interpolation muss immer eine Instanz der Typklasse `ToHtml` sein. Die Interpolation von URLs funktioniert ähnlich:

```
let foo = show (fib 22) in
      [whamlet|
        Value of foo is #{foo}
        Return to <a href=@{Home}>Homepage
        .
      |]
```

Dabei muss `Home` ein Konstruktor/Wert des Datentyps für das *Routing* dieser Webanwendung sein, welches wir später genauer betrachten werden.

11.1.1.1. Hamlet

Hamlet funktioniert wie gewöhnliches HTML mit Interpolation, wobei jedoch zusätzlich gilt: Schließende HTML-Tags werden durch Einrücken ersetzt, z.B. wird

```
<p>Some paragraph.
  <ul><li>Item 1</li>
      <li>Item 2</li>
  </ul></p>
<p>Next paragraph.</p>
```

in Hamlet so geschrieben:

```
<p>Some paragraph.
  <ul>
    <li>Item 1
    <li>Item 2
<p>Some paragraph.
```

Der QuasiQuoter generiert den oberen Code aus dem unteren. Es sind jedoch kurze geschlossene inline-Tags zulässig:

```
<p>Some <i>italic</i> paragraph.
```

Ein Tag am Zeilenanfang wird immer nur durch Einrückung geschlossen. Leerzeichen vor und nach Tags und an Zeilenanfang und -ende werden entfernt. An diesen Positionen explizit gewollte Leerzeichen müssen mit `#` und `\` markiert werden:

```
<p>
  Some #
  <i>italic
  \ paragraph.
```

HTML-Attribute funktionieren wie in HTML, d.h. Gleichheitszeichen, Wert und Anführungszeichen sind meist optional. Abkürzungen für IDs, Klassen und Konditionale sind erlaubt:

```
<p #paragraphid .class1 .class2>
<p :someBool:style="color:red">
<input type=checkbox :isChecked:checked>
```

Ein Attribut-Paar `attr::(Text,Text)` oder mehrere `attrs::[(Text,Text)]` können auch direkt eingebunden werden:

```
 <p *{attrs}>
```

Hamlet erlaubt auch logische Konstrukte: Ein Konditional wird geschrieben als

```
$if isAdmin
  <p>Hallo mein Administrator!
$elseif isLoggedIn
  <p>Du bist nicht mein Administrator.
$else
  <p>Wer bist Du?
```

Auch einfache Schleifen sind möglich:

```
$if null people
  <p>Niemand registriert.
$else
  <ul>
    $forall person <- people
      <li>#{show person}
```

Ein Maybe-Konstrukt gibt es auch:

```
$maybe name <- maybeName
  <p>Dein Name ist #{name}
$nothing
  <p>Ich kenne Dich nicht.
```

Dieses erlaubt Pattern-Matching und unvollständige Fälle:

```
$maybe Person vorname nachname <- maybePerson
  <p> Dein Name ist #{vorname} #{nachname}
```

Auch volles Pattern-Matching ist mit dem `$case`-Konstrukt möglich:

```
$case foo
  $of Left bar
      <p>Dies war links: #{bar}
  $of Right baz
      <p>Dies war rechts: #{baz}
```

Das Konstrukt `$with` ist ähnlich zum `let` und dient für lokale Definitionen, z.B.

```
$with foo <- myfun argument $ otherfun more args
  <p>
    Einmal ausgewertetes foo hier #{foo}
    und da #{foo} und dort #{foo} verwendet.
```

Es gibt weitere Abkürzungen für Standard-Komponenten, z.B. steht `$doctype 5` für `<!DOCTYPE html>`.

11.1.1.2. Lucius

Lucius akzeptiert ganz normales CSS. Zusätzlich ist die Interpolation für Variablen `#{}`, URLs `@{}` und Mixins `^{}` erlaubt, CSS-Blöcke dürfen verschachtelt werden und es können lokale Variablen deklariert werden. Ein Beispiel ist:

```
article code { background-color: grey; }
article p { text-indent: 2em; }
article a { text-decoration: none; }
```

Dieses kann bei Bedarf umgeschrieben werden zu

```
@backgroundcolor: grey;
article {
  code { background-color: #{backgroundcolor}; }
  p { text-indent: 2em; }
  a { text-decoration: none; } }
```

11.1.1.3. Cassius

Cassius ist eine Alternative zu Lucius, die Whitespace-sensitiv ist. Cassius wird zu Lucius übersetzt. Klammern und Semikolons müssen dabei immer durch Einrücken ersetzt werden, z.B.:

```
#banner
  border: 1px solid #{bannerColor}
  background-image: url(@{BannerImageR})
```

11.1.1.4. Julius

Julius akzeptiert gewöhnliches JavaScript, jedoch erweitert um

- `#{}` für die Variablen-Interpolation,
- `@{}` für die URL Interpolation und
- `^{}` für das Einbetten von Templates anderer JavaScript-Templates

Sonst ändert sich nichts, auch nicht an Einrückungen.

11.1.1.5. Widgets

Widgets fassen einzelne Templates von verschiedenen Shakespeare-Sprachen zu einer Einheit zusammen:

```
getRootR = defaultLayout $ do
  setTitle "My Page Title"
  toWidget [lucius| h1 { color: green; } |]
  addScriptRemote "https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js"
  toWidget [julius|
    $(function() {
      $("h1").click(function(){ alert("Clicked the heading!"); });
    });
  |]
  toWidgetHead [hamlet| <meta name=keywords content="keywords">|]
  toWidget [hamlet| <h1>Here's one way for including content |]
  [whamlet| <h2>Here's another |]
  toWidgetBody [julius| alert("This is included in the body"); |]
```

Die Widget-Monade erlaubt das Kombinieren dieser Bausteine, alles wird automatisch dahin sortiert, wo es hingehört.

Template-Embedding erlaubt normalerweise nur die Einbettung aus der gleichen Template-Sprache. Dagegen erlauben `whamlet` bzw. `.whamlet`-Dateien die Einbettung von Widgets in Hamlet:

```
page = [whamlet|
    <p>This is my page. I hope you enjoyed it.
    ^{footer}
  |]
```

```
footer = do
    toWidget [lucius| footer { font-weight: bold;
                               text-align: center } |]
    toWidget [hamlet|
        <footer>
          <p>That's all folks!
      |]
```

Bei der Kombination von Widgets könnten Namenskonflikte auftreten. Dies kann durch dynamische IDs verhindert werden:

```
getHomeR :: Handler Html
getHomeR = defaultLayout $
  do headerClass <- newIdent
     toWidget [hamlet|<h1 .#{headerClass} >My Header|]
     toWidget [lucius| .#{headerClass} { color: green; } |]
```

Die Funktion `newIdent` erlaubt die Erzeugung von frischen IDs.

11.1.2. Foundation-Typ

Einstellungen einer Yesod-Applikation werden mit Hilfe eines Foundation-Datentyps vorgenommen. Der Wert muss dazu nichts direkt speichern, Einstellungen werden über die Instanzdeklaration zu `Yesod`-Typklassen vorgenommen.

```
  data MyWebApp = MkWebApp       -- Foundation Type
  instance Yesod MyWebApp        -- Defaults
```

Diese Klasse fasst alle möglichen Einstellungen zusammen, wie das Rendern und Parsen von URLs, die Funktion `defaultLayout` zum Festlegen des Layouts der dynamisch generierten Webseiten, die Authentifizierung, die Sitzungsdauer, die Behandlung von Cookies, die Fehlerbehandlung und Aussehen der Fehlerseiten, externe CSS, Skripte und statische Dateien, usw.
Für alle Einstellungen gibt es vordefinierte Standards, welche bei Bedarf überschrieben werden können. Z.B. mit selbstdefinierter Fehlerbehandlung:

```
instance Yesod MyWebApp where
    errorHandler NotFound = myNotFoundHandler
    errorHandler other    = defaultErrorHandler other
```

Die Default-Definition `errorHandler = defaultErrorHandler` wird dadurch für den Fall `NotFound` geändert, so dass ein selbstgeschriebener Handler `myNotFoundHandler` verwendet wird.

Der Foundation-Typ kann auch Parameter tragen. Innerhalb der `Handler`-Monade können diese mit `getYesod` wieder ausgelesen werden. Das folgende Beispiel verwendet eine Zahl und eine `TVar` für diesen Parameter:

```
{-# LANGUAGE OverloadedStrings, TypeFamilies, TemplateHaskell, QuasiQuotes #-}
module Main where
import Yesod
```

```
[parseRoutes|
  /               RootR     GET
  /blog/help      BlogHelpR GET
  /blog/#Int      BlogPostR GET POST
  /wiki/*WikiPfad WikiR
  /static         StaticR   Static getStatic
|]
```

Abbildung 11.1.: Beispiel für eine Routen-Spezifizierung in Yesod

```
import Control.Concurrent.STM

data MyWebApp = MkWebApp { intPar :: Int
                         , varPar :: TVar String }
instance Yesod MyWebApp

mkYesod "MyWebApp" [parseRoutes|
  / HomeR GET
  |]

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  master <- getYesod                          -- master :: MyWebApp
  let myint = intPar  master                  -- myint  :: Int
  mystr <- liftIO $ readTVarIO $ varPar master -- mystr  :: String
  toWidget [whamlet|
      #{mystr}
    |]

main :: IO ()
main = do v <- newTVarIO "Hello"
          warp 3000 MkWebApp {intPar = 0, varPar = v}
```

Damit sammelt man alle statischen Parameter an einer Stelle. Beachte, dass man eine `MVar` bzw. `TVar` verwenden sollte, um einen gemeinsamen Zustand für Handler zu speichern (und keine `IORef`, da man nebenläufige Zugriffe hat).

Yesod nutzt eine domänenspezifische Sprache zur Spezifizierung von Routen und Handling. In Abbildung 11.1 wird ein Beispiel hierfür gezeigt. Diese definiert die gesamte Sitemap der Webapplikation (außer Unterseiten wie im obigen Beispiel `static`). Die eingebettete Sprache wird innerhalb des Quasiquoters `parseRoutes` angegeben, oder in einer separaten Datei. Beim Splicen können View-Patterns entstehen, weshalb die Erweiterung `ViewPatterns` aktiviert sein muss. Die Syntax jedes Eintrags besteht aus drei Teilen. Zuerst wird der Pfad angegeben. Es gibt drei Arten von Pfaden:

1. Statische Pfade, wie z.B. `/blog/help`
2. Dynamische Pfade *enthalten* (mehrere) `/#<Typ>` Fragmente, wobei `<Typ>`-Instanzen für `PathPiece,Eq,Read,Show` haben müssen.
3. Dynamische Multipfade *enden* mit `/*<Typ>`, wobei `<Typ>` Instanzen für `PathMultiPiece,Eq,Read,Show` haben muss

Die Klassen `PathPiece` und `PathMultiPiece` aus dem Modul `Yesod.Dispatch` legen lediglich Parser fest:

```
class PathPiece s where
  fromPathPiece :: Text -> Maybe s
  toPathPiece   :: s -> Text

class PathMultiPiece s where
  fromPathMultiPiece :: [Text] -> Maybe s
  toPathMultiPiece   :: s -> [Text]
```

`PathPiece`-Instanzen für `Int`, `Integer`, `String`, `Text` und `PathMultiPiece`-Instanzen für `[String]` und `[Text]` sind vordefiniert. Pfade werden automatisch standardisiert, gereinigt und zerlegt. Durch Überschreiben von Default-Funktionen der `Yesod`-Typklasse kann man dies aber auch anpassen:

- `joinPath` fügt Pfadteile wieder zusammen
- `cleanPath` kümmert sich um doppelte `/`, usw.

Yesod kann nicht inferieren, dass `/blog/help` und `/blog/#Int` nicht überlappen. Überlappende Routen wie etwa `/blog/#Int` und `/blog/#Text` erzeugen eine Fehlermeldung. Dies kann mit einem vorgestellten Ausrufezeichen verhindert werden, z.B. `!/blog/#Text`. Die von oben-nach-unten zuerst geparste Route wird eingeschlagen.

Der zweite Teil der Routenbeschreibung definiert den *Konstruktor* für die typsichere URL. Die Deklaration für den Datentyp `Route <FoundationType>` wird erzeugt und kann direkt in der URL-Interpolation `@{ }` verwendet werden (z.B. bei dynamischen Pfaden: `@{BlogPostR 7}`).

Der dritte Teil einer Route spezifiziert entweder:

1. Erlaubte HTTP-Anfragen `GET,PUT,POST,DELETE,...`
2. Keine Angabe, d.h. alle HTTP-Anfragen erlaubt (diese werden dann über einen gemeinsamen Handler abgewickelt)
3. Den Foundation-Typ einer Yesod-Subsite, welche die Anfrage beantwortet, und eine Funktion, welche den Wert des aktuellen Foundation-Typs in den Wert des Foundation-Typs der Subsite umrechnen kann.

Für alle Anfragen muss ein Handler definiert werden. Diese Funktion behandelt die Route. Der Name der Funktion muss `<AnfrageTyp> ++ <Resource>` sein, z.B.

```
getRootR      :: Handler Html
getBlogHelpR  :: Handler Html
getBlogPostR  :: Int -> Handler Html
postBlogPostR :: Int -> Handler Html
handleWikiR   :: [WikiPfad] -> Handler Html
```

oder `handle ++ <Resource>`, falls alle HTTP-Anfragen erlaubt.

Im Beispiel aus Abbildung 11.1 wird die Route `/static` durch eine Unterseite geleitet, welche darunterliegende Routen und Handler selbst definiert. Dabei ist `Static` der Foundation-Typ der Unterseite und die angegebene Funktion `getStatic` muss Werte des Typs `Static` aus dem aktuellem Foundation-Typ generieren können. Bei Scaffolding mit `yesod init` wird diese Unterseite automatisch für statische Ressourcen definiert, um besseres Caching zu ermöglichen.

11.1.3. Handling

Handler-Funktionen leben in der Handler-Monade und sind meist vom Typ `Handler Html`. Die Definition des Typs ist:

```
type Handler a = HandlerFor App (IO a)
data HandlerFor site a = ...
```

Daher ist `Handler` ein Typsynonym für einen Datentyp mit Foundation-Typ (`site`) und dem Rückgabewert der Monade (`a`). Anstelle von `Html` kann ein Handler auch Werte anderer Typen liefern, zum Beispiel eine Grafikdatei, CSS, JSON, etc. Der Inhaltstyp muss jedoch bekannt sein, d.h. Handler müssen als Ergebnistyp eine Instanz von `ToTypedContent` liefern. Es ist auch möglich, unterschiedliche Repräsentationen je nach MIME-typ über eine URL abzuwickeln:

```
mkYesod "App" [parseRoutes|
  /person PersonR GET
|]

getPersonR :: Handler TypedContent
getPersonR = selectRep $ do
  provideRep $ return [shamlet| <p>Name #{name}, Age #{age} |]
  provideRep $ return $ object [ "name" .= name, "age" .= age ]
 where
  name = "Horst" :: Text
  age  = 40 :: Int
```

Je nach Anfrage wird HTML ausgeliefert oder JSON. Wichtige Funktionen der `Handler`-Monade sind:

- `getYesod`, um den Wert des Foundation-Typs, z.B. Parameter auszulesen.
- `getUrlRender`, um Renderer für Werte des `Route`-Typs zu erhalten.
- `getRequest`, um die Anfrage im Roh-Format zu erhalten.
- `liftIO`, zum Ausführen von IO-Aktionen.
- `sendFile`, um eine Datei zu versenden.
- `addHeader`, um den Antwort-Header festzulegen.
- `redirect` zur Umleitung zu einer anderen Ressource.
- `notFound`, `permissionDenied` für explizite Fehlermeldungen.
- `setCookie`, `lookupCookie`, um Cookies zu bearbeiten.

Der Typ `Handler` ist auch eine Instanz von `MonadLogger`, sodass er für Logging verwendet werden kann. Erzeugen von Log-Messages innerhalb von Templates geht mit `$logError`, `$logWarn`, `$logInfo` und `$logDebug`.

11.1.4. Webformulare

Webformulare erlauben dem Benutzer, Daten zu übermitteln. Dies erfordert die Darstellung der Formulare in HTML, die Zuordnung der übermittelten Daten, die Konvertierung von UTF8-Strings in Haskell-Datentypen, Prüfungen, ob Daten im erlaubten Bereich sind, und JavaScript zu Daten-Prüfung und zur Bearbeitung auf dem Client. Schließlich ist ea manchmal auch notwendig verschiedene Formular(-Teile) zu kombinieren. Das Paket `yesod-form` stellt Formulare bereit, wobei es drei Varianten gibt, Formulare zu erstellen:

- Ein applikatives Interface, welches einfach zu programmieren ist, aber die Kontrolle über das Aussehen ist eingeschränkt.
- Ein monadisches Interface, das flexibel gestaltbare Formulare erlaubt, wobei die Verwendung jedoch etwas komplizierter ist.
- Ein Input-Interface, welches keine eigene HTML-Darstellung verwendet, sondern hauptsächlich zur Verwendung mit bestehenden Formularen gedacht ist.

Es ist möglich, applikative Formulare automatisch in monadische umzuwandeln, und manchmal auch umgekehrt (siehe `aformToForm` und `formToAForm`). In Yesod wird die Konvention verwendet, dass der Präfix der Funktionen für Formulare je nach Variante `a`, `m` oder `i` ist. Z.B. erhält man ein Pflichtfeld in einem applikativen Formular mit `areq` und ein optionales Feld in einem monadischen Formular mit `mopt`.

11.1.4.1. Applikative Formularfelder

Der folgende Code definiert zunächst einen Datentyp `Car` für Autos und im Anschluss wird ein applikatives Formular zur Eingabe einzelner Autos erzeugt:

```
data Car = Car { carModel :: Text, carYear  :: Int,  carColor :: Maybe Text}
  deriving Show
carAForm :: AForm Handler Car
carAForm = Car <$> areq textField "Model" Nothing
               <*> areq intField  "Year"  (Just 1994)
               <*> aopt textField "Color" Nothing
```

Dabei wird `areq` für erforderliche Felder und `aopt` für optionale Felder eines `Maybe`-Typen verwendet. Das erste Argument definiert den Typ des Eingabefeldes und erklärt damit dessen Parser, wobei es für viele Typen vordefiniert Felder gibt. Das zweite Argument ist ein Bezeichner, Tooltip, id- und name-Attribut, welches durch ein Objekt des Typs `FieldSettings` gegeben ist (durch `OverloadedStrings` kann wie oben auch nur ein String verwendet werden, sodass der String nur den Bezeichner setzt). Das dritte Argument ist ein optionaler Default-Wert (vom Typ `Maybe`).
Das Formular vom Typ `AForm` kann in ein Formular vom Typ `MForm` mit `renderTable`, `renderDivs`, oder `renderBootstrap` umgewandelt werden:

```
carForm :: Html -> MForm Handler (FormResult Car, Widget)
carForm = renderTable carAForm
```

Anschließend kann es wie folgt verwendet werden:

```
getCarR :: Handler Html
getCarR = do (widget, enctype) <- generateFormPost carForm
             defaultLayout [whamlet|
    <h2>Form Demo
    <form method=post action=@{CarR} enctype=#{enctype}>
      ^{widget}
      <button>Submit
  |]
```

Mit `generateFormGet` oder `generateFormPost` wird das Formular erzeugt (jenachdem, ob die GET- oder die POST-Methode verwendet wird, muss die entsprechende Funktion verwendet werden).

Der `form`-Tag und Knopf zum Absenden ist noch nicht enthalten, damit Formulare kombiniert werden können, und muss daher erzeugt werden.

Zum Auswerten eines Formulars stehen `runFormGet` bzw. `runFormPost` zur Verfügung, so dass wir z.B. definieren können:

```
postCarR :: Handler Html
postCarR = do
 ((result,widget), enctype) <- runFormPost carForm
 case result of
   FormSuccess car -> defaultLayout [whamlet|
          <h2>Car received:
          <p>#{show car}  |]
   FormMissing -> addMessage "i" "No data" >> redirect CarR
   FormFailure msgs  -> defaultLayout [whamlet|
          <h2>Fehler:
          <ul>
           $forall msg <- msgs
             <li>#{msg}
          <form method=post action=@{CarR} enctype=#{enctype}>
            ^{widget}
            <button>Submit
        |]
```

Es gibt drei mögliche Ergebnisse für `result`: Das Ergebnis `FormSuccess a` bedeutet Erfolg. In diesem Fall erzeugt obiger Code eine HTML-Seite, die das Auto anzeigt. Das Ergebnis `FormFailure [Text]` bedeutet, dass das Parsen fehlgeschlagen ist (obiger Code zeigt die Fehler an und danach das Formular). Das Ergebnis `FormMissing` bedeutet, dass keine Daten vorhanden sind.

Beim Erstellen der Formulare ist vieles weitere möglich, wie das Prüfen der Daten auf Korrektheit usw. Wir verweisen auf die Yesod-Dokumentation für diese Details. Wir zeigen noch einige gebräuchliche Formular-Komponenten.

11.1.4.1.1. Auswahllisten Die Funktion `selectFieldList` nimmt eine Liste von (`Text`,Wert)-Paaren und kreiert eine Auswahlliste. Z.B.:

```
data Car   = Car {carModel::Text, carYear::Int, carColor::Maybe Color} deriving Show
data Color = Red | Blue | Gray | Black deriving (Show, Eq, Bounded, Enum)

carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt (selectFieldList colors) "Color" Nothing
  where
    colors :: [(Text, Color)]
    colors = [("Rot", Red), ("Blau", Blue), ("Grau", Gray), ("Schwarz", Black)]
```

11.1.4.1.2. Auswahlknöpfe Die Funktion `radioFieldList` nimmt eine Liste von (`Text`,Wert)-Paaren, genau wie `selectFieldList` auch, aber erzeugt Auswahlknöpfe. Z.B.

```
carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt (radioFieldList colors) "Color" Nothing
  where
    colors :: [(Text, Color)]
    colors = [("Rot", Red), ("Blau", Blue), ("Grau", Gray), ("Schwarz", Black)]
```

11.1.4.2. Input-Formulare ohne Layout

Input-Formulare erlauben direkt die Verwendung von HTML-Formularen, z.B.

```
data Person = Person { personName :: Text, personAge  :: Int } deriving Show

getHomeR :: Handler Html
getHomeR = defaultLayout
    [whamlet|
        <form action=@{InputR}>
            <p>
                My name is
                <input type=text name=name>
                and I am
                <input type=text name=age>
                years old.
                <input type=submit value="Introduce myself">
    |]

getInputR :: Handler Html
getInputR = do
    person <- runInputGet $ Person
                <$> ireq textField "name"
                <*> ireq intField "age"
    defaultLayout [whamlet|<p>#{show person}|]
```

Ein großer Nachteil ist, dass die Übereinstimmung der Name-Tags (also z.B. `age` in `<input type=text name=age>` und `ireq intField "age"` vom Programmierer gewährleistet werden muss, da es keinen logischen Zusammenhang mehr gibt. Die Funktionen `ireq` und `iopt` haben nur noch zwei Argumente, den Feld-Typ und den Feld-Namen. Wenn die übermittelten Daten nicht passen, erfolgt eine Umleitung auf eine „Invalid Arguments“-Fehlerseite.

11.1.4.3. Monadische Formulare

Monadische Formulare erlauben ebenfalls ein eigenes Layout, aber kümmern sich für uns um einzigartige Name-Tags, usw. Die Funktionen `mreq` und `mopt` funktionieren analog zu `areq` und `aopt`, aber die Namen der Eingabefelder werden ignoriert, denn das Layout wird explizit angegeben.

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show
```

```
personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                          width: 3em;}
                                |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHomeR = do ((res, widget), enctype) <- runFormGet personForm
              defaultLayout [whamlet|
                  <p>Result: #{show res}
                  <form enctype=#{enctype}>
                      ^{widget} |]
```

Felder werden durch die zwei Teile `FormResult` und `FormView` beschrieben. Das `extra`-Argument muss irgendwo ins Formular eingebaut werden. Bei `GET`-Formularen signalisiert es, dass das Formular abgesendet wurde. Bei `POST`-Formularen dient es zur Verhinderung von Cross-Site-Request-Forgery-Angriffen. Alle Felder des Formulars werden mit den monadischen Funktionen `mreq` und `mopt` beschrieben, wie in

```
do (nameRes, nameView) <- mreq textField "not used" Nothing
   (ageRes , ageView)  <- mreq intField  "not used" Nothing
```

Für jedes Feld erhalten wir zwei Rückgabewerte, nämlich das Ergebnis des Feldes vom Typ `FormResult a`, um später das Gesamtergebnis zusammen zu bauen, wie in `let personRes = Person <$> nameRes <*> ageRes` und die Anzeige des Feldes vom Typ `FieldView`, zum Einbau in Widgets. Mit `fvInput` wird ein Input-Feld dafür erzeugt, wie z.B. in `^{fvInput ageView}`. Mit `fvId` kann auf die Id des Feldes zugegriffen werden, wie z.B. in `##{fvId ageView}`. Genauer ist ein Wert des Typs `FieldView` ein Record mit den Feldern `fvLabel`, `fvTooltip`, `fvId`, `fvInput`, `fvErrors` und `fvRequired`. Dieses wird primär aus den Vorschlägen des zweiten Arguments von `mreq` oder `mopt` erzeugt, welches vom Typ `FieldSettings` sein muss.

11.1.5. Sessions

HTTP kennt wie Haskell keinen Zustand. Webapplikationen benötigen aber einen Zustand, z.B. für das Login, den Warenkorb usw. Eine Lösung dazu sind Sitzungen (Sessions), d.h. eine kleine Menge an Daten (z.B. eine Sitzung-ID), welche der Browser mit jeder Anfrage an den Webserver übermittelt. Benutzerdaten mit Sitzungen zu speichern, skaliert gut mit mehreren Servern, da jeder Request in sich abgeschlossen ist und keine zentrale Koordination/Datenbank benötigt wird. Dabei sollten die Sitzungsdaten möglichst klein sein. Yesod wendet per Default automatisch Verschlüsselung und Signatur von Sitzungsdaten an, um Manipulationen zu verhindern. Sitzung verfallen automatisch nach 2 Stunden. Dies wird alles in der Yesod-Instanz des Foundation-Typs eingestellt:

```
instance Yesod App where
  makeSessionBackend _ = Just <$>
      defaultClientSessionBackend minutes file
    where minutes = 2 * 60
          file    = "client-session-key.aes"
  -- Sitzungen komplett deaktivieren:
  -- makeSessionBackend _ = return Nothing
```

Der Schlüssel wird in einer separaten Datei gespeichert und das Ablaufdatum der Sitzung wird mit jedem Request erneuert. Yesod ignoriert abgelaufene Sitzungen automatisch. Eine Sitzung ist eine *ungetypte* Map:

```
type SessionMap = Map Text ByteString
```

Wesentliche Operationen dafür sind

- `getSession :: MonadHandler m => m SessionMap` liefert die gesamte Sitzungs-Map.
- `lookupSessionBS :: MonadHandler m => Text -> m (Maybe ByteString)` schlägt Schlüssel in der Map nach.
- `lookupSession :: MonadHandler m => Text -> m (Maybe Text)` schlägt ebenfalls Schlüssel nach.
- `setSession :: MonadHandler m => Text -> Text -> m ()` zum Setzen eines Schlüssel-Wert-Paars.
- `deleteSession :: MonadHandler m => Text -> m ()` löscht einen Schlüssel.
- `clearSession :: MonadHandler m => m ()` löscht die gesamte Sitzungs-Map.

Das folgende Beispiel erlaubt das Hinzufügen und Löschen von einzelnen Schlüssel/Wert-Paaren in der Session über ein Input-Formular. Zudem wird die Session noch angezeigt:

```
getHomeR :: Handler Html
getHomeR = do
    sess <- getSession
    defaultLayout [whamlet|
                       <form method=post>
                           <input type=text name=key>
                           <input type=text name=val>
                           <input type=submit>
                       <h1>#{show sess}
                  |]

postHomeR :: Handler ()
postHomeR = do
    (key, mval) <- runInputPost $ (,) <$> ireq textField "key"
                                      <*> iopt textField "val"
    case mval of Nothing  -> deleteSession key
                 Just val -> setSession key val
    liftIO $ print (key, mval) --debug to konsole
    redirect HomeR
```

11.1.5.1. Messages

Ein Problem bei der Verwendung von Post/Redirect/Get ist, dass der Benutzer nach erfolgreichem Ausfüllen eines Formulars umgeleitet wird, aber man ihn gleichzeitig informieren möchte, dass die Daten korrekt empfangen wurden. Eine Lösung besteht darin, dass jede Seite prüft, ob es ein spezielles Sitzungs-Feld für Nachrichten gibt. Yesod bietet eine eigene Schnittstelle dafür:

- `addMessage :: MonadHandler m => Text -> Html -> m ()` setzt eine Nachricht in der Sitzung.
- `getMessages :: MonadHandler m => m [(Text, Html)]` liest Nachrichten aus und löscht diese.

Das vorgefertigte `defaultLayout` nutzt `getMessages`, um etwaige Botschaften anzuzeigen. Ein Beispiel zur Demonstration ist:

```
getA = do page <- defaultLayout $ do
                  [whamlet|
                            You are at A
                  |]
          addMessage "info" "Previous: A"
          return page
getB = do msgs <- getMessages
          page <- defaultLayout $ do
                  [whamlet|
                     <p>You are at B
                     $forall (cls,msg) <- msgs
                        <p class="#{cls}">Message: #{msg}
                  |]
          addMessage "warning" "Previous: B"
          return page
```

11.1.6. Persistenz – Anbindung an Datenbanken

Manchmal sollen Daten länger als eine Sitzung halten. Die Bibliotheken `Database.Persist` und `Yesod.Persistent` stellen dazu eine *typsichere* Schnittstelle für Standard-Datenbanken bereit.

Die Bibliothek `Persistent` unterstützt verschiedene Datenbanken, u.a. SQLite, PostgreSQL, MySQL, MongoDB. Die Bibliothek `Persistent` führt viele SQL-Migrationen automatisch aus. Das Modul `Database.Persist` ist unabhängig von Yesod, und kann generell zur Anbindung einer Datenbank an ein Haskell-Programm verwendet werden.

Das Beispiel in Abbildung 11.2 erfordert, dass die Pakete `persistent` und `persistent-sqlite` installiert sind. Wir zeigen hier den vollständigen Code und besprechen ihn im Anschluss.

11.1.6.1. Spezifikation

Die Spezifikation der Datenbanken erfolgt mit TemplateHaskell. Der Aufruf der Form

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
      [persistLowerCase|
         Person   ...
         BlogPost ...
```

```
{-# LANGUAGE EmptyDataDecls, FlexibleContexts,GADTs,
             GeneralizedNewtypeDeriving,MultiParamTypeClasses,
             OverloadedStrings,QuasiQuotes,TemplateHaskell,
             TypeFamilies,DerivingStrategies,StandaloneDeriving,
             UndecidableInstances,DataKinds, FlexibleInstances
#-}


module Main where

import Control.Monad.IO.Class  (liftIO)
import Database.Persist
import Database.Persist.Sqlite
import Database.Persist.TH
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
      [persistLowerCase|
        Person
            name String
            age Int Maybe
            deriving Show
        BlogPost
            title String
            authorId PersonId
            deriving Show
        |]

main :: IO ()
main = runSqlite "dbfile.sql" $ do
    runMigration migrateAll
    johnId <- insert $ Person "John Doe" $ Just 35
    janeId <- insert $ Person "Jane Doe" Nothing

    insert $ BlogPost "My f1rst p0st" johnId
    insert $ BlogPost "One more for good measure" johnId

    oneJohnPost <- selectList [BlogPostAuthorId ==. johnId] [LimitTo 1]
    liftIO $ print (oneJohnPost :: [Entity BlogPost])
    john <- get johnId
    liftIO $ print (john :: Maybe Person)
    delete janeId
    deleteWhere [BlogPostAuthorId ==. johnId]
```

Abbildung 11.2.: Ein Beispiel zur Verwendung von `persist`

(siehe Abbildung 11.2) erzeugt Hilfsfunktionen und die Haskell-Datentypen:

```
data Person { personName :: String, personAge :: Int }
        deriving (Show, Read, Eq)
type PersonId = Key Person

data BlogPost { blogPostTitle    :: String,
                blogPostAuthorId :: PersonId }
        deriving (Read, Eq)
```

Feldtypen müssen Instanzen der Klasse `PersistField` sein. Für Enumerations kann die Instanz-Deklaration mit einer TemplateHaskell Funktion automatisch abgeleitet werden:

```
  data Employment = Employed | Unemployed | Retired
      deriving (Show, Read, Eq)
  derivePersistField "Employment"
  -- Andere Datei:
  share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
  Person
      name String
      employment Employment
  |]
```

Konstruktoren werden in der Datenbank als String gespeichert, welche mit `Show` und `Read` verarbeitet werden. Dies erlaubt nachträgliche Erweiterung der Konstruktoren. Zeilen, die mit Großbuchstaben beginnen, spezifizieren Einschränkungen zu Einzigartigkeit von Datenbankeinträgen, z.B. könnten wir schreiben:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int
      UniquePerson firstName
      deriving Show
  |]
```

Dabei wird ein Konstruktor generiert, der gezieltes Nachschlagen mit der Funktion `getBy` erlaubt, das Feld wird also auch zum Schlüssel und eine Anfrage `getBy $ UniquePerson "Horst"` ist möglich.

Werden mehrere Felder angegeben, dann muss nur die entsprechende Kombination einzigartig sein. Der eigentliche Schlüssel bleibt unverändert. Mit `Primary firstName` würde das Feld zum echten Datenbankschlüssel werden.

Optionale Felder erhalten zusätzlich ein `Maybe` ganz *hinten* angestellt. Dies macht das Feld optional in der Datenbank. Ein Beispiel ist

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
```

```
      lastName String
      age Int Maybe
      deriving Show
  |]
```

11.1.6.2. Migration

Die Bibliothek `Persist` kann sich auch um nachträgliche Änderungen an der Datenbank kümmern und übernimmt die Migration, z.B. kann mit

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int Maybe
      timestamp UTCTime default=CURRENT_TIME
      deriving Show
|]
```

ein Zeitstempel hinzufügt werden. Das Hinzufügen optionaler Felder ist problemlos. Mit `default` können Standardwerte vorgegeben werden. Automatische Migration ist möglich mit der Funktion `runMigration`, wenn man zusätzliche Datentypen hinzufügen möchte, zusätzliche Felder mit Default hinzufügen möchte oder einen Typwechsel bei Feldern durchführen möchte, sofern die Konversion möglich ist (z.B. Wechsel zwischen Optional- und Pflichtfeld).

Will man Felder löschen oder Felder umbenennen, so ist eine manuelle Intervention notwendig.
Die Datenbank Schnittstelle für SQLite ist über die Funktion `runSqlite` gegeben. Siese erwartet als erstes Argument entweder die Datenbank als Dateinamen oder `":memory:"`, falls die Datenbank im flüchtigen Speicher gehalten wird. Danach folgt die Transaktion. Pro Aufruf von `runSqlite` wird eine Datenbanktransaktion durchgeführt. Das folgende Beispiel:

```
main = runSqlite ":memory:" $  do
  runMigration $ migrateAll -- ggf. Migration durchführen
  dieId <- insert $ Person "Horst" 37   -- Einfügen
  horst   <- get horstId                -- Abfragen
  liftIO $ print horst
```

führt eine Default-Migration durch. Diese sollte man mindestens einmal beim Erstellen einer neuen Datenbank durchführen. Die Klasse `PersistStore b m` definiert u.a.

- `insert ::... => val -> m (Key val)`, um einen Wert in die Datenbank einzufügen.
- `get :: ... => Key b val -> m (Maybe val)`, um einen Schlüssel in der Datenbank nachzuschlagen.
- `getBy :: ... => Unique val -> m (Maybe (Entity val))`, um ein `Unique` nachzuschlagen. Das Ergebnis ist vom Typ `Entity valId val`.
- `delete :: ... => Key val -> m ()`, um einen Schlüssel in der Datenbank zu löschen.
- `repsert :: ... => Key val -> val -> m ()`, um einen Schlüssel einzufügen oder zu ersetzen.

Neben `get` und `getBy` gibt es noch echte Datenbank-Abfragen, z.B. mit

```
selectList :: ... => [Filter val] -> [SelectOpt val] -> m [Entity val]
```

Die beiden Argumente sind eine Liste von Filtern und eine Liste von Auswahl-Optionen. Zum Beispiel extrahiert die folgende Abfrage alle Personen im Alter zwischen 26 und 30:

```
people25bis30 <- selectList [PersonAge >. 25, PersonAge <=. 30] []
```

Die Liste der Filter ist UND-verknüpft, wobei die Operatoren wie gewohnt verwendet werden, jedoch mit einem Punkt am Ende versehen sind. Zudem wird `!=.` anstelle von `/=.` verwendet. Die Operatoren `<-.` und `/<-.` stehen für „ist Element von“ und „ist kein Element von“.

Oder-Verknüpfungen müssen mit `||.` erzeugt werden. Damit können kompliziertere Anfragen programmiert werden. Z.B. alle Personen zwischen 26 und 30, oder deren Namen nicht „Adam“ oder „Bonny“ lautet, oder deren Alter genau 50 oder 60 beträgt:

```
people <- selectList
  (       [PersonAge >. 25, PersonAge <=. 30]
      ||. [PersonFirstName /<-. ["Adam", "Bonny"]]
      ||. ([PersonAge ==. 50] ||. [PersonAge ==. 60])
  )
  []
```

Die Auswahl-Optionen sind

- `Asc` *`Feld`* für aufsteigend sortierte Ergebnisse,
- `Desc` *`Feld`* für absteigend sortierte Ergebnisse,
- `LimitTo` n, um die Anzahl der Ergebnisse auf n zu begrenzen und
- `OffSetBy` n, um die ersten n-Ergebnisse zu überspringen.

Z.B. kann man damit die folgende Abfrage programmieren:

```
let resultsPerPage = 10
selectList
    [ PersonAge >=. 18 ]
    [ Desc PersonAge
    , Asc PersonLastName
    , Asc PersonFirstName
    , LimitTo resultsPerPage
    , OffsetBy $ (pageNumber - 1) * resultsPerPage
    ]
```

Diese zeigt alle Erwachsenen an, die absteigend nach Alter, anschließend aufsteigend nach Name und Vorname sortiert sind, wobei nur 10 Einträge angezeigt werden, und `(pageNumber-1)*10` Einträge übersprungen werden.

Alternative Abfragemöglichkeiten sind

- `selectList :: ... => [Filter val] -> [SelectOpt val] -> m [Entity val]` (liefert Ergebnis-Liste),
- `selectFirst :: ... => [Filter val] -> [SelectOpt val] ->  m (Maybe (Entity val))` (liefert nur das erste Ergebnis) und
- `selectKeys::PersistEntity val => [Filter val]->Source (ResourceT (b m)) (Key val)` (liefert nur die Schlüssel der Ergebnisse).

Datenbank-Manipulationen sind möglich mit

- `update :: PersistEntity val => Key val -> [Update val] -> m ()`
  (Datenbankwert verändern),
- `updateWhere :: PersistEntity val => [Filter val] -> [Update val] -> m ()`
  (nur spezielle Werte verändern) und
- `deleteWhere :: PersistEntity val => [Filter val] -> m ()`
  (nur spezielle Werte löschen).

Einige Beispielaufrufe sind:

```
personId <- insert $ Person "Horst" "Hans" 39
update personId [PersonAge =. 40]
updateWhere [PersonFirstName ==. "Horst"] [PersonAge +=. 1]
```

und mögliche Operatoren sind `=.`, `+=.`, `-=.`, `*=.` und `/=.`.

11.1.6.3. Integration in Yesod

Die Datenbank/Yesod-Schnittstelle ist in `Yesod.Persist` definiert durch

```
runDB :: YesodDB site a -> HandlerFor site a
```

Dies erlaubt den Datenbankzugriff in der Handler-Monade. Eine Alternative zu `get` ist:

```
get404 :: ... => Key val -> m val
```

die bei Fehlschlagen der Operation direkt eine 404-Fehlerseite liefert. Die `YesodPersist`-Instanz des Foundation-Typs hält fest, welche Datenbank verwendet wird. Der Foundation-Typ erhält ein Argument für die Datenbank, damit diese überall zugänglich ist. Das Yesod-Scaffolding-Tool kümmert sich bereits um alles.
Ein minimales Yesod-Beispiel ist:

```
data App = App ConnectionPool -- Parameter für Foundation

instance YesodPersist PersistTest where
    type YesodPersistBackend PersistTest = SqlBackend
    runDB action = do
        App pool <- getYesod
        runSqlPool action pool

openConnectionCount :: Int
openConnectionCount = 10

main :: IO ()
main = runStderrLoggingT $ withSqlitePool "myfile.db3"
  openConnectionCount $ \pool -> liftIO $ do
    runResourceT $ flip runSqlPool pool $ do
        runMigration migrateAll
        insert $ Person "Michael" "Snoyman" 26
    warp 3000 $ PersistTest pool
```

```
getPersonR :: PersonId -> Handler String
getPersonR personId = do  person <- runDB $ get404 personId
                          return $ show person
```

Innerhalb von Widgets sind keine Datenbankanfragen erlaubt, d.h. der folgende Code compiliert nicht:

```
[whamlet|
  <ul>
    $forall Entity blogid blog <- blogs
      $with author <- runDB $ get404 $ blogAuthor -- Error
        <li>
          <a href=@{BlogR blogid}>
            #{blogTitle blog} by #{authorName author}
|]
```

Der Grund ist, dass wir innerhalb der Widgets nicht mehr in der `Handler`-Monade sind. Die Abhilfe dazu ist, die Abfrage vorher durchführen:

```
getHomeR :: Handler Html
getHomeR = do blogs <- runDB $ selectList [] []
              defaultLayout $ do
                setTitle "Blog posts"
                [whamlet|
                  <ul>
                    $forall blogEntity <- blogs
                      ^{showBlogLink blogEntity}
                |]
showBlogLink :: Entity Blog -> Widget
showBlogLink (Entity blogid blog) = do
  author <- handlerToWidget $ runDB $ get404 $ blogAuthor blog
  [whamlet|
      <li>
          <a href=@{BlogR blogid}>
              #{blogTitle blog} by #{authorName author}
  |]
```

Mehrere aufeinanderfolgende Datenbankzugriffe sollten möglichst zusammengefasst werden, um die Zugriffe zu beschleunigen. D.h. besser

```
 getHomeR :: Handler Html
 getHomeR do
    (p1Id,p2Id,list) <- runDB $ do
      pers1Id <- insert ...
      pers2Id <- insert ...
      list    <- selectList ...
      return (pers1Id, pers2Id, list)
```

anstelle von

```
getHomeR do
   pers1Id <- runDB $ insert ...
   pers2Id <- runDB $ insert ...
   list    <- runDB $ selectList ...
```

verwenden.

11.2. GUI-Programmierung

Es gibt viele verschiedene Ansätze für die Programmierung grafischer Benutzeroberflächen in Haskell. Eine Übersicht ist z.B. unter `https://wiki.haskell.org/Applications_and_libraries/GUI_libraries` zu finden. Ein (allgemeines) Problem ist die Wartung der Softwarebibliotheken und die Anpassung an neue Versionen. Daher kann manchmal auch die Installation auf dem eigenen System problematisch oder schwierig sein.

11.2.1. Gtk2Hs

Gtk2Hs verwendet das C-Framework Gtk (`https://www.gtk.org/`), welches ein Plattform-übergreifendes Framework ist und für GIMP ToolKit steht und für Linux, Mac OS X, Windows, usw. verfügbar ist. Es gibt verschiedene Bibliotheken:

- GLib: Kernbibliothek, Kompatibilitätsschicht,Thread-Handling
- Cairo: Bibliothek für 2D Vektor-Grafik
- GDK: Rendern, Bitmaps
- Pango: Textdarstellung, Internationalisierung
- ATK: Zugänglichkeit, z.B. Vergrößern, Vorlesen
- Glade: Grafisches GUI Design Tool

Gtk2Hs (`https://wiki.haskell.org/Gtk2Hs`) bietet eine Anbindung von Haskell an *Gtk+ 2.x* über das Paket `gtk` und eine Anbindung an *Gtk 3.x* über das Paket `gtk3` (`https://hackage.haskell.org/package/gtk3`). Die Dokumentation unter `https://wiki.haskell.org/Gtk2Hs` ist teilweise veraltet und bezieht sich nur auf die Version 2.x. Schließlich gibt es eine weitere Anbindung über `gi-gtk` (`https://hackage.haskell.org/package/gi-gtk`). Die Benutzung ist rein durch Haskell möglich und eine Plattform-unabhängige Unterstützung wird geboten. Auch GUI-Beschreibungen mit Glade 3.8 für Gtk+ 2.x und höheren Glade-Versionen für Gtk 3 sind möglich.

Gtk2Hs mit Anbindung an Gtk+ 2.x ist etwas veraltet, zur Verwendung von Gtk 3 gibt es unterschiedliche Empfehlungen, ob die Anbindung über `gi-gtk` oder `gtk3` verwendet werden sollte. Wir verwenden die Anbindung über das Paket `gtk3`, auch deshalb, da es besser dokumentiert scheint und sich nur wenig von der früheren Anbindung `gtk2hs` unterscheidet. Neben den Haskell-Bibliotheken müssen entsprechende C-Bibliotheken auf Systemebene installiert werden. Im Anschluss kann mit `stack new` ein neues Projekt erzeugt werden, und in `package.yaml` unter `dependencies` das Paket `gtk3` eingefügt werden. Die erforderlichen Einträge für `-extra-deps` in `stack.yaml` werden von `stack` beim Kompilieren vorgeschlagen.

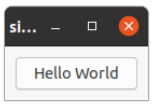
11.2.1.1. Grundgerüst eines Programms

Im Grunde ist Gtk2Hs eine Anbindung an Gtk und daher sehr imperativ. Die Benutzerinteraktion erfolgt durch imperatives I/O, wobei die Eingaben *Ereignisse* (wie Klicks, Tastendrücke, interne Nachrichten, Nachrichten von anderen Applikationen) sind und die *Ausgabe* in der graphischen Darstellung von Inhalten und der Steuerung besteht. Die GUI-Programmierung mit Gtk2Hs erfolgt daher in der IO-Monade.

Gtk ist *ereignisgesteuert*, d.h. der Hauptthread durchläuft eine Schleife in Gtk und Ereignisse werden mit *Callback*-Funktionen behandelt. Daher benötigen die Ereignisbehandler Zugriff auf einen gemeinsamen Zustand. Wir betrachten ein Hello-World-Beispiel, welches ein Fenster mit einem Knopf erzeugt und das Ereignis „Drücken des Knopfes“ behandelt, indem der Text `"Hello World"` auf der Konsole ausgegeben wird.

```
module Main where
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window <- windowNew
  button <- buttonNew
  set window [ containerBorderWidth := 10,
               containerChild       := button ]
  set button [ buttonLabel := "Hello World"   ]
  on button buttonActivated (callback1)
  on window objectDestroy mainQuit
  widgetShowAll window
  mainGUI

callback1 :: IO ()
callback1 = putStrLn "Hello World"
```

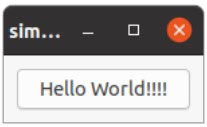
Von der Ausführung erzeugtes Fenster (unter Linux):



Die Funktion `initGUI :: IO [String]` dient zur Initialisierung von Gtk und verwendet dabei mögliche Kommandozeilenparameter. Die Rückgabe (als Liste von Strings) enthält die ungenutzten Kommandozeilenparameter. Die Funktion `mainGUI :: IO ()` übergibt die Kontrolle des Hauptthreads an die Gtk-Ereignisschleife. Die Funktion `mainQuit :: IO ()` beendet die Gtk-Ereignisschleife. Wird ein Ereignis ausgelöst, so wird die Ereignisschleife unterbrochen, um die entsprechende Callback-Funktion auszuführen. Die Funktionen `windowNew` und `buttonNew` erzeugen ein Fenster bzw. einen Knopf. Mit `set` werden Attribute gesetzt, mit `on` werden Ereignisse gebunden. Zunächst wird für den Knopf das Klick-Ereignis mit der Callback-Funktion `callback1` verbunden. Wenn die Hauptschleife ein Klick-Ereignis für den Knopf erhält, so ruft sie daher `callback1` auf. Anschließend wird für das Hauptfenster das Ereignis gebunden, dass das Fenster geschlossen wurde. In diesem Fall wird das Programm mit `mainQuit` ordnungsgemäß beendet. Das folgende Beispiel verwendet zusätzlich noch `get`, um Attributwerte auszulesen:

```
main = do
  initGUI
  window <- windowNew
  button <- buttonNew
  set window [ containerBorderWidth := 10,
               containerChild       := button ]
  set button [ buttonLabel := "Hello World"   ]
  on button buttonActivated $ callback2 button
  on window objectDestroy mainQuit
  widgetShowAll window
  mainGUI

callback2 :: ButtonClass o => o -> IO ()
callback2 b = do
  l <- get b buttonLabel              -- GUI Call
  set b [ buttonLabel := (l ++ "!") ] -- GUI Call
```

Von der Ausführung erzeugtes Fenster (unter Linux), nach viermaligem Klicken des Buttons:



11.2.1.2. Widgets

Elemente einer GUI werden oft als Widget bezeichnet. Widgets haben verschiedene Attribute, welche zur Laufzeit verändert werden können, wie z.B. der Text eines Labels oder eines Knopfs. Widgets können andere Widgets enthalten, z.B. Window, Dialog, etc. Die Anordnung enthaltener Widgets wird durch das übergeordnete Widget bestimmt, die Attribute der Widgets können dies beeinflussen. Z.B. das Padding-Attribut von Container-Widgets.
Das Modul `System.Glib.Attributes` definiert:

```
data ReadWriteAttr o a b
```

für Attribute eines Objekts `o`, wobei `a` der Lesetyp und `b` der Schreibtyp ist. Spezialisierte Typen sind:

```
-- Gewöhnliches Attribut mit identischem Typ für beide Zugriffe
type Attr o a = ReadWriteAttr o a a

-- Attribut kann nur gelesen werden
type ReadAttr  o a = ReadWriteAttr o a ()}

-- Attribut kann nur geschrieben werden
type WriteAttr o b = ReadWriteAttr o () b}
```

Die Funktionen zum Auslesen und Setzen von Attributen sind `get` und `set`:

```
get :: o -> ReadWriteAttr o a b -> IO a
set :: o -> [AttrOp o] -> IO ()
```

Der Datentyp `AttrOp` verwendet existentielle Quantifizierung:

```
data  AttrOp o  =  forall a b.  (:=)   (ReadWriteAttr o a b) b
                |  forall a b.  (:~)   (ReadWriteAttr o a b) (a -> b)
                |  forall a b.  (:=>)  (ReadWriteAttr o a b) (IO b)
                |  forall a b.  (:~>)  (ReadWriteAttr o a b) (a -> IO b)
                |  forall a b.  (::=)  (ReadWriteAttr o a b) (o -> b)
                |  forall a b.  (::~)  (ReadWriteAttr o a b) (o -> a -> b)
```

Daher können in *einer* Liste des Typs `[AttrOp o]` die Elemente verschiedene Typen für `a` und `b` besitzen, z.B.

```
set button [ buttonLabel        := "OK",
             buttonFocusOnClick := False ]
```

Die Operationen sind

- `:=` für die Zuweisung eines Wertes,
- `:~` für die Anwendung einer Funktion auf den aktuellen Wert,
- `:=>` für die Zuweisung des Wertes einer IO-Aktion,
- `:~>` für die Anwendung einer IO-Funktion auf den aktuellen Wert,
- `::=` für die Anwendung einer Funktion, welche als Argument das aktuelle Objekt bekommt, und
- `::~` für die Anwendung einer Funktion, welche als Argumente das aktuelle Objekt und den aktuellen Wert bekommt.

Das Modul `Graphics.UI.Gtk.Abstract.Widget` definiert

- `widgetShowAll :: WidgetClass self => self -> IO ()` zum Anzeigen eines vorbereiteten Widgets auf dem Bildschirm und
- `widgetDestroy :: WidgetClass self => self -> IO ()` zum dauerhaften Entfernen eines Widgets.

Das Modul `Graphics.UI.Gtk.Abstract.Object` definiert das Signal `objectDestroy :: WidgetClass self => Signal self (IO ())`, welches beim Entfernen des Widgets ausgelöst wird. Durch `on widget objectDestroy callback` kann eine Callback-Funktion für das Entfernen-Ereignis angelegt werden.

11.2.1.2.1. Buttons Zu jeder wichtigen Art von Widgets gibt es entsprechende Module, welche spezialisierte Funktionen zur Verfügung stellen. Das Modul `Graphics.UI.Gtk.Buttons.Button` definiert u.a.

- `buttonNew :: IO Button` zum Erzeugen eines Buttons,
- `buttonNewWithLabel :: GlibString string => string -> IO Button` zum Erzeugen eines Buttons mit gesetzter Beschriftung und
- `buttonActivated :: ButtonClass self => Signal self (IO ())` als Signal, dass ein Knopf gedrückt und wieder losgelassen wurde.

Mit `on :: object -> Signal object callback -> callback -> IO (ConnectId object)` wird eine Callback-Funktion an ein Objekt für ein bestimmtes Signal gebunden. Für einen Button `button` können wir daher mit

```
on button buttonActivated callback
```

die Callback-Funktion `callback` für das Signal `buttonActivated`binden.

In `Graphics.UI.Gtk.Buttons.Button` werden verschiedene Attribute für Buttons definiert, die wie vorher beschrieben mit `get` ausgelesen und mit `set` gesetzt werden können. Diese enthält u.a. `buttonLabel` für die Beschriftung des Buttons.

11.2.1.2.2. Entries In `Graphics.UI.Gtk.Entry.Entry` werden einzeilige Textfelder definiert. Mit `entryNew :: IO Entry` kann ein neues Textfeld erzeugt werden. Mit `entrySetText :: ... => self -> string -> IO ()` und `entryGetText :: ... => self -> IO string` kann der Text gesetzt und gelesen werden. Es werden dort auch einige Signale für Tastatureingaben definiert wie z.B. `entryBackspace`, welches bei Drücken der Backspace-Taste im Eingabefeld aktiviert wird. Auch einige Attribute werden dort definiert, z.B. `entryEditable`, welches festlegt, ob das Textfeld editiert werden kann oder nicht.

11.2.1.2.3. Container *Container*-Widgets können Kinder-Widgets enthalten. Das Modul `Graphics.UI.Gtk.Abstract.Container` definiert den Typ `Container` als (leeren) Datentyp und die Typklasse `ContainerClass`, die Methoden bereitstellt, welche von Container-Widgets unterstützt werden:

- `containerAdd::(ContainerClass self, WidgetClass widget) => self->widget->IO ()`, um einem Container ein Widget als Kind hinzuzufügen.
- `containerRemove::(ContainerClass self, WidgetClass widget) => self->widget->IO ()`, um ein Kind zu entfernen.
- `containerGetChildren::ContainerClass self => self -> IO [Widget]`, um alle Kinder auszulesen.
- `containerForeach::ContainerClass self => self -> ContainerForeachCB -> IO ()`, um eine Funktion auf die Kinder anzuwenden. Das Typsynonym `ContainerForeachCB` ist definiert als `type ContainerForeachCB = Widget -> IO ()`.

Dies erlaubt das geordnete Hinzufügen und Entfernen von Widgets zu Containern. Beispiele für Container-Widgets sind Fenster (vom Typ `Window`) und `Bin`, welches Container mit genau einem Kind sind. Dabei ist `Window` eine Unterklasse von `Bin`, d.h. ein Fenster beinhaltet genau ein Kind-Objekt.

Im Modul `Graphics.UI.Gtk.Windows.Window` wird u.a. die Funktion `windowNew :: IO Window` zum Erstellen von Fenstern definiert. Attribute sind u.a. `windowDefaultWidth` (Breite bei Initialisierung) und `windowTitle` (Fenstertitel). Die Funktion `windowPresent` schiebt ein Fenster nach vorne in der Ansicht.

Das Container-Widget `HBox` dient zur horizontalen Ausrichtung seiner Kinder. Es ist in `Graphics.UI.Gtk.Layout.HBox` definiert (analog gibt es Container zur vertikalen Ausrichtung als `VBox`). Mit `hBoxNew :: Bool -> Int -> IO HBox` wird ein neuer Container erstellt, wobei das erste Argument `True` ist, wenn die Kinder mit gleichmäßigem Platz angeordnet werden sollen, und das zweite Argument die Anzahl an freien Pixeln zwischen Kindern angibt.

Wie Widgets sich bei Größenänderungen der `HBox` anpassen, kann durch den Typ `data Packing` aus dem Modul `Graphics.UI.Gtk.Abstract.Box` festgelegt werden: Der Konstruktor `PackGrow` bedeutet, dass Kinder mitwachsen, der Konstruktor `PackNatural` bedeutet, dass Kinder die eigene Größe selbst bestimmen, und `PackRepel` bedeutet, dass der Rand um die Kinder wächst. Das Einfügen der Kinder kann mit

```
boxPackStart::(BoxClass self, WidgetClass child) => self->child->Packing->Int->IO ()
```

geschehen. Der `Int`-Wert legt zusätzlichen Abstand fest. Wir betrachten ein Beispiel:

```
main = do
  initGUI
  window <- windowNew
  hbox     <- hBoxNew True 10
  button1 <- buttonNewWithLabel "Button 1"
  button2 <- buttonNewWithLabel "Button 2"
  button3 <- buttonNewWithLabel "Button 3"
  set window [ containerBorderWidth := 10,
               windowDefaultWidth    :=  500,
               windowDefaultHeight   :=  100,
               containerChild        := hbox ]
  boxPackStart hbox button1 PackGrow 0
  boxPackStart hbox button2 PackGrow 0
  boxPackStart hbox button3 PackRepel 0
  on window objectDestroy mainQuit
  widgetShowAll window
  mainGUI
```

Von der Ausführung erzeugtes Fenster (unter Linux):

Die Ausrichtung von Widgets ist mit dem `Box`-Container nicht möglich. Zur Ausrichtung gibt es jedoch `Alignment`-Widgets aus dem Modul `Graphics.UI.Gtk.Layout.Alignment`. Die `Alignment`-Widgets sind in der `BinClass`, d.h. haben genau ein Kind, dessen Ausrichtung sie bestimmen. Das Erzeugen eines `Alignment`-Widgets ist möglich mit

```
alignmentNew :: Float -> Float -> Float -> Float -> IO Alignment
```

Ein Aufruf ist von der Form `alignmentNew xalign yalign xscale yscale`, wobei `xalign` und `yalign` die relative Ausrichtung festlegen (0=Links/Oben bis 1=Rechts/Unten; 0.5=Mitte), und `xscale`, `yscale` die relative Skalierung des Kindes festlegen (0=wächst nicht; 1=wächst voll). Wir betrachten ein Beispiel:

```
main = do
  initGUI
  window <- windowNew
  hbox   <- hBoxNew True 10
  align1 <- alignmentNew 1 0 0.5 0.5
  align2 <- alignmentNew 0.5 0.5 0.5 0.5
  align3 <- alignmentNew 0 1 0.5 0.5
  button1 <- buttonNewWithLabel "Button 1"
  button2 <- buttonNewWithLabel "Button 2"
  button3 <- buttonNewWithLabel "Button 3"
  set window [ containerBorderWidth := 10,
               windowDefaultWidth    :=  500,
               windowDefaultHeight   :=  100,
               containerChild        := hbox ]
  containerAdd align1 button1
  containerAdd align2 button2
  containerAdd align3 button3
  boxPackStart hbox align1 PackNatural 0
  boxPackStart hbox align2 PackNatural 0
  boxPackStart hbox align3 PackNatural 0
  on window objectDestroy mainQuit
  widgetShowAll window
  mainGUI
```

Von der Ausführung erzeugtes Fenster (unter Linux):

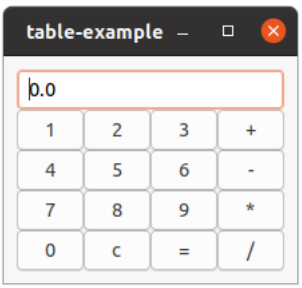
Tabellen erlauben die detaillierte Platzierung. Sie sind ebenfalls `Container`-Widgets. Das Modul `Graphics.UI.Gtk.Layout.Table` stellt u.a. die Funktionen `tableNew` zum Erzeugen der Tabelle und `tableAttachDefaults` zum Hinzufügen der Einträge (mit Default-Werten) zur Verfügung:

```
tableNew :: Int      -- Anzahl Zeilen
         -> Int      -- Abzahl Spalten
         -> Bool     -- Alle Felder gleich groß
         -> IO Table

tableAttachDefaults ::(TableClass self,WidgetClass widget)
  => self -> widget  -- Tabelle, einzufügendes Kind
  -> Int  -> Int     -- Linke und Rechte Spalte
  -> Int  -> Int     -- Obere und Untere Zeile
  -> IO ()
```

Ein Kind-Widget kann mehrere Zellen einer Tabelle belegen, Ein Beispiel für ein Tabellenlayout ist:

```
main = do
    initGUI
    window  <- windowNew
    buttons <- replicateM 16 buttonNew
    entry   <- entryNew
    table   <- tableNew 5 5 False
    set window [ containerBorderWidth := 10,
                 containerChild       := table]
    entrySetText entry "0.0"
    set entry [entryEditable := False]
    sequence_ [do
                tableAttachDefaults table b i (i+1) j (j+1)
                set b [buttonLabel := lab]
               | (b,(i,j),lab) <-
                   zip3
                    buttons
                    [(j,i) | i <- [1..4], j <- [1..4]]
                    ["1","2","3","+"
                    ,"4","5","6","-"
                    ,"7","8","9","*"
                    ,"0","c","=","/"]
                                                        ]
    tableAttachDefaults table entry 0 5 0 1
    on window objectDestroy mainQuit
    widgetShowAll window
    mainGUI
```

Von der Ausführung erzeugtes Fenster (unter Linux):



11.2.1.3. Glade

Glade (siehe https://glade.gnome.org/) ist ein grafisches Werkzeug zum Erstellen statischer GUIs. Man legt damit fest, welche Widgets es in der Anwendung gibt, welches Widget in welchem anderen wo enthalten ist, welche Attribute anfangs eingestellt sind, z.B. Ausrichtung, Größe, Beschriftung, etc. Das Ergebnis wird sofort auf dem Bildschirm angezeigt, d.h. man kann die GUI damit recht einfach entwerfen. Beim Abspeichern wird die GUI durch eine XML Datei

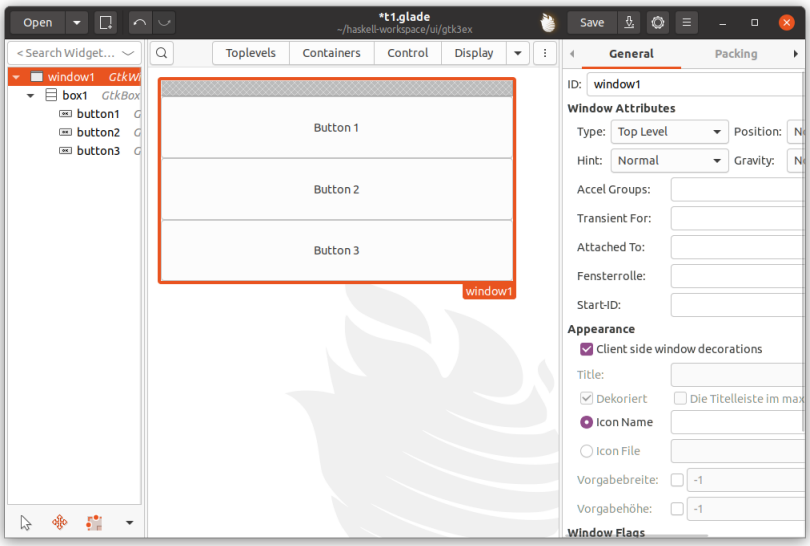


Abbildung 11.3.: Glade-Designer

beschrieben, die dann zur Laufzeit eingelesen wird und mitgeliefert werden muss (Alternativ kann die XML-Beschreibung als auch String im Haskell-Code untergebracht werden.). Zum Verwenden muss man das Modul `Graphics.UI.Gtk.Builder` explizit importieren und kann anschließend die folgenden Funktionen verwenden:

- `builderNew :: IO Builder` zum Erzeugen eines leeren Builder-Objekts.
- `builderAddFromFile :: GlibFilePath fp => Builder -> fp -> IO ()`, um eine Definition aus einer Datei zum Builder hinzufügen.
- `builderAddFromString :: Builder -> string -> IO ()`, um eine Definition als String zm Builder hinzufügen.
- `builderGetObject:: ... => Builder -> (GObject -> cls) -> string -> IO cls`, um die Objekte wieder aus dem Builder zu extrahieren. Das zweite Argument ist ein Type-cast, z.B. `castToWindow`, `castToButton`, ... und das dritte Argument das Namen-Attribut des gesuchten Objektes in XML.

Das Einlesen des XML-Codes wird erst zur Laufzeit durchgeführt, sodass hier Laufzeitfehler entstehen können. Eine Beispielansicht im Designerprogramm Glade ist in Abbildung 11.3 zu sehen. Das Programm zum Einbinden der erstellten Beschreibung (namens `t1.glade`) ist:

```
module Main where

import Graphics.UI.Gtk
import Graphics.UI.Gtk.Builder

main :: IO ()
main = do
   initGUI
   builder <- builderNew
   builderAddFromFile builder "t1.glade"
   window <- builderGetObject builder castToWindow "window1"
   box1   <- builderGetObject builder castToBox "box1"
```

```
   button1 <- builderGetObject builder castToButton "button1"
   button2 <- builderGetObject builder castToButton "button2"
   button3 <- builderGetObject builder castToButton "button3"
   widgetShowAll window
   on button1 buttonActivated $ callback button1
   on button2 buttonActivated $ callback button2
   on button3 buttonActivated $ callback button3
   on window objectDestroy mainQuit
   mainGUI

callback b = do
  l <- get b buttonLabel              -- GUI Call
  set b [ buttonLabel := (l ++ "!") ] -- GUI Call
```

Um Fehler zu vermeiden, ist zu empfehlen, bei Initialisierung der GUI *alle Elemente* auf einmal anzulegen und in zentralen Datentypen abzulegen. Passt die XML-Beschreibung dann nicht zum Code, so werden die Fehler am Programmstart erkannt. Danach garantiert das Typsystem alles weitere. D.h. für unser vorheriges Beispiel ist folgender Code besser:

```
module Main where

import Graphics.UI.Gtk
import Graphics.UI.Gtk.Builder
data MainGUI = MainGUI {  mainWindow   :: Window
                       ,  mainBox       ::Box
                       ,  button1  :: Button
                       ,  button2  :: Button
                       ,  button3  :: Button
                       }
main = do initGUI
          gui <- loadGUI    -- selbstdefinierte Funktion
          on (mainWindow gui) objectDestroy mainQuit
          on (button1 gui) buttonActivated $ callback $ button1 gui
          on (button2 gui) buttonActivated $ callback $ button2 gui
          on (button3 gui) buttonActivated $ callback $ button3 gui
          widgetShowAll (mainWindow gui)
          mainGUI

loadGUI = do -- all XML loading errors must occur here
  builder <- builderNew
  builderAddFromFile builder "t1.glade"
  mainWindow  <- builderGetObject builder castToWindow "window1"
  mainBox     <- builderGetObject builder castToBox    "box1"
  button1     <- builderGetObject builder castToButton "button1"
  button2     <- builderGetObject builder castToButton "button2"
  button3     <- builderGetObject builder castToButton "button3"
  return $ MainGUI mainWindow mainBox button1 button2 button3
```

11.2.1.4. Eingefrorene GUI

Betrachte das folgende Beispiel:

```
module Main where
import Graphics.UI.Gtk
main = do
  initGUI
  window <- windowNew
  button <- buttonNew
  set window [ containerBorderWidth := 10,
               containerChild       := button ]
  set button [ buttonLabel := "Fib 1"   ]
  on button buttonActivated $ callback3 button
  on window objectDestroy mainQuit
  widgetShowAll window
  mainGUI

callback3 :: ButtonClass o => o -> IO ()
callback3 b = do
  l1 <- get b buttonLabel
  let n   = read $ (words l1) !! 1
  let fin = show $ fib n
  let l2  = "Fib " ++ (show $ n+1) ++ " = " ++ fin
  set b [ buttonLabel := l2 ]

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

Die GUI friert hier bei der Berechnung größerer Fibonacci-Zahlen ein, d.h. sie reagiert nicht mehr auf Knöpfe, Größenänderung, etc. Wird ein Ereignis ausgelöst, so wird die Ereignisschleife unterbrochen, um die entsprechende Callback-Funktion im Hauptthread auszuführen. Das Problem ist, dass die Berechnung der Callback-Funktion aufwändig ist und dabei die GUI einfriert. Die Abhilfe ist, die Berechnung nebenläufig durchzuführen. Ein einfaches `forkIO` ist jedoch die falsche Lösung, da Gtk+ nicht Thread-sicher ist, d.h. GUI-Funktionen dürfen nur aus dem Hauptthread heraus aufgerufen werden, sonst drohen unvorhersehbare Abstürze. Daher bietet Gtk2hs die folgenden Funktionen an:

- `postGUISync :: IO a -> IO a` führt die Aktion im Hauptthread aus und blockiert den aktuellen Thread, bis das Ergebnis zurückgegeben wird.
- `postGUIAsync :: IO () -> IO ()` führt die Aktion im Hauptthread aus, der aktuelle Thread läuft weiter.

Damit kann man vom nebenläufigen Thread aus, GUI-Aktionen im Hauptthread ausführen. Das Beispiel kann daher besser programmiert werden als:

```
...
callback3 :: ButtonClass o => o -> IO ()
callback3 b = do
  l1 <- get b buttonLabel
  let n   = read $ (words l1) !! 1
  let fibn = fib n
  let fin = show $ fibn
  let l2  = "Fib " ++ (show $ n+1) ++ " = " ++ fin
  forkIO (seq fibn $ postGUIAsync (set b [ buttonLabel := l2 ]))
  return ()
```

11.2.1.5. Menüs und Werkzeugleisten

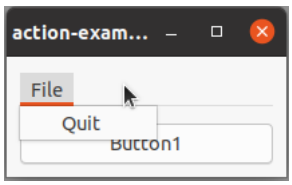
Menüs, Werkzeugleisten und Tastenkombinationen starten Aktionen, welche durch den Typ `Action` repräsentiert werden. Aktionen können aktiviert und deaktiviert werden.

```
actionNew :: GlibString string =>
      string        --  name   : unique name for the action
  ->  string        --  label  : displayed in items & btns
  ->  Maybe string  --  tooltip
  ->  Maybe StockId --  stockId: icon to be displayed
  ->  IO Action
actionActivated :: ActionClass self => Signal self (IO ())
```

Das folgende Beispiel zeigt ein einfaches Menü, welches die `File -> Quit`-Option zum Schließen des Fensters anbietet:

```
module Main where
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window <- windowNew
  vbox     <- vBoxNew True 10
  createMenu vbox
  button <- buttonNewWithLabel "Button1"
  boxPackStart vbox button PackNatural 0
  set window [ containerBorderWidth := 10,
               containerChild       := vbox ]
  on window objectDestroy mainQuit
  widgetShowAll window
  mainGUI
-- XML-Menübeschreibung:
uiDecl = "\
\<ui>\
\  <menubar>\
\    <menu action=\"FILE_MENU\">\
\      <menuitem action=\"QUIT\"/>\
\    </menu>\
\  </menubar>\
\</ui>"

createMenu :: VBox -> IO ()
createMenu box = do
  actFileMenu <- actionNew "FILE_MENU" "File" Nothing Nothing
  actQuit     <- actionNew "QUIT" "Quit" (Just "Exit") Nothing
  on actQuit actionActivated mainQuit
  actGroup   <- actionGroupNew "ACTION_GROUP"
  mapM (actionGroupAddAction actGroup) [actFileMenu,actQuit]
  ui <- uiManagerNew
  uiManagerAddUiFromString ui uiDecl
  uiManagerInsertActionGroup ui actGroup 0
  Just menubar <- uiManagerGetWidget ui "/ui/menubar"
  boxPackStart box menubar PackNatural 0
```

Von der Ausführung erzeugtes Fenster (unter Linux):



11.2.1.6. Beispiel: Taschenrechner

Zum Abschluss zeigen wir in Abbildung 11.4 noch den Code für einen Taschenrechner. Wir verwenden dabei den in Kapitel 5 entwickelt Code wieder, aber verwenden statt der Zustandsmonde eine `IORef`, um den Zustand zu speichern. Die Verwendung eines Monad-Transformers wäre auch möglich, würde aber tieferes Eingreifen in die `gtk3`-Bibliothek erfordern, da z.B. `buttonActivated` als Handler eine IO-Aktion möchte, der natürliche Typ aber ein Monad-Transformer an dieser Stelle wäre.

11.2.2. Threepenny-GUI

Die Bibliothek `Threepenny-GUI` (`https://wiki.haskell.org/Threepenny-gui`) ist eine einfache, sehr stabile und Plattform-unbhängige Lösung zur Programmierung einer GUI. Sie benutzt den Browser als Display. Sie kompiliert nach JavaScript. Das Taschenrechner-Beispiel kann damit wie in Abbildung 11.5 gezeigt programmiert werden.

Ausführung des Programms erzeugt einen Webserver, der den Taschenrechner bereitstellt und unter `http://localhost:8023` aufgerufen werden kann. Wir erläutern das Programm nicht weiter, sondern verweisen auf die Dokumentation.

11.3. Quellen

Der Abschnitt zu Yesod stammt im Wesentlichen aus dem Material von Steffen Jost (Jost, 2019) und dem Buch von Michal Snoyman (Snoyman, 2012). Der Abschnitt zu Gtk2Hs stammt ebenfalls ursprünglich aus (Jost, 2019) wurde jedoch auf `gtk3` angepasst und anders aufgebaut. Das Buch (O'Sullivan et al., 2008) enthält ebenfalls ein Kapitel zu Gtk2hs, aber es basiert auf der veralteten Gtk+ 2.x-Bibliothek.

```
module Main where

import Graphics.UI.Gtk
import Control.Concurrent
import Control.Monad
import Data.Char
import Data.IORef
type InternalCalcState = (Double -> Double,Double)
main = do
    initGUI
    window  <- windowNew
    buttons <- replicateM 16 buttonNew
    entry   <- entryNew
    let start  = (id, 0.0)
    calcState <- newIORef start
    let callback b = do
            l:_ <- get b buttonLabel
            calcStep l
            x <- readIORef calcState
            entrySetText entry (show $ snd x)
        calcStep   x  | isDigit x = digit  (fromIntegral $ digitToInt x)
        calcStep  '+' = oper  (+)
        calcStep  '-' = oper  (-)
        calcStep  '*' = oper  (*)
        calcStep  '/' = oper  (/)
        calcStep  '=' = total
        calcStep  'c' = clear
        calcStep  _   = return ()
        oper  op =  modifyIORef calcState (\(fn,num) -> (op (fn num), 0))
        total    =  modifyIORef calcState (\(fn,num) -> (id,fn num))
        clear    =  modifyIORef calcState (\(fn,num) -> if num == 0.0 then start
                                                       else (fn,0.0))
        digit  i =  modifyIORef calcState (\(fn,num) -> (fn,num*10 + (fromIntegral i) ))
    table   <- tableNew 5 5 False
    set window [ containerBorderWidth := 10, containerChild       := table]
    entrySetText entry "0.0"
    set entry [entryEditable := False]
    sequence_ [do tableAttachDefaults table b i (i+1) j (j+1)
                  set b [buttonLabel := lab]
               | (b,(i,j),lab) <- zip3
                                  buttons
                                  [(j,i) | i <- [1..4], j <- [1..4]]
                                  ["1","2","3","+"
                                  ,"4","5","6","-"
                                  ,"7","8","9","*"
                                  ,"0","c","=","/"]
            ]
    tableAttachDefaults table entry 0 5 0 1
    sequence_ [on b buttonActivated $ callback b | b <- buttons]
    on window objectDestroy mainQuit
    widgetShowAll window
    mainGUI
```

Abbildung 11.4.: Taschenrechner mit Gtk2Hs

```
import Control.Monad
import Data.IORef
import Data.Char
import qualified Graphics.UI.Threepenny as UI
import Graphics.UI.Threepenny.Core
main :: IO ()
main = startGUI defaultConfig setup
setup :: Window -> UI ()
setup w = void $ do
 entryItem <- UI.input # set UI.value "0"
 let start     = (id, 0.0)
 calcState <- liftIO $ newIORef start
 let callback b (l:_) =  do
          calcStep l
          x <- liftIO $ readIORef calcState
          element entryItem # set UI.value (show $ snd x)
     calcStep   x | isDigit x = digit  (fromIntegral $ digitToInt x)
     calcStep  '+' = oper  (+)
     calcStep  '-' = oper  (-)
     calcStep  '*' = oper  (*)
     calcStep  '/' = oper  (/)
     calcStep  '=' = total
     calcStep  'c' = clear
     calcStep  _   = return ()
     oper  op =  liftIO $ modifyIORef calcState (\(fn,num) -> (op (fn num), 0))
     total    =  liftIO $ modifyIORef calcState (\(fn,num) -> (id,fn num))
     clear    =  liftIO $ modifyIORef calcState (\(fn,num) -> if num == 0.0
                                                              then start
                                                              else (fn,0.0))
     digit  i =  liftIO $ modifyIORef calcState
                                      (\(fn,num) -> (fn,num*10 +(fromIntegral i)))
 return w # set title "Buttons"
 buttons <- replicateM 16 UI.button
 let labeledbuttons = zip buttons ["1","2","3","+"
                                  ,"4","5","6","-"
                                  ,"7","8","9","*"
                                  ,"0","c","=","/"]
 sequence_ [element b # set UI.text t | (b,t) <- labeledbuttons]
 sequence_ [on UI.click b  (\_ -> callback b t) | (b,t) <- labeledbuttons]
 getBody w #+ [UI.table #+ (
                 [UI.tr #+ [(UI.td # set UI.colspan 4) #+ [element entryItem]]]
              ++ [UI.tr #+ [UI.td #+ [element b] | b <- take 4 buttons]]
              ++ [UI.tr #+ [UI.td #+ [element b] | b <- take 4 $ drop 4 buttons]]
              ++ [UI.tr #+ [UI.td #+ [element b] | b <- take 4 $ drop 8 buttons]]
              ++ [UI.tr #+ [UI.td #+ [element b] | b <- take 4 $ drop 12 buttons]])
              ]
```

Abbildung 11.5.: Taschenrechner mit Threepenny-GUI

Literatur

Allen, C., Moronuki, J., & Syrek, S. (**2019**). *Haskell Programming from First Principles*. Lorepub LLC.

Ariola, Z. M. & Felleisen, M. (**1997**). The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., & Wadler, P. (**1995**). The call-by-need lambda calculus. In *POPL '95: Proceedings of the 22th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 233–246. ACM Press, New York, NY, USA.

Baader, F. & Nipkow, T. (**1998**). *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA.

Bancilhon, F. & Spyratos, N. (**1981**). Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575.

Barendregt, H. P. (**1984**). *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York.

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

Bird, R. (**2014**). *Thinking Functionally with Haskell*. Cambridge University Press.

Bird, R. & Wadler, P. (**1988**). *Introduction to Functional Programming*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, Upper Saddle River, NJ, USA.

Bohannon, A., Vaughan, J. A., & Pierce, B. C. (**2005**). Relational lenses: A language for defining updateable views. Poster presented at Greater Philadelphia DB/IR Day.

Bohannon, A., Vaughan, J. A., & Pierce, B. C. (**2006**). Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

Bragilevsky, V. (**2021**). *Haskell in Depth*. Manning.

Chakravarty, M. & Keller, G. (**2004**). *Einführung in die Programmierung mit Haskell*. Pearson Studium.

Church, A. (**1941**). *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey.

Damas, L. & Milner, R. (**1982**). Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, New York, NY, USA.

Davie, A. J. T. (**1992**). *An introduction to functional programming systems using Haskell*. Cambridge University Press, New York, NY, USA.

Hall, C. V., Hammond, K., Jones, S. L. P., & Wadler, P. (**1996**). Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138.

Hankin, C. (**2004**). *An introduction to lambda calculi for computer scientists*. Number 2 in Texts in Computing. King's College Publications, London, UK.

HaskellWiki (**2020**). A practical template haskell tutorial. `https://wiki.haskell.org/A_practical_Template_Haskell_Tutorial`.

Henglein, F. (**1993**). Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289.

Hindley, J. R. (**1969**). The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60.

Hoare, C. A. R. (**1969**). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Hughes, J. (**1989**). Why Functional Programming Matters. *Computer Journal*, 32(2):98–107.

Hutton, G. (**2007**). *Programming in Haskell*. Cambridge University Press.

Jones, M. P. (**1995**). A system of constructor classes: Overloading and implicit higher-order polymorphism. *J. Funct. Program.*, 5(1):1–35.

Jost, S. (**2019**). Material zur Vorlesung Fortgeschrittene Funktionale Programmierung im WS 2018/19 an der LMU München. `https://www.tcs.ifi.lmu.de/lehre/ws-2018-19/fun/`.

Kfoury, A. J., Tiuryn, J., & Urzyczyn, P. (**1990**a). Ml typability is dexptime-complete. In *CAAP '90: Proceedings of the fifteenth colloquium on CAAP'90*, pages 206–220. Springer-Verlag New York, Inc., New York, NY, USA.

Kfoury, A. J., Tiuryn, J., & Urzyczyn, P. (**1990**b). The undecidability of the semi-unification problem. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 468–476. ACM, New York, NY, USA.

Kfoury, A. J., Tiuryn, J., & Urzyczyn, P. (**1993**). Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311.

Kiselyov, O., Jones, S. P., & Shan, C. (**2010**). Fun with type functions. In A. W. Roscoe, C. B. Jones, & K. R. Wood, editors, *Reflections on the Work of C. A. R. Hoare*, pages 301–331. Springer.

Klop, J. W. (**2007**). New fixed point combinators from old. In E. Barendsen, H. Geuvers, V. Capretta, & M. Niqui, editors, *Reflections on Type Theory, Lambda Calculus, and the Mind – Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday*.

Kmett, E. A. (**2020**). Webseite zur lens-bibliothek. `https://github.com/ekmett/lens/`.

Kutzner, A. (**2000**). *Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic choice: Operationale Semantik, Programmtransformationen und Anwendungen*. Dissertation, J. W. Goethe-Universität Frankfurt. In german.

Kutzner, A. & Schmidt-Schauß, M. (**1998**). A nondeterministic call-by-need lambda calculus. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 324–335. ACM Press.

Lipovaca, M. (**2011**). *Learn You a Haskell for Great Good! A Beginner?s Guide*. No Starch Press, USA, 1st edition.

Maguire, S. (**2019**). *Thinking with Types – Type-Level Programming in Haskell*. `https://leanpub.com/thinking-with-types`. `https://leanpub.com/thinking-with-types`.

Mairson, H. G. (**1990**). Deciding ml typability is complete for deterministic exponential time. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 382–401. ACM, New York, NY, USA.

Mann, M. (**2005**a). *A Non-Deterministic Call-by-Need Lambda Calculus: Proving Similarity a Precongruence by an Extension of Howe's Method to Sharing*. Dissertation, J. W. Goethe-Universität, Frankfurt.

Mann, M. (**2005**b). Congruence of bisimulation in a non-deterministic call-by-need lambda calculus. *Electronic Notes in Theoretical Computer Science*, 128(1):81–101.

Maraist, J., Odersky, M., & Wadler, P. (**1998**). The call-by-need lambda calculus.

Journal of Functional Programming, 8(3):275–317.

Marlow, S., editor (**2010**). *Haskell 2010 Language Report*.

Marlow, S. (**2013**). *Parallel and Concurrent Programming in Haskell*. O'Reilly.

Marlow, S., Peyton Jones, S., Kmett, E., & Mokhov, A. (**2016**). Desugaring haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, page 92–104. Association for Computing Machinery, New York, NY, USA.

McBride, C. & Paterson, R. (**2008**). Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13.

Milner, R. (**1978**). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Moggi, E. (**1991**). Notions of computation and monads. *Inf. Comput.*, 93(1):55–92.

Morris, J. H., Jr. (**1968**). *Lambda Calculus Models of Programming Languages*. Ph.D. thesis, MIT, Cambridge, MA.

Mycroft, A. (**1984**). Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228. Springer-Verlag, London, UK.

O'Connor, R. (**2012**). Polymorphic update with van laarhoven lenses. `http://r6.ca/blog/20120623T104901Z.html`.

O'Sullivan, B., Goerzen, J., & Stewart, D. (**2008**). *Real World Haskell*. O'Reilly Media, Inc.

Penner, C. (**2020**). *Optics By Example*. `https://leanpub.com/optics-by-example`. `https://leanpub.com/optics-by-example`.

Pepper, P. (**1998**). *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. Springer-Lehrbuch. ISBN 3-540-64541-1.

Pepper, P. & Hofstedt, P. (**2006**). *Funktionale Programmierung - Weiterführende Konzepte und Techniken*. Springer-Lehrbuch. ISBN 3-540-20959-X.

Peyton Jones, S., editor (**2003**). *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, New York, NY, USA. `www.haskell.org`.

Peyton-Jones, S. (**2007**). Beautiful concurrency. In A. Oram & G. Wilson, editors, *Beautiful code*. O'Reilly.

Peyton Jones, S. L. (**1987**). *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Peyton Jones, S. L. (**2001**). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In T. Hoare, M. Broy, & R. Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, Amsterdam, The Netherlands.

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

Plotkin, G. D. (**1975**). Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159.

Schmidt-Schauß, M. (**2009**). Skript zur Vorlesung „Einführung in die Funktionale Programmierung", WS 2009/10. http://www.ki.informatik.uni-frankfurt.de/WS2009/EFP.

Schmidt-Schauß, M., Sabel, D., & Machkasova, E. (**2010**). Simulation in the call-by-need lambda-calculus with letrec. In C. Lynch, editor, *Proceedings of the 21st In-*

ternational Conference on Rewriting Techniques and Applications, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 295–310. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

Schmidt-Schauß, M., Schütz, M., & Sabel, D. (**2008**). Safety of Nöcker's strictness analysis. *Journal of Functional Programming*, 18(4):503–551.

Snoyman, M. (**2012**). *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, Inc. Online verfügbar (angepasst an aktuelle Versionen) unter `https://www.yesodweb.com/book-1.6`.

Snoyman, M. (**2017**). Functors, applicatives, and monads. `http://www.snoyman.com/blog/2017/01/functors-applicatives-and-monads`.

Thompson, S. (**1999**). *Haskell: The Craft of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

van Laarhoven, T. (**2009**). Cps based functional references. `https://www.twanvl.nl/blog/haskell/cps-functional-references`.

Wadler, P. (**1987**). Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 307–313. ACM Press.

Wadler, P. (**1992**). Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493.

Wadler, P. (**1995**). Monads for functional programming. In J. Jeuring & E. Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer.

Wadler, P. & Blott, S. (**1989**). How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76. ACM, New York, NY, USA.

Yorgey, B. A., Weirich, S., Cretin, J., Jones, S. L. P., Vytiniotis, D., & Magalhães, J. P. (**2012**). Giving haskell a promotion. In B. C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 53–66. ACM.

A

Simulation von Turingmaschinen in Haskell

```
-- ------------------------------------------------------------
-- Importe:
import Data.List
import Data.Maybe
-- ------------------------------------------------------------

-- Datentypen

-- Der Move-Typ stellt dar, wohin der Lese/Schreib-Kopf bewegt
-- werden soll (Links, Rechts, Stehenbleiben)

data Move = MLeft | MRight | MNothing
 deriving (Eq, Show)

-- Der TM-Typ dient zur Darstellung einer Turingmaschine,
-- Eingabe- und Bandalphabet sowie die gesamte Zustandsmenge
-- werden nur implizit dargestellt,
-- daher verbleiben 3 Komponenten:
--  start: Der Startzustand der Turingmaschine
--  accepting: Menge der akzeptierenden Zust"ande
--  delta: "Ubergangsfunktion
-- Der Datentyp ist polymorph "uber dem Bandalphabet und den
-- Zust"anden definiert. Beachte, dass "Leer"-Symbol auf dem
-- Band wird dargestellt, indem anstelle von alphabet der Typ
-- (Maybe alphabet) verwendet wird, wobei
--  - Nothing ist der Inhalt eines uninitialisierten
--    (d.h. leeren) Eintrags
--  - Just x ist der Inhalt eines Bandeintrags mit
--    Symbol x aus dem Alphabet


data TM alphabet state =
 TM {
  start     :: state,
  accepting :: [state],
  -- delta erh"alt den Zustand und das aktuelle Symbol
  -- und liefert den Nachfolgezustand, das neue Symbol,
  -- und die Bewegung des Lese/Schreibe-Kopfs:
  delta     :: (state,Maybe alphabet) -> (state,Maybe alphabet,Move)
  }

-- Datentyp f"ur eine Turingmaschinenkonfiguration:
```

```
-- Das aktuelle Band ist before ++ [current] ++ after,
-- wobei der Kopf auf current steht
-- currState ist der aktuelle Zustand der Maschine

data TMConfig alphabet state =
 TMConfig {
  before    :: [Maybe alphabet],
  current   :: Maybe alphabet,
  after     :: [Maybe alphabet],
  currState :: state
 }

-- ------------------------------------------------------------
-- Funktionen

-- oneStep berechnet einen Schritt der Turingmaschine
-- oneStep erwartet als Eingaben eine Turingmaschine und eine
-- Konfiguration
-- Falls die Maschine bereits in einem akzeptierenden Zustand ist,
-- liefert oneStep Nothing, anderenfalls liefert oneStep Just c,
-- wobei c die Konfiguration nach Ausf"uhren eines Schrittes ist.

oneStep :: Eq s => TM a s -> (TMConfig a s) -> Maybe (TMConfig a s)
oneStep tm tc
 -- akzeptierender Zustand schon erreicht?
 | (currState tc) `elem` (accepting tm) = Nothing
 -- sonst:
 | otherwise =
    case (delta tm) (currState tc, current tc) of
    (s',a',m) -> -- Nachfolgezustand, Symbol, Kopfbewegung
      case m of
       MNothing -> Just $ tc {currState = s', current = a'}
       MRight ->
        if null (after tc) then
          Just $ tc {currState = s',
                     current = Nothing,
                     before = (before tc)++[a'],
                     after = []}
        else Just $ tc {currState = s',
                        current = head (after tc),
                        before = (before tc) ++ [a'],
                        after = tail (after tc)}
       MLeft ->
        if null (before tc) then
           error "move left on start position"
         else if (null (after tc)) && (isNothing a') then
                Just $ tc {currState = s',
                           current = last (before tc),
                           before = withoutLast (before tc),
                           after = []}
              else
                Just $ tc {currState = s',
```

```
                            current = last (before tc),
                            before = withoutLast (before tc),
                            after = a':(after tc)}

    where
     -- Hilfsfunktion: alle Elemente einer Liste ohne Letztes:
     withoutLast xs = reverse (tail (reverse xs))

-- runMachine erwartet eine Turingmaschine und die Eingabe auf
-- auf dem Band, und simuliert die Turingmaschine.
-- Sie liefert die Endkonfiguration, falls die Maschine in einem
-- akzeptierenden Zustand landet

runMachine tm inputtape =
 let startconfig = TMConfig {current = head inputtape,
                             before = [],
                             after = tail inputtape,
                             currState = (start tm)}
     go tc = case oneStep tm tc of
             Nothing -> tc
             Just tc' -> go tc'
 in go startconfig

-- tmEncode erwartet eine Turingmaschine und die Eingabe auf dem
-- Band und liefert True, falls die Turingmaschine die Eingabe
-- akzeptiert.
tmEncode tm input = case runMachine tm input of
                   (TMConfig _ _ _ _) -> True


-- ------------------------------------------------------------
-- Beispiel

-- ex1 TM die letztes Symbol der Eingabe sucht
ex1 =
 let
  d (1,Just 0) = (1,Just 0, MRight)
  d (1,Just 1) = (1,Just 1,  MRight)
  d (1,Nothing) = (2,Nothing, MLeft)
  d (2,_) = undefined
  input = [Just 0,Just 1,Just 0, Nothing]
  tm =
   TM {
    delta = d,
    accepting = [2],
    start = 1
   }
 in runMachine tm input
```