Modellieren und Implementieren in Haskell

Modeling and Implementing in Haskell

Peter Padawitz
TU Dortmund, Germany
11. März 2023

actual version: https://fldit-www.cs.tu-dortmund.de/∼peter/Essen.pdf
report version: https://fldit-www.cs.tu-dortmund.de/∼peter/EssenR.pdf

Inhalt

Zur Navigation auf Titel, nicht auf Seitenzahlen klicken!

1 Vorwort 7

2 Typen und Funktionen 12

- 1 Produkttypen 13
- 2 Summentypen 16
- 3 Funktionale Typen 18
- 4 Gleichungen, die Konstanten oder Funktionen definieren 21
- 5 Funktionen höherer Ordnung 25
- 6 Rekursiv definierte Funktionen 38

3 Listen 43

- 1 Listenteilung 48
- 2 Lokale Definitionen sind λ-Applikationen 49
- 3 Teillisten und Listenmischung 50
- 4 Funktionslifting auf Listen 51
- 5 Strings sind Listen von Zeichen 52

6 Listen mit Zeiger auf ein Element 53
7 Relationsdarstellung von Funktionen 56
8 Listenfaltung 58
9 Teillisten, Permutationen, Partitionen, Kantenzüge 69
10 Listenlogik 75
11 Listenkomprehension 76
12 Unendliche Listen 86
4 Rekursive Datentypen 92
1 Arithmetische und Boolesche Ausdrücke 103
2 Signaturen, Algebren und Faltungen 107
3 Hilbertkurven 118
4 Zweidimensionale Figuren 123
5 Typklassen und Bäume 128
1 Mengenoperationen auf Listen 129
2 Unterklassen 131
3 Sortieralgorithmen 132
4 Binäre Bäume 133

5 Binäre Bäume mit Zeiger auf einen Knoten 135
6 Ausgeben 141
7 Arithmetische Ausdrücke ausgeben 142
8 Einlesen 145
9 Bäume mit beliebigem Ausgrad 149
10 Baumfaltungen 155
11 Arithmetische Ausdrücke kompilieren 159
12 Arithmetische Ausdrücke reduzieren 164
13 Resümee: Signaturen, Algebren und Faltungen von Kapitel 4 und 5 168
6 Fixpunkte, Graphen, Modallogik und logisches Programmieren 175
1 CPOs und Fixpunkte 175
2 CPO-Semantik rekursiver Gleichungen 178
3 Semiringe 186
4 Graphen 190
5 Semantik modallogischer Formeln 198
6 Vom logischen Programmieren zur μ-Abstraktion 207
7 Funktoren und Monaden 224

1 Kinds: Typen von Typen 224
2 Funktoren 226
3 Monaden und Plusmonaden 231
4 Monaden-Kombinatoren 237
5 Identitätsmonade 241
6 Maybe-Monade 242
7 Listenmonade 245
8 Monadische Definition partieller und mehrwertiger Funktionen 253
9 Substitution und Unifikation 259
10 Generische Tiefen- und Breitensuche in Bäumen 263
11 Leser- und Schreibermonaden 266
12 Zustandsmonaden 270
13 Die IO-Monade 274
14 Funktionale Abhängigkeiten berechnen 277
15 Zustandsvariablen 279

8 Felder 287

1 Die Typklasse für Indexmengen 287
2 Dynamische Programmierung 289

3 Matrizenrechnung 290
4 Graphen als Matrizen 293
5 Alignments 299
9 Monadentransformer und Comonaden 307
1 Verschmelzung von IO- und Maybe-Monade 312
2 Verschmelzung von IO- und Listenmonade 315
3 Verschmelzung von IO-, Maybe- oder Listenmonade mit der Zustandsmonade 317
4 Generische Compiler 319
5 Arithmetische Ausdrücke kompilieren II 323
6 Comonaden 327
7 Nochmal Listen mit Zeiger auf ein Element 331
8 Bäume, comonadisch 336
10 Literatur 343
11 Index 346

1 Vorwort

Interne Links (einschließlich der Seitenzahlen im Index) sind an ihrer blauen Färbung, externe Links (u.a. zu Wikipedia) an ihrer magenta-Färbung erkennbar.

Jede Kapitelüberschrift und jede Seitenzahl in der rechten unteren Ecke einer Folie ist mit dem Inhaltsverzeichnis verlinkt. Namen von Haskell-Modulen wie Examples.hs sind mit den jeweiligen Programmdateien verknüpft.

Links zum Haskell-Download, -Reports, -Tutorials, etc. stehen auf der Webseite Funktionale Programmierung zur LV.

Alle im Folgenden verwendeten Haskell-Funktionen und -Datentypen – einschließlich derjenigen aus dem Haskell-Prelude – werden hier auch definiert.

C- oder Java-Programmierer sollten ihnen geläufige Begriffe wie Variable, Zuweisung oder Prozedur erstmal komplett vergessen und sich von Beginn an auf das Einüben der i.w. algebraischen Begriffe, die funktionalen Daten- und Programmstrukturen zugrundeliegen, konzentrieren. Erfahrungsgemäß bereiten diese mathematisch geschulten und von Java, etc. weniger verdorbenen LeserInnen weniger Schwierigkeiten. Ihr Einsatz in programmiersprachlichen Lösungen algorithmischer Probleme aus ganz unterschiedlichen Anwendungsbereichen ist aber auch für diese Leserschaft häufig Neuland.

Diese Folien enthalten daher viele, für verschieden Anwendungen prototypische Programme aus verschiedenen Anwendungsbereichen, auf die zurückgegriffen werden kann, wenn ein dem jeweiligen Beispiel ähnliches Problem funktionalsprachlich gelöst werden soll. Wichtige Konzepte und Konstrukte von Haskell werden zwar allgemein eingeführt, aber nicht in der klassischen Form formaler Grammatiken, sondern vor allem an Beispielen, von einfachen hin zu komplexen, die den praktischen Gebrauch des jeweiligen Konstrukts – auch im Vergleich mit semantisch äquivalenten nichtfunktionalen und oft weniger transparenten Lösungen – schnell erkennbar werden lassen.

Wer mehr allgemeine Motivation für funktionale Konzepte oder eine vollständige formale Sprachdefinition wünscht, ziehe eines der zahlreichen Lehrbücher und Tutorials oder einen offiziellen Sprachreport zu Rate (siehe Haskell-Wiki, die hiesige Literaturliste und die Webseite zur o.g. LV).

Highlights der Programmierung mit Haskell

- Das mächtige Typkonzept bewirkt die Erkennung der meisten semantischen Fehler eines Haskell-Programms während seiner Compilation bzw. Interpretation.

- Polymorphe Typen, generische Funktionen und Funktionen höherer Ordnung machen die Programme leicht wiederverwendbar, an neue Anforderungen anpassbar sowie – mit Hilfe mathematischer Methoden – verifizierbar.

- Algebraische Datentypen erlauben komplexe Fallunterscheidungen entlang differenzierter Datenmuster.

- Lazy evaluation als standardmäßige Auswertungsstrategie ermöglicht die Implementierung unendlicher Objekte wie Datenströmen, Prozessbäumen u.ä., sowie die Berechnung von Gleichungslösungen wie in logischer Programmierung (Prolog, SQL).

- Datentypen mit Destruktoren sowie Monaden erlauben es, auch imperativ und objekt- oder aspektorientiert programmieren.

Ein funktionales Programms ist ein System rekursiver Gleichungen zwischen funktionalen

Ausdrücken. Seine Ausführung besteht in der Auswertung der Ausdrücke durch Anwendung der Gleichungen. Auch Ein- und Ausgabedaten sind funktionale Ausdrücke, deren Funktionen Konstruktoren genannt werden, weil sie nicht ausgeführt werden, sondern nur den Aufbau der Daten(muster) beschreiben.

Dieses Sprachmodell und der damit einhergehende Programmierstil unterscheiden sich zunächst stark von dem einer objektorientierten, imperativen und zustandsorientierten Sprache wie Java. Sie erlauben oft problemnahere und flexiblere Lösungen, die leicht an neue Anforderungen, modifizierte Datenstrukturen, etc. anpassbar sind. Mit den Konzepten der funktionalen Programmierung, insbesondere mit geeigneten Typklassen, lässt sich aber auch zustandsorientiert und sogar aspektorientiert implementieren, so dass z.B. der gleiche Algorithmus leicht von einer deterministischen in eine nichtdeterministische oder um differenzierte Ausnahmebehandlungen erweiterte Variante übertragbar ist.

Ebenso kann die traditionell der logischen oder relationalen Programmierung vorbehaltene Beantwortung prädikatenlogischer Anfragen funktional realisiert werden. Dies ist eine Konsequenz der lazy evaluation-Strategie, der die Ausführung jedes Haskell-Programms folgt. Sie erlaubt u.a. das Rechnen mit unendlichen Datenströmen, Prozessbäumen, etc. So werden im Folgenden nicht nur ein bestimmter Programmierstil behandelt, sondern ganz allgemein die Klassifikation von Daten- und Programmtypen auf der Basis mathematischer Strukturen sowie deren direkte Implementierung. Damit wird Haskell auch als Entwurfssprache einsetzbar, in der sich formale Modelle direkt ausführen lassen (rapid prototyping).

Die LeserInnen lernen den Umgang mit Konzepten funktionaler, musterbasierter und monadischer Programmierung und deren Einsatz in verschiedenen Anwendungsbereichen, inklusive präziser Modellierungen, was wiederum dazu befähigt, die Konzepte sowie entsprechende Werkzeuge auch in Entwicklungsumgebungen zu nutzen, in denen am Ende in Java, C++ oder einer anderen nichtfunktionalen Sprache programmiert wird. Allerdings werden auch dort zunehmend funktionale Konstrukte eingebaut, was bereits eine Reihe hybrider Sprachen wie z.B. F-Sharp, Objective CAML, Python, Ruby und Scala hervorgebracht hat.

2 Typen und Funktionen

Alle Typen von Haskell sind aus Standardtypen (*Bool*, *Int*, *Float*, etc.), **Typvariablen** und **Typkonstruktoren** zusammengesetzt.

Typvariablen beginnen stets mit einem kleinen Buchstaben, andere Typen mit einem großen oder bestehen aus Sonderzeichen.

Ein Typ ohne Typvariablen heißt **monomorph**. Andernfalls ist er polymorph.

Eine Funktion heißt mono- bzw. polymorph, wenn ihr Typ (Definitions- und Wertebereich) mono- bzw. polymorph ist.

Ein Typ t heißt **Instanz eines Typs** u, wenn t durch Ersetzung von Typvariablen von u aus u entsteht.

Jeder monomorphe Typ bezeichnet eine Menge. Jeder Typ mit Typvariablen $x_1, \ldots, x_n$ bezeichnet eine n-stellige Funktion, die jedem n-Tupel von Instanzen von $x_1, \ldots, x_n$ eine Menge zuordnet.

Der Haskell-Typ () heißt **unit-Typ** und bezeichnet eine Menge mit genau einem Element, das ebenfalls mit () bezeichnet wird.

In der Mathematik schreibt man häufig 1 für den unit-Typ und $*$ für sein einziges Element.

Der Haskell-Typ *Bool* bezeichnet eine zweielementige Menge. Ihre beiden Elemente sind *True* und *False*.

Die wichtigsten Typkonstruktoren sind **Produktbildung**, **Summenbildung** und die Bildung der Menge aller Funktionen mit gegebenem Definitions- und Wertebereich.

Seien $A_1, \ldots, A_n$ Mengen (Typen).

2.1 Produkttypen

Das (kartesische) **Produkt** $A_1 \times \cdots \times A_n$ von $A_1, \ldots, A_n$ wird in Haskell mit runden Klammern und Kommas notiert:

$$(A_1, \ldots, A_n).$$

Man hat diese Notation gewählt, um sie der Bezeichnung der *Elemente* von $A_1 \times \cdots \times A_n$, den n**-Tupeln**, anzugleichen, die man auch mit runden Klammern schreibt:

$$(a_1, \ldots, a_n) \ \in \ A_1 \times \cdots \times A_n. \tag{1}$$

Für alle $1 \leq i \leq n$ nennt man a_i die i-te **Komponente** von $(a_1, \ldots, a_n)$.

Mathematisch betrachtet ist jedes Objekt eines objektorientierten Programms ein Tupel von Werten seiner jeweiligen **Attribute** (Größe, Farbe, Position, Orientierung, o.ä.). Letztere entsprechen den Indizes von (1).

Beispiel

An bestimmten Raumpunkten positionierte farbige Rechtecke sind als Elemente des folgenden Typs dargestellt:

```
type Rect = ((Float,Float),Float,Float,Color)
```

Die erste Komponente ist selbst ein zweistelliges Produkt, dessen Elemente die Koordinaten des Zentrums des jeweiligen Rechtecks wiedergeben. Die zweite und dritte (reellwertige) Komponente liefern Breite und Höhe des Rechtecks. Ein Typ *Color* wird weiter unten definiert.

Jedes Element von *Rect* ist ein Quadrupel der Form $((x, y), b, h, c)$ mit $x, y, b, h \in \mathit{Float}$ und $c \in \mathit{Color}$.

Alternativ kann die Menge der Blöcke als **Datentyp** definiert werden:

```
data DRect = DRect Point Float Float Color
data Point = Point Float Float
```

Die linken Seiten der Gleichungen sind Typnamen, auf der rechten Seite steht ein **Konstruktor**, gefolgt von den Komponententypen des jeweiligen Produkts.

Ein Quadrupel $((x, y), b, h, c) \in \mathit{Rect}$ hat als Element von *DRect* die Form

$$\mathit{DRect}(\mathit{Point}(x)(y))(b)(h)(c).$$

Im Unterschied zu Typkonstruktoren ist *DRect* ein Element- oder Datenkonstruktor: Aus vier Daten des Typs *Point*, *Float* bzw. *Color* bildet *DRect* ein Objekt des gleichnamigen Typs *DRect*.

Die Zuordnung von Attributen (s.o.) zu den Indizes eines Produkts erfolgt durch entsprechende Benennung der Komponenten(typen):

```
data Point = Point {x,y :: Float}
data DRect = DRect {center :: Point, width,height :: Float,
                    color :: Color}
```

Damit können Elemente von *Point* bzw. *DRect* wie z.B.

```
p = Point 5.7 3.66
rect = DRect (Point 5.7 3.66) 11 5 Red
```

informativer und strukturierter definiert werden:

```
p = Point {x=5.7, y=3.66}
rect = DRect {center=p, width=11, height=5, color=Red}
```

2.2 Summentypen

Die **Summe** oder **disjunkte Vereinigung** $A_1 + \cdots + A_n$ von $A_1, \ldots, A_n$ ist in der Mathematik wie folgt definiert:

$$A_1 + \cdots + A_n \ =_{def} \ \{(a, i) \mid a \in A_i,\ 1 \leq i \leq n\}. \tag{2}$$

In Haskell kann $A_1 + \cdots + A_n$ nur als Datentyp definiert werden. Die Indizes von (2) werden zu Konstruktoren. Der Standardtyp Bool ist z.B. eine zweistellige Summe:

```
data Bool = True | False
```

Der obige Komponententyp *Color* von *Block* könnte als sechsstellige Summe definiert werden:

```
data Color = Red | Magenta | Blue | Cyan | Green | Yellow
```

Color besteht aus nullstelligen Konstruktoren, während der obige Konstruktor *DRect* vier Argumente hat. Die Summe $Int + \{\infty\}$ (s.o.) könnte als Datentyp mit einem einstelligen und einem nullstelligen Konstruktor implementiert werden:

```
data Int' = Fin Int | Infinity
```

Während *Int* die Menge der ganzen Zahlen bezeichnet, sind ganze Zahlen als Elemente von *Int'* Ausdrücke der Form *Fin(i)* mit $i \in Int$.

Häufig verwendet werden die folgenden *polymorphen* Summentypen:

```
data Maybe a = Just a | Nothing
data Either a b = Left a | Right b
```

Maybe(a) erweitert eine beliebige (als Instanz der Typvariablen a gegebene) Menge um das Element *Nothing*. *Either(a)(b)* implementiert die Summe zweier beliebiger (als Instanzen von a bzw. b gegebene) Mengen.

Z.B. besteht der monomorphe Typ *Maybe(Int)* aus *Nothing* und den Ausdrücken der Form *Just(i)* mit $i \in Int$. Der monomorphe Typ *Either(Int)(Bool)* besteht aus den Ausdrücken der Form *Left(i)* oder *Right(b)* mit $i \in Int$ und mit $b \in Bool$.

Prinzipiell ist jede endliche Menge mit mindestens zwei Elementen als Summentyp darstellbar: $\{a_1, \ldots, a_n\}$ ist isomorph zu $\{a_1\} + \cdots + \{a_n\}$.

Mit Hilfen von Datentypen können Mengen auch induktiv definiert werden. Man spricht dann von **rekursiven Datentypen**. Sie werden ausführlich in Kapitel 4 behandelt, wo auch allgemeine Schemata für Datentyp-Definitionen zu finden sind. Der am häufigsten verwendete rekursive Datentyp dient der Implementierung von Folgen von Elementen eines beliebigen Typs und ist Thema von Kapitel 3.

2.3 Funktionale Typen

Der dritte Typkonstruktor (neben Produkt- und Summenbildung) ist der Pfeil $\rightarrow$ zur Bildung von Funktionsmengen: Für Typen A und B bezeichnet $A \rightarrow B$ die Menge der Funktionen von A nach B. Jedes Element von $A \rightarrow B$, also jede Funktion $f : A \rightarrow B$ ist eine Standardfunktion

- oder wird durch Gleichungen definiert (s.u.)
- oder **anonym** (unbenannt) als **λ-Abstraktion** $\lambda p.e$ (Haskell-Notation: `\p -> e`) dargestellt, wobei p ein Muster für Argumente (Parameter) von f ist und der Rumpf (*body*) e von $\lambda p.e$ ein aus beliebigen Funktionen und Variablen zusammengesetzter Ausdruck.

Eine Haskell-Funktion f ist **polymorph**, wenn der Definitionsbereich oder der Wertebereich von f ein polymorpher Typ ist.

Muster (**patterns**) bestehen aus **Individuenvariablen** (= Variablen für einzelne Objekte – im Unterschied zu Typvariablen, die für Mengen von Objekten stehen), Konstanten eines arithmetischen Typs und Konstruktoren von Produkt- oder Datentypen. Erstere sind aus einer öffnenden Klammer, Kommas und einer schließenden Klammer zusammengesetzt. Jede Individuenvariable kommt in einem Muster höchstens einmal vor.

Beispiele für Muster:

```
(x,y)   ((x,y),color)   ((x,7),color,True)    Point x y     Point 5 y
Point {x=x,y=6.0}       DRect point b h red   DRect point b h Red
DRect (Point x 7) b h Red
```

Welche Symbole bezeichnen hier Variablen, Konstruktoren bzw. Attribute?

Ein Ausdruck der Form $f(e)$ heißt **Funktionsapplikation** (auch: Funktionsanwendung oder -aufruf). Ist f eine λ-Abstraktion, dann nennt man $f(e)$ eine **λ-Applikation**.

Die λ-Applikation $(\lambda p.e)(e')$ is auswertbar, wenn e' eine **Instanz** von p ist (e' **matcht** p). D.h. e' ist das Ergebnis der Anwendung einer passenden **Substitution**, also einer Funktion σ, die zunächst einmal nur Variablen auf Terme (Ausdrücke) abbildet, dann aber auch selbst auf Terme t angewendet wird: $\sigma(t)$ ist der Term, der aus t entsteht, wenn jede Variable x von t durch den jeweiligen Funktionswert $\sigma(x)$ ersetzt wird.

Daraus folgt, dass die λ-Applikation $(\lambda p.e)(\sigma(p))$ semantisch äquivalent ist zu $\sigma(e)$ und deshalb zu $\sigma(e)$ ausgewertet wird. In Haskell passiert das aber nur dann, wenn $\sigma(p)$ eine **Grundinstanz** ist, d.h. keine Variablen enthält.

Z.B. wird die Applikation $(\lambda(x,y).x*y+5+x)(7,8)$ zunächst zu $7*8+5+7$ ausgewertet und dann weiter zu einer Konstanten:

$$(\lambda(x,y).x*y+5+x)(7,8) \rightsquigarrow 7*8+5+7 \rightsquigarrow 56+5+7 \rightsquigarrow 68$$

Link zur schrittweisen Reduktion dieser Applikation

Die **Redizes**, das sind die Teilausdrücke, die im jeweils nächsten Schritt ersetzt werden, sind rot gefärbt. Die **Redukte**, das sind die Teilausdrücke, welche die Redizes ersetzen, sind grün gefärbt. Reduktionen können mit der reduce-Funktion des Painters erzeugt werden.

Die Redexauswahl folgt der **parallel-outermost**-, **call-by-need**- oder **lazy-Strategie** (siehe [16, 29]) : In jedem Reduktionsschritt werden alle maximalen reduzierbaren Teilausdrücke gleichzeitig durch ihre jeweiligen Redukte ersetzt.

Link zur schrittweisen Auswertung der λ-Applikation

$$(\lambda F.F(\mathit{True}, 4, 77) + F(\mathit{False}, 4, 77))(\lambda(x, y, z).\mathit{if}\ x\ \mathit{then}\ y + 5\ \mathit{else}\ z * 6)$$

In klassischer Algebra und Analysis taucht λ bei der Darstellung von Funktionen nicht auf, wenn Symbole wie x, y, z konventionsgemäß als Variablen betrachtet und daher z.B. für die Polynomfunktion $\lambda x.2 * x^3 + 55 * x^2 + 33 : \mathbb{R} \to \mathbb{R}$ einfach nur $2 * x^3 + 55 * x^2 + 33$ oder $2x^3 + 55x^2 + 33$ geschrieben wird.

Mit dem ghci-Befehl `:type` können die Typen von λ- und anderen Ausdrücken ausgegeben werden:

```
:type \(x,y)->x*y+5+x     ⇝     Num a => (a,a) -> a
```

Das Backslash-Symbol `\` und der Pfeil `->` sind offenbar die Haskell-Notationen des Symbols λ bzw. des Punktes einer λ-Abstraktion. *Num(a)* beschränkt die Instanzen von a auf numerische Typen (siehe Kapitel 5).

Typinferenzregeln geben an, wie sich der Typ eines Ausdrucks aus den Typen seiner Teilausdrücke zusammensetzt:

$$\frac{e_1 :: t_1, \ldots, e_n :: t_n}{(e_1, \ldots, e_n) :: (t_1, \ldots, t_n)} \qquad \frac{p :: t, \quad e :: t'}{\lambda p.e :: t \to t'} \qquad \frac{e :: t \to t', \quad e' :: t}{e(e') :: t'}$$

$$\frac{}{Nothing :: Maybe\ t} \qquad \frac{e :: t}{Just\ e :: Maybe\ t}$$

$$\frac{e :: t}{Left\ e :: Either\ t\ t'} \qquad \frac{e' :: t'}{Right\ e' :: Either\ t\ t'}$$

2.4 Gleichungen, die Konstanten oder Funktionen definieren

Variablen und Konstanten sind die einfachsten Muster. Die Ausführung einer Gleichung mit einer Variablen auf der linken Seite wird diese mit dem Wert, den die Auswertung des Ausdrucks auf der rechten Seite liefert, belegt und damit zu einer gleichnamigen Konstanten.

Mehrere Variablen können durch eine einzige Gleichung mit Werten belegt werden, wenn man sie in ein Muster einbettet, das vom Wert des Ausdrucks auf der rechten Seite gematcht wird:

```
p = Point 5.8 3.3
Point x y = p
```

```
x ⇝ 5.8
y ⇝ 3.3
```

Die letzten beiden Zeilen deuten an, dass aus den beiden Gleichungen darüber bestehende Haskell-Programm x und y mit dem Wert 5.8 bzw. 3.3 belegt.

Konstanten und Funktionen können benannt und mit Hilfe von Gleichungen zwischen Ausdrücken gleichen Typs definiert werden, wobei eine argumentfreie Definition der Form

`f = \p -> e` *äquivalent ist zur* **applikativen** *Definition* `f p = e`

Die so definierte Funktion $f : A \to B$ ist (bzgl. ihres Definitionsbereiches) **partiell**, d.h. sie ist nur auf Elementen von A definiert, die Instanzen von p sind (s.o.).

Umgekehrt ist $f(p) = e$ eine vollständige Definition von f und damit f eine **totale** Funktion, falls jedes Element von A das Muster p matcht. Das gilt offenbar für jede Menge A, wenn p eine Variable ist.

Haskell erlaubt die Definition partieller Funktionen. Wird allerdings bei der *Auswertung* eines Ausdrucks eine partielle Funktion auf ein Argument angewendet, für das sie nicht definiert ist, dann bricht die Auswertung mit einer Fehlermeldung ab.

Soll f den Elementen von A, abhängig von deren jeweiligem Muster, verschiedene Werte in B zuordnen, dann definieren wir f durch mehrere Gleichungen, für jedes Muster eine:

```
f p1 = e1;  f p2 = e2;  ...  f pn = en
```
(1)

Eine Funktion, die den jeweiligen Nachfolger im obigen Typ *Color* berechnet, könnte z.B. wie folgt applikativ definiert werden:

```
nextCol :: Color -> Color
nextCol Red = Magenta; nextCol Magenta = Blue; nextCol Blue = Cyan
nextCol Cyan = Blue; nextCol Green = Yellow; nextCol Yellow = Red
```

nextCol ist eine totale Funktion, weil sie für jedes Element von *Color* eine Wert hat.

Mehrere Definitionsgleichungen in einer Zeile müssen durch ; getrennt werden. Mehrere Zeilen derselben Definition müssen linksbündig untereinander stehen.

Alternativ zu (1) kann man das **case-Konstrukt** verwenden:

```
f x = case x of p1 -> e1;  p2 -> e2;  ...  pn -> en
```
(2)

```
nextCol :: Color -> Color
nextCol c = case c of Red -> Magenta; Magenta -> Blue; Blue -> Cyan
                      Cyan -> Blue; Green -> Yellow; Yellow -> Red
```

Das case-Konstrukt kann auch auf beliebige Ausdrücke (anstelle der Variablen x bzw. c) angewendet werden.

(2) kann wie folgt vereinfacht werden:

```
f = \case p1 -> e1;  p2 -> e2;  ...  pn -> en
```

Der Fall $n = 1$ entspricht einer λ-Abstraktion:

`\case p -> e` *ist äquivalent zu* `\p -> e`

Zur Verwendung von `\case` muss die Spracherweiterung (*language extension*) `LambdaCase` in das *LANGUAGE Pragma*

```
{-# LANGUAGE ... #-}
```

am Anfang des jeweiligen Haskell-Moduls eingefügt werden:

```
{-# LANGUAGE LambdaCase, ... #-}
```

Beispiel

```
nextCol :: Color -> Color
nextCol = \case Red -> Magenta; Magenta -> Blue; Blue -> Cyan
                Cyan -> Blue; Green -> Yellow; Yellow -> Red
```

2.5 Funktionen höherer Ordnung

Funktionen, die andere Funktionen als Argumente oder Werte haben, heißen **Funktionen höherer Ordnung**. Der Typkonstruktor $\rightarrow$ ist **rechtsassoziativ**, d.h., lässt man bei einem Funktionstyp der Form

$$F \quad = \quad A_1 \rightarrow (A_2 \rightarrow \cdots \rightarrow (A_n \rightarrow B) \ldots) \tag{1}$$

die Klammern weg, dann ist immer noch (1) gemeint.

Demgegenüber ist die Applikation von $f \in F$ **linksassoziativ**: Für alle $a_1 \in A_1, \ldots, a_n \in A_n$ ist

$$((\ldots((f\ a_1)\ a_2)\ldots)\ a_{n-1})\ a_n \tag{2}$$

ein Element von B und wir können auch die Klammern von (2) weglassen.

Sei für alle $1 \leq i \leq n$ p_i ein Muster von Elementen des Typs A_i. Dann ist z.B. eine geschachelte λ-Abstraktion der Form $\lambda p_1.\lambda p_2.\ldots.\lambda p_n.e$ ein Element von F und kann durch $\lambda p_1\ p_2\ \ldots\ p_n.e$ abgekürzt werden.

Standardfunktionen auf dem oben eingeführten Summentyp *Maybe(a)* bzw. *Either(a)(b)*:

```
fromJust :: Maybe a -> a
fromJust (Just a) = a

isJust, isNothing:: Maybe a -> Bool
isJust    = \case Just _ -> True; _ -> False
isNothing = \case Just _ -> False; _ -> True

either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g = \case Left a -> f a; Right b -> g b
```

Summenextension

fromJust ist auf *Nothing* nicht definiert, also eine partielle Funktion (s.o.).

Die Fallunterscheidung nach Mustern kann verfeinert werden durch Fallunterscheidungen nach Bedingungen an die Variablen des jeweiligen Musters. Als Beispiel dafür wollen wir den durch *nextCol* definierten, aus sechs Farben bestehenden Farbkreis zu einem aus 1530 reinen oder **Hue-Farben** bestehenden Farbkreis erweitern. Anstelle von *Color* starten wir mit folgendem Datentyp:

```
RGB = RGB Int Int Int
```

RGB(r)(g)(b) ist (die Codierung) eine(r) Hue-Farbe, wenn $r, g, b \in \{0, \ldots, 255\}$ und $0, 255 \in \{r, g, b\}$ gilt. Elemente *RGB(r)(g)(b)* von *RGB* mit $r, g, b \in \{0, \ldots, 255\}$, aber $0, 255 \notin \{r, g, b\}$ repräsentieren eine aufgehellte oder abgedunkelte Variante einer Hue-Farbe. Wo sich die Farben von *Color* in der RGB-Codierung wiederfinden, zeigt die folgende Definition von Konstanten (= nullstelligen Funktionen):

```
red,magenta,green,cyan,blue,yellow,black,white :: RGB
red   = RGB 255 0 0;   magenta = RGB 255 0 255
blue  = RGB 0 0 255;   cyan    = RGB 0 255 255
green = RGB 0 255 0;   yellow  = RGB 255 255 0
black = RGB 0 0 0;     white   = RGB 255 255 255
```

Die zwischen Rot, Magenta, Blau, Cyan, Grün und Gelb liegenden Hue-Farben lassen sich mit folgender Verfeinerung von *nextCol* aufzählen:

```
nextRGB :: RGB -> RGB
nextRGB = \case RGB 255 n 0 | n > 0   -> RGB 255 (n-1) 0
                RGB 255 0 n | n < 255 -> RGB 255 0 (n+1)
                RGB n 0 255 | n > 0   -> RGB (n-1) 0 255
                RGB 0 n 255 | n < 255 -> RGB 0 (n+1) 255
                RGB 0 255 n | n > 0   -> RGB 0 255 (n-1)
                RGB n 255 0 | n < 255 -> RGB (n+1) 255 0
```

Wie das Beispiel zeigt, können Muster einer applikativen Funktionsdefinition um Boolesche Ausdrücke ergänzt werden, die in diesem Zusammenhang **Guards** genannt werden.

Das Symbol | trennt einen Guard vom durch ihn bewachten Muster. Verletzt die bei einem Matching erzeugte Variablensubstitution den Guard, dann wird mit dem nächsten Fall der Funktionsdefinition fortgefahren.

Die folgende Funktion *isHue* prüft für jedes Element von *RGB*, ob es eine Hue-Farbe repräsentiert oder nicht. Ihre Gleichung enthält einige Boolesche Standardfunktionen, nämlich Gleichheit (==), Konjunktion (&&) und Disjunktion (||) sowie eine **lokale Definition** einer Funktion $f : Int \to Int \to Int \to Bool$:

```
isHue :: RGB -> Bool
isHue (RGB r g b) = f r g b || f g r b || f b r g where
                    f x y z = x == 0 && (y == 255 || z == 255)
```

als let-Ausdruck:

```
isHue (RGB r g b) = let f x y z = x == 0 && (y == 255 || z == 255)
                    in f r g b || f g r b || f b r g
```

Offenbar haben == und && eine höhere Priorität als && bzw. ||. Mit Hilfe des im nächsten Kapitel behandelten Listendatentyps lässt sich die Definition von *isHue* weiter vereinfachen.

Funktionen höherer Ordnung auf dem Datentyp *Point* (s.o.)

Abstand zwischen zwei Punkten:

```
distance :: Point -> Point -> Float
distance (Point x1 y1) (Point x2 y2) = sqrt $ (x2-x1)^2+(y2-y1)^2
```

Mit den Attributen x und y von Point, die hier als Funktionen des Typs Point → Float verwendet werden:

```
distance p q = sqrt $ (x q-x p)^2+(y q-y p)^2
```

Änderung der Koordinaten:

```
updateX,updateY :: Float -> Point -> Point
updateX x (Point _ y) = Point x y
updateY y (Point x _) = Point x y
```

Mit den Attributen von Point:

```
updateX x p = p {x = x}
updateY y p = p {y = y}
```

Liegt (x_2, y_2) auf einer Geraden durch (x_1, y_1) und (x_3, y_3)?

```
straight :: Point -> Point -> Point -> Bool
straight (Point x1 y1) (Point x2 y2) (Point x3 y3) =
         x1 == x2 && x2 == x3 ||
         x1 /= x2 && x2 /= x3 && (y2-y1)/(x2-x1) == (y3-y2)/(x3-x2)
```

Rotation eines Punktes p im Uhrzeigersinn um einen Punkt q um a Grad:

```
rotate :: Point -> Float -> Point -> Point
rotate _ 0 p = p
rotate q a p = if p == q then p else (i+x1*c-y1*s,j+x1*s+y1*c)
               where Point i j = q; Point x y = p
                     x1 = x-i; y1 = y-j; s = sin rad
                     c = cos rad; rad = a*pi/180
```

Mit **pattern binding***:*

```
rotate _ 0 p = p
rotate q@(Point i j) a p@(Point x y)
             = if p == q then p else (i+x1*c-y1*s,j+x1*s+y1*c)
               where x1 = x-i; y1 = y-j; s = sin rad
                     c = cos rad; rad = a*pi/180
```

Mit den Attributen von Point:

```
rotate _ 0 p = p
rotate q a p = if p == q then p else (x q+x1*c-y1*s,y q+x1*s+y1*c)
               where x1 = x p-x q; y1 = y p-y q; s = sin rad
                     c = cos rad; rad = a*pi/180
```

Offenbar können lokale Definitionen wie Fallunterscheidungen mehrere linksbündig untereinander stehende Zeilen einnehmen. Dabei werden die Definitionen in *einer* Zeile durch ; voneinander getrennt.

Das oben verwendete Konditional *if_then_else_* ist eine – *mixfix* notierte – Funktion des Typs

$$Bool \to Point \to Point \to Point.$$

In Baumdarstellungen funktionaler Ausdrücke schreiben wir *ite* dafür.

Viele arithmetische Standardfunktionen sind in Haskell höherer Ordnung. So hat z.B. die Addition (+) auf einem arithmetischen Typ A den Typ $A \to A \to A$.

Summenausdrücke können sowohl in Präfixdarstellung (erst die Funktion, dann ihre Argumente) als auch in Infixdarstellung notiert werden. In letzterer entfallen die runden Klammern um den Funktionsnamen:

`5 + 6` *ist äquivalent zu* `(+) 5 6`

Jede Funktion f eines Typs der Form $A \to B \to C$ kann in beiden Darstellungen verwendet werden. Besteht f aus Sonderzeichen, dann wird f bei der Präfixdarstellung in runde Klammern gesetzt. Andernfalls ist f ein String ohne Sonderzeichen, der mit einem Kleinbuchstaben beginnt und bei der Infixdarstellung in Akzentzeichen gesetzt wird:

```
mod :: Int -> Int -> Int
mod 9 5    ist äquivalent zu    9 `mod` 5
```

Die Infixdarstellung wird auch verwendet, um die in f enthaltenen **Sektionen** (Teilfunktionen) des Typs $A \to C$ bzw. $B \to C$ zu benennen. Z.B. sind die folgenden Sektionen Funktionen des Typs `Int -> Int`, während `(+)` und `mod` den Typ `Int -> Int -> Int` haben.

```
(5+)       ist äquivalent zu    (+) 5     ist äquivalent zu    \x -> 5+x
(9`mod`)   ist äquivalent zu    mod 9     ist äquivalent zu    \x -> 9`mod`x
```

Eine Sektion wird stets in runde Klammern eingeschlossen. Die Klammern gehören zum Namen der jeweiligen Funktion.

Der **Applikationsoperator** ist die wie folgt definierte polymorphe Funktion:

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

führt die Anwendung einer gegebenen Funktion auf ein gegebenes Argument durch.

$ ist rechtsassoziativ und hat unter allen Operationen die **niedrigste Priorität**. Daher kann durch Verwendung von $ manch schließende Klammer vermieden werden:

```
f1 $ f2 $ ... $ fn a   ⇝   f1 (f2 (...(fn a)...)))
```

Demgegenüber ist der **Kompositionsoperator**

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(g . f) a = g (f a)
```

links- und rechtsassoziativ und hat – nach den Funktionen in Präfixdarstellung – die **höchste Priorität**. Auch durch Verwendung von . kann manch schließende Klammer vermieden werden:

```
(f1 . f2 . ... . fn) a   ⇝   f1 (f2 (...(fn a)...)))
```

U.a. benutzt man den Kompositionsoperator, um in einer applikativen Definition Argumentvariablen einzusparen.

So sind z.B. die folgenden drei Definitionen einer Funktion $f : a \to b \to c$ äquivalent zueinander:

```
f a b = g (h a) b
f a   = g (h a)
f     = g . h
```

Beispiel

```
isNothing = not . isJust
```

Weitere polymorphe Funktionen, die Funktionen erzeugen bzw. verändern

```
id :: a -> a
id a = a
id   = \a -> a
```

Identität

```
const :: a -> b -> a
const a _ = a
const a   = \b -> a
const     = \a -> \b -> a
```

konstante Funktion

Der – auch **Wildcard** genannte – Unterstrich ist eine Individuenvariable (s.o), die nur auf der linken Seite einer Funktionsdefinition vorkommen darf, was zur Folge hat, dass ihr aktueller Wert den Wert der gerade definierten Funktion nicht beeinflussen kann.

```
update :: Eq a => (a -> b) -> a -> b -> a -> b
update f a b a' = if a == a' then b else f a'
```

Funktionsupdate (nicht im Prelude)

```
update(+2) 5 10 111+update(+2) 5 10 5 ⇝ 123
```

	Typ	*äquivalente Schreibweisen*
`update f`	`:: a -> b -> a -> b`	`update(f)`
`update f a`	`:: b -> a -> b`	`update(f)(a)` `(update f) a`
`update f a b`	`:: a -> b`	`update(f)(a)(b)` `((update f) a) b`
`update f a b a'`	`:: b`	`update(f)(a)(b)(a')` `(((update f) a) b) a'`

Link zur schrittweisen Auswertung von $update(+2)(5)(10)(111) + update(+2)(5)(10)(5)$

Jeder Schritt einer schrittweisen Auswertung eines Terms t besteht in der Anwendung aller jeweils anwendbaren Definitionsgleichungen auf alle maximalen Teilterme von t. Diese Auswertungsstrategie nennt man – wie schon bei der Auswertung von λ- Applikationen bemerkt wurde – **parallel-outermost** oder **lazy**.

Um die Zwischenergebnisse einer Auswertung nicht zu groß werden zu lassen, werden bei den hier verlinkten Berechnungsfolgen vor jeder Gleichungsanwendung alle Teilausdrücke ausgewertet, die nur aus Standardfunktionen, für die es keine Gleichungen gibt, und Konstanten bestehen.

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

`flip mod 11` *ist äquivalent zu* `(`mod`` `11)` *(s.o.)*

```
(&) :: a -> (a -> b) -> b
(&) = flip ($)
```
objektorientierte Applikation (Punktnotation)

```
p = Point {x=5.7, y=3.66}
p&x ⇝ 5.7

rect = DRect {center=p, width=11, height=5, color=Red}

rect&width    ⇝ 11
rect&center&x ⇝ 5.7
```

Link zur schrittweisen Auswertung von *rect&center&x*

(Attribute sind weiß unterlegt und als *Kanten*markierungen zu lesen. Die Attribute x und y sind groß geschrieben, weil der Painter sie sonst mit Variablen verwechselt.)

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f a b = f (a,b)
```
Kaskadierung (Currying)

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b
```
Dekaskadierung

```
lift :: (a -> b -> c)
     -> (state -> a) -> (state -> b) -> (state -> c)
lift op f g state = f state `op` g state
```
Operationslifting (nicht im Prelude)

Produktextension

```
(***) :: (a -> b) -> (a -> c) -> a -> (b,c)
(***) = lift (,)

lift (+) (+3) (*3) 5 ⇝ 23
((+3) *** (*3)) 5    ⇝ (8,15)
```

2.6 Rekursiv definierte Funktionen

$f : A \to B$ ist **rekursiv definiert**, wenn mindestens eine der Gleichungen für f auf ihrer rechten Seite (einen Aufruf von) f enthält.

Rekursive Definition der Fakultätsfunktion (nicht im Prelude)

```
fact :: Int -> Int
fact n = if n > 1 then n*fact (n-1) else 1
```

Mit Fallunterscheidung:

```
fact1 = \case 0 -> 1; n -> n*fact1(n-1)
```

Mit Fallunterscheidung und Guards:

```
fact2 = \case 0 -> 1; n | n > 0 -> n*fact2(n-1)
```

`fact 5` $\rightsquigarrow$ `120`

Während Anwendungen von *fact* und *fact1* auf negative Zahlen nicht terminieren, erhält man bei *fact2* die Fehlermeldung

```
Exception: . . . : Non-exhaustive patterns in case
```

Links zur schrittweisen Auswertung von *fact(5)*:

applikative Version; erste case-Version; zweite case-Version

$\lambda(p_1 \to t_1, \ldots, p_n \to t_n)$ steht hier für $\backslash case\ p_1 \to t_1; \ldots; p_n \to t_n$, wobei im Fall von guarded patterns $(\to)(p, b, t)$ für $p|b \to t$ steht.

Eine rekursive Definition heißt **iterativ** oder **endrekursiv** (*tail-recursive*), wenn keiner ihrer Aufrufe auf der rechten Seite einer Definitionsgleichung in eine andere Funktion (z.B. $(n*)$ bei *fact*) eingebettet ist. Da die einbettende Funktion erst angewendet werden kann, wenn der rekursive Aufruf ausgewertet ist, führen nicht-iterative Definitionen i.d.R. zu längeren Ausführungszeiten und höherem Platzbedarf für Zwischenergebnisse als iterative

Definitionen derselben Funktion.

Iterative Definition der Fakultätsfunktion

```
factI :: Int -> Int
factI n = loop 1 n

loop :: Int -> Int -> Int                                    Schleifenfunktion
loop state n = if n > 1 then loop (n*state) (n-1) else state
```

Der **Zustand** *state* (auch Akkumulator oder Schleifenvariable genannt) speichert Zwischenwerte.

Link zur schrittweisen Auswertung von *factI(5)*

Iterative Definitionen können direkt in eine imperative (zustandsorientierte) Sprache wie Java übersetzt werden. Aus den Parametern der Schleifenfunktion werden die rechten Seiten von Zuweisungen:

```
int factI(int n) {state = 1;
                  while n > 1 {state = n*state; n = n-1};
                  return state}
```

Diese und andere Programmtransformationen zur Entrekursivierung werden z.B. in [22], Kapitel 6 behandelt.

Rekursive Definition der Funktionsiteration (nicht im Prelude)

```
(^^^) :: (a -> a) -> Int -> a -> a
f^^^n = if n == 0 then id else f . (f^^^(n-1))
```

```
((+2)^^^4) 10 ⇝ 18
```

Link zur schrittweisen Auswertung von

```
((+2)^^^4) 10
```

Beispiel

Für jede Hue-Farbe $c \in RGB$ berechnet *complColor(c)* die Komplementärfarbe von c im oben definierten Farbkreis von 1530 Hue-Farben:

```
complColor :: RGB -> RGB
complColor = nextRGB^^^765
```

3 Listen

Sei A eine Menge. Die Elemente der Mengen

$$A^+ =_{def} \bigcup_{n>0} A^n \quad \text{und} \quad A^* =_{def} A^+ \cup \{\epsilon\}$$

heißen **Listen** oder **Wörter** über A. Wörter sind also n-Tupel, wobei $n \in \mathbb{N}$ beliebig ist. In Haskell schreibt man $[A]$ anstelle von A^* und für die Elemente dieses Typs $[a_1, \ldots, a_n]$ anstelle von $(a_1, \ldots, a_n)$.

$[A]$ bezeichnet den Typ der Listen, deren Elemente den Typ A haben.

Eine n-elementige Liste kann extensional oder als funktionaler Ausdruck dargestellt werden:

$$[a_1, \ldots, a_n] \quad \textit{ist äquivalent zu} \quad a_1 : (a_2 : (\ldots (a_n : []) \ldots))$$

Die Konstante $[]$ (leere Liste) vom Typ $[A]$ und die Funktion $(:)$ (*append*; Anfügen eines Elementes von A ans linke Ende einer Liste) vom Typ $A \to [A] \to [A]$ heißen – analog zum Tupelkonstruktor für kartesische Produkte (siehe Abschnitt 2.1) – **Listenkonstruktoren**, da sich mit ihnen jede Haskell-Liste darstellen lässt.

Die Klammern in $a_1 : (a_2 : (\ldots (a_n : []) \ldots))$ können weggelassen werden, weil der Typ von $(:)$ keine andere Klammerung zulässt.

Analog zu den Typinferenzregeln für Tupel, λ-Abstraktionen und λ-Applikationen in Kapitel 2 erhalten wir folgende Typinferenzregeln für Listenausdrücke:

$$\frac{}{[] :: [t]} \qquad \frac{e :: t, \quad e' :: [t]}{(e : e') :: [t]}$$

Die durch mehrere Gleichungen ausgedrückten Fallunterscheidungen bei den folgenden Funktionsdefinitionen ergeben sich aus verschiedenen **Mustern** der Funktionsargumente bzw. Bedingungen an die Argumente (Boolesche Ausdrücke hinter |).

Seien x, y, s Individuenvariablen. s ist ein Muster für alle Listen, [] das Muster für die leere Liste, $[x]$ ein Muster für alle einelementigen Listen, $x : s$ ein Muster für alle nichtleeren Listen, $x : y : s$ ein Muster für alle mindestens zweielementigen Listen, usw.

```
single :: a -> [a]
single a = [a]
```

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

`length [3,44,-5,222,29]` $\rightsquigarrow$ `5`

Link zur schrittweisen Auswertung von *length[3,44,-5,222,29]*

```
indices :: [a] -> [Int]
indices s = [0..length s-1]
```

indices [3,2,8,4] ⇝ [0..3]

Liste aller natürlichen Zahlen von 0 *bis* length s-1

```
null :: [a] -> Bool
null [] = True
null _  = False
```

null [3,2,8,4] ⇝ False

```
head :: [a] -> a
head (a:_) = a
```

head [3,2,8,4] ⇝ 3

```
tail :: [a] -> [a]
tail (_:s) = s
```

tail [3,2,8,4] ⇝ [2,8,4]

```
(++) :: [a] -> [a] -> [a]
(a:s)++s' = a:(s++s')
_++s      = s
```

[3,2,4]++[8,4,5] ⇝ [3,2,4,8,4,5]

```
(!!) :: [a] -> Int -> a
(a:_)!!0         = a
(_:s)!!n | n > 0 = s!!(n-1)
```

[3,2,4]!!1 ⇝ 2

```
init :: [a] -> [a]
init [_]   = []
init (a:s) = a:init s
```
init [3,2,8,4] ⇝ [3,2,8]

```
last :: [a] -> a
last [a]   = a
last (_:s) = last s
```
last [3,2,8,4] ⇝ 4

```
take :: Int -> [a] -> [a]
take 0 _              = []
take n (a:s) | n > 0 = a:take (n-1) s
take _ []             = []
```
take 3 [3,2,4,8,4,5] ⇝ [3,2,4]

```
drop :: Int -> [a] -> [a]
drop 0 s              = s
drop n (_:s) | n > 0 = drop (n-1) s
drop _ []             = []
```
drop 4 [3,2,4,8,4,5] ⇝ [4,5]

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile f (a:s) = if f a then a:takeWhile f s else []
takeWhile f _     = []
```
takeWhile (<4) [3,2,8,4] ⇝ [3,2]

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile f s@(a:s') = if f a then dropWhile f s' else s
dropWhile f _        = []
```

`dropWhile (<4) [3,2,8,4]` ⇝ `[8,4]`

```
updList :: [a] -> Int -> a -> [a]
updList s i a = take i s++a:drop (i+1) s
```

`updList [3,2,8,4] 2 9` ⇝ `[3,2,9,4]`

(nicht im Prelude)

```
reverse :: [a] -> [a]
reverse (a:s) = reverse s++[a]
reverse _     = []
```

`reverse [3,2,8,4]` ⇝ `[4,8,2,3]`

ist aufwändig wegen der Verwendung von `++`

Weniger aufwändig ist der folgende iterative Algorithmus, der die Werte von `reverse` *in einer Schleife akkumuliert:*

```
reverseI :: [a] -> [a]
reverseI = loop []

loop :: [a] -> [a] -> [a]
loop state (a:s) = loop (a:state) s
loop state _     = state
```

Link zur schrittweisen Auswertung von reverseI[2,4,5,7,88]

3.1 Listenteilung

... zwischen n-tem und $n+1$-tem Element:

```
splitAt :: Int -> [a] -> ([a],[a])      splitAt(3)[5..12]
splitAt 0 s                = ([],s)                ⇝ ([5,6,7],[8..12])
splitAt _ []               = ([],[])
splitAt n (a:s) | n > 0 = (a:s1,s2)                                    (1)
                          where (s1,s2) = splitAt (n-1) s
                                lokale Definition mit Pattern (s1,s2)
```

... beim ersten Element, das f nicht erfüllt:

```
span :: (a -> Bool) -> [a] -> ([a],[a])
span f s@(a:s') = if f a then (a:s1,s2) else ([],s)                    (2)
                  where (s1,s2) = span f s'
span _ _        = ([],[])
```

3.2 Lokale Definitionen sind λ-Applikationen

Die (bedingten) Gleichungen

$$t = t' \; where \; pat = u \quad \text{und} \quad t = (let \; pat = u \; in \; t')$$

sind beide äquivalent zu $t = (\lambda pat.t')(u)$, sofern *pat* und u keine gemeinsamen Variablen enthalten! Z.B. sind

```
splitAt n (a:s) | n > 0 = (\(s1,s2) -> (a:s1,s2))
                          $ splitAt (n-1) s
span f s@(a:s') = (\(s1,s2) -> if f a then (a:s1,s2) else ([],s))
                  $ span f s'
```

äquivalent zu (1) bzw. (2).

Link zur schrittweisen Auswertung von *splitAt(3)[5..12]*

In einer bewachten Definitionsgleichung

$$t \mid guard \;=\; t' \; where \; pat = u \tag{3}$$

darf *guard* Variablen von *pat* enthalten. In diesem Fall wird auch *guard* bei der Eliminierung der lokalen Defintion $p = u$ in eine Applikation eingebettet. Aus (3) wird

$$t \mid (\lambda pat.guard)(u) \;=\; (\lambda pat.t')(u).$$

3.3 Teillisten und Listenmischung

$sublist(s)(i)(j)$ berechnet die Teilliste von s vom i-ten bis zum j-ten Element:

```
sublist :: [a] -> Int -> Int -> [a]                    (nicht im Prelude)
sublist (a:_) 0 0                    = [a]
sublist (a:s) 0 j | j > 0            = a:sublist s 0 $ j-1
sublist (_:s) i j | i > 0 && j > 0 = sublist s (i-1) $ j-1
sublist _ _ _                        = []
```

$merge(s_1, s_2)$ mischt die Elemente von s_1 und s_2 so, dass das Ergebnis eine geordnete Liste ist, falls s_1 und s_2 geordnete Listen sind:

```
merge :: [Int] -> [Int] -> [Int]                       (nicht im Prelude)
merge s1@(x:s2) s3@(y:s4) | x < y = x:merge s2 s3
                          | x > y = y:merge s1 s4
                          | True  = merge s1 s4
merge [] s = s
merge s _  = s
```

3.4 Funktionslifting auf Listen

```
map :: (a -> b) -> [a] -> [b]
map f (a:s) = f a:map f s
map _ _     = []
```

```
map (+3) [2..9]             ⇝ [5..12]
map ($ 7) [(+1),(+2),(*5)] ⇝ [8,9,35]
map ($ a) [f1,f2,...,fn]   ⇝ [f1 a,f2 a,...,fn a]
```

Link zur schrittweisen Auswertung von *map(+3)[2..9]*

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:s) (b:s') = f a b:zipWith f s s'
zipWith _ _ _          = []
```

```
zipWith (+) [3,2,8,4] [8,9,35] ⇝ [11,11,43]
```

```
zip :: [a] -> [b] -> [(a,b)]
zip = zipWith (,)
```

```
zip [3,2,8,4] [8,9,35] ⇝ [(3,8),(2,9),(8,35)]
```

```
unzip :: [(a,b)] -> ([a],[b])
unzip ((a,b):s) = (a:s1,b:s2) where (s1,s2) = unzip s

unzip [(3,8),(2,9),(8,35)] ⇝ ([3,2,8],[8,9,35])
```

3.5 Strings sind Listen von Zeichen

Strings werden als Listen von Zeichen betrachtet, d.h. die Typen `String` und `[Char]` sind identisch.

Z.B. haben die folgenden Booleschen Ausdrücke den Wert *True*:

```
"" == []
"H" == ['H']
"Hallo" == ['H','a','l','l','o']
```

Also sind alle Listenfunktionen auf Strings anwendbar.

`words :: String -> [String]` und `unwords :: [String] -> String` zerlegen bzw. konkatenieren Strings, wobei Leerzeichen, Zeilenumbrüche (`'\n'`) und Tabulatoren (`'\t'`) als Trennsymbole fungieren.

`unwords` fügt Leerzeichen zwischen die zu konkatenierenden Strings.

`lines :: String -> [String]` und `unlines :: [String] -> String` zerlegen bzw. konkatenieren Strings, wobei nur Zeilenumbrüche als Trennsymbole fungieren.

`unlines` fügt `'\n'` zwischen die zu konkatenierenden Strings.

3.6 Listen mit Zeiger auf ein Element

```
type ListIndex a  = ([a],Int)
type ListZipper a = ([a],[a])
```

Liste und Index n

Zerlegung einer Liste s mit Index n in die **Kontextliste** $c = reverse(take(n)(s))$ *und die* **Suffixliste** $drop(n)(s)$

```
listToZipper :: ListIndex a -> ListZipper a
listToZipper = loop [] where
               loop :: [a] -> ([a],Int) -> ([a],[a])
               loop c (s,0)   = (c,s)
               loop c (a:s,n) = loop (a:c) (s,n-1)
```

`listToZipper ([1..9],4)` ⇝ `([4,3,2,1],[5..9])`

```
zipperToList :: ListZipper a -> ListIndex a
zipperToList (c,s) = loop c (s,0) where
                     loop :: [a] -> ([a],Int) -> ([a],Int)
                     loop (a:c) (s,n) = loop c (a:s,n+1)
                     loop _ sn        = sn
```

```
zipperToList ([4,3,2,1],[5..9]) ⇝ ([1..9],4)
```

`listToZipper` und `zipperToList` sind in folgendem Sinne invers zueinander:

$$\mathit{ListIndex}(A) \supseteq \{(s,n) \in A^+ \times \mathbb{N} \mid 0 \leq n < \mathit{length}(s)\} \cong A^* \times A^+ \subseteq \mathit{ListZipper}(A).$$

Zeigerbewegungen und Zugriff auf das indizierte Element

```
backI,forthI :: ListIndex a -> ListIndex a
backI (s,n) | n > 0          = (s,n-1)
forthI (s,n) | n < length s = (s,n+1)

getIndexedI :: ListIndex a -> a
getIndexedI (s,n) = s!!n
```

```
back,forth :: ListZipper a -> ListZipper a
back (a:c,s)  = (c,a:s)
forth (c,a:s) = (a:c,s)

getIndexed :: ListZipper a -> a
getIndexed (_,a:_) = a
```

Während `getIndexedI` die rekursive Funktion `(!!)` aufruft, also stets die gesamte Liste von ersten bis zum indizierten Element durchläuft, findet es `getIndexed` sofort am Anfang der jeweiligen Suffixliste.

3.7 Relationsdarstellung von Funktionen

Eine Funktion f mit endlichem Definitionsbereich lässt sich als Liste ihrer (Argument,Wert)-Paare implementieren und wäre damit vom Typ `[(arg,val)]`. Sie entspricht dem **Graphen von** f (siehe Abschnitt 6.4). Als programmiersprachliches Konstrukt wird sie oft als **Assoziationsliste** oder **Dictionary** bezeichnet.

Eine Applikation von f entspricht einem Zugriff auf ihre Listendarstellung. In Haskell erfolgt er mit der Standardfunktion *lookup*:

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup a ((a',b):r) = if a == a' then Just b else lookup a r
lookup _ _          = Nothing
```

Maybe wurde in Kapitel 2 eingeführt. Das Typklasse `Eq a` (siehe Kapitel 5) beschränkt die Instanzen von a auf Typen mit einer Funktion `(==) :: a -> a -> a`.

Mit `[ ]` anstelle von *Maybe* geht es auch:

```
lookupL :: Eq a => a -> [(a,b)] -> [b]
lookupL a ((a',b):r) = if a == a' then b:lookup a r else lookup a r
lookupL _ _          = []
```

Angewendet auf eine Relation R, die keine Funktion darstellt, wenn es also $(a, b), (a, b') \in R$ mit $b \neq b'$ gibt, liefert *lookupL* die Liste aller b mit $(a, b) \in R$ und nicht nur das erste b - wie es *lookup* tut.

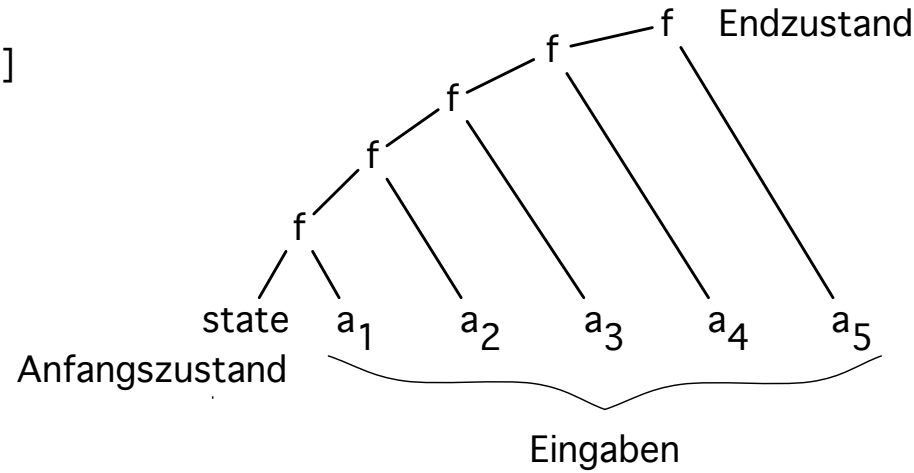
Die folgende Funktion *updRel* überträgt die Funktion *update* von Kapitel 2 auf die Listendarstellung von Funktionen:

```
updRel :: Eq a => [(a,b)] -> a -> b -> [(a,b)]          (nicht im Prelude)
updRel ((a,b):r) c d = if a == c then (a,d):r else (a,b):updRel r c d
updRel _ a b         = [(a,b)]
```

3.8 Listenfaltung

Faltung einer Liste von links her

foldl f state [a_1,a_2,a_3,a_4,a_5]




```
foldl :: (state -> a -> state) -> state -> [a] -> state
foldl f state (a:as) = foldl f (f state a) as
foldl _ state _      = state
```

f ist Zustandsüberführung

```
sum     = foldl (+) 0
product = foldl (*) 1
and     = foldl (&&) True
or      = foldl (||) False
concat  = foldl (++) []
```

`sum[2..9]` ⇝ `44`

Link zur schrittweisen Auswertung von *sum[2..9]*

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f
```

```
reverseF :: [a] -> [a]
reverseF = foldl (flip (:)) []
```

Listenrevertierung als Faltung

```
factF :: Int -> Int
factF 0 = 1
factF n = product [2..n]
```

Die Fakultätsfunktion als Faltung

Faltung nichtleerer Listen im Fall $state = a$:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (a:as) = foldl f a as

minimum = foldl1 min
maximum = foldl1 max
```

Beispiel Horner-Schema

Die Werte eines reellwertigen Polynoms

$$\lambda x. \sum_{i=0}^{n} a_i * x^i$$

können durch Faltung der Koeffizientenliste $as = [a_n, \ldots, a_0]$ berechnet werden:

$$horner(as)(x) \;=\; ((\ldots(a_n * x + a_{n-1}) * x \ldots) * x + a_1) * x + a_0$$

```
horner :: [Float] -> Float -> Float
horner as x = foldl1 f as where f state a = state*x+a
```

Beispiel Binomialkoeffizienten

$$\binom{n}{i} \;=\; \frac{n!}{i!(n-i)!} \;=\; \frac{(i+1) * \cdots * n}{(n-i)!}$$

```
binom :: Int -> Int -> Int
binom n i = product[i+1..n] `div` product[1..n-i]
```

Die induktive Definition der Binomialkoeffizienten:

$$\forall\, n \in \mathbb{N}: \quad \binom{n}{0} = \binom{n}{n} = 1$$

$$\forall\, n \in \mathbb{N}\ \ \forall\, i < n: \quad \binom{n}{i} = \binom{n-1}{i-1} + \binom{n-1}{i}$$

liefert folgendes Programm:

```
binom :: Int -> Int -> Int
binom n 0           = 1
binom n i | i == n = 1
          | i < n  = binom (n-1) (i-1) + binom (n-1) i
```

Daraus ergibt sich, dass $\binom{n}{i}$ die Anzahl der i-elementigen Teilmengen einer n-elementigen Menge ist. Daher kann die Anzahl der Partitionen einer n-elementigen Menge ebenfalls induktiv berechnet werden:

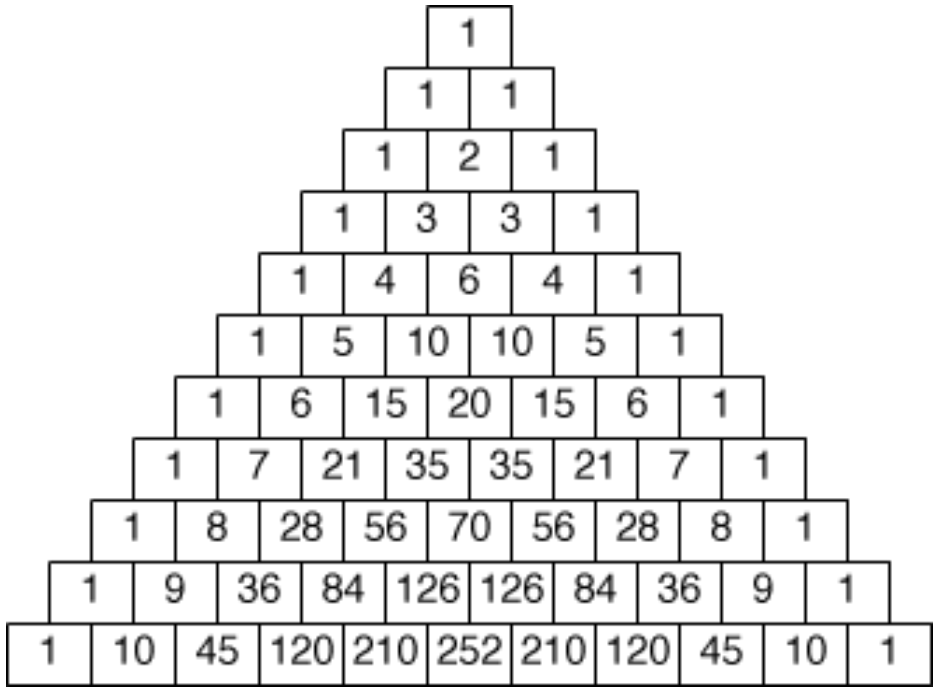
$$\begin{aligned} & partsno(0) = 1 \\ \forall\, n > 0: \ & partsno(n) = \textstyle\sum_{i=0}^{n-1} \binom{n-1}{i} * partsno(i) \end{aligned}$$

```
partsno :: Int -> Int
partsno 0 = 1
partsno n = sum $ map f [0..n-1] where f i = binom (n-1) i*partsno i
```

```
map partsno [1..10] ⇝ [1,2,5,15,52,203,877,4140,21147,115975]
```

Es gilt also $partsno(n) = length(parts[1..n])$.

Außerdem ist $\binom{n}{i}$ das i-te Element der n-ten Zeile des **Pascalschen Dreiecks**:



Unter Verwendung der induktiven Definition von $\binom{n}{i}$ ergibt sich die n-te Zeile wie folgt aus der $(n-1)$-ten Zeile:

```
pascal :: Int -> [Int]
pascal 0 = [1]
pascal n = zipWith (+) (s++[0]) $ 0:s where s = pascal $ n-1
```

Die obige Darstellung des Pascalschen Dreiecks wurde mit Expander2 aus dem Wert von $f(10)$ erzeugt, wobei

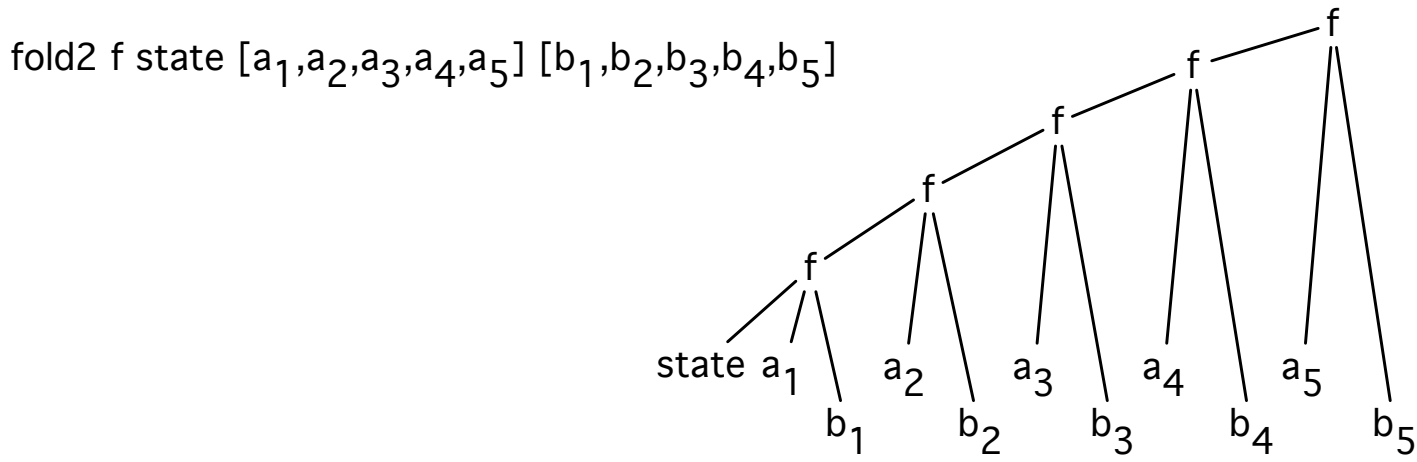
$$f \;=\; \lambda n.\mathit{shelf}\,(1)\$\mathit{map}(\mathit{shelf}\,(n+1)\circ\mathit{map}(\mathit{frame}\circ\mathit{text})\circ\mathit{pascal})[0..n].$$

text(n) fasst n als String auf. *frame(x)* rahmt den String x. *shelf(n)(s)* teilt die Liste s in Teillisten der Länge n auf, deren jeweilige Elemente horizontal aneinandergefügt werden. (Die letzte Teilliste kann kürzer sein.) Dann werden die Teillisten zentriert gestapelt.

$f(n)$ wendet zunächst *shelf(n+1)* auf jede Zeile des Dreiecks $\mathit{map}(\mathit{pascal})[0..n]$ an. Da jede Zeile höchstens $n+1$ Elemente hat, werden also *alle* Zeilenelemente horizontal aneinandergefügt. *shelf(1)* teilt dann die Liste aller Zeilen in Teillisten der Länge 1 auf, was bewirkt, dass die Zeilen zentriert gestapelt werden.

Link zur schrittweisen Auswertung von $f(5)$ bis zu dem Ausdruck, den Expander2 über seine Haskell-Schnittstelle zu Tcl/Tk in das $f(5)$ entsprechende Dreieck übersetzt.

Parallele Faltung zweier Listen von links her *(nicht im Prelude)*




```
fold2 :: (state -> a -> b -> state) -> state -> [a] -> [b] -> state
fold2 f state (a:as) (b:bs) = fold2 f (f state a b) as bs
fold2 _ state _ _           = state
```

Beispiel

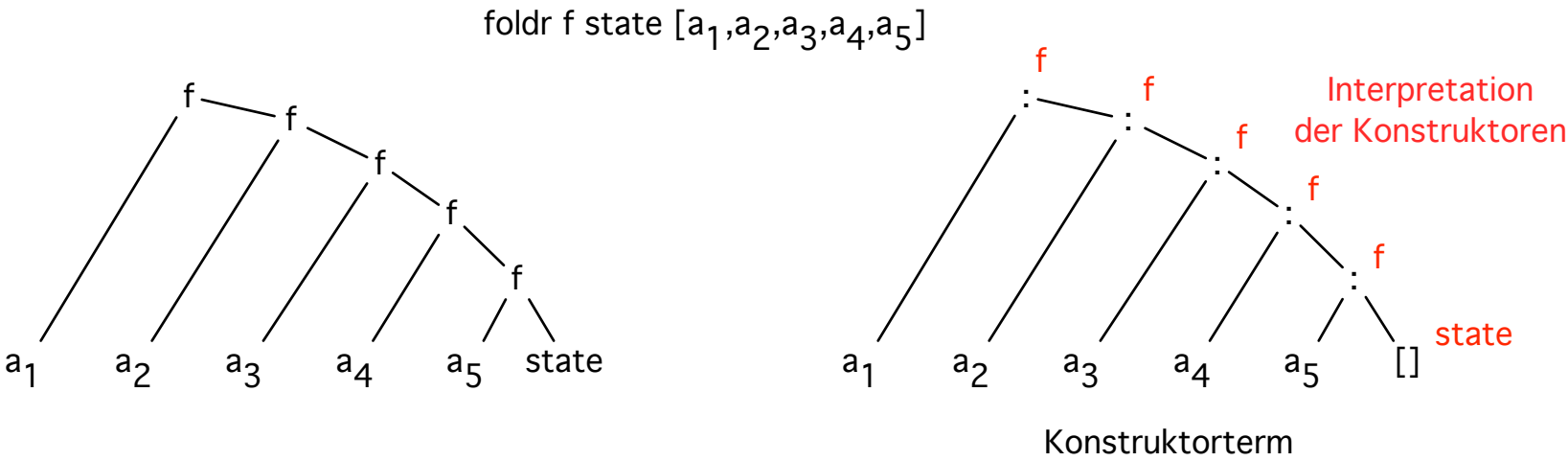
```
rel2fun :: Eq a => [(a,b)] -> a -> b
rel2fun r@((a,b):r') = fold2 update (const b) as bs where (as,bs) = u
```

rel2fun(r) transformiert jede nichtleere Assoziationsliste r in die jeweilige Funktion, die sie repräsentiert. *rel2fun(r)(a)* ist die zweite Komponente des *letzten* Paares von r mit erster Komponente a, falls r ein solches Paar enthält:

$$
\begin{aligned}
rel2fun(r)(a) &= \begin{cases} snd(r!!i) & \text{falls } fst(r!!i) = a \ \wedge \ \forall\, k > i : fst(r!!k) \neq a, \\ snd(head(r)) & \text{sonst} \end{cases} \\
&= fromJust(lookup(a)(reverse(r))) \qquad \text{(siehe Abschnitt 3.7)}
\end{aligned}
$$

fst und *snd* bilden (a, b) auf a bzw. b ab.

Faltung einer Liste von rechts her




```
foldr :: (a -> state -> state) -> state -> [a] -> state
foldr f state (a:as) = f a $ foldr f state as
foldr _ state _      = state
```

Beispiel

Sammeln der Werte desselben Arguments in einer Assoziationsliste (vgl. 3.7):

```
rel2funL :: (Eq a,Eq b) => [(a,b)] -> a -> [b]
rel2funL = foldr g $ const []
           where g (a,b) f = update f a $ insert b $ f a
```

Stets liefert *foldr(f)(st)(as)* den Wert von *as* unter den durch $st : state$ bzw. $f : a \to state \to state$ gegebenen Interpretationen der Konstruktoren $[] : [a]$ und $(:) : a \to [a] \to [a]$.

In entsprechender Weise können Faltungen auf den Elementen anderer Datentypen durchgeführt werden: Zugrunde liegt immer eine Interpretation der jeweiligen Konstruktoren (siehe Abschnitt 5.10).

Faltung nichtleerer Listen:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [a] = a
foldr1 f (a:as) = f a $ foldr1 f as
```

Strikte Faltung von links her

Ist $f :: state \to a \to state$ **strikt**, d.h. werden zur Berechnung von $f(state)(a)$ die Werte von $state$ **und** a benötigt, dann erzeugt der Aufruf $foldl(f)(state)[a_1, \ldots, a_n]$ zunächst den Ausdruck

$$f(f(\ldots(f(f(state, a_1), a_2), \ldots), a_{n-1}), a_n), \tag{1}$$

bevor dieser von innen nach außen ausgewertet wird
(siehe z.B. die Auswertung von $sum[2..9]$)

Abhilfe schafft die Funktion *foldl'* des Moduls *Data.List*:

```
foldl' :: (state -> a -> state) -> state -> [a] -> state
foldl' f state (a:as) = seq state' $ foldl' f state' as
                        where state' = f state a
foldl' _ state _      = state
```

$seq :: a \to b \to b$ ist eine Standardfunktion. Bei der Auswertung eines Ausdrucks der Form $seq(a)(g(a))$ wird zunächst der Wert val von a berechnet und anschließend der von $g(val)$. Dieser wird dann von $seq(a)(g(a))$ zurückgegeben.

$foldl'(f)(state)[a_1, \ldots, a_n]$ erzeugt also nicht den einen großen Ausdruck (1), sondern eine Folge kleinerer Ausdrücke, die hintereinander ausgewertet werden:

$$f(state, a_1) \rightsquigarrow state_1,\ f(state_1, a_2) \rightsquigarrow state_2,\ \ldots,\ f(state_{n-1}, a_n) \rightsquigarrow state_n \tag{2}$$

$state_n$ ist aber auch der Wert von (1).

Link zur Auswertung von *sum[2..9]* mit strikter Faltung

Beispiel (mit Laufzeiten, die vom ghci-Kommando `:set +s` ausgegeben werden)

```
foldl (+) 0 [1..1000000]  ⇝  500000500000      -- 0.76 secs
foldl' (+) 0 [1..1000000]                      -- 0.12 secs
foldr (+) 0 [1..1000000]                       -- 0.28 secs
```

Der Applikationsoperator ($) als Parameter von Listenfunktionen

```
foldr ($) a [f1,f2,f3,f4]          ⇝  f1 $ f2 $ f3 $ f4 a

foldl (flip ($)) a [f4,f3,f2,f1]   ⇝  f1 $ f2 $ f3 $ f4 a

map f [a1,a2,a3,a4]                ⇝  [f a1,f a2,f a3,f a4]
map ($a) [f1,f2,f3,f4]             ⇝  [f1 a,f2 a,f3 a,f4 a]

zipWith ($) [f1,f2,f3,f4] [a1,a2,a3,a4]
                                   ⇝  [f1 a1,f2 a2,f3 a3,f4 a4]
```

3.9 Teillisten, Permutationen, Partitionen, Kantenzüge

Liste aller Teillisten einer Liste

```
sublists :: [a] -> [[a]]
sublists []     = [[]]
sublists (a:as) = lists ++ map (a:) lists where
                  lists = sublists as
```

```
sublists[1..4] ⇝
      [[],[4],[3],[3,4],[2],[2,4],[2,3],[2,3,4],[1],[1,4],[1,3],
       [1,3,4],[1,2],[1,2,4],[1,2,3],[1,2,3,4]]
```

Liste aller Permutationen einer Liste, rekursiv (`perms`) und iterativ (`permsL`)

```
perms,permsI :: [a] -> [[a]]
perms [] = [[]]
perms s  = concatMap f $ indices s where
           f i = map (s!!i:) $ perms $ take i s++drop (i+1) s
permsI s = loop s [] where
           loop [] s = [s]
           loop s s' = concatMap f $ indices s where
                       f i = loop (take i s++drop (i+1) s) $ s!!i:s'
```

Die einbettende Funktion `(s!!i:)` macht **perms** rekursiv (siehe Abschnitt 2.6).

```
perms[1..3]  ⇝ [[3,2,1],[2,3,1],[3,1,2],[1,3,2],[2,1,3],[1,2,3]]
perms[5,6,5] ⇝ [[5,6,5],[6,5,5],[5,5,6],[5,5,6],[6,5,5],[5,6,5]]
```

Liste aller Partitionen (Zerlegungen) einer Liste

```
partsL :: [a] -> [[[a]]]
partsL [a]   = [[[a]]]
partsL (a:s) = concatMap glue $ partsL s
               where glue :: [[a]] -> [[[a]]]
                     glue part@(s:rest) = [[a]:part,(a:s):rest]
```

```
partsL[1..4] ⇝
  [[[1],[2],[3],[4]],[[1,2],[3],[4]],[[1],[2,3],[4]],[[1,2,3],[4]],
   [[1],[2],[3,4]],[[1,2],[3,4]],[[1],[2,3,4]],[[1,2,3,4]]]
```

Demnach ist 2^{n-1} die Anzahl der Zerlegungen einer n-elementigen Liste.

Liste aller Partitionen einer Menge (in Listendarstellung)

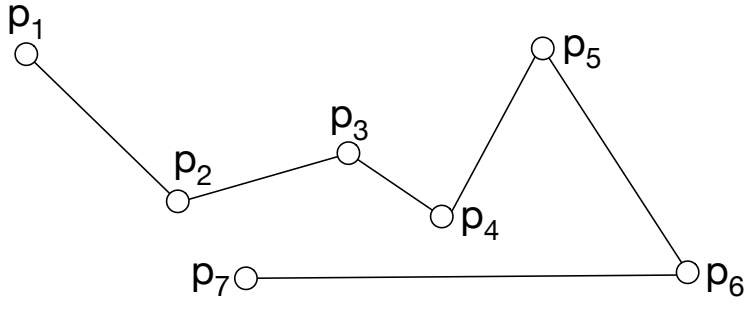
```
parts :: [a] -> [[[a]]]
parts [a]   = [[[a]]]
parts (a:s) = concatMap (glue []) $ parts s
```

```
        where glue :: [[a]] -> [a] -> [[[a]]]
              glue part (s:rest) = ((a:s):part++rest):
                                   glue (s:part) rest
              glue part _        = [[a]:part]
```

```
parts[1..4] ⇝
  [[[1,2,3,4]],[[1],[2,3,4]],[[1,2],[3,4]],[[1,3,4],[2]],
    [[1],[3,4],[2]],[[1,2,3],[4]],[[1,4],[2,3]],[[1],[4],[2,3]],
    [[1,2,4],[3]],[[1,3],[2,4]],[[1],[3],[2,4]],[[1,2],[4],[3]],
    [[1,4],[2],[3]],[[1,3],[4],[2]],[[1],[3],[4],[2]]]
```

Die Anzahl der Zerlegungen einer n-elementigen Menge ist kombinatorisch (siehe oben Beispiel Binomialkoeffizienten).

Kantenzüge als Punktlisten



$[p_1,p_2,p_3,p_4,p_5,p_6,p_7]$

```
type Path  = [(Float,Float)]
```
Typ für Punktlisten

```
lengthPath :: Path -> Float
lengthPath ps = sum $ zipWith distance ps $ tail ps
```
Länge eines Kantenzugs *siehe 2.5*

```
area :: Path -> Float
area ps = abs (sum $ zipWith f (last ps:init ps) ps)/2
```
Fläche eines Polygons (geschlossenen Kantenzugs)

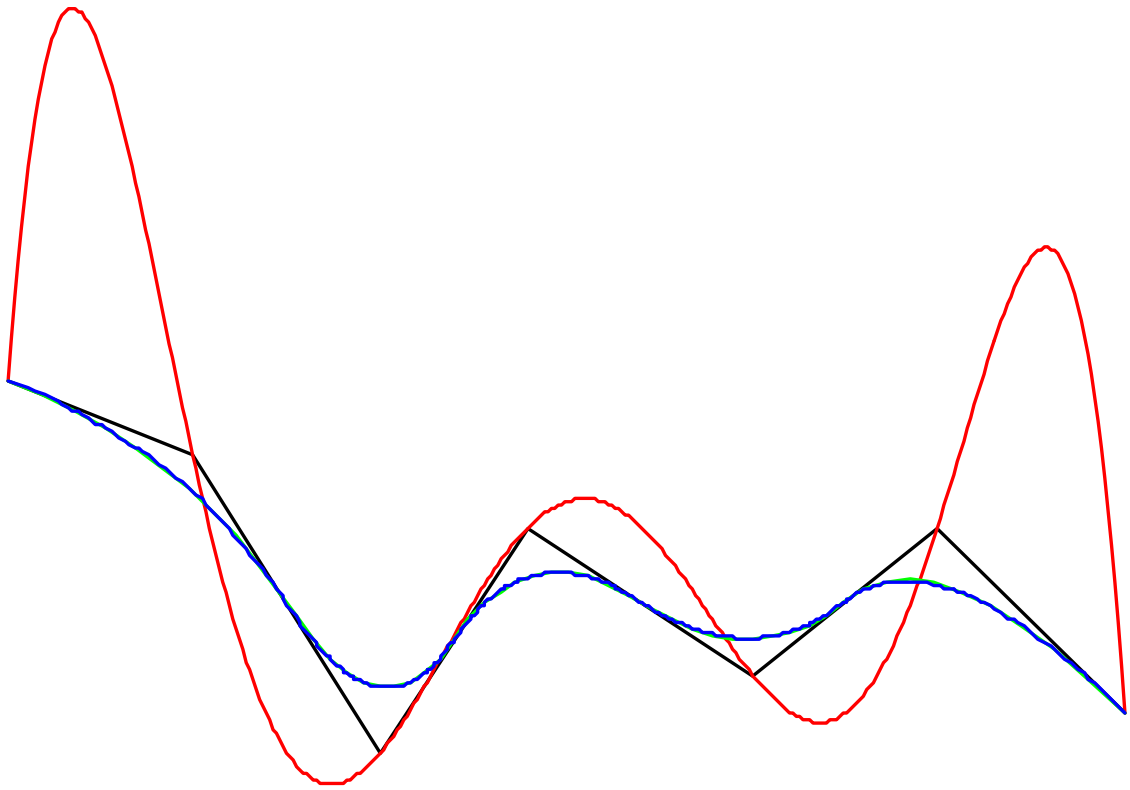
Polynominterpolation nach Lagrange (siehe hier)

```
context :: Int -> [a] -> [a]
context i s = take i s++drop (i+1) s
```
Streiche i-tes Element einer Liste

```
curve :: Path -> Float -> Float
curve ps x = sum $ map f $ indices ps where
             f i = yi*product [(x-c)/(xi-c) | (c,_) <- context i ps]
                   where (xi,yi) = ps!!i
```
curve(ps) trifft alle Punkte von ps

curve(ps) scheitert, falls *ps* keine rechtseindeutige Relation ist!

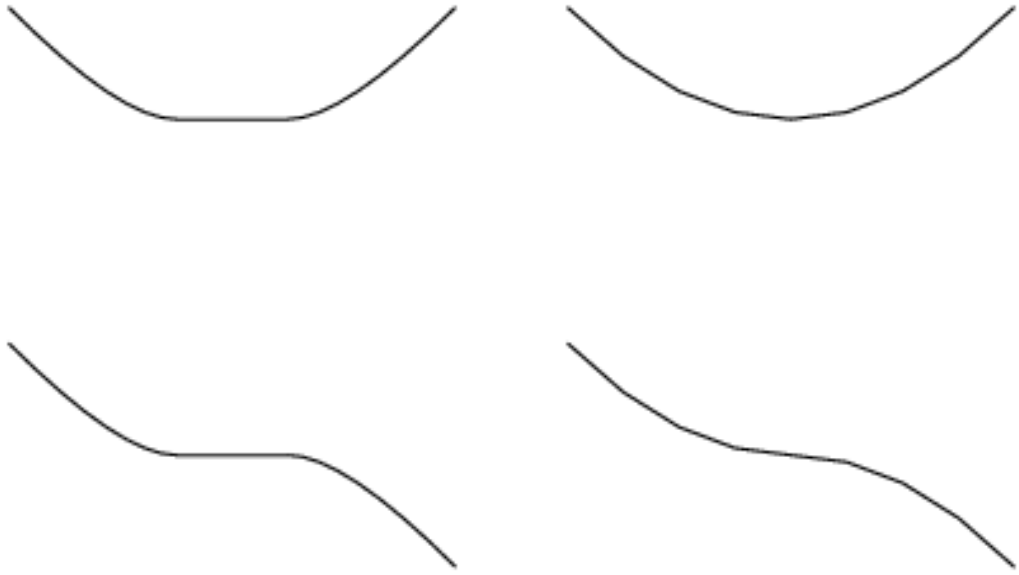
Ein aus 7 Punkten bestehender Kantenzug *ps* (schwarz), *curve(ps)* (rot) und die Glättung von *ps* (blau):



Die folgende Funktion minimiert einen Kantenzug *ps*, indem sie aus ihm jeden Punkt entfernt, der zusammen mit seinen linken und rechten Nachbarn auf einer Geraden liegt. So werden bei der Glättung von *ps* unerwünschte Plateaus vermieden:

```
minimize :: Path -> Path
minimize (p:ps@(q:r:s)) | straight p q r = minimize $ p:r:s
                        | True           = p:minimize ps
minimize ps                              = ps
```

Der rechte Kantenzug wurde vor der Minimierung geglättet, der linke nicht:



3.10 Listenlogik

```
any :: (a -> Bool) -> [a] -> Bool
any f = or . map f
```
`any (>4) [3,2,8,4]` ⇝ `True`

```
all :: (a -> Bool) -> [a] -> Bool
all f = and . map f
```
`all (>2) [3,2,8,4]` ⇝ `False`

```
elem :: Eq a => a -> [a] -> Bool
elem a = any (a ==)
```
`elem 2 [3,2,8,4]` ⇝ `True`

```
notElem :: Eq a => a -> [a] -> Bool
notElem a = all (a /=)
```
`notElem 9 [3,2,8,4]` ⇝ `True`

```
filter :: (a -> Bool) -> [a] -> [a]
filter f (a:s) = if f a then a:filter f s else filter f s
filter f _     = []
```
`filter (<8) [3,2,8,4]` ⇝ `[3,2,4]`

```
card :: Eq a => [a] -> a -> Int
card s a = length $ filter (== a) s
```
Anzahl der Vorkommen von a in s

```
isPerm :: Eq a => [a] -> [a] -> Bool
isPerm s s' = f s == f s' where          Ist s eine Permutation von s'?
              f s0 = map (card s0) $ s++s'
```

3.11 Listenkomprehension

Mit Listenfunktionen aufgebaute Ausdrücke lassen sich häufig als **Listenkomprehension** darstellen. So liefert z.B.

```
map(f)(filter(g)(s))
```

dieselbe Liste wie die Komprehension

```
[f a | a <- s, g a].
```
(1)

Umgekehrt entsprechen die Werte von Listenfunktionen Komprehensionen:

```
map f s         = [f a | a <- s]
filter g s      = [a | a <- s, g a]
concat s        = [a | s' <- s, a <- s']
concatMap f s   = [b | a <- s, s' <- f a, b <- s']
zipWith f s s' = [f a b | (a,b) <- zip s s']
```

Allgemeines Schema einer Listenkomprehension

$$[e \mid e_1, \ldots, e_n] \;::\; [a] \tag{2}$$

e ist ein Ausdruck des Typs a und für alle $1 \leq i \leq n$ ist e_i

- ein Generator, d.h. von der Form $p \leftarrow e'$, wobei e' den Typ $[b]$ hat, falls p ein Muster vom Typ b ist, oder
- ein Guard, also ein Boolescher Ausdruck, oder
- eine lokale Definition $let\ p = e'$, wobei – wie bei anderen lokalen Definitionen – p ein Muster und e' ein Ausdruck desselben Typs ist.

Die Ausdrücke $e_1, \ldots, e_n$ werden nacheinander ausgewertet:

Ist e_i ein Generator $p \leftarrow e'$ oder eine lokale Definition $let\ p = e'$, dann werden die Variablen von p erst bei der Auswertung von e' mit Werten belegt. Diese Variablen sollten deshalb in $e_1, \ldots, e_{i-1}$ nicht vorkommen.

Während die Auswertung von $let\ p = e'$ genau eine Variablenbelegung liefert, erzeugt $p \leftarrow e'$ für jedes Element a der Liste (!) e' eine Variablenbelegung. Die mit diesen Variablenbelegungen gebildeten Instanzen von e liefern schließlich die Elemente der Komprehension $[e \mid e_1, \ldots, e_n]$.

Kartesische Produkte und Relationen lassen sich oft als Komprehensionen darstellen, z.B.:

```
binprod :: [a] -> [b] -> [(a,b)]
binprod s1 s2 = [(a,b) | a <- s1, b <- s2]
```

repräsentiert das binäre Produkt $s1 \times s2$

```
terrel :: [a] -> [b] -> [c] -> ([a] -> [b] -> [c]) -> [(a,b,c)]
terrel f s1 s2 s3 = [(a,b,c) | a <- s1, b <- s2, c <- s3, f a b c]
```

repräsentiert die ternäre Relation $\{(a,b,c) \in s1 \times s2 \times s3 \mid f(a)(b)(c) = True\}$

Das letzte Beispiel zeigt den Unterschied zu in der Mathematik verwendeten *Mengen*komprehensionen: Generatoren stehen dort manchmal *vor* den senkrechten Strich (wann?) und das Elementsymbol $\in$ ersetzt den Pfeil $\leftarrow$.

`[f a | a <- s]`	*repräsentiert die Menge* $\{f(a) \mid a \in s\}$,
`[a | a <- s, g a]`	*repräsentiert die Menge* $\{a \in s \mid g(a)\}$,
`[a | a <- s, h a `elem` s']`	*repräsentiert die Menge* $\{a \in s \mid h(a) \in s'\}$.

Beispiel Minima einer Liste bzgl. einer Halbordnung *le* berechnen:

```
minis :: Eq a => (a -> a -> Bool) -> [a] -> [a]
minis le s = [a | a <- s, all (not . flip le a) $ remove a s]
```

Beispiel Kryptogramm = Kodierung von Zeichen durch Ziffern, die Gleichungen lösen, hier: *send+more=money*.

Jede Kodierung ist gegeben als Zuordung der Liste [s,e,n,d,m,o,r,y] ganzzahliger Variablen zu einer Permutation der Liste [0..9], die folgende Bedingung erfüllt:

```
correct :: [Int] -> Bool
correct [s,e,n,d,m,o,r,y] = s /= 0 && m /= 0 &&
                            1000*(s+m)+100*(e+o)+10*(n+r)+d+e ==
                        10000*m+1000*o+100*n+10*e+y
```

Die folgende Funktion berechnet alle korrekten Kodierungen. Tatsächlich gibt es genau eine (die allerdings mehrfach berechnet wird):

```
codes :: [[(Char,Int)]]
codes = [zip "sendmory" nums | nums <- perms [0..9],
                               correct $ take 8 nums]
```

```
head codes ⇝
     [('s',9),('e',5),('n',6),('d',7),('m',1),('o',0),('r',8),('y',2)]
```

Test einer Kodierung auf Korrektheit:

```
test :: [(Char,Int)] -> Bool
test code = correct $ map (fromJust . flip lookup code) "sendmory"
```

```
test [('s',9),('e',5),('n',6),('d',7),('m',1),('o',0),('r',8),('y',2)]
     ⇝ True
```

Beispiel Zebra- oder Einsteinrätsel

Fünf Häuser haben unterschiedliche Farben, werden von Menschen unterschiedlicher Nationalität bewohnt, die unterschiedliche Getränke und Zigarettenmarken bevorzugen sowie verschiedene Haustiere haben. Aus den folgenden Bedingungen lässt sich eine eindeutige Zuordnung aller fünf Attribute zu den einzelnen Häusern berechnen.

01. The Englishman lives in the red house.
02. The Spaniard owns the dog.
03. Coffee is drunk in the green house.
04. The Ukrainian drinks tea.
05. The green house is immediately to the right of the white house.
0 6. The Old Gold smoker owns snails.
07. Kools are smoked in the yellow house.
08. Milk is drunk in the middle house.
09. The Norwegian lives in the first house.
10. The man who smokes Chesterfields lives in the house next to the man with the fox.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.

13. The Japanese smokes Dunhill.
14. The Norwegian lives next to the blue house.

Im Kryptogramm-Beispiel war eine Lösung eine eindeutige Zuordnung der Buchstaben s,e,n,d,m,o,r,y zu den Ziffern 0 bis 9. Hier ist sie eine 5x5-Matrix, dargestellt als Liste von Listen:

$$[color, nation, drink, smoke, pet] :: [[String]]$$

Die Listen *color,nation,drink,smoke,pet* sind Permutationen der Listen

```
colors  = ["Red      ","Green    ","White    ","Yellow   ","Blue     "]
nations = ["British  ","Spaniard ","Ukrainian","Japanese ","Norwegian"]
drinks  = ["Coffee   ","Tea      ","Milk     ","Juice    ","Water    "]
smokes  = ["Gold     ","Chester  ","Kools    ","Lucky    ","Dunhill  "]
pets    = ["Dog      ","Snails   ","Fox      ","Horse    ","Zebra    "]
```

Die Spalten der Matrix repräsentieren die fünf Häuser. Hat z.B. *color!!2* den Wert `"Red` dann ist das dritte Haus rot.

Lösungen sind solche Matrizen, die obige 14 Bedingungen erfüllen. Bei deren Implementierung benutzen wir folgende Hilfsprädikate, die Einträge der Matrix vergleichen:

```
firstHouse,middleHouse :: Eq a => a -> [a] -> Bool
firstHouse  x xs = head xs == x
middleHouse x xs = xs!!2 == x
```

```
sameHouse,rightHouse,nextHouse :: Eq a => a -> [a] -> a -> [a] -> Bool
sameHouse  x xs y ys = (x,y) `elem` zip xs ys
rightHouse x xs y ys = sameHouse x (tail xs) y ys
nextHouse  x xs y ys = rightHouse x xs y ys || rightHouse y ys x xs
```

Den obigen Bedingungen entsprechen dann folgende Funktionen:

```
cond1  color nation = sameHouse   "Red      " color  "British  " nation
cond2  nation pet   = sameHouse   "Spaniard " nation "Dog      " pet
cond3  color drink  = sameHouse   "Green    " color  "Coffee   " drink
cond4  nation drink = sameHouse   "Ukrainian" nation "Tea      " drink
cond5  color        = rightHouse  "Green    " color  "White    " color
cond6  smoke pet    = sameHouse   "Gold     " smoke  "Snails   " pet
cond7  color smoke  = sameHouse   "Yellow   " color  "Kools    " smoke
cond8  drink        = middleHouse "Milk     " drink
cond9  nation       = firstHouse  "Norwegian" nation
cond10 smoke pet    = nextHouse   "Chester  " smoke  "Fox      " pet
cond11 smoke pet    = nextHouse   "Kools    " smoke  "Horse    " pet
cond12 drink smoke  = sameHouse   "Juice    " drink  "Lucky    " smoke
cond13 nation smoke = sameHouse   "Japanese " nation "Dunhill  " smoke
cond14 color nation = nextHouse   "Blue     " color  "Norwegian" nation
```

Demnach liefert die folgende Komprehension diejenigen Matrizen, die alle 14 Bedingungen erfüllen:

```
[[color,nation,drink,smoke,pet] |
 color <- perms colors,
 cond5 color,
 nation <- perms nations,
 cond1 color nation && cond9 nation && cond14 color nation,
 drink <- perms drinks,
 cond3 color drink && cond8 drink && cond4 nation drink,
 smoke <- perms smokes,
 cond7 color smoke && cond12 drink smoke && cond13 nation smoke,
 pet <- perms pets,
 cond2 nation pet && cond6 smoke pet && cond10 smoke pet &&
                     cond11 smoke pet]
```

Die einzige Lösung lautet wie folgt:

```
house  | 1         2         3         4         5
color  | Yellow    Blue      Red       White     Green
nation | Norwegian Ukrainian British   Spaniard  Japanese
drink  | Water     Tea       Milk      Juice     Coffee
```

```
smoke  | Kools     Chester   Gold      Lucky     Dunhill
pet    | Fox       Horse     Snails    Dog       Zebra
```

Das gesamte Programm einschließlich einer Funktion *showMat*, die Lösungen in obiger Form ausgibt, und einer Funktion *showRel*, die sie in dreistellige Relationen umwandelt, die dann mit Expander2 wie unten dargestellt werden, steht hier.

	1	2	3	4	5
pet	Fox	Horse	Snails	Dog	Zebra
color	Yellow	Blue	Red	White	Green
drink	Water	Tea	Milk	Juice	Coffee
smoke	Kools	Chester	Gold	Lucky	Dunhill
nation	Norwegian	Ukrainian	British	Spaniard	Japanese

Man beachte, dass es $(5!)^5$ 5x5-Matrizen gibt, deren Zeilen Permutationen jeweils einer der obigen fünf Listen ist. Für die Berechnung der Lösungen in vertretbarer Zeit ist daher die *Anordnung* von Generatoren und Guards in der Komprehension entscheidend. Sie muss dem Branch-and-bound-Prinzip folgen, das hier fordert, Guards, wann immer möglich, Generatoren voranzustellen. So werden z.B. in obiger Komprehension zunächst nur diejenigen Farbzuordnungen zu den fünf Häusern berechnet, die Bedingung 5 erfüllen. Nur sie werden an darauffolgende Generatoren weitergereicht und mit Zuordnungen anderer Attribute kombiniert, die dann auch verworfen werden, sobald sie einen Guard verletzen. ❏

```
prod :: [[a]] -> [[a]]
prod []       = [[]]
prod (s:s') = [a:as | a <- s, as <- prod s']
```

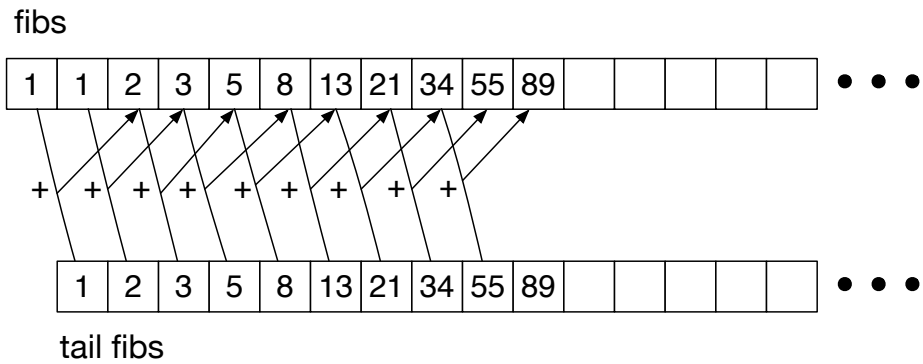
oder effizienter mit lokaler Definition:

```
prod (s:s') = [a:as | a <- s, as <- ps'] where ps' = prod s'
prod[[1,2],[3,4],[5..7]] ⇝
[[1,3,5],[1,3,6],[1,3,7],[1,4,5],[1,4,6],[1,4,7],
 [2,3,5],[2,3,6],[2,3,7],[2,4,5],[2,4,6],[2,4,7]]
```

3.12 Unendliche Listen (Folgen, Ströme) entsprechen Funktionen auf $\mathbb{N}$
(Haskell-Modul: Lazy.hs)

```
blink :: [Int]
blink = 0:1:blink                    blink  ⇝  0:1:0:1:...

nats :: Int -> [Int]
nats n = n:map (+1) (nats n)         nats 3  ⇝  3:4:5:6:...
                                     nats n   ist äquivalent zu   [n..]
```

```
fibs :: [Int]                                   Fibonacci-Folge
fibs = 1:1:zipWith (+) fibs (tail fibs)
```




```
take 11 fibs  ⇝  [1,1,2,3,5,8,13,21,34,55,89]
```

Link zur Auswertung von *(take(4)(fibs), fibs!!3)*

Mit **pattern binding** an Stelle von `tail`:

```
fibs@(1:tfibs) = 1:1:zipWith (+) fibs tfibs
```

```
primes :: [Int]                                 Primzahlfolge
primes = sieve $ nats 2
```

```
sieve :: [Int] -> [Int]                         Sieb des Erathostenes
```

```
sieve (p:s) = p:sieve [n | n <- s, n `mod` p /= 0]

take 11 prims  ⇝  [2,3,5,7,11,13,17,19,23,29,31]

hamming :: [Int]                          Folge aller Hammingzahlen
hamming  = 1:foldl1 merge (map (\x -> map (*x) hamming) [2,3,5])

take 30 hamming  ⇝  [1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,
                     30,32,36,40,45,48,50,54,60,64,72,75,80]
```

Standardfunktionen zur Erzeugung unendlicher Listen

```
repeat :: a -> [a]
repeat a = a:repeat a                    repeat 5  ⇝  5:5:5:5:...

replicate :: Int -> a -> [a]
replicate n a = take n $ repeat a  replicate 4 5  ⇝  [5,5,5,5]

iterate :: (a -> a) -> a -> [a]
iterate f a = a:iterate f (f a)    iterate (+2) 10  ⇝  10:12:14:...
```

Link zur schrittweisen Auswertung von *take(5)$iterate(+2)(10)*

Funktionsiteration mit (siehe Abschnitt 2.6) mit *iterate* oder *replicate*

```
(^^^) :: (a -> a) -> Int -> a -> a
f^^^n = \x -> iterate f x!!n
f^^^n = foldl (.) id $ replicate n f
```

Mit Funktionskomposition:

```
f^^^n = (!!n) . iterate f
(^^^) f  = foldl (.) id . flip replicate f
```

Sequentielles *map* mehrerer Funktionen

```
seqMap :: [a -> b] -> [a] -> [b]
seqMap fs = zipWith ($) repeatfs where repeatfs = fs++repeatfs

seqMap [(+1),(+3),(+7)] [1..9]  ⇝  [2,5,10,5,8,13,8,11,16]
```

Unendliche Listen von Listen

Die Definition der Menge B^* aller Listen mit Elementen der Menge $B \subseteq A$ ist die Vereinigung aller Potenzen von B:

$$B^* = \{\epsilon\} \cup \bigcup_{n>0} B^n. \tag{1}$$

Folgende Haskell-Implementierung des Sternoperators gibt (1) wieder:

```
star1 :: [a] -> [[a]]
star1 b = concatMap power [0..]
          where power 0 = [[]]
                power n = [x:xs | x <- b, xs <- power $ n-1]

take 20 $ star1 [1..3]
 ⇝ [[],[1],[2],[3],[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],
    [3,3],[1,1,1],[1,1,2],[1,1,3],[1,2,1],[1,2,2],[1,2,3],[1,3,1]]
```

Äquivalent zu (1) ist die Definition von B^* als kleinste Lösung der Gleichung

$$M = \{\epsilon\} \cup B \times M \tag{2}$$

in der Mengenvariablen M. In Kapitel 4 und Abschnitt 6.2 werden wir näher auf solche Mengendefinitionen und ihre mathematische Begründung eingehen.

Folgende Haskell-Implementierung des Sternoperators gibt (2) wieder:

```
star2 :: [a] -> [[a]]
star2 b = m where m = []:[a:as | as <- m, a <- b]          (3)

take 20 $ star2 [1..3]
 ↝ [[],[1],[2],[3],[1,1],[2,1],[3,1],[1,2],[2,2],[3,2],[1,3],[2,3],
    [3,3],[1,1,1],[2,1,1],[3,1,1],[1,2,1],[2,2,1],[3,2,1],[1,3,1]]
```

star1 und *star2* produzieren Listen, die nach ihrer Länge geordnet sind. Das liegt an der Reihenfolge der Generatoren von (3). Da m unendlich ist, würde bei umgekehrter Reihenfolge der beiden Generatoren jeder Aufruf der Form *take(n)(star2[1..3])* nur Listen mit lauter Einsen zurückgeben. Generatoren, die aus *unendlichen* Listen auswählen, sollten also stets am Anfang stehen!

star1 und *star2* erzeugen unendliche Listen endlicher Listen. Bei den anderen Beispielen in diesem Abschnitt ging es meistens um unendliche Listen einfacher Objekte wie Zahlen. In beiden Fällen werden unendliche Listen mit Hilfe von Gleichungen *endlich repräsentiert* (und in Haskell auch als solche implementiert), wobei es genaugenommen eine *Lösung* der Gleichungen ist, die die jeweilige Liste repräsentiert.

4 Rekursive Datentypen

Das allgemeine Schema einer Datentypdefinition entspricht dem eines Summentypen (siehe Abschnitt 2.2):

```
data DT a_1 ... a_m = C_1 typ_11 ... typ_1n_1 | ... |
                      C_k typ_k1 ... typ_kn_k
```

$a_1, \ldots, a_m$ sind Typvariablen. Die Funktionssymbole $C_1, \ldots, C_k$ heißen **Konstruktoren**. Als Funktion hat C_i den Typ

```
typ_i1 -> ... -> typ_in_i -> DT a_1 ... a_m.
```

Der Typ typ_{ij} mit $1 \leq i \leq k$ und $1 \leq j \leq n_i$, darf keine Typvariablen außer $a_1, \ldots, a_m$ enthalten. Gibt es $1 \leq i \leq k$ und $1 \leq j \leq n_i$ derart, dass typ_{ij} (eine Instanz von) DT enthält, dann ist DT rekursiv, ansonsten ist DT ein Summentyp.

Die durch DT bezeichnete Menge besteht aus allen Ausdrücken der Form

```
C_i e_1 ... e_n_i,
```

wobei $1 \leq i \leq k$ und für alle $1 \leq j \leq n_i$ e_j ein Element des Typs typ_{ij} ist.

Alle mit einem Großbuchstaben beginnenden Funktionssymbole und alle mit einem Doppelpunkt beginnenden aus Sonderzeichen bestehenden Strings werden vom Haskell-Compiler als Konstruktoren eines Datentyps aufgefasst und müssen deshalb irgendwo im Programm in einer Datentypdefinition vorkommen. Da Konstruktoren Funktionen sind, gelten im Übrigen bei Konstruktoren dieselben Unterschiede zwischen der Infix- und der Präfixdarstellung wie bei anderen Funktionen (siehe Abschnitt 2.5).

Der in Kapitel 3 behandelte Standardtyp für Listen ist als rekursiver Datentyp wie folgt definiert:

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

Sonderzeichen in Typnamen *selbstdefinierter* Datentypen sind nicht erlaubt!

Für alle Mengen A besteht die Menge $[A]$ aus dem Ausdruck [] sowie

- allen endlichen Ausdrücken $a_1 : \ldots : a_n : []$ mit $a_1, \ldots, a_n \in A$ und
- allen unendlichen Ausdrücken $a_1 : a_2 : a_3 : \ldots$ mit $\{a_i \mid i \in \mathbb{N}\} \subseteq A$.

$[A]$ ist die größte Lösung der Gleichung

$$M \;=\; \{[]\} \cup \{a : s \mid a \in A,\; s \in M\} \tag{1}$$

in der Mengenvariablen M.

Ein unendlicher Ausdruck lässt sich oft als die eindeutige Lösung einer sog. iterativen Gleichung darstellen. So ist z.B. der Ausdruck $0 : 1 : 0 : 1 : 0 : 1 : \ldots$ die eindeutige Lösung der Gleichung

$$blink \;=\; 0 : 1 : blink \tag{2}$$

in der Individuenvariablen $blink$.

Haben alle Konstruktoren eines Datentyps DT mindestens ein Argument desselben Typs, dann sind *alle* Ausdrücke, aus denen DT besteht, unendlich.

Würde man z.B. den Konstruktor `[]` aus dem Datentyp `[a]` entfernen, dann bestünde die größte Lösung von (1) nur noch aus allen unendlichen Ausdrücken $a_1 : a_2 : a_3 : \ldots$ mit $\{a_i \mid i \in \mathbb{N}\} \subseteq A$.

Die endlichen Ausdrücke von $[A]$ bilden die *kleinste* Lösung von (1).

Natürliche und ganze Zahlen als Datentypelemente

```
data Nat    = Zero | Succ Nat                    (3)
data PosNat = One | Succ' PosNat                 (4)
data Int'   = Zero' | Plus PosNat | Minus PosNat (5)
```

Die größten Lösungen von (3), (4) und (5) sind zu $\mathbb{N} \cup \{\infty\}$, $\mathbb{N}_{<0} \cup \{\infty\}$ bzw. $\mathbb{Z} \cup \{\infty, -\infty\}$ isomorph.

Int' ist offenbar nicht rekursiv, also ein Summentyp (siehe Abschnitt 2.2). Er realisiert demnach die disjunktive Vereinigung seiner drei Argumenttypen:

$$\begin{aligned} Int' &= \{Zero'\} \cup \{Plus(e) \mid e \in PosNat \cup \{Minus(e) \mid e \in PosNat\} \\ &\cong \{Zero'\} \cup \{(n,1) \mid n \in PosNat\} \cup \{(n,2) \mid n \in PosNat\} \\ &= \{Zero'\} \uplus PosNat \uplus PosNat \end{aligned}$$

$A \cong B$ bezeichnet einen Isomorphismus, d.h. die Existenz einer bijektiven Abbildung zwischen den Mengen A und B.

Datentypen mit Destruktoren

Um auf die Argumente eines Konstruktors zugreifen zu können, ordnet man ihnen Namen zu, die **field labels** oder **Destruktoren**. Dazu wird

```
C_i typ_i1 ... typ_in_i
```

in obiger Definition von DT erweitert zu:

```
C_i {d_i1 :: typ_i1,..., d_in_i :: typ_in_i}
```

Wie C_i, so ist auch d_{ij} eine Funktion. Als solche hat sie den Typ

```
DT a1 ... am -> typ_ij
```

Kommt DT in typ_{ij} nicht vor, dann wird d_{ij} auch **Attribut** oder **Selektor** genannt.

Nicht-rekursive Datentypen mit genau einem Konstruktor entsprechen den in Kapitel 2 behandelten Produkttypen. Folglich sind alle Destruktoren eines Produkttyps Attribute.

Rekursive Datentypen mit genau einem Konstruktor liefern die Haskell-Realisierung der aus imperativen und objektorientierten Programmiersprachen bekannten **Records** und **Objektklassen**. Deren Destruktoren, die keine Attribute sind, werden dort **Methoden** genannt. Semantisch entsprechen sie Zustandstransformationen, sofern man die Elemente des Datentyps als Zustände auffasst. Außerdem notiert man in objektorientierten Sprachen in der Regel den Aufruf $d_{ij}(x)$ eines Destruktors in der "qualifizierten" Form $x.d_{ij}$.

Destruktoren sind invers zu Konstruktoren. Z.B. hat der folgende Ausdruck den Wert e_j:

```
d_ij (C_i e1 ... eni)
```

Mit Destruktoren lautet das allgemeine Schema einer Datentypdefinition also wie folgt:

```
data DT a_1 ... a_m = C_1 {d_11 :: typ_11,..., d_1n_1 :: typ_1n_1} |
                      ... |
                      C_k {d_k1 :: typ_k1,..., d_kn_k :: typ_kn_k}
```

Elemente von DT können mit oder ohne Destruktoren definiert werden:

```
obj = C_i e_i1 ... e_in_i                    ist äquivalent zu
obj = C_i {d_i1 = e_i1,..., d_in_i = e_in_i}
```

Die Werte einzelner Destruktoren von *obj* können wie folgt verändert werden:

```
obj' = obj {d_ij_1 = e_1,..., d_ij_m = e_m}
```

obj′ unterscheidet sich von *obj* dadurch, dass den Destruktoren $d_{ij_1}, \ldots, d_{ij_m}$ neue Werte, nämlich $e_1, \ldots, e_m$ zugewiesen wurden.

Destruktoren dürfen nicht rekursiv definiert werden. Folglich deutet der Haskell-Compiler jedes Vorkommen eines Destruktors d_{ij} auf der rechten Seite einer Definitionsgleichung als eine gleichnamige, aber von d_{ij} verschiedene Funktion und sucht nach deren Definition.

Dies kann man nutzen, um d_{ij} doch rekursiv zu definieren, indem man in der zweiten Definition von *obj* (s.o.) die Gleichung $d_{ij} = e_j$ durch $d_{ij} = d_{ij}$ ersetzt und die neue Funktion auf der rechten Seite lokal definiert:

```
obj = C_i {d_i1 = e_1,..., d_ij = d_ij,..., d_in_i = en_i}
      where d_ij ... = ... d_ij ...
```

Ein Konstruktor darf nicht zu mehreren Datentypen gehören.
Ein Destruktor darf nicht zu mehreren Konstruktoren unterschiedlicher Datentypen gehören.

Listen mit Destruktoren

```
data List a = Nil | Cons {hd :: a, tl :: List a}
```

Da nur die mit dem Konstruktor *Cons* gebildeten Elemente von $List(A)$ die Destruktoren

`hd :: List a -> a` und `tl :: List a -> List a`

haben, sind *hd* und *tl* *partielle* Funktionen.

$hd(s)$ und $tl(s)$ liefern den Kopf bzw. Rest einer nichtleeren Liste s.

Da sich die Definitionsbereiche partieller Destruktoren erst aus der jeweiligen Datentypdefinition erschließen, sollten (und können!) Datentypen mit Destruktoren und *mehreren* Konstruktoren grundsätzlich vermieden werden. So kann der obige Datentyp *List(a)* auch als destruktiver Datentyp mit genau einem Konstruktor definiert werden:

Listen mit totalem Destruktor

```
data Colist a = Colist {split :: Maybe (a,Colist a)}
```

oder ohne Destruktor:

```
data Colist a = Colist (Maybe (a,Colist a))
```

Die leere Liste hat in $Colist(A)$ folgende Darstellung:

```
nil :: Colist a
nil = Colist Nothing
```

Für jede Menge A ist die Menge $Colist(A)$ die größte Lösung der Gleichung

$$M \;=\; \{Colist(Nothing)\} \cup \{Colist(Just(a,s)) \mid a \in A,\; s \in M\} \tag{6}$$

in der Mengenvariablen M.

Wie man leicht sieht, ist die größte Lösung von (6) isomorph zur größten Lösung von (1), besteht also aus allen endlichen und allen unendlichen Listen von Elementes der Menge A.

Als Elemente von $Colist(\mathbb{Z})$ lassen sich z.B. die Folgen $(0,1,0,1,\dots)$ und $(1,0,1,0,\dots)$ wie folgt implementieren:

```
blink,blink' :: Colist Int
blink  = Colist $ Just (0,blink')
blink' = Colist $ Just (1,blink)
```

Ausschließlich unendliche Listen können auch als Elemente des folgenden Datentyps implementiert werden:

```
data Stream a = (:<) {hd :: a, tl :: Stream a}
```

oder ohne Destruktor:

```
data Stream a = a :< Stream a
```

Für jede Menge A ist die Menge $Stream(A)$ die größte Lösung der Gleichung

$$M \;=\; \{a :< s \mid a \in A,\; s \in M\} \tag{7}$$

in der Mengenvariablen M. Sie ist u.a. isomorph zur Menge $A^{\mathbb{N}}$ der Funktionen von $\mathbb{N}$ nach A.

Als Elemente von $Stream(Int)$ lauten z.B. *blink* und *blink'* (s.o.) wie folgt:

```
blink,blink' :: Stream Int
blink  = 0:<blink'
blink' = 1:<blink
```

Conat

Entsprechend der Isomorphie der größten Lösungen von (1) bzw. (6) sind auch die größten Lösungen von (3) und der folgenden Datentypdefinition isomorph:

```
data Conat = Conat {pred :: Maybe Conat}
```

Die Null hat in *Conat* folgende Darstellung:

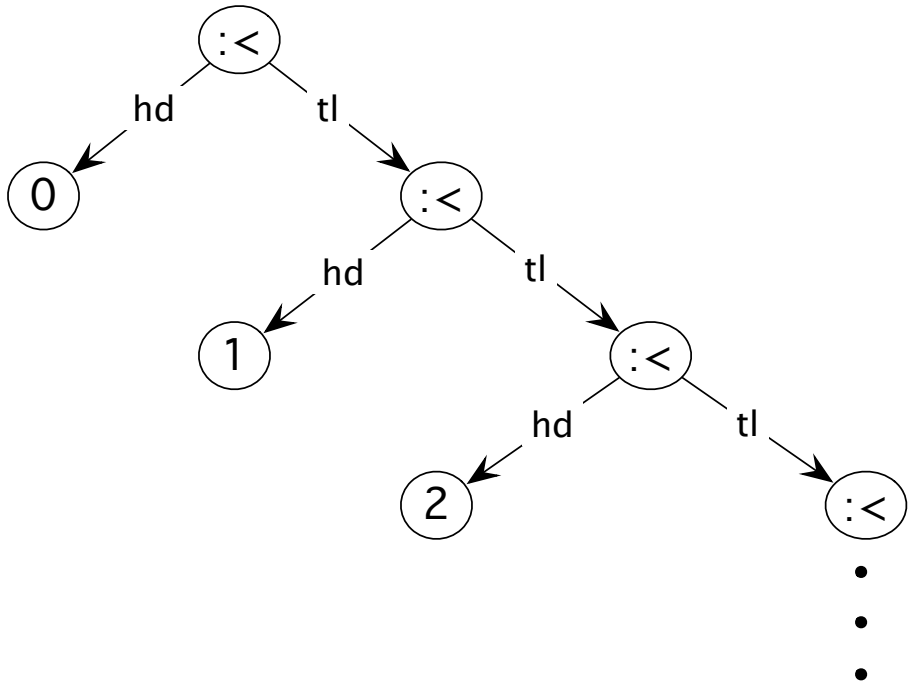
```
zero :: Conat
zero = Conat Nothing
```

So wie die unendlichen Listen *blink* und *blink'* durch die eindeutigen Lösungen von Gleichungen zwischen endlichen Ausdrücken beschrieben werden können, so lässt sich ∞ als eindeutige Lösung solcher Gleichungen darstellen:

```
infinity :: Nat
infinity = Succ infinity

infinity' :: Conat
infinity' = Conat $ Just infinity'
```

Die oben geforderte Vermeidung von Datentypen mit mindestens einem Destruktor, aber mehr als einem Konstruktor garantiert, dass alle Destruktoren totale Funktionen sind, und erlaubt es uns deshalb, in den üblichen Darstellungen der Elemente eines Datentyps DT als – u.U. unendliche – Bäume, deren Knoten mit Konstruktoren markiert sind, die Kanten mit Destrukturen zu markieren.

So hat z.B. der Strom aller natürlichen Zahlen als Element von $Stream(Int)$ die folgende Baumdarstellung:



Mathematisch können Bäume mit Knoten- und Kantenmarkierungen aus der Menge C bzw. D als partielle Funktionen von D^* nach C dargestellt werden (siehe [23], Kapitel 2 und 12).

Zurück zu Datentypen mit mehreren Konstruktoren, aber ohne Destruktoren.

4.1 Arithmetische und Boolesche Ausdrücke (Haskell-Modul: Expr.hs)

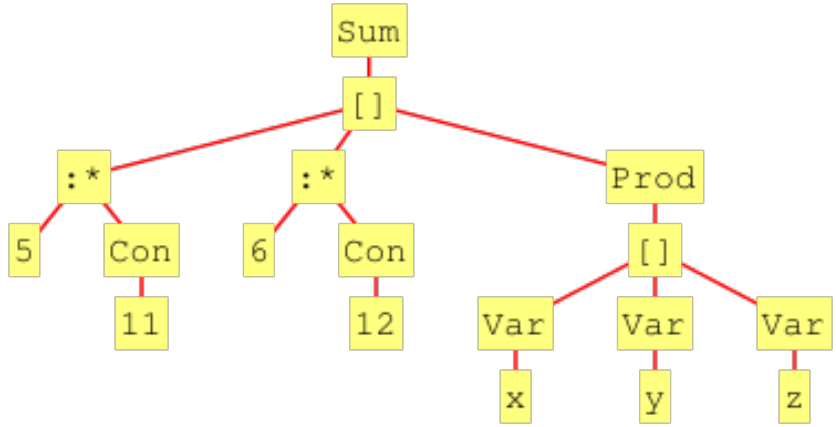
```
data Exp x = Con Int | Var x | Sum [Exp x] | Prod [Exp x] |
             Exp x :- Exp x | Exp x :/ Exp x | Int :* Exp x |
             Exp x :^ Int
```

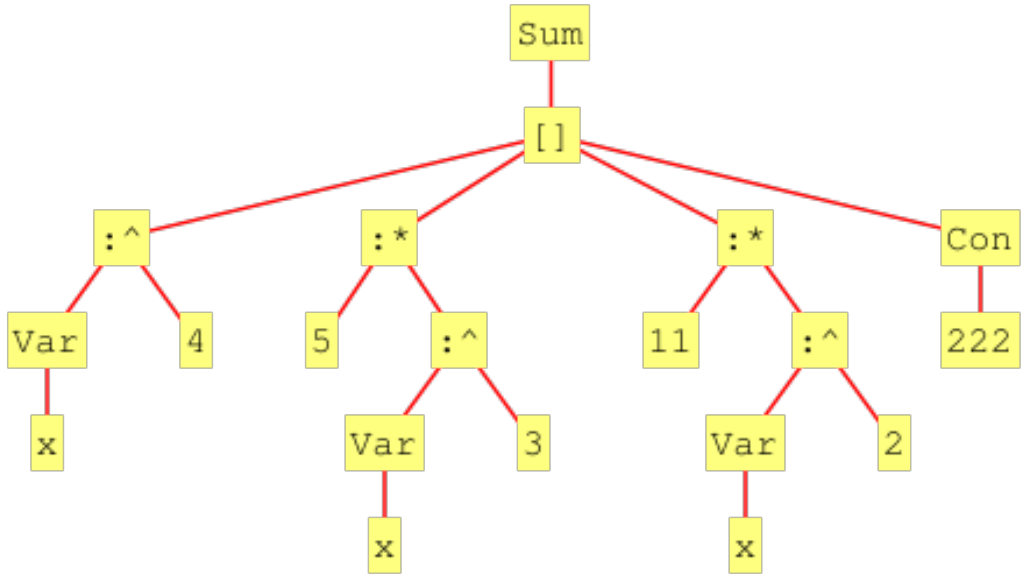
oder in der Form eines generalized algebraic data type (GADT):

```
data Exp x where Con       :: Int -> Exp x
                 Var       :: x -> Exp x
                 Sum,Prod  :: [Exp x] -> Exp x
                 (:-),(:/) :: Exp x -> Exp x -> Exp x
                 (:*)      :: Int -> Exp x -> Exp x
                 (:^)      :: Exp x -> Int -> Exp x
zero = Con 0
one  = Con 1
```

Z.B. lauten die Ausdrücke $5 * 11 + 6 * 12 + x * y * z$ und $x^4 + 5 * x^3 + 11 * x^2 + 222$ als Elemente des Typs *Exp(String)* wie folgt:

```
Sum [5:*Con 11, 6:*Con 12, Prod [Var"x", Var"y", Var"z"]]
Sum [Var"x":^4, 5:*(Var"x":^3), 11:*(Var"x":^2), Con 222]
```


Sum
[]
:*
:*
Prod
5
Con
6
Con
[]
11
12
Var
Var
Var
x
y
z


Sum
[]
:^
:*
:*
Con
Var
4
5
:^
11
:^
222
x
Var
3
Var
2
x
x

Boolesche Ausdrücke

```
data BExp x = True_ | False_ | BVar x | Or [BExp x] |
              And [BExp x] | Not (BExp x) | Exp x := Exp x |
              Exp x :<= Exp x
```

oder als GADT (s.o.):

```
data BExp x where True_,False_ :: BExp x
                 BVar          :: x -> BExp x
                 Or,And        :: [BExp x] -> BExp x
                 Not           :: BExp x -> BExp x
                 (:=) ,(:<=)   :: Exp x -> Exp x -> BExp x
```

Darüberhinaus erlaubt ein GADT erlaubt unterschiedliche Instanzen seiner Typvariablen und damit die Zusammenfassung mehrerer Datentypen zu einem einzigen wie in folgendem Beispiel:

Arithmetische, Boolesche, If-, Paar- und Listenausdrücke in einem einzigen GADT

```
data GExp x a where
     Con           :: Int -> GExp x Int
     Var           :: x -> GExp x a
     Sum,Prod      :: [GExp x Int] -> GExp x Int
     (:-),(:/)     :: GExp x Int -> GExp x Int -> GExp x Int
     (:*)          :: Int -> GExp x Int -> GExp x Int
     (:^)          :: GExp x Int -> Int -> GExp x Int
     True_,False_  :: GExp x Bool
     Or,And        :: [GExp x Bool] -> GExp x Bool
     Not           :: GExp x Bool -> GExp x Bool
     (:=), (:<=)   :: GExp x Int -> GExp x Int -> GExp x Bool
     If            :: GExp x Bool -> GExp x a -> GExp x a -> GExp x a
     Pair          :: GExp x a -> GExp x b -> GExp x (a,b)
     List          :: [GExp x a] -> GExp x [a]
```

Die verwendeten Instanzen der Typvariable a sind grün markiert.

GADTs sind hier nur der Vollständigkeit halber erwähnt. Sie kommen im Folgenden nicht weiter vor.

4.2 Signaturen, Algebren und Faltungen

Ein Haskell-Datentyp DT ist nicht nur die Menge seiner Konstruktoren, sondern liefert gleichzeitig deren Interpretation als Funktionen, die aus Ausdrücken neue Ausdrücke bilden. Demgegenüber sollte ein als *abstrakter* Datentyp bezeichnetes Sprachkonstrukt auch interpretierende Funktionen zulassen, die anstelle von Ausdrücken Elemente anderer **Trägermengen** berechnen.

Eine Trägermenge A zusammen mit einer (typkonformen) Interpretation der Konstruktoren von DT als **Operationen** auf A nennen wir **Algebra**. Ein Ausdruck (mathematisch: **Term**) t wird in ihr ausgewertet, indem die Konstruktoren von t durch die interpretierenden Operationen ersetzt und diese von innen nach außen angewendet werden. So wird aus t ein Element von A, anschaulich gesprochen: t wird schrittweise zu einem Wert **gefaltet**.

In der mathematischen Logik werden abstrakte Datentypen **Signaturen** genannt. In Haskell lässt sich jede Signatur Σ so als destruktiver Datentyp implementieren, dass jedes seiner Objekt eine Σ-Algebra bildet.

Umgekehrt ist jeder Haskell-Datentyp DT (ohne funktionale Typen) die Trägermenge einer **Termalgebra** T, die in der Logik auch *Herbrand-Basis* genannt wird. Die Signatur Σ, bzgl. der T eine Algebra ist, enthält für jeden Konstruktor von DT genau eine Operation und wird deshalb **konstruktiv** genannt.

Beispiele (Haskell-Modul: Coalg.hs)

```
data NatSig val = NatSig {zero_ :: val,                    Signatur NatSig
                          succ :: val -> val}

foldNat :: NatSig val -> Nat -> val                        NatSig-Termfaltung
foldNat alg = \case Zero   -> zero_ alg
                    Succ n -> succ alg $ foldNat alg sn

natT :: NatSig Nat                                         NatSig-Termalgebra
natT = NatSig Zero Succ

data List x val = List {nil :: val,                        Signatur List(x)
                        cons :: x -> val -> val}

foldList :: List x val -> [x] -> val                       List(x)-Termfaltung
foldList alg = \case []    -> nil alg
                     (x:s) -> cons alg x $ foldList alg s

listT :: List x [x]                                        List(x)-Termalgebra
listT = List [] (:)
```

```
intAlg:: List Int Int                                   List(Int)-Algebra
intAlg = List {nil = 0, cons = (+)}

foldList intAlg [1..8] ⇝ 36                             Faltung von [1..8] in intAlg
```

foldList haben wir schon in Abschnitt 3.8 kennengelernt:

$$foldList(alg) \;=\; foldr(cons(alg))(nil(alg)) :: [x] \to val$$

Die Faltung von Termen in der zugehörigen *Termalgebra* ist immer die Identität:

$$\begin{aligned} foldNat(natT) &= id :: Nat \to Nat, \\ foldList(listT) &= id :: [x] \to [x]. \end{aligned}$$

Beispiel (Haskell-Modul: Expr.hs)

```
data Arith x val = Arith {con       :: Int -> val,
                          var_      :: x -> val,
                          sum_,prod :: [val] -> val,
                          sub,div_  :: val -> val -> val,
                          scal      :: Int -> val -> val,
                          expo      :: val -> Int -> val}
```

Signatur Arith(x)

```
foldArith :: Arith x val -> Exp x -> val
foldArith alg = \case Con i   -> con alg i
                      Var x   -> var_ alg x
                      Sum es  -> sum_ alg $ map f es
                      Prod es -> prod alg $ map f es
                      e :- e' -> sub alg (f e) (f e')
                      e :/ e' -> div_ alg (f e) (f e')
                      i :* e  -> scal alg i $ f e
                      e :^ i  -> expo alg (f e) i
                where f = foldArith alg
```

Arith(x)-Termfaltung

```
arithT :: Arith x (Exp x)
arithT = Arith Con Var Sum Prod (:-) (:/) (:*) (:^)
```

Arith(x)-Termalgebra

$$foldArith(arithT) \;=\; id :: Exp(x) \to Exp(x)$$

Auswertung von *Arith(x)*-Termen (= *Exp(x)*-Objekten) zu ganzen Zahlen

```
type Store x = x -> Int
```

Belegung der Variablen

```
eval :: Exp x -> Store x -> Int
eval e store = case e of Con i   -> i
                         Var x   -> st x
                         Sum es  -> sum $ map f es
                         Prod es -> product $ map f es
                         e :- e' -> f e - f e'
                         e :/ e' -> f e `div` f e'
                         i :* e  -> i * f e
                         e :^ i  -> f e ^ i
               where f = flip eval store
```

Auswertung von *Arith(x)*-Termen zu ganzen Zahlen als Faltung in einer Algebra

```
evalAlg :: Arith x (Store x -> Int)
evalAlg = Arith {con  = const,
                 var_ = \x       -> ($x),
                 sum_ = \bs st   -> sum $ map ($st) bs,
                 prod = \bs st   -> product $ map ($st) bs,
                 sub  = \b b' st -> b st - b' st,
                 div_ = \b b' st -> b st `div` b' st,
                 scal = \i b st  -> i * b st,
                 expo = \b i st  -> b st ^ i}
```

Faltung in *evalAlg* = Auswertung von *Arith(x)*-Termen:

$$foldArith(evalAlg) \;=\; eval$$

Beispiel

```
exp1 :: Exp String
exp1 = Sum [Var"x":^4, 5:*(Var"x":^3), 11:*(Var"x":^2), Con 222]
```

```
eval exp1 $ \"x" -> 4   ⇝   974
```

Link zu den Berechnungsschritten

In der Animation steht *evalC* für *flip(eval)*($\lambda X.4$), X für `"x"` und **e1:+...:+en** für `Sum[e1,...,en]`.

Jede auf einem Datentyp DT mit Konstruktoren $C_1, \ldots, C_k$ durch Gleichungen der Form

$$f(C_i(t_1, \ldots, t_n), b) \;=\; C_i^A(f(t_1), \ldots, f(t_n), b), \quad 1 \leq i \leq k, \tag{1}$$

definierte Funktion $f : DT \to A$ entspricht der Faltung von DT-Termen in einer Algebra (mit Trägermenge) A, die den Konstruktor $C_i : DT^n \times B \to DT$ durch die Funktion $C_i^A : A^n \times B \to A$ interpretiert.

(1) wirkt zunächst sehr eingeschränkt. So sind z.B. *primitiv-rekursive Funktionen* oder *Paramorphismen* durch Gleichungen der Form

$$f(C_i(t_1, \ldots, t_n), b) \;=\; h_i(f(t_1), t_1, \ldots, f(t_n), t_n, b), \quad 1 \leq i \leq k, \tag{2}$$

definiert, wo h_i im Gegensatz zu C_i^A auch auf die Argumente $t_1, \ldots, t_n$ von C_i Bezug nimmt.

Beispiel Symbolische Differentiation

```
diffE :: Eq x => Exp x -> x -> Exp x
diffE e x = case e of Con _   -> zero
                      Var y   -> if x == y then one else zero
```

```
                Sum es  -> Sum $ map d es
                Prod es -> Sum $ zipWith h [0..] es
                           where h i = Prod . updList es i . d
                e :- e' -> d e :- d e'
                e :/ e' -> (Prod[d e,e']:-Prod[e,d e']):/(e':^2)
                i :* e  -> i :* d e
                e :^ i  -> i :* Prod[d e,e:^(i-1)]
          where d = flip diffE x
```

Link zur schrittweisen Auswertung von *diffE(exp1)(x)*, hier geflippt: *diff(x)(exp1)*. Neben der Definition von *diffE* werden zur Vereinfachung der Zwischenergebnisse weitere Gleichungen wie z.B. $e + 0 = e$, $e * 1 = e$ und $3 * 5 * e = 15 * e$ angewendet. ❑

(2) kann als Speziallfall der Verallgemeinerung von (1) auf die gleichzeitige Definition zweier Funktionen $f_1 : DT \to A_1, f_2 : DT \to A_2$ als **Produktextension**

$$\langle f_1, f_2 \rangle : DT \to A_1 \times A_2$$

mit Gleichungen der Form

$$\langle f_1, f_2 \rangle(C_i(t_1, \ldots, t_n), b) \;=\; C_i^{A_1 \times A_2}(\langle f_1, f_2 \rangle(t_1), \ldots, \langle f_1, f_2 \rangle(t_n), b), \quad 1 \leq i \leq k, \qquad (3)$$

formuliert werden.

$\langle f_1, f_2 \rangle$ wurde am Ende von Abschnitt 2.5 durch die Haskell-Funktion `(***)` implementiert und bildet $t \in DT$ auf das Tupel $(f_1(t), f_2(t))$ ab, während die Funktion

$$C_i^{A_1 \times A_2} : (A_1 \times A_2)^n \times B \to A_1 \times A_2 \tag{4}$$

jedem $(n+1)$-Tupel

$$((a_{11}, a_{12}), \ldots, (a_{n1}, a_{n2}), b) \in (A_1 \times A_2)^n \times B$$

das Paar

$$(C_i^{A_1}(a_{11}, \ldots, a_{n1}, b), C_i^{A_2}(a_{12}, \ldots, a_{n2}, b)) \in A_1 \times A_2$$

zuordnet.

Sind A_1, A_2 Algebren, dann interpretiert (3) C_i in der *Produktalgebra* mit Trägermenge $A_1 \times A_2$.

Im Fall von *diffE* ist $A_1 = (x \to Exp(x)$, $A_2 = Exp(x)$, $f_1 = diffE : Exp(x) \to A_1$ und $f_2 = id : Exp(x) \to A_2$. Die Produktalgebra mit Trägermenge $A_1 \times A_2$ lässt sich wie folgt implementieren:

```
diffAlg :: Eq x => Arith x (x -> Exp x, Exp x)
diffAlg = Arith
      {con  = \i -> (const zero, Con i),
       var_ = \x -> (\y -> if x == y then one else zero, Var x),
       sum_ = \fes -> let (fs,es) = unzip fes
                      in (\x -> Sum $ map ($x) fs, Sum es),
       prod = \fes -> let (fs,es) = unzip fes
                          h x i e = Prod $ updList es i $ (fs!!i) x
                      in (\x Sum $ zipWith (h x) [0..] es, Prod es),
       sub  = \(f,e) (g,e') -> (\x -> f x :- g x, e :- e'),
       div_ = \(f,e) (g,e') -> (\x -> (Prod [f x,e'] :- Prod [e,g x])
                                      :/ (e' :^ 2), e :/ e'),
       scal = \i (f,e) -> (\x -> i :* f x, i :* e),
       expo = \(f,e) i -> (\x -> i :* Prod [f x,e:^(i-1)], e :^ i)}
```

diffE entspricht der ersten Komponente der Faltung in *diffAlg*:

$$foldArith(diffAlg) \;=\; \langle diffE, id\rangle : Exp(x) \to ((x \to Exp(x)) \times Exp(x))$$

Aufgabe Gegeben ist folgende rekursive Definition der Fakultätsfunktion:

```
fact :: Nat -> Int
fact Zero     = 1
fact (Succ n) = fact n * (mkInt n + 1)

mkInt :: Nat -> Int
mkInt Zero     = 0
mkInt (Succ n) = mkInt n + 1
```

Wie muss eine *NatSig*-Algebra

$$\texttt{factAlg} :: \texttt{NatSig}\ (\texttt{Int}, \texttt{Int})$$

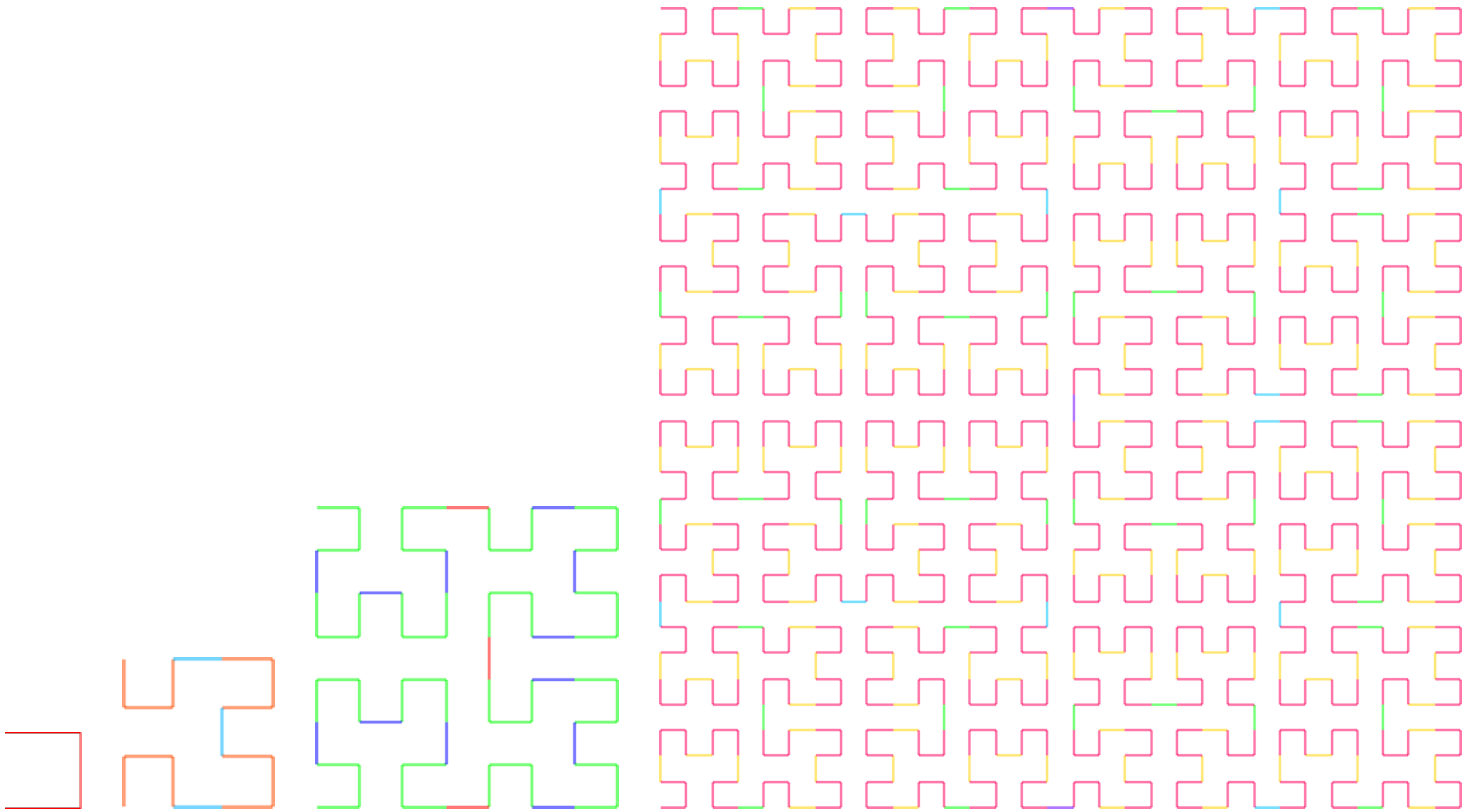
definiert werden, damit sie die Gleichungen

$$foldNat(factAlg) \ = \ \langle fact, mkInt \rangle : Nat \to Int^2$$

erfüllt? ❏

4.5 Hilbertkurven

gehören zu den FASS-Kurven unter den Fraktalen, die u.a. als Antennen zum Einsatz kommen. Hilbertkurven können auf unterschiedliche Weise mit dem Painter gezeichnet und in svg-Dateien gespeichert werden. Die folgenden Darstellungen wurden damit erzeugt:



Hilbertkurven der Tiefen 1, 2, 3 und 5.
Auf gleicher Rekursionstiefe erzeugte Kanten haben dieselbe Farbe.

Solche Kantenzüge lassen sich nicht nur als Punktlisten (s.o.), sondern auch als Listen von Aktionen repräsentieren, die auszuführen sind, um einen Kantenzug zu zeichnen. Ein Schritt von einem Punkt zum nächsten erfordert die Drehung um einen Winkel a (*Turn(a)*) und die anschließende Vor- bzw. Rückwärtsbewegung um eine Distanz d (*Move(d)*).

```
data Action = Turn Float | Move Float
up,down :: Action
up    = Turn $ -90
down  = Turn 90

north,east,south,west :: [Action]
north = [up,Move 5,down]
east  = [Move 5]
south = [down,Move 5,up]
west  = [Move $ -5]
```

Die Hilbertkurve der Tiefe n wird – abhängig von einer Anfangsrichtung vom Typ *Direction* – aus vier Hilbertkurven der Tiefe $n-1$ zusammengesetzt, die durch drei – unten rot markierte – Aktionsfolgen miteinander verbunden werden:

```
data Direction = North | East | South | West
```

```
hilbertActs :: Int -> Direction -> [Action]
hilbertActs 0 = const []
hilbertActs n =
      \case East -> hSouth++east++hEast++south++hEast++west++hNorth
            West -> hNorth++west++hWest++north++hWest++east++hSouth
            North -> hWest++north++hNorth++west++hNorth++south++hEast
            South -> hEast++south++hSouth++east++hSouth++north++hWest
      where h = hilbertActs (n-1)
            hEast = h East; hWest = h West
            hNorth = h North; hSouth = h South
```

Mit Hilfe von *foldl* kann eine Aktionsliste in einen Kantenzug vom Typ *Path* überführt werden (siehe Abschnitt 3.6):

```
executeActs :: [Action] -> Path
executeActs = fst . foldl f ([(0,0)],0) where
              f (ps,a) (Move d) = (ps++[successor (last ps) d a],a)
              f (ps,a) (Turn b) = (ps,a+b)
              successor (x,y) d a = rotate (x,y) a (x+d,y)
```

rotate wurde in Kapitel 2 definiert.

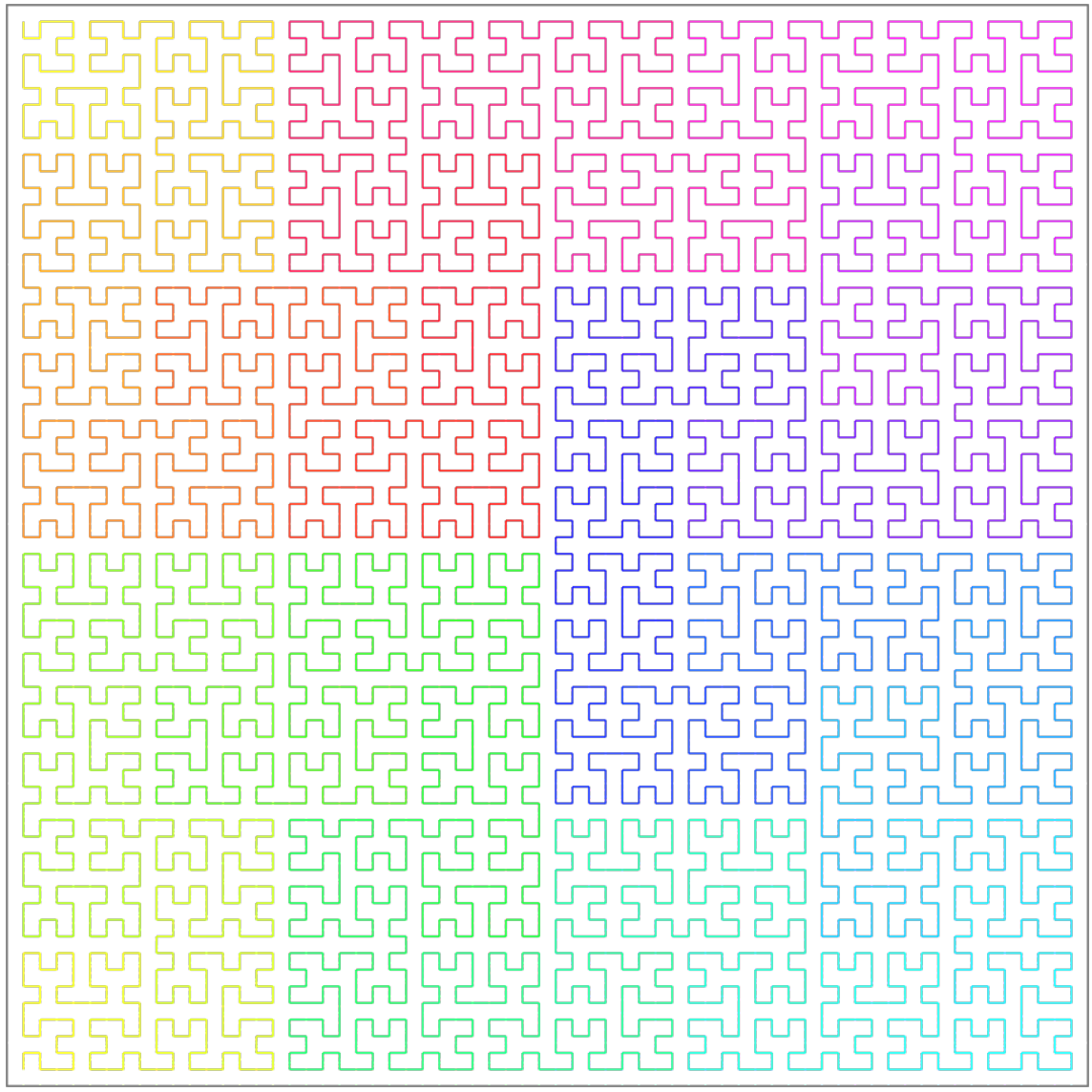
Die folgende Funktion *addColors* ordnet den Elementen einer Liste mit $n \leq 1530$ Elementen n verschiedene – bzgl. des im vorigen Kapitel definierten Farbkreises von 1530 Hue-Farben – äquidistante Farben zu:

```
addColors :: [a] -> [(a,RGB)]
addColors s = zip s $ map f [0..] where
              f i = iterate nextCol red!!round (i*1530/length s)
```

Z.B. ordnet *addColors* den jeweiligen Elementen einer 11-elementigen Liste die folgenden Farben zu:

Die Anwendung von *addColors* auf eine Hilbertkurve projiziert einen Farbkreis auf ihre einzelnen Kanten und deutet damit die Reihenfolge an, in der ihre Eckpunkte aufgelistet werden.

Ist *East* die Startrichtung, dann führt der Weg insgesamt, aber auch bei jedem rekursiven Aufruf, von links oben nach rechts oben, von dort nach rechts unten und schließlich nach links unten:



Anwendung von `addColors` *auf die Hilbertkurve der Tiefe 6*

4.4 Zweidimensionale Figuren

Der Painter enthält einen Datentyp für Graphen, die als Listen von Kantenzügen dargestellt werden:

```
data Curves = C {file :: String, paths :: [Path], colors :: [RGB],
                 modes :: [Int], points :: [Point]}
type Path  = [Point]
type Point = (Float,Float)
```

Für alle Graphen g ist $file(g)$ die Datei, in der das Quadrupel

$$(paths(g), colors(g), modes(g), points(g))$$

abgelegt wird. $paths(g)$ ist eine Zerlegung des Graphen in Kanten. $colors(g)$, $modes(g)$ und $points(g)$ ordnen jedem Weg von g eine (Start-)Farbe, einen fünfstelligen Zahlencode, der steuert, wie er gezeichnet und gefärbt wird, bzw. einen Rotationsmittelpunkt zu.

Mit dem Aufruf $drawC(g)$ wird g in die Datei $file(g)$ eingetragen und eine Schleife gestartet, in der zur – durch Leerzeichen getrennten – Eingabe reellwertiger horizontaler und vertikaler Skalierungsfaktoren aufgefordert wird.

Nach Drücken der return-Taste wird svg-Code für g erzeugt und in die Datei *Pix/file(g).svg* geschrieben, so dass beim Öffnen dieser Datei mit einem Browser dort das Bild von g erscheint.

Verlassen wird die Schleife, wenn anstelle einer Parametereingabe die return-Taste gedrückt wird.

Der Painter stellt zahlreiche Operationen zur Erzeugung, Veränderung oder Kombination von Graphen des Typs *Curves* zur Verfügung, u.a. (hier z.T. in vereinfachter Form wiedergegeben):

```
combine :: [Curves] -> Curves
combine cs@(c:_) = c {paths = f paths, colors = f colors,
                      modes = f modes, points = f points}
                   where f = flip concatMap cs

zipCurves :: (Point -> Point -> Point) -> Curves -> Curves -> Curves
zipCurves f c d = c {paths = zipWith (zipWith f) (paths c) $ paths d,
                     points = zipWith f (points c) $ points d}

morphing :: Int -> [Curves] -> Curves
morphing n cs = combine $ zipWith f (init cs) $ tail cs where
                f c d = combine $ map g [0..n] where
                        g i = zipCurves h c d where
                        h (xc,yc) (xd,yd) = (t xc xd,t yc yd)
                t x x' = (1-step)*x+step*x'
```

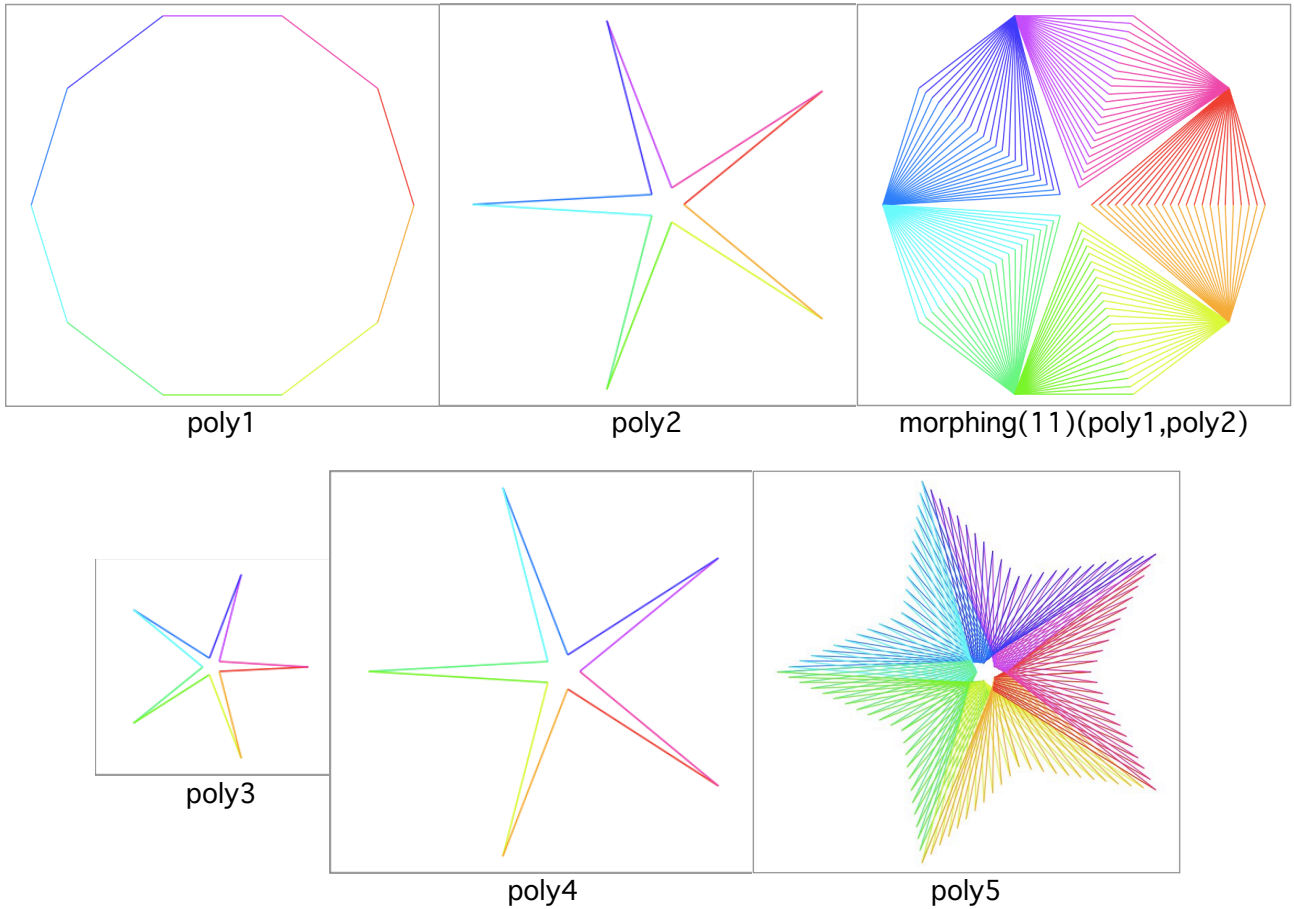
```
step = float i/float n
```

zipCurves(f)(g)(g') erzeugt einen neuen Graphen aus den Graphen g und g', indem die Funktion $f : Point \to Point \to Point$ auf jedes Paar sich entsprechender Punkte von g bzw. g' angewendet wird. $combine(gs)$ vereinigt alle Graphen der Liste gs zu einem einzigen Graphen, ohne ihre jeweiligen Kantenzüge zu verschieben. $morphing(n)(gs)$ fügt zwischen je zwei benachbarte Graphen der Liste gs n von einem Morphing-Algorithmus erzeugte äquidistante Zwischenstufen ein.

Beispiele

```
poly1,poly2,poly3,poly4 :: Curves
poly1 = poly 12111 10 [44]
poly2 = poly 12111 5 [4,44]
poly3 = turn 36 $ scale 0.5 poly2
poly4 = turn 72 poly2
poly5 = morphing 11 [poly2,poly3,poly4]
```

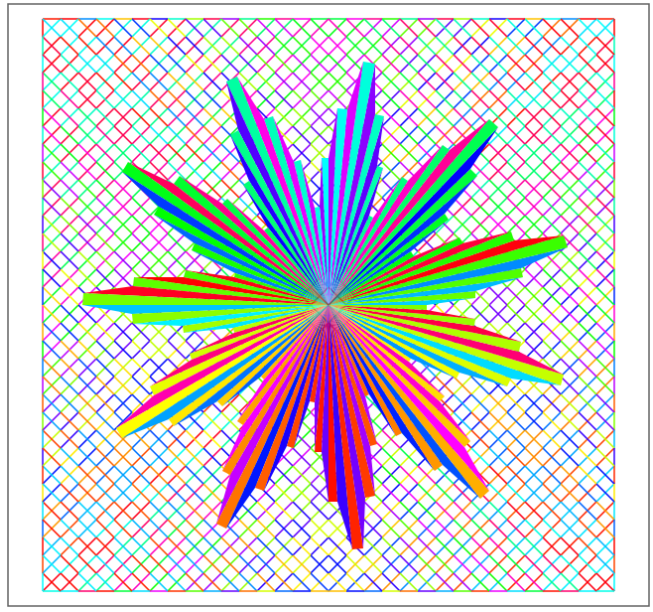
$poly(mode)(n)(rs)$ erzeugt ein Polygon mit $n * |rs|$ Ecken. Für alle $1 \leq i \leq |rs|$ liegt die $(i * n)$-te Ecke auf einem Kreis mit Radius $rs!!i$ um den Mittelpunkt des Polygons.



Einige Modes bewirken, dass von Kantenzügen aufgespannte Dreiecke mit Farben gefüllt werden, wie es z.B. bei der Polygon-Komponente des folgenden Graphen der Fall ist:

```
graph :: Curves
graph = overlay [g,flipV g,scale 0.25 $ poly 13123 11 rs]
        where g = cant 12121 33
              rs = [22,22,33,33,44,44,55,55,44,44,33,33]
```

cant(*mode*)(*n*) erzeugt eine Cantorsche Diagonalkurve der Dimension *n*, *flipV(g)* spiegelt *g* an der Vertikalen durch den Mittelpunkt von *g*, *scale*(0.25)(*g*) verkleinert *g* auf ein Viertel der ursprünglichen Größe, *overlay*(*gs*) legt alle Elemente der Graphenliste *gs* übereinander. *drawC(graph)* zeichnet schließlich folgendes Bild in die Datei *cant.svg*:



5 Typklassen und Bäume

(Haskell-Module: Examples.hs, Coalg.hs)

Typklassen stellen Bedingungen (*Constraints*) an die Instanzen einer Typvariablen. Die Bedingungen bestehen in der Existenz bestimmter Funktionen, z.B. einer Gleichheit und einer Ungelichheit:

```
class Eq a where (==), (/=) :: a -> a -> Bool
                 a /= b  =  not $ a == b
                 a == b  =  not $ a /= b
```

Der Typ jeder Funktion einer Typklasse muss deren Typvariable enthalten.

Eine **Instanz einer Typklasse** besteht aus Instanzen ihrer Typvariablen sowie Definitionen in ihr deklarierter Funktionen, z.B.

```
instance Eq (Int,Bool) where (x,b) == (y,c) = x == y && b == c

instance Eq a => Eq [a] where
         s == s' = length s == length s' && and (zipWith (==) s s')
```

Die Definitionen von `(/=)` und `(==)` in der Typklasse `Eq` sind Defaults. Instanzen von `Eq` dürfen Sie überschreiben werden.

Wegen der zyklischen Abhängigkeit von `(/=)` und `(==)` in *Eq* muss jede Instanz mindestens einer der beiden Relationen definieren. Darüberhinaus benutzt die Listeninstanz von *Eq* Gleichheiten auf *Int* und *a*. Dass letztere existiert, wird durch das Constraint *Eq(a)* ausgedrückt.

5.1 Mengenoperationen auf Listen

```
insert :: Eq a => a -> [a] -> [a]
insert a s@(b:s') = if a == b then s else b:insert a s'
insert a _        = [a]

union :: Eq a => [a] -> [a] -> [a]                    Mengenvereinigung
union = foldl $ flip insert

unionMap :: Eq b => (a -> [b]) -> [a] -> [b]          concatMap für Mengen
unionMap f = foldl union [] . map f
```

```
meet :: Eq a => [a] -> [a] -> [a]
meet = filter . flip elem
```
Mengendurchschnitt

```
remove :: Eq a => a -> [a] -> [a]
remove = filter . (/=)
```
Entfernung (aller Vorkommen) eines Elementes

```
diff :: Eq a => [a] -> [a] -> [a]
diff = foldl $ flip remove
```
Mengendifferenz

```
subset :: Eq a => [a] -> [a] -> Bool
s `subset` s' = all (`elem` s') s
```
Mengeninklusion

```
eqset :: Eq a => [a] -> [a] -> Bool
s `eqset` s' = s `subset` s' && s' `subset` s
```
Mengengleichheit

```
powerset :: Eq a => [a] -> [[a]]
powerset (a:s) = if a `elem` s then ps else ps ++ map (a:) ps
                 where ps = powerset s
powerset _     = [[]]
```
Potenzmenge

Berechnung der Äquivalenzklassen des Äquivalenzabschlusses einer Relation $R \subseteq M^2$, wobei M und R als Listen vom Typ `[a]` bzw. `[(a,a)]` übergeben werden:

```
mkPartition :: Eq a => [a] -> [(a,a)] -> [[a]]
mkPartition = foldl f . map (\a -> [a]) where
              f part (a,b) = if eqset cla clb then part
                             else (cla++clb):diff part s where
                             s@[cla,clb] = map g [a,b]
                             g a = head $ filter (elem a) part
```

5.2 Unterklassen

Typklassen können wie Objektklassen objektorientierter Sprachen andere Typklassen erben. Die jeweiligen Oberklassen werden vor dem Erben vor dem Pfeil `=>` aufgelistet.

```
class Eq a => Ord a where
              (<=), (<), (>=), (>) :: a -> a -> Bool
              max, min :: a -> a -> a
              a < b   = a <= b && a /= b
              a >= b  = b <= a
              a > b   = b < a
              max x y = if x >= y then x else y
              min x y = if x <= y then x else y
```

5.3 Sortieralgorithmen

```
quicksort :: Ord a => [a] -> [a]
quicksort (x:s) = quicksort [y | y <- s, y <= x]++x:
                  quicksort [y | y <- s, y > x]
quicksort s     = s
```

Quicksort ist ein divide-and-conquer-Algorithmus mit mittlerer Laufzeit $O(n * log_2(n))$, wobei n die Listenlänge ist. Wegen der 2 rekursiven Aufrufe in der Definition von *quicksort* ist $log_2(n)$ die (mittlere) Anzahl der Aufrufe von *quicksort*. Wegen des einen rekursiven Aufrufs in der Definition der *conquer*-Operation ++ ist n die Anzahl der Aufrufe von ++. Entsprechendes gilt für Mergesort mit der *divide*-Operation *split* oder *splitAt* (siehe Listen) anstelle von *filter* und der *conquer*-Operation *merge* anstelle von ++:

```
mergesort :: Ord a => [a] -> [a]
mergesort (x:y:s) = merge (mergesort $ x:s1) $ mergesort $ y:s2
                    where (s1,s2) = split s
mergesort s       = s

split :: [a] -> ([a],[a])
split (x:y:s) = (x:s1,y:s2) where (s1,s2) = split s
split s       = (s,[])
```

```
merge :: Ord a => [a] -> [a] -> [a]
merge s1@(x:s2) s3@(y:s4) = if x <= y then x:merge s2 s3
                                      else y:merge s1 s4
merge [] s                = s
merge s _                 = s

msort :: Ord a => [a] -> [a]
msort s = if n < 2 then s else merge (msort s1) $ msort s2
          where n = length s
                (s1,s2) = splitAt (n `div` 2) s
```

5.4 Binäre Bäume

```
data Bintree a = Empty | Bjoin a (Bintree a) (Bintree a)

leaf :: a -> Bintree a
leaf a = Bjoin a Empty Empty
```

Binäre Bäume ausbalancieren

```
baltree :: [a] -> Bintree a
baltree [] = Empty
baltree s  = Bjoin a (baltree s1) (baltree s2)
             where (s1,a:s2) = splitAt (length s`div`2) s
```

Binäre Bäume als Suchbäume nutzen

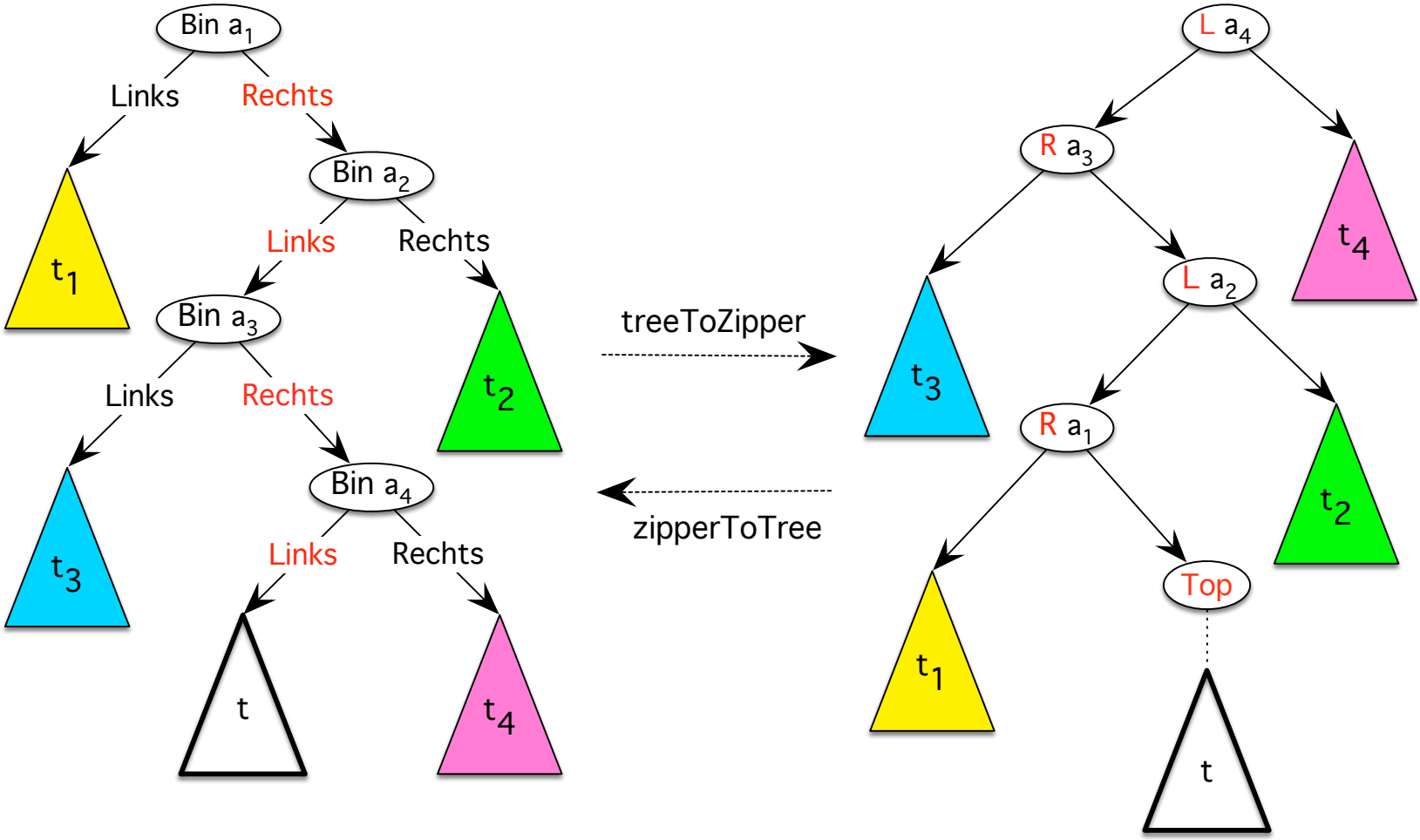
```
insertTree :: Ord a => a -> Bintree a -> Bintree a
insertTree a t@(Bjoin b t1 t2) | a == b = t
                               | a < b  = Bjoin b (insertTree a t1) t2
                               | True   = Bjoin b t1 (insertTree a t2)
insertTree a _ = leaf a
```

Binäre Bäume ordnen

```
instance Eq a => Ord (Bintree a) where
       Empty <= _ = True
       Bjoin a t1 t2 <= Bjoin b t3 t4 = a == b && t1 <= t3 && t2 <= t4
       _ <= Empty = False
```

5.5 Binäre Bäume mit Zeiger auf einen Knoten

```
data BintreeL a = Leaf a | Bin a (BintreeL a) (BintreeL a)

data Edge = Links | Rechts

type Node = [Edge]                    Repräsentation eines Knotens als Weg,
                                      der von der Wurzel aus zu ihm führt

type TreeNode a = (BintreeL a,Node)   Baum und indizierter Knoten

data Context a =
     Top |                                          leerer Kontext
     L a (Context a) (BintreeL a) |      Kontext eines linken Teilbaums
     R a (BintreeL a) (Context a)        Kontext eines rechten Teilbaums

type TreeZipper a = (Context a,BintreeL a)
                    Zerlegung eines Baums t mit indiziertem Knoten node
                    in den Kontextbaum c = t(Top)
                    und den indizierten Unterbaum getIndexedN(t,node) (s.u.)
```


Bin a_1
Links
Rechts
t_1
Bin a_2
Links
Rechts
Bin a_3
t_2
Links
Rechts
t_3
Bin a_4
Links
Rechts
t
t_4
treeToZipper
zipperToTree
L a_4
R a_3
t_4
L a_2
t_3
R a_1
t_2
Top
t_1
t

```
treeToZipper :: TreeNode a -> TreeZipper a
treeToZipper (t,node) = loop Top t node where
            loop :: Context a -> BintreeL a -> Node -> TreeZipper a
            loop c (Bin a t u) (Links:node)  = loop (L a c u) t node
            loop c (Bin a t u) (Rechts:node) = loop (R a t c) u node
            loop c t _                       = (c,t)

zipperToTree :: TreeZipper a -> TreeNode a
zipperToTree (c,t) = loop c t [] where
            loop :: Context a -> BintreeL a -> Node -> TreeNode a
            loop (L a c t) u node = loop c (Bin a u t) (Links:node)
            loop (R a t c) u node = loop c (Bin a t u) (Rechts:node)
            loop _ t node         = (t,node)
```

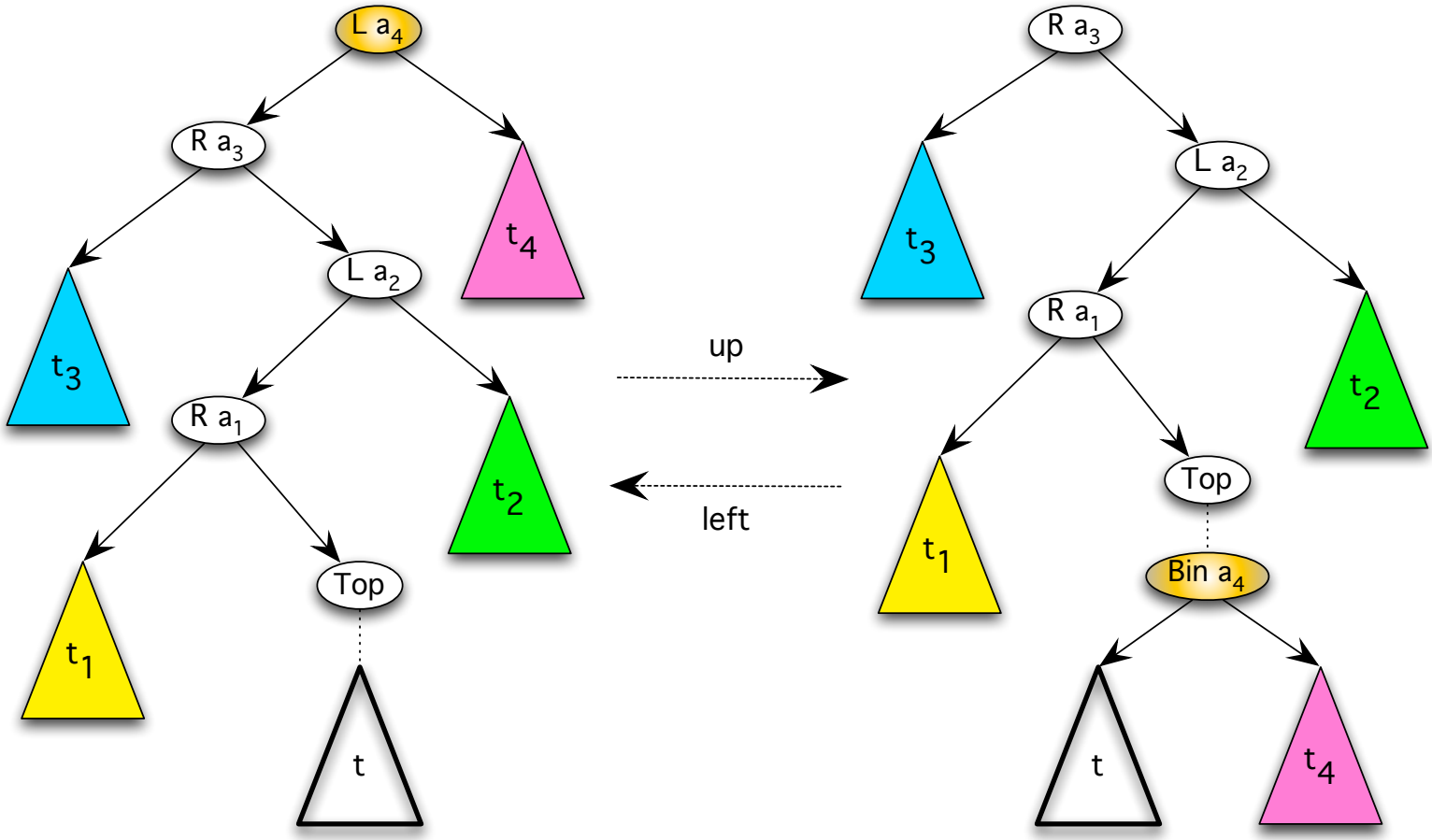
`treeToZipper` und `zipperToTree` sind invers zueinander:

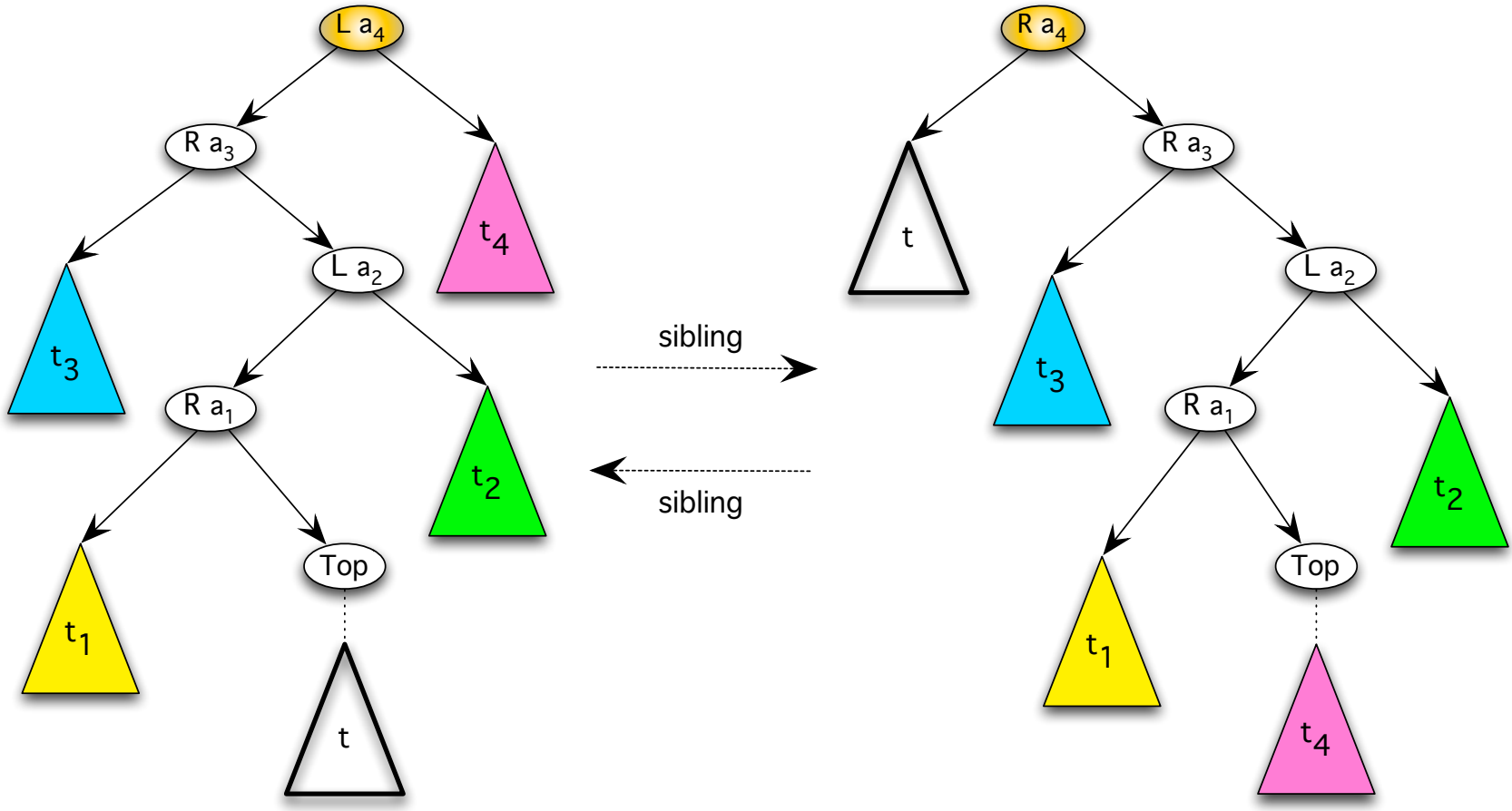
$$\mathit{TreeNode}(A) \supseteq \{(t, node) \in \mathit{BinTreeL}(A) \times \mathit{Edge}^* \mid node \in t\} \cong \mathit{TreeZipper}(A).$$

Zeigerbewegungen und Zugriff auf den indizierten Knoten

```
upN,leftN,rightN,siblingN :: TreeNode a -> TreeNode a
upN (t,node@(_:_))      = (t,init node)
leftN (t,node)          = (t,node++[Links])
rightN (t,node)         = (t,node++[Rechts])
siblingN (t,node@(_:_)) = (t,init node++[other $ last node])
                          where other Links = Rechts
                                other _     = Links

getIndexedN :: TreeNode a -> a
getIndexedN (Bin a t u,Links:node)  = getIndexedN (t,node)
getIndexedN (Bin a t u,Rechts:node) = getIndexedN (u,node)
getIndexedN (t,[])                  = t

up,left,right,sibling :: TreeZipper a -> TreeZipper a
up (L a c u,t)      = (c,Bin a t u)
up (R a t c,u)      = (c,Bin a t u)
left  (c,Bin a t u) = (L a c u,t)
right (c,Bin a t u) = (R a t c,u)
sibling (L a c u,t) = (R a t c,u)
sibling (R a t c,u) = (L a c u,t)
```

```
getIndexed :: TreeZipper a -> a
getIndexed (_,Leaf a)     = a
getIndexed (_,Bin a _ _) = a
```





Analog zu Abschnitt 3.6 erlaubt die Zipperdarstellung binärer Bäume mit indiziertem Knoten *node* den direkten Zugriff auf *node* (siehe `getIndexed`), während bei der `TreeNode`-Darstellung der jeweilige Baum erst von seiner Wurzel bis *node* durchlaufen wird (siehe `getIndexedN`).

5.6 Ausgeben

Bei der Ausgabe von Daten eines Typs T wird automatisch die T-Instanz der Funktion `show` aufgerufen, die zur Typklasse `Show a` gehört.

```
class Show a where
      show :: a -> String
      show x = shows x ""

      shows :: a -> String -> String
      shows = showsPrec 0

      showsPrec :: Int -> a -> String -> String
```

Das String-Argument von **`showsPrec`** und **`showsPrec`** wird an die Ausgabe des Argumentes vom Typ **`a`** angefügt.

Steht `deriving Show` am Ende der Definition eines Datentyps, dann werden dessen Elemente in der Darstellung ausgegeben, in der sie im Programmen vorkommen.

Für andere Ausgabeformate müssen entsprechende Instanzen von `show` oder `showsPrec` definiert werden.

Binäre Bäume ausgeben

```
instance Show a => Show (Bintree a) where
         showsPrec _ Empty                  = id
         showsPrec _ (Bjoin a Empty Empty) = shows a
         showsPrec _ (Bjoin a left right)  = shows a . ('(':) .
                                             shows left . (',':) .
                                             shows right . (')':)

instance Show a => Show (BintreeL a) where
         showsPrec _ (Leaf a)           = shows a
         showsPrec _ (Bin a left right) = shows a . ('(':) .
                                          shows left . (',':) .
                                          shows right . (')':)
```

5.7 Arithmetische Ausdrücke ausgeben (Haskell-Modul: Expr.hs)

Die Festlegung unterschiedlicher Prioritäten binärer Operationen erlaubt es, die Klammerung von Teilausdrücken t' eines Ausdrucks t auf diejenigen zu beschränken, deren führende Operation op' eine Priorität hat, die geringer ist als die Priorität der Operation op, die in t auf t' angewendet wird.

Daher übersetzt die folgende Show-Instanz von *Exp(String)* jeden Ausdruck vom Typ *Exp(String)* (siehe Abschnitt 4.1) in einen äquivalenten String mit minimaler Klammerung (bzgl. der üblichen Prioritäten arithmetischer Operatoren; nach Richard Bird, Thinking Functionally with Haskell, Abschnitt 11.5):

```
instance Show x => Show (Exp x) where
   showsPrec _ (Con i)   = shows i
   showsPrec _ (Var x)   = (x++)
   showsPrec p (Sum es)  = enclose p q $ showMore q '+' es
                           where q = prio '+'
   showsPrec p (Prod es) = enclose p q $ showMore q '*' es
                           where q = prio '*'
   showsPrec p (e :- e') = enclose p q $ showsPrec q e . ('-':) .
                           showsPrec (q+1) e' where q = prio '-'  (1)
   showsPrec p (e :/ e') = enclose p q $ showsPrec q e . ('/':) .
                           showsPrec (q+1) e' where q = prio '/'  (2)
   showsPrec p (i :* e)  = enclose p q $ shows i . ('*':) .
                           showsPrec q e where q = prio '*'
   showsPrec p (e :^ i)  = enclose p q $ showsPrec q e . ('^':) .
                           shows i where q = prio '^'
```

```
prio :: Char -> Int
prio = \case '+' -> 0; '-' -> 0; '*' -> 1; '/' -> 1; '^' -> 2

enclose :: Int -> Int -> (String -> String) -> String -> String
enclose p q f | p > q = ('(':) . f . (')':)
              | True  = f

showMore :: Exp x => Int -> Char -> [Exp x] -> String -> String
showMore p op (e:es) = foldl f (showsPrec p e) es
                       where f g e = g . (op:) . showsPrec p e
```

Sind s_1, s_2, s_3 die Stringdarstellungen dreier Bäume t_1, t_2, t_3, dann bewitkt die um 1 erhöhte Priorität im Fall (1), dass *showsPrec* den Baum $(t_1 :- t_2) :- t_3$, wie die übliche Auswertung von links her nahelegt, als String $s_1 - s_2 - s_3$ wiedergibt, $t_1 :- (t_2 :- t_3)$ jedoch als geklammerten String $s_1 - (s_2 - s_3)$. Analoges gilt für / anstelle von - (siehe (2)).

Beispiele

```
show $ Sum[5:*Con 11,6:*Con 12,Prod[x,y,z]] ⇝ 5*11+6*12+x*y*z
show $ Prod[x,Con 5,5:*Prod[x:-y,y,z]]      ⇝ x*5*5*(x-y)*y*z
```

```
show $ Sum[11:*(x:^3),5:*(x:^2),16:*x,Con 33,x:-(y:-z),
          Prod[x:^5,Sum[x:^5,x:^6]]]
                          ⇝ 11*x^3+5*x^2+16*x+33+x-(y-z)+x^5*(x^5+x^6)
show $ Sum[11:*(x:^3),5:*(x:^2),16:*x,Con 33,(x:-y):-z,
          Prod[x:^5,Sum[x:^5,x:^6]]]
                          ⇝ 11*x^3+5*x^2+16*x+33+x-y-z+x^5*(x^5+x^6)
```

5.8 Einlesen

Vor der Eingabe von Daten eines Typs T wird automatisch die T-Instanz der Funktion `read` aufgerufen, die zur Typklasse `Read a` gehört:

```
class Read a where
      readsPrec :: Int -> String -> [(a,String)]

      reads :: String -> [(a,String)]
      reads = readsPrec 0

      read :: String -> a
      read s = case [x | (x,t) <- reads s, ("","") <- lex t] of
                    [x] -> x
                    []  -> error "PreludeText.read: no parse"
                    _   -> error "PreludeText.read: ambiguous parse"
```

`reads s` liefert eine Liste von Paaren, bestehend aus dem als Element vom Typ `a` erkannten Präfix von `s` und der jeweiligen Resteingabe (= Suffix von `s`).

`lex :: String -> [(a,String)]` ist eine Standardfunktion, die ein evtl. aus mehreren Zeichen bestehendes Symbol erkennt, vom Eingabestring abspaltet und sowohl das Symbol als auch die Resteingabe ausgibt.

Der Generator `("","") <- lex t` in der obigen Definition von `read s` bewirkt, dass nur die Paare `(x,t)` von `reads s` berücksichtigt werden, bei denen die Resteingabe `t` aus Leerzeichen, Zeilenumbrüchen und Tabulatoren besteht (siehe Beispiele unten).

Steht `deriving Read` am Ende der Definition eines Datentyps, dann werden dessen Elemente in der Darstellung erkannt, in der sie in Programmen vorkommen. Für andere Eingabeformate müssen entsprechende Instanzen von `readsPrec` definiert werden.

Binäre Bäume einlesen

Die `Show`-Instanz von `BintreeL(a)` übersetzt Bäume dieses Typs in entsprechende Klammerstrukturen. Die entprechende `Read`-Instanz erkennt solche Klammerstrukturen und übersetzt sie in Objekte des Typs `BintreeL(a)`, wobei Leerzeichen in der Klammerstruktur unberücksichtigt bleiben.

```
instance Read a => Read (BintreeL a) where
```

```
readsPrec _ s = [(Leaf a,s) | (a,s) <- reads s] ++
                [(Bin a left right,s) |
                     (a,s) <- reads s,     ("(",s) <- lex s,
                     (left,s) <- reads s,  (",",s) <- lex s,
                     (right,s) <- reads s, (")",s) <- lex s]
```

Da der Generator `(a,s) <- reads s` einer Zuweisung an die "Variablen" `a` und `s` entspricht, kann `s` auf der linken Seite der Zuweisung einen anderen Wert als auf der rechten Seite haben. Tatsächlich ist der linke String `s` ein Suffix des rechten. Das abgespaltene Präfix wurde von `reads` in das Datentypelement `a` übersetzt.

Die Aufrufe von `reads` in der Definition von `readsPrec` sind je nach Kontext (*Leaf* oder *Bin*) Aufrufe von

```
reads :: String -> [(a,String)]                (1)
```

oder

```
reads :: String -> [(BintreeL a,String)]       (2)
```

Wichtig ist, dass der erste Generator beider Listenkomprehensionen der Definition von `readsPrec` keinen Aufruf von (2) enthält. Hier hat `s` nämlich noch denselben Wert wie auf der linken Seite der Gleichung. Der Aufruf von `readsPrec` würde also in eine Endlosschleife laufen! Die restlichen Generatoren enthalten nur Anwendungen von `reads` auf kürzere Strings und garantieren deshalb die Termination des Aufrufs von `readsPrec`.

Beispiele

```
reads "5(7(3, 8),6 ) " :: [(BintreeL Int,String)]
                ⇝ [(Leaf 5,"(7(3, 8),6 ) "),
                    (Bin 5 (Bin 7 (Leaf 3) (Leaf 8)) (Leaf 6)," ")]

read "5(7(3, 8),6 ) " :: BintreeL Int
                ⇝ Bin 5 (Bin 7 (Leaf 3) (Leaf 8)) (Leaf 6)

reads "5(7(3,8),6)hh" :: [(BintreeL Int,String)]
                ⇝ [(Leaf 5,"(7(3,8),6)hh"),
                    (Bin 5 (Bin 7 (Leaf 3) (Leaf 8)) (Leaf 6),"hh")]

read "5(7(3,8),6)hh" :: BintreeL Int
                ⇝ Exception: PreludeText.read: no parse
```

5.9 Bäume mit beliebigem Ausgrad

Im Unterschied zum obigen Datentyp `Bintree(a)` von Bäumen mit Knotenausgrad 2 definiert der Datentyp

```
data Tree a = V a | F a [Tree a]
```

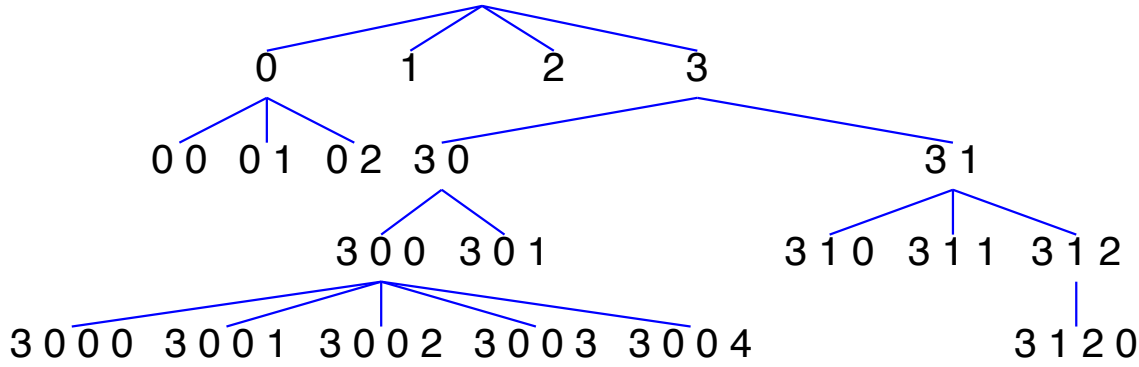
knotenmarkierte Bäume mit beliebigem (endlichem) Knotenausgrad.

Wie bei `Bintree(a)` wird die Menge möglicher Markierungen durch Instanzen der Typvariablen a festgelegt.

Außerdem erlaubt `Tree(a)` zwei Blattarten: Sowohl Ausdrücke der Form `V(a)` als auch solche der Form `F(a)[]` stellen Blätter dar. `Tree(a)` wird meist in Zusammenhängen verwendet, wo `V(a)` eine Variable mit Name a darstellt und `F(a)(ts)` die Anwendung einer Funktion mit Name a auf die Argumentliste ts. Dann repräsentiert `F(a)[]` eine Konstante (= nullstellige Funktion).

```
root :: Tree a -> a
root (V a)   = a
root (F a _) = a
```

```
subtrees :: Tree a -> [Tree a]
subtrees (F _ ts) = ts
subtrees _        = []

tree1 :: Tree Int
tree1 = F 1 [F 2 [F 2 [V 3,V(-1)],V(-2)],F 4 [V(-3),V 5]]

subtrees tree1  ⇝  [F 2 [F 2 [V 3,V(-1)],V(-2)], F 4 [V(-3),V 5]]

size,height :: Tree a -> Int
size (F _ ts) = 1+sum (map size ts)
size _        = 1
height (F _ ts) = 1+foldl max 0 (map height ts)
height _        = 1

size tree1    ⇝  9
height tree1  ⇝  4

type Node = [Int]
```



Die eindeutige Identifizierung von Baumknoten durch Listen natürlicher Zahlen

```
nodes, leaves:: Tree a -> [Node]
nodes (F _ ts)  = []:[i:node | (t,i) <- zip ts [0..],
                                node <- nodes t]
nodes _         = [[]]
leaves (F _ []) = [[]]
leaves (F _ ts) = [i:node | (t,i) <- zip ts [0..], node <- leaves t]
leaves _        = [[]]

nodes tree1  ⇝ [[],[0],[0,0],[0,0,0],[0,0,1],[0,1],[1],[1,0],[1,1]]
leaves tree1 ⇝ [[0,0,0],[0,0,1],[0,1],[1,0],[1,1]]
```

label(t)(node) liefert die Markierung des Knotens *node* von t:

```
label :: Tree a -> Node -> a
label t [] = root t
label (F _ ts) (i:node) | i < length ts
           = label (ts!!i) node
label _ _  = error "label"

label tree1 [0,0,1]  ⇝  -1
```

depthfirst(t) liefert die Liste der Knoteneinträge des Baums t in der Reihenfolge, in der die Knoten bei einem depthfirst- (oder preorder-) Durchlauf von t besucht werden.

```
depthfirst :: Tree a -> [a]
depthfirst t = root t:concatMap depthfirst (subtrees t)
```

breadthfirst(ts) liefert die Liste der Knoteneinträge der Baumliste ts in der Reihenfolge, in der die Knoten bei einem breadthfirst- (oder Breiten-) Durchlauf von ts besucht werden.

```
breadthfirst :: [Tree a] -> [a]
breadthfirst [] = []
breadthfirst ts = map root ts ++ breadthfirst (concatMap subtrees ts)
```

getSubtree(t)(node) ist der Unterbaum von t mit der Wurzel *node*:

```
getSubtree :: Tree a -> Node -> Tree a
getSubtree t [] = t
getSubtree (F _ ts) (i:node) | i < length ts
                = getSubtree (ts!!i) node
getSubtree _ _  = error "getSubtree"

getSubtree tree1 [0,0,1]  ⇝  V(-1)
```

putSubtree(t)(node)(u) ersetzt *getSubtree(t)(node)* durch u:

```
putSubtree :: Tree a -> Node -> Tree a -> Tree a
putSubtree t [] u = u
putSubtree (F a ts) (i:node) u | i < length ts
                  = F a $ updList ts i $ putSubtree (ts!!i) node u
putSubtree _ _ _  = error "putSubtree"

putSubtree tree1 [0,0,1] $ getSubtree tree1 [1]
 ⇝  F 1 [F 2 [F 2 [V 3,F 4 [V (-3),V 5]],V (-2)],F 4 [V (-3),V 5]]
```

mapTree(f)(t) wendet die Funktion $h : a \to b$ auf jede Knotenmarkierung von t an:

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (V a)    = V $ f a
mapTree f (F a ts) = F (f a) $ map (mapTree f) ts

mapTree show tree1
       ⇝   F"1"[F"2"[F"2"[V"3",V"-1"],V"-2"],F"4"[V"-3",V"5"]]
```

Bäume als destruktiver Datentyp

Wie die Datentypen für Listen (siehe Kapitel 4), so enthalten auch Datentypen für Bäume unendliche Objekte. So kann z.B. für Bäume mit beliebigem (endlichem oder unendlichem) Ausgrad folgender zu *Tree(a)* semantisch äquivalenter Datentyp mit Destruktoren (*splitT* für Bäume und *split* für Baumlisten) verwendet werden:

```
data Cotree a = Cotree {splitT :: (a,Colist (Cotree a))}
```

5.10 Baumfaltungen (Haskell-Modul: Coalg.hs)

Analog zu den Beispielen in Abschnitt 4.2 definieren wir Signaturen für binäre bzw. beliebige Bäume, erweitern *Bintree(a)* und *Tree(a)* zu Termalgebren dieser Signaturen und implementieren die Faltungen von *Bintree(a)*- bzw. *Tree(a)*-Termen:

```
data BinSig a val = BinSig {empty_ :: val,
                            bjoin  :: a -> val -> val -> val}

data TreeSig a val = TreeSig {var :: a -> val,
                              fun :: a -> [val] -> val}

foldBin:: BinSig a val -> Bintree a -> val
foldBin alg Empty                 = empty_ alg
foldBin alg (Bjoin a left right) = bjoin alg a (foldBin alg left)
                                               (foldBin alg right)

foldTree :: TreeSig a val -> Tree a -> val
foldTree alg (V a)    = var alg a
foldTree alg (F a ts) = fun alg a $ map (foldTree alg) ts
```

Beispiele

```
sumA :: Num a => Tree a -> a
sumA = foldTree $ TreeSig {var = id, fun = \a as -> a+sum as}

preorder,postorder :: Tree a -> [a]
preorder  = foldTree $ TreeSig {var = single,        fun = \a ass -> a:concat ass}
postorder = foldTree $ TreeSig {var = single,        fun = \a ass -> concat ass++[a]}

tree1 = F 1 [F 2 [F 2 [V 3,V(-1)],V(-2)],F 4 [V(-3),V 5]]
```

preorder = depthfirst

```
sumA tree1       ⇝  11
preorder tree1   ⇝  [1,2,2,3,-1,-2,4,-3,5]
postorder tree1  ⇝  [3,-1,2,-2,2,-3,5,4,1]

arithA :: TreeSig String Int
arithA = TreeSig {var = \case "x" -> 5; "y" -> -66; "z" -> 13,
                  fun = \case "+" -> sum; "*" -> product}
```

```
tree2 = F "+" [F "*" [V "x",V "y"], V "z"]
```

```
foldTree arithA tree2   ⇝   -317
```

Tree(String) als universeller Datentyp

Jedes Element eines beliebigen Datentyps kann in einen Baum vom Typ *Tree(String)* übersetzt werden. Z.B. transformiert die Faltung *foldArith* in folgender *Arith*-Algebra *treeAlg* Ausdrücke des Typs *Exp(String)* in Bäume vom Typ *Tree(String)* (siehe Abschnitt 4.2):

```
treeAlg :: Show x => Arith x (Tree String)
treeAlg = Arith {con  = \i    -> int i,
                 var  = \x    -> V $ show x,
                 sum_ = \ts   -> F "Sum" ts,
                 prod = \ts   -> F "Prod" ts,
                 sub  = \t t' -> F "-" [t,t'],
                 div_ = \t t' -> F "/" [t,t'],
                 scal = \i t  -> F "*" [int i,t],
                 expo = \t i  -> F "^" [t,int i]}
          where int i = F (show i) []
```

Die Faltung *foldTree* in folgender *TreeSig*-Algebra *storeAlg(store)* belegt die Variablen in Ausdrücke des Typs *Tree(String)* mit ihren jeweiligen *store*-Werten:

```
storeAlg :: Read x => Store x -> TreeSig String Int
storeAlg store = TreeSig {var = store . read,
                          fun = \case "Sum" -> sum
                                      "Prod" -> product
                                      "-" -> \[i,k] -> i-k
                                      "/" -> \[i,k] -> i`div`k
                                      "*" -> \[i,k] -> i*k
                                      "^" -> \[i,k] -> i^k
                                      i -> const $ read i
```

Die Komposition von *foldArith(treeAlg)* mit der Faltung der berechneten Bäume in *storeAlg* liefert dieselben Ergebnisse wie *flip(eval)* (siehe Abschnitt 4.2):

$$\begin{aligned} flip(eval) \;=\; & flip(foldTree \circ storeAlg) \circ foldArith(treeAlg) \\ & :: (Read(x), Show(x)) \Rightarrow Exp(x) \to Store(x) \to Int \end{aligned}$$

5.11 Arithmetische Ausdrücke kompilieren (Haskell-Modul: Expr.hs)

Die Faltung arithmetischer Ausdrücke in der unten definierten *Arith*-Algebra *codeAlg* liefert Assemblerprogramme. *executeE* führt diese in einer Kellermaschine aus.

Die Zielkommandos sind durch folgenden Datentyp gegeben:

```
data StackCom x = Push Int | Load x | Add Int | Mul Int | Sub |Div|Up
```

Die (virtuelle) Zielmaschine besteht aus einem Keller für ganze Zahlen und einem Speicher (Menge von Variablenbelegungen) wie beim Interpreter arithmetischer Ausdrücke (s.o.). Genaugenommen beschreibt ein Typ für diese beiden Objekte nicht diese selbst, sondern die Menge ihrer möglichen Zustände:

```
type Estate x = ([Int],Store x)
```

Die Bedeutung der einzelnen Zielkommandos wird durch einen Interpreter auf *State* definiert:

```
executeCom :: StackCom x -> Estate x -> Estate x
executeCom (Push a) (stack,store) = (a:stack,store)
executeCom (Load x) (stack,store) = (store x:stack,store)
executeCom (Add n) st             = executeOp sum n st
executeCom (Mul n) st             = executeOp product n st
```

```
executeCom Sub st                    = executeOp (foldl1 (-)) 2 st
executeCom Div st                    = executeOp (foldl1 div) 2 st
executeCom Up st                     = executeOp (foldl1 (^)) 2 st
```

Die Ausführung eines arithmetischen Kommandos besteht in der Anwendung der jeweiligen arithmetischen Operation auf die obersten n Kellereinträge, wobei n die Stelligkeit der Operation ist:

```
executeOp :: ([Int] -> Int) -> Int -> Estate x -> Estate x
executeOp f n (stack,store) = (f (reverse as):bs,store)
                              where (as,bs) = splitAt n stack
```

Die Ausführung einer Kommandoliste besteht in der Hintereinanderausführung ihrer Elemente:

```
execute :: [StackCom x] -> Estate x -> Estate x
execute = flip $ foldl $ flip executeCom
```

Tatsächlich werden zwei Flips benötigt, um auf den Typ von *execute* zu kommen, wie die folgende Typableitung zeigt:

```
  flip executeCom          :: Estate x -> StackCom x -> Estate x

⊢ foldl $ flip executeCom :: Estate x -> [StackCom x] -> Estate x

⊢ flip $ foldl $ flip executeCom
                           :: [StackCom x] -> Estate x -> Estate x
```

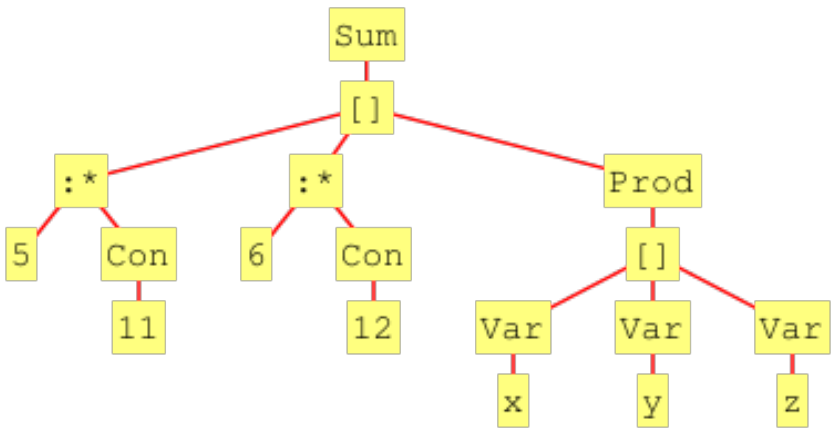
Die Übersetzung eines arithmetischen Ausdrucks von seiner Baumdarstellung in eine Befehlsliste erfolgt als Faltung in folgender *Arith*-Algebra (siehe Abschnitt 4.2):

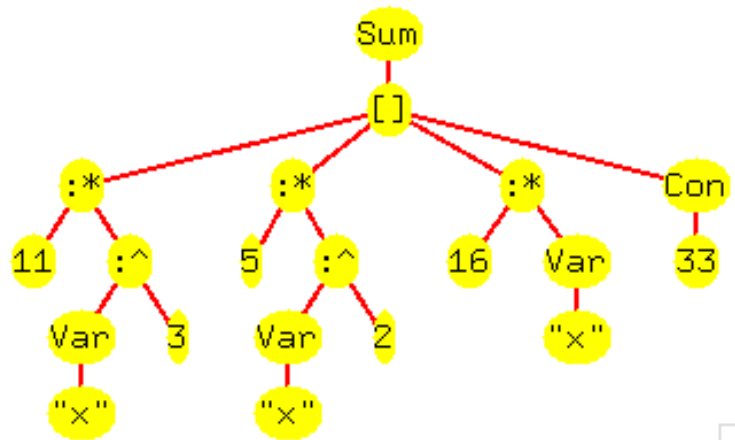
```
codeAlg :: Arith x [StackCom x]
codeAlg = ExpAlg {con  = \i -> [Push i],
                  var_ = \x -> [Load x],
                  sum_ = \es -> concat es++[Add $ length es],
                  prod = \es -> concat es++[Mul $ length es],
                  sub  = \e e' -> e++e'++[Sub],
                  div_ = \e e' -> e++e'++[Div],
                  scal = \i e -> Push i:e++[Mul 2],
                  expo = \e i -> e++[Push i,Up]}
```

$foldArith(codeAlg)$ ist **korrekt** in folgendem Sinn: Beginnt die Ausführung des Zielcodes eines Ausdrucks e im Zustand $(stack, store)$, dann endet sie im Zustand $(a : stack, store)$, wobei a der Wert von e unter der Variablenbelegung $store$ ist, d.h. es gilt die folgende Gleichung

$$execute(foldArith(codeAlg)(e))(stack, store) = foldArith(evalAlg)(e)(store) : stack, store) :: Estate(x)$$

Das lässt sich durch Induktion über den Aufbau von e zeigen.





Z.B. übersetzt *foldArith*(*codeAlg*) die oben als *Exp(String)*-Objekte dargestellten Ausdrücke

$$5 * 11 + 6 * 12 + x * y * z \quad \text{bzw.} \quad 11 * x\hat{}3 + 5 * x\hat{}2 + 16 * x + 33$$

in folgende Kommandosequenzen (in die noch Befehlsnummern eingefügt wurden):

```
0: Push 5
1: Push 11
2: Mul 2
3: Push 6
4: Push 12
5: Mul 2
6: Load "x"
7: Load "y"
8: Load "z"
9: Mul 3
10: Add 3
```

```
0: Push 11
1: Load "x"
2: Push 3
3: Up
4: Mul 2
5: Push 5
6: Load "x"
7: Push 2
8: Up
9: Mul 2
10: Push 16
11: Load "x"
12: Mul 2
13: Push 33
14: Add 4
```

5.12 Arithmetische Ausdrücke reduzieren (Haskell-Modul: Expr.hs)

Die folgende Funktion *reduceE* wendet folgende Gleichungen auf einen arithmetischen Ausdruck an:

$$
\begin{array}{lll}
0 + e = e & 0 * e = 0 & 1 * e = e \\
(m * e) + (n * e) = (m + n) * e & e^m * e^n = e^{m+n} & e^0 = 1 \\
m * (n * e) = (m * n) * e & (e^m)^n = e^{m*n} & e^1 = e
\end{array}
$$

Die Reduktion von Ausdrücken der Form $Sum[e_1, \ldots, e_n]$ oder $Prod[e_1, \ldots, e_n]$ erfordern ein Zustandsmodell zur schrittweisen Verarbeitung von Skalarfaktoren bzw. Exponenten:

```
type Rstate = (Int,[Exp x],Exp x -> Int)

updState :: Eq x => Rstate x -> Exp x -> Int -> Rstate x
updState (c,bases,f) e i = (c,insert e bases,update f e $ f e+i)

applyL :: ([a] -> a) -> [a] -> a
applyL _ [a] = a
applyL f as  = f as
```

Die Reduktionsfunktion kann damit wie folgt implementiert werden:

```
reduceE :: Eq x => Exp x -> Exp x
reduceE = \case e :- e'        -> reduceE e :- reduceE e'
                i :* Con j     -> Con $ i*j
                0 :* e         -> zero
                1 :* e         -> reduceE e
                i :* (j :* e)  -> reduceE $ (i*j) :* e
                i :* e         -> i :* reduceE e
                e :/ e'        -> reduceE e :/ reduceE e'
                Con i :^ j     -> Con $ i^j
                e :^ 0         -> one
                e :^ 1         -> reduceE e
                (e :^ i) :^ j  -> reduceE $ e :^ (i*j)
                e :^ i         -> reduceE e :^ i
                Sum es -> case f es of (c,[]) -> Con c
                                       (0,es) -> applyL Sum es
                                       (c,es) -> applyL Sum $ Con c:es
                Prod es -> case g es of (c,[]) -> Con c
                                        (1,es) -> applyL Prod es
                                        (c,es) -> c :* applyL Prod es
                e -> e
```

```
  where f es = (c,map summand bases) where
          (c,bases,scal) = foldl trans (0,[],const 0) $ map reduceE es
          summand e = if i == 1 then e else i :* e where i = scal e
          trans state@(c,bases,scal) = \case Con 0 -> state
                                             Con i -> (c+i,bases,scal)
                                             i:*e -> updState state e i
                                             e -> updState state e 1
        g es = (c,map factor bases) where
          (c,bases,expo) = foldl trans (1,[],const 0) $ map reduceE es
          factor e = if i == 1 then e else e :^ i where i = expo e
          trans state@(c,bases,expo) = \case Con 1 -> state
                                             Con i -> (c*i,bases,expo)
                                             e:^i -> updState state e i
                                             e -> updState state e 1
```

$reduceE(Sum(es))$ wendet $reduceE$ zunächst auf alle Ausdrücke der Liste es an. Dann wird die Ergebnisliste

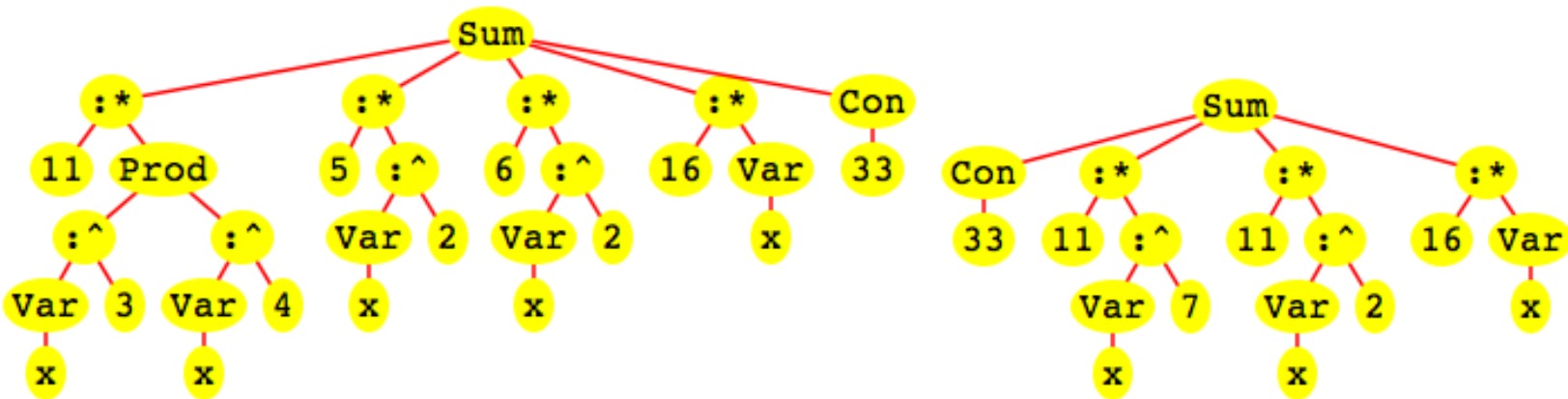
$$res = map(reduceE)(es),$$

ausgehend vom Anfangszustand $(0, [], const(0))$ mit der Zustandsüberführung $trans$ zum Endzustand $(c, bases, scal)$ gefaltet, der schließlich in eine reduzierte Summe der Elemente von res überführt wird.

Bei der Faltung werden gemäß der Gleichung $0 + e = e$ die Nullen aus res entfernt und alle Konstanten von res sowie alle Skalarfaktoren von Summanden mit derselben Basis gemäß der Gleichung $(m * e) + (n * e) = (m + n) * e$ summiert.

Im Endzustand $(c, bases, scal)$ ist c die Summe aller Konstanten von res und $bases$ die Liste aller Summanden von res. Die Funktion $scal : Exp(x) \to Int$ ordnet jedem Ausdruck e die Summe der Skalarfaktoren der Summanden von res mit der Basis e zu. Nur im Fall $c \neq 0$ wird $Con(c)$ in den reduzierten Summenausdruck eingefügt.

Demnach minimiert $reduceE$ die Liste es von Skalarprodukten eines Summenausdrucks $Sum(es)$. Analog minimiert $reduceE$ die Liste es von Potenzen eines Produktausdrucks $Prod(es)$.



Der Ausdruck `11*x^3*x^4+5*x^2+6*x^2+16*x+33` und seine reduzierte Form als $Exp(x)$-Objekte

reduceE ist **korrekt**, d.h. jeder Ausdruck e ist semantisch äquivalent zu seiner reduzierten Form, d.h. es gilt die Gleichung

$$\mathit{flip}(\mathit{eval}) \;=\; \mathit{flip}(\mathit{eval}) \circ \mathit{reduceE}.$$

Das lässt sich wieder durch Induktion über den Aufbau von e zeigen.

5.13 Resümee: Signaturen, Algebren und Faltungen von Kapitel 4 und 5

Signaturen

```
data List x val  = List {nil :: val, cons :: x -> val -> val}
data Arith x val = Arith {con       :: Int -> val,
                          var_      :: x -> val,
                          sum_,prod :: [val] -> val,
                          sub,div_  :: val -> val -> val,
                          scal      :: Int -> val -> val,
                          expo      :: val -> Int -> val}
data BinSig a val  = BinSig {empty_ :: val,
                             bjoin  :: a -> val -> val -> val}
data TreeSig a val = TreeSig {var :: a -> val,
                              fun :: a -> [val] -> val}
```

Termalgebra

```
listT :: List x [x]
listT = List {nil = [], cons = (:)}
```

```
intAlg:: List Int Int
intAlg = List {nil = 0, cons = (+)}
```

Termalgebra

```
arithT :: Arith x (Exp x)
arithT = Arith Con Var Sum Prod (:-) (:/) (:*) (:^)
```

```
evalAlg :: Arith x (Store x -> Int)
evalAlg = Arith {con  = const,
                 var  = \x       -> ($x),
                 sum_ = \bs st   -> sum $ map ($st) bs,
                 prod = \bs st   -> product $ map ($st) bs,
                 sub  = \b b' st -> b st-b' st,
                 div_ = \b b' st -> b st`div`b' st,
                 scal = \i b st  -> i*b st,
                 expo = \b i st  -> b st^i}
```

```
codeAlg :: Arith x [StackCom x]
codeAlg = ExpAlg {con  = \i -> [Push i],
                  var_ = \x -> [Load x],
                  sum_ = \es -> concat es++[Add $ length es],
                  prod = \es -> concat es++[Mul $ length es],
                  sub  = \e e' -> e++e'++[Sub],
                  div_ = \e e' -> e++e'++[Div],
                  scal = \i e -> Push i:e++[Mul 2],
                  expo = \e i -> e++[Push i,Up]}

treeAlg :: Show x => Arith x (Tree String)
treeAlg = Arith {con  = \i    -> int i,
                 var_ = \x    -> V $ show x,
                 sum_ = \ts   -> F "Sum" ts,
                 prod = \ts   -> F "Prod" ts,
                 sub  = \t t' -> F "-" [t,t'],
                 div_ = \t t' -> F "/" [t,t'],
                 scal = \i t  -> F "*" [int i,t],
                 expo = \t i  -> F "^" [t,int i]}
         where int i = F (show i) []
```

```
diffAlg :: Eq x => Arith x (x -> Exp x, Exp x)
diffAlg = Arith
     {con  = \i -> (\x -> zero, Con i),
      var_  = \x -> (\y -> if x == y then one else zero, Var x),
      sum_ = \fes x  -> let (fs,es) = unzip fes
                        in (\x -> Sum $ map ($x) fs, Sum es),
      prod = \fes x -> let (fs,es) = unzip fes
                           h x i e = Prod $ updList es i $ (fs!!i) x
                       in (\x Sum $ zipWith (h x) [0..] es, Prod es),
      sub  = \(f,e) (g,e') -> (\x -> f x :- g x, e :- e'),
      div_ = \(f,e) (g,e') -> (\x -> (Prod [f x,e'] :- Prod [e,g x])
                                     :/ (e' :^ 2), e :/ e'),
      scal = \i (f,e) -> (\x -> i :* f x, i :* e),
      expo = \(f,e) i -> (\x -> i :* Prod [f x,e:^(i-1)], e :^ i)}

arithA :: TreeSig String Int
arithA = TreeSig {var_ = \case "x" -> 5; "y" -> -66; "z" -> 13,
                  fun  = \case "+" -> sum; "*" -> product}

storeAlg :: Read x => Store x -> TreeSig String Int
storeAlg st = TreeSig {var  = st . read,
```

```
fun  = \case "Sum" -> sum
             "Prod" -> product
             "-" -> \[i,k] -> i-k
             "/" -> \[i,k] -> i`div`k
             "*" -> \[i,k] -> i*k
             "^" -> \[i,k] -> i^k
             i -> const $ read i
```

Faltungen

```
foldList :: List x val -> [x] -> val
foldList alg = \case []     -> nil alg
                     (x:s) -> cons alg x $ foldList alg s

foldArith :: Arith x val -> Exp x -> val
foldArith alg = \case Con i   -> con alg i
                      Var x   -> var_ alg x
                      Sum es  -> sum_ alg $ map f es
                      Prod es -> prod alg $ map f es
                      e :- e' -> sub alg (f e) (f e')
                      e :/ e' -> div_ alg (f e) (f e')
                      i :* e  -> scal alg i $ f e
```

```
                    e :^ i  -> expo alg (f e) i
                where f = foldArith alg

foldBin:: BinSig a val -> Bintree a -> val
foldBin alg Empty                 = empty_ alg
foldBin alg (Bjoin a left right) = bjoin alg a (foldBin alg left)
                                               (foldBin alg right)

foldTree :: TreeSig a val -> Tree a -> val
foldTree alg (V a)    = var alg a
foldTree alg (F a ts) = fun alg a $ map (foldTree alg) ts
```

Gültige Gleichungen

$$
\begin{aligned}
foldList(alg) &= foldr(cons(alg))(nil(alg)) \;\; :: [x] \to val \\
foldList(listT) &= id \;\; :: [x] \to [x] \\
foldArith(arithT) &= id \;\; :: Exp(x) \to Exp(x) \\
eval &= foldArith(evalAlg) \;\; :: Exp(x) \to Store(x) \to Int \\
diffE &= fst.foldArith(diffAlg) \;\; :: Exp(x) \to Exp(x) \\
eval &= flip(foldTree \circ storeAlg) \circ foldArith(treeAlg) \\
&\quad :: (Read(x), Show(x)) \Rightarrow Exp(x) \to Store(x) \to Int \\
eval &= eval \circ reduceE
\end{aligned}
$$

Korrektheit des Compilers von *Exp(x)* nach *[StackCom(x)]* (siehe Abschnitt 5.11)

Für alle $e \in Exp(x)$, $stack \in [Int]$ und $store \in Store(x)$ gilt

$$execute(foldArith(codeAlg)(e))(stack, store) \;=\; (eval(e)(store) : stack, store).$$

6 Fixpunkte, Graphen, Modallogik und logisches Programmieren

6.1 CPOs und Fixpunkte (Haskell-Modul: Examples.hs)

Die in diesem Kapitel behandelten Algorithmen basieren größtenteils auf Fixpunktberechnungen. Deshalb zunächst einige Grundbegriffe der Theorie, in der sich Fixpunkte iterativ berechnen lassen.

Eine reflexive, transitive und antisymmetrische Relation $\leq$ auf einer Menge A heißt **Halbordnung** und A eine **halbgeordnete Menge**, kurz: **Poset** (*partially ordered set*).

Eine **Kette** bzw. **co-Kette** von A ist eine abzählbare Teilmenge $\{a_i \mid i \in \mathbb{N}\}$ von A mit $a_i \leq a_{i+1}$ bzw. $a_i \geq a_{i+1}$ für alle $i \in \mathbb{N}$.

Ein Poset A mit Halbordnung $\leq$ ist **vollständig**, kurz ein **CPO** (*complete partial order*), wenn A ein kleinstes Element $\bot$ (*bottom*) und Suprema $\bigsqcup B$ aller Ketten B von A besitzt.

Ein Poset A mit Halbordnung $\leq$ ist **co-vollständig**, kurz ein **co-CPO** (*co-complete partial order*), wenn A ein größtes Element $\top$ (*top*) und Infima $\bigsqcap B$ aller co-Ketten B von A besitzt.

Ein Poset A mit Halbordnung $\leq$ heißt **vollständiger Verband** (*complete lattice*), wenn A Suprema und Infima beliebiger Teilmengen von A besitzt.

Ein vollständiger Verband A ist ein CPO und ein co-CPO, weil er mit $\bigsqcap A$ und $\bigsqcup A$ ein kleinstes bzw. größtes Element besitzt.

Beispiele

Bool ist ein vollständiger Verband mit Halbordnung $\{(b, c) \in \mathit{Bool} \mid b = \mathit{False} \vee c = \mathit{True}\}$ kleinstem Element *False*, größtem Element *True*, der Disjunktion als Supremum und der Konjunktion als Infimum.

Die Menge $\mathbb{Z}' =_{def} \mathbb{Z} \cup \{\infty, -\infty\}$ der ganzen Zahlen mit kleinstem und größtem Element (in Kapitel 4 durch `Int'` implementiert) ist ein vollständiger Verband mit der dort wie üblich definierten Halbordnung $\leq$ und dem Maximum bzw. Minimum einer Teilmenge von $\mathbb{Z}'$ als deren Supremum bzw. Infimum.

Die Potenzmenge $\mathcal{P}(A)$ einer Menge A ist ein vollständiger Verband mit der Mengeninklusion $\subseteq$ als Halbordnung, kleinstem Element $\emptyset$, größtem Element A, der Mengenvereinigung als Supremum und dem Mengendurchschnitt als Infimum. ❑

Eine Funktion $\Phi : A \to B$ zwischen zwei CPOs A und B heißt **stetig**, falls sie mit der Supremumsbildung verträglich ist, d.h. für alle Ketten C von A gilt:

$$\Phi(\bigsqcup C) = \bigsqcup\{\Phi(c) \mid c \in C\}.$$

Eine Funktion $\Phi : A \to B$ zwischen zwei co-CPOs A und B heißt **co-stetig**, falls sie mit der Infimumsbildung verträglich ist, d.h. für alle co-Ketten C von A gilt:

$$\Phi(\bigsqcap C) = \bigsqcap\{\Phi(c) \mid c \in C\}.$$

Aufgabe Zeigen Sie: Jede stetige oder co-stetige Funktion Φ ist **monoton**, d.h. für alle $a \in A$ gilt:

$$a \leq b \;\Rightarrow\; \Phi(a) \leq \Phi(b). \qquad \text{❑}$$

$a \in A$ heißt **Fixpunkt von** $\Phi : A \to A$, falls $\Phi(a) = a$ gilt.

Fixpunktsatz von Kleene

Sei $\Phi : A \to A$ stetig. $\mathit{lfp}(\Phi) =_{\mathit{def}} \bigsqcup_{i\in\mathbb{N}} \Phi^i(\bot)$ ist der (bzgl. $\leq$) kleinste Fixpunkt von Φ.

Sei $\Phi : A \to A$ co-stetig. $\mathit{gfp}(\Phi) =_{\mathit{def}} \bigsqcap_{i\in\mathbb{N}} \Phi^i(\top)$ ist der (bzgl. $\leq$) größte Fixpunkt von Φ. ❑

Φ wird deshalb auch **Schrittfunktion** (der Fixpunktberechnung) genannt.

Aus der Monotonie von Φ folgt $\Phi^i(\bot) \leq \Phi^{i+1}(\bot)$ und $\Phi^i(\top) \geq \Phi^{i+1}(\top)$ für alle $i \in \mathbb{N}$, so dass, falls A endlich ist, $i, k \in \mathbb{N}$ existieren mit $\Phi^i(\bot) = \Phi^{i+1}(\bot) = \mathit{lfp}(\Phi)$ und $\Phi^k(\top) = \Phi^{k+1}(\top) = \mathit{gfp}(\Phi)$.

Also können in diesem Fall kleinster und größter Fixpunkt von Φ mit folgendem Haskell-Programm berechnet werden:

```
fixpt :: (a -> a -> Bool) -> (a -> a) -> a -> a
fixpt le phi a = if b `le` a then a else fixpt le phi b
                 where b = phi a
```

6.2 CPO-Semantik rekursiver Gleichungen

Mit Hilfe des Fixpunktsatzes von Kleene wird z.B. gezeigt, dass eine Rekursionsgleichung wie z.B.

```
fact = \n -> if n > 1 then n*fact (n-1) else 1
```
(1)

tatsächlich eine Funktion $f : A \to B$ gibt, die sie in der gleichbenannten Funktions**variablen** löst. In (1) ist $A = B = \mathbb{N}$ und $f = fact$.

Um den Fixpunktsatz anwenden zu können, muss zunächst die gesamte Funktionsmenge $A \to B$ zu einem CPO erweitert werden.

Man beginnt mit dem **flachen CPO** $B_\perp =_{def} B \cup \{\perp_B\}$ **über** B, dessen Halbordnung wie folgt definiert ist: Für alle $a, b \in B_\perp$,

$$b \leq_B c \quad \Leftrightarrow_{def} \quad b = \perp_B \vee b = c.$$

Für beliebige Mengen A werden beliebige CPOs $B, B_1, \ldots, B_n$ wie folgt auf die Funktionsmenge $A \to B$ bzw. das Produkt $B_1 \times \cdots \times B_n$ fortgesetzt:

Für alle $f, g : A \to B$ und Ketten $K \subseteq C =_{def} (A \to B)$,

$$\begin{aligned} f \leq_C g &\Leftrightarrow_{def} \forall\, a \in A : f(a) \leq_B g(a), \\ \bot_C &=_{def} \lambda a.\bot_B, \\ \bigsqcup_C K &=_{def} \lambda a.\bigsqcup_B \{f(a) \mid f \in K\}. \end{aligned}$$

$\bigsqcup_C K$ ist wohldefiniert, weil die Menge $\{f(a) \mid f \in K\}$ eine Kette von B ist und damit ihr Supremum existiert.

Für alle $b = (b_1, \ldots, b_n),\ c = (c_1, \ldots, c_n) \in C =_{def} B_1 \times \cdots \times B_n$ und Ketten $K \subseteq B$,

$$\begin{aligned} b \leq_C c &\Leftrightarrow_{def} \forall\, 1 \leq i \leq n : b_i \leq_{B_i} c_i, \\ \bot_C &=_{def} (\bot, \ldots \bot), \\ \bigsqcup_C K &=_{def} (\bigsqcup_{B_1}\{b_1 \mid (b_1, \ldots, b_n) \in K\}, \ldots, \bigsqcup_{B_n}\{b_n \mid (b_1, \ldots, b_n) \in K\}). \end{aligned}$$

$\bigsqcup_C K$ ist wohldefiniert, weil für alle $1 \leq i \leq n$ die Menge $\{b_i \mid (b_1, \ldots, b_n) \in K\}$ eine Kette von B_i ist und damit ihr Supremum existiert.

Das kleinste Element des aus einer Summe $C =_{def} B_1 + \cdots + B_n$ von CPOs gebildeten CPOs kommt zu C hinzu: $C_\bot =_{def} C \cup \{\bot_C\}$ – so wie oben eine Menge B durch Hinzunahme von $\bot_B$ zum flachen CPO $B_\bot$ erweitert wurde. Das passt zur Darstellung endlicher Mengen als Summen einelementiger Mengen (siehe Abschnitt 2.2).

Für alle $b, c \in C_\perp$ Ketten $K \subseteq C_\perp$ und $K' =_{def} \{b \in B_i \mid (b,i) \in K,\ 1 \leq i \leq n\}$,

$$b \leq_C c \ \Leftrightarrow_{def}\ b = \perp_C \vee \exists\, 1 \leq i \leq n,\ b', c' \in B_i : b' \leq_{B_i} c' \wedge (b', i) = b \wedge (c', i) = c,$$

$$\bigsqcup_C K \ =_{def}\ \begin{cases} \perp_C & \text{falls } K = \{\perp_C\}, \\ \bigsqcup_{B_i} K' & \text{sonst.} \end{cases}$$

$\bigsqcup_{C_\perp} K$ ist wohldefiniert: Da K eine Kette von $B_\perp$ ist, gilt $i = j$ für alle $(b,i),(c,j) \in K$. Deshalb gibt es $1 \leq i \leq n$ mit $K' = \{b \in B_i \mid (b,i) \in K\}$. Also ist K' eine Kette von B_i und damit ihr Supremum existiert.

Gleichung (1) liefert die Schrittfunktion

$$\begin{aligned} \Phi_{fact} : (\mathbb{N} \to \mathbb{N}_\perp) &\to (\mathbb{N} \to \mathbb{N}_\perp) \\ f &\mapsto \lambda n.\mathit{if}\ n > 1\ \mathit{then}\ n * f(n-1)\ \mathit{else}\ 1 \end{aligned} \tag{2}$$

Φ_{fact} ist stetig, weil sich Multiplikation und Konditional – adäquat (!) – zu monotonen Funktionen auf $\mathbb{N}_\perp$ erweitern lassen.

Eine beliebige Schrittfunktion $\Phi : (A \to B) \to (A \to B)$ mit CPO B ist stetig, wenn die in der ursprünglichen Rekursionsgleichung verwendeten Hilfsfunktionen monoton bzgl. der Halbordnung von B sind (siehe [16], Theorem 5-1, oder [20], Satz 10.1.9).

Nach dem Fixpunktsatz von Kleene hat Φ den kleinsten Fixpunkt

$$\mathit{lfp}(\Phi) \;=\; \bigsqcup_{i\in\mathbb{N}} \Phi^i(\bot_{A\to B}). \tag{3}$$

Insbesondere ist $\mathit{lfp}(\Phi_{\mathit{fact}})$ die kleinste Lösung von Gleichung (1) in *fact*.

$\mathit{lfp}(\Phi)(a) = \bot_B$ modelliert den Fall, dass die vom Aufruf $\mathit{fixpt}(\leq)(\mathit{Phi})(\bot_{A\to B})(a)$ angestoßene Berechnung von $\mathit{lfp}(\Phi)(a)$ nicht terminiert. Bei $\Phi = \Phi_{\mathit{fact}}$ tritt er nicht auf, d.h. für alle $n \in \mathbb{N}$ gilt $\mathit{lfp}(\Phi)(n) \in \mathbb{N}$.

Beweis durch Induktion über n:

Für alle $n \in \{0, 1\}$ gilt $\mathit{lfp}(\Phi)(n) = \Phi(\mathit{lfp}(\Phi))(n) \overset{(2)}{=} 1$.

Für alle $n > 1$ ist $\mathit{lfp}(\Phi)(n) = \Phi(\mathit{lfp}(\Phi))(n) \overset{(2)}{=} n * \mathit{lfp}(\Phi)(n-1)$ eine natürliche Zahl, weil das nach Induktionsvoraussetzung für $\mathit{lfp}(\Phi)(n-1)$ gilt. ❏

Der kleinste Fixpunkt von Φ ist also eine Funktion von $\mathbb{N}$ nach $\mathbb{N}$. Andere Fixpunkte von Φ wären also Funktionen $f : \mathbb{N} \to \mathbb{N}_\bot$ mit $\mathit{lfp}(\Phi)(n) \leq_{\mathbb{N}} f(n)$ für alle $n \in \mathbb{N}$. Daraus folgt $f = \mathit{lfp}(\Phi)$ wegen $f(n) \in \mathbb{N} \cup \{\bot_{\mathbb{N}}\}$ und $\mathit{lfp}(\Phi)(n) \in \mathbb{N}$. $\mathit{lfp}(\Phi)$ ist also der *einzige* Fixpunkt von Φ ist und damit die einzige Lösung von (1), was endgültig die Behauptung rechtfertigt, dass *fact* durch (1) definiert wird.

Auch der Nachweis, dass eine Gleichung wie z.B.

$$\texttt{blink = 0:1:blink} \tag{4}$$

eine (unendliche) Liste definiert, kann durch Einbettung der Menge, in der mögliche Lösungen der Gleichung liegen sollen (hier $\mathbb{N} \to \mathbb{Z}$), in einen CPO und Anwendung des Fixpunktsatzes von Kleene geführt werden.

Wir beginnen wieder mit einem flachen CPO, nämlich $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$, setzen dessen Halbordnung wie oben auf die Menge

$$\mathit{Stream}_\perp =_{def} \{s : \mathbb{N} \to \mathbb{Z}_\perp \mid \forall n \in \mathbb{N} : s(n+1) \in \mathbb{Z} \Rightarrow s(n) \in \mathbb{Z}\}$$

fort und definieren das kleinste Element von $\mathit{Stream}_\perp$ und das Supremum einer Kette von Funktionen von $\mathit{Stream}_\perp$ wie oben. Gleichung (4) liefert folgende Schrittfunktion auf $\mathit{Stream}_\perp$:

$$\begin{array}{rcl} \Phi : \mathit{Stream}_\perp & \to & \mathit{Stream}_\perp \\ s & \mapsto & \lambda n.\mathit{if}\ n < 2\ \mathit{then}\ n\ \mathit{else}\ s(n-2) \end{array} \tag{5}$$

Da Φ stetig ist, hat Φ nach dem Fixpunktsatz von Kleene den kleinsten Fixpunkt

$$\mathit{lfp}(\Phi) = \bigsqcup_{i \in \mathbb{N}} \Phi^i(\perp).$$

M.a.W.: $\mathit{lfp}(\Phi)$ ist die kleinste Lösung von Gleichung (4) in der Variablen *blink*.

Wie oben gilt $\mathit{lfp}(\Phi)(n) \neq \perp$ für alle $n \in \mathbb{N}$. Also ist wieder $\mathit{lfp}(\Phi)$ der einzige Fixpunkt

von Φ und daher die einzige Lösung von (4). Folglich stimmt $lfp(\Phi)$ u.a. mit der erwarteten Lösung $blink =_{def} \lambda n.if\ even(n)\ then\ 0\ else\ 1$ von Gleichung (4) überein.

Weitere Beispiele unendlicher Listen und ihrer CPO-Semantik finden sich in Bird, Introduction to Functional Programming using Haskell, Kapitel 9.

Unendliche Bäume

Die Semantik unendlicher Listen als Suprema endlicher Approximationen kann auf unendliche Objekte eines beliebigen (konstruktiven) Datentyps fortgesetzt werden. Auch diese Objekte lassen sich partiell ordnen, wenn man sie als partielle Funktionen definiert:

Sei $\Sigma = (S, F)$ eine konstruktive Signatur mit Basismengen BS (siehe [21], Kapitel 2, oder [23], Kapitel 11).

Die $(BS \cup S)$-sortige Menge CT_Σ der **Σ-Bäume** besteht aus allen partiellen Funktionen

$$t : \mathbb{N}^* \to F \cup (\cup BS)$$

derart, dass gilt:

- für alle $B \in BS$, $CT_{\Sigma,B} = B$,
- für alle $s \in S$, $t \in CT_{\Sigma,s}$ gdw $ran(t(\epsilon)) = s$ und für alle $w \in \mathbb{N}^*$,

  $dom(t(w)) = e_1 \times \cdots \times e_n \to s \ \Rightarrow\ \forall\, 0 \leq i < n : (t(wi) \in e_{i+1} \vee ran(t(wi)) = e_{i+1})$.

Wir setzen voraus, dass es für alle $s \in S$ eine Konstante $\bot_s : \epsilon \to s$ in F gibt und definieren damit eine S-sortige Halbordnung auf CT_Σ: Für alle $s \in S$ und $t, u \in CT_{\Sigma,s}$,

$$t \leq u \;\Leftrightarrow_{def}\; \forall w \in \mathbb{N}^* : t(w) \neq \bot \Rightarrow t(w) = u(w).$$

Bezüglich dieser Halbordnung ist der Σ-Baum Ω_s mit

$$\Omega_s(w) =_{def} \begin{cases} \bot_s & \text{falls } w = \epsilon \\ \text{undefiniert} & \text{sonst} \end{cases}$$

das kleinste Element von $CT_{\Sigma,s}$. Außerdem hat jede Kette $t_1 \leq t_2 \leq t_3 \leq \ldots$ von Σ-Bäumen ein Supremum: Für alle $w \in \mathbb{N}^*$,

$$(\bigsqcup_{i \in \mathbb{N}} t_i)(w) \;=_{def}\; \begin{cases} t_i(w) & \text{falls } t_i(w) \neq \bot \text{ für ein } i \in \mathbb{N}, \\ \bot & \text{sonst.} \end{cases}$$

Rekursiv definierte Mengen kennen wir bereits aus Kapitel 4, wo konstruktorbasierte Datentypen als kleinste oder größte Lösungen von Mengengleichungen identifiziert wurden. Häufig gibt es eine (meist unendliche) Obermenge A, die es erlaubt, Teilmengen von A als Lösungen von Gleichungen im Potenzmengenverband $\mathcal{P}(A)$ (s.o.) darzustellen. So ist z.B. $\mathbb{N}$ im Fall $A = \mathbb{R}$ die kleinste Lösung von

$$Nat \;=\; \{0\} \cup \{x + 1 \mid x \in Nat\} \tag{6}$$

in Nat.

Die zugehörige Schrittfunktion lautet wie folgt:

$$\begin{array}{rcl} \Phi : \mathcal{P}(\mathbb{R}) & \to & \mathcal{P}(\mathbb{R}) \\ M & \mapsto & \{0\} \cup \{x+1 \mid x \in M\} \end{array} \tag{7}$$

Da Φ stetig ist, hat Φ nach dem Fixpunktsatz von Kleene den kleinsten Fixpunkt

$$\mathit{lfp}(\Phi) \;=\; \bigsqcup_{i \in \mathbb{N}} \Phi^i(\emptyset).$$

Man sieht sofort, dass $\mathbb{N}$ eine Lösung von (6) ist.

Sei $M \subseteq \mathbb{R}$ ein Fixpunkt von Φ, der $\mathbb{N}$ nicht enthält. Dann hat $\mathbb{N} \setminus M$ ein kleinstes Element n. Wegen $n \in \mathbb{N} \setminus M = \mathbb{N} \setminus \Phi(M)$ ist weder $n = 0$ noch gibt es $m \in M$ mit $n = m + 1$. Demnach gehört $n - 1 = m$ nicht zu M, im Widerspruch zur Voraussetzung, dass n das kleinste Element von $\mathbb{N} \setminus M$ ist. Daraus folgt $\mathbb{N} \subseteq M$, also $\mathit{lfp}(\Phi) = \mathbb{N}$.

Wegen $\mathcal{P}(\mathbb{R}) \cong (\mathbb{R} \to \mathit{Bool})$ ist die (auch Indikatorfunktion genannte) charakteristische Funktion $f : \mathbb{R} \to \mathit{Bool}$ von $\mathbb{N}$ die kleinste Lösung der Funktionsgleichung

$$\mathit{nat} \;=\; \lambda x.x = 0 \vee \mathit{nat}(x-1) \tag{8}$$

im Verband $\mathbb{R} \to \mathit{Bool}$, der wie $\mathbb{N} \to \mathbb{N}_\perp$ (s.o.) aus dem jeweiligen Werteverband (Bool bzw. $\mathbb{N}_\perp$) gebildet wird.

6.3 Semiringe

Als mathematische Modelle von Datentypen haben Semiringe eine ähnlich herausragende Bedeutung wie CPOs. Während ein CPO eine Ordnungsstruktur bildet, sind Semiringe algebraische Strukturen, also durch Operationen geprägt, die bestimmte Gleichungen erfüllen.

Ein **Semiring** R ist eine Menge mit einer Addition, einer Multiplikation, einer Null und einer Eins, die für alle $a, b, c \in R$ folgende Gleichungen erfüllen:

$a + (b + c) = (a + b) + c$	Assoziativität von $+$
$a + b = b + a$	Kommutativität von $+$
$0 + a = a = a + 0$	Neutralität von 0 bzgl. $+$
$a * (b * c) = (a * b) * c$	Assoziativität von $*$
$1 * a = a = a * 1$	Neutralität von 1 bzgl. $*$
$0 * a = 0 = a * 0$	Annihilierung von A durch 0
$a * (b + c) = (a * b) + (a * c)$	Linksdistributivität von $*$ über $+$
$(a + b) * c = (a * c) + (b * c)$	Rechtsdistributivität von $*$ über $+$

Ein **Ring** A hat außerdem additive Inverse. Aus deren Existenz kann man die Annihilierung von A durch 0 ableiten. Ist auch die Multiplikation kommutativ und haben alle $a \in R \setminus \{0\}$ multiplikative Inverse, dann ist R ein **Körper** (engl. **field**).

In einem **vollständigen Semiring** sind auch unendliche Summen definiert. Die obigen Gleichungen gelten entsprechend (siehe G. Karner, On Limits in Complete Semirings, oder

B. Mahr, A Bird's Eye View to Path Problems).

Alternativ zum vollständigen Semiring wird der Begriff der **Kleene-Algebra** verwendet. Hier werden anstelle beliebiger unendlicher Summen einstellige **Abschlussoperatoren** (*closure operators*) $^+$ (transitiver Abschluss) oder * (reflexiv-transitiver Abschluss) gefordert, die die Grundlage vieler Algorithmen auf Semiringen bilden und aus $a \in R$ die unendliche Summe aller endlicher Potenzen von a berechnen:

$$a^+ = a + a * a + a * a * a + \ldots, \qquad a^* = 1 + a^+.$$

In Haskell implementieren wir Semiringe als Instanzen der folgenden Typklasse:

```
class Semiring r where add,mul :: r -> r -> r
                       zero,one :: r

instance Semiring Bool where add = (||); mul = (&&)
                             zero = False; one = True

instance Semiring Int  where add = (+);  mul = (*)
                             zero = 0; one = 1

type BinRel a = [(a,a)]
```

```
instance (Eq a,Enum a,Bounded a) => Semiring (BinRel a) where
        add = union
        mul rel rel' = [(a,c) | (a,b) <- rel, (b',c) <- rel', b == b']
        zero = []
        one = [(a,a) | a <- [minBound..maxBound]]
```

siehe Abschnitt 5.1

Der transitive Abschluss R^+ einer Relation $R \subseteq A^2$

Die Schrittfunktion

$$\begin{aligned} \Phi : \mathcal{P}(A^2) &\to \mathcal{P}(A^2) \\ R' &\mapsto R + (R * R') \end{aligned}$$

ist stetig, hat also nach dem Fixpunktsatz von Kleene den kleinsten Fixpunkt $\bigsqcup_{i\in\mathbb{N}} \Phi^i(\emptyset)$. R^+ kann deshalb - falls A endlich ist - mit dem Fixpunktoperator von Abschnitt 6.1 wie folgt berechnet werden:

```
plus :: (Eq a,Enum a,Bounded a) => BinRel a -> BinRel a
plus rel = fixpt subset (add rel . mul rel) zero
```

Relationen als mehrwertige oder nichtdeterministische Funktionen

```
type NDfun a = a -> [a]

instance Eq a => Semiring (NDfun a) where
         add sucs sucs' = lift union sucs sucs'
         mul sucs sucs' = unionMap sucs' . sucs
         zero = const []
         one  = single
```

siehe Abschnitt 2.5

Der transitive Abschluss f^+ einer mehrwertigen Funktion $f : A \to \mathcal{P}(A)$

Die Schrittfunktion

$$\begin{aligned} \Phi : (A \to \mathcal{P}(A)) &\to (A \to \mathcal{P}(A)) \\ f' &\mapsto f + (f * f') \end{aligned}$$

ist stetig, hat also nach dem Fixpunktsatz von Kleene den kleinsten Fixpunkt $\bigsqcup_{i \in \mathbb{N}} \Phi^i(\lambda a.\emptyset)$. f^+ kann deshalb - falls A endlich ist - mit dem Fixpunktoperator von Abschnitt 6.1 wie folgt berechnet werden:

```
plusND :: Eq a => [a] -> NDfun a -> NDfun a
plusND nodes f = fixpt le (add f . mul f) zero where
                 le f g = all (lift subset f g) nodes
```

siehe Abschnitt 2.5

6.4 Graphen (Haskell-Modul: Examples.hs)

Die Funktion *rel2fun* : $(A \times B)^* \to (A \to B)$ in Abschnitt 3.8 überführt Assoziationslisten in semantische äquivalente Funktionen mit endlichem Definitionsbereich A. Man nennt sie auch kurz **endliche Funktionen**. *rel2fun* ist aber nicht injektiv, d.h. dieselbe endliche Funktion $f : A \to B$ hat i.d.R. viele Darstellungen als Assoziationsliste. Ist A (total) geordnet, dann gibt es jedoch genau eine *geordnete* Liste $s \in rel2fun^{-1}(f)$, in der keine zwei Paare mit derselben ersten Komponente vorkommen. Der Haskell-Typ *Map(A)(B)* besteht aus effizienten Repräsentationen genau solcher Assoziationslisten.

Zusammen mit seinen Funktionen steht *Map(A)(B)* im Modul `Data.Map.Strict`, den man am besten qualifiziert importiert:

```
import qualified Data.Map.Strict as DMS
```

Das vermeidet Namenskonflikte zwischen Map- und anderen im selben Programm verwendeten Funktionen. Z.B. ist dann das Map-Update

$$insert : A \to B \to Map(A)(B) \to Map(A)(B)$$

unter *DMS.insert* verfügbar.

Unmarkierte bzw. kantenmarkierte Graphen können als Map-Instanzen implementiert werden:

Unmarkierte Graphen: `type Graph a = DMS.Map a [a]`
Kantenmarkierte Graphen: `type GraphL a label = DMS.Map a [(label,a)]`

Ein Graph g wird hier als endliche Funktion dargestellt, die jedem Knoten von g die Liste seiner direkten Nachfolger zuordnet – bei kantenmarkiertem g zusammen mit der Markierung der in den Jeweiligen Nachfolger einlaufenden Kante.

`DMS.fromList` bildet eine Assoziationsliste vom Typ $(A \times B)^*$ in die semantisch äquivalente Map ab.

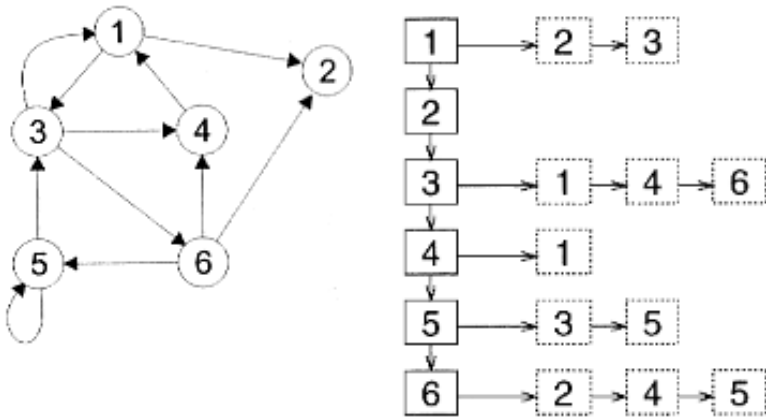
`DMS.keys(m)` bildet die Map m auf die geordnete Liste der (stets höchstens endlich vielen) Argumente von m ab.

`DMS.member(a)(m)` prüft, ob die Map m an der Stelle a definiert ist, d.h.

$$DMS.member(a)(m) \quad = \quad a \in DMS.keys(m).$$

Beispiele

```
graph1,graph2,graph3 :: Graph Int

graph1 = DMS.fromList [(1,[2,3]),(2,[]),(3,[1,4,6]),(4,[1]),(5,[3,5]),
                       (6,[2,4,5])]
```




```
graph2 = DMS.fromList $ (6,[]):[(a,[a+1]) | a <- [1..5]]

graph3 = DMS.fromList $ [(a,[a*10+1,a*10+2]) | a <- [1,11,12,112]] ++
                        [(a,[]) | a <- [111,121,122,1121,1122]]
```

Zugriff auf Map-Werte, Erzeugung und Update von Maps, Wiedergabe nichtdeterministischer Maps

```
apply :: Ord a => DMS.Map a b -> a -> b
apply = (DMS.!)

fun2Map :: Ord a => [a] -> (a -> b) -> Map a b
fun2Map as f = foldl h DMS.empty as where h m a = updMap m a $ f a

updMap :: Ord a => DMS.Map a b -> a -> b -> DMS.Map a b
updMap m a b = DMS.insert a b m

showNDMap :: (Ord a,Show a,Show b) => Map a [b] -> String
showNDMap m = concatMap f $ filter (not . null . apply m)
                          $ DMS.keys m
              where f a = '\n':show a++" -> "++show (apply m a)
```

```
showNDMap graph1  ⇝  1 -> [2,3]
                     3 -> [1,4,6]
                     4 -> [1]
```

```
                    5 -> [3,5]
                    6 -> [2,4,5]

showNDMap graph2  ⇝ 1 -> [2]
                    2 -> [3]
                    3 -> [4]
                    4 -> [5]
                    5 -> [6]

showNDMap graph3  ⇝ 1 -> [11,12]
                    11 -> [111,112]
                    12 -> [121,122]
                    112 -> [1121,1122]
```

Transformation der Map- in die Relationsdarstellung unmaekierter Graphen und umgekehrt

```
graph2rel :: Ord a => Graph a -> BinRel a
graph2rel g = [(a,b) | a <- DMS.keys g, b <- apply g a]
```

```
rel2graph :: Ord a => BinRel a -> Graph a
rel2graph = foldl f DMS.empty
            where f g (a,b) = updMap g a $ if a `member` g
                                           then insert b $ apply g a
                                           else [b]
```

Transitiver Abschluss

Die drei folgenden Funktionen berechnen den **transitiven Abschluss** eines Graphen G, also seine Erweiterung um alle Kanten (a, b), für die in G ein Weg von a nach b existiert.

```
fixClosure,treeClosure,warshall :: Ord a => Graph a -> Graph a

fixClosure g = fun2Map nodes $ plusND nodes $ apply g        (siehe 6.3)
               where nodes = DMS.keys g

treeClosure g = fun2Map nodes sucs
                where nodes = DMS.keys g
                      f = apply g
                      sucs = add f $ mul f sucs
```

treeClosure terminiert nicht auf Graphen mit Zyklen!

Floyd-Warshall-Algorithmus

```
warshall g = foldl trans g nodes
      where nodes = DMS.keys g
            trans g b = fold2 updMap g nodes $ map f nodes
                        where f a = if b `elem` ga
                                    then ga `union` apply g b else ga
                                    where ga = apply g a
```

warshall(g) berechnet eine Folge $(g_1, \ldots, g_n)$ unmarkierter Graphen, wobei g_{i+1} aus g_i entsteht, indem für alle Knotenpaare (a, b) mit $b \in apply(g_i)(a)$ $apply(g_i)(b)$ zu $apply(g_i)(a)$ hinzugefügt wird.

warshall berechnet den transitiven Abschluss mit Aufwand $O(n^3)$: Die äußere Faltung *foldl(trans)(g)(nodes)* durchläuft die Liste *nodes*; die innere Faltung *trans(g)(b)* durchläuft die Liste *map(f)(nodes)*. Jeder Aufruf $f(a)$ erzeugt die Faltung $ga \cup apply(g)(b)$, die im schlechtesten Fall ($apply(g)(b) = nodes$) ein weiteres Mal die Liste *nodes* durchläuft.

Beispiele

```
fixClosure/warshall graph1        ⇝  1 -> [1,2,3,4,5,6]
                                     3 -> [1,2,3,4,5,6]
                                     4 -> [1,2,3,4,5,6]
                                     5 -> [1,2,3,4,5,6]
                                     6 -> [1,2,3,4,5,6]

fixClosure/treeClosure/warshall graph2  ⇝  1 -> [2,3,4,5,6]
                                           2 -> [3,4,5,6]
                                           3 -> [4,5,6]
                                           4 -> [5,6]
                                           5 -> [6]

fixClosure/treeClosure/warshall graph3
                                  ⇝  1 -> [11,12,111,112,1121,1122,121,122]
                                     11 -> [111,112,1121,1122]
                                     12 -> [121,122]
                                     112 -> [1121,1122]
```

6.5 Semantik modallogischer Formeln

Graphen repräsentieren binäre (oder, falls sie kantenmarkiert sind, ternäre) Relationen. Demnach sind auch die in der LV Logik für Informatiker behandelten Kripke-Strukturen Graphen: Zustände ("Welten") entsprechen den Knoten, Zustandsübergänge den Kanten des Graphen. Hinzu kommt eine Funktion, die jedem Zustand eine Menge *lokaler* atomarer Eigenschaften zuordnet. Dementsprechend liefern die Werte dieser Funktion Knotenmarkierungen.

Modallogische Formeln beschreiben lokale, aber vor allem auch *globale* Eigenschaften von Zuständen, das sind Eigenschaften, die von der gesamten Kripke-Struktur $\mathcal{K}$ abhängen. Um eine modallogische Formel φ so wie einen anderen Ausdruck auswerten zu können, weist man ihr folgende – vom üblichen Gültigkeitsbegriff abweichende, aber dazu äquivalente – Semantik zu: φ wird interpretiert als die Menge aller Zustände von $\mathcal{K}$, die φ erfüllen sollen.

Zu diesem Zweck definieren eine **Kripke-Struktur** $\mathcal{K}$ als Quadrupel

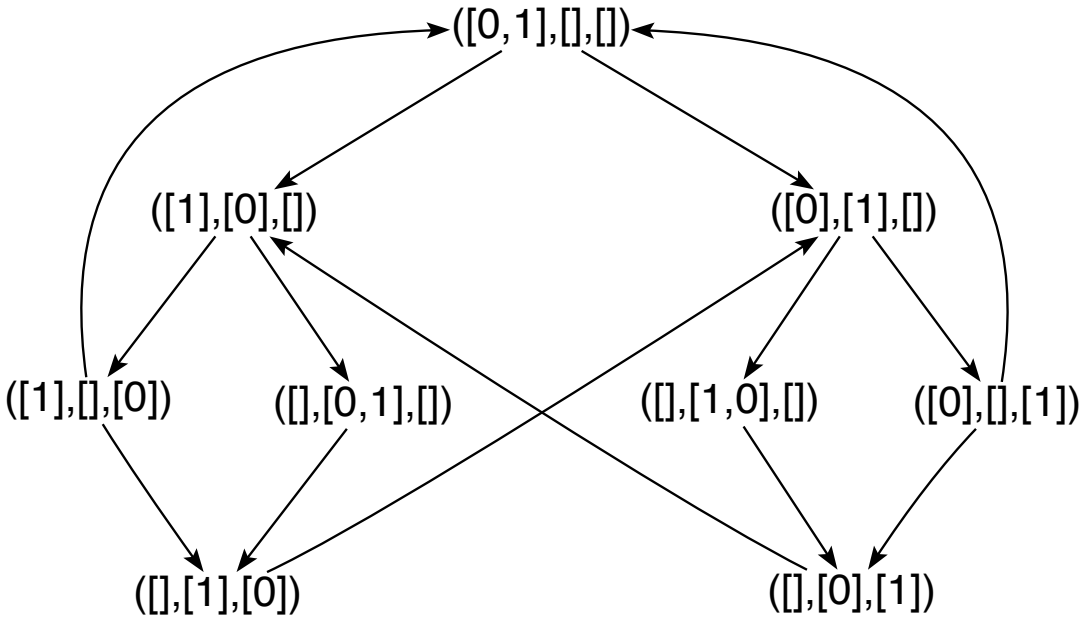
$$(State,\ Atom,\ trans,\ atoms),$$

bestehend aus einer **Zustandsmenge** $State$, einer Menge $Atom$ **atomarer Formeln**, einer **Transitionsfunktion** $trans : State \to \mathcal{P}(State)$, die jedem Zustand von $State$ die Menge seiner möglichen Nachfolger zuordnet, und einer Funktion

$$atoms : State \to \mathcal{P}(Atom),$$

die jeden Zustand auf die Menge seiner atomaren Eigenschaften abbildet.

Beispiel Mutual exclusion (siehe z.B. [13], Example 3.3.1)



Ein Zustand besteht aus drei Listen: 1. den Prozessen, die den *kritischen Abschnitt* nicht betreten wollen; 2. der Schlange der Prozesse, die auf ihren Einlass warten; 3. den Prozessen im kritischen Abschnitt. Dieser darf nur höchstens einen Prozess enthalten. Der Graph stellt δ dar für den Fall, dass es genau zwei Prozesse gibt. In diesem Fall könnten für $i = 1, 2$ die Atome n_i, w_i und c_i gewählt werden mit folgender Bedeutung: Prozess i tut nichts, wartet auf Einlass in den bzw. befindet sich im kritischen Abschnitt. ❑

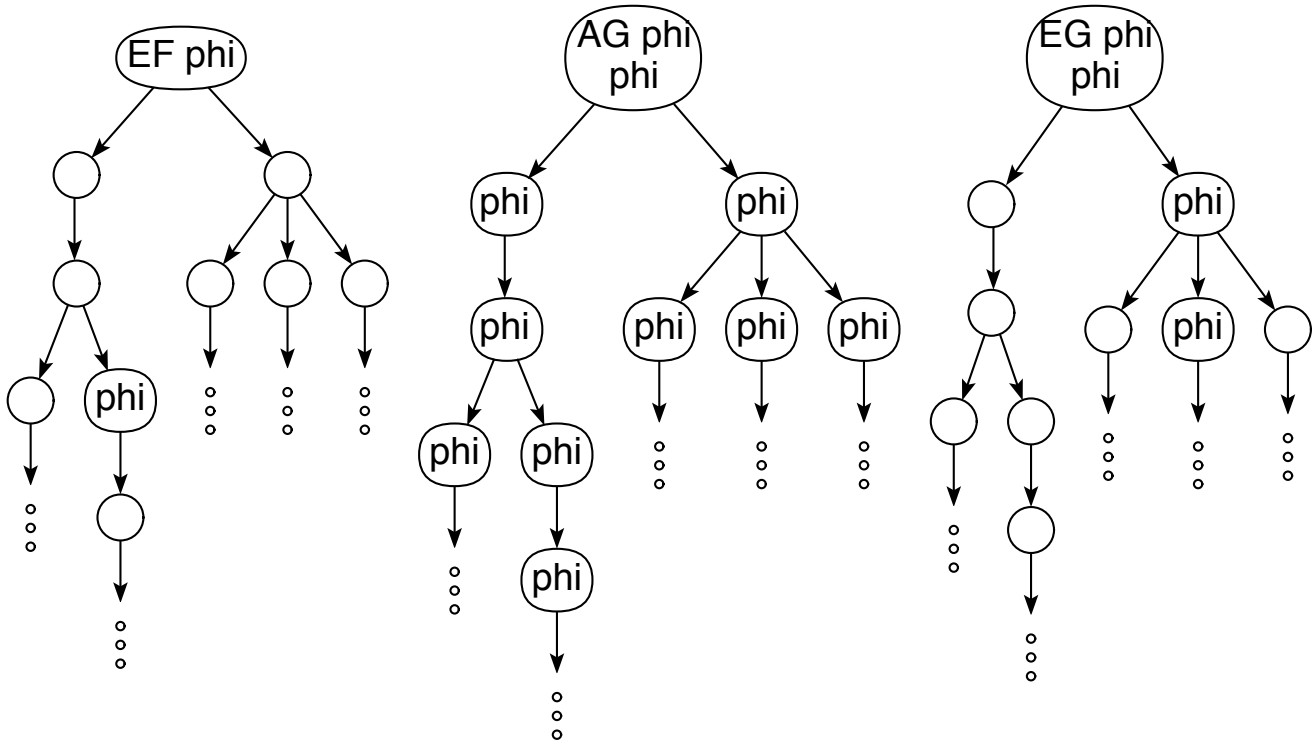
Formeln, die Eigenschaften des Systems beschreiben, sollen mit den modallogischen Operatoren von CTL (computation tree logic) gebildet werden. Die Semantik der Operatoren definieren wir (analog zu [17]) unter Zuhilfenahme dem μ-Kalkül ähnlicher Fixpunktoperatoren. Die entsprechende Formelmenge MF ist induktiv definiert:

Sei V eine Menge von Variablen.

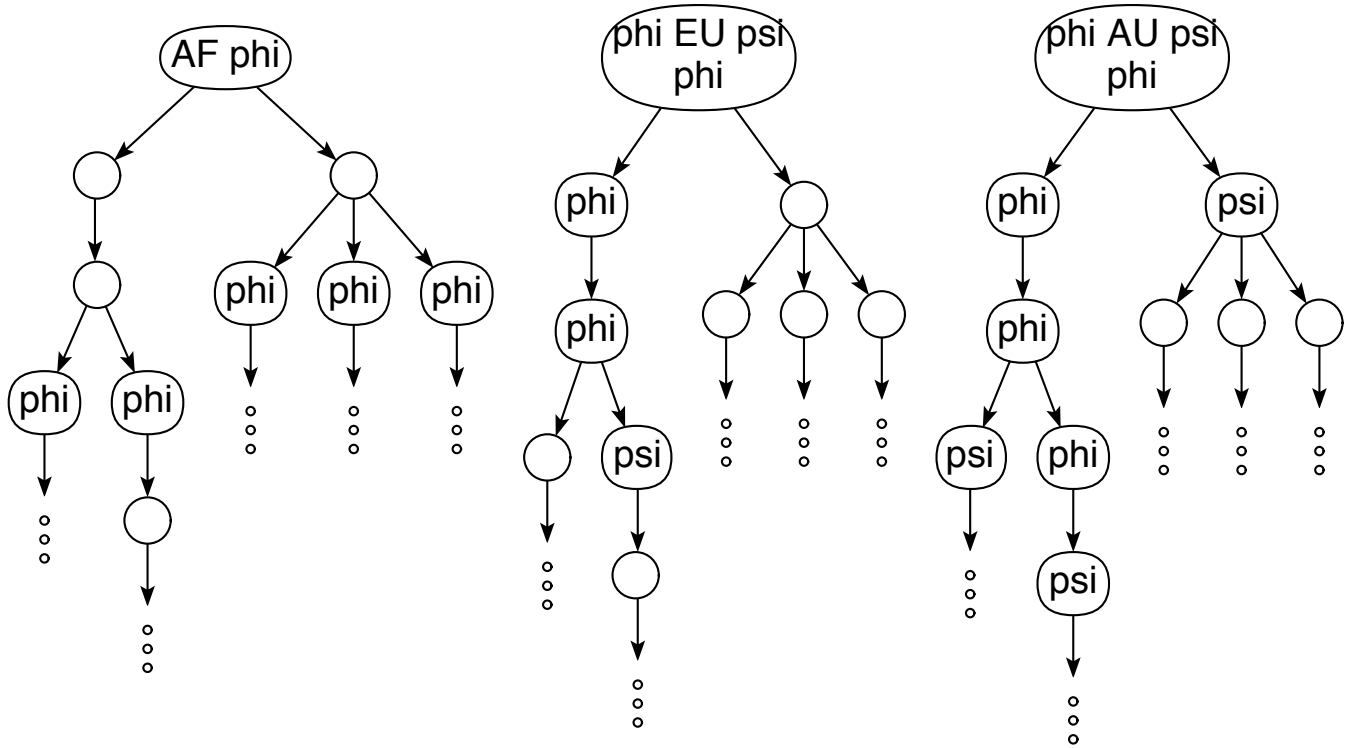
$$\begin{array}{lll} \{True, False\} \cup Atom \cup V & \subseteq & MF, \\ \varphi, \psi \in MF & \Rightarrow & \neg\varphi, \varphi \wedge \psi, \varphi \vee \psi, EX\varphi, AX\varphi \in MF, \\ x \in V \wedge \varphi \in MF & \Rightarrow & \mu x.\varphi, \nu x.\varphi \in MF. \qquad (\mu\text{-Formeln}) \end{array}$$

Alle anderen CTL-Formeln sind spezielle μ-Formeln:

$$\begin{array}{lcll} EF\varphi & = & \mu x.(\varphi \vee EX\ x) & \textit{exists finally} \\ AF\varphi & = & \mu x.(\varphi \vee (EX\,True \wedge AX\ x)) & \textit{always finally} \\ AG\varphi & = & \nu x.(\varphi \wedge AX\ x) & \textit{always generally} \\ EG\varphi & = & \nu x.(\varphi \wedge (AX\,False \vee EX\ x)) & \textit{exists generally} \\ \varphi EU\psi & = & \mu x.(\psi \vee (\varphi \wedge EX\ x)) & \textit{exists } \varphi \textit{ until } \psi \\ \varphi AU\psi & = & \mu x.(\psi \vee (\varphi \wedge AX\ x)) & \textit{always } \varphi \textit{ until } \psi \\ \varphi \Rightarrow \psi & = & \neg\varphi \vee \psi & \end{array}$$



Transitionsgraphen, deren Wurzeln $EF\varphi$, $AG\varphi$ bzw. $EG\varphi$ erfüllen



Transitionsgraphen, deren Wurzeln $AF\varphi$, $\varphi EU\psi$ bzw. $\varphi AU\psi$ erfüllen

Wie bei arithmetischen oder Booleschen Ausdrücken hängt die Auswertung modaler Formeln von einer Variablenbelegung ab, das ist hier eine Funktion des Typs

$$Store \quad = \quad V \to \mathcal{P}(State).$$

Für eine gegebene Kripke-Struktur $\mathcal{K} = (State, Atom, trans, value)$ ist die Auswertungsfunktion

$$eval : MF \to (Store \to \mathcal{P}(State))$$

daher wie folgt induktiv über der Struktur modallogischer Formeln definiert:

Sei $atom \in Atom$, $x \in V$, $\varphi, \psi \in MF$ und $st \in Store$.

$$\begin{array}{lcll}
eval(True)(st) & = & State, & \\
eval(False)(st) & = & \emptyset, & \\
eval(atom)(st) & = & \{s \in State \mid atom \in atoms(s)\}, & \\
eval(x)(st) & = & st(x), & \\
eval(\neg\varphi)(st) & = & State \setminus eval(\varphi)(st), & \\
eval(\varphi \wedge \psi)(st) & = & eval(\varphi)(st) \cap eval(\psi)(st), & \\
eval(\varphi \vee \psi)(st) & = & eval(\varphi)(st) \cup eval(\psi)(st), & \\
eval(EX\varphi)(st) & = & \{state \in State \mid trans(state) \cap eval(\varphi)(st) \neq \emptyset\}, & \textit{exists next} \\
eval(AX\varphi)(st) & = & \{state \in State \mid trans(state) \subseteq eval(\varphi)(st)\}, & \textit{for all next} \\
eval(\mu x.\varphi)(st) & = & \bigcup_{i \in \mathbb{N}} \Phi^i(\emptyset), & \\
eval(\nu x.\varphi)(st) & = & \bigcap_{i \in \mathbb{N}} \Phi^i(State). &
\end{array}$$

Die Schrittfunktion Φ ist hier wie folgt definiert:

$$\begin{aligned} \Phi : \mathcal{P}(State) &\to \mathcal{P}(State) \\ Q &\mapsto eval(\varphi)(st[Q/x]), \end{aligned}$$

wobei für alle $y \in V$,

$$st[Q/x](y) = \begin{cases} Q & \text{falls } x = y, \\ st(y) & \text{sonst.} \end{cases}$$

Ist *trans* bildendlich, d.h. hat jeder Zustand höchstens endlich viele direkte Nachfolger, und werden alle Vorkommen der gebundenen Variablen einer μ-Formel in deren Rumpf von einer geraden Anzahl von Negationen präfixiert, dann ist Φ stetig und co-stetig bzgl. der o.g. CPO-Struktur von Potenzmengen.

Also ist $eval(\mu x.\varphi)(st)$ nach dem Fixpunktsatz von Kleene der kleinste und $eval(\nu x.\varphi)(st)$ der größte Fixpunkt von Φ. Ist $State$ endlich, dann ist auch $\mathcal{P}(State)$ endlich, so dass er mit `fixpt` berechnet werden kann (siehe Abschnitt 6.1).

In ähnlicher Weise können Relationen zwischen Knoten von Dokumentbäumen als kleinste bzw. größte Fixpunkte passender Schrittfunktionen definiert und berechnet werden (siehe [23], Kapitel 28).

Beispiel Mutual exclusion (s.o.)

Jede der folgenden modalen Formeln φ gilt in *Mutex* (s.o), d.h.

$$eval(\varphi)(\lambda x.\emptyset) = State.$$

safety	Es befindet sich immer nur ein Prozess im kritischen Abschnitt. $\neg(c_1 \wedge c_2)$
liveness	Wenn ein Prozess um Einlass in den kritischen Abschnitt bittet, wird er diesen auch irgendwann betreten. $w_i \Rightarrow AFc_i$, $i = 1, 2$
non-blocking	Ein Prozess kann stets um Einlass in den kritischen Abschnitt bitten. $n_i \Rightarrow EXw_i$, $i = 1, 2$
no strict sequencing	Es kann vorkommen, dass ein Prozess nach Verlassen des kritischen Abschnitts diesen wieder betritt, bevor der andere Prozess dies tut. $\Diamond(c_i \wedge (c_i EU(\neg c_i \wedge (\neg c_j EU c_i))))$, $i = 1, 2$, $j = 1, 2$, $i \neq j$

Zustandsäquivalenz ist ebenfalls ein größter Fixpunkt. Die Schrittfunktion Φ ist hier wie folgt definiert:

$$\begin{aligned} \Phi : \mathcal{P}(State^2) &\to \mathcal{P}(State^2) \\ \sim &\mapsto \{(s, s') \in State^2 \mid atoms(s) = atoms(s'),\ trans(s) \sim trans(s')\}. \end{aligned}$$

Zwei Zustände s und s' heißen **äquivalent**, **verhaltensgleich** oder **bisimilär**, wenn (s, s') zu $gfp(\Phi)$ gehört.

Aufgabe Zeigen Sie, dass $gfp(\Phi)$ eine Äquivalenzrelation ist. ❑

Übrigens liefert der Quotient einer Kripke-Struktur nach ihrer Bisimilarität (wie der entsprechende Quotient eines endlichen Automaten) für jeden "Anfangszustand" von *State* die bzgl. der Anzahl ihrer Zustände minimale Struktur.

Aufgabe

Implementieren Sie die obige Modallogik in Haskell in drei Schritten:

- Geben Sie einen Datentyp für die Formelmenge *MF* an sowie Typen für Kripke-Strukturen und die Menge *Store*.

- Programmieren Sie die Auswertungsfunktion *eval* unter Verwendung der Funktionen *lfp* und *gfp* von Abschnitt 6.1. Verwenden Sie den Datentyp *Set* und die Mengenoperationen auf Listen.
- Testen Sie Ihre Implementierung an einigen Kripke-Strukturen aus einschlägiger Literatur, z.B. an *Mutex* (s.o.). oder [19], Beispiel 7.2. ❑

6.6 Vom logischen Programmieren zur μ-Abstraktion (Haskell-Modul: Lazy.hs)

Zur Berechnung von Lösungen – von Gleichungen oder anderen Formeln – sind *logische* oder *relationale* Programmiersprachen wie Prolog oder SQL entwickelt worden. Lösungen einer Formel φ, der **Zielformel** (*Goal*), werden aus ihr hergeleitet, indem ihre freien Variablen so mit Termen (Ausdrücken) belegt werden, dass – u.a. aus dem logischen Programm gebildete – Inferenzregeln angewendet werden können.

Die Regeln transformieren φ schrittweise in andere Formeln, bis hin zu Belegungen der freien Variablen, sagen wir: $x_1, \ldots, x_n$. Im Verifikationstool Expander2 wird eine Belegung von $x_1, \ldots, x_n$ als Konjunktion der Form $x_1 = t_1 \wedge \cdots \wedge x_n = t_n$ repräsentiert.

Dass die abgeleiteten Belegungen tatsächlich Lösungen von φ sind, ergibt sich aus der Korrektheit der angewendeten Inferenzregeln, die sicherstellt, dass aus der beschriebenen **logischen Reduktion** die Implikation

$$\forall\, x_1, \ldots, x_n : (x_1 = t_1 \wedge \cdots \wedge x_n = t_n \quad \Rightarrow \quad \varphi)$$

gefolgert werden kann. Mehr zu solchen Lösungsverfahren findet sich z.B. in [19], Kapitel 9. Die von Expander2 in logischen Reduktionen verwendeten Inferenzregeln (*Simplifikationen*, *Resolution*, *Coresolution*, *Narrowing*, *Induktion*, *Coinduktion*, ...) werden ausführlich in [24] behandelt – Resolution und Coresolution auch in [19].

Beispiel Client-Server-Interaktion 1 (siehe [11], Abschnitt 14.3)

```
iterate2 :: (a -> b) -> (b -> a) -> a -> [a]
iterate2 f g = iterate $ g . f
```

```
take 10 $ iterate2 (+1) (*2) 0  ⇝  [0,2,6,14,30,62,126,254,510,1022]
```

Die elementweise Erzeugung des Stroms *iterate2(f)(g)* soll auf einen Client und einen von diesem unabhängigen Server verteilt werden. Letzterer erhält vom Client einen Strom von *Requests* des Typs a, wendet $f : a \to b$ auf den ersten Request an und fügt das Ergebnis an einen Strom von *Responses* des Typs b an, auf dessen Elemente der Client g anwendet und die Ergebnisse dem Server schickt, usw.

Eine logische Reduktion des Goals

```
take(3)$RequestsL(+1)(*2)$0 = s                                    (1)
```

mit Expander2 erhält man durch abwechselnde Anwendung folgender zwei Gleichungen (in Expander2-Syntax) sowie *map*, *head* und *tail* betreffende partielle Auswertungsschritte:

```
RequestsL(f)(g) = ClientL(g)(0).map(f).RequestsL(f)(g)             (2)
ClientL(g)(a)$s = a:ClientL(g)(g$b)$s' <=== s = b:s'               (3)
```

Anwendungen von (2) entsprechen der Funktion des Servers (f), während Applikationen von (3) die Tätigkeit des Clients wiedergeben (g).

(2) kann unverändert in Haskell implementiert werden. Der Implikationspfeil einer Hornformel wie (3) entspricht Haskell's *where*:

```
ClientL g a s = a:ClientL g (g b) s'  where  b:s' = s              (4)
```

ClientL müsste in Haskell klein geschrieben werden, da *ClientL* kein Konstruktor ist.

Warum wird eine bedingte Gleichung verwendet und nicht die folgende unbedingte?

```
ClientL g a (b:s') = a:ClientL g (g b) s'                          (5)
```

Da zur Anwendung von (5) das dritte Argument von $ClientL$ das Pattern $b : s'$ matchen muss, ein Term der Form $ClientL(g)(a)(b : v)$ bei einer Reduktion von (1) aber nicht erreicht wird. (3) hingegen kann im Kontext einer Formel $\varphi(t)$, die einen Term t der Form $ClientL(g)(a)(u)$ enthält, auch dann auf t angewendet werden, wenn u keine Instanz von $b : s'$ ist.

Neben der Ersetzung von t durch die entsprechende Instanz t' der rechten Seite von (3) wird nämlich die gesamte Ergebnisformel $\varphi(t')$ mit der Prämisse von (3) konjunktiv verknüpft. Damit der Reduktionsschritt semantisch korrekt ist, müssen die dabei eingeführten "frischen" Variablen b und s' der Prämisse von (3) existenzquantifiziert und, falls sie in φ vorkommen, umbenannt werden.

Insgesamt besteht also die Anwendung von (3) darin, dass

$$\varphi(t) \quad \text{zu} \quad \varphi(t') \wedge \exists\, b, s' : u = b : s' \quad \text{reduziert.}$$

Hier können die einzelnen Schritte der logischen Reduktion von (1) nachvollzogen werden. Expander2 erzeugt auch ein Protokoll der Reduktion. *Any* für den Existenzquantor, *All* für den Allquantor. Der letzte Schritt der Reduktion besteht in der Anwendung des Lemmas

$$\exists\, b, b_1, b_2, s : \ map(h)\$RequestsL(f)(g)\$a = b : b_1 : b_2 : s,$$

dessen Gültigkeit unmittelbar klar sein dürfte, auf den Term

$$map(+1)\$RequestsL(+1)(*2)\$0. \qquad \blacksquare$$

Im Folgenden sind weitere von Expander2 erstellte Visualisierungen logischer oder algebraischer Reduktionen verlinkt.

Beispiel splitAt (siehe Abschnitte 3.1 und 3.2)

Die hier (und in Abschnitt 3.2) verlinkte Auswertung des Terms *splitAt(3)[5..12]* entspricht der hier verlinkten logischen Reduktion der Gleichung

$$\texttt{splitAt}(3)[5..12] = \texttt{s}. \tag{6}$$

Während eine Termauswertung aus Anwendungen *unbedingter* Gleichungen besteht, können in einer logischen Reduktion – wie im vorigen Beispiel beschrieben – auch bedingte Gleichungen, allgemeiner: Hornformeln, angewendet werden. Gemäß Abschnitt 3.2 ist die bedingte Gleichung

```
splitAt n (a:s) | n > 0 = (a:s1,s2)
                          where (s1,s2) = splitAt (n-1) s      (7)
```

äquivalent zur unbedingten

```
splitAt n (a:s) | n > 0 = (\(s1,s2) -> (a:s1,s2))
                          $ splitAt (n-1) s                    (8)
```

Die rechte Seite von (7) wird zusammen mit der Prämisse von (7) zur λ-Applikation von (8). Tatsächlich unterscheidet sich die Auswertung von der logischen Reduktion von (6) bzgl. der angewendeten Gleichungen nur darin, dass in letzterer (7) anstelle von (8) angewendet wird. ❑

Die Prämissengleichungen der Hornformeln (4) und (6) sind insofern nicht-rekursiv, als keine Variable auf beiden Seiten gleichzeitig vorkommt. Mehr noch: Die Variablen des Terms t auf der rechten Seite der Prämisse werden bei der Anwendung von (4) bzw. (6) mit Werten belegt und die entsprechende Instanz von t wird zu einem Wert u reduziert, der (hoffentlich) das Pattern auf der linken Seite matcht. Dabei werden nun auch dessen Variablen belegt, so dass die entsprechenden Instanzen von weiteren Prämissen oder der rechten Seite der Konklusion benutzt werden können.

Im Abschnitt 3.12 haben wir die Möglichkeit kennengelernt, (unendliche) Objekte durch rekursive Gleichungen zu spezifizieren, z.B. *blink*, *fibs* und *hamming*. Solche Gleichungen können auch als Prämissen bedingter Gleichungen Sinn machen. Ein Beispiel dafür ist die Prämisse der Definition von *star2* (siehe Abschnitt 3.12):

```
m = []:[a:as | as <- m, a <- b]
```

Auch hier ist die Lösung (in m) ein unendliches Objekt, nämlich die Liste aller endlichen Listen mit Elementen aus b, so dass keine Anwendung von *star2* terminiert – es sei denn, man bettet sie in eine Funktion wie *take* ein, die nur einen endlichen Teil des Objektes berechnet. Wir wollen hier aber rekursive Gleichungen betrachten, die wie die (nicht-rekursiven) Prämissen von (4) und (5) endliche Lösungen haben, die vollständig berechnet und dann in anderen Teilen der jeweiligen Funktionsdefinition verwendet werden können.

Soll also z.B. eine Gleichung der Form $x = f(t, x)$ in x gelöst werden, dann darf die Definition der Funktion f nicht von der Form ihres zweiten Arguments abhängen. Sonst kann $f(a, x)$ nicht ausgewertet, also $x = f(t, x)$ nicht gelöst werden. Man muss dann die Definition von f ändern.

Beispiel Iterativer Palindromtest (siehe [1], Abschnitt 3)

```
palI :: Eq a => [a] -> Bool
palI s = b where (r,b) = revEqI s r []                              (9)

revEqI :: Eq a => [a] -> [a] -> [a] -> ([a],Bool)
revEqI [] _ acc          = (acc,True)                               (10)
revEqI (x:s) (y:s') acc = (r, x == y && b)
                           where (r,b) = revEqI s s' $ x:acc        (11)
```

Da $revEqI(s)(s')(acc)$ das Paar $(reverse(s){+\!+}acc,\ s == s')$ berechnet, ist

$$r = reverse(s) \wedge b = (s == reverse(s))$$

die kleinste Lösung der Prämisse von (9) und damit ein für jede endliche Liste s definierter Wert. Die Reduktion eines Terms mit Muster $palI(a : t)$ liefert diesen Wert jedoch nicht, weil die Prämisse (*where*-Klausel) von (9) unlösbar ist. Dazu müsste nämlich (11) auf $revEqI(a:t)(r)[]$ anwendbar sein, wozu $r\ y:s'$ matchen müsste, was offenbar nicht geht.

M.a.W.: $palI(a : t)$ erhält im zugrundeliegenden CPO den Wert $\bot$ als **operationelle** Semantik, während die **denotationelle** Semantik von $palI(a : t)$ die kleinste Lösung der Prämisse von (9), also stets $\neq \bot$ ist. Um diese zu berechnen, muss die Prämisse offenbar anders ausgewertet werden als es der Fall wäre, wenn r nicht schon auf der rechten Seite der Prämissengleichung vorkäme. Zwei Möglichkeiten bieten sich an: eine *logische* Reduktion ähnlich der von (1) oder (6) oder eine *μ-Abstraktion*.

Eine logische Reduktion der Zielformel $palI\,[2, 3, 2] = b$ steht hier. Diese kann Gleichung (11) anwenden, weil in logischen Reduktionen Terme der Zielformel φ nicht gematcht werden müssen, sondern mit ihnen **unifiziert** werden dürfen, d.h. Variablen von φ dürfen durch Terme substituiert werden: $revEqI[2, 3, 2](r)[]$ wird zu $revEqI[2, 3, 2](y:s')[]$.

In der verlinkten Reduktion wird anstelle eines Datentyps für Boolesche Arithmetik die isomorphe Typ der natürlichen Zahlen 0 und 1 verwendet.

Expander2 unterscheidet nur zwei Typen: einen für logische Formeln und einen für algebraische Terme. Während wie üblich Terme Teile von Formeln sein können, macht der Konstruktor *bool* aus einer Formel einen Term.

Um bei der *Term*reduktion von *palI*[2,3,2] das oben beschriebene Problem zu vermeiden, ersetzen wir (11) zunächst durch die folgende bedingte Gleichung:

```
revEqI (x:s) s' acc = (r, x == head s' && b)
                      where (r,b) = revEqI s (tail s') $ x:acc      (12)
```

Analog zum Schritt von (7) zu (8) wird (12) in eine äquivalente unbedingte Gleichung überführt:

```
revEqI (x:s) s' acc = (\(r,b) -> (r, x == head s' && b))
                      (revEqI s (tail s') $ x:acc)                  (13)
```

(9) kann so nicht transformiert werden, weil die Variable r der Prämisse von (9) auf *beiden* Seiten der Gleichung auftaucht. Im Rahmen einer Termreduktion kann deren Lösung jedoch durch Auswertung einer aus (9) gebildeten μ-Abstraktion berechnet werden.

Eine **μ-Abstraktion** $\mu x.t$ bezeichnet die kleinste Lösung der Gleichung $x = t$, wobei x eine Variable und t ein Term ist. x ist die **gebundene** Variable, andere Variablen von t sind die **freien** (oder **globalen**) Variablen der μ-Abstraktion.

Sind $x_1, \ldots, x_n$ die freien Variablen von t, dann wird $\mu x.t$ — bzgl. jeweils zugrundeliegender CPOs $A, A_1, \ldots, A_n$ — als diejenige Funktion

$$Sem(\mu x.t) : A_1 \times \cdots \times A_n \to A$$

interpretiert, die $a \in A_1 \times \cdots \times A_n$ den kleinsten Fixpunkt $lfp(\Phi(a)) \in A$ der Schrittfunktion $\Phi(a) : A \to A$ zuordnet, die $b \in A$ auf den Wert desjenigen Terms abbildet, der aus t entsteht, wenn a komponentenweise für $(x_1, \ldots, x_n)$ und b für x eingesetzt werden.

Haskell verzichtet zwar auf das syntaktische Konstrukt der μ-Abstraktion, greift aber auf deren Semantik zurück, wenn es Werte einer durch

$$f(x_1, \ldots, x_n) = g(x) \;\; where \;\; x = t \tag{14}$$

definierten Funktion $f : A_1 \times \cdots \times A_n \to A$ — wie z.B. *palI* durch (9) — berechnet und die Variablen von t zu $\{x_1, \ldots, x_n, x\}$ gehören.

Durch (14) wird folgende Funktion definiert: Für alle $(a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$,

$$f(a_1, \ldots, a_n) = g(\mu x.t[a_i/x_i \mid 1 \leq i \leq n]). \tag{15}$$

$t[a/z]$ bezeichnet den Term, der aus t entsteht, wenn alle Vorkommen der Variablen z in t durch a ersetzt werden ("a für z").

Für $f = palI$ (siehe (9)) lautet (15) wie folgt:

$$palI(s) = \pi_2(\mu\ rb.revEqI(s)(\pi_1(rb))[]). \tag{16}$$

π_1 bezeichnet die i-te Projektion, die ein n-Tupel auf dessen erste Komponente abbildet. In (16) wird π_1 natürlich erst dann angewendet, wenn die Auswertung der μ-Abstraktion eine Tupelinstanz der Variable rb erzeugt hat.

Die Werte von $Sem(\mu x.t)$ können von einem iterativen Programm berechnet werden (wie z.B. `fixpt` in Abschnitt 6.1) oder durch eine Termreduktion, in der neben Gleichungen für die Funktionen von t folgende **Expansionsgleichung** angewendet wird:

$$\mu x.t \;=\; t[\mu x.t)/x]. \tag{17}$$

Die Gültigkeit von (17) ergibt sich aus der Fixpunkteigenschaft von $Sem(\mu x.t)$ (s.o.). In unserem Beispiel lautet (17) offenbar wie folgt:

$$\mu\ rb.revEqI(s)(\pi_1(rb))[] \;=\; revEqI(s)(\pi_1(\mu\ rb.revEqI(s)(\pi_1(rb))[])). \tag{18}$$

Damit Termreduktionen tatsächlich den kleinsten Fixpunkt von $x = t$ berechnen, müssen die Redizes von Gleichungsanwendungen gemäß der lazy- oder parallel-outermost-Strategie ausgewählt werden (siehe Kapitel 2).

Außerdem darf (17) nur dann zum Einsatz kommen, wenn keine andere Gleichung anwendbar ist (siehe [16], Abschnitt 5-2.2, oder [20], Abschnitt 10.3). Schließlich müssen die Funktionen von t bzgl. der Halbordnung auf A monoton sein.

Eine Reduktion von $\mu x.t$ terminiert, wenn die Anwendung von Gleichungen irgendwann alle Vorkommen von $\mu x.t$ zum Verschwinden bringt. Da jede Anwendung von (17) aber nur neue Vorkommen dieser Variablen erzeugt, sollte t mindestens eine Funktion f und eine Gleichung e für f enthalten, die mehr Variablen auf der linken als auf der rechten Seite hat.

Da die Anwendung von e der lazy-Strategie folgt, werden die Instanzen ihrer Variablen, sagen wir: $x_1, \ldots, x_n$, die links, aber nicht rechts vorkommen, nicht ausgewertet. Insbesondere verschwinden in den Instanzen enthaltene μ-Abstraktionen, so dass (17) zumindest darauf nicht mehr anwendbar ist.

Die Funktion f heißt in diesem Fall **nicht-strikt in** $x_1, \ldots, x_n$. So ist z.B. *if-then-else* im zweiten bzw. dritten Argument nicht-strikt, falls das erste Argument *False* bzw. *True* ist. In den jeweils zugrundeliegenden CPOs sind *if-then-else* und andere *sequentielle* Funktionen (siehe [29]) monoton und tragen deshalb dazu bei, dass die Reduktion von $\mu x.t$ tatsächlich den kleinsten Fixpunkt von $x = t$ berechnet (s.o.).

Leider genügt die Existenz von Funktionen mit nicht-strikten Argumenten in einer μ-Abstraktion noch nicht, um die Termination der Reduktion ihrer Aufrufe zu gewährleisten.

Solange keine Gleichungen für f, bei deren Anwendung Teilterme verschwinden, anwendbar sind, darf die Anwendung anderer Gleichungen für f nicht blockiert werden, weil z.B. Argumente von f nicht gematcht werden. Das wäre z.B. der Fall, wenn wir in unserem Beispiel $f = \mathit{revEqI}$ durch (10) und (11) anstelle von (10) und (12) definiert wäre.

Hier stehen die einzelnen Schritte einer Reduktion des Terms *palI[2,3,2]. geti* mit $i \in \mathbb{N}$ bezeichnet die Projektion π_{i+1}. Einige Redukte sind als Graphen dargestellt. Sie entstehen bei der Anwendung der Expansionsgleichung (18) — z.B. beim Schritt von palI232/012.png nach palI232/013.png. Zunächst werden nämlich alle Vorkommen der Variable *rb* durch Verzeigerung identifiziert, so dass bei der Anwendung von (18) nur eine Kopie der μ-Abstraktion erzeugt wird.

Beim Vergleich der Termreduktion mit der oben verlinkten logischen Reduktion von *palI[2,3,2]=b* fällt Folgendes auf:

- Wie beim *splitAt*-Beispiel entspricht die Termreduktion von *palI[2,3,2]* zu 1 der logischen Reduktion der Zielformel *palI[2,3,2]=b* zu ihrer Lösung $1 = b$.
- Anwendungen der Expansionsgleichung in der Termreduktion entsprechen in der logischen Reduktion echten Unifikationsschritten (s.o.), bei denen nicht wie beim Matching nur Variablen der jeweils angewendeten Gleichung, sondern auch solche der Zielformel oder ihrer Redukte substituiert werden.
- *splitAt* und *palI* sind zwar beide durch bedingte Gleichungen ((7) bzw. (9)) definiert. Die Bedingung von (7) ist aber nicht-rekursiv und kann deshalb in die λ-Applikation (8) umgewandelt werden, während die Prämisse von (9) — wie oben beschrieben — der Applikation einer μ-Abstraktion entspricht.

Im folgenden Beispiel werden zwei unendliche Listen wechselseitig-rekursiv definiert. Da die Elemente der einen von denen der anderen abhängen, müssen auch hier in einer Funktion anstelle des Listenmusters $x:s$ eine Variable und die Projektionen *head* und *tail* auf seine Komponenten x bzw. s verwendet werden.

Beispiel Client-Server-Interaktion 2

```
csi :: (a -> b) -> (b -> a) -> a -> [a]
csi f g a = requests where                                        (19)
            requests = client a $ map f $ requests
            client a s = a:client (g $ head s) (tail s)
take 10 $ csi (+1) (*2) 0  ⇝  [0,2,6,14,30,62,126,254,510,1022]
```

So wie in obigem Beispiel (9) als (16) zu verstehen ist, denotiert auch (19) eine μ-Abstraktion, diesmal mit drei freien Variablen (f, g und a) und der gebundenen Variable rc, die wie rb in (16) für ein Paar von Werten steht:

$$csi(f)(g)(a) = \pi_1(\mu\ rc.(\pi_2(rc)(a)(map(f))(\pi_1(rc)), \\ \lambda a.\lambda s.a : \pi_2(rc)(g(head(s)))(tail(s))) \tag{20}$$

Hier stehen die einzelnen Schritte einer Reduktion des Terms $take(3)(csi(+1)(*2)(0))$ mit Anwendungen der Expansionsgleichung für die μ-Abstraktion von (20). Wieder bezeichnet $geti$ mit $i \in \mathbb{N}$ die Projektion π_{i+1}. Außerdem entspricht $rsec(+, t)$ der Sektion $(+t)$.

In dieser Reduktion werden Variablen nicht nur vor Anwendungen der Expansionsgleichung, sondern auch vor der Ausführung von λ-Applikationen identifiziert. Dennoch enthalten viele Terme der Reduktion mehrere Kopien derselben μ-Abstraktion. Die entstehen bei der Reduktion von Teiltermen, in die aus dem Kontext Kanten hineinlaufen. Um beim jeweiligen Reduktionsschritt die Semantik des gesamten Terms zu erhalten, müssen die entsprechenden Zeiger vorher dereferenziert, d.h. die Zielterme kopiert werden.

Beispiel Operationen auf binären Bäumen mit Blatteinträgen

```
data Btree a = L a | Btree a :# Btree a
```

Wie *star2* und *palI* (siehe Abschnitte 3.12 bzw. 6.6), so berechnen auch die unten definierten Funktionen *replace*, *sortT* und *sortTI* auf binären Bäumen ihre Werte aus der Lösung einer Prämissengleichung.

```
replace :: (a -> a -> a) -> Btree a -> Btree a
replace f t = u where (x,u) = foldRep f t x

replace min ((L 3:#(L 22:#L 4)):#(L 2:#L 11))
                                          ⇝  ((2#(2#2))#(2#2))
replace (+) ((L 3:#(L 22:#L 4)):#(L 2:#L 11))
                                          ⇝  ((42#(42#42))#(42#42))
```

foldRep$(t)(x)$ ersetzt alle Blatteinträge von t durch x:

```
foldRep :: (a -> a -> a) -> Btree a -> a -> (a,Btree a)
foldRep _ (L x) y    = (x,L y)
foldRep f (t1:#t2) x = (f y z,u1:#u2) where (y,u1) = foldRep f t1 x
                                            (z,u2) = foldRep f t2 x
sortT :: Ord a => Btree a -> Btree a
sortT t = u where (ls,u,_) = leavesRep t (sort ls)

sort ((L 3:#(L 22:#L 4)):#((L 3:#(L 22:#L 4)):#(L 2:#L 11)))
                                  ⇝ ((2#(3#3))#((4#(4#11))#(22#22)))
```

leavesRep$(t)(s)$ liefert gleichzeitig die Blatteinträge von t, einen modifizierten Baum, in dem alle Blatteinträge von t durch die ersten Elemente der Liste s ersetzt sind, und die restlichen Elemente von s:

```
leavesRep :: Btree a -> [a] -> ([a],Btree a,[a])
leavesRep (L x) (y:s) = ([x],L y,s)
leavesRep (t1:#t2) s  = (ls1++ls2,u1:#u2,s2)
                        where (ls1,u1,s1) = leavesRep t1 s
                              (ls2,u2,s2) = leavesRep t2 s1
```

Die folgenden Versionen vermeiden Listenkonkatenationen:

```
sortTI :: Ord a => Btree a -> Btree a
sortTI t = u where (ls,u,_) = leavesRepI t (sort ls) []

leavesRepI :: Btree a -> [a] -> [a] -> ([a],Btree a,[a])
leavesRepI (L x) (y:s) acc = (x:acc,L y,s)
leavesRepI (t1:#t2) s acc  = (ls2,u1:#u2,s2)
                             where (ls1,u1,s1) = leavesRepI t1 s acc
                                   (ls2,u2,s2) = leavesRepI t2 s1 ls1
```

7 Funktoren und Monaden

7.1 Kinds: Typen von Typen

Typen erster Ordnung sind parameterlose Typen wie z.B. *Int*, *Exp(String)* und *Curves* (s.o.). Sie beschreiben einzelne Mengen.

Typen zweiter Ordnung wie z.B. `[ ]`, *Exp* (siehe 4.1), *Bintree* (siehe 5.4), *BintreeL* (siehe 5.4), *Tree* (siehe 5.9) und *Graph* (siehe 6.4). Sie beschreiben Funktionen, die jeder Menge eine Menge zuordnen. So ordnet z.B. *Bintree* einer Menge A eine Menge binärer Bäume zu, deren Knoteneinträge Elemente von A sind.

GraphL (siehe 6.4), *Array* (siehe Kapitel 8) und *GraphM* (siehe 8.4) sind Typen dritter Ordnung: Sie ordnen je zwei Mengen A und B eine Menge von Graphen mit Knotenmenge A und Kantenmarkierungen aus B bzw. die Menge der Funktionen von A nach B zu.

Dementsprechend werden auch Typvariablen erster, zweiter, dritter, ... Ordnung verwendet. In diesem Kapitel geht es hauptsächlich um Typklassen mit einer Typvariable höherer Ordnung, nämlich *Functor*, *Monad*, *MonadPlus*, *TreeC* und *Comonad*, und deren Instanzen.

Allgemein werden Typen nach ihren **Kinds** (englisch für Art, Sorte) klassifiziert:

Typen erster, zweiter oder dritter Ordnung haben den Kind $*$, $* \to *$ bzw. $* \to (* \to *)$.

Weitere Kinds ergeben sich aus anderen Kombinationen der Kind-Konstruktoren $*$ und $\to$. Z.B. ist $(* \to *) \to *$ der Kind eines Typs, der eine Funktion darstellt, die jedem Typ des Kinds $* \to *$ (also jeder Funktion *von* einer Menge von Mengen *in* eine Menge von Mengen) einen Typs des Kinds $*$ (also eine Menge) zuordnet.

Kinds erlauben es u.a., in Typklassen nicht nur Funktionen, sondern auch Typen zu deklarieren, z.B.:

```
class TK a where type T a :: *
                 f :: [a] -> T a
instance TK Int where type T Int = Bool
                      f = null
```

Eine Typklasse mit Typdeklarationen nennt man auch **Typfamilie**.

Alternativ kann die Typklasse um eine Typvariable t erweitert werden.

Die **funktionale Abhängigkeit** (*functional dependency*) $a \to t$ (a *bestimmt* t) wird dann wie folgt in die Klassendefinition eingebaut. Sie verbietet Instanzen von `TK` mit derselben Instanz von a, aber unterschiedlichen Instanzen von t.

```
class TK a t | a -> t where f :: [a] -> t
instance TK Int Bool where f = null
```

Kommen im Typ einer Funktion einer Typklasse nicht alle Typvariablen der Klasse vor, dann müssen die fehlenden von den vorkommenden abhängig gemacht werden.

Die Menge der im Programm definierten Instanzen einer Typklasse muss deren funktionale Abhängigkeiten tatsächlich erfüllen. So wäre z.B. neben der Instanz *TK(Int)(Bool)* keine Instanz *TK(Int)(Int)* erlaubt. Daher sind Typfamilien in der Regel funktionalen Abhängigkeiten vorzuziehen.

7.2 Funktoren (Haskell-Modul: Coalg.hs)

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

Wie der Name andeutet, verallgemeinert *fmap* die polymorphe Funktion

```
map :: (a -> b) -> [a] -> [b]
```

von Listen auf beliebige Datentypen. Umgekehrt bilden Listen eine Instanz von `Functor`:

```
instance Functor [ ] where fmap = map
```
Listenfunktor

```
fmap (+1) [1..4]  ⇝  [2,3,4,5]
```

Anforderungen an die Instanzen von `Functor`:

Für alle Mengen a, b, c, $f : a \to b$ und $g : b \to c$,

```
fmap id       =  id
fmap (g . f)  =  fmap g . fmap f
```

Im Folgenden verwenden wir manchmal das Schlüsselwort *newtype* anstelle von *data*. Dies ist immer dann erlaubt, wenn der jeweilige Datentyp genau einen Konstruktor und höchstens einen Destruktor hat.

Weitere Instanzen von *Functor*

```
newtype Id a = Id {run :: a}
```
Identitätsfunktor

```
run $ Id 12  ⇝  12
```

```
instance Functor Id where
         fmap f (Id a) = Id $ f a
```

```
fmap (+1) $ Id 12  ⇝  Id 13
```

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

siehe Kapitel 4

```
fmap (+1) $ Just 12  ⇝  Just 13
fmap (+1) $ Nothing  ⇝  Nothing
```

```
instance Functor Bintree where
         fmap f (Bjoin a left right)
                  = Bjoin (f a) (fmap f left) (fmap f right)
         fmap _ _ = Empty
```

siehe Abschnitt 5.9

```
fmap (+1) $ Bjoin 5 Empty $ leaf 7
         ⇝ Bjoin 6 Empty $ Bjoin 8 Empty Empty
```

```
instance Functor Tree where                          -- siehe Abschnitt 5.9
                 fmap = mapTree
fmap (+1) $ F 5 [V 9,V 7]  ⇝ F 6 [V 10,V 8]
```

Im Gegensatz zu `Tree` können die Repräsentationen von Funktionssymbolen bzw. Variablen bei folgendem Typ unterschiedlichen Mengen angehören:

```
data Term a b = FT a [Term a b] | VT b deriving Show        -- Termfunktor

instance Functor (Term a) where
                 fmap f (FT a ts) = FT a $ map (fmap f) ts
                 fmap f (VT b)    = VT $ f b
fmap (+1) $ FT 5 [VT 9,VT 7]  ⇝ F 5 [V 10,V 8]

instance Functor ((->) state) where                        -- Leserfunktor
                 fmap f h = f . h
fmap (+1) (+2) 5  ⇝ 8

instance Functor ((,) state) where                         -- Schreiberfunktor
                 fmap f (st,a) = (st,f a)
fmap (+1) (3,4)  ⇝ (3,5)
```

Kompositionen von Leser- und Schreiberfunktoren liefern *Zustandsfunktoren*:

```
newtype State state a = State {runS :: state -> (a,state)}
runS (State $ (+2)***(*3)) 5  ⇝  (7,15)
```

siehe Abschnitt 2.5

```
instance Functor (State state) where
         fmap f (State h) = State $ (\(a,st) -> (f a,st)) . h
runS (fmap (+1) $ State $ (+2)***(*3)) 5  ⇝  (8,15)
```

```
data Costate state a = (:#) {out :: state -> a, final :: state}
out ((+2):#5) 8  ⇝  10                  final ((+2):#5)  ⇝  5
```

```
instance Functor (Costate state) where
                  fmap f (h:#st) = (f . h):#st
out (fmap (+1) $ (+2):#5) 8  ⇝ 11
```

Da der Destruktor

$$runS \;:\; State(state) \;\rightarrow\; (state \rightarrow (a, state))$$

invers ist zum Konstruktor

$$State \;:\; (state \rightarrow (a, state)) \;\rightarrow\; State(state),$$

sind *State(state)* und *state → (a,state)* isomorphe Funktoren. Analog sind *Costate(state)* und *(state → a,state)* isomorph, weil die folgenden Funktionen invers zueinander sind:

```
c2wr :: Costate state a -> (state -> a,state)
c2wr (h:#st) = (h,st)
wr2c :: (state -> a,state) -> Costate state a
wr2c (h,st) = h:#st
```

7.3 Monaden und Plusmonaden (Haskell-Modul: Coalg.hs)

```
class Functor m => Monad m where
      return :: a -> m a                    Einbettung, unit
      (>>=)  :: m a -> (a -> m b) -> m b    sequentielle Komposition, bind
      (>>)   :: m a -> m b -> m b           bind ohne Wertübergabe
      fail   :: String -> m a               Wert im Fall eines Matchfehlers
      m >> m' = m >>= const m'

class Monad m => MonadPlus m where
      mzero :: m a                  scheiternde Berechnung   heißt zero in hugs
      mplus :: m a -> m a -> m a
                                    parallele Komposition    heißt (++) in hugs
```

Kurz gesagt, stellen Objekte vom Typ $m(a)$ Prozeduren dar, die Werte vom Typ a zurückgeben. Was das genau bedeutet, legt die jeweilige die Instanz von *Monad* bzw. *MonadPlus* fest.

MonadPlus gehört zum ghc-Modul `Control.Monad`.

Anforderungen an die Instanzen von *Monad* bzw. *MonadPlus*:

Für alle $m \in m(a)$, $f : a \to m(b)$ und $g : b \to m(c)$,

```
(m >>= f) >>= g     =  m >>= ((>>= g) . f)

m >>= return        =  m
(>>= f) . return    =  f

mzero >>= f         =  mzero
m >>= const mzero   =  mzero

mzero `mplus` m     =  m
m `mplus` mzero     =  m
```

Monadisches Lookup (vgl. 3.7)

```
lookupM :: (Eq a,MonadPlus m) => a -> [(a,b)] -> m b
lookupM a ((a',b):s) = if a == a'
                       then return b `mplus` lookupM a s
                       else lookupM a s
lookupM _ _          = mzero
```

Gemäß Abschnitt 3.7 gilt *lookupM* = *lookup* im Fall $m = Maybe$ und *lookupM* = *lookupL* im Fall $m =$ `[ ]`.

Monadisches Lifting (vgl. 2.5)

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f ma = ma >>= return . f

liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f ma mb = ma >>= \a -> mb >>= \b -> return $ f a b
```

liftM, *liftM2* und entsprechende Varianten für 3-, 4- bzw. 5-stellige Funktionen (*liftM3*, *liftM4*, *liftM5*) gehören zum ghc-Modul `Control.Monad`.

Die **do-Notation** bringt monadische Programme näher an die imperative Sicht, nach der ein Programm eine Folge von Variablenzuweisungen und anderen Kommandos ist. I.w. ersetzt sie die bind-Operatoren durch zwei polymorphe Funktionen:

```
(<-) :: Monad m => a -> m a -> m ()        Zuweisung
(;)  :: Monad m => m a -> m b -> m b       Sequentialisierungsoperator
```

Durch wiederholte Anwendung folgender Regeln kann jede **do-Sequenz** ms in einen äquivalenten monadischen Ausdruck mit bind-Operatoren und λ-Abstraktionen zurückübersetzt werden:

$$\begin{aligned}
\texttt{do}\ \{x \leftarrow m;\ ms\} &= m >\!\!>\!\!= \lambda x.\ \texttt{do}\ \{ms\} \\
\texttt{do}\ \{\mathit{let}\ \mathit{decls};\ ms\} &= \mathit{let}\ \mathit{decls}\ \mathit{in}\ \texttt{do}\ \{ms\} \\
\texttt{do}\ \{m;\ ms\} &= m >\!\!>\ \texttt{do}\ \{ms\} \\
\texttt{do}\ \{m\} &= m
\end{aligned}$$

m ist hier eine beliebiger Ausdruck eines monadischen Typs (ohne $\leftarrow$, *let* oder ;).

So entspricht z.B. die do-Sequenz

$$do\ a \leftarrow m_1;\ m_2;\ b \leftarrow m_3;\ a \leftarrow m_4;\ m_5;\ return(a, b)$$

für den monadischen Ausdruck

$$m_1 >\!\!>\!\!= (\lambda a.m_2 >\!\!>\ (m_3 >\!\!>\!\!= (\lambda b.m_4 >\!\!>\!\!= (\lambda a.m_5 >\!\!>\ return(a, b)))))$$

Die rechtsassoziative Klammerung ergibt sich automatisch aus den Typen der bind-Operatoren. Sie bestimmt die Gültigkeitsbereiche der Variablen: In m_2, m_3 und m_4 gilt der von m_1 erzeugte Wert von a; in m_4 und m_5 gilt der von m_3 erzeugte Wert von b; in m_5 gilt der von m_4 erzeugte Wert von a.

Durch das Semikolon voneinander getrennte monadische Objekte können auch linksbündig untereinander geschrieben werden.

Da Variablen, denen monadische Objekte zugewiesen werden, an λ-Abstraktionen gebunden sind, können an iher Stelle auch komplexe Muster stehen.

Passt bei der Ausführung einer Zuweisung $p \leftarrow m$ die Ausgabe von m nicht zum Muster p, dann wird anstelle der Zuweisung der – zur jeweiligen Monade gehörige – Wert von *fail(matchError)* zurückgegeben.

Ab Version 7.10 verlangt der Glasgow-Haskell-Compiler, dass *Monad*- und *MonadPlus*-Instanzen auch Instanzen von folgende Unterklassen *Applicative* bzw. *Alternative* von *Functor* sind:

```
class Functor f => Applicative f where
                   pure  :: a -> f a
                   (<*>) :: f (a -> b) -> f a -> f b
```

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

Deshalb haben *Monad* und *MonadPlus* im Prelude das Constraint *Applicative(m)* bzw. *Alternative(m)*. Für eine Monade bzw. Plusmonade M entsprechen *pure*, `(<*>)`, *empty* und `(<|>)` oben definierten monadischen Operationen:

```
instance Applicative M where pure  = return
                             (<*>) = liftM2 id

instance Alternative M where empty = mzero
                             (<|>) = mplus
```

Sei $f : A_1 \to \cdots \to A_{n+1}$ und für alle $1 \leq i \leq n$ sei $m_i \in M(A_i)$.

Die obigen Definitionen implizieren, dass eine do-Sequenz der Form

$$[do\ a_1 \leftarrow m_1;\ \ldots;\ a_n \leftarrow m_n;\ return(f(a_1)\ldots(a_n))\ ::\ M(A_{n+1}) \tag{1}$$

und die äquivalente bind-Sequenz

$$m_1 >\!\!>\!= \lambda a_1.m_2 >\!\!>\!= \lambda a_2.\ \ldots\ \lambda a_n.return(f(a_1)\ldots(a_n))\ ::\ M(A_{n+1}) \tag{2}$$

zu folgenden variablenfreien (!) Ausdrücken semantisch äquivalent sind:

$$liftM\,(f)(m_1) \mathbin{<\!*\!>} m_2 \mathbin{<\!*\!>} \cdots \mathbin{<\!*\!>} m_n \tag{3}$$

$$liftM2\,(f)(m_1)(m_2) \mathbin{<\!*\!>} m_3 \cdots \mathbin{<\!*\!>} m_n \tag{4}$$

$$liftM3\,(f)(m_1)(m_2)(m_3) \mathbin{<\!*\!>} m_4 \cdots \mathbin{<\!*\!>} m_n \tag{5}$$

Man beachte, dass (2) rechtsbündig, (3)-(5) aber linksbündig geklammert sind, z.B.:

$$(\ldots(liftM\,(f)(m_1) \mathbin{<\!*\!>} m_2) \cdots \mathbin{<\!*\!>} m_{n-1}) \mathbin{<\!*\!>} m_n$$

Oft wird anstelle von *liftM* die semantisch äquivalente Funktion (<$>) verwendet:

$$f \mathbin{<\!\$\!>} m_1 \mathbin{<\!*\!>} \cdots \mathbin{<\!*\!>} m_n$$

7.4 Monaden-Kombinatoren

```
guard :: MonadPlus m => Bool -> m ()
guard b = if b then return () else mzero
```

```
do guard True; m1; ...; mn     ist äquivalent zu     do m1; ...; mn
do guard False; m1; ...; mn    ist äquivalent zu     mzero
```

```
checkResult :: MonadPlus m => (a -> m b) -> (b -> Bool) -> a -> m b
checkResult f p a = do b <- f a; guard $ p b; return b

when :: Monad m => Bool -> m () -> m ()
when b m = if b then m else return ()                    bedingte Monade

(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
g <=< f = (>>= g) . f                                    Kleisli-Komposition

(>=>) = flip (<=<)

(=<<) = flip (>>=)                                       monadische Extension

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

In der Kategorientheorie wird eine Monade als Funktor mit *join* und *return* definiert und (>>=) wie folgt aus *join* abgeleitet:

```
(>>= f) = join . fmap f
```

Umgekehrt liefert jede Monade M eine *Functor*-Instanz:

```
instance Functor M where fmap = liftM
```

some(m) und *many*(m) wiederholen die Prozedur m solange, bis sie scheitert. Beide Funktionen listen die Ausgaben der einzelnen Iterationen von m auf:

```
some, many :: MonadPlus m => m a -> m [a]
some m = do a <- m; as <- many m; return $ a:as
many m = some m `mplus` return []
```

some(m) scheitert, wenn bereits die erste Iteration von m scheitert. *many*(m) scheitert in diesem Fall nicht, sondern liefert die leere Liste von Ausgaben.

msum setzt *mplus* von zwei Prozeduren auf Listen beliebig vieler Prozeduren fort:

```
msum :: MonadPlus m => [m a] -> m a
msum = foldr mplus mzero                          heißt concat in hugs
```

sequence(ms) führt die Prozeduren der Liste ms hintereinander aus. Wie bei *some*(m) und *many*(m) werden die dabei erzeugten Ausgaben aufgesammelt:

```
sequence :: Monad m => [m a] -> m [a]
sequence (m:ms) = do a <- m; as <- sequence ms; return $ a:as
sequence _      = return []                       heißt accumulate in hugs
```

Im Gegensatz zu $some(m)$ und $many(m)$ ist die Ausführung von $sequence(ms)$ erst beendet, wenn ms leer ist und nicht schon dann, wenn eine Wiederholung von m scheitert.

$sequence\_(ms)$ arbeitet wie $sequence(ms)$, vergisst aber die erzeugten Ausgaben.

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) $ return ()
```
heißt sequence in hugs

Die folgenden Funktionen führen die Elemente mit *map* bzw. *zipWith* erzeugter Prozedurlisten hintereinander aus:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f = sequence . map f

forM = flip mapM

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f = sequence_ . map f

forM_ = flip mapM_

zipWithM :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f s = sequence . zipWith f s
```

```
zipWithM_ :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
zipWithM_ f s = sequence_ . zipWith f s
```

7.5 Identitätsmonade

Die meisten der oben definierten Funktoren lassen sich zu Monaden erweitern.

```
instance Monad Id where return = Id
                        Id a >>= f = f a
```

Die Identitätsmonade dient dazu, Funktionsdefinitionen in eine prozedurale Form zu bringen:

Monadische Version von *foldBin* (siehe Abschnitt 5.10)

```
foldBinM :: val -> (a -> val -> val -> val) -> Bintree a -> Id val
foldBinM val _ Empty                    = return val
foldBinM val f (Bjoin a left right) = do valL <- foldBinM val f left
                                         valR <- foldBinM val f right
                                         return $ f a valL valR
```

7.6 Maybe-Monade

```
instance Monad Maybe where return = Just
                           Just a >>= f = f a
                           _ >>= _      = Nothing
                           fail _ = Nothing

instance MonadPlus Maybe where mzero = Nothing
                               Nothing `mplus` m = m
                               m `mplus` _       = m
```

Eine **partielle Funktion** $f : A \multimap B$ kann an manchen Stellen undefiniert sein. f wird in Haskell als (totale) Funktion vom Typ $A \to Maybe(B)$ implementiert, wobei den undefinierten Stellen der Wert *Nothing* zugeordnet wird.

Die üblichen Komposition zweier partieller Funktionen ist eine Instanz der oben definierten Kleisli-Komposition. In der Maybe-Monade gilt:

```
(g <=< f) a   =   f a >>= g
              =   case f a of Just b -> g b; _ -> Nothing
```

checkResult$(f)(p)$ (siehe Abschnitt 7.4) ist in der Maybe-Monade äquivalent zu:

```
case f a of Just b | p b -> return b; _ -> Nothing
```

Eine Fallunterscheidung der Form

```
case f a of Nothing -> g a; b -> b
```

ist in der Maybe-Monade äquivalent zu:

```
f a `mplus` g a
```

Beispiel

Die folgende Variante von *filter* wendet zwei Boolesche Funktionen f und g auf die Elemente einer Liste s an und ist genau dann definiert, wenn für jedes Listenelement x $f(x)$ oder $g(x)$ gilt. Im definierten Fall liefert *filter2(f)(g)(s)* das Listenpaar, das aus *filter(f)(s)* und *filter(g)(s)* besteht:

```
filter2 :: (a -> Bool) -> (a -> Bool) -> [a] -> Maybe ([a],[a])
filter2 f g (x:s) = do (s1,s2) <- filter2 f g s
                       if f x then Just (x:s1,s2)
                              else do guard $ g x; Just (s1,x:s2)
filter2 _ _ _     = Just ([],[])
```

❑

Für die monadische Multiplikation *join* (siehe Abschnitt 7.4) gilt in der Maybe-Monade:

```
join $ Just $ Just a   =   Just a
join _                 =   Nothing
```

lookupM(a)(s) (siehe Abschnitt 7.3) liefert in der Maybe-Monade die zweite Komponente des ersten Paares von s, dessen erste Komponente mit a übereinstimmt.

Arithmetische Ausdrücke partiell auswerten (Haskell-Modul: Expr.hs)

Wir modifizieren die *Arith*-Algebra *evalAlg* aus Abschnitt 4.2 zur Auswertung von *Arith*-Termen um die Erkennung von Divisionen durch 0:

```
evalMAlg :: Arith x (Store x -> Maybe Int)
evalMAlg = Arith {con  = const . Just,
                  var_ = \x        -> Just . ($x),
                  sum_ = \bs st    -> do is <- mapM ($st) bs
                                         Just $ sum is,
                  prod = \bs st    -> do is <- mapM ($st) bs
                                         Just $ product is,
                  sub  = \b b' st -> do i <- b st; k <- b' st
                                        Just $ i-k,
```

```
div_ = \b b' st -> do i <- b st; k <- b' st
                      guard $ k /= 0
                      Just $ div i k,
scal = \i b st  -> do k <- b st; Just $ i*k,
expo = \b i st  -> do k <- b st; Just $ k^i}
```

Für alle Ausdrücke $e \in Exp(x)$ und $st \in Store(x)$ gilt:

$$foldArith(evalMAlg)(e)(st) = \begin{cases} Nothing & \text{falls } e \text{ Divisionen durch 0 enthält,} \\ Just(foldArith(evalAlg)(e)(st)) & \text{sonst.} \end{cases}$$

7.7 Listenmonade

```
instance Monad [ ] where return a = [a]
                         (>>=) = flip concatMap
                         fail _ = []

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

Aus der Instanziierung von *mzero* und *mplus* folgt $msum = concat$ (siehe Abschnitt 7.4).

Eine **mehrwertige** oder **nichtdeterministische Funktion** $f : A \to \mathcal{P}(B)$ ordnet jedem Element von A eine Teilmenge von B zu. Ist diese immer endlich, dann kann – im Sinne der Darstellung endlicher Mengen durch endliche Listen (siehe Abschnitt 5.1) – f als Funktion des Typs $A \to B^*$ implementiert werden, wobei die leere Menge als leere Liste wiedergegeben wird.

Die üblichen Komposition zweier mehrwertiger Funktionen ist eine Instanz der oben definierten Kleisli-Komposition. In der Listenmonade gilt:

```
(g <=< f) a   =   f a >>= g
              =   concat [g b | b <- f a]
```

Eine Listenkomprehension der Form

```
[g b | a <- s, let b = f a, p b]
```

ist in der Listenmonade äquivalent zu:

```
do a <- s; let b = f a; guard $ p b; [g b]
```

Eine lokale Definition einer Variablen innerhalb eines monadischen Ausdrucks gilt stets bis zu ihrer nächsten lokalen Definition, falls es diese gibt, ansonsten bis zum Ende des Ausdrucks.

checkResult$(f)(p)$ (siehe Abschnitt 7.4) ist in der Listenmonade äquivalent zur Komprehension

```
[b | b <- f a, p b]
```

Die monadische Multiplikation *join* (siehe Abschnitt 7.4) fällt n der Listenmonade mit *concat* : $[[a]] \to [a]$ zusammen.

lookupM(a)(s) (siehe Abschnitt 7.3) liefert in der Listenmonade die jeweils zweite Komponente aller Paare von s, deren erste Komponente mit a übereinstimmt.

Kartesische Produkte

Die Listeninstanz des Monadenkombinators *sequence* (s.o.) hat den Typ $[[a]] \to [[a]]$ und liefert das – als Liste von Listen dargestellte – kartesische Produkt ihrer jeweiligen Argumentlisten:

$$sequence[as_1, \ldots, as_n] \;=\; [[a_1, \ldots, a_n] \mid a_i \in as_i,\; 1 \le i \le n] \;=\; as_1 \times \cdots \times as_n.$$

So gilt z.B.

```
    sequence $ replicate 3 [1..4]
=   sequence [[1..4],[1..4],[1..4]]
=   [[1,1,1],[1,1,2],[1,1,3],[1,1,4],[1,2,1],[1,2,2],[1,2,3],[1,2,4],
     [1,3,1],[1,3,2],[1,3,3],[1,3,4],[1,4,1],[1,4,2],[1,4,3],[1,4,4],
     [2,1,1],[2,1,2],[2,1,3],[2,1,4],[2,2,1],[2,2,2],[2,2,3],[2,2,4],
     [2,3,1],[2,3,2],[2,3,3],[2,3,4],[2,4,1],[2,4,2],[2,4,3],[2,4,4],
     [3,1,1],[3,1,2],[3,1,3],[3,1,4],[3,2,1],[3,2,2],[3,2,3],[3,2,4],
     [3,3,1],[3,3,2],[3,3,3],[3,3,4],[3,4,1],[3,4,2],[3,4,3],[3,4,4],
     [4,1,1],[4,1,2],[4,1,3],[4,1,4],[4,2,1],[4,2,2],[4,2,3],[4,2,4],
     [4,3,1],[4,3,2],[4,3,3],[4,3,4],[4,4,1],[4,4,2],[4,4,3],[4,4,4]]
```

Beispiel Permutationen

Die beiden in Kapitel 3 definierten mehrwertigen Funktionen *perms* und *permsI* zur rekursiven bzw. iterativen Berechnung der Liste aller Permutationen einer Liste haben z.B. die logischen Programmen entsprechenden monadische Implementierungen:

```
perms,permsI :: [a] -> [[a]]
perms [] = [[]]
perms s  = do i <- indices s
              s' <- perms $ take i s++drop (i+1) s
              [s!!i:s']
permsI s = loop s [] where
           loop :: [a] -> [a] -> [[a]]
           loop [] s = [s]
           loop s s' = do i <- indices s
                          loop (take i s++drop (i+1) s) $ s!!i:s'
```

Da *concatMap* der bind-Operator der Listenmonade ist, erkennt man sofort die semantische Äquivalenz der ursprünglichen Versionen von *perms* bzw. *permsI* mit der jeweiligen monadischen Implementierung.

Beispiel Damenproblem (Haskell-Modul: Examples.hs)

Jede Belegung (*valuation*) eines $(n \times n)$-Schachbrettes mit Damen wird als Liste *val* von n Zahlen aus der Menge $\{1, \ldots, n\}$ repräsentiert. An der Brettposition (i, j) steht genau dann eine Dame, wenn j der i-te Wert von *val* ist.

Die *Int*-Instanz der Schleifenfunktion *loop* in folgendem iterativen Algorithmus *queens* zur Berechnung aller sicheren Damenplatzierungen, also solcher, bei denen sich keine zwei Damen schlagen können, entspricht einem nichtdeterministischen Automaten mit der Zustandsmenge $\mathbb{Z}^* \times \mathbb{Z}^*$.

Jeder vom Anfangszustand $([1..n], [])$ aus erreichbare Zustand (s, val) besteht aus einer k-elementigen Liste s noch nicht vergebener Damenpositionen (= Spaltenindizes) mit $k \leq n$ und einer $(n - k)$ elementigen Liste *val* vergebener sicherer Damenpositionen in den $n - k$ unteren Zeilen des Schachbrettes.

```
queens :: Int -> [[Int]]
queens n = permsIC (safe 1) [1..n]

permsIC :: ([a] -> Bool) -> [a] -> [[a]]
permsIC c s = loop s [] where
              loop :: [a] -> [a] -> [[a]]
              loop [] s = [s]
```

```
        loop s s' = do i <- indices s
                       let new = s!!i:s'
                       guard $ c new
                       loop (take i s++drop (i+1) s) new
```

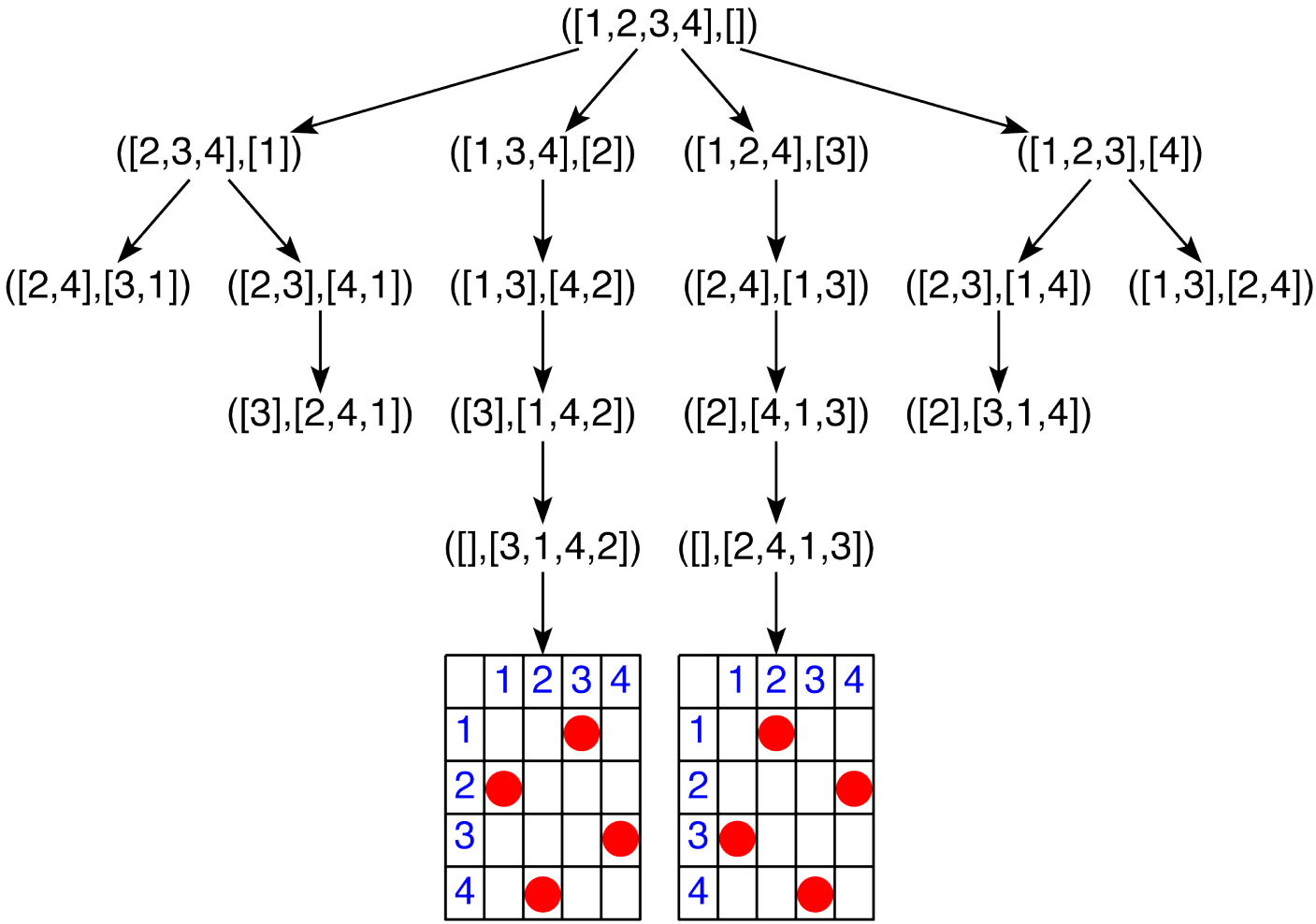
permsIC ist eine Variante von *permsI*, die nur jene Permutationen einer Liste erzeugt, die eine bestimmte Bedingung erfüllen. Hier sind das diejenigen, die sichere Damenplatzierungen repräsentieren.

```
safe :: Int -> [Int] -> Bool
safe i (k:col:val) = col-i /= k && k /= col+i && safe (i+1) (k:val)
safe _ _           = True
```

safe(1)(k:val) ist genau dann *True*, wenn unter der Voraussetzung, dass *val* eine sichere Damenplatzierung auf den unteren $n-1$ Brettzeilen repräsentiert, *k:val* eine sichere Damenplatzierung auf dem gesamten $(n \times n)$-Brett darstellt, d.h.: Es gibt keine Diagonale, auf der sowohl k als auch die Dame auf einer unteren Brettzeile stehen.

Beispiel

queens(4) erzeugt die folgenden Zustandsübergänge. Jeder von ihnen besteht in der die Hinzufügung einer Dame in der Zeile *über* den bereits vergebenen Damenpositionen.



7.8 Monadische Definition partieller und mehrwertiger Funktionen

Eine Menge $\mathcal{F}$ partieller Funktionen lässt sich oft durch ein System bedingter Gleichungen folgender Form definieren:

$$\begin{array}{lcl} f(p_1) = t_{11} & \Leftarrow & \bigwedge_{r=1}^{n_{11}} f_{11r}(t_{11r}) = p_{11r}, \\ & \vdots & \\ f(p_1) = t_{1m_1} & \Leftarrow & \bigwedge_{r=1}^{n_{1m_1}} f_{1m_1r}(t_{1m_1r}) = p_{1m_1r}, \\ & \vdots & \\ f(p_k) = t_{k1} & \Leftarrow & \bigwedge_{r=1}^{n_{k1}} f_{k1r}(t_{k1r}) = p_{k1r}, \\ & \vdots & \\ f(p_k) = t_{km_k} & \Leftarrow & \bigwedge_{r=1}^{n_{km_k}} f_{km_kr}(t_{km_kr}) = p_{km_kr}. \end{array} \tag{1}$$

Hier sind $p_1, \ldots, p_k, p_{111}, \ldots, p_{km_kn_{km_k}}$ Muster, $f, f_{111}, \ldots, f_{km_kn_{km_k}}$ Elemente von $\mathcal{F}$ und

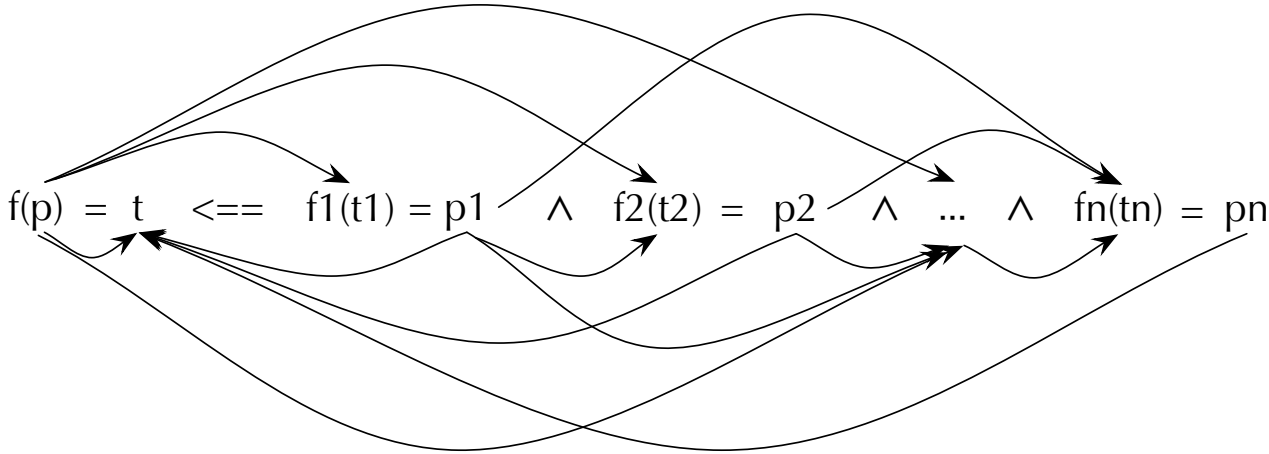
$$t_{11}, \ldots, t_{km_k}, t_{111}, \ldots, t_{km_kn_{km_k}}$$

beliebige Terme derart, dass für alle $1 \leq i \leq k$, $1 \leq j \leq m_i$ und $1 \leq r \leq n_{jr}$

$$V_0 = var(p_i) \quad \text{und} \quad V_r = V_{r-1} \uplus var(p_{ijr})$$

Folgendes gilt:

$$var(t_{ijr}) \subseteq V_{r-1} \quad \text{und} \quad var(t_{ij}) \subseteq V_{n_{ij}}.$$



Die geschwungenen Kanten beschreiben den Datenfluss bei der Anwendung einer der obigen Gleichungen $f(p) = t \Leftarrow \varphi$: Jede Variable eines Terms $u \in \{t, t_1, \ldots, t_n\}$ kommt in der Quelle einer auf u zeigenden Kante vor.

(1) kann in Haskell direkt implementiert werden (mit `where` anstelle von $\Leftarrow$ oder wie folgt in die Maybe-Monade eingebettet werden (siehe Abschnitt 7.6), was den Datenfluss verdeutlicht:

```
f :: A -> Maybe B                                                    (2)
f p_1 = msum [do p_111 <- f_111 t_111
                 ⋮
                 p_11n_11 <- f_11n_1 t_11n_11
                 return t_11,
```

```
                  ⋮
                  do p_1m_11 <- f_1m_11 t_1m_11
                     ⋮
                     p_1m_1n_11 <- f_1m_1_n_1m_1 t_1m_1n_1m_1
                     return t_1m_1]
⋮
f p_k = msum [do p_k11 <- f_k11 t_k11
                  ⋮
                  p_k1n_k1 <- f_k1n_1 t_k1n_1
                  return t_k1,
                  ⋮
                  do p_km_k1 <- f_km_k1 t_km_k1
                     ⋮
                     p_km_kn_km_k <- f_km_kn_km_k t_km_kn_km_k
                     return t_km_k]
f _ = mzero
```

(2) wie auch die direkte Implementierung von $\mathcal{F}$ realisiert partielle Funktionen, weil sie jede Grundinstanz von $f(x)$ mit $f \in \mathcal{F}$ zu *Nothing* oder einem Wert der Form $Just(t)$ ($= return(t)$) auswertet.

Man beachte, dass der Wert *Nothing* (= *mzero*) nicht nur dann entsteht, wenn die letzte Gleichung zur Anwendung kommt, sondern auch, wenn in einer vorangehenden Gleichung eine Zuweisung $p \leftarrow t$ scheitert, genauer gesagt: wenn der Wert von t nicht das Muster p matcht. Gemäß der Definition von *fail* und des bind-Operators der Maybe-Instanz von *Monad* bewirkt das Scheitern von $p \leftarrow t$ den Sprung zur darauffolgenden Gleichung (siehe Abschnitt 7.6).

Hiermit lassen sich auch Boolesche Constraints realisieren: Ist t vom Typ $Maybe(Bool)$ und hat t bei der Ausführung der Zuweisung $True \leftarrow t$ einen Wert $\neq True$, dann scheitert sie. $True \leftarrow t$ ist semantisch äquivalent zu $guard(t)$.

Beispiel Pfad zu Knoten a im Suchbaum ermitteln

```
getPath :: Ord a => a -> Bintree a -> Maybe [Int]
getPath _ Empty          = mzero
getPath a (Bjoin b t u) = msum [do guard $ a == b
                                   return [],
                                do guard $ a < b
                                   node <- getPath a t
                                   return $ 0:node,
                                do node <- getPath a u
                                   return $ 1:node]
```

Ob die von (2) berechnete Funktion für jedes Argument *genau* einen Wert berechnet, also einer totalen Funktion von A nach B entspricht, hängt von weiteren Kriterien ab, die in Abschnitt 3.13 und 4.2 behandelt werden.

Umgekehrt kann das obige monadische Programmschema jedoch immer zur Berechnung mehrwertiger Funktionen $f : A \to \mathcal{P}(B)$ verwendet werden (siehe Abschnitt 7.7). Dazu muss lediglich der Typ $A \to Maybe(B)$ durch $A \to [B]$ ersetzt werden. Die Korrektheit des Schemas für diese Verwendung ergibt sich folgenden in der Listenmonade gültigen Definitionen $msum = concat$, $mzero = []$, $fail = const[]$ und

$$(s >>= f) \;=\; concat(map(f)(s)).$$

Die letzte Definition bewirkt, dass bei der Ausführung einer Sequenz $do\ p \leftarrow f(t);\ m$ *alle* Werte von $f(t)$, die p matchen, von m weiterverarbeitet werden. Also liefert eine Sequenz $do\ m;\ return(t)$ die Liste *aller* Instanzen des Terms t, die sich aus der Belegung jeder seiner Variablen x mit den durch Zuweisungen von m erzeugten Werte von x ergeben.

Wegen $\mathcal{P}(A \times B) \cong (A \to \mathcal{P}(B))$ entspricht jede mehrwertige Funktion $f : A \to \mathcal{P}(B)$ einer Relation $R \subseteq A \times B$:

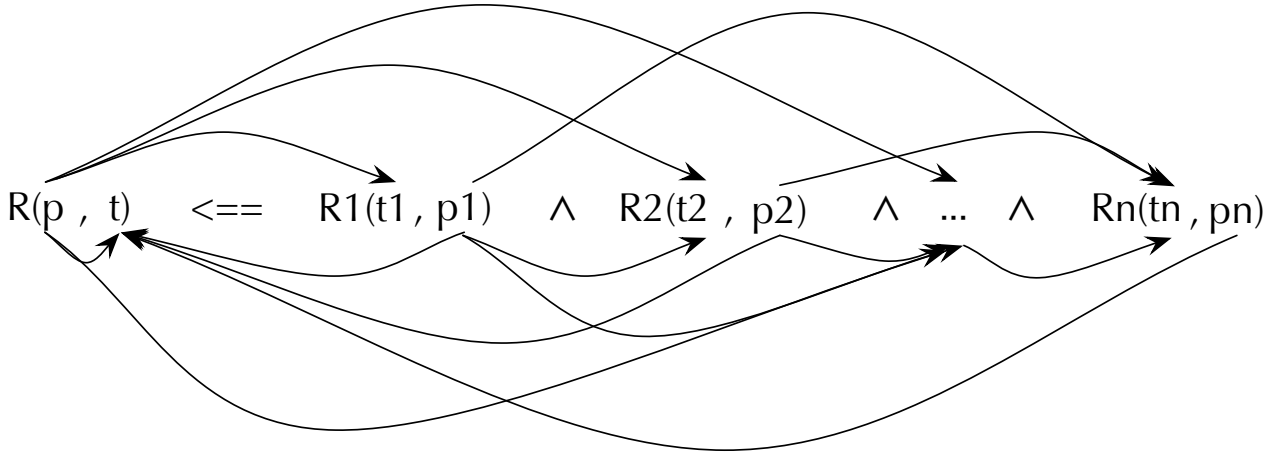
$$b \in f(a) \;\Leftrightarrow\; R(a, b)$$

(siehe z.B. Logik für Informatiker, Kapitel 9). Daher entspricht (2) dem folgenden, aus Hornformeln bestehenden logischen Programmschema:

$$
\begin{array}{lcl}
R(p_1, t_{11}) & \Leftarrow & \bigwedge_{r=1}^{n_{11}} R_{11r}(t_{11r}, p_{11r}), \\
 & \vdots & \\
R(p_1, t_{1m_1}) & \Leftarrow & \bigwedge_{r=1}^{n_{1m_1}} R_{1m_1r}(t_{1m_1r}, p_{1m_1r}), \\
 & \vdots & \\
R(p_k, t_{k1}) & \Leftarrow & \bigwedge_{r=1}^{n_{k1}} R_{k1r}(t_{k1r}, p_{k1r}), \\
 & \vdots & \\
R(p_k, t_{km_k}) & \Leftarrow & \bigwedge_{r=1}^{n_{km_k}} R_{km_kr}(t_{km_kr}, p_{km_kr}).
\end{array}
\tag{3}
$$

Anstatt Gleichungen von (1) auf einen Term $f(p)$ anzuwenden, um ihn zu einem äquivalenten Term t zu führen, der als Wert von $f(p)$ interpretiert wird, besteht eine Ausführung von (3) in der schrittweisen Resolution einer *Zielformel* $R(p, x)$ über Hornformeln von (3) hin zu einer äquivalenten atomaren Formel $R(p, t)$, so dass t als *Lösung* von $R(p, x)$ in der Variablen x angesehen werden kann.

Der Datenfluss während eines Resolutionsschrittes über eine Hornformel von (3) entspricht dem oben veranschaulichten bei der Anwendung einer Gleichung von (1) :



7.9 Substitution und Unifikation (Haskell-Modul: Coalg.hs)

Wir erweitern den Termfunktor von Abschnitt 7.2 zur Monade:

```
instance Monad Term a where
         return = VT
         t >>= sub = case t of FT a ts -> FT a $ map (>>= sub) ts
                               VT b -> sub b
```

Hier substituiert (ersetzt) der bind-Operator alle Variablen eiens Terms t wiederum durch Terme gemäß der **Substititution**(sfunktion) $sub : B \to Term(A)(B)$.

Der Term $t >>= sub$ heißt sub**-Instanz von** t.

Beispiel

```
data Vars = X | Y | Z | X1 | Y1 | Z1 deriving (Eq,Show)

t :: Term String Vars                                          (5*x)+y+11
t = FT"+"[FT"*"[FT"5"[],VT X],VT Y,FT"11"[]]

sub1 :: Vars -> Tree String Vars
sub1 = \case X -> FT"/"[FT"-"[FT"6"[]],FT"9"[],VT Z]                (-6)/9/z
             Y -> FT"-"[FT"7"[],FT"*"[FT"8"[],FT"0"[]]]             7-(8*0)
             x -> VT x

t >>= sub1 ⇝ FT"+"[FT"*"[FT"5"[],FT"/"[FT"-"[FT"6"[]],FT"9"[],VT Z]],
                   FT"-"[FT"7"[],FT"*"[FT"8"[],FT"0"[]]],
                   FT"11"[]                          (5*((-6)/9/z))+(7-(8*0))+11
```

Zwei Terme t und t' heißen **unifizierbar**, falls sie einen **Unifikator** haben, d.i. eine Substitution $sub : B \to Term(A)(B)$ mit

$$t >>= sub \quad = \quad t' >>= sub.$$

Unifikationsalgorithmus

```
notIn :: Eq b => b -> Term a b -> Bool
b `notIn` VT b'   = b /= b'
b `notIn` FT _ ts = all (b `notIn`) ts

unify :: (Eq a,Eq b) => Term a b -> Term a b -> Maybe (b -> Term a b)
unify (VT b) (VT c)         = Just $ if b == c then VT
                                                else update VT b $ VT c
unify(VT b) t               = do guard $ b `notIn` t
                                 Just $ update VT b t
unify t (VT b)              = unify (VT b) t
unify (FT a ts) (FT a' us) = do guard $ a == a'  &&
                                        length ts == length us
                                unifyList ts us
```

```
unifyList :: (Eq a,Eq b) => [Term a b] -> [Term a b]
                                       -> Maybe (b -> Term a b)
unifyList [] []         = Just VT
unifyList (t:ts) (u:us) = do sub <- unify t u
                             let msub = map (>>= sub)
                             sub' <- unifyList (msub ts) $ msub us
                             Just $ sub >=> sub'
```

Sind t und t' unifizierbar, dann liefert $unify(t)(t')$ einen Unifikator, der allen Elementen von A möglichst kleine Bäume zuweist.

Beispiel

```
sub2 = fromJust $ unify (FT"+"[FT"-"[VT X,VT Y],VT Z])
                        (FT"+"[VT X1,FT"*"[VT Y1,VT Z1]])
                                ⇝  sub2 Z  = FT"*"[VT Y1,VT Y2]
                                   sub2 X1 = FT"-"[VT X,VT Y]
                                   sub2 x  = VT x
```

7.10 Generische Tiefen- und Breitensuche in Bäumen

(Haskell-Modul: Examples.hs)

Hier sollen die Funktionen *depthfirst* und *breadthfirst* Knoteneinträge eines Baums suchen, die eine als Boolesche Funktion p dargestellte Bedingung erfüllen.

Dabei bleibt in der Definition der Funktionen nicht nur der Baumtyp offen, sondern auch welche der p erfüllenden Knoten als Ergebnis zurückgegeben werden. Das bestimmt beim Aufruf von *depthfirst* bzw. *breadthfirst* die jeweilige Instanz der Baumtypvariablen t bzw. Monadenvariablen m.

Da *depthfirst* und *breadthfirst* nur die Wurzel und die Liste der größten echten Unterbäume eines Baums benötigt, enthält unsere Baumtypklasse zwei entsprechende Funktionen:

```
class TreeC t where rootC :: t a -> a
                    subtreesC :: t a -> [t a]
```

Die im Kapitel 5 behandelten Baumtypen entsprechen folgenden Instanzen von *TreeC*:

```
instance TreeC Bintree where rootC (Bjoin a _ _) = a
                             subtreesC Empty          = []
                             subtreesC (Bjoin _ t u) = [t,u]
```

```
instance TreeC BintreeL where rootC (Leaf a)    = a
                               rootC (Bin a _ _) = a
                               subtreesC (Leaf _)    = []
                               subtreesC (Bin _ t u) = [t,u]

instance TreeC Tree where rootC = root; subtreesC = subtrees


checkRoot :: (TreeC t,MonadPlus m) => (a -> Bool) -> t a -> m a
checkRoot p = checkResult return p . rootC
```

siehe Abschnitt 7.4

```
depthfirst,breadthfirst :: (TreeC t,MonadPlus m)
                                        => (a -> Bool) -> t a -> m a
depthfirst p t = msum $ checkRoot p t :
                        map (depthfirst p) (subtreesC t)

breadthfirst p t = visit [t] where
                   visit [] = mzero
                   visit ts = msum $ map (checkRoot p) ts ++
                                         [visit $ ts >>= subtreesC]
```

depthfirst(p)(t) und *breadthfirst(p)(t)* liefern in der Maybe-Monade den – bzgl. Tiefen- bzw. Breitensuche – ersten Knoteneintrag des Baums t, der die Bedingung p erfüllt, während in der Listenmonade beide Aufrufe alle Knoteneinträge in Prä- bzw. Heapordnung, die p erfüllen, auflisten.

Der bind-Operator in der Definition von *breadthfirst* bezieht sich immer auf die Listenmonade, entspricht also *flip(concatMap)* (siehe Abschnitt 7.6).

Beispiele

```
t1 :: Tree Int
t1 = F 1 [F 2 [F 2 [V 3 ,V (-1)],V (-2)],F 4 [V (-3),V 5]]
depthfirst (< 0) t1 :: Maybe Int    ⇝  Just (-1)
depthfirst (< 0) t1 :: [Int]        ⇝  [-1,-2,-3]
breadthfirst (< 0) t1 :: Maybe Int  ⇝  Just (-2)
breadthfirst (< 0) t1 :: [Int]      ⇝  [-2,-3,-1]

t2 :: BintreeL Int
t2 = read "5(4(3,8(9,3)),6(1,2))"
depthfirst (> 5) t2 :: Maybe Int    ⇝  Just 8
depthfirst (> 5) t2 :: [Int]        ⇝  [8,9,6]
breadthfirst (> 5) t2 :: Maybe Int  ⇝  Just 6
breadthfirst (> 5) t2 :: [Int]      ⇝  [6,8,9]
```

7.11 Leser- und Schreibermonaden

Jeder Leserfunktor (siehe Abschnitt 7.2) ist eine Monade:

```
instance Monad ((->) state) where return = const
                                  (h >>= f) st = f (h st) st
```

Demnach ist das in Abschnitt 2.5 definierte Lifting

$$lift : (a \to b \to c) \to (state \to a) \to (state \to b) \to state \to c$$

binärer Operationen die Leserfunktor-Instanz des Monadenkombinators *liftM2* (siehe Abschnitt 7.4).

Ein Schreiberfunktor (,)(*state*) (siehe Abschnitt 7.2) lässt sich zur Monade erweitern, wenn *state* eine Instanz der Typklasse *Monoid* ist:

```
class Monoid a where mempty :: a; mappend :: a -> a -> a

mconcat :: Monoid a => [a] -> a
mconcat = foldr mappend mempty

instance Monoid Int where mempty = 0; mappend = (+)
instance Monoid [a] where mempty = []; mappend = (++)
```

Anforderungen an die Instanzen von *Monoid*:

```
(a `mappend` b) `mappend` c  =  a `mappend` (b `mappend` c)
mempty `mappend` a           =  a
a `mappend` mempty           =  a
```

Die Schreiberfunktor-Instanz der Monadenklasse:

```
instance Monoid state => Monad ((,) state) where
         return a = (mempty,a)
         (st,a) >>= f = (st `mappend` st',b) where (st',b) = f a
```

Beispiel (siehe Abschnitt 4.2)

Da *String* mit dem Listentyp [*Char*] übereinstimmt, ist *String* ein Monoid mit *mempty* = *[]* und *mappend* = *(++)* und damit *(String,Int)* die entsprechende Schreibermonade.

```
writeVal :: Show x => Exp x -> Store x -> (String,Int)
writeVal e st =
   case e of Con i   -> ("",i)
             Var x   -> out $ st x
             Sum es  -> do is <- mapM f es; out $ sum is
             Prod es -> do is <- mapM f es; out $ product is
```

```
            e :- e' -> do i <- f e; k <- f e'; out $ i-k
            e :/ e' -> do i <- f e; k <- f e'; out $ i`div`k
            i :* e  -> do k <- f e; out $ i*k
            e :^ i  -> do k <- f e; out $ k^i
    where out i = ('\n':filter (/='\"') (show e)++" = "++show i,i)
          f = flip writeVal st
```

Sei z.B. t die Darstellung des Ausdrucks $5*6*7+x-5*2*3$ als Element von *Expr(String)*. Dann gilt $writeVal(t)(\lambda x.66) = (str, 246)$, wobei str der folgende String ist:

```
6*7 = 42
5*6*7 = 210
x = 66
5*6*7+x = 276
2*3 = 6
5*2*3 = 30
5*6*7+x-5*2*3 = 246
```

Zur Darstellung von *writeVal* als Faltung muss analog zu

```
diffAlg :: Eq x => Arith x (x -> Exp x,Exp x)
```

(siehe Abschnitt 4.2) eine Produktalgebra

```
evalWAlg :: Eq x => Show x => Arith x (Store x -> (String,Int),Exp x)
```

definiert werden, weil auch *writeVal* außerhalb rekursiver Aufrufe auf den zu faltenden Ausdruck *e* zugreift. Die Definition von *evalWAlg* steht im Haskell-Modul Expr.hs.

Dort wird auch ein generischer Compiler (*exp2alg*) definiert, der den Inhalt einer Datei in einen Ausdruck vom Typ *Exp(String)* übersetzt und je nach Zahlparameter (1 bis 23) in einer bestimmten Algebra auswertet. Die Compilerinstanzen, die bei ihrer Ausführung die Funktion

```
compE :: Arith String val -> Compiler val
```

aufrufen, übersetzen direkt in die jeweilige Zielalgebra, ohne vorher einen *Syntaxbaum* vom Typ *Exp(String)* zu erzeugen. Mehr dazu in den Abschnitten 9.4 und 9.5.

Einige Instanzen von *exp2alg* erzeugen Schleifen zur wiederholten Eingabe von Variablenbelegungen (oder deren Auslesen aus einer Datei), mit denen das vom Compiler berechnete Zielprogramm getestet werden kann.

7.12 Zustandsmonaden (siehe Abschnitt 7.2)

```
instance Monad (State state) where
         return a = State $ \st -> (a,st)
         State h >>= f = State $ (\(a,st) -> runS (f a) st) . h
```

Hier komponiert der bind-Operator (>>=) zwei Zustandstransformationen (h und dann f) sequentiell. Dabei liefert die von der ersten erzeugte Ausgabe die Eingabe der zweiten.

State(state) wird auch **Seiteneffektmonade** genannt. Zur Abgrenzung von Leser- oder Schreibermonaden, die ja auch einen Zustandstyp enthalten, sollte man sie eigentlich *Transitionsmonade* nennen, weil sie aus Zustands*transformationen* besteht. "Transitionsmonade" könnte aber mit "Monadentransformer" verwechselt werden (siehe Kapitel 9). Deshalb bleiben wir lieber bei dem in der Haskell-Welt üblichen Begriff *Zustandsmonade*.

Beispiel DefUse (Haskell-Modul: Expr.hs)

Eine Aufgabe aus der Datenflussanalyse: Ein imperatives Programm wird auf die Liste der Definitions- und Verwendungsstellen seiner Variablen reduziert, z.B. auf ein Objekt folgenden Typs, bei dem x die Menge möglicher Variablen repräsentiert:

```
data DefUse x = Def x (Exp x) | Use x
```

Die folgende Funktion *trace* filtert die Verwendungsstellen aus der Liste heraus und ersetzt sie schrittweise durch Paare, die aus der jeweils benutzten Variable und deren an der jeweiligen Verwendungsstelle gültigen Wert bestehen. Die Anfangsbelegung der Variablen wird *trace* als zweiter Parameter übergeben.

```
trace :: Eq x => [DefUse x] -> Store x -> ([(x,Int)],Store x)
trace (Def x e:s) store = trace s $ updStore x e store
trace (Use x:s) store   = ((x,i):s',store')
                          where i = store x
                                (s',store') = trace s store
trace _ store           = ([],store)

updStore :: Eq x => x -> Exp x -> Store x -> Store x
updStore x e = update store x $ eval e store
```

(siehe Abschnitt 4.2)

Monadische Version:

```
traceS :: Eq x => [DefUse x] -> State (Store x) [(x,Int)]
traceS (Def x e:s) = do store <- get
                        put $ updStore x e store
                        traceS s
traceS (Use x:s)   = do store <- get
                        s <- traceS s
                        return $ (x,store x):s
traceS _           = return []
```

Im Gegensatz zu *trace* hat *traceS* keinen Zustandsparameter (*store*). Alle Änderungen von bzw. Zugriffe auf *store* erfolgen über Aufrufe der (Standard-)Hilfsfunktionen *put* und *get*:

```
put :: state -> State state ()
put state = State $ const ((),state)

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

Beispiel

```
data V = X | Y | Z deriving (Eq,Show)

dflist :: [DefUse V]
dflist = [Def X $ Con 1,Use X,Def Y $ Con 2,Use Y,
          Def X $ Sum [Var X,Var Y],Use X,Use Y]

fst $ trace dflist $ const 0          ⇝  [(X,1),(Y,2),(X,3),(Y,2)]
fst $ runS (traceS dflist) $ const 0  ⇝  [(X,1),(Y,2),(X,3),(Y,2)]
```

kann man sich vorstellen als Zustandsmonade, deren Operationen die Zustände von Ein/Ausgabemedien abfragen bzw. ändern. Die Abfragen bzw. Änderungen können jedoch nur indirekt über elementare Funktionen (ähnlich den obigen Funktionen *def* und *use*) erfolgen. Dazu gehören u.a.:

```
readFile :: String -> IO String
```
`readFile "source"` *liest den Inhalt der Datei* `source` *und gibt ihn als String zurück.*

```
writeFile :: String -> String -> IO ()
```
`writeFile "target"` *schreibt einen String in die Datei* `target`.

```
putStr :: String -> IO ()
```
`putStr str` *schreibt* `str` *ins Shell-Fenster.*

```
putStrLn :: String -> IO ()
```
`putStrLn str` *schreibt* `str` *ins Shell-Fenster und springt dann zur nächsten Zeile.*

```
getLine :: IO String
```
`getLine` *liest den eingebenen String und springt dann zur nächsten Zeile.*

readFile ist eine partielle Funktion. Ihre Ausführung bricht ab, wenn es die Datei, deren Namen ihr als Parameter übergeben wird, nicht gibt. Die Funktion

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

dient der Fehlerbehandlung: Tritt bei der Ausführung ihres Arguments vom Typ *IO(a)* ein Fehler $err \in IOError$ auf, dann wird anstelle eines Programmabbruchs das Bild von *err* unter dem zweiten Argument $f : IOError \to IO(a)$ ausgeführt.

Beispiele

```
readFileContinue :: a -> String -> (String -> IO a) -> IO a
readFileContinue err file act =
            do str <- readFile file `catch` handler
               if null str then do putStrLn $ file++" does not exist"
                                   return err
                           else act str
            where handler :: IOError -> IO String
                  handler _ = return ""
```

readFileContinue(err)(file)(act) liest den Inhalt der Datei *file* und übergibt ihn zur Weiterverarbeitung an die Funktion *act*, falls *file* existiert.

Andernfalls wird der Text *file does not exist* ausgegeben und der Aufruf liefert den Wert *return(err)* .

```
loop :: IO ()
loop = do putStrLn "Enter an integer value for x!"
          str <- getLine
          let x = read str
          if x < 5 then putStrLn "x < 5"
                   else do putStrLn $ "x = "++show x
                           loop
```

loop liest wiederholt einen Wert einer Variablen x ein und gibt ihn wieder aus, bis der eingelesene Wert kleiner als 5 ist.

```
scaleAndDraw :: (Float -> Float -> IO ()) -> IO ()
scaleAndDraw draw = do
        putStrLn "Enter a horizontal and a vertical scaling factor!"
        str <- getLine
        let strs = words str
        when (length strs == 2) $ do let [hor,ver] = map read strs
                                     draw hor ver
                                     scaleAndDraw draw
```

scaleAndDraw(draw) liest wiederholt einen horizontalen und einen vertikalen Skalierungsfaktor ein und zeichnet mit der Funktion *draw* in entsprechender Größe ein bestimmtes graphisches Objekt, bis die Eingabe nicht mehr aus zwei Strings besteht.

7.14 Funktionale Abhängigkeiten berechnen (Haskell-Modul: Examples.hs)

Die folgende Tabelle einer relationalen Datenbank repräsentiert eine vierstellige Relation. Ihre Spaltenindizes *Alter, Kind, Vater, Mutter* entsprechen Attributen, deren Werte die darunter als Zeilen dargestellte Objekte bestimmen. Die Relation enthält die neben der Tabelle aufgelisteten funktionalen Abhängigkeiten von Attributen.

Kind	Alter	Vater	Mutter
Harald	5	Franz	Helga
Maria	4	Franz	Helga
Sabine	2	Franz	Ursula
Gertrud	7	Moritz	Melanie
Maria	4	Moritz	Melanie
Sabine	2	Moritz	Melanie
Robert	9	Peter	Christina

```
Alter -> Kind
Kind -> Alter
Mutter -> Vater
Kind,Vater -> Mutter
Alter,Vater -> Mutter
```

So ist z.B. in jeder Zeile die Mutter sowohl durch das Alter des Kindes *und* dessen Vater wie auch durch das Kind *und* seinen Vater eindeutig bestimmt. Betrachtet man Attribute als Typen, dann hat die Pfeilnotation für funktionale Abhängigkeiten hier dieselbe Bedeutung wie in Abschnitt 7.1.

Die folgende Funktion berechnet die funktionalen Abhängigkeiten der Relation in der Datei *file* und speichert das Ergebnis in der Datei *file_fd*.

```
fundeps :: String -> IO ()
fundeps file = do str <- readFile file
                  let first:rest = lines str
                      attrs:rows = words first:map words rest
                  writeFile (file++"_fd") $ fd attrs rows where
   fd attrs rows = foldl out "" $ map mindeps attrs where
      inds = indices attrs
      mindeps i = [(i,is) | is <- minis subset $ rel2funL deps i]
      deps =  [(i,is) | i <- inds,
                        is <- remove [] $ powerset $ remove i inds,
                        check i is]
      check i is = all det $ map fst rel
            where rel = map f rows; f row = (map (row!!) is,row!!i)
                  det vals = length (rel2funL rel vals) == 1
      out str (i,k:is) = str ++ foldl f (attrs!!k) is
                             ++ " -> " ++ attrs!!i ++ "\n"
                       where f str i = str ++ ',':attrs!!i
```

$check(i, is) = True$ gilt genau dann, wenn für je zwei Zeilen row, row' der Relation gilt:

$$map(row!!)(is) = map(row'!!)(is) \;\Rightarrow\; row(i) = row'(i).$$

7.15 Zustandsvariablen (Haskell-Modul: Examples.hs)

Variablen in Haskell sind grundsätzlich *logische* Variablen, also nur "Platzhalter" für Werte, die, einmal zugewiesen, nicht verändert werden können. Taucht derselbe Variablenname in einem Programm mehrfach auf, dann handelt es sich entweder um denselben Wert wie z.B. in der Gleichung $x = 0 : 1 : x$ oder die beiden (gleichlautenden) Variablen sind verschiedenen Gültigkeitsbereichen (Scopes) zugeordnet wie z.B. in einem monadischen Ausdruck der Form

$$do\ x \leftarrow m;\ f(x);\ x \leftarrow g(x);\ h(x).$$

Da dieser für

$$m\ >>= (\lambda x.f(x) >>= (g(x) >> (\lambda x.h(x))))$$

steht (siehe Kapitel 7), bilden $f(x)$ und $g(x)$ den Scope der ersten Variablen mit Namen x, während der Scope der zweiten Variablen mit Namen x aus $h(x)$ besteht. Und die Gleichung $x = 0 : 1 : x$ beschreibt auch keine Änderung des Wertes von x, sondern repräsentiert die unendliche Liste $0 : 1 : 0 : 1 : 0 : 1 : \ldots$ (siehe Abschnitt 3.12).

Wie in imperativen und objektorientierten Sprachen, können *veränderliche* Variablen in Haskell als **Zeiger** (*references*) implementiert werden. Auch wenn das viele Sprachen verschleiern: Ein Zeiger hat stets einen anderen Typ als der Wert, auf den er zeigt. In Haskell hat ein Zeiger auf einen Wert vom Typ *a* den Typ *IORef(a)*. Das Erzeugen, Zugreifen bzw. Setzen von Zeigern bzw. Werten, auf die sie zeigen, erfolgt mit IO-monadischen Basisfunktionen:

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

Weitere Details zur Implementierung von IORefs und anderen Haskell-Zeigertypen findet man z.B. hier.

In Kapitel 4 haben wir behauptet, dass die für objektorientierte Sprachen zentralen Objektklassen Haskell-Datentypen mit Destruktoren entsprechen. Wie die Elemente jedes Haskell-Datentyps sind so implementierte Objekte jedoch **statisch**, d.h. jeder Update eines Objektdestruktors *destr* erzeugt ein neues Objekt: Aus *obj* wird $obj\{destr = value\}$ (siehe Kapitel 4).

Demgegenüber lassen sich Objekte auch unter Beibehaltung ihrer Namen verändern, indem sie zusammen mit Zeigern auf ihre Komponenten in die IO-Monade eingebettet werden. Ein Update verändert dann nur den *Zustand* des Objekts und nicht seine Identität, weshalb es auch als **dynamisches Objekt** bezeichnet wird.

M.a.W.: Ein dynamisches Objekt ist eine Zustandsvariable, die als Zeiger auf Zustände implementiert wird, die wiederum durch Werte von Attributen des Objektes gegeben sind. Demzufolge hat ein dynamisches Objekt stets einen anderen Typ als seine möglichen Zustände.

Beispiel Kantenzüge als verkettete Listen

In Kapitel 3 wurden Kantenzüge als Punktlisten des Datentyps *[Point]* implementiert. Im Folgenden realisieren wir sie als **verkettete Listen** (linked lists) dynamischer Objekte, die wir als Zellen bezeichnen. Jede Zelle besteht aus einem Punkt *point* und einem Zeiger *next* auf eine Zelle oder dem nil-Zeiger, der durch *Nothing* implementiert wird.

Wir beschränken uns auf Listen, deren Punkte paarweise verschieden sind. Daher entsprechen Polygone, also geschlossene Kantenzüge, zyklischen Listen, während die Listendarstellung jedes anderen Kantenzuges eine Zelle enthält, deren Zeiger *pointRef* auf *Nothing* zeigt.

```
data Cell = Cell {point :: Point, next :: Maybe (IORef Cell)}
```

```
lengthCyclic :: IORef Cell -> IO (Float,Bool)
lengthCyclic ref = do
  Cell pt0 next <- readIORef ref
  let loop :: Float -> Point -> Maybe (IORef Cell) -> IO (Float,Bool)
      loop dist _ Nothing     = return (dist,False)
      loop dist pt (Just ref) = do Cell pt' nx' <- readIORef ref
                                   let dist' = dist+distance pt pt'
                                   if pt0 == pt'
                                   then return (dist',True)
                                   else loop dist' pt' nx'
  loop 0 pt0 next
```

lengthCyclic(ref) berechnet die Länge des Kantenzuges, dessen erster Punkt in der Zelle steht, auf die *ref* zeigt, und stellt fest, ob der Kantenzug ein Polygon ist oder nicht.

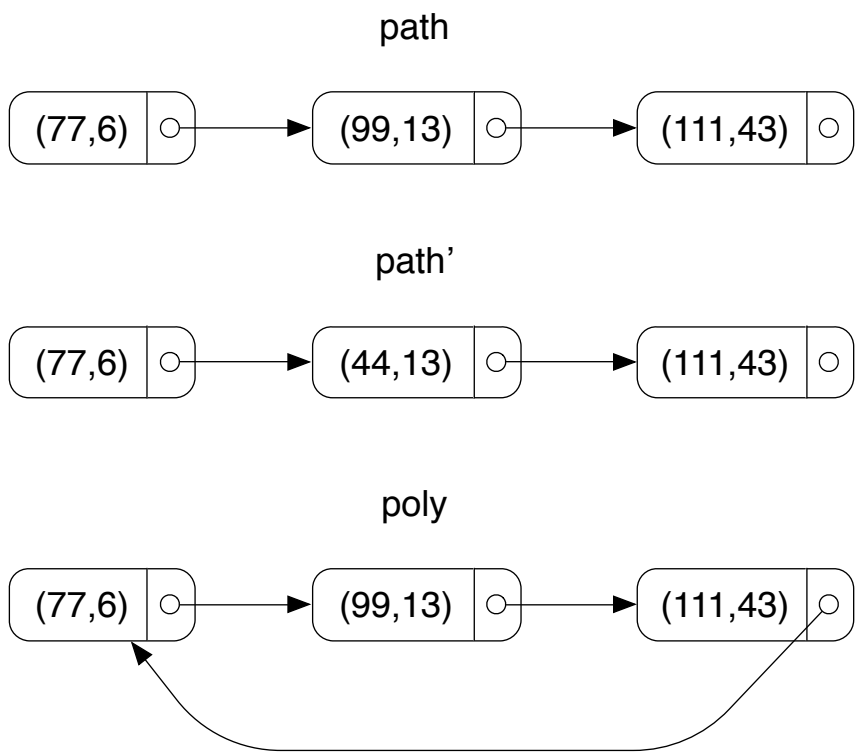
Die ursprüngliche Version dieser Funktion operiert auf Punktlisten:

```
lengthCyclicL :: [Point] -> (Float,Bool)
lengthCyclicL (pt0:s) = loop 0 pt0 s where
        loop :: Float -> Point -> [Point] -> (Float,Bool)
        loop dist _ []       = (dist,False)
        loop dist pt (pt':s) = if pt0 == pt' then (dist',True)
```

```
                else loop dist' pt' s
                where dist' = dist+distance pt pt'
```

Demnach sind es gar nicht einzelne Punkte, sondern Punktlisten, die *lengthCyclic* durch den Typ *Maybe (IORef Cell)* implementiert.

Drei verkettete Listen



```
outPathInfo :: [Point] -> IORef Cell -> IO ()
outPathInfo path ref = do
        putStrLn $ "\npath = " ++ show (take 4 path) ++
                   "\nlengthCyclicL = " ++ show (lengthCyclicL path)
        lgb <- lengthCyclic ref
        putStrLn $ "lengthCyclic = " ++ show lgb

testPath :: IO ()
testPath = do
      let path@[pt1,pt2,pt3] = [Point 77 6,Point 99 13,Point 111 43]
      rec ref1 <- newIORef $ Cell pt1 $ Just ref2
          ref2 <- newIORef $ Cell pt2 $ Just ref3
          ref3 <- newIORef $ Cell pt3 Nothing
      outPathInfo path ref1

      cell2 <- readIORef ref2
      let pt = cell2&point
             pt2 = pt {x = pt&x-55}
             path' = [pt1,pt2,pt3]
      writeIORef ref2 $ cell2 {point = pt2}
      outPathInfo path' ref1
```

rec bewirkt hier die gleichzeitige Erzeugung dreier Zeiger

```
        cell3 <- readIORef ref3
        writeIORef ref3 $ cell3 {next = Just ref1}
        let poly = path'++poly
        outPathInfo poly ref1

testPath  ⇝  path = [(77.0,6.0),(99.0,13.0),(111.0,43.0)]
             lengthCyclicL = (55.39778,False)
             lengthCyclic = (55.39778,False)

             path' = [(77.0,6.0),(44.0,13.0),(111.0,43.0)]
             lengthCyclicL = (107.14406,False)
             lengthCyclic = (107.14406,False)

             poly = [(77.0,6.0),(44.0,13.0),(111.0,43.0),(77.0,6.0)]
             lengthCyclicL = (157.39343,True)
             lengthCyclic = (157.39343,True)
```

Verkettete Listen sind zwar ein gutes Beispiel, um das Prinzip der Erzeugung und Verarbeitung dynamischer Objekte zu demonstrieren. In konkreten Anwendungen spielen sie aber als relativ aufwändige Alternative zu Haskell's Listentyp eine untergeordnete Rolle.

Ein Effizienzgewinn durch den Einsatz dynamischer Objekte lässt sich eher in verteilten Systemen mit komplexer Verknüpfungsstruktur und zahlreichen Attributen erreichen – wie überhaupt erst bei größeren Anwendungen objektorientierte Sprachkonstrukte Sinn machen. Für das Operieren mit Zahlen, Punkten oder Listen von Punkten offeriert Haskell einfachere Mittel, die vermutlich die objektorientierten an Effizienz sogar noch überbieten.

8 Felder

8.1 Die Typklasse für Indexmengen

```
class Ord a => Ix a where
  range :: (a,a) -> [a]
  index :: (a,a) -> a -> Int
  inRange :: (a,a) -> a -> Bool

  rangeSize :: (a,a) -> Int
  rangeSize (a,b) = index (a,b) b+1
```

```
instance Ix Int where
  range (a,b) = [a..b]
  index (a,b) c = c-a
  inRange (a,b) c =
                  a <= c && c <= b
  rangeSize (a,b) = b-a+1
```

Die Standardfunktion *array* bildet eineListe von (Index,Wert)-Paare auf ein Feld ab:

```
array :: Ix a => (a,a) -> [(a,b)] -> Array a b
```

mkArray(a,b) wandelt die Einschränkung einer Funktion $f : A \to B$ auf das Intervall $[a, b] \subseteq A$ in ein Feld um:

```
mkArray :: Ix a => (a,a) -> (a -> b) -> Array a b
mkArray (a,b) f = array (a,b) [(x,f x) | x <- range (a,b)]
```

Zugriffsoperator für Felder:

```
(!) :: Ix a => Array a b -> a -> b
```

Funktionsapplikation wird zum Feldzugriff: Für alle $i \in [a, b]$, $f(i) = mkArray(f)!i$.

Update-Operator für Felder:

```
(//) :: Ix a => Array a b -> [(a,b)] -> Array a b
```

Für alle Felder arr mit Indexmenge A und Wertemenge B,
$s = [(a_1, b_1), \ldots (a_n, b_n)] \in (A \times B)^*$ and $a \in A$ gilt also:

$$(arr//s)!a \;=\; \begin{cases} b_i & \text{falls } a = a_i \text{ für ein } 1 \leq i \leq n, \\ arr!a & \text{sonst.} \end{cases}$$

$a_1, \ldots, a_n$ sind genau die Indizes des Feldes arr, an denen es sich von $arr//s$ unterscheidet.

Feldgrenzen:

```
bounds :: Ix a => Array a b -> (a,a)
```

$bounds(arr)$ liefert die kleinsten und größten Index, an dem das Feld arr definiert ist.

verbindet die rekursive Implementierung einer oder mehrerer Funktionen mit **Memoization**, das ist die Speicherung der Ergebnisse rekursiver Aufrufe in einer Tabelle (die üblicherweise als Feld implementiert wird), so dass diese nur einmal berechnet werden müssen, während weitere Vorkommen desselben rekursiven Aufrufs durch Tabellenzugriffe ersetzt werden können. Exponentieller Zeitaufwand wird auf diese Weise oft auf linearen heruntergedrückt.

Beispiel Fibonacci-Zahlen

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

Wegen der binärbaumartigen Rekursion in der Definition von *fib* benötigt *fib(n)* 2^n Rechenschritte. Ein äquivalentes dynamisches Programm lautet wie folgt:

```
fibA = mkArray (0,1000000) fib where fib 0 = 1
                                     fib 1 = 1
                                     fib n = fibA!(n-1) + fibA!(n-2)
```

fibA!n benötigt nur $O(n)$ Rechenschritte. Der Aufruf führt zur Anlage des Feldes *fibA*, in das die Werte der Funktion *fib* von *fib(0)* bis *fib(n)* eingetragen werden.

Für alle $i > 1$ errechnet sich *fib(i)* aus Funktionswerten an Stellen $j < i$. Diese stehen aber bereits in *fibA*, wenn der i-te Eintrag vorgenommen wird. Folglich sind alle rekursiven Aufrufe in der ursprünglichen Definition von *fib* als Zugriffe auf bereits belegte Positionen von *fibA* implementierbar.

ghci gibt z.B. 19,25 Sekunden als Rechenzeit für *fib(33)* an. Für *fibA!33* liegt sie hingegen unter 1/100 Sekunde.

Generell sollten Funktionen mit einem Definitionsbereich ein Typ der Klasse *Ix* (siehe Abschnitt 8.1) als Felder implementiert werden. Diese benötigen erheblich weniger Speicherplatz als die ursprünglichen Funktionen, weil Funktionen durch – manchmal sehr umfangreiche – λ-Ausdrücke repräsentiert werden. So haben z.B. Matrixoperationen und sie verwendende Algorithmen einen deutlich geringeren Platzverbrauch, wenn man die jeweiligen Funktionen als zweidimensionale Felder implementiert – wie die folgenden drei Abschnitte zeigen.

8.3 Matrizenrechnung (Haskell-Modul: Examples.hs)

Viele Graphalgorithmen lassen sich aus Matrixoperationen zusammensetzen, die generisch definiert sind für Matrixeinträge unterschiedlichen Typs. Die Einträge gehören in der Regel einem Semiring R an (siehe Abschnitt 6.3).

```
type Pos = (Int,Int)
type Matrix = Array Pos

dim :: Matrix r -> Int
dim = fst . snd . bounds

mkMat :: Int -> (Pos -> r) -> Matrix r
mkMat d = mkArray ((1,1),(d,d))

zeroM,oneM :: Semiring r => Int -> Matrix r
zeroM d = mkMat d $ const zero
oneM d  = mkMat d $ \(i,j) -> if i == j then one else zero
```

dim liefert die Dimension, d.h. die Anzahl der Spalten und Zeilen einer quadratischen Matrix. $mkMat(d)$ übersetzt eine Funktion des Typs $Pos \to r$ in ihre Darstellung als quadratische Matrix der Dimension n.

Die Null bzw. Eins des Semirings e der Einträge wird mit $zeroM(d)$ bzw. $oneM(d)$ zur Null- bzw. Einsmatrix der Dimension d geliftet. Entsprechend werden auch die Addition und Multiplikation des Semirings R zur Addition bzw. Multiplikation zweier quadratischer Matrizen über e geliftet, wobei wir voraussetzen, dass beide Matrizen dieselbe Dimension haben:

```
instance Semiring r => Semiring (Matrix r) where
              add m m' = mkMat (dim m) $ lift add (m!) (m'!)
              mul m m' = mkMat d $ \p -> foldl1 add $ map (f p) [1..d]
                         where d = dim m
                               f (i,j) k = mul (m!(i,k)) $ m'!(k,j)
              zero = zeroM 1; one = oneM 1
```

Der transitive Abschluss M^+ einer Matrix M

Die Schrittfunktion

$$\begin{aligned} \Phi : Matrix(R) &\to Matrix(R) \\ M' &\mapsto M + M * M' = ((M+) \circ (M*))(M') \end{aligned}$$

ist stetig, hat also nach dem Fixpunktsatz von Kleene den kleinsten Fixpunkt

$$M^+ = \bigsqcup_{i \in \mathbb{N}} \Phi^i(0)$$

und kann daher mit dem Fixpunktoperator von Abschnitt 6.1 wie folgt berechnet werden:

```
plus :: (Eq r,Semiring r) => (r -> r -> Bool) -> Matrix r -> Matrix r
plus le m = fixpt leM (add m . mul m) $ zeroM $ dim m
            where leM m m' = all (lift le (m!) (m'!)) $ binprod s s
                             where s = [1..dim m]
```

lift wurde in Abschnitt 2.5 definiert, *binprod* in Abschnitt 3.11.

8.4 Graphen als Matrizen (Haskell-Modul: Examples.hs)

```
data GraphM a r = M [a] (Matrix r)

Transformation der Map- in die Matrixdarstellung von Graphen und umgekeh

graph2mat :: Ord a => Graph a -> Matrix Bool
graph2mat g = mkMat (length nodes) f
              where nodes = DMS.keys g
                    f (i,j) = nodes!!(j-1) `elem` sucs (nodes!!(i-1))

graphL2mat :: (Ord a,Semiring r) => GraphL a r -> Matrix r
graphL2mat g = mkMat (length nodes) f where
           nodes = DMS.keys g
           f (i,j) = case lookup (nodes!!(j-1)) $ map (snd *** fst)
                                                $ sucs (nodes!!(i-1))
                     of Just label -> label; _ -> zero
```

```
position :: Eq a => [a] -> a -> Int
position nodes a = case search (== a) nodes of Just i -> i+1; _ -> 0

mkNDMap :: Ord a => [a] -> (Int -> [b]) -> Map a [b]
mkNDMap s f = DMS.fromList [(a,f $ position s a) | a <- s]

mat2graph :: Ord a => GraphM a Bool -> Graph a
mat2graph (M nodes m) = mkNDMap nodes f
          where f i = [nodes!!(j-1) | j <- [1..length nodes], m!(i,j)]

mat2graphL :: (Ord a,Eq r,Semiring r) => GraphM a r -> GraphL a r
mat2graphL (M nodes m) = mkNDMap nodes f
          where f i = [(label,nodes!!(j-1)) | j <- [1..length nodes],
                                              let label = m!(i,j),
                                              label /= zero]
```

Transitiver Abschluss eines unmarkierten bzw. markierten Graphen als Matrixabschluss

```
matClosure :: Ord a => Graph a -> Graph a
matClosure g = mat2graph $ M (DMS.keys g) $ plus (<=) $ graph2mat g
```

```
matClosureL :: (Ord a,Ord r,Semiring r)
                        => (r -> r -> Bool) -> GraphL a r -> GraphL a r
matClosureL le g = mat2graphL $ M (DMS.keys g) $ plus le $ graphL2mat g
```

Da die Multiplikation zweier Matrizen kubische Kosten in der Dimension der Matrizen hat, ist auch im Fall (1) der Aufwand von *plus* biquadratisch, so dass eine Matrix-Implementierung des "nur" kubischen Floyd-Warshall-Algorithmus *warshall* keinen Effizienzgewinn verspricht. Andererseits erzeugt *matClosure* keine so komplexen rekursiven Aufrufe wie die dreifach geschachtelten Faltungen von *warshall* (siehe Abschnitt 6.4).

Berechnung des kürzesten Weges zwischen je zwei Knoten eines kantenmarkierten Graphen und seiner jeweiligen Länge

```
type Path a = ([a],Int)

instance Semiring (Path a) where
                  add (p,m) (q,n) = if m <= n then (p,m) else (q,n)
                  mul (p,m) (q,n) = (p++q, if maxBound `elem` [m,n]
                                           then maxBound else m+n)
                  zero = ([],maxBound); one = ([],0)
```

minpaths(g) markiert jede Kante $a \to b$ von g mit dem kürzesten Weg von a nach b und dessen Länge:

```
applyToVals :: Ord a => (b -> c) -> Map a [b] -> Map a [c]
applyToVals f m = DMS.fromList [(a,map f $ apply m a) | a <- DMS.keys m
```

```
minpaths :: Ord a => GraphL a Int -> GraphL a (Path a)
minpaths g = matClosureL le $ applyToVals f g
             where le (_,m) (_,n) = m >= n
                   f (n,a) = (([a],n),a)
```

Beispiel

```
graph5 :: GraphL Int Int
graph5 = DMS.fromList [(1,[(100,5),(40,2)]),
                       (2,[(50,5),(10,3)]),
                       (3,[(20,4)]),
                       (4,[(10,5)]),
                       (5,[])]
```

```
paths graph5   ⇝
1 -> [(([2],40),2),(([2,3],50),3),(([2,3,4],70),4),(([2,3,4,5],80),5)]
2 -> [(([3],10),3),(([3,4],30),4),(([3,4,5],40),5)]
```

```
3 -> [(([4],20),4),(([4,5],30),5)]
4 -> [(([5],10),5)]
5 -> []
```

Z.B. führt der kürzeste Weg von 1 nach 4 über die Knoten 2 und 3 und hat die Länge 70.

Berechnung aller Wege zwischen je zwei Knoten eines unmarkierten Graphen und ihrer jeweiligen Anzahl

```
type Paths a = ([[a]],Int)

instance Eq a => Semiring (Paths a) where
         add (ps,m) (qs,n) = (rs,length rs) where rs = union ps qs
         mul (ps,m) (qs,n) = (rs,length rs) where
                             rs = filter f [p++q | p <- ps, q <- qs]
                             f p = length p == length (union [] p)
                             --  f(p) prüft p auf Kreisfreiheit
         zero = ([],0); one = ([],0)
```

allpaths(g) markiert jede Kante $a \to b$ von g mit (einer Liste und) der Anzahl aller Wege zwischen a und b, die jeden Knoten von g außer a höchstens einmal enthalten:

```
allpaths :: Ord a => Graph a -> GraphL a Int
allpaths g = h $ matClosureL le $ applyToVals f g
             where le (_,m) (_,n) = m <= n
                   f a = (([[a]],1),a)
                   h g = applyToVals f g where f ((_,n),a) = (n,a)
```

Beispiele (siehe Beispiele in Abschnitt 6.4)

```
allpaths graph1   ⇝   1 -> [(3,1),(5,2),(1,3),(2,4),(1,5),(1,6)]
                      3 -> [(5,1),(8,2),(4,3),(5,4),(3,5),(3,6)]
                      4 -> [(1,1),(2,2),(1,3),(2,4),(1,5),(1,6)]
                      5 -> [(6,1),(8,2),(2,3),(4,4),(2,5),(2,6)]
                      6 -> [(4,1),(7,2),(2,3),(3,4),(2,5),(2,6)]
```

Z.B. gibt es 3 Wege von 6 nach 4 mit der o.g. Eigenschaft.

```
allpaths graph2   ⇝   1 -> [(1,2),(1,3),(1,4),(1,5),(1,6)]
                      2 -> [(1,3),(1,4),(1,5),(1,6)]
                      3 -> [(1,4),(1,5),(1,6)]
                      4 -> [(1,5),(1,6)]
                      5 -> [(1,6)]
```

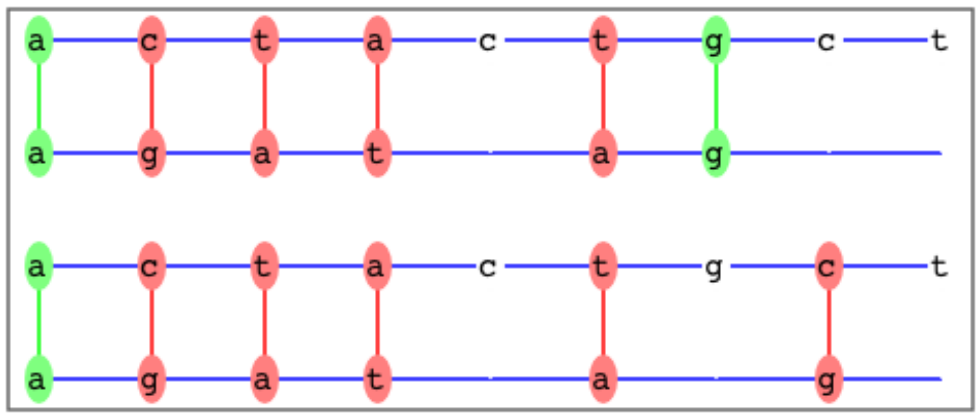
8.5 Alignments (Haskell-Modul: Align.hs)

Die algebraische Behandlung bioinformatischer Probleme geht zurück auf: R. Giegerich, A Systematic Approach to Dynamic Programming in Bioinformatics, Bioinformatics 16 (2000) 665-677.

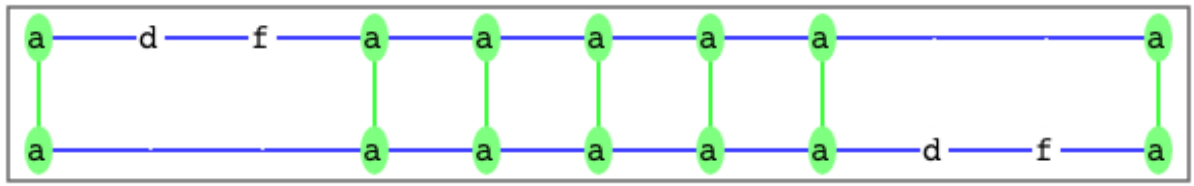
Zwei Listen xs und ys des Typs $String^*$ sollen in die Menge $alis(xs, ys)$ ihrer Alignments von übersetzt werden. Wir setzen eine Boolesche Funktion

$$compl : String^2 \to Bool$$

voraus, die für je zwei Strings x und y angibt, ob x und y komplementär zueinander sind und deshalb aneinander "andocken" können (was auch im Fall $x = y$ möglich ist).



Zwei Alignments von `a c t a c t g c t` *und* `a g a t a g`



Ein Alignment von `a d f a a a a a a` *und* `a a a a a a d f a`

Darstellung von Alignments als Tripellisten

Sei $A = String \uplus \{Nothing\}$ und $h : A^* \to String^*$ die Funktion, die aus einem Wort über A alle Vorkommen von *Nothing* streicht.

Dann ist die Menge der **Alignments von** $xs, ys \in String^*$ wie folgt definiert:

$$
\begin{aligned}
alis(xs, ys) \;=_{def}\; & \bigcup_{n=max(|xs|,|ys|)}^{|xs|+|ys|} \\
& \{[(a_1, b_1, c_1), \ldots, (a_n, b_n, c_n)] \in (A \times A \times RGB)^n \mid \\
& \quad h(a_1 \ldots a_n) = xs \wedge h(b_1 \ldots b_n) = ys \wedge \\
& \quad \forall\, 1 \leq i \leq n : \; (c_i = red \wedge compl(a_i, b_i)) \vee \\
& \qquad\qquad (c_i = green \wedge a_i = b_i \neq Nothing) \vee \\
& \qquad\qquad (c_i = white \wedge ((a_i = Nothing \wedge b_i \neq Nothing) \\
& \qquad\qquad\qquad\qquad \vee (b_i = Nothing \wedge a_i \neq Nothing))\}
\end{aligned}
$$

Alignments lassen sich demnach als Listen von Tripeln des folgenden Datentyps implementieren:

```
type Align  = [Triple]
type Triple = (Maybe String,Maybe String,RGB)

third (_,_,c) = c
```

Hilfsfunktionen für die Alignment-Berechnung

```
matchcount :: Align -> Int
matchcount = length . filter (/= white) . map third
```

$matchcount(s)$ zählt die Vorkommen von red und $green$ im Alignment s.

```
maxmatch :: Align -> Int
maxmatch s = max i m where
             (_,i,m) = foldl trans (False,0,0) $ map third s
             trans (b,i,m) c = if c == white then (False,0,max i m)
                               else (True,if b then i+1 else 1,m)
```

$maxmatch(s)$ berechnet die Länge der maximalen zusammenhängenden Matches von s mit ausschließlich grünen oder roten Farbkomponenten. Dazu realisiert $maxmatch$ die Transitionsfunktion eines Moore-Automaten mit der Eingabemenge $Triple$ und der Zustandsmenge $State = Bool \times \mathbb{Z} \times \mathbb{Z}$.

Die Boolesche Komponente eines Zustands gibt an, ob der Automat gerade einen zusammenhängenden Match von s liest. Die erste ganzzahlige Komponente ist die Anzahl der bisher gelesenen Spalten dieses Matches. Die zweite ganzzahlige Komponente ist die Länge des Maximums der bisher gelesenen zusammenhängenden Matches von s. Daraus ergibt sich der Anfangszustand $(False, 0, 0)$ und das Ergebnis $max(i, m)$ im Endzustand (b, i, m).

```
maxima :: Ord b => (a -> b) -> [a] -> [a]
maxima f s = filter ((== m) . f) s where m = maximum $ map f s
```

$maxima(f)(s)$ ist die Teilliste aller $a \in s$ mit maximalem Wert $f(a)$.

Aufgabe Implementieren Sie die Funktion

```
maximum . map maxmatch :: Align -> Align
```

durch zwei ineinandergeschachtelte Faltungen ohne Verwendung von *maximum*. ❑

```
consx,consy :: String -> Align -> Align
consx x ali = (Just x,Nothing,white):ali
consy y ali = (Nothing,Just y,white):ali

equal,match :: String -> String -> Align -> Align
equal x y ali = (Just x,Just y,green):ali
match x y ali = (Just x,Just y,red):ali

compl :: String -> String -> Bool
compl x y = f x y || f y x where
            f x y = x == "a" && y == "t" || x == "c" && y == "g"
```

a, *t*, *c* und *g* stehen für die Nukleinbasen Adenin, Thymin, Cytosin bzw. Guanin. Deren paarweise Bindungen – *a* an *t* oder *c* an *g* – bilden die in einem Erbmolekül gespeicherte Information ("genetischer Code").

Zur Berechnung von Alignments zweier Stringlisten xs und ys müssen die Komponenten aller Paare von $xs \times ys$ auf Gleichheit und Komplementarität geprüft werden. Die Alignments sollen unabhängig von der konkreten Repräsentation von $xs \times ys$ berechnet werden. Da die erforderlichen Hilfsfunktionen zwei nicht funktional voneinander abhängige Typen involvieren (siehe 7.1), fassen wir sie nicht in einer Typklasse, sondern in folgender Signatur zusammen (siehe Abschnitt 4.2):

```
data AliSig g as = A {first :: (g,g), next :: g -> g, maxi,maxj :: g,
                      getx,gety :: g -> String,
                      entry :: as -> (g,g) -> [Align],
                      mkAlis :: ((g,g) -> [Align]) -> as}
```

Hier sind drei – mit zwei Stringlisten ("Genen") parametrisierte – Algebren vom Typ *AliSig*, die Alignments in unterschiedlicher Weise abspeichern:

```
type Genes = ([String],[String])

lists :: Genes -> AliSig [String] (Genes -> [Align])
lists gs = A {first = gs, next = tail, maxi = [], maxj = [],
              getx = head, gety = head, entry = ($), mkAlis = id}
```

```
fun :: Genes -> AliSig Int (Pos -> [Align])
fun (xs,ys) = A {first = (1,1), next = (+1),
                 maxi = length xs+1, maxj = length ys+1,
                 getx = \i -> xs!!(i-1), gety = ı -> ys!!(i-1),
                 entry = ($), mkAlis = id}

arr :: Genes -> AliSig Int (Matrix [Align])
arr (xs,ys) = A {first = (1,1), next = (+1), maxi = maxi, maxj = maxj,
                 getx = \i -> xs!!(i-1), gety = ı -> ys!!(i-1),
                 entry = (!), mkAlis = mkArray ((1,1),(maxi,maxj))}
               where maxi = length xs+1; maxj = length ys+1
```

maxAlis(*alg*) berechnet zunächst alle Alignments mit maximalem *matchcount*-Wert und wählt dann daraus diejenigen mit zusammenhängenden Matches maximaler Länge aus:

```
maxAlis :: Eq s => AliSig s as -> [Align]
maxAlis alg = maxima maxmatch $ entry alg align $ first alg where
              align = mkAlis alg $ maxima matchcount . f
              f (i,j) | b          = if c then [[]] else alis4
                      | c          = if b then [[]] else alis3
                      | x == y     = alis1++alis3++alis4
```

```
                    | compl x y = alis2++alis3++alis4
                    | True      = alis3++alis4
                      where b  = i == maxi alg; c  = j == maxj alg
                            x  = getx alg i;    y  = gety alg j
                            ni = next alg i;    nj = next alg j;
                            alis1 = h (equal x y) (ni,nj)
                            alis2 = h (match x y) (ni,nj)
                            alis3 = h (consx x) (ni,j)
                            alis4 = h (consy y) (i,nj)
h op = map op . entry alg align
```

maxAlis(lists(xs,ys)) arbeitet direkt auf den Stringlisten xs und ys. *maxAlis(fun(xs,ys))* operiert auf Paaren von Positionen der Elemente von xs bzw. ys und macht *align* damit zu einer rekursiv definierten Funktion auf der Indexmenge $\mathbb{N}^2$. *maxAlis(arr(xs,ys))* arbeitet – analog zu *fibA* (s.o.) – auf dem entsprechenden Feld. Wegen der baumartigen Rekursion von *maxAlis* benötigen die ersten beiden Aufrufe $O(3^{length(xs++ys)})$ Rechenschritte, während das Füllen des Feldes beim dritten Aufruf nur $O(length(xs ++ ys)$ benötigt.

Die Funktionen, die die Ergebnisse von *maxAlis* wie in den obigen Beispielen graphisch darstellen, stehen hier.

Die Variablen einer Folge monadischer "Befehle"

$$do\ x \leftarrow m_1;\ m_2;\ y \leftarrow m_3;\ z \leftarrow m_4;\ m_5 \tag{1}$$

können zwar Werte verschiedener Typen annehmen, die Ausdrücke $m_1, \ldots, m_5$ müssen aber stets vom selben Monadentyp sein. Das schränkt die Anwendungsmöglichkeiten ein und verbietet insbesondere die Einbettung von IO-Befehlen in eine Folge von Aufrufen partieller oder mehrwertiger Funktionen (siehe Abschnitt 7.6).

Abhilfe schaffen Monadentransformer, die eine feste Monade M in eine beliebige Monade m einbetten. M induziert den Monadentransformer T, der, auf m angewendet, eine neue Monade $T(m)$ liefert, die m mit M verschmelzt. Für die Maybe-, Listen- und Zustandsmonade wird T jeweils wie folgt implementiert:

```
newtype MaybeT m a = MaybeT {runMT :: m (Maybe a)}

instance Monad m => Functor (MaybeT m) where fmap = liftM

instance Monad m => Monad (MaybeT m) where
         return = MaybeT . return . Just
         m >>= f = MaybeT $ do ma <- runMT m
```
(2)

```
                                case ma of Just a -> runMT $ f a
                                           _ -> return Nothing

instance Monad m => MonadPlus (MaybeT m) where
         mzero = MaybeT $ return Nothing
         m `mplus` m' = MaybeT $ do ma <- runMT m
                                    if isJust ma then return ma
                                                 else runMT m'
newtype ListT m a = ListT {runLT :: m [a]}

instance Monad m => Functor (ListT m) where fmap = liftM

instance Monad m => Monad (ListT m) where
         return = ListT . return . single
         m >>= f = ListT $ do mas <- runLT m
                              mass <- mapM (runLT . f) mas
                              return $ concat mass

instance Monad m => MonadPlus (ListT m) where
         mzero = ListT $ return []
         m `mplus` m' = ListT $ do mas <- runLT m; mas' <- runLT m'
```

```
                                    return $ mas++mas'

newtype StateT state m a = StateT {runST :: state -> m (a,state)}
```

Im Gegensatz zu *runMT* und *runLT* bettet *runST* nicht die ganze (Zustands-)Monade, sondern nur ihren Wertebereich in m ein.

```
instance Monad m => Functor (StateT state m) where fmap = liftM

instance Monad m => Monad (StateT state m) where
         return a = StateT $ \st -> return (a,st)
         StateT h >>= f = StateT $ (\(a,st) -> runST (f a) st) <=< h

instance MonadPlus m => MonadPlus (StateT state m) where
         mzero = StateT $ const mzero
         StateT g `mplus` StateT h = StateT $ liftM2 mplus g h
```

Die folgende Typklasse *MonadTrans* enthält zwei Funktionen, die m- bzw. M-Objekte zu $T(M)$-Objekten liften:

```
class MonadTrans t where type M t :: * -> *
                         lift  :: Monad m => m a -> t m a
                         liftT :: Monad m => M t a -> t m a

instance MonadTrans MaybeT where
         type M MaybeT = Maybe
         lift  = MaybeT . liftM Just -- :: m a -> MaybeT m a
         liftT = MaybeT . return     -- :: Maybe a -> MaybeT m a

instance MonadTrans ListT where
         type M ListT = [ ]
         lift  = ListT . liftM single -- :: m a -> ListT m a
         liftT = ListT . return       -- :: [a] -> ListT m a

instance MonadTrans (StateT state) where
         type M (StateT state) = State state
         lift m = StateT $ \st -> do a <- m; return (a,st)
                                     -- :: m a -> StateT state m a
         liftT (T f) = StateT $ return . f
                              -- :: State state a -> StateT state m a
```

(3)

Enthält z.B. (1) sowohl m- als auch M-Objekte, sagen wir: m_1, m_2 und m_5 sind m- und m_3, m_4 M-Ausdrücke, dann ist der folgende *T(m)*-Ausdruck nicht nur semantisch äquivalent zu (1), sondern – im Gegensatz zu (1) – auch syntaktisch korrekt:

$$do\ x \leftarrow lift(m_1);\ lift(m_2);\ y \leftarrow liftT(m_3);\ z \leftarrow liftT(m_4);\ lift(m_5) \tag{4}$$

Ist z.B. $T = MaybeT$ und m_4 von der Form *Just(a)*, dann vereinfacht sich (4) wegen

$$liftT \circ Just \overset{(3)}{=} MaybeT \circ return \circ Just \overset{(2)}{=} return$$

zu:

$$do\ x \leftarrow lift(m_1);\ lift(m_2);\ y \leftarrow liftT(m_3);\ z \leftarrow return(a);\ lift(m_5) \tag{5}$$

9.1 Verschmelzung von IO- und Maybe-Monade (Haskell-Modul: Expr.hs)

Wir stellen eine Variablenbelegung vom Typ *Store(x)* als binäre Relation dar (siehe Abschnitt 3.7), legen diese in der Datei *store* ab und führen die partielle Auswertung arithmetischer Ausdrücke (siehe `evalMAlg` in Abschnitt 7.6) in der Monade $MaybeT(IO)$ durch, um im Fall des Auftretens eines der drei folgenden Fehler eine entsprechende Meldung abzusetzen:

- *store* existiert nicht,
- eine benutzte Variable ist in *store* nicht definiert,
- Versuch einer Division durch 0.

```
evalIOAlg :: Arith String (String -> MaybeT IO Int)
evalIOAlg = Arith {con  =  const . return,
                   var_ = \x file -> do
                             store <- lift $ readFileContinue [] file
                                           $ return . read
                             let val = lookup x store
                             put $ x ++ case val of
                                          Just i -> " = " ++ show i
                                          _ -> " is undefined"
                             liftT val,
```

```
                sum_ = \bs file   -> do is <- mapM ($file) bs
                                       return $ sum is,
                prod = \bs file   -> do is <- mapM ($file) bs
                                       return $ product is,
                sub  = \b b' file -> do i <- b file; k <- b' file
                                       return $ i-k,
                div_ = \b b' file -> do i <- b file; k <- b' file
                                       if k /= 0
                                       then return $ i`div`k
                                       else do put "division by 0"
                                               mzero,
                scal = \i b file  -> do k <- b file; return $ i*k,
                expo = \b i file  -> do k <- b file; return $ k^i}
          where put = lift . putStrLn

evalIO = foldArith evalIOAlg
```

Beispiele

```
e1 = Sum [Var"x":^4,5:*(Var"x":^3),11:*(Var"x":^2),Con 222]
e2 = Con 5:/(Con 4:-Var"x")
```

contents("store1") = [("x",4),("y",55)]

$\Rightarrow$ `runMT $ evalIO e1 "store1"` $\leadsto$

```
x = 4
x = 4
x = 4
Just 974
```

`runMT $ evalIO e2 "store1"` $\leadsto$

```
x = 4
division by 0
Nothing
```

contents("store2") = [("z",4),("y",55)]

$\Rightarrow$ `runMT $ evalIO e1 "store2"` $\leadsto$

```
x is undefined
Nothing
```

`runMT $ evalIO e2 "store2"` $\leadsto$

```
x is undefined
Nothing
```

"store3" does not exist

$\Rightarrow$ `runMT $ evalIO e1 "store3"` $\leadsto$

```
store3 does not exist
x is undefined
Nothing
```

9.2 Verschmelzung von IO- und Listenmonade (Haskell-Modul: Examples.hs)

Wir erweitern die iterative Funktion *queens* zur Berechnung von Lösungen des Damenproblems um die Ausgabe der dazu erforderlichen Zustandsübergänge (siehe Abschnitt 7.7):

```
queensTS :: Int -> IO ()
queensTS n = do writeFile "queensT" $ "Transitions of " ++ show n ++
                                      " queens:\n"
                solutions <- runLT $ qloop $ qfirst n
                putStrLn $ "Solutions:\n" ++ show solutions

type Qstate = ([Int],[Int])

qfirst :: Int -> Qstate
qfirst n = ([1..n],[])

qloop :: Monad m => Qstate -> ListT m [Int]
qloop ([],val) = return val
qloop st       = do lift $ appendFile "queensT" $ showTrans st sts
                    msum $ map qloop sts
                 where sts = qsuccessors st
```

```
showTrans :: Show a => a -> [a] -> String
showTrans a []  = ""
showTrans a [b] = show a ++ " -> " ++ show b ++ "\n"
showTrans a s   = show a ++ " -> " ++ show s ++ "\n"
```

Beispiel

```
queensTS 4  ⇝  Solutions:
               [[3,1,4,2],[2,4,1,3]]
```

Inhalt von queensT:

```
Transitions of 4 queens:
([1,2,3,4],[]) -> [([2,3,4],[1]),([1,3,4],[2]),
                   ([1,2,4],[3]),([1,2,3],[4])]
([2,3,4],[1]) -> [([2,4],[3,1]),([2,3],[4,1])]
([2,3],[4,1]) -> ([3],[2,4,1])
([1,3,4],[2]) -> ([1,3],[4,2])
([1,3],[4,2]) -> ([3],[1,4,2])
([3],[1,4,2]) -> ([],[3,1,4,2])
([1,2,4],[3]) -> ([2,4],[1,3])
([2,4],[1,3]) -> ([2],[4,1,3])
([2],[4,1,3]) -> ([],[2,4,1,3])
```

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

Transitionssysteme können von Expander2 eingelesen und weiterverarbeitet, z.B. modallogisch verifiziert oder graphisch darsgestellt werden.

9.3 Verschmelzung von IO-, Maybe- oder Listenmonade mit der Zustandsmonade (Haskell-Modul: Expr.hs)

Wir formulieren eine weitere monadische Version der *trace*-Funktion aus Abschnitt 7.10. Anstatt am Ende eine Liste von Paaren auszugeben, schreibt *traceIO* jedes Paar sofort auf die Konsole in eine eigene Zeile, sobald es berechnet ist:

```
traceIO :: (Eq x,Show x) => [DefUse x] -> StateT (Store x) IO ()

traceIO (Def x e:s) = do store <- liftT get
                         liftT $ put $ updStore x e store
                         traceIO s
traceIO (Use x:s)   = do store <- liftT get
                         lift $ putStrLn $ show (x,store x)
                         traceIO s
traceIO _           = return ()
```

Beispiel

```
data V = X | Y | Z deriving (Eq,Show)

dflist :: [DefUse V]
dflist = [Def X $ Con 1,Use X,Def Y $ Con 2,Use Y,
          Def X $ Sum [Var X,Var Y],Use X,Use Y]

runST (traceIO dflist)   ⇝   (X,1)
                             (Y,2)
                             (X,3)
                             (Y,2)
```

Die Verschmelzung der in Abschnitt 7.6 behandelten Maybe- bzw. Listenmonade mit der Zustandsmonade liefert Automaten:

`StateT state Maybe` implementiert die Menge **partieller Automaten** mit Zustandsmenge *state*. $fst \circ runST$ und $snd \circ runST$ bilden die jeweilige partielle Ausgabe- bzw. Übergangsfunktion.

`StateT state [ ]` implementiert die Menge **nichtdeterministischer Automaten** mit Zustandsmenge *state*. $fst \circ runST$ und $snd \circ runST$ bilden die jeweilige mehrwertige Ausgabe- bzw. Übergangsfunktion.

Zwei mit der parallelen Komposition *mplus* (siehe Abschnitt 7.3) verknüpfte Automaten m und m' realisieren Backtracking: Erreicht m, ausgehend von einem Anfangszustand st, einen Zustand st', von dem aus kein Übergang möglich ist, dann wird der st wiederhergestellt und m' gestartet.

9.4 Generische Compiler (Haskell-Modul: Expr.hs)

Compiler lesen Wörter Zeichen für Zeichen von links nach rechts und transformieren gelesene Teilwörter in Objekte verschiedener Ausgabetypen sowie die jeweils noch nicht verarbeiteten Restwörter. Deshalb lassen sie sich als partielle Automaten mit *String* als Zustandsmenge implementieren:

```
type Compiler = StateT String Maybe
```

Mit Hilfe von Monaden-Kombinatoren können komplexe Compiler aus einfachen zusammengesetzt werden wie z.B. den folgenden, die typische Scannerfunktionen realisieren.

Monadische Scanner

Scanner sind Compiler, die einzelne Symbole erkennen. Der folgende Scanner *sat(f)* erwartet, dass das Zeichen am Anfang des Eingabestrings die Bedingung f erfüllt:

```
sat :: (Char -> Bool) -> Compiler Char
sat f = StateT $ \str -> do c:str <- return str
                            if f c then return (c,str) else mzero

char :: Char -> Compiler Char
char chr = sat (== chr)

nchar :: String -> Compiler Char
nchar chrs = sat (`notElem` chrs)
```

Darauf aufbauend, erwarten die folgenden Scanner eine Ziffer, einen Buchstaben bzw. einen Begrenzer am Anfang des Eingabestrings:

```
digit,letter,delim :: Compiler Char
digit  = msum $ map char ['0'..'9']
letter = msum $ map char $ ['a'..'z']++['A'..'Z']
delim  = msum $ map char " \n\t"
```

Der folgende Scanner *string(str)* erwartet den String *str* am Anfang des Eingabestrings:

```
string :: String -> Compiler String
string  = mapM char
```

Die folgenden Scanner erkennen Elemente von Standardtypen und übersetzen sie in entsprechende Haskell-Typen:

```
bool :: Compiler Bool
bool = msum [string "True"  >> return True,
             string "False" >> return False]

nat,int :: Compiler Int
nat = fmap read $ some digit
int = msum [nat, do char '-'; n <- nat; return $ -n]

identifier :: Compiler String
identifier = liftM2 (:) letter $ many $ nchar "();=!>+-*/^ \t\n"
```

Die Kommas trennen die Elemente der Argumentliste von *msum*.

token(*comp*) erlaubt vor und hinter dem von *comp* erkannten String Leerzeichen, Zeilenumbrüche oder Tabulatoren:

```
token :: Compiler a -> Compiler a
token comp = do many delim; a <- comp; many delim; return a

tchar       = token . char
tstring     = token . string
tbool       = token bool
tint        = token int
tidentifier = token identifier
```

Binäre Bäume übersetzen

Der folgende Compiler ist äquivalent zur Funktion `read :: BintreeL a` in Abschnitt 5.8:

```
bintree :: Compiler a -> Compiler (BintreeL a)
bintree comp = do a <- comp
                  msum [do tchar '('; left <- bintree comp
                           tchar ','; right <- bintree comp
                           tchar ')'; return $ Bin a left right,
                        return $ Leaf a]
```

9.5 Arithmetische Ausdrücke kompilieren II (Haskell-Modul: Expr.hs)

In Abschnitt 5.10 haben wir an den Beispielen *Bintree(A)* und *Tree(A)* gezeigt, dass jede auf einem Datentyp DT induktiv definierte Funktion f als Termfaltung darstellbar ist. Man muss dazu die Konstruktoren von DT so durch Funktionen auf dem Wertebereich von f interpretieren, dass die Faltung der Ausführung von f entspricht. Der Faltungsalgorithmus selbst ist generisch, weil er für jede Interpretation der Konstruktoren von DT in gleicher Weise abläuft.

Soll auch eine nicht induktiv definierte Funktion $f : A \to B$ als Faltung implementiert werden, dann müssen die Elemente von A zunächst in Terme, also Elemente eines konstruktiven Datentyps übersetzt und diese in einem zweiten Schritt gefaltet werden. Der erste Schritt ist eine typische Kompilation, die erzeugten Terme werden üblicherweise **Syntaxbäume** genannt. Beide Schritte (Übersetzung und Faltung) können so integriert werden, dass Syntaxbäume nicht explizit berechnet werden müssen.

Als Beispiel dafür definieren wir im Folgenden einen **generischen Compiler**, der Stringdarstellungen arithmetischer Ausdrücke – ohne daraus zunächst Objekte vom Typ *Exp(x)* (siehe Abschnitt 4.1) zu erzeugen – direkt in eine gewünschte, als *Exp(x)*-Algebra dargestellte Interpretation übersetzt.

Wendet man den folgenden generischen Compiler für Stringdarstellungen arithmetischer Ausdrücke auf *evalAlg* an, dann wertet er den eingegebenen Ausdruck aus. Wendet man ihn auf *codeAlg* an, dann erzeugt er Assemblercode für den eingegebenen Ausdruck (siehe Abschnitte 4.2 und 5.11).

```
compE :: Arith String val -> Compiler val
compE alg = summand >>= moreSummands where

    summand = mplus scalar factor >>= moreFactors

    factor  = msum [fmap (con alg) tint >>= power,
                    fmap (var_ alg) tidentifier >>= power,
                    do tchar '('; a <- compE alg; tchar ')'
                       power a]

    moreSummands a = msum [tchar '-' >> fmap (sub alg a) summand
                                        >> moreSummands,
                           some (tchar '+' >> summand)
                                >>= moreSummands . sum_ alg . (a:),
                           return a]
```

```
moreFactors a  = msum [tchar '/' >> fmap (div_ alg a) factor
                                  >> moreFactors,
                       some (tchar '*' >> mplus scalar factor)
                            >>= moreFactors . prod alg . (a:),
                       return a]

scalar = do i <- tint
            let lift = fmap $ scal alg i
            msum [do tchar '*'
                     msum [lift scalar,
                           do x <- tidentifier
                              lift $ power $ var_ alg x,
                           do tchar '('; a <- compE alg
                              tchar ')'; return $ scal alg i a],
                  power $ con alg i]

power a  = msum [tchar '^' >> fmap (expo alg a) tint,
                 return a]
```

Die Unterscheidung zwischen Compilern für Ausdrücke, Summanden bzw. Faktoren dient der Berücksichtigung von Operatorprioritäten ($+$ und $-$ vor $*$ und $\hat{}$) wie auch der Vermeidung *linksrekursiver* Aufrufe des Compilers: Zwischen je zwei aufeinanderfolgenden Aufrufen muss mindestens ein Zeichen gelesen werden, damit der zweite Aufruf ein kürzeres Argument hat als der erste und so die Termination des Compilers gewährleistet ist.

Korrektheit von *compE*

Für alle in Kapitel 4 und 5 definierten *Arith*-Algebren A ist die Compiler-Instanz $compE(A)$ korrekt bzgl. der Faltung arithmetischer Ausdrücke in A, d.h. für alle $e \in Exp(String)$ gilt:

$$runST(compE(A))(show(e)) = Just(foldArith(A)(e), []).$$

In den Kapiteln 4 bis 6, 9 bis 11 und 14 von [21] werden die hier behandelte Sprache arithmetischer Audrücke um Zuweisungen, Kontrollstrukturen und Prozeduren zu einer kompletten imperativen Sprache, die in Abschnitt 5.12 definierte Assemblersprache um entsprechende Kommandos und der obige generische Compiler um die Übersetzung der zusätzlichen Sprachkonstrukte erweitert.

9.6 Comonaden (Haskell-Modul: Coalg.hs)

```
class Functor cm => Comonad cm where
      extract :: cm a -> a                          -- counit
      (<<=) :: (cm a -> b) -> (cm a -> cm b)        -- comonadische Extension, cobind
```

Anforderungen an die Instanzen von *Comonad*:

Für alle $m \in cm(a)$, $f : cm(a) \to b$ und $g : cm(b) \to c$,

```
f <<= (g <<= cm)  =  (g . f <<=) <<= cm
extract <<= cm    =  cm
extract . f <<=   =  f
```

Während die monadische Extension (siehe Abschnitt 7.4)

$$(=<<) : (a \to m(b)) \to (m(a) \to m(b))$$

die – durch m gegebene – **effekt-erzeugende Kapselung der Ausgabe** einer Funktion $f : a \to m(b)$ auf deren Eingabe überträgt und damit solche Funktionen – mittels der Kleisli-Komposition <=< (s.o.) – komponierbar macht, transportiert dual dazu die comonadische Extension

$$(<<=) : (cm(a) \to b) \to (cm(a) \to cm(b))$$

die – durch cm gegebene – **kontextabhängige Kapselung der Eingabe** einer Funktion $f : cm(a) \to b$ zu deren Ausgabe und macht damit auch solche Funktionen komponierbar:

```
(=<=) :: Comonad cm => (cm b -> c) -> (cm a -> b) -> (cm a -> c)
g =<= f  =  g . (f <<=)                                  co-Kleisli-Komposition

duplicate :: Comonad cm => cm a -> cm (cm a)
duplicate  =  (id <<=)                                   Comultiplikation
```

In der Kategorientheorie wird eine Comonade als Funktor mit *duplicate* und *extract* definiert und $<<=$ wie folgt aus *duplicate* abgeleitet:

```
(f <<=) = fmap f . duplicate
```

Umgekehrt liefert jede Comonade CM eine *Functor*-Instanz:

```
instance Functor CM where fmap f = (f . extract <<=)
```

Comonadische Funktoren

Die jeweiligen Instanzen der Klasse *Functor* stehen in Abschnitt 7.2.

Identitätscomonade

```
instance Comonad Id where
         extract = run
         f <<= cm = Id $ f cm
```

Lesercomonaden

```
instance Monoid state => Comonad ((->) state) where
         extract h = h mempty
         (f <<= h) st = f $ h . mappend st
```

Schreibercomonaden

```
instance Comonad ((,) state) where
         extract (_,a) = a
         f <<= p@(st,_) = (st,f p)
```

Zustandscomonaden

```
instance Comonad (Costate state) where
         extract (h:#st) = h st
         f <<= (h:#st) = (f . (:#) h):#st
```

Listencomonade

```
instance Comonad [ ] where
         extract = head
```

```
f <<= [] = []
f <<= s  = f s:(f <<= tail s)
```

Der cobind-Operator der Listencomonade erweitert eine Operation $f : [a] \to b$ zur Operation

$$(f \mathbin{<<=}) : [a] \to [b].$$

Während $f(s)$ ein einzelner Wert von b ist, liefert f <<= s die Liste der Werte aller Suffixe von s unter f:

$$f \mathbin{<<=} [a_1, \ldots, a_n] \quad = \quad [f[a_1, \ldots, a_n], f[a_2 \ldots, a_n], \ldots, f[a_n]]. \tag{6}$$

Beispiele

id <<= s erzeugt aus s eine Liste der Suffixe von s:

```
id [1..5]        ⇝  [1,2,3,4,5]
id <<= [1..5]    ⇝  [[1,2,3,4,5],[2,3,4,5],[3,4,5],[4,5],[5]]
```

sum(s) berechnet die Summe der Elemente von s:

```
sum [1..8]       ⇝  36
```

sum <<= s schreibt an jede Position i von s die Summe der Elemente von *drop(i)(s)*:

```
sum <<= [1..8]   ⇝   [36,35,33,30,26,21,15,8]
```

9.7 Listen mit Zeiger auf ein Element, comonadisch (siehe Abschnitt 3.6)

```
data ListPos a = (:@) {list :: [a], pos :: Int}

instance Functor ListPos where fmap f (s:@i) = map f s:@i

instance Comonad ListPos where
         extract (s:@i) = s!!i
         f <<= s:@i = map (f . (s:@)) (indices s):@i
```

Die Objekte von *ListPos(a)* sind – wie die von *ListIndex* (siehe Abschnitt 3.6) – Paare, die aus einer Liste und einer Listenposition bestehen.

ListPos(a) wird zu *Costate(a)*, wenn man den Listenfunktor in *ListPos(a)* durch den (semantisch äquivalenten) Leserfunktor $(\rightarrow)(Int)$ ersetzt. Während Listen des Typs $[a]$ zum Programmieren mit *Costate(a)* zunächst in Funktionen des Typs $Int \rightarrow a$ übersetzt werden müssen, kann *ListPos(a)* direkt auf die Listen angewendet werden.

Der cobind-Operator von *ListPos* erweitert eine Operation $f : [a] \times Int \rightarrow b$ zur Operation

$$(f\ <<=) : [a] \times Int \rightarrow [b] \times Int.$$

Im Gegensatz zum cobind-Operator der Listencomonade, der an jede Position i einer Liste

s einen nur von $drop(i)(s)$ abhängigen Wert schreibt, markiert f <<= (s,i) jede Position von s mit einen möglicherweise von ganz s abhängigen Wert:

$$f <<= (s,i) \;=\; ([f(s,0), f(s,1), \ldots, f(s, length(s)-1)], i). \tag{7}$$

Beispiele

```
prefixSum,suffixSum,neighbSum :: ListPos Int -> Int
prefixSum (s:@i) = sum $ take (i+1) s
suffixSum (s:@i) = sum $ drop i s
neighbSum (s:@0) = s!!0+s!!1
neighbSum (s:@i) | i < length s-1 = s!!(i-1)+s!!i+s!!(i+1)
                 | True           = s!!(i-1)+s!!i
```

neighbSum(s:@i) berechnet die Summe von $s!!i$ und den ein bzw. zwei Nachbarn von $s!!i$.

```
list $ prefixSum <<= [1..8]:@0  ⇝  [1,3,6,10,15,21,28,36]
list $ suffixSum <<= [1..8]:@0  ⇝  [36,35,33,30,26,21,15,8]
list $ neighbSum <<= [1..8]:@0  ⇝  [3,6,9,12,15,18,21,15]
```

list(*prefixSum* <<= s : @i) schreibt an jede Position i von s die Summe der ersten $i+1$ Elemente von s.

list(*neighbSum* <<= s : @i) schreibt an jede Position i von s die Summe von $s!!i$ und den

ein bzw. zwei Nachbarn von $s!!i$.

Analog zum Listenfunktor lässt sich auch der Typ *Matrix* quadratischer Matrizen (siehe Abschnitt 8.3) zu einer Comonade erweitern:

```
data MatPos a = (:%) {matrix :: Matrix a, pos2 :: Pos}

instance Functor MatPos where fmap f (mat:%pos) = fmap f mat:%pos

instance Comonad MatPos where
         extract (mat:%pos) = mat!pos
         f <<= mat:%pos = mkMat (dim mat) (f . (mat:%)):%pos
```

Beispiel

```
neighbSum2 :: MatPos Int -> Int
neighbSum2 (mat:%(i,j)) =
     sum $ map (mat!) $ meet [(i-1,j),(i,j-1),(i,j),(i,j+1),(i+1,j)]
                        $ range $ bounds mat
```

neighbSum2(mat:%p) berechnet die Summe von *mat*!*p* und den Nachbarn von *mat*!*p* in der Horizontalen bzw. Vertikalen.

matrix(neighbSum2<<=mat:%p) schreibt an jede Position p von *mat* die Summe von

mat!p und den Nachbarn von *mat!p* in der Horizontalen bzw. Vertikalen.

```
mat1,mat2 :: Matrix Int

mat1 = mkMat 10 $ \(i,j) -> i+j
```

```
    1  2  3  4  5  6  7  8  9 10
  -----------------------------
 1|  2  3  4  5  6  7  8  9 10 11
 2|  3  4  5  6  7  8  9 10 11 12
 3|  4  5  6  7  8  9 10 11 12 13
 4|  5  6  7  8  9 10 11 12 13 14
 5|  6  7  8  9 10 11 12 13 14 15
 6|  7  8  9 10 11 12 13 14 15 16
 7|  8  9 10 11 12 13 14 15 16 17
 8|  9 10 11 12 13 14 15 16 17 18
 9| 10 11 12 13 14 15 16 17 18 19
10| 11 12 13 14 15 16 17 18 19 20
```

erzeugt mit der *Show*-Instanz von *Matrix(Int)* in Coalg.hs

```
mat2 = matrix $ neighbSum2 <<= mat1:%(1,1)
```

```
    1  2  3  4  5  6  7  8  9 10
   ----------------------------
 1|  8 13 17 21 25 29 33 37 41 33
 2| 13 20 25 30 35 40 45 50 55 47
 3| 17 25 30 35 40 45 50 55 60 51
 4| 21 30 35 40 45 50 55 60 65 55
 5| 25 35 40 45 50 55 60 65 70 59
 6| 29 40 45 50 55 60 65 70 75 63
 7| 33 45 50 55 60 65 70 75 80 67
 8| 37 50 55 60 65 70 75 80 85 71
 9| 41 55 60 65 70 75 80 85 90 75
10| 33 47 51 55 59 63 67 71 75 58
```

Offenbar lassen sich mit `MatPos` oder ähnlichen Comonaden **Zellularautomaten** (siehe z.B. [7, 30, 15, 6]) und andere kontextsensitive Systeme implementieren.

9.8 Bäume, comonadisch

Im Folgenden übertragen wir die comonadische Behandlung von Listen mit Zeiger auf binäre und beliebige Bäume mit Zeiger auf einen Knoten (siehe Abschnitte 5.5, 5.9 und 5.10).

```
instance Comonad Bintree where
         extract (Bjoin a _ _) = a
         f <<= Empty           = Empty
         f <<= t@(Bjoin _ u v) = Bjoin (f t) (f <<= u) $ f <<= v

btree1 :: Bintree Int
btree1 = Bjoin 6 (Bjoin 7 (Bjoin 11 (leaf 55) $ leaf 33) $ Empty)
                 $ leaf 9
```

foldBin(t) berechnet die Summe aller Knoteneinträge von t:

```
foldBin btree1 :: Int                ⇝  121          (siehe Abschnitt 5.10)
```

foldBin <<= t markiert jeden Knoten *node* von t mit der Summe der Einträge des Teilbaums von t mit Wurzel mit *node*:

```
foldBin <<= btree1 :: Bintree Int  ⇝  121(106(99(55,33),),9)
```

```
instance Comonad Tree where
         extract = root
         f <<= t@(V _)    = V $ f t
         f <<= t@(F _ ts) = F (f t) $ map (f <<=) ts

tree1 = F 11 $ map (\x -> F x [V $ x+1]) [3..11]
    ⇝  F 11 [F 3 [V 4],F 4 [V 5],F 5 [V 6],F 6 [V 7],F 7 [V 8],
             F 8 [V 9],F 9 [V 10],F 10 [V 11],F 11 [V 12]]
```

$foldTree(\lambda a.\lambda as.a + sum(as))(t)$ berechnet die Summe aller Knoteneinträge von t:

```
foldTree (\a as -> a+sum as) tree1      ⇝  146
```
(siehe Abschnitt 5.10)

$foldTree(\lambda a.\lambda as.a + sum(as))$ <<= t markiert jeden Knoten *node* von t mit der Summe der Einträge des Teilbaums von t mit Wurzel *node*:

```
foldTree (\a as -> a+sum as) <<= tree1  ⇝
      F 146 [F 7 [V 4],F 9 [V 5],F 11 [V 6],F 13 [V 7],F 15 [V 8],
             F 17 [V 9],F 19 [V 10],F 21 [V 11],F 23 [V 12]]

tree2 = F "+" [F "*" [V "x",V "y"], V "z"]
```

```
ops1 :: String -> [Int] -> Int
ops1 = \case "+" -> sum; "*" -> product
             "x" -> const 5; "y" -> const $ -66; "z" -> const 13
```

foldTree(*ops*)(*t*) faltet den Baum *t* zu einem Wert gemäß der durch *ops* gegebenen Interpretation seiner Knotenmarkierungen:

```
foldTree ops1 tree2      ⇝  -317
```

foldTree(*ops*) <<= *t* markiert jeden Knoten *node* von *t* mit dem Wert der *ops*-Faltung des Teilbaums von *t* mit Wurzel *node*:

```
foldTree ops1 <<= tree2  ⇝  F (-317) [F (-330) [V 5,V (-66)],V 13]
```

Der Übergang von der *Tree*- zur *TreeNode*-Comonade ist genauso motiviert wie derjenige von der Listen- zur *ListPos*-Comonade (s.o.):

```
data TreeNode a = (:&) {tree :: Tree a, node :: Node}

instance Functor TreeNode where fmap f (t:&node) = mapTree f t:&node
```

```
mkTN :: Tree a -> TreeNode a
mkTN t = t:&[]


labelTN :: TreeNode a -> a
labelTN (t:&node) = label t node

children :: TreeNode a -> [Tree a]
children (t:&node) = subtrees $ getSubtree t node

rootTN,leafTN,fstchild :: TreeNode a -> Bool
rootTN (_:&node)   = null node
leafTN             = null . children
fstchild (_:&node) = last node == 0

parent,prevchild,lastchild :: TreeNode a -> TreeNode a
parent (t:&node)       = t:&init node
prevchild (t:&node)    = t:&init node++[last node-1]
lastchild tn@(t:&node) = t:&node++[lg-1]
                         where lg = length $ children tn
```

```
nodeTree :: Tree a -> Node -> Tree Node
nodeTree (V _) node    = V node
nodeTree (F _ ts) node = F node $ zipWith f ts [0..length ts-1]
                         where f t i = nodeTree t $ node++[i]
```

nodeTree$(t)[]$ markiert jeden Knoten von t mit seiner Darstellung als Liste ganzer Zahlen (siehe Abschnitt 5.9):

```
nodeTree tree1 []
        ⇝  F [] [F [0] [V [0,0]],F [1] [V [1,0]],F [2] [V [2,0]],
                F [3] [V [3,0]],F [4] [V [4,0]],F [5] [V [5,0]],
                F [6] [V [6,0]],F [7] [V [7,0]],F [8] [V [8,0]]]
```

```
instance Comonad TreeNode where
                extract (t:&node) = label t node
                f <<= (t:&node) = mapTree (f . (t:&)) nt:&node
                                  where nt = nodeTree t []

prefixSumTN :: TreeNode Int -> Int
prefixSumTN tn | rootTN tn = root $ tree tn
               | True      = prefixSumTN (parent tn)+labelTN tn
```

prefixSumTN$(t : \&node)$ berechnet die Summe der Markierungen des Knotens *node* und seiner Vorgänger (siehe Abschnitt 5.9).

tree(*prefixSumTN* $<<=$ t:&*node*) markiert jeden Knoten *node* von t mit der Summe der Markierungen seiner Vorgänger einschließlich *node*:

```
tree $ prefixSumTN <<= mkTN tree1
       ↝  F 11 [F 14 [V 18],F 15 [V 20],F 16 [V 22],F 17 [V 24],
               F 18 [V 26],F 19 [V 28],F 20 [V 30],F 21 [V 32],
               F 22 [V 34]]

depthTN,breadthTN :: TreeNode a -> Int
depthTN (t:&node)   = fromJust $ lookup node
                                 $ zip (depthfirst $ nodeTree t [])
                                       [0..]
breadthTN (t:&node) = fromJust $ lookup node
                                 $ zip (breadthfirst [nodeTree t []])
                                       [0..]
```

depthTN$(t : \&node)$ bzw. *breadthTN*$(t : \&node)$ berechnet die Position von *node* in der Knotenliste, die ein depthfirst- bzw. breadthfirst-Durchlauf von t erzeugt (siehe Abschnitt 5.9).

tree(*depthTN* <<= *t* : &*node*) bzw. *tree*(*breadthTN* <<= *t* : &*node*) markiert jeden Knoten *node* von *t* mit der Position von *node* in der Knotenliste, die ein depthfirst- bzw. breadthfirst-Durchlauf von *t* erzeugt:

```
tree $ depthTN <<= mkTN tree1
       ⇝  F 0 [F 1 [V 2],F 3 [V 4],F 5 [V 6],F 7 [V 8],F 9 [V 10],
           F 11 [V 12],F 13 [V 14],F 15 [V 16],F 17 [V 18]]
```

Wir schließen mit einer direkteren und daher – bei intelligenter Implementierung von *TreeNode*, z.B. durch *Data.Map.Strict.Map(Node)* – möglicherweise effizientere Version von *depthTN* (angelehnt an das numin/numout-Beispiel von T. Uustalu, V. Vene, *Comonadic functional attribute evaluation*, in: M. van Eekelen, ed., *Trends in Functional Programming 6*, Intellect 2007, S. 145-162):

```
depthUV :: TreeNode a -> Int
depthUV tn | rootTN tn   = 0
           | fstchild tn = depthUV (parent tn)+1
           | True        = aux (prevchild tn)+1 where
                           aux tn | leafTN tn = depthUV tn
                                  | True      = aux $ lastchild tn
```

Literatur

[1] Richard Bird, Using Circular Programs to Eliminate Multiple Traversals of Data, Acta Informatica 21 (1984) 239-250

[2] Richard Bird, Introduction to Functional Programming using Haskell, Prentice Hall 1998 (in der Lehrbuchsammlung unter L Sr 449/2)

[3] Richard Bird, Pearls of Functional Algorithm Design, Cambridge University Press 2010

[4] Richard Bird, Thinking Functionally with Haskell, Cambridge University Press 2014

[5] Marco Block, Adrian Neumann, Haskell-Intensivkurs, Springer 2011

[6] Silvio Capobianco, Tarmo Uustalu, A Categorical Outlook on Cellular Automata, Journées Automates Cellulaires 2010 (Turku) 88-99

[7] Conway's Game of Life, https://en.wikipedia.org/wiki/Conway's_Game_of_Life

[8] Manuel M. T. Chakravarty, Gabriele C. Keller, Einführung in die Programmierung mit Haskell, Pearson Studium 2004

[9] Ernst-Erich Doberkat, Haskell: Eine Einführung für Objektorientierte, Oldenbourg 2012

[10] Kees Doets, Jan van Eijck, The Haskell Road to Logic, Maths and Programming, Texts in Computing Vol. 4, King's College 2004

[11] Paul Hudak, The Haskell School of Expression: Learning Functional Programming through Multimedia, Cambridge University Press 2000

[12] Paul Hudak, John Peterson, Joseph Fasel, A Gentle Introduction to Haskell, Yale and Los Alamos 2000

[13] Michael Huth, Mark Ryan, Logic in Computer Science, Cambridge University Press 2004

[14] Graham Hutton, Programming in Haskell, Cambridge University Press 2007

[15] Jarkko Kari, Theory of cellular automata: A survey, Theoretical Computer Science 334 (2005) 3-33

[16] Zohar Manna, Mathematical Theory of Computation, McGraw-Hill 1974

[17] P. Padawitz, *Algebraic Model Checking*, in: F. Drewes, A. Habel, B. Hoffmann, D. Plump, eds., Manipulation of Graphs, Algebras and Pictures, Electronic Communications of the EASST Vol. 26 (2010)

[18] P. Padawitz Grundlagen und Methoden funktionaler Programmierung, TU Dortmund 1999

[19] P. Padawitz, Logik für Informatiker, Technical Reports in Computer Science No. 867, TU Dortmund 2019

[20] P. Padawitz, Formale Methoden des Systementwurfs, TU Dortmund 2007

[21] P. Padawitz Übersetzerbau-Folienskript, TU Dortmund 2017

[22] P. Padawitz, Übersetzerbau, TU Dortmund 2015

[23] P. Padawitz, Fixpoints, Categories, and (Co)Algebraic Modeling, TU Dortmund 2020

[24] P. Padawitz, From Modal Logic to (Co)Algebraic Reasoning, TU Dortmund 2020

[25] Peter Pepper, Petra Hofstedt, Funktionale Programmierung: Sprachdesign und Programmiertechnik, Springer 2006

[26] Fethi Rabhi, Guy Lapalme, Algorithms: A Functional Programming Approach, Addison-Wesley 1999

[27] Simon Thompson, Haskell: The Craft of Functional Programming, 3. Auflage, Addison-Wesley 2011

[28] Raymond Turner, Constructive Foundations for Functional Languages, McGraw-Hill 1991

[29] Jean Vuillemin, Correct and Optimal Implementations of Recursion in a Simple Programming Language, Journal of Computer and System Sciences 9 (1974) 332-354

[30] Stephen Wolfram, A New Kind of Science, Wolfram Media 2002

Index

Σ-Baum, 183
λ-Abstraktion, 18
λ-Applikation, 19
μ-Abstraktion, 215
n-Tupel, 13
(++), 45
(//), 288

Abschlussoperator, 187
Algebra, 107
all, 75
Alternative, 236
anonyme Funktion, 18
any, 75
Applicative, 235
Applikationsoperator, 33
Array, 287
array, 287
Attribut, 96

bind, 231
Bintree, 133
BintreeL, 135

call-by-need-Strategie, 20
card, 75
checkResult, 238
co-CPO, 175
co-Kette, 175
co-stetig, 177
co-vollständig, 175
cobind, 327
Comonad, 327
Compiler, 319
const, 34
Constraints, 128
context, 72
Cotree, 154
CPO, 175

curry, 37

Datentyp, 14
Destruktor, 95
diff, 130
disjunkte Vereinigung, 16
do-Notation, 234
drop, 46
duplicate, 328
dynamisches Objekt, 281

elem, 75
endliche Funktion, 190
Eq, 128
eqset, 130
Exp, 103
Expansionsgleichung, 217
extract, 327

fail, 231
Faltung, 107
field label, 95
filter, 75
Fixpunkt, 177
Fixpunktsatz von Kleene, 177
flacher CPO, 178
flip, 36
fold2, 64
foldl, 58
foldlfl, 67
foldr, 65
fst, 65
Functor, 226
Funktion höherer Ordnung, 25
funktionale Abhängigkeit, 226
Funktionsapplikation, 19
Funktionsiteration, 42, 89

Grundinstanz, 19
Guard, 28
guard, 237

Halbordnung, 175
head, 45
Hue-Farbe, 26

id, 34
indices, 45
Individuenvariable, 18
init, 46
insert, 129
Instanz, 19
Instanz eines Terms, 259
Instanz eines Typs, 12
isPerm, 76
iterate, 88

join, 238

Kellermaschine, 159
Kette, 175
Kind, 225
Komponente eines Tupels, 13
Kompositionsoperator, 33
konstruktive Signatur, 107
Konstruktor, 14, 92

last, 46
lazy-Strategie, 20, 36
liftM, 233
liftM2, 233
lines, 53
Liste, 43
Listenkomprehension, 76
logische Reduktion, 207
lokale Definition, 28
lookup, 56
lookupL, 57
lookupM, 233

many, 239
map, 51
mapM, 240
Matching, 19
meet, 130
mehrwertige Funktion, 246
Methode, 96
minis, 78
mkArray, 287
Monad, 231

MonadPlus, 231
monomorpher Typ, 12
monoton, 177
mplus, 231
msum, 239
mzero, 231

nichtdeterministische Funktion, 246
notElem, 75
null, 45

Objektklasse, 96
Operation, 107

parallel-outermost-Strategie, 20, 36
partielle Funktion, 242
parts, 70
pattern binding, 30, 87
perms, 69
permsI, 69
polymorphe Funktion, 18
polymorpher Typ, 12
Poset, 175
powerset, 130
Produktbildung, 13
Produktextension, 114

range, 287
Record, 96
Redex, 20
reduceE, 164
Redukt, 20
rekursiv definierte Funktion, 38
rekursiver Datentyp, 17
rel2fun, 64
rel2funL, 66
remove, 130
repeat, 88
replicate, 88
return, 231
reverse, 47
Ring, 186
root, 149

Schrittfunktion, 177

Seiteneffektmonade, 270
Sektion, 32
Selektor, 96
Semiring, 186
sequence, 239
Show, 141
Signatur, 107
snd, 65
some, 239
splitAt, 48
StackCom(x), 159
statisches Objekt, 280
stetig, 176
strikte Funktion, 67
subset, 130
Substititution, 259
Substitution, 19
subtrees, 150
Summenbildung, 13
Summentyp, 16
symbolischen Differentiation, 113
Syntaxbäume, 323

tail, 45
take, 46
Term, 107
Trägermenge, 107
transitiver Abschluss, 195
Tree, 149
Typfamilie, 225
Typinferenzregeln, 21
Typkonstruktor, 12
Typvariable, 12

uncurry, 37
Unfikator, 261
unifizierbar, 261
union, 129
unionMap, 129
unit-Typ, 12
unlines, 53
unwords, 52
unzip, 52

update, 35
updList, 47

vollständig, 175
vollständiger Semiring, 186
vollständiger Verband, 175

when, 238
Wildcard, 35
words, 52
Wort, 43

zip, 51
zipWith, 51
zipWithM, 240
Zustandsäquivalenz, 206