



*Modellieren und Implementieren in Haskell*  
*Modeling and Implementing in Haskell*

Peter Padawitz, TU Dortmund, Germany  
23. Oktober 2017

(actual version: <http://fdit-www.cs.uni-dortmund.de/~peter/Essen.pdf>)

Webseiten der zugehörigen Lehrveranstaltungen:

Funktionale Programmierung

Funktionales und regelbasiertes Programmieren

# Inhalt

Mit \* markierte Abschnitte bzw. Kapitel werden in der LV **Funktionale Programmierung** nicht behandelt.

Zur Navigation auf Titel, nicht auf Seitenzahlen klicken!

1	Vorbemerkungen	8
2	Typen und Funktionen	11
3	Listen	40
1	Listenteilung	45
2	Listenintervall	47
3	Funktionslifting auf Listen	48
4	Listenmischung	47
5	Strings sind Listen von Zeichen	49
6	Listen mit Zeiger auf ein Element	50
7	Relationsdarstellung von Funktionen	52
8	Listenfaltung	53
9	Teillisten, Permutationen, Partitionen, Linienzüge	62

10	Listenlogik	66
11	Listenkomprehension	67
12	Unendliche Listen	75
4	Rekursive Datentypen	79
1	Arithmetische Ausdrücke	90
2	Boolesche Ausdrücke	92
3	Arithmetische Ausdrücke auswerten	94
4	Symbolische Differentiation	96
5	Hilbertkurven	97
5	Typklassen und Bäume	102
1	Mengenoperationen auf Listen	103
2	Unterklassen	105
3	Sortieralgorithmen	105
4	Binäre Bäume	107
5	Binäre Bäume mit Zeiger auf einen Knoten	109
6	Ausgeben	114
7	Arithmetische Ausdrücke ausgeben	115

8	Einlesen	118
9	Bäume mit beliebigem Ausgrad	122
10	Baumfaltungen	128
11	Arithmetische Ausdrücke kompilieren	132
12	* Arithmetische Ausdrücke reduzieren	137
6	Fixpunkte, Graphen und Modallogik	142
1	CPOs und Fixpunkte	142
2	Semantik rekursiver Funktionsgleichungen	145
3	Semiringe	148
4	Graphen	152
5	* Semantik modallogischer Formeln	157
6	* Zweidimensionale Figuren	165
7	Funktoren und Monaden	170
1	Kinds: Typen von Typen	170
2	Funktoren	173
3	Monaden und Plusmonaden	177
4	Monaden-Kombinatoren	180

5	Die Identitätsmonade	186
6	Maybe- und Listenmonade	187
7	Lesermonaden	203
8	Schreibermonaden	204
9	* Substitution und Unifikation	207
10	Zustandsmonaden	211
11	Die IO-Monade	214
12	Zustandsvariablen (mutable, dynamic objects)	217
8	Felder	226
1	<b>ix</b> , die Typklasse für Indexmengen	226
2	Dynamische Programmierung	228
3	Matrizenrechnung	229
4	Graphen als Matrizen	232
5	* Alignments	238
9	Monadentransformer und Comonaden	246
1	Huckepack-Zustandsmonaden	246
2	Monadentransformer zur Verschmelzung zweier Monaden	248

3	Verschmelzung von Leser- und Maybe-Monade	251
4	Verschmelzung von IO- und Maybe-Monade	253
5	Verschmelzung von IO- und Listenmonade	255
6	Verschmelzung von IO- und Huckepack-Zustandsmonade	258
7	Generische Compiler	259
8	Arithmetische Ausdrücke kompilieren II	263
9	* Comonaden	269
10	* Nochmal Listen mit Zeiger auf ein Element	273
11	* Bäume, comonadisch	275
10	Semantik und Verifikation funktionaler Programme	280
1	* Das relationale Berechnungsmodell	281
2	* Das funktionale Berechnungsmodell	287
3	Schemata zur Definition partieller Funktionen	295
4	* Termination und Konfluenz	301
5	* Partiell-rekursive Funktionen	305
6	* Funktionen mit beliebigen lokalen Definitionen	309
7	* Die lazy-evaluation-Strategie	312
8	* Auswertung durch Graphreduktion	317

9	* Unendliche Objekte	325
10	* Verifikation	328
11	Bücher und Skripte	336
12	Index	337

## 1 Vorbemerkungen

Die gesternten Abschnitte werden nicht in der Bachelor-LV [Funktionale Programmierung](#) behandelt, sondern zum Teil in den Wahlveranstaltungen [Funktionales und regelbasiertes Programmieren](#) (Master und Diplom), [Einführung in den logisch-algebraischen Systementwurf](#) (Bachelor und Diplom) und [Logisch-algebraischer Systementwurf](#) (Master und Diplom).

Die Folien dienen dem Vor- (!) und Nacharbeiten der Vorlesung, können und sollen aber deren regelmäßigen Besuch nicht ersetzen!

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 Seite [Funktionale Programmierung](#) zur LV.

Alle im Folgenden verwendeten Haskell-Funktionen – 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 HörerInnen weniger Schwierigkeiten. Ihr Einsatz in programmiersprachlichen Lösungen algorithmischer Probleme aus ganz unterschiedlichen Anwendungsbereichen ist aber auch für diese Hörerschaft vorwiegend Neuland.

Diese Folien bilden daher i.w. eine **Sammlung prototypischer Programmbeispiele**, auf die, falls sie eingehend studiert und verstanden worden sind, zurückgegriffen werden kann, wenn später ein dem jeweiligen Beispiel ähnliches Problem funktionalsprachlich gelöst werden soll. Natürlich werden wichtige Haskell-Konstrukte auch allgemein definiert. Vollständige formale Definitionen, z.B. in Form von Grammatiken, finden sich hier jedoch nicht. Dazu wie auch zur allgemeinen Motivation für einzelne Sprachkonstrukte sei auf die zunehmende Zahl an Lehrbüchern, Tutorials und Sprachreports verwiesen (siehe [Haskell-Lehrbücher](#) und die Webseite [Funktionale Programmierung](#)).

Alle Hilfsfunktionen und -datentypen, die in den Beispielen vorkommen, werden auch hier – manchmal in vorangehenden Abschnitten – eingeführt. Wenn das zum Verständnis nicht ausreicht und auftretende Fragen nicht in angemessener Zeit durch Zugriff auf andere o.g. Quellen geklärt werden können, dann stellt die Fragen in der Übung, dem Tutorium oder der Vorlesung!

## 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**.
- Die standardmäßige *lazy evaluation* von Haskell-Programmen erlaubt die Implementierung von – sonst nur in Entwurfssprachen verwendeten – **unendlichen Objekten** wie Datenströmen, Prozessbäumen u.ä., sowie die Berechnung von Gleichungslösungen wie in **logischer/relationaler Programmierung** (Prolog, SQL).
- **Datentypen mit Destruktoren** sowie **Monaden** erlauben es, auch **imperativ** und **objekt-**oder **aspektorientiert** programmieren.

## 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  $x_1, \dots, x_n$  bezeichnet eine  $n$ -stellige Funktion, die jedem  $n$ -Tupel von Instanzen von  $x_1, \dots, x_n$  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  $\mathbf{1}$  für die Menge und  $\epsilon$  für ihr 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  $A_1, \dots, A_n$  Mengen (Typen).

## Produkttypen

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

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

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

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

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

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  $x, y, b, h \in \textit{Float}$  und  $c \in \textit{Color}$ .

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

$$DRect(Point(x)(y))(b)(h)(c).$$

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}
```

## Summentypen

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

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

In Haskell kann  $A_1 + \dots + A_n$  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  $b \in Bool$ .

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.



## 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 (totalen) Funktionen von  $A$  nach  $B$ . Jedes Element von  $A \rightarrow B$  wird

- entweder benannt und durch Gleichungen definiert (s.u.)
- oder unbenannt (*anonym*) als  **$\lambda$ -Abstraktion**  $\lambda p.e$  (Haskell-Notation: `\p -> e`) dargestellt.

Im zweiten Fall ist  $p$  ein Muster für die möglichen Argumente (Parameter) der durch  $\lambda p.e$  dargestellten Funktion.

$e$  nennt man auch Rumpf (*body*) von  $\lambda p.e$  und ist ein aus beliebigen Funktionen (zu denen auch Konstruktoren zählen) und Variablen zusammengesetzter Ausdruck.

**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:

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

Welche Symbole bezeichnen hier Variablen, Konstruktoren bzw. Attribute?

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

Die  $\lambda$ -Applikation  $(\lambda p.e)(e')$  ist auswertbar, wenn der Ausdruck  $e'$  eine **Instanz** von  $p$  ist ( $e'$  **matcht**  $p$ )., d.h.  $e'$  ist das Ergebnis der Anwendung einer passenden **Substitution** (= Zuordnung von Ausdrücken zu Variablen)  $\sigma$  auf  $p$ , kurz:  $e' = p\sigma$ . Die Anwendung von  $\sigma$  auf  $p$  besteht in der Ersetzung der Variablen von  $p$  durch Ausdrücke gemäß der durch  $\sigma$  gegebenen Zuordnung von Ausdrücken zu Variablen. Gilt  $e' = p\sigma$ , dann ist die Applikation  $(\lambda p.e)(e')$  semantisch äquivalent zu  $e\sigma$  und wird deshalb zunächst zu  $e\sigma$  ausgewertet.

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](#)

Der **Redex**, d.i. der Teilausdruck, der im jeweils nächsten Schritt ersetzt wird, ist **rot** gefärbt. Das **Redukt**, d.i. der Teilausdruck, durch den der Redex ersetzt wird, ist **grün** gefärbt. Reduktionen können mit der reduce-Funktion des **Painters** erzeugt werden.

Die Auswahl eines Redex erfolgt stets nach der **leftmost-outermost-** oder **lazy-Strategie**, die den zu reduzierenden Ausdruck von der Wurzel aus in Präordnung durchläuft und den ersten Teilausdruck, auf den eine Reduktionsregel anwendbar ist, als Redex auswählt.

[Link zur schrittweisen Auswertung der  \$\lambda\$ -Applikation](#)

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

In klassischer Algebra und Analysis taucht  $\lambda$  bei der Darstellung von Funktionen nicht auf, wenn Symbole wie  $x, y, z$  konventionsgemäß als Variablen betrachtet und daher z.B. für die Polynomfunktion  $\lambda x.2 * x^3 + 55 * x^2 + 33 : \mathbb{R} \rightarrow \mathbb{R}$  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  $\lambda$ - und anderen Ausdrücken ausgegeben werden:

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

Das Backslash-Symbol  $\backslash$  und der Pfeil  $\rightarrow$  sind offenbar die Haskell-Notationen des Symbols  $\lambda$  bzw. des Punktes einer  $\lambda$ -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:

$$\frac{e_1 :: t_1, \dots, e_n :: t_n}{(e_1, \dots, e_n) :: (t_1, \dots, t_n)}$$

$$\frac{p :: t, \quad e :: t'}{\lambda p. e :: t \rightarrow t'}$$

$$\frac{e :: t \rightarrow t', \quad e' :: t}{e(e') :: t'}$$

$$\frac{}{Nothing :: Maybe t}$$

$$\frac{e :: t}{Just e :: Maybe t}$$

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

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

## Gleichungen, die Konstanten 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.

## Gleichungen, die Funktionen definieren

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

$f = \backslash p \rightarrow e$       *äquivalent ist zur* **applikativen Definition**       $f\ p = e$

Die so definierte Funktion  $f : A \rightarrow B$  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-*Compiler* erlauben 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:

$$f \text{ p1} = e1; \quad f \text{ p2} = e2; \quad \dots \quad f \text{ pn} = en \quad (1)$$

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* einen 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:

$$f \text{ x} = \text{case } x \text{ of } p1 \rightarrow e1; \quad p2 \rightarrow e2; \quad \dots \quad pn \rightarrow en \quad (2)$$

```
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  $\lambda$ -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
```

## 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

$$F = A_1 \rightarrow (A_2 \rightarrow \cdots \rightarrow (A_n \rightarrow B) \dots) \quad (1)$$

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

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

$$((\dots ((f \ a_1) \ a_2) \dots) \ a_{n-1}) \ a_n \quad (2)$$

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 geschachtelte  $\lambda$ -Abstraktion der Form  $\lambda p_1. \lambda p_2. \dots \lambda p_n. e$  ein Element von  $F$  und kann durch  $\lambda p_1 \ p_2 \ \dots \ p_n. e$  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
```

*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
```

Wo sich die Farben von *Color* in *RGB* 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
```

Offenbar können neben Mustern auch Boolesche Ausdrücke die Fallunterscheidung einer applikativen Funktionsdefinition bestimmen. Sie werden mit dem Symbol `|` vom jeweiligen Muster getrennt.

$RGB(r)(g)(b)$  ist (die Codierung) eine(r) Hue-Farbe, wenn  $r, g, b \in \{0, \dots, 255\}$  und  $0, 255 \in \{r, g, b\}$  gilt. Demnach ist `nextRGB` nur auf Hue-Farben definiert, also eine partielle Funktion.

Andere Elemente  $RGB(r)(g)(b)$  von  $RGB$  mit  $r, g, b \in \{0, \dots, 255\}$  repräsentieren eine aufgehellte oder abgedunkelte Variante einer Hue-Farbe.

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  $f : Int \rightarrow Int \rightarrow Int \rightarrow Bool$ :

```
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)
```

*oder mit dem let-Konstrukt:*

```
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
```

*oder mit den Attributen von Point:*

```
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
```

*oder 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
```

*oder mit @-Symbol:*

```

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

```

oder 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 können. Dabei werden die Definitionen in *einer* Zeile durch ; voneinander getrennt.

Das oben verwendete Konditional *if\_then\_else\_* ist eine – *mixfix* notierte – Funktion des Typs

$$Bool \rightarrow Point \rightarrow Point \rightarrow Point.$$

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  $A \rightarrow A \rightarrow A$ .

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  $A \rightarrow B \rightarrow C$  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`  
 $\text{mod } 9 \ 5$      *ist äquivalent zu*      $9 \ \text{`mod`} \ 5$

Die Infixdarstellung wird auch verwendet, um die in  $f$  enthaltenen **Sektionen** (Teilfunktionen) des Typs  $A \rightarrow C$  bzw.  $B \rightarrow C$  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*      $\backslash x \rightarrow 5+x$

$(9 \text{ mod } x)$  *ist äquivalent zu*  $\text{mod } 9$  *ist äquivalent zu*  $\backslash x \rightarrow 9 \text{ mod } x$

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:

$(\$)$   $:: (a \rightarrow b) \rightarrow a \rightarrow b$   
 $f \$ a = f a$

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:

$f1 \$ f2 \$ \dots \$ fn a \rightsquigarrow f1 (f2 (\dots (fn a) \dots))$

Demgegenüber ist der **Kompositionsoperator**

$(.)$   $:: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$   
 $(g . f) a = g (f a)$

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



$$(f1 \ . \ f2 \ . \ \dots \ . \ fn) \ a \quad \rightsquigarrow \quad f1 \ (f2 \ (\dots (fn \ a) \dots))$$

U.a. benutzt man den Kompositionsoperator, um in einer applikativen Definition Argumentvariablen einzusparen. So sind z.B. die folgenden drei Definitionen einer Funktion  $f : a \rightarrow b \rightarrow c$  äquivalent zueinander:

$$\begin{aligned} f \ a \ b &= g \ (h \ a) \ b \\ f \ a &= g \ (h \ a) \\ f &= g \ . \ h \end{aligned}$$

Beispiel

$$\text{isNothing} = \text{not} \ . \ \text{isJust}$$

Weitere polymorphe Funktionen, die Funktionen erzeugen bzw. verändern

`id :: a -> a`

*Identität*

`id a = a`

`id = \a -> a`

`const :: a -> b -> a`

*konstante Funktion*

```

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

```

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      Funktionsupdate
update f a b a' = if a == a' then b else f a'      (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 besteht in der Anwendung der ersten passenden Definitionsgleichung auf die am weitesten links stehende Funktion, für die eine solche Gleichung existiert. Diese Auswertungsstrategie nennt man – wie schon bei der Auswertung von  $\lambda$ - Applikationen bemerkt wurde – **lazy** oder – bezogen auf die Baumdarstellung funktionaler Ausdrücke – **leftmost-outermost**.

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`      *objektorientierte Applikation (Punktnotation)*  
`(&) = flip ($)`

`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*

(#, *X* und *Y* stehen hier für &, *x* bzw. *y*. Attribute sind weiß unterlegt und als *Kantenmarkierungen* zu lesen.)

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

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

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

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

*Produktextension*

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

## Rekursiv definierte Funktionen

$f : A \rightarrow B$  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
```

*oder mit Fallunterscheidung (und nur auf natürlichen Zahlen definiert):*

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

```
fact 5 ~> 120
```

Link zur schrittweisen Auswertung von  $fact(5)$ : [applikative Version](#); [case-Version](#)  
 $\lambda(p_1, t_1, \dots, p_n, t_n)$  steht hier für  $\backslash case\ p_1 \rightarrow t_1; \dots; p_n \rightarrow t_n$ .

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}
```

Rekursive Definition der Funktionsiteration (nicht im Prelude)

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

```
((+2)`hoch`4) 10  $\rightsquigarrow$  18
```

Link zur schrittweisen Auswertung von  $((+2)`hoch`4) 10$

**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`hoch`765
```

### 3 Listen

Sei  $A$  eine Menge. Die Elemente der Mengen

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

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  $[a_1, \dots, a_n]$  anstelle von  $(a_1, \dots, a_n)$ .

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

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

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

Die Konstante  $[]$  (leere Liste) vom Typ  $[A]$  und die Funktion  $(:)$  (*append*; Anfügen eines Elementes von  $A$  ans linke Ende einer Liste) vom Typ  $A \rightarrow [A] \rightarrow [A]$  heißen – analog zum Tupelkonstruktor für kartesische Produkte (siehe Kapitel 2) – **Listenkonstruktoren**, da sich mit ihnen jede Haskell-Liste darstellen lässt.

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



Analog zu den Typinferenzregeln für Tupel,  $\lambda$ -Abstraktionen und  $\lambda$ -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] ~> 5
```

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

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

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

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

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

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

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

```
indices [3,2,8,4]  $\rightsquigarrow$  [0,1,2,3]
```

```
null [3,2,8,4]  $\rightsquigarrow$  False
```

```
head [3,2,8,4]  $\rightsquigarrow$  3
```

```
tail [3,2,8,4]  $\rightsquigarrow$  [2,8,4]
```

```
[3,2,4]++[8,4,5]  $\rightsquigarrow$  [3,2,4,8,4,5]
```

```
[3,2,4]!!1  $\rightsquigarrow$  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\[2,4,5,7,88\]`](#)

## Listenteilung

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

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt 0 s          = ([],s)
splitAt _ []         = ([],[])
splitAt n (a:s) | n > 0 = (a:s1,s2) where
                                (s1,s2) = splitAt (n-1) s    lokale Definition
```

`splitAt(3) [5..12]`  
 $\leadsto$  `([5,6,7],[8..12])`

*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) where
                    (s1,s2) = span f s'
span _ _        = ([],[])
```

Haskell übersetzt lokale Definitionen in  $\lambda$ -Applikationen:

Aus  $t \text{ where } p = u$  wird  $(\lambda p.t)(u)$ .

Z.B. ist die Gleichung

```
splitAt n (a:s) | n > 0 = (\(s1,s2) -> (a:s1,s2)) $ splitAt (n-1) s
```

äquivalent zur dritten Gleichung der Definition von `splitAt`.

[Link zur schrittweisen Auswertung von `splitAt\(3\)\[5..12\]`](#)

Die Gleichung

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

ist äquivalent zur ersten Gleichung der Definition von `span`.

Ein **logisches Programm** würde anstelle der schrittweisen Auswertung des Ausdrucks `splitAt(3)[5..12]` die Gleichung

$$\text{splitAt}(3)[5..12] = s \tag{1}$$

mit der Listenvariablen  $s$  schrittweise lösen, d.h. in eine Gleichung transformieren, die eine Lösung von (1) in  $s$  repräsentiert. [Link zur schrittweisen Lösung von \(1\)](#)

## Listenintervall

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 _ _ _                                   = []
```

## Listenmischung

$merge(s_1, s_2)$  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
```

## Funktionslifting auf Listen

```
map :: (a -> b) -> [a] -> [b]
```

```
map f (a:s) = f a:map f s
```

```
map _ _     = []
```

```
map (+3) [2..9]            $\rightsquigarrow$  [5..12]
```

```
map ($) 7 [(+1),(+2),(*5)]  $\rightsquigarrow$  [8,9,35]
```

```
map ($) a [f1,f2,...,fn]   $\rightsquigarrow$  [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]  $\rightsquigarrow$  [11,11,43]
```

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

```
zip = zipWith (,)
```

```
zip [3,2,8,4] [8,9,35]  $\rightsquigarrow$  [(3,8),(2,9),(8,35)]
```



## 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.

## Listen mit Zeiger auf ein Element *(nicht im Prelude)*

`type ListIndex a = ([a],Int)`     *Liste und Index  $n$*   
`type ListZipper a = ([a],[a])`     *Zerlegung einer Liste  $s$  mit Index  $n$  in die  
Kontextliste  $c = \text{reverse}(\text{take}(n)(s))$  und  
das Suffix  $\text{drop}(n)(s)$  von  $s$*

```
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)  $\rightsquigarrow$  ([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])  $\rightsquigarrow$  ([1..9],4)`

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

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

Zeigerbewegungen auf Zipper-Listen kommen ohne den Aufruf von Hilfsfunktionen aus:

`back,forth :: ListZipper a -> ListZipper a`

`back (a:c,s) = (c,a:s)`

`forth (c,a:s) = (a:c,s)`

## Relationsdarstellung von Funktionen

Eine Funktion  $f$  mit endlichem Definitionsbereich lässt sich als Liste ihrer (Argument, Wert)-Paare implementieren. Mathematisch nennt man sie den **Graphen von  $f$** . Als programmiersprachliches Konstrukt wird sie auch als **Assoziationsliste** oder **Dictionary** bezeichnet.

Eine Applikation der Funktion 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. Die Typklasse `Eq a` (siehe Kapitel 5) beschränkt die Instanzen von  $a$  auf Typen mit einer Funktion  $(==) :: a -> a -> a$ .

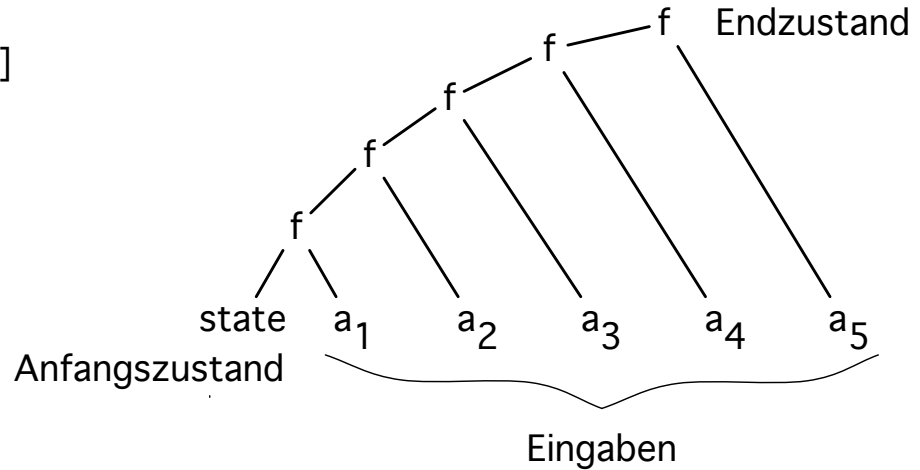
Die folgende Funktion *updRel* überträgt die Funktion *update* von Kapitel 2 auf die Listendarstellung von Funktionen:

```
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)]
```

## Listenfaltung

### Faltung einer Liste von links her

`foldl f state [a1,a2,a3,a4,a5]`



`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`

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

```
reverse = foldl (\s x->x:s) []
reverse = foldl (flip (:)) []
```

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

```
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

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

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

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

```
horner :: [Float] -> Float -> Float
```

```
horner as x = foldl1 f as where f state a = state*x+a
```

## Beispiel Binomialkoeffizienten

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

```
binom :: Int -> Int -> Int
```

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

Die induktive Definition der Binomialkoeffizienten:

$$\begin{aligned} \forall n \in \mathbb{N} : \quad & \binom{n}{0} = \binom{n}{n} = 1 \\ \forall n \in \mathbb{N} \quad \forall i < n : \quad & \binom{n}{i} = \binom{n-1}{i-1} + \binom{n-1}{i} \end{aligned}$$

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:

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

```

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  $partsno(n) = length(parts[1..n])$ , wobei  $parts$  wie auf Folie 37 definiert ist.

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



1																					
1					1																
1			2			1															
1		3		3		1															
1		4		6		4		1													
1		5		10		10		5		1											
1		6		15		20		15		6		1									
1		7		21		35		35		21		7		1							
1		8		28		56		70		56		28		8		1					
1		9		36		84		126		126		84		36		9		1			
1		10		45		120		210		252		210		120		45		10		1	

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

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

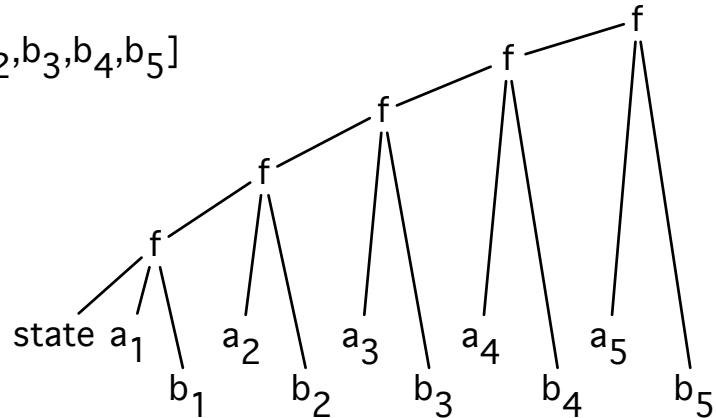
*text*( $n$ ) fasst  $n$  als String auf. *frame*( $x$ ) rahmt den String  $x$ . *shelf*( $n$ )( $s$ ) teilt die Liste  $s$  in Teillisten der Länge  $n$  auf, deren jeweilige Elemente horizontal aneinandergesetzt werden. (Die letzte Teilliste kann kürzer sein.) Dann werden die Teillisten zentriert gestapelt.

$f(n)$  wendet zunächst *shelf*( $n+1$ ) auf jede Zeile des Dreiecks  $map(pascal)[0..n]$  an. Da jede Zeile höchstens  $n+1$  Elemente hat, werden also *alle* Zeilenelemente horizontal aneinandergesetzt. *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 f state [a1,a2,a3,a4,a5] [b1,b2,b3,b4,b5]`



```
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
```

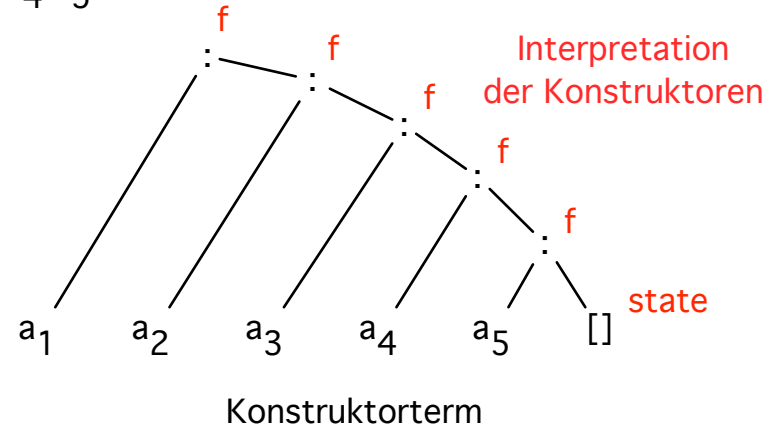
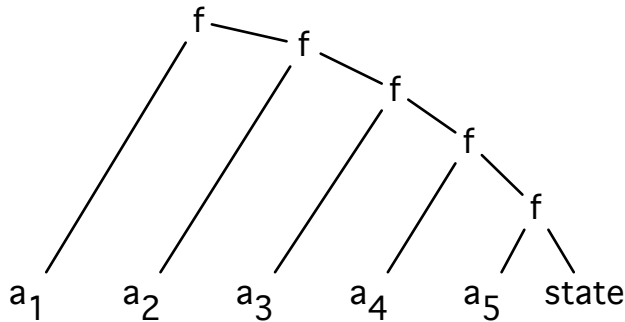
```
listsToFun :: Eq a => b -> [a] -> [b] -> a -> b
listsToFun = fold2 update . const
```

Beginnend mit `const b`, erzeugt `listsToFun b` schrittweise aus einer Argumentliste `as` und einer Werteliste `bs` die entsprechende Funktion:

$$\text{listsToFun } b \text{ as } bs \text{ a} = \begin{cases} bs!!i & \text{falls } i = \max\{k \mid as!!k = a, k < \text{length}(bs)\}, \\ b & \text{sonst.} \end{cases}$$

## Faltung einer Liste von rechts her

foldr f state [a<sub>1</sub>,a<sub>2</sub>,a<sub>3</sub>,a<sub>4</sub>,a<sub>5</sub>]



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

Angewendet auf einen Konstruktorterm

$$t = a_1 : (\cdots : (a_n : [])),$$

liefert die Funktion  $\text{foldr}(f)(st)$  den Wert von  $t$  unter der durch  $st \in \text{state}$  gegebenen Interpretation des Konstruktors  $[] \in [a]$  und der durch  $f : a \rightarrow \text{state} \rightarrow \text{state}$  gegebenen Interpretation des Konstruktors  $(:) : a \rightarrow [a] \rightarrow [a]$ .

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

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

*Die Fakultätsfunktion als Faltung*

## 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]
```

## Teillisten, Permutationen, Partitionen, Linienzüge

Alle Teillisten einer Liste:

```
sublists :: [a] -> [[a]]
sublists []      = [[]]
sublists (a:as) = ass ++ map (a:) ass where
                    ass = 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 bzw. iterativ):

```
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'
```

(*s!!i*.) ist offenbar die einbettende Funktion der rekursiven Variante *perms* (siehe Kapitel 2).

```
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]]]
```

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]

```

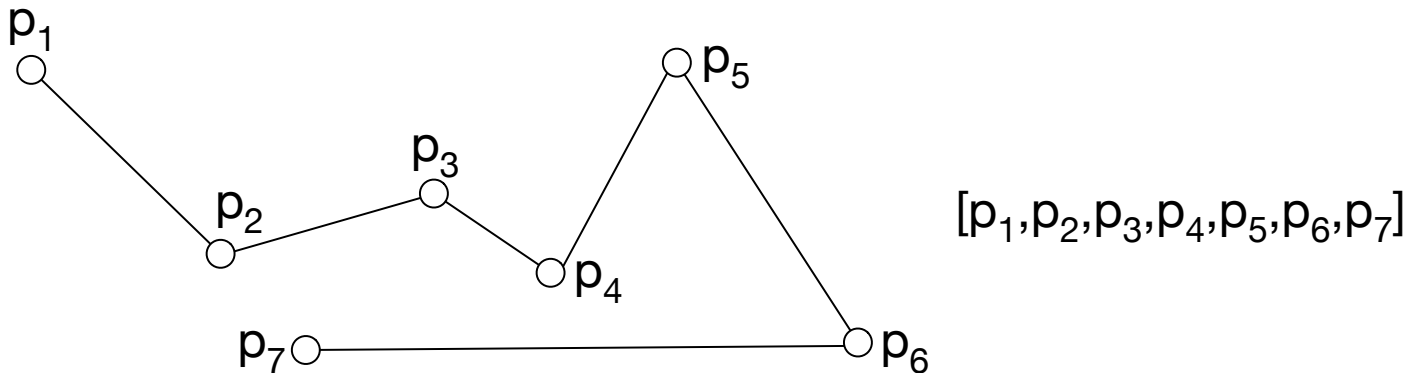
parts[1..4]  $\rightsquigarrow$

```

[[[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]]]

```

Linienzüge als Punktlisten (siehe Kapitel 2)





```
type Path = [Point]
```

```
lengthPath :: Path -> Float
```

*Länge eines Linienzugs*

```
lengthPath ps = sum $ zipWith distance ps $ tail ps
```

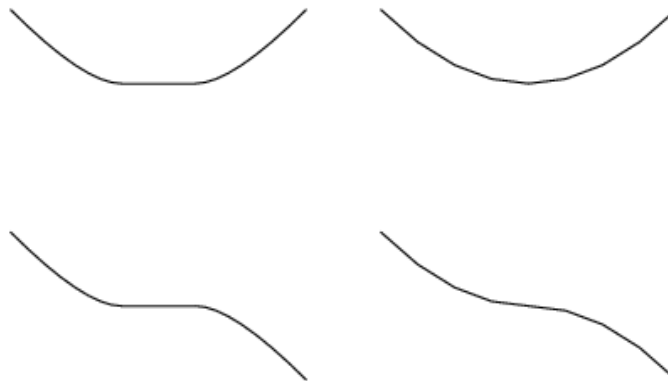
Soll ein Linienzug geglättet werden, dann ist es ratsam, aus ihm vor der Glättung alle Punkte auf Geraden zu entfernen, weil die geglättete Kurve sonst unerwünschte Plateaus enthält:

```
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 Linienzug wurde vor der Minimierung geglättet, der linke nicht:



## Listenlogik

`any :: (a -> Bool) -> [a] -> Bool`  
`any f = or . map f`

`any (>4) [3,2,8,4]  $\rightsquigarrow$  True`

`all :: (a -> Bool) -> [a] -> Bool`  
`all f = and . map f`

`all (>2) [3,2,8,4]  $\rightsquigarrow$  False`

`elem :: Eq a => a -> [a] -> Bool`  
`elem a = any (a ==)`

`elem 2 [3,2,8,4]  $\rightsquigarrow$  True`

`notElem :: Eq a => a -> [a] -> Bool`  
`notElem a = all (a /=)`

`notElem 9 [3,2,8,4]  $\rightsquigarrow$  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]  $\rightsquigarrow$  [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'
```

*map*, *zipWith*, *filter* und *concat* sind auch als **Listenkompansionen** definierbar:

```
map f as          = [f a | a <- as]
zipWith f as bs   = [f a b | (a,b) <- zip as bs]
filter f as       = [a | a <- as, f a]
concat ass        = [a | as <- ass, a <- as]
```

Kartesisches binäres Produkt:

```
prod2 :: [a] -> [b] -> [(a,b)]
prod2 as bs = [(a,b) | a <- as, b <- bs]
```

Allgemeines Schema einer Listenkompansion

$$[e(x_1, \dots, x_n) \mid x_1 \leftarrow s_1, \dots, x_n \leftarrow s_n, be(x_1, \dots, x_n)] \ :: \ [a]$$

- $x_1, \dots, x_n$  sind Variablen,
- $s_1, \dots, s_n$  sind Listen,
- $e(x_1, \dots, x_n)$  ist ein Ausdruck des Typs  $a$ ,

- $x_i \leftarrow s_i$  heißt **Generator** und steht für  $x_i \in s_i$ ,
- $be(x_1, \dots, x_n)$  heißt **Guard** und ist ein Boolescher Ausdruck.

**Beispiel Kryptogramm** = Kodierung von Ziffern durch Zeichen, die sich aus Gleichungen berechnen lässt, hier: *send+more=money*.

Würde die Kodierung aus Permutationen des Wortes *sendmory* anstelle von *sendmory01* berechnet werden, dann blieben die Ziffern 8 und 9 unberücksichtigt. Tatsächlich hat die Gleichung genau eine Lösung (s.u.).

```
codes :: [(Char,Int)]
```

```
codes = [zipWithInds code | code <- perms "sendmory01",
                           let [s,e,n,d,m,o,r,y] = map (getIndex code)
                               1000*(s+m)+100*(e+o)+10*(n+r)+d+e ==
                               10000*m+1000*o+100*n+10*e+y]
```

```
getIndex :: Eq a => [a] -> a -> Int
```

```
getIndex s a = fromJust $ lookup a $ zipWithInds s
```

```
zipWithInds :: [a] -> [(a,Int)]
```

```
zipWithInds s = zip s $ indices s
```

```
take 2 codes ~> [[('o',0),('m',1),('y',2),('1',3),('0',4),('e',5),
                  ('n',6),('d',7),('r',8),('s',9)],
                 [('o',0),('m',1),('y',2),('0',3),('1',4),('e',5),
                  ('n',6),('d',7),('r',8),('s',9)]]
```

□

Jede endlichstellige Relation  $R$  lässt sich als Listenkompensation implementieren. Z.B. entspricht die Menge aller Tripel  $(a, b, c)$  des kartesischen Produkts  $A \times B \times C$ , die ein – als Boolesche Funktion  $p : A \rightarrow B \rightarrow C \rightarrow Bool$  dargestelltes – Prädikat erfüllen, der Kompensation

```
R = [(a,b,c) | a <- as, b <- bs, c <- cs, p a b c]
```

In der Regel ist  $R$  die Schnittmenge mehrerer Relationen, von denen nur wenige die Elemente *aller* Faktoren des jeweiligen Produkts in Beziehung setzen.

## 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.

Alle 14 Bedingungen lassen sich durch Boolesche Funktionen implementieren:

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

```
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

```

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, deren Zeilen Permutationen jeweils einer der folgenden fünf Listen ist:

```

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. Die folgende Komprehension besteht aus denjenigen 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 der obigen Komprehension zunächst nur diejenigen Farbzusordnungen zu den fünf Häusern berechnet, die Bedingung 5 erfüllen. Nur diese Farbzusordnungen werden weitergereicht und mit Zusordnungen anderer Attribute kombiniert, die dann auch verworfen werden, sobald sie einen Guard verletzen. □

Kartesisches Produkt endlich vieler – als Listen dargestellter – Mengen desselben Typs

```
prod :: [[a]] -> [[a]]
```

```
prod [] = [[]]
prod (as:ass) = [a:bs | a <- as, bs <- prod ass]
```

*oder effizienter mit lokaler Definition:*

```
prod (as:ass) = [a:bs | a <- as, bs <- bss] where bss = prod ass
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]]
```

**Unendliche Listen** (Folgen, Ströme) entsprechen Funktionen auf  $\mathbb{N}$

```
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..]

```
addIndex :: [a] -> [(Int,a)]
```

```
addIndex = zip [0..]
```

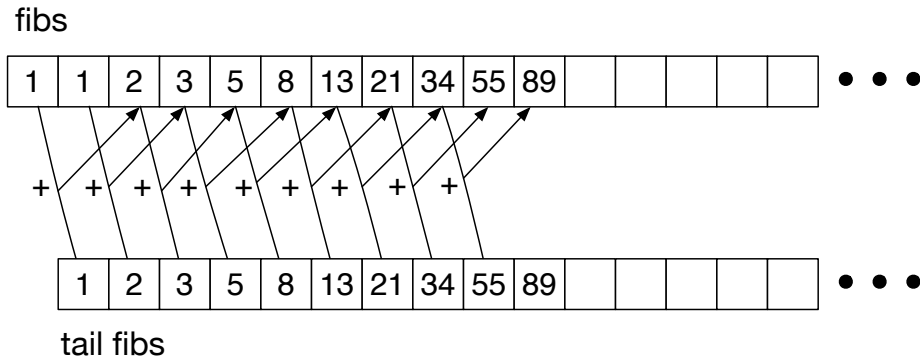
```
addIndex [12,3,55,-2]
```

~> [(0,12),(1,3),(2,55),(3,-2)]

```
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]
```

```
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 primes ~> [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)*

## Definitionen der Funktionsiteration von Kapitel 2 mit *iterate*

```
hoch :: (a -> a) -> Int -> a -> a
f`hoch`n = \x -> iterate f x!!n
```

*oder als Funktionskomposition:*

```
f`hoch`n = (!!n) . iterate f
```

## 4 Rekursive Datentypen

Zunächst das allgemeine Schema einer Datentypdefinition:

$$\text{data } DT \ a_1 \ \dots \ a_m = C_1 \ \text{typ}_{11} \ \dots \ \text{typ}_{1n_1} \mid \dots \mid \\ C_k \ \text{typ}_{k1} \ \dots \ \text{typ}_{kn_k}$$

$\text{typ}_{11}, \dots, \text{typ}_{kn_k}$  sind beliebige Typen, die außer  $a_1, \dots, a_m$  keine Typvariablen enthalten.

$DT$  heißt **rekursiv**, wenn  $DT$  in mindestens einem dieser Typen vorkommt.

Nicht-rekursive Datentypen entsprechen den bereits in Kapitel 2 behandelten Summentypen.

Die durch  $DT$  bezeichnete Menge besteht aus allen Ausdrücken der Form

$$C_i \ e_1 \ \dots \ e_{n_i},$$

wobei  $1 \leq i \leq n$  und für alle  $1 \leq j \leq n_i$   $e_j$  ein Element des Typs  $\text{typ}_{ij}$  ist. Als Funktion hat  $C_i$  den Typ

$$\text{typ}_{i1} \rightarrow \dots \rightarrow \text{typ}_{in_i} \rightarrow DT \ a_1 \ \dots \ a_m.$$

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 Kapitel 2).

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

$[A]$  ist die größte Lösung der Gleichung

$$M = \{[]\} \cup \{a : s \mid a \in A, s \in M\} \quad (1)$$

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  $0 : 1 : 0 : 1 : 0 : 1 : \dots$  die eindeutige Lösung der Gleichung

$$\textit{blink} = 0 : 1 : \textit{blink} \quad (2)$$

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  $a_1 : a_2 : a_3 : \dots$  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

$$\text{data Nat} = \text{Zero} \mid \text{Succ Nat} \quad (3)$$

$$\text{data PosNat} = \text{One} \mid \text{Succ' PosNat} \quad (4)$$

$$\text{data Int' = Zero' \mid Plus PosNat \mid Minus PosNat} \quad (5)$$

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 Kapitel 2). Er realisiert demnach die disjunktive Vereinigung seiner drei Argumenttypen:

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

$A \cong B$  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 in obiger Definition von

$C_i \text{ typ}_{i1} \dots \text{typ}_{in_i}$

erweitert zu:

$C_i \{\textcolor{red}{d}_{i1} :: \text{typ}_{i1}, \dots, \textcolor{red}{d}_{in_i} :: \text{typ}_{in_i}\}$

Wie  $C_i$ , so ist auch  $\textcolor{red}{d}_{ij}$  eine Funktion. Als solche hat sie den Typ

$DT \ a1 \dots am \rightarrow \text{typ}_{ij}$

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:

$\text{obj} = \text{C\_i } e_{i1} \dots e_{in\_i}$  *ist äquivalent zu*  
 $\text{obj} = \text{C\_i } \{d_{i1} = e_{i1}, \dots, d_{in\_i} = e_{in\_i}\}$

Die Werte einzelner Destruktoren von  $\text{obj}$  können wie folgt verändert werden:

$\text{obj}' = \text{obj } \{d_{ij\_1} = e_1, \dots, d_{ij\_m} = e_m\}$

$\text{obj}'$  unterscheidet sich von  $\text{obj}$  dadurch, dass den Destruktoren  $d_{ij_1}, \dots, d_{ij_m}$  neue Werte, nämlich  $e_1, \dots, e_m$  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  $\text{obj}$  (s.o.) die Gleichung  $d_{ij} = e_j$  durch  $d_{ij} = d_{ij}$  ersetzt und die neue Funktion auf der rechten Seite lokal definiert:

$\text{obj} = \text{C\_i } \{d_{i1} = e_1, \dots, d_{ij} = d_{ij}, \dots, d_{in\_i} = e_{n\_i}\}$   
 $\text{where } d_{ij} \dots = \dots d_{ij} \dots$

Ein Konstruktor darf nicht zu mehreren Datentypen gehören.

Ein Destruktor darf nicht zu mehreren Konstruktoren unterschiedlicher Datentypen gehören.

## Listen mit Destrukturen

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

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

```
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 Destrukturen erst aus der jeweiligen Datentypdefinition erschließen, sollte man Datentypen mit Destrukturen und *mehreren* Konstruktoren grundsätzlich vermeiden. Wie das folgende Beispiel nahelegt, machen sie das Datentypkonzept auch nicht ausdrucksstärker.

## 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

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

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 Elementen 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

$$M = \{a :< s \mid a \in A, s \in M\} \quad (7)$$

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  $\infty$  als eindeutige Lösung solcher Gleichungen darstellen:

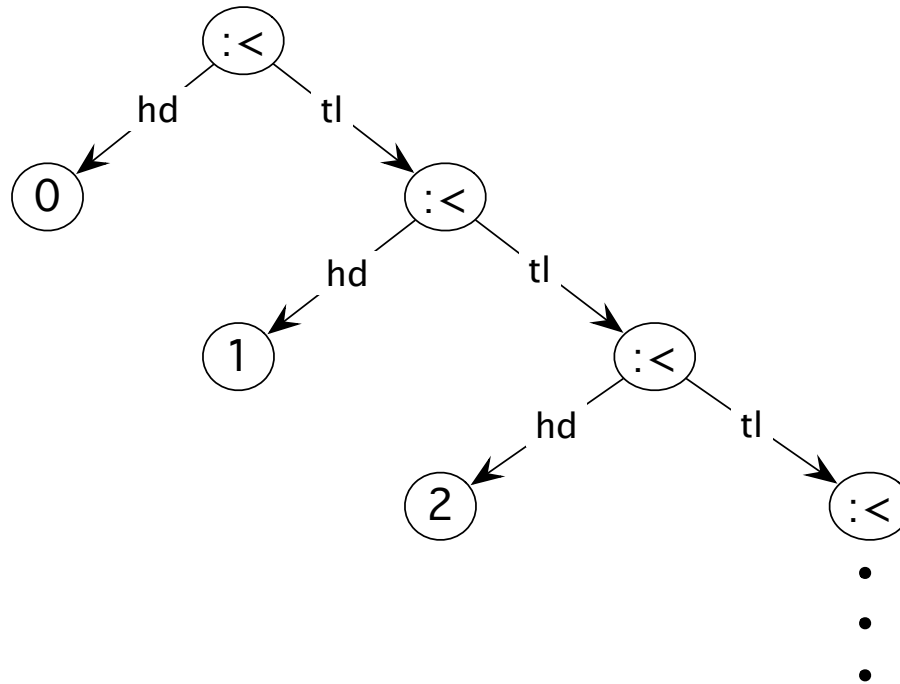
```
infinity :: Nat
infinity = Succ infinity

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

Die oben geforderte Vermeidung von Datentypen mit mindestens einem Destruktor, aber mehr als einem Konstruktor garantiert, dass alle Destrukturen 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 [Pad2], Kapitel 2 und 12).

Zurück zu Datentypen mit mehreren Konstruktoren, aber ohne Destruktoren.

## Arithmetische Ausdrücke (Zugehöriger Haskell-Modul: Expr.hs))

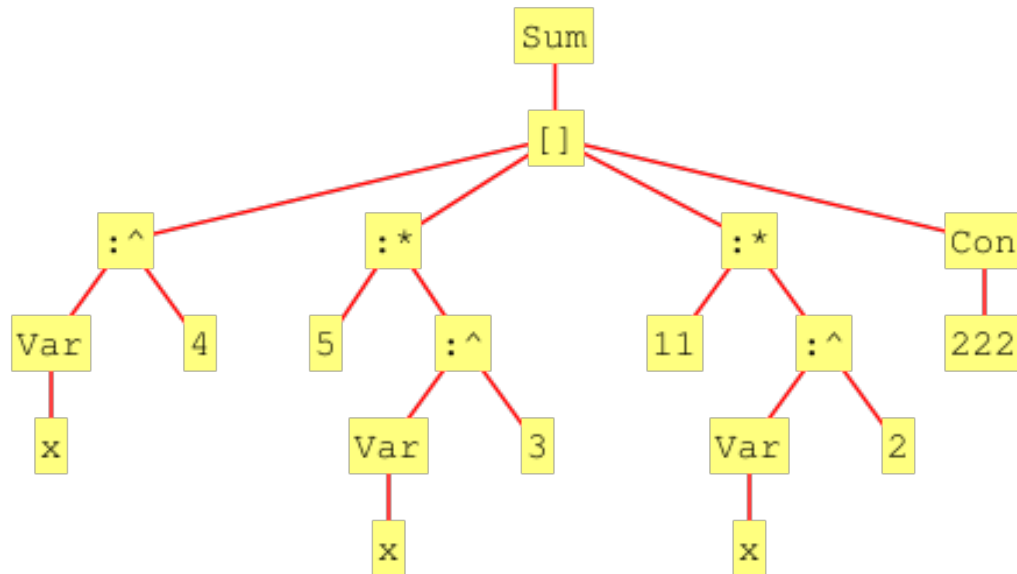
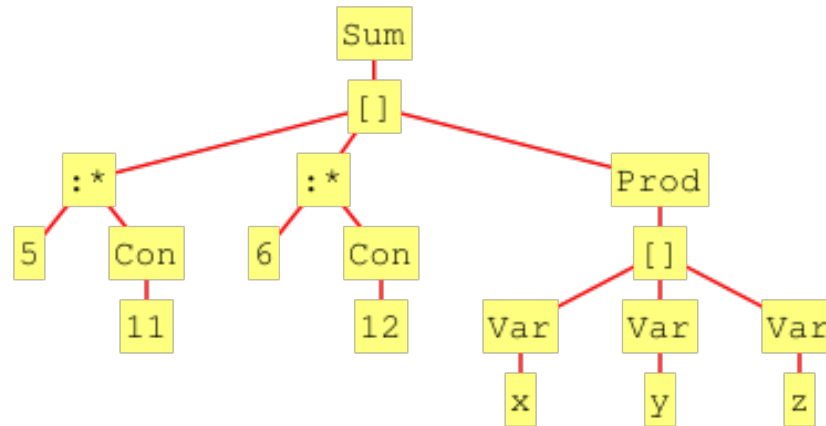
```
data Exp x = Con Int | Var x | Sum [Exp x] | Prod [Exp x] |  
            Exp x :- Exp x | Int :* Exp x | Exp x :^ Int
```

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

```
data Exp x where Con    :: Int -> Exp x  
                  Var    :: x -> Exp x  
                  Sum    :: [Exp x] -> Exp x  
                  Prod   :: [Exp x] -> Exp x  
                  (:-)   :: Exp x -> Exp x -> Exp x  
                  (:*)   :: Int -> Exp x -> Exp x  
                  (:^)   :: Exp x -> Int -> Exp x
```

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]
```



## 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_   :: BExp x  
                  False_  :: BExp x  
                  BVar    :: x -> BExp x  
                  Or      :: [BExp x] -> BExp x  
                  And     :: [BExp x] -> BExp x  
                  Not     :: BExp x -> BExp x  
                  (:=)    :: Exp x -> Exp x -> BExp x  
                  (:<=)   :: Exp x -> Exp x -> BExp x
```

Ein GADT erlaubt unterschiedliche Instanzen der Typvariablen des Datentyps in dessen Definition und damit die Zusammenfassung mehrerer Datentypen zu einem einzigen.

## Arithmetische, Boolesche, bedingte, Paar- und Listenausdrücke

```
data GExp x a where Con      :: Int -> GExp x Int
                      Var      :: x -> GExp x a
                      Sum       :: [GExp x Int] -> GExp x Int
                      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_     :: GExp x Bool
                      False_    :: GExp x Bool
                      Or        :: [GExp x Bool] -> GExp x Bool
                      And       :: [GExp x Bool] -> GExp x Bool
                      Not       :: GExp x Bool -> GExp x Bool
                      (:=)      :: GExp x Int -> GExp x Int -> 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.

Wie die folgenden Beispiele zeigen, bestimmt das Rekursionsschema der Definition eines Datentyps DT das Rekursionsschema der Definition einer Funktion auf DT. Häufig enthält die Definition für jeden Konstruktor eine eigene Gleichung.

## Arithmetische Ausdrücke auswerten

`type Store x = x -> Int`      (*Belegung der Variablen*)

`exp2store :: Exp x -> Store x -> Int`

```
exp2store (Con i) _      = i
exp2store (Var x) st     = st x
exp2store (Sum es) st    = sum $ map (flip exp2store st) es
exp2store (Prod es) st   = product $ map (flip exp2store st) es
exp2store (e :- e') st   = exp2store e st - exp2store e' st
exp2store (i :* e) st    = i * exp2store e st
exp2store (e :^ i) st    = exp2store e st ^ i
```

*oder mit case-Konstrukt:*

```


exp2store e st = case e of Con i    -> i
                          Var x    -> st x
                          Sum es   -> sum $ map eval es
                          Prod es  -> product $ map eval es
                          e :- e'  -> eval e - eval e'
                          i :* e   -> i * eval e
                          e :^ i   -> eval e ^ i
      where eval = flip exp2store st

```

```
exp1 :: Exp String
```

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

[Link](#) zur schrittweisen Auswertung des Ausdrucks

```
exp2store exp1 $ \"x\" -> 4   974
```

(1)

Die Ausdrücke "x" und `Sum[e1,...,en]` sind dort durch **X** bzw. **e1:+...:+en** wiedergegeben.

## Symbolische Differentiation

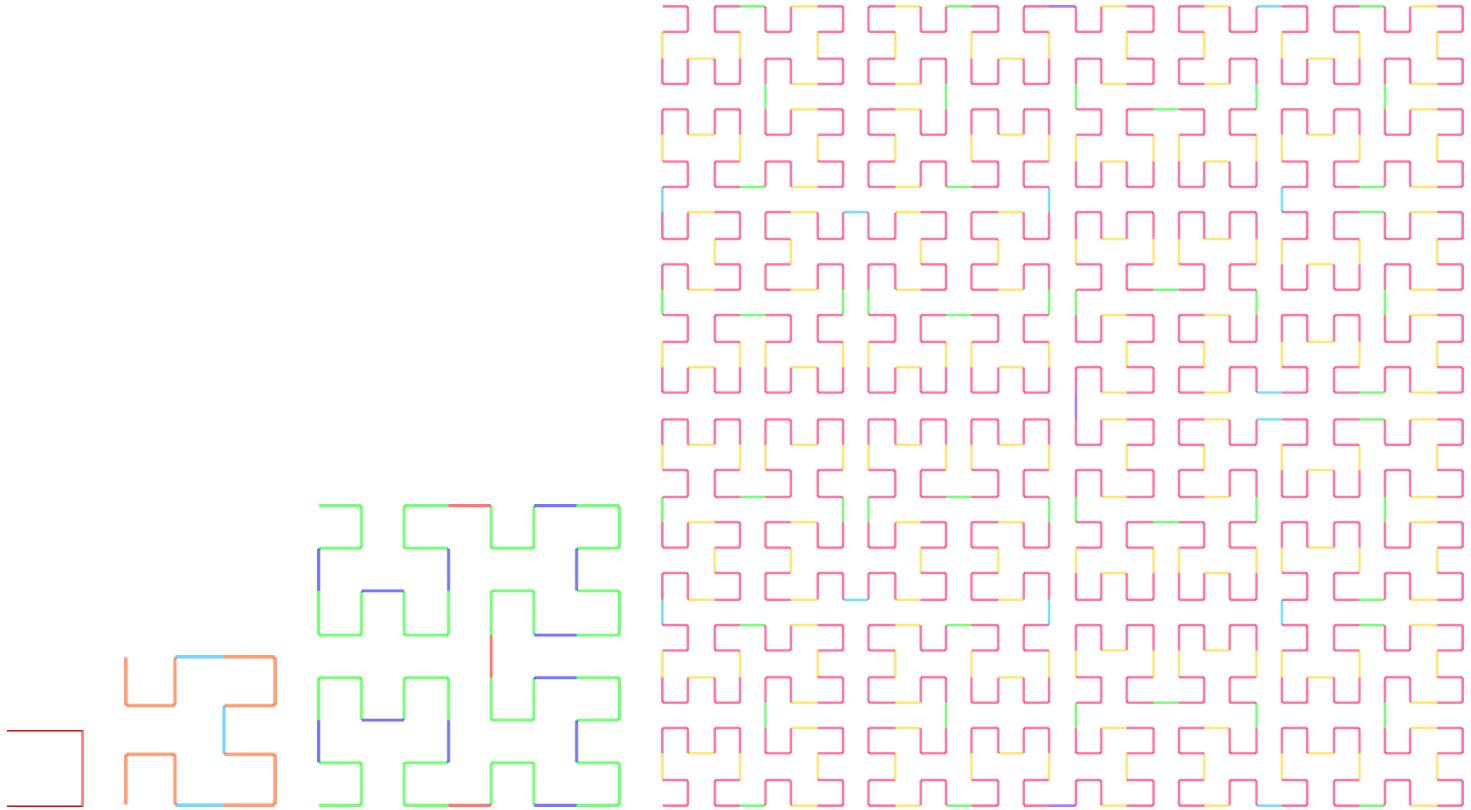
```
diff :: Eq x => Exp x -> x -> Exp x
diff e x = case e of Con _    -> zero
                  Var y      -> if x == y then one else zero
                  Sum es     -> Sum $ map d es
                  Prod es    -> Sum $ zipWith f [0..] es
                              where f i = Prod . updList es i . d
                  e :- e'    -> d e :- d e'
                  i :* e     -> i :* d e
                  e ^ i      -> i :* Prod [d e, e:^(i-1)]
      where d = flip diff x
zero = Con 0
one  = Con 1
```

[Link](#) zur schrittweisen Auswertung von  $\text{diff}(\text{exp1})(x)$ . Hier werden neben den Gleichungen von  $\text{diff}$  zur Vereinfachung der Zwischenergebnisse weitere Gleichungen wie z.B.  $e + 0 = e$ ,  $e * 1 = e$  und  $3 * 5 * e = 15 * e$  angewendet.



## 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. Man erkennt auf gleicher Rekursionstiefe erzeugte Punkte daran, dass sie mit Linien gleicher Farbe verbunden sind.*

Solche Linienzüge lassen sich nicht nur als Punktlisten (s.o.), sondern auch als Listen von Aktionen repräsentieren, die auszuführen sind, um einen Linienzug 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]
```

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

Die Hilbertkurve der Tiefe  $n$  wird – abhängig von einer Anfangsrichtung **dir** – aus vier Hilbertkurven der Tiefe  $n - 1$  zusammengesetzt, die durch die rot markierten Aktionsfolgen miteinander verbunden werden:

```

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 eine Punktliste vom Typ `Path` überführt werden:

```

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)

```

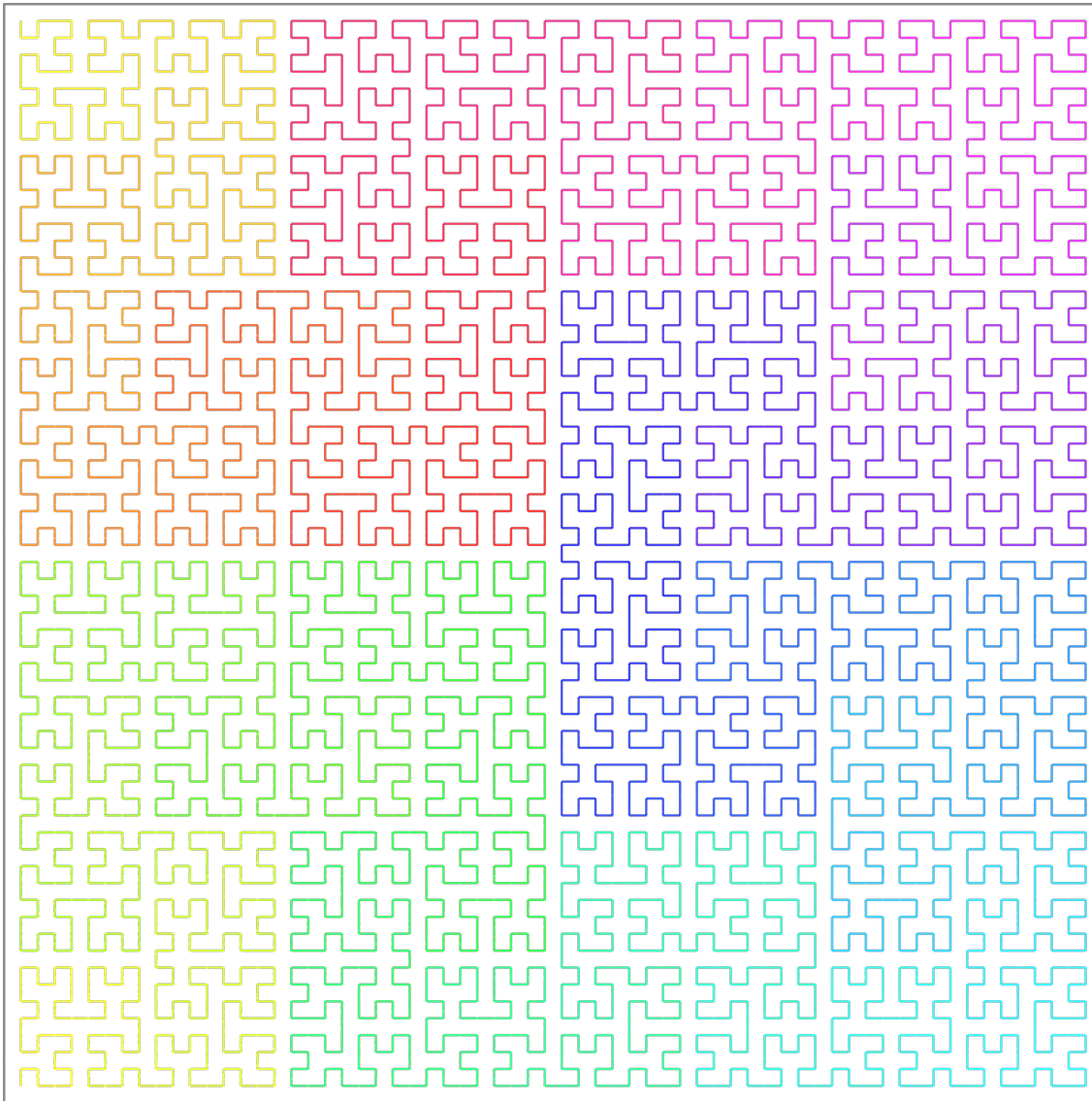
*rotate* wurde in Kapitel 2 definiert.

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:





*Anwendung von `addColors` auf die Hilbertkurve der Tiefe 5*

## 5 Typklassen und Bäume

stellen Bedingungen an die Instanzen einer Typvariablen. Die Bedingungen bestehen in der Existenz bestimmter Funktionen, z.B.

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

Eine **Instanz einer Typklasse** besteht aus den Instanzen ihrer Typvariablen sowie Definitionen der in ihr deklarierten 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')
```

Auch (/=) könnte hier definiert werden. Die Definitionen von (/=) und (==) in der Typklasse **Eq** sind Defaults. Definitionen einer Typklasse können in deren Instanzen durch neue Definitionen überschrieben werden. Jede Instanz von **Eq** muss aber offenbar eine Definition von (==) oder (/=) enthalten.

Der Typ jeder Funktion einer Typklasse muss deren Typvariable enthalten.

## 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]
union = foldl $ flip insert
```

*Mengenvereinigung*

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

*concatMap für Mengen*

```
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]
```

*Mengendifferenz*

```
diff = foldl $ flip remove
```

```
subset :: Eq a => [a] -> [a] -> Bool
```

*Mengeninklusion*

```
s `subset` s' = all (`elem` s') s
```

```
eqset :: Eq a => [a] -> [a] -> Bool
```

*Mengengleichheit*

```
s `eqset` s' = s `subset` s' && s' `subset` s
```

```
powerset :: Eq a => [a] -> [[a]]
```

*Potenzmenge*

```
powerset (a:s) = if a `elem` s then ps else ps ++ map (a:) ps  
                where ps = powerset s
```

```
powerset _ = [[]]
```

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
```



## 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
```

## Beispiel 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
```

## Binäre Bäume

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

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

## Binäre Bäume ausbalancieren

```
baltree :: [a] -> Bintree a
baltree [] = Empty
baltree s  = Fork 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@(Fork b t1 t2) | a == b = t
                              | a < b  = Fork b (insertTree a t1) t2
                              | True   = Fork b t1 (insertTree a t2)
insertTree a _ = leaf a
```

## Binäre Bäume ordnen

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

## 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 mit ausgezeichnetem Knoten*

```
data Context a =
```

```
  Top |
```

*leerer Kontext*

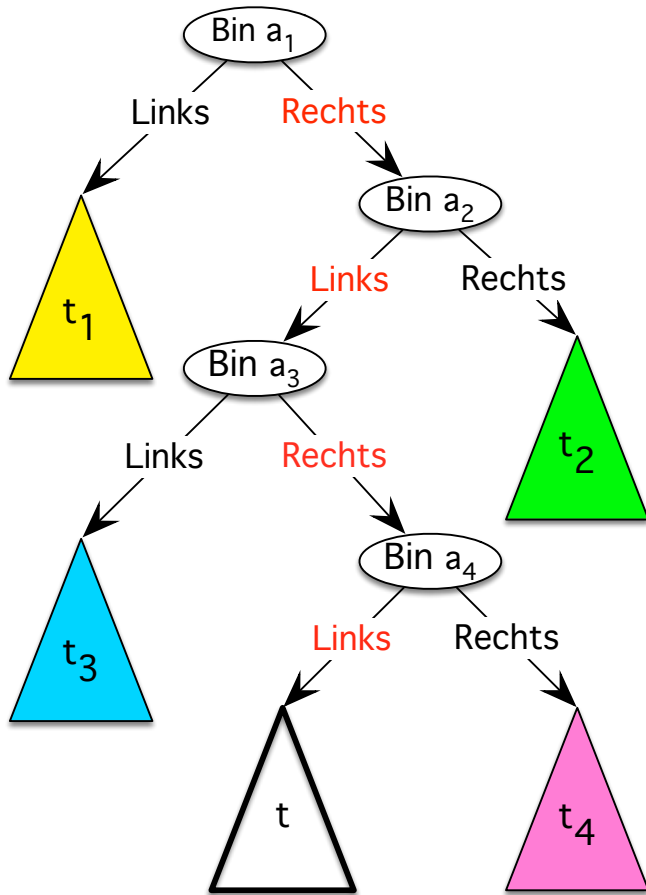
```
  L a (Context a) (BintreeL a) |
```

*Kontext eines linken Teilbaums*

```
  R a (BintreeL a) (Context a)
```

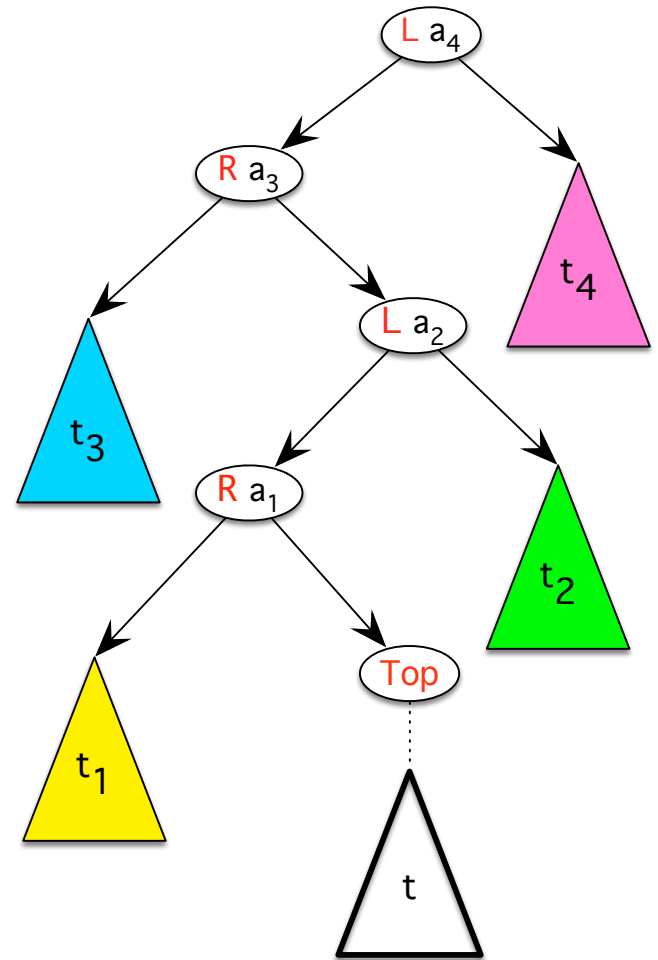
*Kontext eines rechten Teilbaums*

```
type TreeZipper a = (Context a, BintreeL a)
```



treeToZipper  
----->

<-----  
zipperToTree



```

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:

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

`up,sibling,left,right :: TreeZipper a -> TreeZipper a`

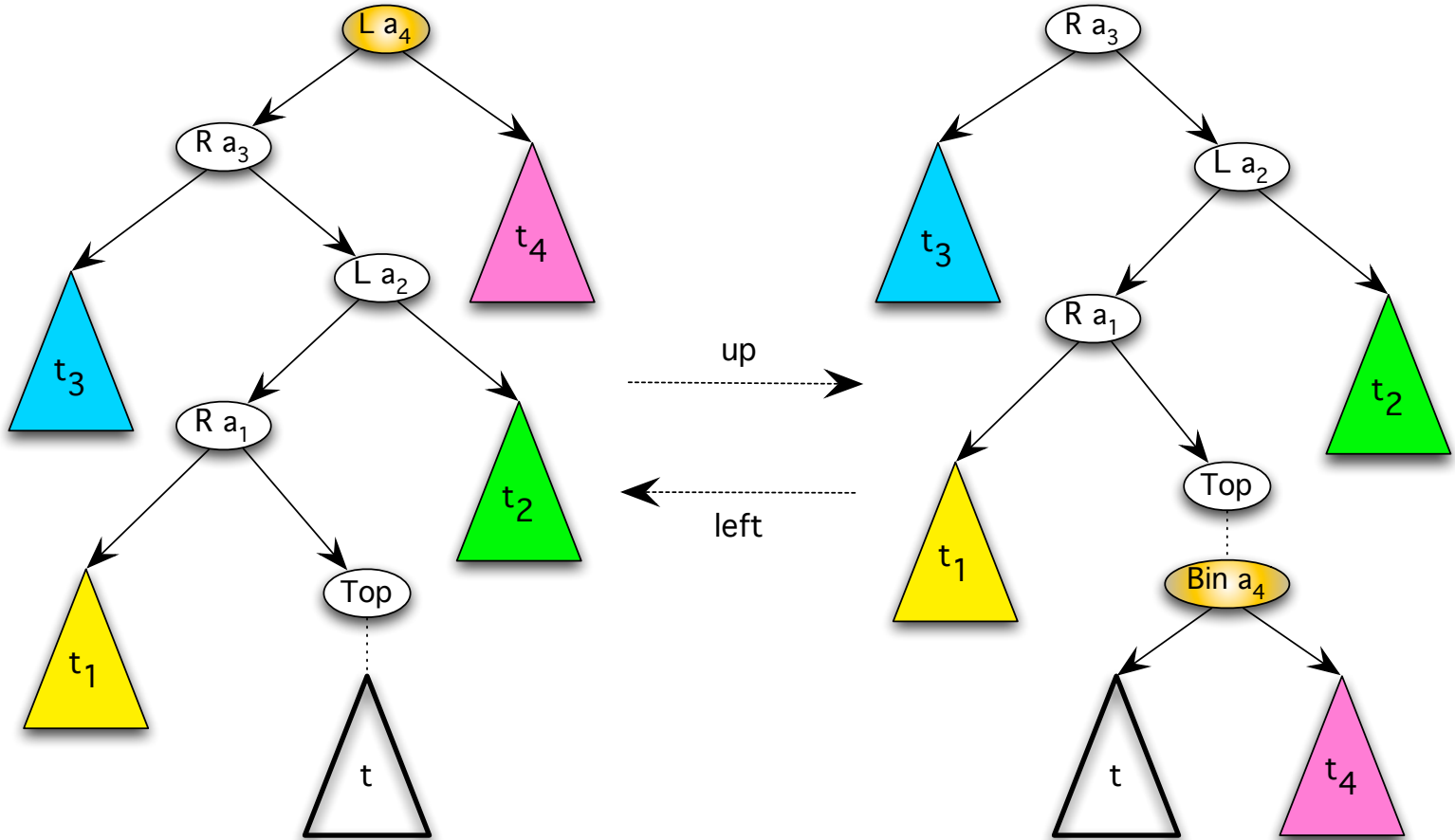
*Zeiger bewegen*

`up (L a c u,t) = (c,Bin a t u)`

`up (R a t c,u) = (c,Bin a t u)`

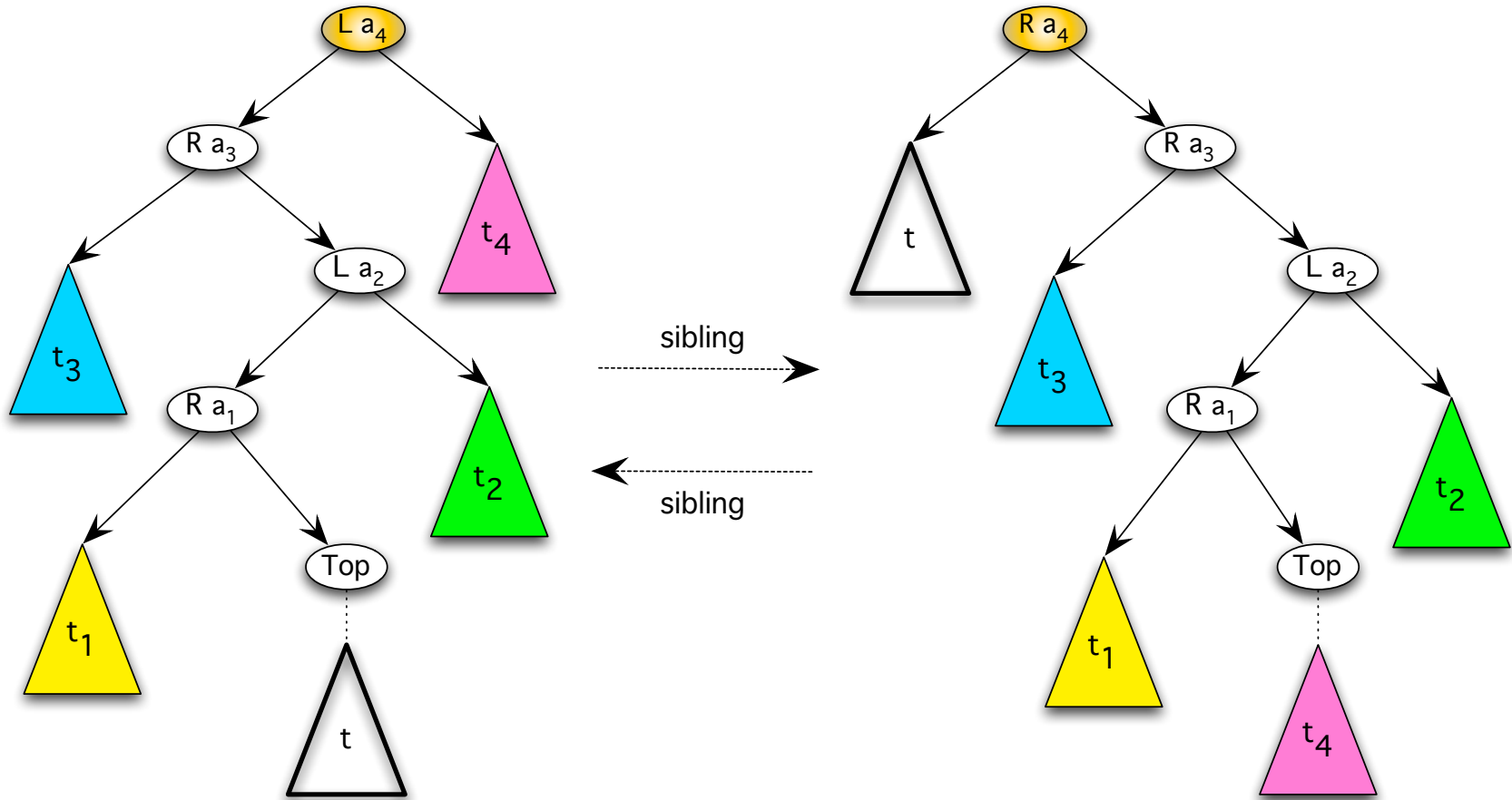
`left (c,Bin a t u) = (L a c u,t)`

`right (c,Bin a t u) = (R a t c,u)`





$\text{sibling}(\text{L a c u}, t) = (\text{R a t c}, u)$   
 $\text{sibling}(\text{R a t c}, u) = (\text{L a c u}, t)$



## 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, wobei auf der rechten Seite definierender Gleichungen anstelle von `showsPrec 0` vorkommen darf.

## Binäre Bäume ausgeben

```
instance Show a => Show (Bintree a) where
  showsPrec _ Empty                = id
  showsPrec _ (Fork a Empty Empty) = shows a
  showsPrec _ (Fork 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 . ('')':)
```

## Arithmetische Ausdrücke ausgeben (siehe auch [hier](#))

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.

$t$  sollte auch dann geklammert werden, wenn  $op$  mit  $op'$  übereinstimmt und nicht assoziativ ist, wie z.B. im Ausdruck  $x - (y - z)$ . Dieser würde sonst nämlich als  $x - y - z$  ausgegeben werden, was man wiederum als  $(x - y) - z$  interpretieren würde.

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 (Exp String) where
  showsPrec _ (Con i)      = shows i
  showsPrec _ (Var x)      = (x++)
  showsPrec p (Sum es)     = enclose (p > 0) $ showMore 0 '+' es
  showsPrec p (Prod es)    = enclose (p > 1) $ showMore 1 '*' es
  showsPrec p (e :- e')    = enclose (p > 0) $ showsPrec 0 e . ('-':) .
                                showsPrec 1 e'
  showsPrec p (i :* e)     = enclose (p > 1) $ shows i . ('*':) .
                                showsPrec 1 e
  showsPrec p (e :^ i)     = enclose (p > 2) $ showsPrec 2 e . ('^':) .
                                shows i
```

```
enclose :: Bool -> (String -> String) -> String -> String
enclose b f = if b then '('(:) . f . (')':) else f
```

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

## Beispiele

```
show $ Sum [5:*Con 11,6:*Con 12,Prod[x,y,z]]  $\rightsquigarrow$  5*11+6*12+x*y*z
```

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

## 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 absplattet 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 entsprechende `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 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
```

Für alle, die damit etwas anfangen können: Der Erkennung von Klammerstrukturen als Bäume des Typs **BintreeL(a)** liegt eine **kontextfreie Grammatik** mit runden Klammern und Kommas als Terminalsymbolen sowie folgenden Regeln zugrunde:

$$\textit{bintree} \rightarrow a$$

$$\textit{bintree} \rightarrow a(\textit{bintree}, \textit{bintree})$$

## 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). Variablen können durch Bäume ersetzt werden, Konstanten nicht (siehe Abschnitt 7.9).

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

```
subtrees :: Tree a -> [Tree a]
```

```
subtrees (F _ ts) = ts
```

```
subtrees t      = []
```

```
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) = sum (map size ts)+1
```

```
size _        = 1
```

```
height (F _ ts) = 1+foldl1 max 0 (map height ts)
```

```
height _       = 1
```

```
size tree1     ~> 9
```

```
height tree1   ~> 4
```

```
type Node = [Int]
```

```

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

```

$getSubtree(t)(node)$  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]  $\rightsquigarrow$  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]
 $\rightsquigarrow$  F 1 [F 2 [F 2 [V 3,F 4 [V (-3),V 5]],V (-2)],F 4 [V (-3),V 5]]
```

$\text{mapTree}(f)(t)$  wendet die Funktion  $h : a \rightarrow b$  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 mit Destruktoren

Wie die Datentypen für Listen (siehe Kapitel 4), so enthalten auch Datentypen für Bäume unendliche Objekte, was in diesem Fall bedeutet, dass sie unendliche Pfade besitzen können. Folglich kann z.B. für Bäume mit beliebigem Ausgrad alternativ zu **Tree(a)** ein **Colist(a)** entsprechender und zu **Tree(a)** semantisch äquivalenter Datentyp mit totalen Destruktoren verwendet werden:

```
data Cotree a      = Cotree {select :: Either a (a,Cotreelist a)}  
data Cotreelist a = Cotreelist {split :: Maybe (Cotree a,  
                                                Cotreelist a)}
```

Diese Lösung enthält zwei – mit *Either* bzw. *Maybe* gebildete – Summentypen. Ebenfalls semantisch äquivalent zu **Tree(a)** wäre auch die folgende Lösung mit *select* als einzigem Destruktor und den zwei Konstruktoren des Standardtyps für Listen:

```
data Cotree a = Cotree {select :: Either a [(a,Cotreelist a)]}
```

Die Dualität von Datentypen mit Konstruktoren einerseits und Destruktoren andererseits wird ausführlich in den Lehrveranstaltungen **Einführung in den logisch-algebraischen Systementwurf** und **Übersetzerbau** behandelt.

## Baumfaltungen

Analog zu *foldr* (siehe Abschnitt 3.8) induziert jeder Haskell-Datentyp  $DT$  mit Konstruktoren  $C_1, \dots, C_n$  eine Funktion

$$\text{fold}DT : typ_1 \rightarrow \dots \rightarrow typ_n \rightarrow DT \rightarrow val,$$

zur Faltung (= Auswertung) der Ausdrücke, aus denen  $DT$  besteht, zu Elementen der Menge  $val$ . Für alle  $1 \leq i \leq n$  ist hier  $typ_i$  der Typ der Operation auf  $val$ , die den Konstruktor  $C_i$  bei der Auswertung von  $DT$ -Ausdrücken interpretieren soll.

Z.B. haben die Datentyp  $Bintree(a)$ ,  $Tree(a)$  bzw.  $[Tree(a)]$  die Konstruktoren

$$Empty : Bintree(a), \quad Fork : a \rightarrow Bintree(a) \rightarrow Bintree(a) \rightarrow Bintree(a),$$

$$V : a \rightarrow Tree(a), \quad F : a \rightarrow [Tree(a)] \rightarrow Tree(a),$$

$$[] : [Tree(a)], \quad (:) : Tree(a) \rightarrow [Tree(a)] \rightarrow [Tree(a)].$$

Die Faltungsfunktionen für diese Datentypen lauten daher wie folgt:

```
foldBtree :: val -> (a -> val -> val -> val) -> Bintree a -> val
foldBtree val _ Empty                = val
foldBtree val f (Fork a left right) = f a (foldBtree val f left)
                                     (foldBtree val f right)
```



```

foldTree :: (a -> val) -> (a -> valL -> val) -> valL
              -> (val -> valL -> valL) -> Tree a -> val

foldTree f _ _ _ (V a)      = f a
foldTree f g nil h (F a ts) = g a $ foldTrees f g nil h ts

foldTrees :: (a -> val) -> (a -> valL -> val) -> valL
              -> (val -> valL -> valL) -> [Tree a] -> valL

foldTrees _ _ nil _ []      = nil
foldTrees f g nil h (t:ts) = h (foldTree f g nil h t)
                              (foldTrees f g nil h ts)

```

Offenbar erzwingt die wechselseitige Rekursion der Definitionen von  $Tree(a)$  und  $[Tree(a)]$ , dass ihren Faltungsfunktionen die Interpretationen der insgesamt vier Konstruktoren beider Datentypen übergeben werden müssen.

Die Menge  $\Sigma$  der Konstruktoren eines oder mehrerer Haskell-Datentypen bezeichnet man als **Signatur** und die Elemente der Datentypen von  $\Sigma$  als  $\Sigma$ -**Terme**. Eine  $\Sigma$ -**Algebra** ist die Zuordnung einer Menge  $val$  zu jedem Datentyp und einer Operation auf  $val$  zu jedem Konstruktor von  $\Sigma$ . Demnach wertet  $foldDT$   $\Sigma$ -Terme in der  $\Sigma$ -Algebra aus, die ihr als Parameter übergeben wird. Wie die obigen Beispiele zeigen, sind Termfaltungen stets nach dem gleichen Schema definiert. Die Zusammenfassung der Funktionsparameter von  $foldDT$  zu einer Algebra macht die Definition von  $foldDT$  übersichtlicher (siehe Abschnitt 9.3).

Umgekehrt kann jede induktiv auf  $\Sigma$ -Termen definierte Funktion als Faltung in einer passenden  $\Sigma$ -Algebra formuliert werden.

## Beispiele

```
sum_ :: Num a => Tree a -> a
sum_ = foldTree id (+) 0 (+)
```

```
preorder,postorder :: Tree a -> [a]
preorder  = foldTree (\a -> [a]) (:) [] (++)
postorder = foldTree (\a -> [a]) (\a s -> s++[a]) [] (++)
```

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

```
sum_ tree1      ~> 11
preorder tree1  ~> [1,2,2,3,-1,-2,4,-3,5]
postorder tree1 ~> [3,-1,2,-2,2,-3,5,4,1]
```

```
var :: String -> Int
var = \case "x" -> const 5; "y" -> const $ -66; "z" -> const 13
```

```

fun :: String -> [Int] -> Int
fun = \case "+" -> sum; "*" -> product

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

foldTree var fun [] (:) tree2  ~>  -317

```

Jedes Element eines beliebigen Datentyps kann in einen Baum vom Typ *Tree(String)* übersetzt werden. Z.B. transformiert die folgende Funktion *exp2tree* Ausdrücke vom Typ *Exp(String)* (siehe Abschnitt 4.1) in Bäume vom Typ *Tree(String)*:

```

exp2tree :: Exp String -> Tree String
exp2tree = \case Con i    -> int i
              Var x      -> V x
              Sum es     -> F "Sum" $ map exp2tree es
              Prod es    -> F "Prod" $ map exp2tree es
              e :- e'    -> F "-" [exp2tree e, exp2tree e']
              i :* e     -> F "*" [int i, exp2tree e]
              e :^ i     -> F "^" [exp2tree e, int i]
      where int i = F (show i) []

```

Die **String**-Instanz der in Abschnitt 4.3 definierte Auswertungsfunktion *exp2store* für arithmetische Ausdrücke entspricht einer Faltung der mit *exp2tree* aus den Ausdrücken gebildeten Bäume:

```
exp2store :: Exp String -> Store String -> Int
exp2store e st = foldTree st fun [] (:) $ exp2tree e

fun :: String -> [Int] -> Int
fun = \case "Sum" -> sum; "Prod" -> product; "-" -> \[i,k] -> i-k
        "*" -> \[i,k] -> i*k; "^" -> \[i,k] -> i^k
        i -> const $ read i
```

In Abschnitt 9.3 werden wir *exp2store* ohne den Umweg über Bäume vom Typ *Tree(String)* als Faltung in einer passenden Interpretation der Konstruktoren von *Exp(String)* darstellen.

## Arithmetische Ausdrücke kompilieren

Die unten definierte Funktion **exp2code** übersetzt Objekte des Datentyps **Expr** in 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 | 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 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
```

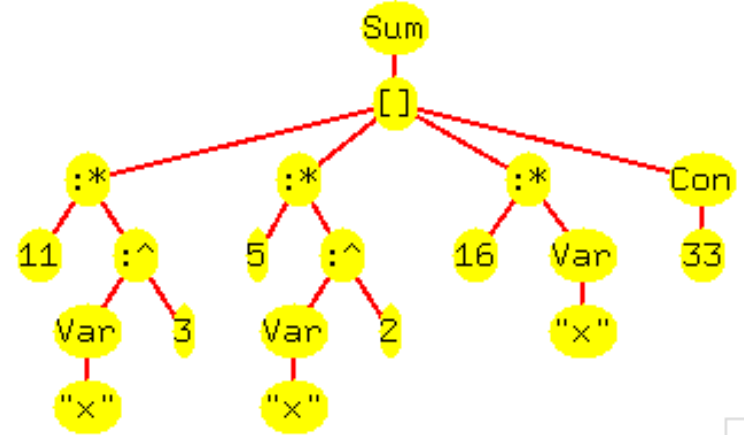
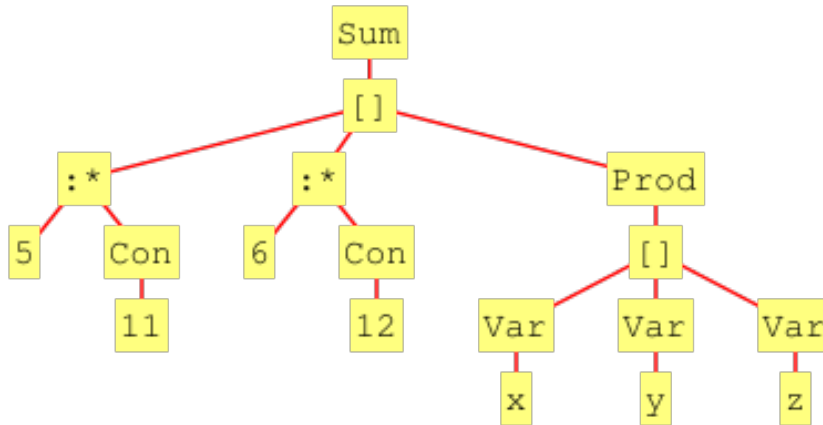
Die Übersetzung eines arithmetischen Ausdrucks von seiner Baumdarstellung in eine Befehlsliste erfolgt wie die Definition aller Funktionen auf *Expr*-Objekten induktiv:

```
exp2code :: Exp x -> [StackCom x]
exp2code = \case Con i    -> [Push i]
              Var x      -> [Load x]
              Sum es     -> concatMap exp2code es++[Add $ length es]
              Prod es    -> concatMap exp2code es++[Mul $ length es]
              e :- e'    -> exp2code e++exp2code e'++[Sub]
              i :* e     -> Push i:exp2code e++[Mul 2]
              e :^ i     -> exp2code e++[Push i,Up]
```

Wie die Korrektheit der Reduktion von Ausdrücken  $e$ , so ist auch die Korrektheit der Übersetzung von  $e$  durch eine Gleichung gegeben, welche die Beziehung zur Interpretation von  $e$  herstellt und durch Induktion über den Aufbau von  $e$  gezeigt werden kann:

$$\text{execute}(\text{exp2code}(e))(\text{stack}, \text{store}) = (\text{exp2store}(e)(\text{store}) : \text{stack}, \text{store}).$$

Beginnt die Ausführung des Zielcodes von  $e$  im Zustand  $(\text{stack}, \text{store})$ , dann endet sie im Zustand  $(a : \text{stack}, \text{store})$ , wobei  $a$  der Wert von  $e$  unter der Variablenbelegung  $\text{store}$  ist.



Z.B. übersetzt *exp2code* die oben als *Exp(String)*-Objekte dargestellten Wörter  
 $5*11+6*12+x*y*z$  bzw.  $11*x^3+5*x^2+16*x+33$   
 in folgende Kommandosequenzen:

0: Push 5	8: Load "z"
1: Push 11	9: Mul 3
2: Mul 2	10: Add 3
3: Push 6	
4: Push 12	
5: Mul 2	
6: Load "x"	
7: Load "y"	

0: Push 11	8: Up
1: Load "x"	9: Mul 2
2: Push 3	10: Push 16
3: Up	11: Load "x"
4: Mul 2	12: Mul 2
5: Push 5	13: Push 33
6: Load "x"	14: Add 4
7: Push 2	



## Arithmetische Ausdrücke reduzieren

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*[ $e_1, \dots, e_n$ ] oder *Prod*[ $e_1, \dots, e_n$ ] 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 $ Sum [e,(-1):*e']
          i :* Con j -> Con $ i*j
          0 :* e -> zero
          1 :* e -> reduceE e
          i :* (j :* e) -> (i*j) :* reduceE e
          i :* e -> i :* 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) e =
        \case Con 1 -> state
              Con i -> (c*i,bases,expo)
              e:^i -> updState state e i
              e -> updState state e 1

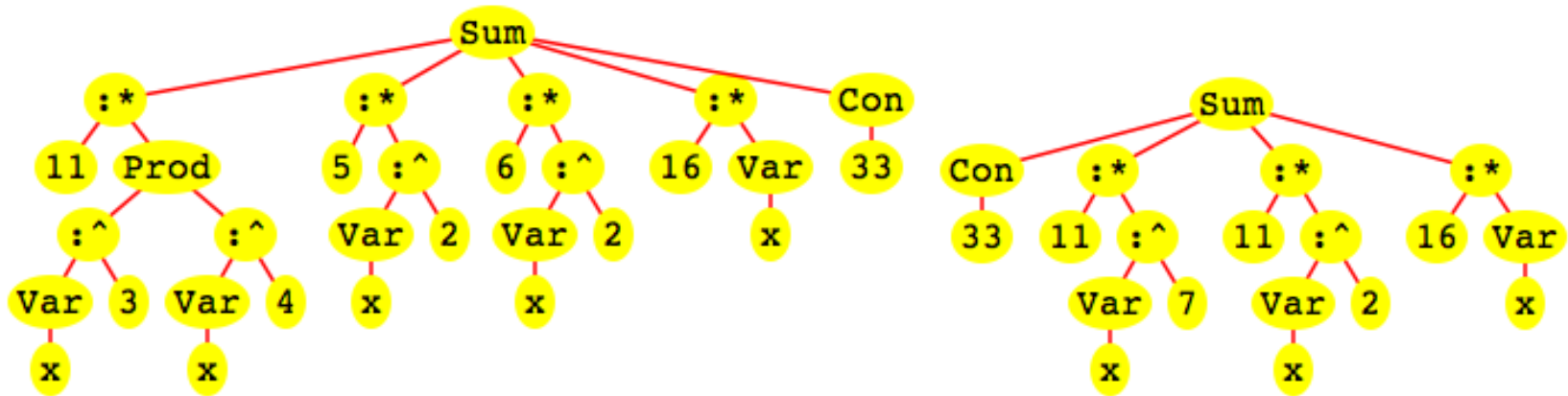
```

*reduceE*(*Sum*(*es*)) wendet *reduceE* zunächst auf alle Ausdrücke der Liste *es* an. Dann wird die Ergebnisliste *res* = *map*(*reduceE*)(*es*), ausgehend vom Anfangszustand  $(0, [], \text{const}(0))$  mit der Zustandsüberführung *trans* zum Endzustand  $(c, \text{bases}, \text{scal})$  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  $scal : Exp(x) \rightarrow Int$  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

$$\textit{exp2store}(\textit{reduceE}(e)) = \textit{exp2store}(e).$$

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

## 6 Fixpunkte, Graphen und Modallogik

**CPOs und Fixpunkte** (Der Haskell-Code steht [hier](#).)

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  $\perp$  (*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  $\bigcap A$  und  $\bigcup A$  ein kleinstes bzw. größtes Element besitzt.

## Beispiele

$Bool$  ist ein vollständiger Verband mit Halbordnung  $\{(b, c) \in Bool \mid b = False \vee c = True\}$  kleinstem Element  $False$ , größtem Element  $True$ , der Disjunktion als Supremumsbildung und der Konjunktion als Infimumsbildung.

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.  $\square$

Eine Funktion  $\Phi : A \rightarrow B$  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  $\Phi : A \rightarrow B$  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  $\Phi$  ist **monoton**, d.h. für alle  $a \in A$  gilt:

$$a \leq b \Rightarrow \Phi(a) \leq \Phi(b). \quad \square$$

$a \in A$  heißt **Fixpunkt von  $\Phi : A \rightarrow A$** , falls  $\Phi(a) = a$  gilt.

## Fixpunktsatz von Kleene

Sei  $\Phi : A \rightarrow A$  stetig.  $\text{lfp}(\Phi) =_{\text{def}} \bigsqcup_{i \in \mathbb{N}} \Phi^i(\perp)$  ist der (bzgl.  $\leq$ ) kleinste Fixpunkt von  $\Phi$ .

Sei  $\Phi : A \rightarrow A$  co-stetig.  $\text{gfp}(\Phi) =_{\text{def}} \bigsqcap_{i \in \mathbb{N}} \Phi^i(\top)$  ist der (bzgl.  $\leq$ ) größte Fixpunkt von  $\Phi$ .  $\square$

Aus der Monotonie von  $\Phi$  folgt  $\Phi^i(\perp) \leq \Phi^{i+1}(\perp)$  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  $\Phi^i(\perp) = \Phi^{i+1}(\perp) = \text{lfp}(\Phi)$  und  $\Phi^k(\top) = \Phi^{k+1}(\top) = \text{gfp}(\Phi)$ . Also können in diesem Fall der kleinste wie auch der größte Fixpunkt von  $\Phi$  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
```

$\Phi$  wird auch **Schrittfunktion** der Fixpunktberechnung genannt.

## Semantik rekursiver Funktionsgleichungen

Mit Hilfe des Fixpunktsatzes von Kleene kann gezeigt werden, dass eine Rekursionsgleichung wie

$$\text{fact } n = \text{if } n > 1 \text{ then } n * \text{fact } (n-1) \text{ else } 1 \quad (1)$$

tatsächlich eine Funktion  $f$  definiert, genauer gesagt: dass es eine Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  gibt, die die Gleichung in der Funktionsvariablen  $\text{fact}$  löst. Um den Fixpunktsatz anzuwenden, muss die Funktionsmenge  $\mathbb{N} \rightarrow \mathbb{N}$  zu einem CPO erweitert werden. Man beginnt mit der Erweiterung von  $\mathbb{N}$  zum **flachen CPO**  $\mathbb{N}_\perp =_{\text{def}} \mathbb{N} \cup \{\perp\}$ , dessen Halbordnung wie folgt definiert ist: Für alle  $a, b \in \mathbb{N}_\perp$ ,

$$a \leq b \iff_{\text{def}} a = \perp \vee a = b.$$

Dann wird diese Halbordnung folgendermaßen auf die Funktionsmenge  $\mathbb{N} \rightarrow \mathbb{N}_\perp$  fortgesetzt:

Für alle  $f, g : \mathbb{N} \rightarrow \mathbb{N}_\perp$ ,

$$f \leq g \iff_{\text{def}} \forall n \in \mathbb{N} : f(n) \leq g(n).$$

Damit wird  $\mathbb{N} \rightarrow \mathbb{N}_\perp$  zum CPO: Eine Kette  $F \subseteq (\mathbb{N} \rightarrow \mathbb{N}_\perp)$  hat folgendes Supremum:

$$\begin{aligned} \bigsqcup F : \mathbb{N} &\rightarrow \mathbb{N}_\perp \\ n &\mapsto \begin{cases} f(n) & \text{falls } f \in F \text{ mit } f(n) \neq \perp \text{ existiert,} \\ \perp & \text{sonst.} \end{cases} \end{aligned}$$

$\bigsqcup F$  ist wohldefiniert, weil für alle  $f, g \in F$  mit  $f \leq g$  oder  $g \leq f$  gilt, also insbesondere  $f(n) \leq g(n)$  oder  $g(n) \leq f(n)$  und daher  $f(n) = g(n)$  im Fall  $f(n) \neq \perp \neq g(n)$ .

Das kleinste Element von  $\mathbb{N} \rightarrow \mathbb{N}_\perp$  ist die mit  $\Omega$  bezeichnete Funktion, die allen natürlichen Zahlen  $\perp$  zuordnet.

Gleichung (1) liefert folgende Schrittfunktion:

$$\begin{aligned} \Phi : (\mathbb{N} \rightarrow \mathbb{N}_\perp) &\rightarrow (\mathbb{N} \rightarrow \mathbb{N}_\perp) \\ f &\mapsto \lambda n. \text{if } n > 1 \text{ then } n * f(n-1) \text{ else } 1 \end{aligned} \quad (2)$$

$\Phi$  ist stetig, wenn man die Multiplikation zur *strikten* Funktion auf  $\mathbb{N}_\perp$  erweitert, d.h.  $n * \perp$  und  $\perp * n$  auf  $\perp$  setzt. Allgemein ist die Schrittfunktion immer dann stetig, wenn die in der zugrundeliegenden Rekursionsgleichung verwendeten Hilfsfunktionen monoton sind (siehe P. Padawitz, [Formale Methoden des Systementwurfs](#), Satz 10.1.9).

Nach dem Fixpunktsatz von Kleene hat  $\Phi$  also den kleinsten Fixpunkt

$$lfp(\Phi) = \bigsqcup_{i \in \mathbb{N}} \Phi^i(\Omega). \quad (3)$$

M.a.W.:  $lfp(\Phi)$  ist die kleinste Lösung von Gleichung (1) in der Funktionsvariablen **fact**.

Intuitiv gesprochen, beschreibt (3) die Lösung von (1) als Grenzwert der Folge wiederholter Anwendungen von (1), die Haskell zur Berechnung der Werte von **fact** durchführt.

$lfp(\Phi)(n) = \perp$  würde den Fall wiedergeben, dass die Berechnung von  $lfp(\Phi)(n)$  nicht terminiert, dass also  $lfp(\Phi)$  an der Stelle  $n$  nicht definiert ist. Bei der oben definierten Schrittfunktion tritt dieser Fall nicht auf: Für alle  $n \in \mathbb{N}$  gilt  $lfp(\Phi)(n) \in \mathbb{N}$ .

*Beweis durch Induktion über  $n$ :*

Für alle  $n \in \{0, 1\}$  gilt  $lfp(\Phi)(n) = \Phi(lfp(\Phi))(n) \stackrel{(2)}{=} 1$ .

Für alle  $n > 1$  gilt  $lfp(\Phi)(n) = \Phi(lfp(\Phi))(n) \stackrel{(2)}{=} n * lfp(\Phi)(n - 1) \in \mathbb{N}$ , weil nach Induktionsvoraussetzung  $lfp(\Phi)(n - 1)$  eine natürliche Zahl ist.  $\square$

Die oben definierte Halbordnung auf  $\mathbb{N}_\perp$  impliziert daher, dass  $lfp(\Phi)$  nicht nur der kleinste, sondern der einzige Fixpunkt von  $\Phi$  ist, dass also die Lösung von (1) in **fact** eindeutig ist. Das erst rechtfertigt die Bezeichnung von (1) als *Definition* von **fact**.

## Semiringe

Als algebraische Strukturen bilden Graphen (und ihre Implementierungen als Matrizen; siehe Abschnitt 8.4) *Semiringe*:

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
$a * (b + c) = (a * b) + (a * c)$	Links distributivität von $*$ über $+$
$(a + b) * c = (a * c) + (b * c)$	Rechts distributivität 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](#); 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:

$$a^+ = a + a * a + a * a * a + \dots, \quad a^* = 1 + a^+.$$

In Haskell implementieren wir Semiringe als Instanzen der folgenden Typklasse:

```
class Semiring r where add,mul :: r -> r -> r
                        zero,one :: r
```

Einige Instanzen von Semiring:

```
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 => Semiring (BinRel a) where
  add = union
  mul rel rel' = [(a,c) | (a,b) <- rel, (b',c) <- rel',
                        b == b']

  zero = []
  one = ?
```

```
type BRfun a = a -> [a]
```

```
instance Eq a => Semiring (BRfun a) where
  add sucs sucs' = liftM2 union sucs sucs'
  mul sucs sucs' = unionMap sucs' . sucs
  zero = const []
  one  = single
```

Die Funktion

$$\text{liftM2} :: (b \rightarrow c \rightarrow d) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow a \rightarrow d$$

liftet jede Operation  $\text{op} :: b \rightarrow c \rightarrow d$  zu einer Operation auf Funktionen und ist wie folgt definiert:

```
liftM2 op f g a = op (f a) $ g a
```

Es handelt sich hierbei um die Instanz einer generischen Standardfunktion für Monaden, die in Kapitel 7 behandelt wird.

Mit `liftM2 subset` als Halbordnung, `add` als Supremumsbildung und `zero` als kleinstem Element bildet `BRfun a` auch einen CPO.

Abschlussoperator von `BinRel a`

Sei  $R \subseteq A^2$ .

$$R^+ =_{def} lfp(\Phi), \text{ wobei } \begin{cases} \Phi : \mathcal{P}(A^2) \rightarrow \mathcal{P}(A^2) \\ R' \mapsto R + (R * R') \end{cases}$$

Haskell-Implementierung:

```
plus :: Eq a => BinRel a -> BinRel a
plus rel = fixpt subset (add rel . mul rel) []
```

## Graphen (Der Haskell-Code steht [hier](#).)

Im Folgenden stehen die Typvariablen `a` und `label` für eine Knotenmenge bzw. eine Menge von Kantenmarkierungen.

```
type TRfun a label = a -> [(label,a)]
```

Unmarkierte Graphen: `data Graph a = G [a] (BRfun a)`

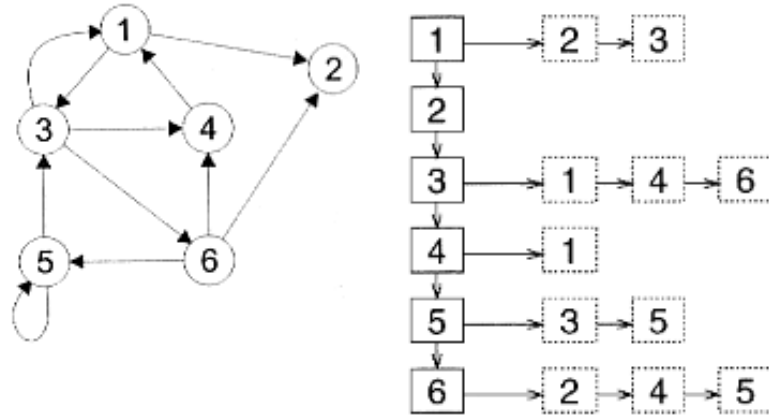
Kantenmarkierte Graphen: `data GraphL a label = GL [a] (TRfun a label)`

Das erste Argument von `G` und `GL` ist eine Liste aller Knoten des Graphen, das zweite Argument eine Funktion, die jedem Knoten die Liste seiner Nachfolgerknoten zuordnet – bei kantenmarkierten Graphen zusammen mit der Markierung der jeweils einlaufenden Kante.

## Beispiele

```
graph1 :: Graph Int
graph1 = G [1..6] $ \case 1 -> [2,3]
                           2 -> []
                           3 -> [1,4,6]
                           4 -> [1]
                           5 -> [3,5]
                           6 -> [2,4,5]
```





```
graph2,graph3,graph4 :: Graph Int
```

```
graph2 = G [1..6] $ \a -> if a `elem` [1..5] then [a+1] else []
```

```
graph3 = G [1..6] [a+1..6]
```

```
graph4 = G [1,11,12,111,112,121,122,1121,1122] $
  \a -> if a `elem` [1,11,12,112] then [a*10+1,a*10+2] else []
```

## Show-Instanz von Graph a

```
instance Show a => Show (Graph a) where
```

```
  show (G nodes sucs) = concatMap f $ filter (not . null . sucs) nodes
    where f a = '\n':show a++" -> "++show (sucs a)
```

## Transformation der Funktions- in die Relationsdarstellung und umgekehrt

```
graph2Rel :: Graph a -> BinRel a
```

```
graph2Rel (G nodes sucs) = [(a,b) | a <- nodes, b <- sucs a]
```

```
rel2Graph :: Eq a => BinRel a -> Graph a
```

```
rel2Graph rel = G nodes sucs where
```

```
    (nodes,sucs) = foldl f ([], const []) rel
```

```
    f (as,g) (a,b) = (insert a as, update g a $ insert b $ g a)
```

## Abschlussoperatoren von `Graph a`

Die drei folgenden Funktionen berechnen den **transitiven Abschluss** eines Graphen  $g$ , das ist die Erweiterung von  $g$  um alle Kanten  $(a,b)$ , für die in  $g$  ein Weg, also eine *Kantenfolge*, von  $a$  nach  $b$  existiert.

```
closureF,closureT,warshall :: Eq a => Graph a -> Graph a
```

```
closureF (G nodes sucs) = G nodes sucs' where
```

```
    sucs' = fixpt le (mul sucs . add one) zero where
```

```
    le sucs sucs' = all (liftM2 subset sucs sucs') nodes
```

```

closureT (G nodes sucs) = G nodes sucs' where
    sucs' = mul sucs $ add one sucs'

warshall (G nodes sucs) = G nodes sucs' where
    sucs' = foldl trans sucs nodes
    trans sucs a = fold2 update sucs nodes $ map f nodes where
        f b = if a `elem` cs then cs `union` sucs a
              else cs
              where cs = sucs b

```

**closureT** terminiert nur für azyklische Graphen.

Ausgehend von  $sucs_1 = sucs$  berechnet **warshall** eine Folge  $(sucs_1, \dots, sucs_n)$  binärer Relationen, wobei  $sucs_{i+1}$  aus  $sucs_i$  entsteht, indem für alle Knotentripel  $(a, b, c)$  mit  $a \in sucs_i(b)$  und  $c \in sucs_i(a)$   $c$  zu  $sucs_i(b)$  hinzugefügt wird.

**warshall** berechnet den transitiven Abschluss mit Aufwand  $O(n^3)$ : Die äußere Faltung  $foldl(trans)(sucs)(nodes)$  durchläuft die Liste  $nodes$ , die innere Faltung  $trans(sucs)(a)$  durchläuft die Liste  $map(f)(nodes)$ . Jeder Aufruf  $f(b)$  erzeugt die Faltung  $cs \cup sucs(a)$ , die im schlechtesten Fall ( $sucs(a) = nodes$ ) ein weiteres Mal die Liste  $nodes$  durchläuft.

## Beispiele

closureF/W graph1  $\rightsquigarrow$

- 1  $\rightarrow$  [1,2,3,4,5,6]
- 3  $\rightarrow$  [1,2,3,4,5,6]
- 4  $\rightarrow$  [1,2,3,4,5,6]
- 5  $\rightarrow$  [1,2,3,4,5,6]
- 6  $\rightarrow$  [1,2,3,4,5,6]

closureF/T/W graph2/3  $\rightsquigarrow$

- 1  $\rightarrow$  [2,3,4,5,6]
- 2  $\rightarrow$  [3,4,5,6]
- 3  $\rightarrow$  [4,5,6]
- 4  $\rightarrow$  [5,6]
- 5  $\rightarrow$  [6]

closureF/T/W graph4  $\rightsquigarrow$

- 1  $\rightarrow$  [11,12,111,112,1121,1122,121,122]
- 11  $\rightarrow$  [111,112,1121,1122]
- 12  $\rightarrow$  [121,122]
- 112  $\rightarrow$  [1121,1122]

## 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  $\varphi$  so wie einen anderen Ausdruck auswerten zu können, weist man ihr folgende – vom üblichen Gültigkeitsbegriff abweichende, aber dazu äquivalente – Semantik zu:  $\varphi$  wird interpretiert als die Menge aller Zustände von  $\mathcal{K}$ , die  $\varphi$  erfüllen sollen.

Zu diesem Zweck definieren eine **Kripke-Struktur**  $\mathcal{K}$  als Quadrupel

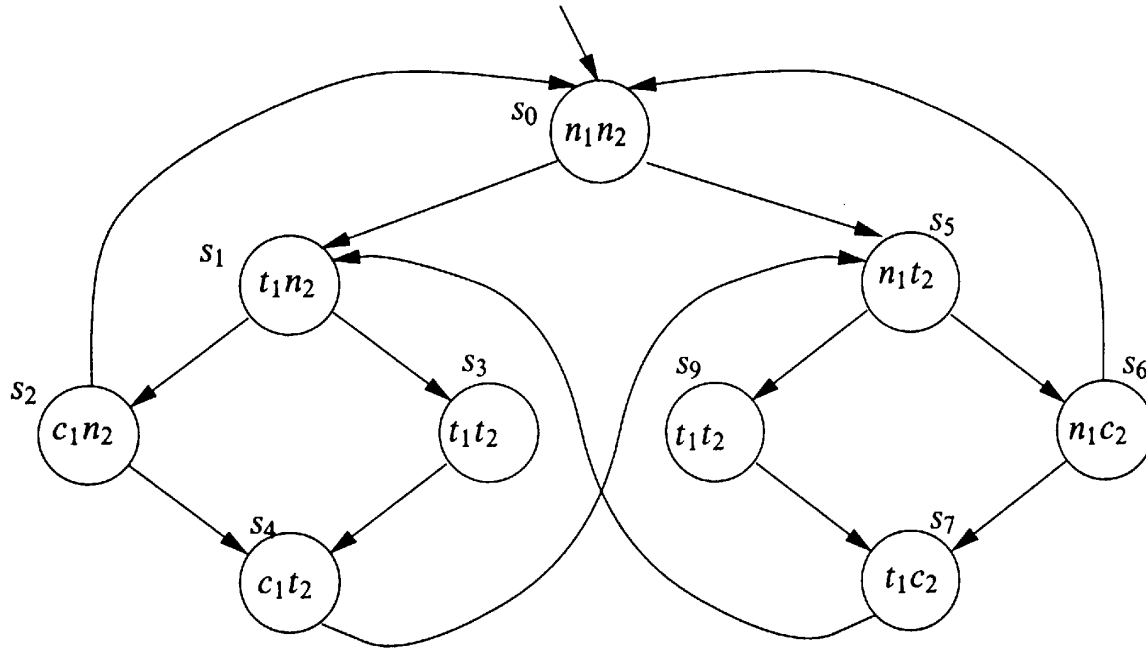
$$(\textit{State}, \textit{Atom}, \textit{trans}, \textit{atoms}),$$

bestehend aus einer **Zustandsmenge**  $\textit{State}$ , einer Menge  $\textit{Atom}$  **atomarer Formeln**, einer **Transitionsfunktion**  $\textit{trans} : \textit{State} \rightarrow \mathcal{P}(\textit{State})$ , die jedem Zustand von  $\textit{State}$  die Menge seiner möglichen Nachfolger zuordnet, und einer Funktion

$$\textit{atoms} : \textit{State} \rightarrow \mathcal{P}(\textit{Atom}),$$

die jeden Zustand auf die Menge seiner atomaren Eigenschaften abbildet.

## Beispiel Mutual exclusion (Huth, Ryan, Logic in Computer Science, 2nd ed., Example 3.3.1)



Die Kanten und Knotenmarkierungen des Graphen definieren die Funktionen *trans* bzw. *atoms* der Kripkestruktur

$$Mutex = (\{s_0, \dots, s_7, s_9\}, \{n_1, n_2, t_1, t_2, c_1, c_2\}, trans, atoms).$$

**Bedeutung der atomaren Formeln:** Sei  $i = 1, 2$ .  $n_i$ : Prozess  $i$  befindet sich ausserhalb des kritischen Abschnitts und hat nicht um Einlass gebeten.  $t_i$ : Prozess  $i$  bittet um Einlass in den kritischen Abschnitt.  $c_i$ : Prozess  $i$  befindet sich im kritischen Abschnitt.

Unter den zahlreichen **Modallogiken** wählen wir hier **CTL** (computation tree logic) und den – alle Modallogiken umfassenden –  **$\mu$ -Kalkül** (siehe auch **Algebraic Model Checking**). Deren Formelmeng  **$MF$**  ist induktiv definiert:

Sei  $V$  eine Menge von Variablen.

$$\begin{aligned} \{True, False\} \cup Atom \cup V &\subseteq MF, \\ \varphi, \psi \in MF &\Rightarrow \neg\varphi, \varphi \wedge \psi, \varphi \vee \psi, EX\varphi, AX\varphi \in MF, \\ x \in V \wedge \varphi \in MF &\Rightarrow \mu x.\varphi, \nu x.\varphi \in MF. \quad (\mu\text{-Formeln}) \end{aligned}$$

Alle anderen CTL-Formeln sind spezielle  $\mu$ -Formeln:

$EF\varphi$	$= \mu x.(\varphi \vee EX x)$	<i>exists finally</i>
$AF\varphi$	$= \mu x.(\varphi \vee (EX True \wedge AX x))$	<i>always finally</i>
$AG\varphi$	$= \nu x.(\varphi \wedge AX x)$	<i>always generally</i>
$EG\varphi$	$= \nu x.(\varphi \wedge (AX False \vee EX x))$	<i>exists generally</i>
$\varphi EU\psi$	$= \mu x.(\psi \vee (\varphi \wedge EX x))$	<i>exists <math>\varphi</math> until <math>\psi</math></i>
$\varphi AU\psi$	$= \mu x.(\psi \vee (\varphi \wedge AX x))$	<i>always <math>\varphi</math> until <math>\psi</math></i>
$\varphi \Rightarrow \psi$	$= \neg\varphi \vee \psi$	

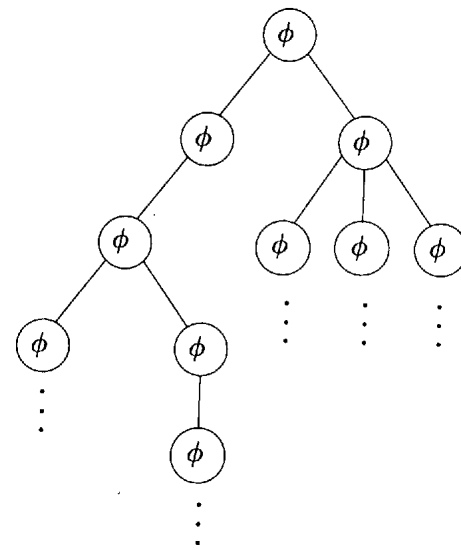


Fig. 3.7. A system whose starting state satisfies  $\text{AG } \phi$ .

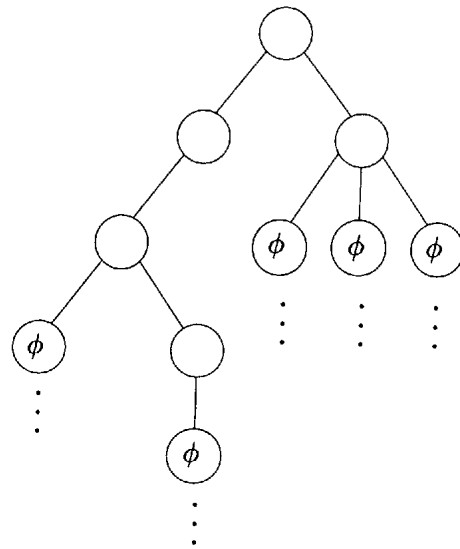


Fig. 3.8. A system whose starting state satisfies  $\text{AF } \phi$ .



Wie bei arithmetischen oder Booleschen Ausdrücken hängt die Auswertung modaler Formeln von einer Variablenbelegung ab, das ist hier eine Funktion des Typs

$$\textit{Store} = V \rightarrow \mathcal{P}(\textit{State}).$$

Für eine gegebene Kripke-Struktur  $\mathcal{K} = (\textit{State}, \textit{Atom}, \textit{trans}, \textit{value})$  ist die Auswertungsfunktion

$$\textit{eval} : \textit{MF} \rightarrow (\textit{Store} \rightarrow \mathcal{P}(\textit{State}))$$

daher wie folgt induktiv über der Struktur modallogischer Formeln definiert:

Sei  $\textit{atom} \in \textit{Atom}$ ,  $x \in V$ ,  $\varphi, \psi \in \textit{MF}$  und  $\textit{st} \in \textit{Store}$ .

$$\begin{aligned} \textit{eval}(\textit{True})(\textit{st}) &= \textit{State}, \\ \textit{eval}(\textit{False})(\textit{st}) &= \emptyset, \\ \textit{eval}(\textit{atom})(\textit{st}) &= \{s \in \textit{State} \mid \textit{atom} \in \textit{atoms}(s)\}, \\ \textit{eval}(x)(\textit{st}) &= \textit{st}(x), \\ \textit{eval}(\neg\varphi)(\textit{st}) &= \textit{State} \setminus \textit{eval}(\varphi)(\textit{st}), \\ \textit{eval}(\varphi \wedge \psi)(\textit{st}) &= \textit{eval}(\varphi)(\textit{st}) \cap \textit{eval}(\psi)(\textit{st}), \\ \textit{eval}(\varphi \vee \psi)(\textit{st}) &= \textit{eval}(\varphi)(\textit{st}) \cup \textit{eval}(\psi)(\textit{st}), \\ \textit{eval}(\textit{EX}\varphi)(\textit{st}) &= \{\textit{state} \in \textit{State} \mid \textit{trans}(\textit{state}) \cap \textit{eval}(\varphi)(\textit{st}) \neq \emptyset\}, \text{ exists next} \\ \textit{eval}(\textit{AX}\varphi)(\textit{st}) &= \{\textit{state} \in \textit{State} \mid \textit{trans}(\textit{state}) \subseteq \textit{eval}(\varphi)(\textit{st})\}, \text{ for all next} \\ \textit{eval}(\mu x.\varphi)(\textit{st}) &= \bigcup_{i \in \mathbb{N}} \Phi^i(\emptyset), \\ \textit{eval}(\nu x.\varphi)(\textit{st}) &= \bigcap_{i \in \mathbb{N}} \Phi^i(\textit{State}). \end{aligned}$$

Die Schrittfunktion  $\Phi$  ist hier wie folgt definiert:

$$\begin{aligned}\Phi : \mathcal{P}(\textit{State}) &\rightarrow \mathcal{P}(\textit{State}) \\ Q &\mapsto \textit{eval}(\varphi)(\textit{st}[Q/x]),\end{aligned}$$

wobei für alle  $y \in V$ ,

$$\textit{st}[Q/x](y) = \begin{cases} Q & \text{falls } x = y, \\ \textit{st}(y) & \text{sonst.} \end{cases}$$

Ist *trans* bildendlich, d.h. hat jeder Zustand höchstens endlich viele direkte Nachfolger, und werden alle Vorkommen der gebundenen Variablen einer  $\mu$ -Formel in deren Rumpf von einer geraden Anzahl von Negationen präfixiert, dann ist  $\Phi$  stetig und co-stetig bzgl. der o.g. CPO-Struktur von Potenzmengen.

Also ist  $\textit{eval}(\mu x.\varphi)(\textit{st})$  nach dem [Fixpunktsatz von Kleene](#) der kleinste und  $\textit{eval}(\nu x.\varphi)(\textit{st})$  der größte Fixpunkt von  $\Phi$ . Ist *State* endlich, dann ist auch  $\mathcal{P}(\textit{State})$  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 [\[Pad2\]](#), Kapitel 28).

## Beispiel Mutual exclusion

Jede der folgenden modalen Formeln  $\varphi$  gilt in *Mutex* (s.o), d.h.

$$eval(\varphi)(\lambda x.\emptyset) = State.$$

<b>safety</b>	Es befindet sich immer nur ein Prozess im kritischen Abschnitt. $\neg(c_1 \wedge c_2)$
<b>liveness</b>	Wenn ein Prozess um Einlass in den kritischen Abschnitt bittet, wird er diesen auch irgendwann betreten. $t_i \Rightarrow AF c_i, i = 1, 2$
<b>non-blocking</b>	Ein Prozess kann stets um Einlass in den kritischen Abschnitt bitten. $n_i \Rightarrow EX t_i, i = 1, 2$
<b>no strict sequencing</b>	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  $\Phi$  ist hier wie folgt definiert:

$$\begin{aligned}\Phi : \mathcal{P}(\text{State}^2) &\rightarrow \mathcal{P}(\text{State}^2) \\ \sim &\mapsto \{(s, s') \in \text{State}^2 \mid \text{atoms}(s) = \text{atoms}(s'), \text{trans}(s) \sim \text{trans}(s')\}.\end{aligned}$$

Zwei Zustände  $s$  und  $s'$  heißen **äquivalent**, **verhaltensgleich** oder **bisimilär**, wenn  $(s, s')$  zu  $\text{gfp}(\Phi)$  gehört.

**Aufgabe** Zeigen Sie, dass  $\text{gfp}(\Phi)$  eine Äquivalenzrelation ist. □

Übrigens liefert der Quotient einer Kripke-Struktur nach ihrer Bisimilarität (wie der entsprechende Quotient eines endlichen Automaten) für jeden “Anfangszustand” von  $\text{State}$  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  $\text{Store}$ .

- 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 dem Mikrowellenmodell in Clarke, Grumberg, Peled, Model Checking, Section 4.1.

## Zweidimensionale Figuren

Der **Painter** enthält einen Datentyp für Graphen, die als Listen von Wegen (= Linienzügen) 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 Wege.

$colors(g)$ ,  $modes(g)$  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  $PainterPix/file(g).svg$  geschrieben, so dass beim Öffnen dieser Datei mit einem Browser dort das Bild von  $g$  erscheint. Verlassen wird die Schleife, wenn anstelle einer Parameter-eingabe 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
```

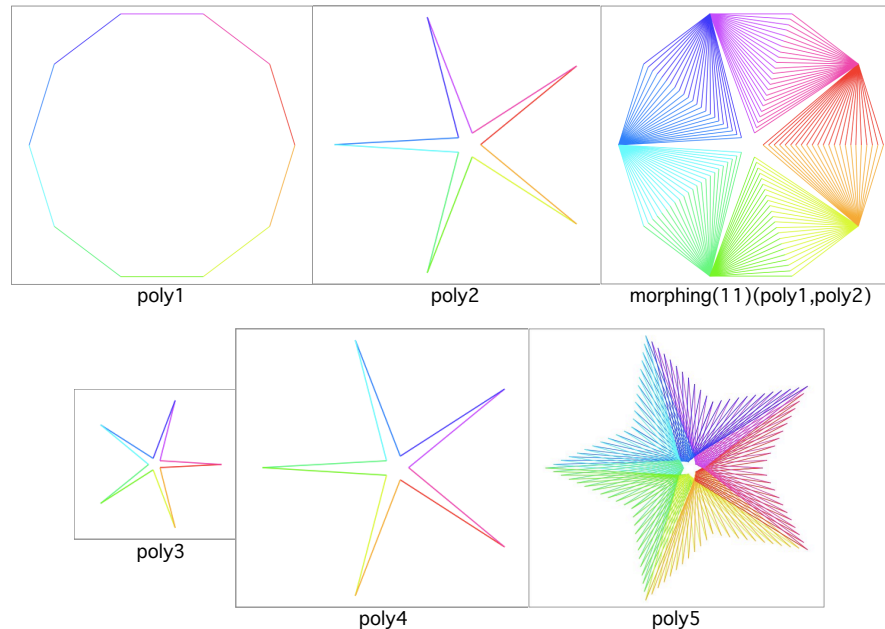
```
morphing n cs = combine $ zipWith f (init cs) $ tail cs where
    f c d = combine $ map g [0..n] where
        g i = zipCurves h c d where
            h (xc,yc) (xd,yd) = (t xc xd,t yc yd)
        t x x' = (1-step)*x+step*x'
        step = float i/float n
```

$zipCurves(f)(g)(g')$  erzeugt einen neuen Graphen aus den Graphen  $g$  und  $g'$ , indem die Funktion  $f : Point \rightarrow Point \rightarrow Point$  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]
```

$\text{poly}(\text{mode})(n)(rs)$  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 anstelle der Linien eines Weges von den Endpunkten der Linien und dem Wegmittelpunkt aufgespannte Dreiecke gezeichnet werden, wie es z.B. bei der Polygon-Komponente des folgenden Graphen der Fall ist:

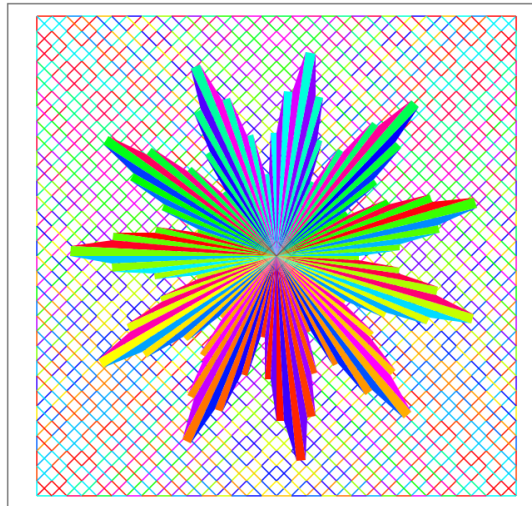
```
graph :: Curves
```

```
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]
```

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



## 7 Funktoren und Monaden

### 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. **[ ]**, **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  $*$ ,  $* \rightarrow *$  bzw.  $* \rightarrow (* \rightarrow *)$ .

Weitere Kinds ergeben sich aus anderen Kombinationen der Kind-Konstruktoren  $*$  und  $\rightarrow$ . Z.B. ist  $(* \rightarrow *) \rightarrow *$  der Kind eines Typs, der eine Funktion darstellt, die jedem Typ des Kinds  $* \rightarrow *$  (also jeder Funktion von einer Menge von Mengen in eine Menge von Mengen) einen Type des Kinds  $*$ , also eine Menge von Mengen 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*)  $a \rightarrow t$  ( $a$  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.

**Funktoren** (Der Haskell-Code steht [hier](#).)

```
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
```

*Listenfunktork*

**Anforderungen** an die Instanzen von **Functor**:

Für alle Mengen  $a, b, c$ ,  $f : a \rightarrow b$  und  $g : b \rightarrow c$ ,

```
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}
```

*Identitätsfunktork*

```
instance Functor Id where
    fmap f (Id a) = Id $ f a
```

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

*siehe Kapitel 4*

```
instance Functor Exp where
    fmap f = \case Con i -> Con i
              Var x -> Var $ f x
              Sum es -> Sum $ map (fmap f) es
              Prod es -> Prod $ map (fmap f) es
              e :- e' -> fmap f e :- fmap f e'
              i :* e -> i :* fmap f e
              e :^ i -> fmap f e :^ i
```

*siehe Abschnitt 4.1*

```
instance Functor Bintree where                                siehe Abschnitt 5.9
    fmap f (Fork a left right) = Fork (f a) (fmap f left)
                                           (fmap f right)
    fmap _ _ = Empty
```

```
instance Functor Tree where                                  siehe Abschnitt 5.9
    fmap = mapTree
```

```
instance Functor ((->) state) where                          Leserfunktork
    fmap f h = f . h
```

```
instance Functor ((,) state) where                           Schreiberfunktork
    fmap f (st,a) = (st,f a)
```

Kompositionen von Leser- und Schreiberfunktoren liefern *Zustandsfunktoren*:

```
newtype State state a = State {runS :: state -> (a,state)}
```

```
instance Functor (State state) where
    fmap f (State h) = State $ \(a,st) -> (f a,st) . h
```

```
data Costate state a = (:#) {out :: state -> a, final :: state}
```

```
instance Functor (Costate state) where  
    fmap f (h:#st) = (f . h):#st
```

Da der Destruktor

$$\text{runS} : \text{State}(\text{state}) \rightarrow (\text{state} \rightarrow (a, \text{state}))$$

invers ist zum Konstruktor

$$\text{State} : (\text{state} \rightarrow (a, \text{state})) \rightarrow \text{State}(\text{state}),$$

sind  $\text{State}(\text{state})$  und  $\text{state} \rightarrow (a, \text{state})$  isomorphe Funktoren.

Analog sind  $\text{Costate}(\text{state})$  und  $(\text{state} \rightarrow a, \text{state})$  isomorph, weil die folgenden Funktionen invers ist 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
```



## Monaden und Plusmonaden (Der Haskell-Code steht [hier](#).)

```
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  $m \in m(a)$ ,  $f : a \rightarrow m(b)$  und  $g : b \rightarrow m(c)$ ,

$$(m \gg= f) \gg= g \quad = \quad m \gg= ((\gg= g) \ . \ f) \quad (1)$$

$$m \gg= \text{return} \quad = \quad m \quad (2)$$

$$(\gg= f) \ . \ \text{return} \quad = \quad f \quad (3)$$

$$m \gg= f \quad = \quad \text{fmap } f \ m \gg= \text{id} \quad (4)$$

$$\text{mzero} \gg= f \quad = \quad \text{mzero} \quad (5)$$

$$m \gg= \text{const mzero} \quad = \quad \text{mzero}$$

$$\text{mzero} \ \texttt{`mplus`} \ m \quad = \quad m$$

$$m \ \texttt{`mplus`} \ \text{mzero} \quad = \quad m$$

Die **do-Notation** bringt monadische Programme näher an die imperative Sicht, nach der ein Programm eine Folge von Befehlen ist, die i.w. aus Variablenzuweisungen besteht. Dementsprechend verwendet sie die folgenden polymorphen Funktionen:

`(<-) :: Monad m => a -> m a -> m ()`

*Zuweisung*

`(;) :: Monad m => m a -> m b -> m b`

*Sequentialisierungsoperator*

Z.B. steht

$$do\ a \leftarrow m_1;\ m_2;\ b \leftarrow m_3;\ a \leftarrow m_4;\ m_5;\ return(a, b)$$

für

$$m_1 \gg= (\lambda a.m_2 \gg (m_3 \gg= (\lambda b.m_4 \gg= (\lambda a.m_5 \gg return(a, b)))))$$

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, in Wirklichkeit gebundene Variablen von  $\lambda$ -Abstraktionen sind, können auch hier komplexe Muster anstelle einfacher Variablen verwendet werden. Passt bei der Ausführung einer Zuweisung  $p \leftarrow m$  die Ausgabe von  $m$  nicht zum Muster  $p$ , dann wird anstelle der Zuweisung der – zu  $m()$  gehörige – Wert von  $fail(matchError)$  zurückgegeben.

Ab Version 7.10 verlangt der Glasgow-Haskell-Compiler, dass **Monad**-Instanzen auch Instanzen der Unterklasse **Applicative** von **Functor** und **MonadPlus**-Instanzen auch Instanzen der Unterklasse **Alternative** von **Applicative** sind. Beide Typklassen werden wir hier nicht behandeln. Stattdessen geben wir an, wie eine beliebige Monade bzw. Plusmonade *M* zur Instanz von **Applicative** bzw. **Alternative** wird:

```
instance Applicative M where
    pure = return
    mf <*> m = mf >>= flip fmap m
    <*> hat den Typ
    M (a -> b) -> M a -> M b

instance Alternative M where
    empty = mzero
    (<|>) = mplus
```

## Monaden-Kombinatoren

```
guard :: MonadPlus m => Bool -> m ()
guard b = if b then return () else mzero
```

Sind (1), (4) und (5) erfüllt, dann gelten die folgenden semantischen Äquivalenzen:

`do guard True; m1; ...; mn` *ist äquivalent zu* `do m1; ...; mn`  
`do guard False; m1; ...; mn` *ist äquivalent zu* `mzero`

`creturn :: MonadPlus m => (a -> Bool) -> a -> m a`  
`creturn f a = do guard $ f a; return a`

*bedingte Einbettung*

`when :: Monad m => Bool -> m () -> m ()`  
`when b m = if b then m else return ()`

*bedingte Monade*

`(=<<) :: Monad m => (a -> m b) -> (m a -> m b)`  
`f =<< m = m >>= f`

*monadische Extension*

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

*Kleisli-Komposition*

`(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)`  
`(>=>) = flip (<=<)`

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

*monadische Multiplikation*

Ist  $m$  ein Funktor und gelten die Gleichungen (1)-(3), dann tun das auch die folgenden:

```
(>>= f)  =  join . fmap f                                (4)
```

```
fmap f   =  (>>= return . f)                            (5)
```

Laut Gleichung (5) liefert jede Monade  $M$  – analog zur o.g. Instanz von **Applicative** – eine **Functor**-Instanz:

```
instance Functor M where fmap f = (>>= return . f)
```

## Weitere Kombinatoren

```
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 []
```

$some(m)$  und  $many(m)$  wiederholen die Prozedur  $m$ , bis sie scheitert. Beide Funktionen listen die Ausgaben der einzelnen Iterationen von  $m$  auf.

*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 :: MonadPlus m => [m a] -> m a
msum = foldr mplus mzero
```

*heißt concat in hugs*

*msum* setzt *mplus* von zwei Prozeduren auf Listen beliebig vieler Prozeduren fort.

```
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*

*sequence(ms)* führt die Prozeduren der Liste *ms* hintereinander aus. Wie bei *some(m)* und *many(m)* werden die dabei erzeugten Ausgaben aufgesammelt.

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_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) $ return ()
```

*heißt sequence in hugs*

*sequence\_(ms)* arbeitet wie *sequence(ms)*, vergisst aber die erzeugten Ausgaben.

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
```



## Monadisches Lookup und Lifting (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
```

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f ma = do a <- ma; return $ f a
```

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f ma mb = do a <- ma; b <- mb; return $ f a b
```

*liftM* und *liftM2* gehören zum ghc-Modul `Control.Monad`.

## Die 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 *foldBtree* (siehe Abschnitt 5.10)

```
foldBtree :: val -> (a -> val -> val -> val) -> Bintree a -> Id val
foldBtree val _ Empty                = return val
foldBtree val f (Fork a left right) = do valL <- foldBtree val f left
                                         valR <- foldBtree val f right
                                         return $ f a valL valR
```

## Maybe- und Listenmonade

```
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
```

```
instance Monad [ ] where return a = [a]
                     (>>=) = flip concatMap
                     fail _ = []
```

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

Eine **partielle Funktion**  $f : A \multimap B$  ist an manchen Stellen undefiniert. “Undefiniert” wird in Haskell durch den obigen Konstruktor *Nothing* wiedergegeben und  $f$  als Funktion vom Typ  $A \rightarrow \text{Maybe}(B)$ .

Eine **nichtdeterministische** oder **mehrwertige Funktion** bildet in eine Potenzmenge ab, ordnet aber einzelnen Elementen ihres Definitionsbereichs in der Regel nur endliche Teilmengen zu. Entsprechend der üblichen Haskell-Darstellung endlicher Mengen als Listen wird eine mehrwertige Funktion  $f : A \rightarrow \mathcal{P}(B)$  in Haskell als Funktion des Typs  $A \rightarrow B^*$  implementiert.

Die üblichen Kompositionen zweier partieller bzw. mehrwertiger Funktionen sind Instanzen der oben definierten Kleisli-Komposition. In der Maybe-Monade gilt nämlich:

```
(g <=< f) a    =    g =<< f a
                =    case f a of Just b -> g b; _ -> Nothing
```

und in der Listenmonade:

```
(g <=< f) a    =    g =<< f a
                =    concat [g b | b <- f a]
```

Ähnlich ist die “bedingte Komposition”

```
do b <- f a; guard $ g b; h b
```

im Kontext der Maybe- bzw. Listenmonade äquivalent zu

```
case f a of Just b | g b -> h b; _ -> Nothing
```

bzw.

```
concat [h b | b <- f a, g b]
```

Außerdem sind eine Fallunterscheidung der Form

```
case f a of Nothing -> g a; b -> b
```

nach Definition der Maybe-Monade äquivalent zu

```
f a `mplus` g a
```

und eine Listenkompensation der Form

```
[g b | a <- s, let b = f a, p b]
```

nach Definition 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.

Für die monadische Multiplikation (s.o.) gilt in der Maybe-Instanz:

```
join $ Just $ Just a   =   Just a
join _                  =   Nothing
```

In der Listeninstanz fällt *join* mit *concat* :  $[[a]] \rightarrow [a]$  zusammen.

## Beispiele partieller Funktionen

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 ([],[ ])
```

## Arithmetische Ausdrücke partiell auswerten

Wir erweitern den Datentyp `Exp(x)` von Abschnitt 4.1 um einen Konstruktor für Divisionen:

```
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
```

Damit wird die Auswertungsfunktion *exp2store* von Abschnitt 4.3 zur partiellen Funktion:

```
exp2storeM :: Exp x -> Store x -> Maybe Int  
exp2storeM e st =  
    case e of Con i    -> Just i  
            Var x      -> Just $ st x  
            Sum es     -> do is <- mapM eval es; Just $ sum is  
            Prod es    -> do is <- mapM eval es; Just $ product is  
            e :- e'     -> do i <- eval e; k <- eval e'; Just $ i-k  
            e :/ e'     -> do i <- eval e; k <- eval e'  
                                guard $ k /= 0; Just $ i `div` k  
            i :* e      -> do k <- eval e; Just $ i*k  
            e :^ i      -> do k <- eval e; Just $ k^i  
    where eval = flip exp2storeM st
```

## Beispiele mehrwertiger Funktionen

Die Listeninstanz des Monadenkombinators *sequence* (s.o.) hat den Typ  $[[a]] \rightarrow [[a]]$  und liefert das – als Liste von Listen dargestellte – kartesische Produkt ihrer jeweiligen Argumentlisten:

$$\text{sequence}([as_1, \dots, as_n]) = [[a_1, \dots, a_n] \mid a_i \in as_i, 1 \leq i \leq n] = as_1 \times \dots \times as_n.$$

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]]
```

*lookupM(a)(s)* (s.o.) liefert in der Maybe-Monade die zweite Komponente des ersten Paares von *s*, dessen erste Komponente mit *a* übereinstimmt.



In der Listenmonade liefert  $lookupM(a)(s)$  die jeweils zweite Komponente aller Paare von  $s$ , deren erste Komponente mit  $a$  übereinstimmt.

## Relationale Programmierung

Wegen der Bijektion

$$(A \rightarrow \mathcal{P}(B)) \cong \mathcal{P}(A \times B)$$

entspricht jede binäre Relation  $R \subseteq A \times B$  einer mehrwertigen Funktion  $f_R : A \rightarrow \mathcal{P}(B)$ , die als Funktion von  $A$  in die Listenmonade  $B^*$  implementiert werden kann. Dies erlaubt uns die Überführung logischer Programme (die immer *Relationen* implementieren) in mehrwertige *Funktionen*, die in do-Notation den ursprünglichen logischen Programmen sehr stark ähneln. Klassische Logik-Programmierung wird z.B. in Kapitel 9 meines *Folienskripts* zur Lehrveranstaltung *Logik für Informatiker* behandelt.

Besteht das logische Programm für  $R$  aus Hornformeln

$$\begin{aligned} R(p_1, t_1) &\Leftarrow \bigwedge_{i=1}^{n_1} R(t_{1i}, p_{1i}) \wedge \varphi_1, \\ &\vdots \\ R(p_k, t_k) &\Leftarrow \bigwedge_{i=1}^{n_k} R(t_{ki}, p_{ki}) \wedge \varphi_k \end{aligned}$$

mit Mustern  $p_1, \dots, p_k, p_{11}, \dots, p_{kn_k}$ , Termen  $t_1, \dots, t_k, t_{11}, \dots, t_{kn_k}$  und Booleschen Ausdrücken  $\varphi_1, \dots, \varphi_k$  derart, dass für alle  $1 \leq i \leq k$ ,  $1 \leq j \leq n_i$ ,  $V_0 = \text{var}(p_i)$  und  $V_j = V_{j-1} + \text{var}(p_{ij})$  Folgendes gilt:

$$\text{var}(t_i) \cup \text{var}(\varphi_i) \subseteq V_{n_i} \quad \text{und} \quad \text{var}(t_{ij}) \subseteq V_{j-1},$$

dann ist  $f_R$  folgendermaßen definiert:

$$\begin{aligned} f_R(p_1) &= \{t_1 \mid \bigwedge_{i=1}^{n_1} (p_{1i} \in R(t_{1i}) \wedge \varphi_1)\}, \\ &\vdots \\ f_R(p_k) &= \{t_k \mid \bigwedge_{i=1}^{n_k} (p_{ki} \in R(t_{ki}) \wedge \varphi_k)\}. \end{aligned}$$

$f_R$  kann wie folgt monadisch implementiert werden:

```
f_R :: A -> [B]
f_R p_1 = do p_11 <- f_R t_11
             :
             p_1n_1 <- f_R t_1n_1
             guard $ phi_1
             [t_1]
:
f_R p_k = do p_k1 <- f_R t_k1
             :
             p_kn_k <- f_R t_kn_k
             guard $ phi_k
             [t_k]
```

```
f_R _ = []
```

Die letzte Gleichung kann entfallen, wenn die Muster  $p_1, \dots, p_k$  alle Elemente von  $A$  abdecken.

Mit Hilfe der parallelen monadischen Komposition *msum* lässt sich  $f_R$  noch kompakter definieren (siehe Kapitel 7):

```
f_R x = msum [do p_1 <- [x]
               p_11 <- f_R t_11
               :
               p_1n_1 <- f_R t_1n_1
               guard $ phi_1
               [t_1],
               :
               do p_k <- [x]
               p_k1 <- f t_k1
               :
               p_kn_k <- f t_kn_k
               guard $ phi_k
               [t_k]]
```

In Abschnitt 10.3 wird ein analoges Schema zur monadischen Definition partieller Funktionen vorgestellt.

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 (Der Haskell-Code steht [hier](#).)

Jede Belegung (*valuation*) eines  $(n \times n)$ -Schachbrettes mit Damen wird als Liste *val* von  $n$  Zahlen aus der Menge  $\{1, \dots, n\}$  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 vergebenen Damenpositionen (= Spaltenindizes) mit  $k \leq n$  und einer  $(n - k)$  elementigen Liste *val* vergebenen 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* (s.o.), 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

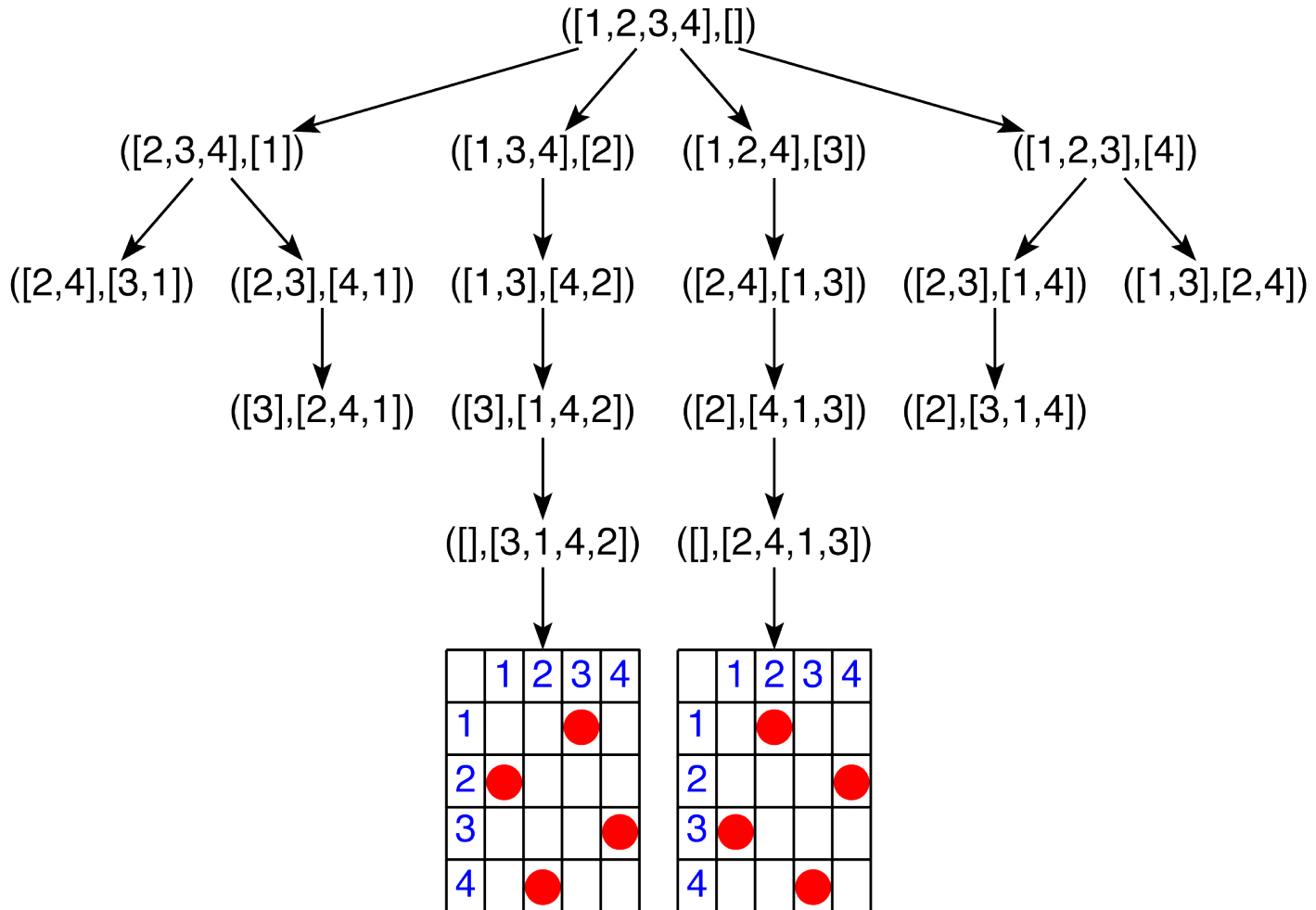
```

*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.

**Aufgabe** Formulieren Sie auf der Basis von *perms* anstelle von *permsI* einen rekursiven Algorithmus *queensR* zur Berechnung sicherer Damenplatzierungen und vergleichen Sie *queensR* mit dem obigen iterativen Algorithmus *queens* bzgl. des Zeitaufwands.

## 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.



## Tiefen- und Breitensuche in Bäumen (Der Haskell-Code steht [hier](#).)

Die Funktionen *depthfirst* und *breadthfirst* suchen nach Knoten eines Baums, die eine als Boolesche Funktion *f* dargestellte Bedingung erfüllen. Dabei bleibt in der Definition der Funktionen nicht nur der Baumtyp offen, sondern auch welche der *f* 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 liefern folgende Instanzen von **TreeC**:

```
instance TreeC Bintree where rootC (Fork a _ _) = a
                             subtreesC Empty      = []
                             subtreesC (Fork _ 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

depthfirst,breadthfirst :: (TreeC t,MonadPlus m)
                                => (a -> Bool) -> t a -> m a

depthfirst f t = msum $ creturn f (rootC t) :
                    map (depthfirst f) (subtreesC t)

breadthfirst f t = visit [t] where
    visit [] = mzero
    visit ts = msum $ map (creturn f . rootC) ts ++
                [visit $ ts >>= subtreesC]

```

`depthfirst(f)(t)` und `breadthfirst(f)(t)` liefern in der Maybe-Monade den – bzgl. Tiefen- bzw. Breitensuche – ersten Knoteneintrag des Baums  $t$ , der die Bedingung  $f$  erfüllt, während in der Listenmonade beide Aufrufe alle Knoteneinträge in Prä- bzw. Heapordnung, die  $f$  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`  $\rightsquigarrow$  `Just (-1)`

`depthfirst (< 0) t1 :: [Int]`  $\rightsquigarrow$  `[-1,-2,-3]`

`breadthfirst (< 0) t1 :: Maybe Int`  $\rightsquigarrow$  `Just (-2)`

`breadthfirst (< 0) t1 :: [Int]`  $\rightsquigarrow$  `[-2,-3,-1]`

`t2 :: BintreeL Int`

`t2 = read "5(4(3,8(9,3)),6(1,2))"`

`depthfirst (> 5) t2 :: Maybe Int`  $\rightsquigarrow$  `Just 8`

`depthfirst (> 5) t2 :: [Int]`  $\rightsquigarrow$  `[8,9,6]`

`breadthfirst (> 5) t2 :: Maybe Int`  $\rightsquigarrow$  `Just 6`

`breadthfirst (> 5) t2 :: [Int]`  $\rightsquigarrow$  `[6,8,9]`

## Lesermonaden (siehe Abschnitt 7.2)

```
instance Monad ((->) state) where return = const
                                   (h >>= f) st = f (h st) st
```

Demnach ist die Funktion

$$\textit{lift} : (a \rightarrow b \rightarrow c) \rightarrow (\textit{state} \rightarrow a) \rightarrow (\textit{state} \rightarrow b) \rightarrow \textit{state} \rightarrow c$$

aus Kapitel 2 die Lesermonaden-Instanz des Monadenkombinators *liftM2* (siehe 7.4).

## Beispiel

Da  $(\rightarrow)(\textit{Store}(x))$  eine Monade ist, können wir die Auswertungsfunktion

$$\textit{exp2store} : \textit{Exp}(x) \rightarrow \textit{Store}(x) \rightarrow \textit{Int}$$

für arithmetische Ausdrücke (siehe Abschnitt 4.3) in eine monadische Form mit do-Notation bringen und so den Zustand *st* aus den Gleichungen eliminieren:

```
exp2store :: Exp x -> Store x -> Int
exp2store =
  \case Con i    -> return i
      Var x      -> ($x)
```

```

Sum es  -> do is <- mapM exp2store es; return $ sum is
Prod es -> do is <- mapM exp2store es; return $ product is
e :- e' -> do i <- exp2store e; k <- exp2store e'
           return $ i-k
i :* e  -> do k <- exp2store e; return $ i*k
e :^ i  -> do k <- exp2store e; return $ k^i

```

## Schreibermonaden

Ein Schreiberfunktork  $(,)(state)$  (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

instance Monoid Int where mempty = 0; mappend = (+)

instance Monoid [a] where mempty = []; mappend = (++)

instance Monoid state => Monad ((,) state) where
    return a = (mempty,a)
    (st,a) >>= f = (st `mappend` st',b) where (st',b) = f a

```

```
mconcat :: Monoid a => [a] -> a
mconcat = foldr mappend mempty
```

Anforderungen an die Instanzen von `Monoid`:

```
(a `mappend` b) `mappend` c = a `mappend` (b `mappend` c)
mempty `mappend` a          = a
a `mappend` mempty          = a
```

**Beispiel** (siehe Abschnitt 4.1)

Da *String* mit dem Listentyp `[Char]` übereinstimmt, ist *String* ein Monoid mit `mempty = []` und `mappend = (++)` und damit *(String,Int)* die entsprechende Schreibermonade.

```
exp2text :: Exp String -> Store String -> (String,Int)
exp2text e st =
  case e of Con i      -> ("",i)
           Var x       -> out $ st x
           Sum es      -> do is <- mapM comp es; out $ sum is
           Prod es     -> do is <- mapM comp es; out $ product is
           e1 :- e2    -> do i <- comp e1; k <- comp e2; out $ i-k
```

```

    e1 :/ e2 -> do i <- comp e1; k <- comp e2; out $ i`div`k
    i :* e'  -> do k <- comp e'; out $ i*k
    e' :^ i  -> do k <- comp e'; out $ k^i
where out i = ("The value of "++show e++" is "++show i++".\n",i)
    comp = flip exp2text st

```

Sei z.B.  $t$  die Darstellung des Ausdrucks  $5*6*7+x-5*2*3$  als Element von  $Expr(String)$ .  
Dann gilt

$$exp2text(t)(\lambda x.66) = (str, 246),$$

wobei  $str$  der folgende String ist:

```

The value of 6*7 is 42.
The value of 5*6*7 is 210.
The value of x is 66.
The value of 5*6*7+x is 276.
The value of 2*3 is 6.
The value of 5*2*3 is 30.
The value of 5*6*7+x-5*2*3 is 246.

```

## Substitution und Unifikation

Wir erweitern den *Exp*-Funktork (siehe Abschnitt 7.2) zur *Exp*-Monade:

```
instance Monad Exp where
  return = Var
  e >>= f = case e of Con i -> Con i
                Var x -> f x
                Sum es -> Sum $ map (>>= f) es
                Prod es -> Prod $ map (>>= f) es
                e :- e' -> (e >>= f) :- (e' >>= f)
                i :* e -> i :* (e >>= f)
                e :^ i -> (e >>= f) :^ i
```

In der *Exp*-Monade hat der *bind*-Operator den Typ

```
Exp x -> (x -> Exp y) -> Exp y
```

und substituiert (ersetzt) Variablen wiederum durch Ausdrücke vom Typ *Exp*(*x*), z.B.

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

```
>>= \"x\" -> 10:*Con 4    ~>
```

```
Sum [(10:*Con 4):^4,5:*((10:*Con 4):^3),11:*((10:*Con 4):^2),Con 222]
```

Auf Bäumen vom Typ  $Tree(a)$  (siehe Abschnitt 5.9) lautet ein entsprechender Substitutionsoperator wie folgt:

```
(>>>) :: Tree a -> (a -> Tree a) -> Tree a
V a >>> sub      = sub a
F a ts >>> sub = F a $ map (>>> sub) ts
```

Vom Typ her könnte  $(>>>)$  ein bind-Operator für  $Tree$  sein.  $(>>=) = (>>>)$  wäre aber nicht mit dem  $map$ -Operator von  $Tree$  (siehe Abschnitt 7.2) kompatibel, d.h. Anforderung (3) an Monaden (siehe Abschnitt 7.3) wäre verletzt.

## Beispiel

```
t :: Tree String
t = F "+" [F "*" [F "5" [], V "x"], V "y", F "11" []]      (5*x)+y+11

sub :: String -> Tree String
sub = \case "x" -> F "/" [F "-" [F "6" []], F "9" [], V "z"]    (-6)/9/z
          "y" -> F "-" [F "7" [], F "*" [F "8" [], F "0" []]]  7-(8*0)
          x  -> V x
```



t >>> sub

→ F "+" [F "\*" [F "5" []], F "/" [F "-" [F "6" []], F "9" []], V "z"]],  
F "-" [F "7" []], F "\*" [F "8" []], F "0" []],  
F "11" []]  $(5 * ((-6) / 9 / z)) + (7 - (8 * 0)) + 11$

Für alle Bäume  $t$  und Substitutionen  $sub : A \rightarrow Tree(A)$  nennt man  $t \ggg sub \in Tree(A)$  die **sub-Instanz von  $t$** .

**Aufgabe** Wie müsste der Datentyp  $Tree(a)$  modifiziert werden, damit er eine Monade wird, deren *bind*-Operator wie der Substitutionsoperator ( $\ggg$ ) nur  $V$ -Knoten ersetzt?  $\square$

Zwei Bäume  $t$  und  $t'$  heißen **unifizierbar**, falls sie einen **Unifikator** haben, d.i. eine Substitution  $sub : A \rightarrow Tree(A)$  mit  $t \ggg sub = t' \ggg sub$ .

Sind  $t$  und  $t'$  unifizierbar, dann liefert  $unify(t)(t')$  einen Unifikator, der allen Elementen von  $A$  möglichst kleine Bäume zuweist:

```
unify :: Eq a => Tree a -> Tree a -> Maybe (a -> Tree a)
unify (V a) (V b)      = Just $ if a == b then V
                        else update V a $ V b
```

```

unify (V a) t          = do guard $ f t; Just $ update V a t
                        where f (V b)      = a /= b
                              f (F _ ts) = all f ts

unify t (V a)          = unify (V a) t
unify (F f ts) (F g us) = do guard $ f == g && length ts == length us
                        unifyall ts us

```

```

unifyall :: Eq a => [Tree a] -> [Tree a] -> Maybe (a -> Tree a)
unifyall [] []          = Just V
unifyall (t:ts) (u:us) = do sub <- unify t u
                            let msub = map (>>> sub)
                            sub' <- unifyall (msub ts) $ msub us
                            Just $ (>>> sub') . sub

```

## Beispiel

```

unify (F "+" [F "-" [V "x", V "y"], V "z"])
      (F "+" [V "a", F "*" [V "b", V "c"]])
      ~> \case "x" -> F "*" [V "b", V "c"]
           "a" -> F "-" [V "x", V "y"]
           x -> V x

```

## 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 *Zustandstransformationen* 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

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 $ update store x
                           $ exp2store e store
trace (Use x:s) store    = ((x,i):s',store')
                           where i = store x
                           (s',store') = trace s store
trace _ store            = ([],store)
```

Zustandsmonadische Version:

```
traceT :: Eq x => [DefUse x] -> State (Store x) [(x,Int)]
traceT (Def x e:s) = do def x e; traceT s
traceT (Use x:s)    = do i <- use x
                           s' <- traceT s
                           return $ (x,i):s'
traceT _            = return []
```

In der transitionsmonadischen Version von *trace* taucht der Zustandsparameter *store* nicht mehr auf. Alle Änderungen von bzw. Zugriffe auf *store* erfolgen über Aufrufe der Elementarfunktionen *def* und *use*:

```
def :: Eq x => x -> Exp x -> State (Store x) ()
def x e = T $ \store -> ((),update store x $ exp2store e store)

use :: x -> State (Store x) Int
use x = T $ \store -> (store x,store)
```

## 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 $ runT (traceT dflist) $ const 0 ~> [(X,1),(Y,2),(X,3),(Y,2)]
```

## Die IO-Monade

kann man sich vorstellen als Zustandsmonade, deren Operationen die Zustände von Ein/Ausgabe 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 eingegebenen `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  $f : IOError \rightarrow IO(a)$  ausgeführt.

## Beispiele

```
readFileContinue :: String -> (String -> IO a) -> a -> IO ()
readFileContinue file a act =
    do str <- readFile file `catch` handler
       if null str then do putStrLn $ file++" does not exist"
                           return a
       else act str
  where handler :: IOError -> IO String
        handler _ = return ""
```

*readFileContinue(file)(a)(continue)* den Inhalt der Datei *file* und übergibt ihn zur Weiterverarbeitung an die Funktion *continue*, falls *file* existiert.

Andernfalls werden eine Fehlermeldung und  $a$  ausgegeben.

```
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.

## Zustandsvariablen (mutable, dynamic objects)

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

$$\text{do } x \leftarrow m; f(x); x \leftarrow g(x); h(x).$$

Da dieser für

$$m \gg= (\lambda x. f(x) \gg= (g(x) \gg (\lambda x. h(x))))$$

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  $0 : 1 : 0 : 1 : 0 : 1 : \dots$  (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)*. Erzeugung, Zugriff, Setzen und Modifizieren von Zeigern bzw. Werten, auf die sie zeigen, erfolgen mit folgenden IO-monadischen Basisfunktionen:

```
newIORef      :: a -> IO (IORef a)
readIORef     :: IORef a -> IO a
writeIORef    :: IORef a -> a -> IO ()
modifyIORef   :: IORef a -> (a -> a) -> IO ()
```

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, was die Bezeichnung **dynamisch** begründet.

M.a.W.: Ein dynamisches Objekt besitzt eine Menge von Zustandsvariablen, die als Zeiger auf Zustände implementiert werden, die durch die aktuellen Werte von Attributen des Objektes gegeben sind. Demzufolge hat ein dynamisches Objekt stets einen anderen Typ als seine möglichen Zustände. Nur ist es in der Regel kein einzelner Zeiger auf einen nicht weiter strukturierten Wert, sondern das Element einer Objektklasse, die wir in Haskell als IO-monadische Erweiterung eines Datentyps mit Destruktoren realisieren.

## Beispiel Linienzüge als verkettete Listen

In Kapitel 3 wurden Linienzüge als Punktlisten des Datentyps *[Point]* implementiert. Im Folgenden realisieren wir sie als **verkettete Listen** (*linked lists*) dynamischer Objekte, die wir hier als *Zellen* bezeichnen. Jede Zelle besteht aus einem Zeiger *pointRef* auf einen Punkt einem Zeiger *nextRef* auf die jeweilige Nachfolgerzelle.

Wir beschränken uns auf Listen, deren Punkte paarweise verschieden sind. Daher entsprechen Polygone, also geschlossene Linienzüge, zyklischen Listen, während die Listendarstellung jedes anderen Linienzuges eine Zelle enthält, deren Zeiger *pointRef* auf *Nothing* zeigt.

```
data Cell = Cell {getPoint :: IO Point, setPoint :: Point -> IO (),
                  getNext  :: IO (Maybe Cell),
                  setNext  :: Maybe Cell -> IO (),
                  getDist  :: IO Float}
```

*Erzeugung einer Zelle:*

```
newCell :: Point -> Maybe Cell -> IO Cell
```

```
newCell point next = do
    pointRef <- newIORef point
    nextRef  <- newIORef next
    let getPoint = readIORef pointRef
        setPoint = writeIORef pointRef
        getNext  = readIORef nextRef
        setNext  = writeIORef nextRef
        getDist  = do pt <- getPoint; next <- getNext
                     case next of
                         Nothing -> return 0
                         Just (Cell getPt _ _ _ _)
                             -> do pt' <- getPt
                                    return $ distance pt pt'
```

```
return $ Cell getPoint setPoint getNext setNext getDist
```

*getDist* liefert die Distanz zum Punkt der nächsten Zelle.

```
lengthCyclic :: Cell -> IO (Float,Bool)
```

```
lengthCyclic cell = do
```

```
  point <- cell&getPoint
```

```
  next <- cell&getNext
```

```
  dist <- cell&getDist
```

```
  let loop :: Maybe Cell -> IO (Float,Bool)
```

```
      loop Nothing      = return (dist,False)
```

```
      loop (Just cell) = do pt <- cell&getPoint
```

```
                           next <- cell&getNext
```

```
                           dist <- cell&getDist
```

```
                           if point == pt then return (dist,True)
```

```
                           else do (dist',b) <- loop next
```

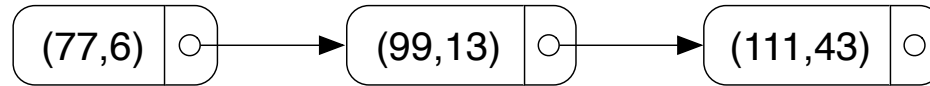
```
                               return (dist+dist',b)
```

```
  loop next
```

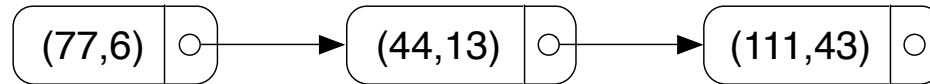
*lengthCyclic(cell)* berechnet die Länge des Linienzuges  $L$ , dessen erster Punkt in *cell* steht, und stellt fest, ob  $L$  ein Polygon ist oder nicht.

## Beispiel

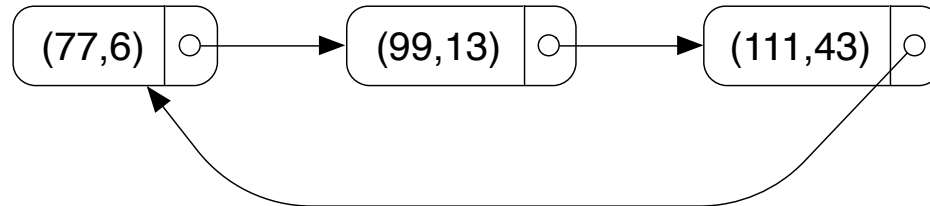
path



path'



poly



```
testPath :: IO ()
```

```
testPath = do
```

```
  let path@[pt1,pt2,pt3] = [Point 77 6,Point 99 13,Point 111 43]
```

```
  putStrLn $ "path = " ++ show path ++
```

```
    "\nlength = " ++ show (lengthPath path)
```

```

~> path = [(77.0,6.0),(99.0,13.0),(111.0,43.0)]
    length = 55.39778
rec cell1 <- newCell pt1 $ Just cell2
    cell2 <- newCell pt2 $ Just cell3
    cell3 <- newCell pt3 Nothing
path <- mapM getPoint [cell1,cell2,cell3]
(lg,b) <- lengthCyclic cell1
putStrLn $ "path = " ++ show path ++
           "\nlength = " ++ show lg ++ "\ncyclic = " ++ show b
~> path = [(77.0,6.0),(99.0,13.0),(111.0,43.0)]
    length = 55.39778
    cyclic = False
pt <- cell2&getPoint
(cell2&setPoint) $ pt {x = pt&x-55}
path' <- mapM getPoint [cell1,cell2,cell3]
(lg,b) <- lengthCyclic cell1
putStrLn $ "path' = " ++ show path' ++
           "\nlength = " ++ show lg ++ "\ncyclic = " ++ show b
~> path' = [(77.0,6.0),(44.0,13.0),(111.0,43.0)]
    length = 107.14406
    cyclic = False

```

```

let cycle = path' ++ cycle
    poly = take 4 cycle
putStrLn $ "poly = " ++ show poly ++
    "\nlength = " ++ show (lengthPath poly)
~>    poly = [(77.0,6.0),(44.0,13.0),(111.0,43.0),(77.0,6.0)]
        length = 157.39343
(cell3&setNext) $ Just cell1
poly <- mapM getPoint [cell1,cell2,cell3,cell1]
(lg,b) <- lengthCyclic cell1
putStrLn $ "poly = " ++ show poly ++
    "\nlength = " ++ show lg ++ "\ncyclic = " ++ show b
~>    poly = [(77.0,6.0),(44.0,13.0),(111.0,43.0),(77.0,6.0)]
        length = 157.39343
        cyclic = True
pt2 <- cell2&getPoint
rec cell1 <- newCell pt1 $ Just cell2
    cell2 <- newCell pt2 $ Just cell3
    cell3 <- newCell pt3 $ Just cell1
poly <- mapM getPoint [cell1,cell2,cell3,cell1]
(lg,b) <- lengthCyclic cell1

```



```
putStrLn $ "poly = " ++ show poly ++  
          "\nlength = " ++ show lg ++ "\ncyclic = " ++ show b  
~> poly = [(77.0,6.0),(44.0,13.0),(111.0,43.0),(77.0,6.0)]  
    length = 157.39343  
    cyclic = True
```

Verkettete Listen sind zwar ein gutes Beispiel, um das Prinzip der Erzeugung und Verarbeitung dynamischer Objekte zu demonstrieren. In vielen konkreten Anwendungen spielen sie aber als zu aufwändige Alternative zu Haskell's Listentyp eine eher untergeordnete Rolle. Dort sollte der Einsatz dynamischer Objekte verteilten Systemen mit komplexer Verknüpfungsstruktur und zahlreichen Attributen vorbehalten bleiben. Erst bei solchen Anwendungen machen objektorientierte Sprachkonstrukte Sinn. Zahlen oder Listen von Zahlen können mit einfacheren Mitteln behandelt werden.

## 8 Felder

### **Ix**, die Typklasse für Indexmengen und ihre **Int**-Instanz

```
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 eine Liste 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  $f : A \rightarrow B$  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:

$(!) :: \text{Ix } a \Rightarrow \text{Array } a \ b \rightarrow a \rightarrow b$

Funktionsapplikation wird zum Feldzugriff: Für alle  $i \in [a, b]$ ,  $f(i) = \text{mkArray}(f)!i$ .

Update-Operator für Felder:

$(//) :: \text{Ix } a \Rightarrow \text{Array } a \ b \rightarrow [(a,b)] \rightarrow \text{Array } a \ b$

Für alle Felder  $arr$  mit Indexmenge  $A$  und Wertemenge  $B$ ,  
 $s = [(a_1, b_1), \dots, (a_n, b_n)] \in (A \times B)^*$  and  $a \in A$  gilt also:

$$(arr//s)!a = \begin{cases} b_i & \text{falls } a = a_i \text{ für ein } 1 \leq i \leq n, \\ arr!a & \text{sonst.} \end{cases}$$

$a_1, \dots, a_n$  sind genau die Indizes des Feldes  $arr$ , an denen es sich von  $arr//s$  unterscheidet.

Feldgrenzen:

$\text{bounds} :: \text{Ix } a \Rightarrow \text{Array } a \ b \rightarrow (a,a)$

$\text{bounds}(arr)$  liefert die kleinsten und größten Index, an dem das Feld  $arr$  definiert ist.

## Dynamische Programmierung

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  $\text{fib}(i)$  aus Funktionswerten an Stellen  $j < i$ . Diese stehen aber bereits in  $\text{fibA}$ , wenn der  $i$ -te Eintrag vorgenommen wird. Folglich sind alle rekursiven Aufrufe in der ursprünglichen Definition von  $\text{fib}$  als Zugriffe auf bereits belegte Positionen von  $\text{fibA}$  implementierbar.

*ghci* gibt z.B. 19,25 Sekunden als Rechenzeit für  $\text{fib}(33)$  an. Für  $\text{fibA}!33$  liegt sie hingegen unter 1/100 Sekunde.

Generell sollten Funktionen mit einem Definitionsbereich ein Typ der Klasse ***Ix*** (siehe Abschnitt 9.1) als Felder implementiert werden. Diese benötigen erheblich weniger Speicherplatz als die ursprünglichen Funktionen, weil Funktionen durch – manchmal sehr umfangreiche –  $\lambda$ -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.

## Matrizenrechnung

(Der Haskell-Code steht [hier](#).)

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  $Pos \rightarrow r$  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) $ liftM2 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 Funktion

$$f : \text{Matrix}(R) \rightarrow \text{Matrix}(R)$$

$$M' \mapsto M + M * M'$$

ist stetig, hat also nach dem Fixpunktsatz von Kleene den kleinsten Fixpunkt  $\bigsqcup_{i \in \mathbb{N}} f^i(0)$ . Da dieser mit

$$M^+ =_{\text{def}} M + M^2 + M^3 + \dots$$

übereinstimmt, kann der transitive Abschluss von  $M$  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 le' (add m . mul m) $ zeroM $ dim m where
  le' m m' = all (liftM2 le (m!) (m'!)) [(i,j) | i <- s, j <- s]
    where s = [1..dim m]
```

## Graphen als Matrizen (Der Haskell-Code steht [hier](#).)

Von Graphen zu Matrizen und umgekehrt (siehe Abschnitt 6.4)

```
data GraphM a r = M [a] (Matrix r)
```

```
graph2mat :: Eq a => Graph a -> Matrix Bool
```

```
graph2mat (G nodes sucs) = mkMat (length nodes) f where  
    f (i,j) = nodes!!(j-1) `elem` sucs (nodes!!(i-1))
```

```
graphL2mat :: (Eq a, Semiring r) => GraphL a r -> Matrix r
```

```
graphL2mat (GL nodes sucs) = mkMat (length nodes) f where  
    f (i,j) = case lookup (nodes!!(j-1)) $ map (snd *** fst)  
                                     $ sucs (nodes!!(i-1))  
              of Just label -> label; _ -> zero
```

```
mat2graph :: Eq a => GraphM a Bool -> Graph a
```

```
mat2graph (M nodes m) = G nodes $ f . rowPos nodes where  
    f i = [nodes!!(j-1) | j <- [1..length nodes], m!(i,j)]
```



```

mat2graphL :: (Eq a,Eq r,Semiring r) => GraphM a r -> GraphL a r
mat2graphL (M nodes m) = GL nodes $ f . rowPos nodes where
    f i = [(label,nodes!!(j-1)) | j <- [1..length nodes],
                                   let label = m!(i,j),
                                   label /= zero]

```

```

rowPos :: Eq a => [a] -> a -> Int
rowPos nodes a = case search (== a) nodes of Just i -> i+1; _ -> 0

```

Transitiver Abschluss eines unmarkierten Graphen als Matrixabschluss

```

closureM :: Eq a => Graph a -> Graph a
closureM graph@(G nodes _) = mat2graph $ M nodes $ plus (<=)
                                   $ graph2mat graph

```

Bei den unten behandelten Instanzen von  $R$  gilt:

$$M^+ = M + M^2 + \dots + M^{\dim(M)}. \quad (1)$$

Da die Multiplikation zweier Matrizen kubische Kosten in der Dimension der Matrizen hat, ist leider auch im Fall (1) der Aufwand von **plus** biquadratisch, so dass eine Matrix-Implementierung des “nur” kubischen Warshall-Algorithmus keine Vorteile zu bieten scheint.

Andererseits erzeugt daher keine komplexen rekursiven Aufrufe wie die dreifach geschachtelten Faltungen von `warshall` (siehe Abschnitt 6.4).

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)
    zero = ([],0); one = ([],0)
```

$allpaths(G)$  markiert jede Kante  $a \rightarrow b$  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 :: Eq a => Graph a -> GraphL a Int
allpaths (G nodes sucs) = g $ mat2graphL $ M nodes $ plus le
                        $ graphL2mat $ GL nodes $ map f . sucs
    where le (_,m) (_,n) = m <= n
```

```

f a = (([[a]],1),a)
g (GL nodes sucs) = GL nodes $ map h . sucs
h ((_ ,n),a) = (n,a)

```

## Beispiele

```

graph1 = G [1..6] $ \case 1 -> [2,3]; 3 -> [1,4,6]; 4 -> [1]
                        5 -> [3,5]; 6 -> [2,4,5]; _ -> []

```

```

graph2 = G [1..6] $ \a -> if a `elem` [1..5] then [a+1] else []

```

```

allpaths graph1    ~>  1 -> [(3,1),(5,2),(1,3),(2,4),(1,5),(1,6)]
                        2 -> []
                        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)]
                      6 -> []
```

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)
```

$\text{minpaths}(G)$  markiert jede Kante  $a \rightarrow b$  von  $G$  mit dem kürzesten Weg von  $a$  nach  $b$  und dessen Länge:

```

minpaths :: Eq a => GraphL a Int -> GraphL a (Path a)
minpaths (GL nodes sucs) = mat2graphL $ M nodes $ plus le $ graphL2mat
                                $ GL nodes $ map f . sucs
                                where le (_,m) (_,n) = m >= n
                                      f (n,a) = ([a],n),a

```

## Beispiel

```

graph5 :: GraphL Int Int
graph5 = G [1..5] $ \case 1 -> [(100,5),(40,2)]
                           2 -> [(50,5),(10,3)]
                           3 -> [(20,4)]
                           4 -> [(10,5)]
                           _ -> []

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.

## Alignments

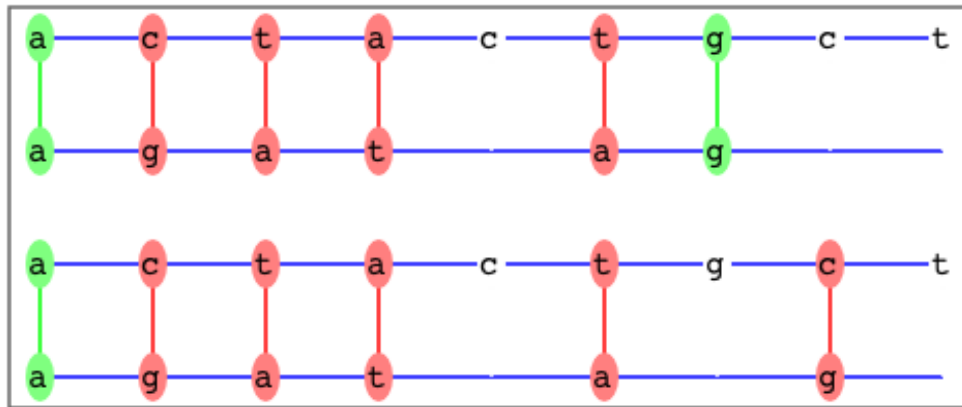
Der Haskell-Code steht [hier](#).

Die algebraische Behandlung bioinformatischer Probleme geht zurück auf: R. Giegerich, A Systematic Approach to Dynamic Programming in Bioinformatics, Bioinformatics 16 (2000) 665-677.

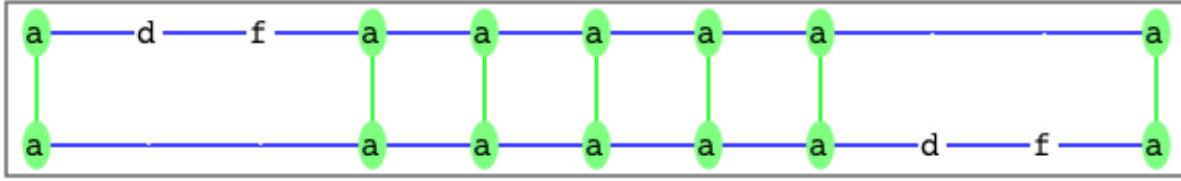
Zwei Listen  $xs$  und  $ys$  des Typs  $String^*$  sollen in die Menge  $alis(xs, ys)$  ihrer Alignments von übersetzt werden. Wir setzen eine Boolesche Funktion

$$compl : String^2 \rightarrow Bool$$

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  $A = \text{String} \uplus \{\text{Nothing}\}$  und  $h : A^* \rightarrow \text{String}^*$  die Funktion, die aus einem Wort über  $A$  alle Vorkommen von *Nothing* streicht.

Dann ist die Menge der **Alignments** von  $xs, ys \in \text{String}^*$  wie folgt definiert:

$$\begin{aligned}
 \text{alis}(xs, ys) =_{\text{def}} & \bigcup_{n=\max(|xs|, |ys|)}^{|xs|+|ys|} \{[(a_1, b_1, c_1), \dots, (a_n, b_n, c_n)] \in (A \times A \times \text{RGB})^n \mid \\
 & h(a_1 \dots a_n) = xs \wedge h(b_1 \dots b_n) = ys \wedge \\
 & \forall 1 \leq i \leq n : (c_i = \text{red} \wedge \text{compl}(a_i, b_i)) \vee \\
 & (c_i = \text{green} \wedge a_i = b_i \neq \text{Nothing}) \vee \\
 & (c_i = \text{white} \wedge ((a_i = \text{Nothing} \wedge b_i \neq \text{Nothing}) \\
 & \vee (b_i = \text{Nothing} \wedge a_i \neq \text{Nothing}))) \}
 \end{aligned}$$

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 folgendem **Algebren**-Datentyp zusammen (siehe auch die Abschnitte 5.10 und 9.3):

```
data AliAlg 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 *AliAlg*, die Alignments in unterschiedlicher Weise abspeichern:

```
type Genes = ([String],[String])
```

```
lists :: Genes -> AliAlg [String] (Genes -> [Align])
lists gs = A {first = gs, next = tail, maxi = [], maxj = [],
              getx = head, gety = head, entry = ($), mkAlis = id}
```

```

fun :: Genes -> AliAlg 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 = \i -> ys!!(i-1),
                 entry = ($), mkAlis = id}

```

```

arr :: Genes -> AliAlg Int (Matrix [Align])
arr (xs,ys) = A {first = (1,1), next = (+1), maxi = maxi, maxj = maxj,
                 getx = \i -> xs!!(i-1), gety = \i -> 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 => AliAlg 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  $O(3^{\text{length}(xs++ys)})$  Rechenschritte, während das Füllen des Feldes beim dritten Aufruf nur  $O(\text{length}(xs++ys))$  benötigt.

Die Funktionen, die die Ergebnisse von *maxAlis* wie in den obigen Beispielen graphisch darstellen, stehen [hier](#).

## 9 Monadentransformer und Comonaden

### Huckepack-Zustandsmonaden

unterscheiden sich von Zustandsmonaden dadurch, dass ihr jeweiliger Wertetyp  $(a, \text{state})$  in eine (Plus-)Monade  $m$  eingebettet ist:

```
newtype StateT state m a = StateT {runST :: state -> m (a,state)}

instance Monad m => Functor (StateT state m) where
    fmap f (StateT h) = StateT $ \(a,st) -> return (f a,st) <=<

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
```

Der Typ  $StateT(state)$  ist dritter Ordnung (!), bildet eine Monade  $m$  auf die Monade  $StateT(state)(m)$  ab und transformiert dabei die auf  $m$  definierten Operationen  $return$ ,  $>>=$ ,  $fail$ ,  $mzero$  und  $mplus$  in entsprechende Operationen auf der Menge – durch den Konstruktor  $StateT$  gekapselter – Funktionen von  $state$  nach  $m(a, state)$ .

$StateT(state)$  wird deshalb auch **Monadentransformer** genannt.

Die Einbettung der in Abschnitt 7.6 behandelten Maybe- bzw. Listenmonade in eine Zustandsmonade liefert Automaten:

$StateT(state)(\textcolor{red}{Maybe})$  ist die Menge aller **partiellen Automaten** mit Zustandsmenge  $state$ .  $fst \circ runST$  und  $snd \circ runST$  bilden die jeweilige partielle Ausgabe- bzw. Übergangsfunktion.

$StateT(state)(\textcolor{red}{[ ]})$  ist die Menge aller **nichtdeterministischen 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.

## Monadentransformer zur Verschmelzung zweier Monaden

Die Variablen einer Folge monadischer “Befehle”

$$do\ x \leftarrow m_1; m_2; y \leftarrow m_3; z \leftarrow m_4; m_5 \quad (1)$$

können zwar Werte verschiedener Typen annehmen, die Ausdrücke  $m_1, \dots, m_5$  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- bzw. Listenmonade  $M$  wird  $T$  wie folgt implementiert:

```
newtype MaybeT m a = MaybeT {runMT :: m (Maybe a)}
```

```
instance Monad m => Monad (MaybeT m) where
    return = MaybeT . return . Just
    m >>= f = MaybeT $ do ma <- runMT m
                        case ma of Just a -> runMT $ f a
                                _ -> return Nothing
```

 (2)



```

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 => 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,mas'] <- mapM runLT [m,m']
                                return $ mas++mas'

```

Der Monadentransformer  $StateT(state)$  (s.o.) bettet nur den Wertebereich der jeweiligen Transitionsfunktion in die Monade  $m$  ein, auf die er angewendet wird:

```

newtype StateT state m a = StateT {runST :: state -> m (a,state)}

```

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
    lift'   :: 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
    lift' = MaybeT . return      -- :: Maybe a -> MaybeT m a    (3)

instance MonadTrans ListT where
    type M ListT = [ ]
    lift  = ListT . liftM single -- :: m a -> ListT m a
    lift' = ListT . return      -- :: [a] -> ListT m a

instance MonadTrans (StateT state) where
    type M (StateT state) = Trans state
    lift m = StateT $ \st -> do a <- m; return (a,st)
    -- :: m a -> StateT state m a
```

```
lift' (T f) = StateT $ return . f
-- :: Trans state a -> StateT state m a
```

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:

$$do\ x \leftarrow lift(m_1); lift(m_2); y \leftarrow lift'(m_3); z \leftarrow lift'(m_4); lift(m_5) \quad (4)$$

Ist z.B.  $T = MaybeT$  und  $m_4$  von der Form  $Just(a)$ , dann vereinfacht sich (4) wegen

$$lift' \circ Just \stackrel{(3)}{=} MaybeT \circ return \circ Just \stackrel{(2)}{=} return$$

zu:

$$do\ x \leftarrow lift(m_1); lift(m_2); y \leftarrow lift'(m_3); z \leftarrow return(a); lift(m_5) \quad (5)$$

## Verschmelzung von Leser- und Maybe-Monade

Wir verschmelzen die Lesermonade  $m = (\rightarrow)(Store(x))$  mit der Maybe-Monade zu  $MaybeT(m)$  und bringen die Auswertungsfunktion

$$exp2storeM : Exp(x) \rightarrow M(Maybe(Int))$$

(siehe Abschnitt 7.6) in eine Form, die arithmetische Ausdrücke in  $MaybeT(M)$ -Objekte übersetzt:

```

exp2storeT :: Exp x -> MaybeT ((->) (Store x)) Int
exp2storeT =
  \case Con i    -> return i
        Var x    -> lift ($x)
        Sum es   -> do is <- mapM exp2storeT es; return $ sum is
        Prod es  -> do is <- mapM exp2storeT es; return $ product is
        e :- e'  -> do i <- exp2storeT e
                        k <- exp2storeT e'
                        return $ i-k
        e :/ e'  -> do i <- exp2storeT e
                        k <- exp2storeT e'
                        guard $ k /= 0; return $ i`div`k
        i :* e   -> do k <- exp2storeT e; return $ i * k
        e :^ i   -> do k <- exp2storeT e; return $ k^i

```

## Beispiele

```

e1 = Sum [Var"x":^4,5:*(Var"x":^3),11:*(Var"x":^2),Con 222]
e2 = Con 5:/(Con 4:-Var"x")

```

```

runMT (exp2storeT e1) $ \"x" -> 4   ~> Just 974
runMT (exp2storeT e2) $ \"x" -> 4   ~> Nothing

```

## Verschmelzung von IO- und Maybe-Monade

Stellt man die Variablenbelegungen von  $Store(x)$  als binäre Relationen dar (siehe Abschnitt 3.7) und legt diese in Dateien ab, dann kann die partielle Auswertung in der Monade  $MaybeT(IO)$  durchgeführt im Fall der Undefiniertheit mit einer entsprechende Fehlermeldung versehen werden (siehe Abschnitt 7.11):

```
exp2storeIO :: (Eq x, Read x, Show x)
              => String -> Exp x -> MaybeT IO Int

exp2storeIO file =
  \case Con i    -> return i
      Var x      -> do store <- lift $ readFileContinue file []
                      $ return . read
                      let val = lookup x store
                      lift $ putStrLn $ show x ++
                        case val of Just i -> " = " ++ show i
                                   _   -> " is undefined"
                      lift' val
      Sum es     -> do is <- mapM eval es; return $ sum is
      Prod es    -> do is <- mapM eval es; return $ product is
      e :- e'    -> do i <- eval e; k <- eval e'; return $ i-k
      e :/ e'    -> do i <- eval e; k <- eval e'
```

```

        if k /= 0 then return $ i`div`k
        else do lift $ putStrLn "divide by zero?"
              mzero
    i :* e  -> do k <- eval e; return $ i*k
    e :^ i  -> do k <- eval e; return $ k^i
  where eval = exp2storeIO file

```

## 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)]

```

```

⇒ runMT $ exp2storeIO "store1" e1    ~>  "x" = 4
                                         "x" = 4
                                         "x" = 4
                                         Just 974
runMT $ exp2storeIO "store1" e2    ~>  "x" = 4
                                         divide by zero?
                                         Nothing

```

```

contents("store2") = [("z",4),("y",55)]

```

```

⇒ runMT $ exp2storeIO e1 "store2"  ~>  "x" is undefined
                                     Nothing
runMT $ exp2storeIO e2 "store2"  ~>  "x" is undefined
                                     Nothing

```

"store3" does not exist

```

⇒ runMT $ exp2storeIO e1 "store3"  ~>  store3 does not exist
                                     "x" is undefined
                                     Nothing

```

## Verschmelzung von IO- und Listenmonade

Wir erweitern die iterative Funktion *queens* zur Berechnung von Lösungen des Damenproblems (siehe Abschnitt 7.6) um die Ausgabe der dazu erforderlichen Zustandsübergänge (siehe Abschnitt 7.6):

```
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" $ showTransR st sts
                  msum $ map qloop sts
                  where sts = qsuccessors st
```

```
showTransR :: Show a => a -> [a] -> String
```

```
showTransR a [] = ""
```

```
showTransR a [b] = show a ++ " -> " ++ show b ++ "\n"
```

```
showTransR 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 dargestellt werden.

## Verschmelzung von IO- und Huckepack-Zustandsmonade

Wir formulieren eine weitere monadische Version der *trace*-Funktion aus Abschnitt 7.10. Anstelle 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 lift' $ def x e
                        traceIO s
traceIO (Use x:s)    = do i <- lift' $ use x
                        lift $ putStrLn $ show (x,i)
                        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)

## Generische Compiler

Im Folgenden werden Übersetzungsfunktionen als Huckepack-Zustandsmonaden implementiert, deren Zustandsmengen aus den – vom jeweiligen Compiler zu verarbeitenden – Eingabewörtern bestehen.

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 durch die Huckepack-Zustandsmonade

```
type Compiler = StateT String Maybe
```

mit *String* als Zustandsmenge implementieren.

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 [do string "True"; return True,
             do string "False"; return False]
```

```
nat,int :: Compiler Int
nat = do ds <- some digit; return $ read ds
int = msum [nat, do char '-'; n <- nat; return $ -n]
```

```
identifizier :: Compiler String
identifizier = 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, Zeilenbrüche 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]

```

## Arithmetische Ausdrücke kompilieren II

In Abschnitt 5.10 haben wir an den Beispielen  $Bintree(A)$  und  $Tree(A)$  gezeigt, dass jede auf einem Datentyp induktiv definierte Funktion  $f$  als Baumfaltung darstellbar ist. Man muss dazu die Konstruktoren des Datentyps 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 **Algebra** (= Interpretation der Konstruktoren) in gleicher Weise abläuft.

Soll auch eine nicht induktiv definierte Funktion  $f : A \rightarrow B$  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 Wortdarstellungen arithmetischer Ausdrücke ohne den Umweg über deren Repräsentation als Objekte vom Typ  $Exp(x)$  (siehe Abschnitt 4.1) direkt in Elemente der ihm als Parameter übergebenen  $Exp(x)$ -Algebra übersetzt, die aus einer Interpretation der Konstruktoren von  $Exp(x)$  als Operationen einer Zielsprache des Compiler besteht.

Jede  $Exp(x)$ -Algebra kann in Haskell als Element des folgenden Datentyps implementiert werden:

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

Hier zunächst die Faltungsfunktion für  $Exp(x)$  gemäß Abschnitt 5.10:

```
foldExp :: ExpAlg x val -> Exp x -> val  
foldExp alg = \case Con i    -> con alg i  
                    Var x    -> var alg x  
                    Sum es   -> sum_ alg $ map (foldExp alg) es  
                    Prod es  -> prod alg $ map (foldExp alg) es  
                    e :- e'  -> sub alg (foldExp alg e)  
                               (foldExp alg e')  
                    i :* e   -> scal alg i $ foldExp alg e  
                    e :^ i   -> expo alg (foldExp alg e) i
```



Die Faltung arithmetischer Ausdrücke in folgender  $Exp(x)$ -Algebra *storeAlg* entspricht ihrer Auswertung mit *exp2store* (siehe 4.3), d.h.

$$foldExp(storeAlg) = exp2store.$$

Wir verwenden die do-Notation, weil  $(\rightarrow)(Store(x))$  eine Monade ist (siehe 7.7):

```
storeAlg :: ExpAlg x (Store x -> Int)
storeAlg = ExpAlg {con = const,
                   var = flip ($),
                   sum_ = \es -> do is <- sequence es
                                return $ sum is,
                   prod = \es -> do is <- sequence es
                                return $ product is,
                   sub = \e e' -> do i <- e; k <- e'; return $ i-k,
                   scal = \i e st -> do k <- e; return $ i*k,
                   expo = \e i st -> do k <- e; return $ k^i}
```

Die Faltung arithmetischer Ausdrücke in folgender  $Exp(x)$ -Algebra *codeAlg* entspricht ihrer Übersetzung mit *exp2code* (siehe 5.11), d.h.

$$foldExp(codeAlg) = exp2code.$$

```

codeAlg :: ExpAlg 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],
                  scal = \i e -> Push i:e++[Mul 2],
                  expo = \e i -> e++[Push i,Up]}

```

Ein generischer Compiler für Wortdarstellungen arithmetischer Ausdrücke mit String-Variablen lautet nun wie folgt:

```

expC :: ExpAlg String val -> Compiler val
expC alg = do e <- summand; moreSummands e where
  summand = do e <- msum [scalar,factor]; moreFactors e
  factor  = msum [do i <- tint; power $ con alg i,
                  do x <- tidentifier; power $ var alg x,
                  do tchar '('; e <- expC alg; tchar ')'; power e]
  moreSummands e = msum [do tchar '-'; e' <- summand
                          moreSummands $ sub alg e e',
                          do es <- some $ do tchar '+'; summand

```

```

        moreSummands $ sum_ alg $ e:es,
        return e]
moreFactors e = msum [do es <- some $ do tchar '*'
                        msum [scalar,factor]
                        moreFactors $ prod alg $ e:es,
                        return e]
power e = msum [do tchar '^'; i <- tint
                return $ expo alg e i,
                return e]
scalar = do i <- tint
          msum [do tchar '*'
                msum [do e <- scalar; return $ scal alg i e,
                    do x <- tidentifier
                        e <- power $ var alg x
                        return $ scal alg i e,
                    do tchar '('; e <- expC alg; tchar ')']
                    return $ scal alg i e],
          power $ con alg i]

```

Die Unterscheidung zwischen Compilern für Ausdrücke, Summanden bzw. Faktoren dient der Berücksichtigung von Operatorprioritäten (+ und – vor \* und ^) 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 $\text{expC}$

Die Compiler-Instanz  $\text{expC}(\text{storeAlg})$  ist korrekt bzgl. der Auswertung arithmetischer Ausdrücke mit  $\text{exp2store}$ , d.h. für alle  $e \in \text{Exp}(\text{String})$  und  $f : \text{Store}(\text{String}) \rightarrow \text{Int}$  gilt:

$$\text{exp2store}(e) = f \Leftrightarrow \text{runST}(\text{expC}(\text{storeAlg}))(\text{showsPrec}(0)(e)[]) = \text{Just}(f, [])$$

(siehe 4.3 und 5.7).

Die Compiler-Instanz  $\text{expC}(\text{codeAlg})$  ist korrekt bzgl. der Übersetzung arithmetischer Ausdrücke mit  $\text{exp2code}$ , d.h. für alle  $e \in \text{Exp}(\text{String})$  und  $cs \in [\text{StackCom}(\text{String})]$  gilt:

$$\text{exp2code}(e) = cs \Leftrightarrow \text{runST}(\text{expC}(\text{codeAlg}))(\text{showsPrec}(0)(e)[]) = \text{Just}(cs, [])$$

(siehe 5.7 und 5.11).

**Aufgabe** Erweitern Sie den Datentyp  $\text{Exp}(x)$ , die Zielsprache  $\text{StackCom}(x)$ , die obigen  $\text{Exp}(x)$ -Algebren und den generischen Compiler  $\text{expC}$  um die ganzzahlige Division.  $\square$

## Comonaden

```
class Functor cm => Comonad cm where
```

```
  extract :: cm a -> a
```

*counit*

```
  (<=<) :: (cm a -> b) -> (cm a -> cm b)
```

*comonadische Extension,  
cobind*

Während die monadische Extension (siehe [Abschnitt 7.4](#))

$$(=<<) : (a \rightarrow m(b)) \rightarrow (m(a) \rightarrow m(b))$$

die – durch  $m$  gegebene – **effekt-erzeugende Kapselung der Ausgabe** einer Funktion  $f : a \rightarrow m(b)$  auf deren Eingabe überträgt und damit solche Funktionen – mittels der Kleisli-Komposition  $<=<$  (s.o.) – komponierbar macht, transportiert dual dazu die comonadische Extension

$$(=<=) : (cm(a) \rightarrow b) \rightarrow (cm(a) \rightarrow cm(b))$$

die – durch  $cm$  gegebene – **kontextabhängige Kapselung der Eingabe** einer Funktion  $f : cm(a) \rightarrow b$  zu deren Ausgabe und macht damit auch solche Funktionen komponierbar:

$$(=<=) :: \text{Comonad } cm \Rightarrow (cm\ b \rightarrow c) \rightarrow (cm\ a \rightarrow b) \rightarrow (cm\ a \rightarrow c)$$

$$g\ =<= f \quad = \quad g \ . \ (f\ <=<)$$

*co-Kleisli-Komposition*

```
duplicate :: Comonad cm => cm a -> cm (cm a)
duplicate = (id <=<=)
```

*Comultiplikation*

Anforderungen an die Instanzen von **Comonad**:

Für alle  $m \in cm(a)$ ,  $f : cm(a) \rightarrow b$  und  $g : cm(b) \rightarrow c$ ,

$$f <=<= (g <=<= cm) = (g . f <=<=) <=<= cm \quad (1)$$

$$\text{extract} <=<= cm = cm \quad (2)$$

$$\text{extract} . (f <=<=) = f \quad (3)$$

$$f <=<= cm = \text{fmap } f \$ \text{id} <=<= cm$$

Ist  $cm$  ein Funktor und gelten die Gleichungen (1)-(3), dann tun das auch die folgenden:

$$(f <=<=) = \text{fmap } f . \text{duplicate} \quad (4)$$

$$\text{fmap } f = (f . \text{extract} <=<=) \quad (5)$$

## Comonadische Funktoren

Die jeweiligen Instanzen der Klasse **Functor** stehen in Abschnitt 7.2.

<pre>instance Comonad Id where     extract = run     f &lt;=&lt;= cm = Id \$ f cm</pre>	<i>Identitätscomonade</i>
<pre>instance Monoid state =&gt; Comonad ((-&gt;) state) where     extract h = h mempty     (f &lt;=&lt;= h) st = f \$ h . mappend st</pre>	<i>Lesercomonaden</i>
<pre>instance Comonad ((,) state) where     extract (_,a) = a     f &lt;=&lt;= p@(st,_) = (st,f p)</pre>	<i>Schreibercomonaden</i>
<pre>instance Comonad (Costate state) where     extract (h:#st) = h st     f &lt;=&lt;= (h:#st) = (f . (:#) h):#st</pre>	<i>Zustandscomonaden</i>
<pre>instance Comonad [ ] where     extract = head</pre>	<i>Listencomonade</i>

$$f \ll= [] = []$$

$$f \ll= s = f \ s : (f \ll= \text{tail } s)$$

Der *cobind*-Operator von  $[]$  erweitert jede Operation  $f : [a] \rightarrow b$  zur Operation  $(f \ll=) : [a] \rightarrow [b]$ . Während  $f(s)$  ein einzelner Wert von  $b$  ist, liefert  $f \ll= s$  die Liste der Werte aller Suffixe von  $s$  unter  $f$ :

$$f \ll= [a_1, \dots, a_n] = [f[a_1, \dots, a_n], f[a_2, \dots, a_n], \dots, f[a_n]]. \quad (6)$$

## Beispiele

$$\text{id } [1..5] \quad \rightsquigarrow \quad [1, 2, 3, 4, 5]$$

$$\text{id } \ll= [1..5] \quad \rightsquigarrow \quad [[1, 2, 3, 4, 5], [2, 3, 4, 5], [3, 4, 5], [4, 5], [5]]$$

$\text{id} \ll= s$  erzeugt aus  $s$  eine Liste der Suffixe von  $s$ .

$$\text{sum } [1..8] \quad \rightsquigarrow \quad 36$$

$\text{sum}(s)$  berechnet die Summe der Elemente von  $s$ .

$$\text{sum } \ll= [1..8] \quad \rightsquigarrow \quad [36, 35, 33, 30, 26, 21, 15, 8]$$

$\text{sum} \ll= s$  erzeugt aus  $s$  eine Liste, die an jeder Position  $i$  die Summe der Elemente von  $\text{drop}(i)(s)$  enthält.



Nach Definition von *extract* in der Listencomonade folgt

$$\text{head}(f \ll s) = \text{extract}(f \ll s) = f(s)$$

für alle  $f : [a] \rightarrow b$  und  $s \in [a]$  aus Gleichung (2).

## Nochmal Listen mit Zeiger auf ein Element (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:@)) [0..length s-1]:@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 Listenfunctor in  $ListPos(a)$  durch den (semantisch äquivalenten) Leserfunctor  $(\rightarrow)(Int)$  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 jede Operation  $f : [a] \times \text{Int} \rightarrow b$  zur Operation  $(f \ll=) : [a] \times \text{Int} \rightarrow [b] \times \text{Int}$ . Im Gegensatz zum *cobind*-Operator von `[ ]`, der aus  $s \in [a]$  eine Liste erzeugt, an deren Position  $k$  ein nur von  $\text{drop}(k)(s)$  abhängiger Wert steht, schreibt  $f \ll=(s, i)$  an deren Position  $k$  einen von  $k$  und möglicherweise von ganz  $s$  abhängigen Wert:

$$f \ll=(s, i) = ([f(s, 0), f(s, 1), \dots, f(s, \text{length}(s) - 1)], i). \quad (7)$$

## Beispiele

```

prefixSum, suffixSum, neighbSum :: ListPos Int -> Int
prefixSum (s:@0) = s!!0
prefixSum (s:@i) = prefixSum (s:@i-1)+s!!i
suffixSum (s:@i) = if i >= length s then 0
                  else s!!i+suffixSum (s:@i+1)
neighbSum (s:@0) = s!!0+s!!1
neighbSum (s:@i) = if i+1 < length s then s!!(i-1)+s!!i+s!!(i+1)
                  else s!!(i-1)+s!!i

```

$\text{prefixSum}(s:@i)$  berechnet die Summe der Elemente von  $\text{take}(i+1)(s)$ .

$\text{suffixSum}(s:@i)$  berechnet, wie  $\text{sum}(\text{drop}(i)(s))$ , die Summe der Elemente von  $\text{drop}(i)(s)$ .

$\text{neighbSum}(s:@i)$  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)* erzeugt aus *s* eine Liste, die an jeder Position *i* die Summe der ersten *i* + 1 Elemente von *s* enthält.

*list(neighbSum<=<=s:@i)* erzeugt aus *s* eine Liste, die an jeder Position *i* die Summe von *s*!!*i* und den ein bzw. zwei Nachbarn von *s*!!*i* enthält.

## Bäume, comonadisch

Im Folgenden übertragen wir die comonadische Behandlung von Listen mit Zeiger auf binäre und beliebige Bäume mit Zeiger auf einen Knoten.

```
instance Comonad Bintree where
  extract (Fork a _ _) = a
  f <=<= Empty          = Empty
  f <=<= t@(Fork _ t u) = Fork (f t) (f <=<= t) $ f <=<= u
```

```
btree1 :: Bintree Int                                     (siehe Abschnitte 5.4 und 5.6)
btree1 = Fork 6 (Fork 7 (Fork 11 (leaf 55) $ leaf 33) $ Empty)
```

`$ leaf 9`  $\rightsquigarrow$  `6(7(11(55,33),),9)`

`foldBintree btree1 :: Int`  $\rightsquigarrow$  `121`

*foldBintree(t)* berechnet die Summe aller Knoteneinträge von *t*.

`foldBintree <=< btree1 :: Bintree Int`  $\rightsquigarrow$  `121(106(99(55,33),),9)`

*foldBintree<=<t* erzeugt aus *t* einen Baum, in dem jeder Knoten *node* mit der Summe der Einträge des Teilbaums markiert ist, dessen Wurzel mit *node* übereinstimmt.

```
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]
 $\rightsquigarrow$  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 (\a as -> a+sum as) tree1`  $\rightsquigarrow$  `146`

*foldTree( $\lambda a.\lambda as.a+sum\ as$ )(t)* berechnet die Summe aller Knoteneinträge von *t*.

```
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]]
```

$foldTree(\lambda a.\lambda as.a+sum\ as)<=<=t$  erzeugt aus  $t$  einen Baum, in dem jeder Knoten  $node$  mit der Summe der Einträge des Teilbaums markiert ist, dessen Wurzel mit  $node$  übereinstimmt.

```
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 ops1 tree2  ~>  -317
```

$foldTree(ops)(t)$  faltet den Baum  $t$  zu einem Wert gemäß der durch  $ops$  gegebenen Interpretation seiner Knotenmarkierungen.

```
foldTree ops1 <=<= tree2  ~>  F (-317) [F (-330) [V 5,V (-66)],V 13]
```

$foldTree(ops)<=<=t$  erzeugt aus  $t$  einen Baum, in dem jeder Knoten  $node$  mit dem Wert der Faltung des Teilbaums gemäß  $ops$  markiert ist, dessen Wurzel mit  $node$  übereinstimmt.

Der Übergang von der *Tree*- zur *TreeNode*-Comonade ist genauso motiviert wie der Übergang 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

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]
```

$\text{nodeTree}(t)[]$  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:&)) (nodeTree t []):&node
```

```

prefixSumT :: TreeNode Int -> Int
prefixSum3 (t:&[])    = root t
prefixSum3 (t:&node) = prefixSum3 (t:&init node)+label t node

```

$prefixSumT(t:&node)$  berechnet die Summe der Markierungen des Knotens  $node$  und seiner Vorgänger (siehe Abschnitt 5.9).

```

tree $ prefixSumT <=& 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]]

```

$tree(prefixSumT<=&t:&node)$  erzeugt aus  $t$  einen Baum, in dem jeder Knoten  $node'$  mit der Summe der Markierungen seiner Vorgänger einschließlich  $node'$  markiert ist.

## 10 Semantik und Verifikation funktionaler Programme

Jeder Aufruf eines Haskell-Programms ist ein Term, der aus Standard- und selbstdefinierten Funktionen zusammengesetzt ist. Demzufolge besteht die Ausführung von Haskell-Programmen in der **Auswertung funktionaler Ausdrücke**. Da sowohl Konstanten als auch Funktionen rekursiv definiert werden können, kann es passieren, dass die Auswertung eines Terms – genauso wie die Ausführung eines imperativen Programms – nicht terminiert. Das kann und soll grundsätzlich auch nicht verhindert werden. Z.B. muss die Funktion, die den Interpreter einer Programmiersprache mit Schleifenkonstrukten darstellt, auch im Fall einer unendlichen Zahl von Schleifendurchläufen eine Semantik haben.

Selbstverständlich spielt die **Auswertungsstrategie**, also die Auswahl des jeweils nächsten Auswertungsschritts eine wichtige Rolle, nicht nur bezüglich des Ergebnisses der Auswertung, sondern auch bei der Frage, ob überhaupt ein Ergebnis erreicht wird. So kann mancher Term mit der einen Strategie in endlicher Zeit ausgewertet werden, mit einer anderen jedoch nicht. Und stellt der Term eine (partielle) Funktion dar, dann kann können sich die Ergebnisse seiner Auswertung mit verschiedenen Strategien in der Größe des Definitionsbereiches der Funktion unterscheiden.



Um alle diese Fragen präzise beantworten und Auswertungsstrategien miteinander vergleichen zu können, benötigt man eine vom Auswertungsprozess, der **operationellen Semantik**, unabhängige **denotationelle Semantik** von Termen, egal ob diese Konstanten oder Funktionen darstellen.

Ein Haskell-Programm besteht i.w. aus Gleichungen. Diese beschreiben zunächst lediglich Anforderungen an die in ihnen auftretenden Konstanten oder Funktionen. Deren denotationelle Semantik besteht in Lösungen der Gleichungen. Lösungen erhält man aber nur in bestimmten mathematischen Strukturen wie z.B. CPOs (siehe Abschnitt 6.1).

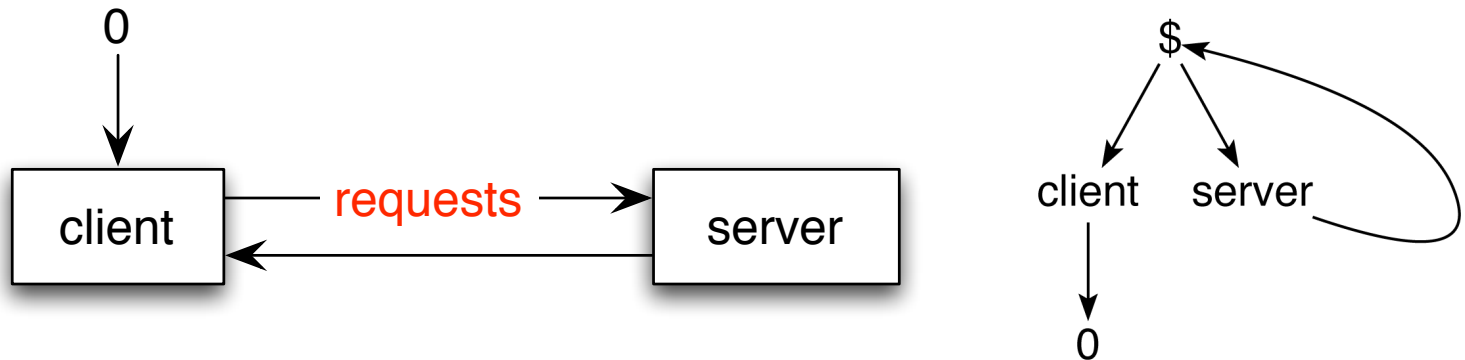
## Das relationale Berechnungsmodell

In der **logischen** oder **relationalen Programmierung** ist das Lösen von Gleichungen und anderen prädikatenlogischen Formeln das eigentliche Berechnungsziel: Die Ausführung eines logischen Programms besteht in der schrittweisen Konstruktion von Belegungen der freien Formelvariablen durch Werte, welche die Formel gültig machen. Kurz gesagt, Formeln werden zu sie erfüllende Belegungen ausgewertet. Sie werden schrittweise konstruiert, indem die auszuwertende Formel  $\varphi$  mit Gleichungen der Form  $x = t$  konjunktiv verknüpft wird.  $x = t$  beschreibt eine Belegung der Variablen  $x$  durch den aus Konstruktoren und Variablen bestehenden (!) Term  $t$ .

Ist die Auswertung von  $\varphi$  erfolgreich, dann endet sie mit einer Konjunktion  $\psi$  der Form  $x_1 = t_1 \wedge \dots \wedge x_n = t_n$ .  $\varphi$ ,  $\psi$  und die von den einzelnen Auswertungsschritten erzeugten Zwischenergebnisse bilden eine **logische Reduktion**, also eine Folge von Formeln, die von ihren jeweiligen Nachfolgern in der Folge impliziert werden. Damit ist klar, dass  $\psi$  Belegungen repräsentiert, die  $\varphi$  erfüllen. Logische Reduktionen und die Regeln, die sie erzeugen (*Simplifikation*, *Co/Resolution* und *relationale Co/Induktion*), spielen in der Verifikation logisch-algebraischer Modelle eine entscheidende Rolle (siehe [From Modal Logic to \(Co\)Algebraic Reasoning](#)).

Den Auswertung eines Terms  $t$  durch Anwendung der Gleichungen eines Haskell-Programms entspricht einer logischen Reduktion der Formel  $x = t$ .

### Beispiel client/server



```

client :: a -> [b] -> [a]
client a s = a:client (f b) s' where b:s' = s

server :: [a] -> [b]
server (a:s) = g a:server s

requests :: [a]
requests = client 0 $ server requests

```

Wir wollen den Term  $take(3)(requests)$  auswerten und konstruieren dazu eine logische Reduktion der Gleichung  $s = take(3)(requests)$  mit der freien Variable  $s$ . Jeder Reduktionsschritt besteht in der Anwendung einer der obigen Gleichungen von links nach rechts, wobei das jeweilige Ergebnis im Fall der **client**-Gleichung mit der entsprechenden Instanz der lokalen Definition  $b : s' = s$  konjunktiv verknüpft wird. Die Variablen  $b$  und  $s'$  sind implizit existenzquantifiziert und müssen daher werden bei jeder Anwendung der **client**-Gleichung durch neue Variablen ersetzt werden ( $a_0, \dots, a_5$  bzw.  $s_0, \dots, s_5$ ). Für jeden Reduktionsschritt  $\varphi \rightarrow \psi$  gilt:

$$\exists a_0, \dots, a_5, s_0, \dots, s_5 : \psi \quad \Rightarrow \quad \exists a_0, \dots, a_5, s_0, \dots, s_5 : \varphi.$$

Sei  $f = (*2)$  und  $g = (+1)$ . Dann lautet die gesamte Reduktion wie folgt:

```

s = take 3 requests
→ s = take 3 $ client 0 $ server requests
→ s = take 3 $ 0:client (f a0) s0 ∧ a0:s0 = server requests
→ s = 0:take 2 (client (f a0) s0) ∧ a0:s0 = server requests
→ s = 0:take 2 (f a0:client (f a1) s1) ∧ a1:s1 = s0 ∧
  a0:s0 = server requests
→ s = 0:f a0:take 1 (client (f a1) s1) ∧ a1:s1 = s0 ∧
  a0:s0 = server requests
→ s = 0:f a0:take 1 (f a1:client (f a2) s2) ∧ a2:s2 = s1 ∧
  a1:s1 = s0 ∧ a0:s0 = server requests
→ s = 0:f a0:f a1:take 0 (client (f a2) s2) ∧ a2:s2 = s1 ∧
  a1:s1 = s0 ∧ a0:s0 = server requests
→ s = 0:f a0:f a1:[] ∧ a2:s2 = s1 ∧ a1:s1 = s0 ∧
  a0:s0 = server requests
→ s = [0,f a0,f a1] ∧ a2:s2 = s1 ∧ a1:s1 = s0 ∧
  a0:s0 = server requests
→ s = [0,f a0,f a1] ∧ a2:s2 = s1 ∧ a1:s1 = s0 ∧
  a0:s0 = server $ client 0 $ server requests

```

$\rightarrow s = [0, f\ a0, f\ a1] \wedge a2:s2 = s1 \wedge a1:s1 = s0 \wedge$   
 $a0:s0 = \text{server } \$\ 0:\text{client } (f\ a3)\ s3 \wedge a3:s3 = \text{server requests}$

$\rightarrow s = [0, f\ a0, f\ a1] \wedge a2:s2 = s1 \wedge a1:s1 = s0 \wedge$   
 $a0:s0 = g\ 0:\text{server } (\text{client } (f\ a3)\ s3) \wedge a3:s3 = \text{server requests}$

$\rightarrow s = [0, f\ a0, f\ a1] \wedge a2:s2 = s1 \wedge a1:s1 = s0 \wedge$   
 $a0:s0 = 1:\text{server } (\text{client } (f\ a3)\ s3) \wedge a3:s3 = \text{server requests}$

$\rightarrow s = [0, f\ 1, f\ a1] \wedge a2:s2 = s1 \wedge a1:s1 = s0 \wedge$   
 $s0 = \text{server } \$\ \text{client } (f\ a3)\ s3 \wedge a3:s3 = \text{server requests}$

$\rightarrow s = [0, 2, f\ a1] \wedge a2:s2 = s1 \wedge$   
 $a1:s1 = \text{server } \$\ \text{client } (f\ a3)\ s3 \wedge a3:s3 = \text{server requests}$

$\rightarrow s = [0, 2, f\ a1] \wedge a2:s2 = s1 \wedge$   
 $a1:s1 = \text{server } \$\ f\ a3:\text{client } (f\ a4)\ s4 \wedge a4:s4 = s3 \wedge$   
 $a3:s3 = \text{server requests}$

$\rightarrow s = [0, 2, f\ a1] \wedge a2:s2 = s1 \wedge$   
 $a1:s1 = g\ (f\ a3):\text{server } (\text{client } (f\ a4)\ s4) \wedge a4:s4 = s3 \wedge$   
 $a3:s3 = \text{server requests}$

$\rightarrow s = [0, 2, f\ (g\ (f\ a3)) ] \wedge a2:s2 = s1 \wedge$   
 $s1 = \text{server } \$\ \text{client } (f\ a4)\ s4 \wedge a4:s4 = s3 \wedge$   
 $a3:s3 = \text{server requests}$

$\rightarrow s = [0, 2, f\ (g\ (f\ a3)) ] \wedge a2:s2 = \text{server } \$\ \text{client } (f\ a4)\ s4 \wedge$   
 $a4:s4 = s3 \wedge a3:s3 = \text{server requests}$

$$\begin{aligned}
&\rightarrow s = [0,2,f (g (f a3))] \wedge a2:s2 = \text{server } \$ \text{ client } (f a4) s4 \wedge \\
&\quad a4:s4 = s3 \wedge a3:s3 = \text{server } \$ \text{ client } 0 \$ \text{ server requests} \\
&\rightarrow s = [0,2,f (g (f a3))] \wedge a2:s2 = \text{server } \$ \text{ client } (f a4) s4 \wedge \\
&\quad a4:s4 = s3 \wedge a3:s3 = \text{server } \$ 0:\text{client } (f a5) s5 \wedge a5:s5 = s4 \\
&\rightarrow s = [0,2,f (g (f a3))] \wedge a2:s2 = \text{server } \$ \text{ client } (f a4) s4 \wedge \\
&\quad a4:s4 = s3 \wedge a3:s3 = g 0:\text{server } \$ \text{ client } (f a5) s5 \wedge a5:s5 = s4 \\
&\rightarrow s = [0,2,f (g (f a3))] \wedge a2:s2 = \text{server } \$ \text{ client } (f a4) s4 \wedge \\
&\quad a4:s4 = s3 \wedge a3:s3 = 1:\text{server } (\text{client } (f a5) s5) \wedge a5:s5 = s4 \\
&\rightarrow s = [0,2,f (g (f 1))] \wedge a2:s2 = \text{server } \$ \text{ client } (f a4) s4 \wedge \\
&\quad a4:s4 = s3 \wedge s3 = \text{server } \$ \text{ client } (f a5) s5 \wedge a5:s5 = s4 \\
&\rightarrow s = [0,2,6] \wedge a2:s2 = \text{server } \$ \text{ client } (f a4) s4 \wedge \\
&\quad a4:s4 = \text{server } \$ \text{ client } (f a5) s5 \wedge a5:s5 = s4
\end{aligned}$$

Die logische Reduktion zeigt die prinzipielle Vorgehensweise bei der Auswertung eines Terms durch Anwendung von Gleichungen eines Haskell-Programms. Da funktionale Programme, als prädikatenlogische Formeln betrachtet, nur ein einziges Relationssymbol enthalten, nämlich das Gleichheitssymbol, können sie – wie in den folgenden Abschnitten ausgeführt wird – selbst als Terme repräsentiert werden. Logische Reduktionen können dann durch Termreduktionen simuliert werden. Das spart Platz und Zeit, insbesondere weil die häufige Erzeugung neuer Variablen (siehe obiges Beispiel) überflüssig wird.

## Das funktionale Berechnungsmodell

An die Stelle der obigen logischen Reduktion

$$s = \textit{take}(3)(\textit{requests}) \rightarrow \dots \rightarrow s = [0, 2, 6] \wedge \varphi(a_2, a_4, a_5, s_2, s_4, s_5) \quad (1)$$

tritt eine **Termreduktion**:

$$\textit{take}(3)(\textit{requests}) \rightarrow \dots \rightarrow [0, 2, 6] \quad (2)$$

Während die logische Reduktion (1) als umgekehrte **Implikation** interpretiert wird:

$$\begin{aligned} s &= \textit{take}(3)(\textit{requests}) \\ \iff s &= [0, 2, 6] \wedge \exists a_2, a_4, a_5, s_2, s_4, s_5 : \varphi(a_2, a_4, a_5, s_2, s_4, s_5), \end{aligned} \quad (3)$$

ist die Semantik einer Termreduktion  $t \rightarrow^* t'$  durch die **Äquivalenz** der Terme  $t$  und  $t'$  bzgl. ihrer Interpretationen in einer bestimmten  $\Sigma$ -Algebra  $A$  gegeben (s.u.).

Um (1) in (2) umwandeln zu können, müssen wir zunächst die in (1) verwendeten Konstanten- und Funktionsdefinitionen in  $\lambda$ - und  $\mu$ -Abstraktionen transformieren.  $\lambda$ -Abstraktionen kennen wir schon aus dem Abschnitt **Funktionen**. Eine  **$\mu$ -Abstraktion** <sup>\*\*\*\*</sup>  $\mu p.t$  bezeichnet die kleinste Lösung der Gleichung  $p = u$ . Sie existiert, wenn  $t$  in einem CPO interpretiert wird (siehe Abschnitt 6.1).

Wir müssen zunächst grundlegende Begriffe, auf denen das Rechnen mit Termen basiert, präzisieren.

## Terme und Substitutionen

Sei  $\Sigma = (S, F, B\Sigma)$  eine Signatur,  $B\Sigma$  die Menge der Sorten von  $B\Sigma$  und  $C \subseteq F$  eine  $(S \setminus B\Sigma)$ -sortige Menge von Konstruktoren derart, dass für alle  $s \in S \setminus B\Sigma$ ,  $C_s$  nicht leer ist (siehe [Pad1], Kapitel 2, oder [Pad2], Kapitel 11).

Eine  **$S$ -sortige Menge**  $A$  ist eine Mengenfamilie:  $A = \{A_s \mid s \in S\}$ . Man sagt “Mengenfamilie” und nicht, was die Schreibweise nahelegt, “Menge von Mengen”, um Antinomien (logische Widersprüche) wie die Menge *aller* Mengen zu vermeiden.

Die Menge  $types(S)$  der **Typen über  $S$**  ist induktiv definiert:

- $S \subseteq types(S)$ ,
- $e_1, \dots, e_n \in types(S) \implies e_1 \times \dots \times e_n \in types(S)$ ,
- $e, e' \in types(S) \implies e \rightarrow e' \in types(S)$ .

Die Interpretation von  $S$  in einer  $\Sigma$ -Algebra  $A$  wird wie folgt auf  $types(S)$  fortgesetzt:

$$\begin{aligned} A_{e_1 \times \dots \times e_n} &=_{def} A_{s_1} \times \dots \times A_{s_n}, \\ A_{e \rightarrow e'} &=_{def} A_e \rightarrow A_{e'}. \end{aligned}$$



Sei  $X$  eine  $types(S)$ -sortige Menge. Die  $S$ -sortige Menge  $P(X, C)$  der  $\Sigma$ -**Muster über  $X$  und  $C$**  und die  $types(S)$ -sortige Menge  $T_\Sigma(X, C)$  der  $\Sigma$ -**Terme** sind induktiv definiert:

- Für alle  $e \in types(S)$ ,  $X_e \subseteq P(T, X)_e \cap T_\Sigma(X, C)_e$ .
- Für alle  $e = e_1 \times \cdots \times e_n \in types(S)$ ,  $p_1 \in P(X, C)_{e_1}, \dots, p_n \in P(X, C)_{e_n}$ ,  

$$\forall 1 \leq i < j \leq n : var(p_i) \cap var(p_j) = \emptyset \implies (p_1, \dots, p_n) \in P(X, C)_e.$$
- Für alle  $f : e \rightarrow s \in C$  and  $p \in P(X, C)_e$ ,  $f(p) \in P(X, C)_s$ .
- Für alle  $f : e \rightarrow s \in F$ ,  $f \in T_\Sigma(X, C)_{e \rightarrow s}$ .
- Für alle  $e = e_1 \times \cdots \times e_n \in types(S)$ ,  $t_1 \in T_\Sigma(X, C)_{e_1}, \dots, t_n \in T_\Sigma(X, C)_{e_n}$ ,  
 $(t_1, \dots, t_n) \in T_\Sigma(X, C)_e$ .
- Für alle  $e, e' \in types(S)$ ,  $t \in T_\Sigma(X, C)_{e \rightarrow e'}$  und  $u \in T_\Sigma(X, C)_e$ ,  $t(u) \in T_\Sigma(X, C)_{e'}$ .
- Für alle  $e, e' \in types(S)$ ,  $p \in P(X, C)_e$  und  $t \in T_\Sigma(X, C)_{e'}$ ,  $\lambda p.t \in T_\Sigma(X, C)_{e \rightarrow e'}$ .
- Für alle  $e \in types(S)$ ,  $p \in P(X, C)_e$  und  $t \in T_\Sigma(X, C)_e$ ,  $\mu p.t \in T_\Sigma(X, C)_e$ .

Sei  $t = \lambda p.u$  oder  $t = \mu p.u$ .  $\lambda p$  bzw.  $\mu p$  nennt man den **Kopf** und  $u$  den **Rumpf von  $t$** . Die Variablen des Kopfes heißen **gebunden in  $t$** , die restlichen Variablen sind **frei in  $t$** .

Für alle  $t \in T_\Sigma(X, C)$  ist  $var(t)$  die Menge aller Variablen von  $t$  und  $free(t)$  die Menge aller Variablen von  $t$ , die in keiner in  $t$  enthaltenen Abstraktion gebunden sind.

Sei  $s \in BS$ .  $x \in X_s$  heißt  **$\Sigma$ -primitiv**.  $BT_\Sigma(X, C)$  bezeichnet die Menge der  $\Sigma$ -Terme über  $X$  und  $C$ , deren freie Variablen  $\Sigma$ -primitiv sind.

Seien  $A$  und  $B$   $S$ -sortige Mengen. Eine  **$S$ -sortige Funktion**  $f : A \rightarrow B$  ist eine Menge von Funktionen:  $f = \{f_s : A_s \rightarrow B_s \mid s \in S\}$ .

Eine  $S$ -sortige Funktion  $\sigma : X \rightarrow T_\Sigma(X, C)$  heißt **Substitution**.  $\sigma$  wird wie folgt zur Funktion  $\sigma^* : T_\Sigma(X, C) \rightarrow T_\Sigma(X, C)$  fortgesetzt:

$$\begin{aligned}
\sigma^*(x) &= \sigma(x) && \text{für alle } x \in X \\
\sigma^*(f) &= f && \text{für alle } f \in F \\
\sigma^*((t_1, \dots, t_n)) &= (\sigma^*(t_1), \dots, \sigma^*(t_n)) \\
\sigma^*(t(u)) &= \sigma^*(t)(\sigma^*(u)) \\
\sigma^*(\lambda p.t) &= \lambda \rho_{var(p)}^*(p). \sigma_{var(p)}^*(t) \\
\sigma^*(\mu p.t) &= \mu \rho_{var(p)}^*(p). \sigma_{var(p)}^*(t)
\end{aligned}$$

Für alle  $V \subseteq X$  sind  $\rho_V : X \rightarrow X$  bzw.  $\sigma_V : X \rightarrow T_\Sigma(X, C)$  wie folgt definiert:

$$\begin{aligned}
\rho_V(x) &=_{def} \begin{cases} x' & \text{falls } x \in V \cap \text{free}(\sigma(\text{free}(t))) \text{ und } x' \notin V \cap \text{free}(\sigma(\text{free}(t))), \\ x & \text{sonst,} \end{cases} \\
\sigma_V(x) &=_{def} \begin{cases} x' & \text{falls } x \in V, \\ \sigma(x) & \text{sonst.} \end{cases}
\end{aligned}$$

Die **Variablenumbenennung**  $\rho_V$  stellt sicher, dass die Instanziierung des Rumpfes einer Abstraktion keine zusätzlichen Vorkommen ihrer gebundenen Variablen in den Term einführt. Im klassischen  **$\lambda$ -Kalkül** (in dem es anstelle beliebiger Muster nur einzelne Variablen gibt) spricht man von  **$\alpha$ -Konversion**.

$\sigma^*(t)$  ist die  **$\sigma$ -Instanz von  $t$** . Aus Instanziierungen ergibt sich die **Subsumptionsordnung**  $\leq$  auf Termen:

$$u \leq t \iff_{def} t \text{ ist eine Instanz von } u.$$

In entsprechender Weise müssen manchmal gebundene Variablen quantifizierter prädikatenlogischer Formeln vor der Substitution ihrer freien Variablen umbenannt werden.

Für alle  $\sigma : X \rightarrow T_\Sigma(X, C)$ ,  $t, u \in T_\Sigma(X, C)$  und  $x \in X$  sind  $\sigma[u/x] : X \rightarrow T_\Sigma(X, C)$  bzw.  $t[u/x] \in T_\Sigma(X, C)$  wie folgt definiert:

$$\sigma[u/x](z) = \begin{cases} u & \text{falls } z = x, \\ \sigma(z) & \text{sonst,} \end{cases}$$

$$t[u/x] = id[u/x]^*(t).$$

## Termreduktionen

$t \rightarrow t'$  ist eine **Reduktionsregel**, falls  $t$  und  $t'$  Terme mit  $free(t') \subseteq free(t)$  sind.

Sei  $Red$  eine Menge von Reduktionsregeln. Die **Reduktionsrelation**  $\rightarrow_{Red} \subseteq BT_{\Sigma}(X, C)^2$  ist wie folgt definiert:

$$t \rightarrow_{Red} t' \iff_{def} \left\{ \begin{array}{l} \exists v \rightarrow v' \in Red, u \in T_{\Sigma}(X, C), x \in free(u), \\ \sigma : X \rightarrow T_{\Sigma}(X, C) : t = u[\sigma^*(v)/x] \wedge t' = u[\sigma^*(v')/x]. \end{array} \right.$$

$v$  und  $t'$  nennt man einen **Red-Redex** bzw. ein **Red-Redukt von  $t$** .

## Reduktionsregeln für einige Standardfunktionen

$x + 0$	$\rightarrow x$	
$x * 0$	$\rightarrow 0$	
$True \ \&\& \ x$	$\rightarrow x$	
$False \ \&\& \ x$	$\rightarrow False$	
$\pi_i(x_1, \dots, x_n)$	$\rightarrow x_i$	$1 \leq i \leq n$
$head(x : xs)$	$\rightarrow x$	
$tail(x : xs)$	$\rightarrow xs$	
$if \ True \ then \ x \ else \ y$	$\rightarrow x$	
$if \ False \ then \ x \ else \ y$	$\rightarrow y$	

Regeln für Standardfunktionen werden im klassischen  **$\lambda$ -Kalkül**  **$\delta$ -Regeln** genannt.

## Regeln für $\lambda$ -Applikationen

Sei  $p \in P(X, C)$ ,  $t, u \in T_\Sigma(X, C)$ ,  $x \in X$ ,  $f \in F$  und  $\sigma : X \rightarrow T_\Sigma(X, C)$ .

$$\begin{aligned} \lambda x. f(x) &\rightarrow f \\ (\lambda p. t)(p\sigma) &\rightarrow t\sigma \\ (\lambda p. t)(u) &\rightarrow \text{fail} && \text{falls } u \in P(X, C) \text{ und } p \not\leq u \\ (\text{if } b \text{ then } x \text{ else } y)(z) &\rightarrow \text{if } b(z) \text{ then } x(z) \text{ else } y(z) \\ \dots & \end{aligned}$$

Die ersten beiden Regeln heißen im klassischen  $\lambda$ -Kalkül  **$\eta$ -Regel** bzw.  **$\beta$ -Regel**.

Die Konstante *fail* ähnelt der Konstanten **Nothing** einer Instanz des Haskell-Datentyps **Maybe**. Der Operator **mplus** der Typklasse **MonadPlus** war für **Maybe** wie folgt definiert:

```
m `mplus` m' = case m of Nothing -> m'; _ -> m
```

Umgekehrt werden bei der Reduktion von **case**-Ausdrücken und induktiv definierten Funktionen in  $\lambda$ -Ausdrücke (s.u.) Ausdrücke der Form  $e \parallel e'$  eingeführt. Die Regeln für den Operator  $\parallel$  entsprechen der Definition von **mplus**: Reduktionsregeln: Sei  $x, x_1, \dots, x_n, z \in X$  und  $p \in P(X, C)$ .

$$\begin{aligned} \text{fail} \parallel x &\rightarrow x \\ p \parallel x &\rightarrow p && \text{falls } p \neq \text{fail} \\ (x_1 \parallel \dots \parallel x_n)(z) &\rightarrow x_1(z) \parallel \dots \parallel x_n(z) \end{aligned}$$

## Reduktionsregel für case-Ausdrücke

$$\text{case } t \text{ of } \left. \begin{array}{l} p_1 \mid b_1 \rightarrow t_1 \\ \vdots \\ p_n \mid b_n \rightarrow t_n \end{array} \right\} \rightarrow \left\{ \begin{array}{l} (\lambda p_1. \text{if } b_1 \text{ then } t_1 \text{ else fail})(t) \\ \vdots \\ \parallel (\lambda p_n. \text{if } b_n \text{ then } t_n \text{ else fail})(t) \end{array} \right.$$

## Korrektheit von Termreduktionen

Sei  $A$  eine  $\Sigma$ -Algebra derart, dass  $C$  aus Konstruktoren von  $A$  besteht (siehe [Pad1], Kapitel 2, oder [Pad2], Kapitel 12).

Eine  $S$ -sortige Funktion  $\beta : X \rightarrow A$  heißt **Variablenbelegung** in  $A$ . Für alle Variablenbelegungen  $\gamma : X \rightarrow A$  und  $x \in X$  ist  $\beta[\gamma/V] : X \rightarrow A$  wie folgt definiert:

$$\beta[\gamma/V](x) = \begin{cases} \gamma(x) & \text{falls } x \in V, \\ \beta(x) & \text{sonst.} \end{cases}$$

Die von  $\beta$  abhängige **Auswertungsfunktion**  $\beta^* : T_\Sigma(X, C) \rightarrow A$  ist induktiv definiert:

- Für alle  $x \in X$ ,  $\beta^*(x) = \beta(x)$ .
- Für alle  $f \in F$ ,  $\beta^*(f) = f^A$ .
- Für alle  $t = (t_1, \dots, t_n) \in T_\Sigma(X, C)$ ,  $\beta^*(t) = (\beta^*(t_1), \dots, \beta^*(t_n))$ .
- Für alle  $t(u) \in T_\Sigma(X, C)$ ,  $\beta^*(t(u)) = \beta^*(t)(\beta^*(u))$ .

- Für alle  $e \in \text{types}(s)$ ,  $p \in P(X, C)_e$ ,  $t = \lambda p.u \in T_\Sigma(X, C)$  und  $a \in A_e$ ,

$$\beta^*(t)(a) = \begin{cases} \beta[\gamma/\text{var}(p)]^*(u) & \text{falls } \exists \gamma : X \rightarrow A : a = \gamma^*(p), \\ \text{fail} & \text{sonst.} \end{cases}$$

- Für alle  $t = \mu p.u \in T_\Sigma(C, X)$ ,  $\beta^*(t) = \beta[\gamma/\text{var}(p)]^*(u)$ , wobei  $\gamma : X \rightarrow A$  die kleinste Variablenbelegung mit  $\gamma^*(p) = \gamma^*(u)$  ist (siehe [Partiell-rekursive Funktionen](#)).

$R \subseteq T_\Sigma(X, C)^2$  heißt **korrekt bzgl.**  $A$ , falls für alle  $(t, t') \in R$ ,  $\beta : X \rightarrow A$  und  $V \subseteq X$

$$\beta^*(t) = \beta^*(t').$$

Ist  $Red$  korrekt bzgl.  $A$ , dann ist auch  $\rightarrow_{Red}^*$  korrekt bzgl.  $A$ .

## Schemata zur Definition partieller Funktionen

Wir betrachten das folgende Schema einer durch  $k$  Gleichungen definierten Funktion  $f : A \rightarrow 1 + B$ :

$$\begin{aligned} f(p_1) = t_1 & \Leftarrow \bigwedge_{i=1}^{n_1} p_{1i} = f(t_{1i}) \wedge \varphi_1, \\ & \vdots \\ f(p_k) = t_k & \Leftarrow \bigwedge_{i=1}^{n_k} p_{ki} = f(t_{ki}) \wedge \varphi_k \end{aligned}$$

mit Mustern  $p_1, \dots, p_k, p_{11}, \dots, p_{kn_k}$ , Termen  $t_1, \dots, t_k, t_{11}, \dots, t_{kn_k}$  und Booleschen Ausdrücken  $\varphi_1, \dots, \varphi_k$  derart, dass für alle  $1 \leq i \leq k$ ,  $1 \leq j \leq n_i$ ,  $V_0 = \text{var}(p_i)$  und

$V_j = V_{j-1} + \text{var}(p_{ij})$  Folgendes gilt:

$$\text{var}(t_i) \cup \text{var}(\varphi_i) \subseteq V_{n_i}, \quad \text{var}(t_{ij}) \subseteq V_{j-1} \quad \text{und} \quad f \notin \text{op}(t_i) \cup \text{op}(t_{ij}).$$

$f$  kann wie folgt monadisch implementiert werden:

```
f :: A -> Maybe B
f p_1 = do p_11 <- f t_11
          :
          p_1n_1 <- f t_1n_1
          guard $ phi_1
          Just t_1
:
f p_k = do p_k1 <- f t_k1
          :
          p_kn_k <- f t_kn_k
          guard $ phi_k
          Just t_k
f _ = Nothing
```

Die letzte Gleichung kann entfallen, wenn die Muster  $p_1, \dots, p_k$  alle Elemente von  $A$  abdecken.



Mit Hilfe der parallelen monadischen Komposition *msum* lässt sich *f* noch kompakter definieren (siehe Kapitel 7):

```
f x = msum [do p_1 <- Just x
              p_11 <- f t_11
              :
              p_1n_1 <- f t_1n_1
              guard $ phi_1
              Just t_1,
            :
            do p_k <- Just x
              p_k1 <- f t_k1
              :
              p_kn_k <- f t_kn_k
              guard $ phi_k
              Just t_k]
```

Liegen alle Werte in *B*, dann ist *f* total und kann mit Hilfe lokaler Definitionen implementiert werden:

```
f p_1 | phi_1 = t_1 where p_11 = f t_11
      :
```

```

                                p_1n_1 = f t_1n_1
      :
f p_k | phi_k = t_k where p_k1 = f t_k1
                                :
                                p_kn_k = f t_kn_k

```

Mit  $\lambda$ -Ausdrücken anstelle lokaler Definitionen (siehe Kapitel 3):

```

f p_1 | phi_1 = (\p_1n_1 -> ...
                -> (\p_11 -> t_1)(f t_11) ... )(f t_1n_1)
      :
f p_k | phi_k = (\p_kn_k -> ...
                -> (\p_k1 -> t_k)(f t_k1) ... )(f t_kn_k)

```

Mit einem einzigen case-Ausdruck:

```

f = \case p_1 | phi_1
      -> (\p_1n_1 -> ...
        -> (\p_11 -> t_1)(f t_11) ... )(f t_1n_1)
      :
      p_k | phi_k
        -> (\p_kn_k -> ...

```

$\rightarrow (\backslash p_{k1} \rightarrow t_k)(f\ t_{k1}) \dots )(f\ t_{kn_k})$

## Beispiel Listenrevertierung mit Palindromtest

```
revEq :: Eq a => [a] -> [a] -> ([a],Bool)
revEq (x:s1) (y:s2) = (r++[x],x==y && b) where (r,b) = revEq s1 s2
revEq _ _          = ([],True)
```

Die folgende Version vermeidet Listenkonkatenationen:

```
revEqI :: Eq a => [a] -> [a] -> [a] -> ([a],Bool)
revEqI (x:s1) (y:s2) acc = (r,x==y && b)
                        where (r,b) = revEqI s1 s2 (x:acc)
revEqI _ _ acc          = (acc,True)
```

## Beispiel Operationen auf binären Bäumen mit Blatteinträgen

```
data Btree a = L a | Btree a :# Btree a
```

$foldRepl(t)(x)$  ersetzt alle Blatteinträge von  $t$  durch  $x$ :

```
foldRepl :: (a -> a -> a) -> Btree a -> a -> (a,Btree a)
foldRepl _ (L x) y      = (x,L y)
```

```
foldRepl f (t1:#t2) x = (f y z,u1:#u2) where (y,u1) = foldRepl f t1 x
                                                (z,u2) = foldRepl f t2 x
```

*tipsReplRest*(*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*:

```
tipsReplRest :: Btree a -> [a] -> ([a],Btree a,[a])
tipsReplRest (L x) (y:s) = ([x],L y,s)
tipsReplRest (t1:#t2) s  = (ls1++ls2,u1:#u2,s2)
                           where (ls1,u1,s1) = tipsReplRest t1 s
                                (ls2,u2,s2) = tipsReplRest t2 s1
```

Die folgende Version vermeidet Listenkonkatenationen:

```
tipsReplRestI :: Btree a -> [a] -> [a] -> ([a],Btree a,[a])
tipsReplRestI (L x) (y:s) acc = (x:acc,L y,s)
tipsReplRestI (t1:#t2) s acc  = (ls2,u1:#u2,s2)
                           where (ls1,u1,s1) = tipsReplRestI t1 s acc
                                (ls2,u2,s2) = tipsReplRestI t2 s1 ls1
```

## Termination und Konfluenz

Um sicherzustellen, dass alle mit den Regeln für  $\Phi$  durchgeführten Reduktionen terminieren, setzen wir eine Termrelation  $\gg$  voraus, für die Folgendes gilt:

- $\gg$  ist **wohlfundiert**, d.h. jede nichtleere Teilmenge von  $T_\Sigma(X)$  hat ein minimales Element.
- Für alle  $1 \leq i \leq k$ ,  $g \in \Phi$ ,  $0 \leq j \leq n_i$  und alle Teilterme  $g(t)$  von  $t_{ij}$  gilt:

$$g \succ^+ f_i \implies p_{i0} \gg t,$$

wobei die Relation  $\succ \subseteq \Phi^2$  wie folgt definiert ist: Sei  $1 \leq i \leq k$  und  $g \in \Phi$ .

$$f_i \succ g \iff_{def} \text{ es gibt } 0 \leq j \leq n_i \text{ derart, dass } g \text{ in } t_{ij} \text{ vorkommt.}$$

Kurz gesagt: Entweder wird  $f$  in der Definition von  $g$  nicht benutzt oder die Argumente der rekursiven Aufrufe von  $g$  in (1) sind kleiner als  $p_0$ .

Sei  $Red$  die Menge der Regeln für  $\Phi$ .

Aus  $\succ$  und  $\gg$  lässt sich eine transitive und wohlfundierte Termrelation  $\rightsquigarrow$  konstruieren, die  $\rightarrow_{Red}$  und die **Teiltermrelation**  $\supset$  enthält:

$$t \supset u \iff_{def} u \text{ ist ein echter Teilterm von } t.$$

Die Termination von Reduktionen, in denen die  $\beta$ -Regel:

$$(\lambda p.t)(p\sigma) \rightarrow t\sigma$$

verwendet wird, folgt aus der impliziten Voraussetzung, dass alle  $\Sigma$ -Terme in ihrem jeweiligen Kontext eindeutig **typisierbar** sind. Damit ist insbesondere die Anwendung einer Funktion auf sich selbst ausgeschlossen, also auch die unendliche Reduktion

$$(\lambda x.(x(x)))(\lambda x.(x(x))) \rightarrow (\lambda x.(x(x)))(\lambda x.(x(x))) \rightarrow \dots$$

Hat umgekehrt der Redex  $(\lambda p.t)(p\sigma)$  der  $\beta$ -Regel einen eindeutigen Typ, dann repräsentiert er eine Funktion  $n$ -ter Ordnung. Da deren Bilder Funktionen  $(n-1)$ -ter Ordnung sind, sind auch alle Funktionen, die durch Teilterme des Reduktes  $t\sigma$  dargestellt werden, von höchstens  $(n-1)$ -ter Ordnung.

Folglich bleiben  $\rightarrow_{Red}$  und  $\rightarrow_{Red}^+$  wohlfundiert, auch wenn man die Regeln für  $\lambda$ -Applikationen zu  $Red$  hinzunimmt.

$\rightarrow_{Red}^+$  ist nicht nur wohlfundiert, sondern auch **konfluent**, d.h. für je zwei Reduktionen  $t \rightarrow_{Red}^* u$  und  $t \rightarrow_{Red}^* u'$  desselben Terms  $t$  gibt es einen Term  $v$  mit  $u \rightarrow_{Red}^* v$  und  $u' \rightarrow_{Red}^* v$ .

Die Konfluenz von  $\rightarrow_{Red}^*$  folgt aus der Konfluenz der induktiv definierten Relation  $\Rightarrow_{Red}$ , die simultan auf einem Term durchgeführte Reduktionsschritte beschreibt und wie folgt definiert ist:

- $Red \subseteq \Rightarrow_{Red}$ .
- Für alle  $t \in T_\Sigma(X, C)$ ,  $t \Rightarrow_{Red} t$ .
- Für alle Applikationen  $t = t_0(t_1, \dots, t_n)$  und  $t' = t'_0(t'_1, \dots, t'_n)$ ,

$$t_0 \Rightarrow_{Red} t'_0 \wedge \dots \wedge t_n \Rightarrow_{Red} t'_n \text{ impliziert } t \Rightarrow_{Red} t'.$$

- Für alle  $\lambda$ -Abstraktionen  $t = \lambda p.u$  und  $t' = \lambda p.u'$ ,  $u \Rightarrow_{Red} u'$  impliziert  $t \Rightarrow_{Red} t'$ .
- Für alle  $\mu$ -Abstraktionen  $t = \mu p.u$  und  $t' = \mu p.u'$ ,  $u \Rightarrow_{Red} u'$  impliziert  $t \Rightarrow_{Red} t'$ .

Aus  $\rightarrow_{Red} \subseteq \Rightarrow_{Red} \subseteq \rightarrow_{Red}^*$  folgt  $\rightarrow_{Red}^* = \Rightarrow_{Red}^*$ .

Sei  $t \Rightarrow_{Red}^* u$  und  $t \Rightarrow_{Red}^* u'$ . Die Existenz eines Terms  $v$  mit  $u \Rightarrow_{Red}^* v$  und  $u' \Rightarrow_{Red}^* v$  erhält durch Induktion über die Anzahl der  $\Rightarrow_{Red}$ -Schritte, aus denen sich die Reduktionen  $t \Rightarrow_{Red}^* u$  und  $t \Rightarrow_{Red}^* u'$  zusammensetzen.

Ein Term  $t \in BT_\Sigma(X, C)$ , auf den keine Regel von  $Red$  anwendbar ist (formal:  $t \rightarrow_{Red}^* t' \Rightarrow t = t'$ ), heißt **Red-Normalform über  $X$  und  $C$** . Die  $S$ -sortige Menge der  $Red$ -Normalformen wird mit  $NF_{Red}(X, C)$  bezeichnet.

Ein Term  $t \in BT_\Sigma(X, C)$ . Eine  $Red$ -Normalform  $t'$  mit  $t \rightarrow_{Red} t'$  heißt **Red-Normalform von  $t$** .

Da  $\rightarrow_{Red}^+$  wohlfundiert ist, hat jeder Term von  $BT_\Sigma(X, C)$  eine *Red*-Normalform.

Da  $\rightarrow_{Red}^*$  konfluent ist, stimmen die Normalformen zweier Terme von  $BT_\Sigma(X, C)$  mit gemeinsamer unterer Schranke bzgl.  $\rightarrow_{Red}^*$  überein.

Zusammengenommen folgt aus der Wohlfundiertheit und Konfluenz von  $\rightarrow_{Red}^+$ , dass jeder Term von genau eine *Red*-Normalform  $nf(t)$  hat, die man auch als **Reduktions-** oder **operationelle Semantik von  $t$**  bezeichnet.

Da  $\rightarrow_{Red}^*$  konfluent ist, sind Term  $t \in BT_\Sigma(X, C)$  genau eine *Red*-Normalform  $nf(t)$ , die man auch als **Reduktions-** oder **operationelle Semantik von  $t$**  bezeichnet.

Mehr noch: Da *Red* keine Regeln enthält, deren linke Seiten  $\Sigma$ -Muster sind, sind alle  $\Sigma$ -Muster von  $t \in BT_\Sigma(X, C)$  *Red*-Normalformen. Umgekehrt ist auf jeden Term von  $BT_\Sigma(X, C)$  eine Regel von *Red* anwendbar. Also gilt:

$$NF_{Red}(X, C) = P(X, C) \cap BT_\Sigma(X, C).$$

Seien  $BS$  und  $BF$  die Mengen der Sorten bzw. Funktionssymbole von  $B\Sigma$  und  $B$  eine  $B\Sigma$ -Algebra. Für alle  $s \in BS$ , sei  $X_s = B_s$ . Dann bildet  $NF_{Red}(X, C)$  die Trägermenge einer  $\Sigma$ -Algebra  $A$ :

- Für alle  $s \in S$ ,  $A_s = NF_{Red}(X, C)_s$ .
- Für alle  $f : s_1 \dots s_n \rightarrow s \in F$  und  $t \in NF_{Red}(X, C)_{s_i}$ ,  $1 \leq i \leq n$ ,

$$f^A(t_1, \dots, t_n) =_{def} nf(f(t_1, \dots, t_n)).$$



Die Auswertung in  $A$  eines Terms von  $BT_\Sigma(X, C)$  liefert seine *Red*-Normalform, d.h. für alle  $t \in BT_\Sigma(X, C)$  gilt:

$$id^*(t) = nf(t).$$

*Beweis durch Induktion über  $size(t)$ .*

*Fall 1:*  $t \in B$ . Dann gilt  $id^*(t) = id(t) = t = nf(t)$ .

*Fall 2:*  $t = f(t_1, \dots, t_n)$  für ein  $f \in F$ . Dann gilt nach Induktionsvoraussetzung:

$$\begin{aligned} id^*(t) &= f^A(id^*(t_1), \dots, id^*(t_n)) = f^A(nf(t_1), \dots, nf(t_n)) = nf(f(nf(t_1), \dots, nf(t_n))) \\ &= nf(f(t_1, \dots, t_n)) = nf(t). \end{aligned} \quad \square$$

## Partiell-rekursive Funktionen

Es fehlen noch Reduktionsregeln für  $\mu$ -Abstraktionen. Diese beschreiben partiell-rekursive Funktionen, das sind partielle Funktionen, die berechenbar sind, obwohl ihr Definitionsbereich möglicherweise nicht entscheidbar ist. Das zeigt sich bei ihrer Auswertung durch Termreduktion darin, dass manche Reduktionen einiger Aufrufe solcher Funktionen nicht terminieren. Schuld daran ist gerade die – unvermeidliche – Regel zur Reduktion von  $\mu$ -Abstraktionen (deren Korrektheit sich direkt aus der Interpretation von  $\mu p.t$  als kleinste Lösung der Gleichung  $p = t$  ergibt):

$$\mu p.t \rightarrow t[(\lambda p.x)(\mu p.t)/x \mid x \in var(p)] \quad \text{Expansionsregel}$$

Man sieht sofort, dass diese Regel unendlich oft hintereinander angewendet werden kann.

Eine **Reduktionsstrategie** legt für jeden Term  $t$  fest, welcher Teilterm von  $t$  durch welche (anwendbare) Regel in einem Reduktionsschritt ersetzt wird. Da Konfluenz eindeutige Normalformen impliziert, unterscheiden sich Reduktionsstrategien bezüglich der jeweils erzielten Ergebnisse nur in der Anzahl der Terme, die sie zu Normalformen reduzieren.

Eine Reduktionsstrategie heißt **vollständig**, wenn sie jeden Term, der eine Normalform hat, dort auch hinführt. Da nur die Anwendung der Expansionsregel unendliche Reduktionen erzeugt, hängt die Vollständigkeit der Strategie i.w. davon ab, wann und wo sie die Expansionsregel anwendet.

Sind alle Reduktionen eines Terms  $f(t)$  unendlich, dann ist  $f$  an der Stelle  $t$  nicht definiert. Andererseits muss  $f$  in einer  $\Sigma$ -Algebra  $A$  als totale Funktion interpretiert werden. Dazu werden die Trägermengen von  $A$  zu CPOs erweitert (siehe Kapitel 6). Einem “undefinierten” Term  $f(t)$  des Typs  $e$  wird das kleinste – durch  $\perp_e$  bezeichnete – Element des CPOs  $A_e$  zugeordnet.

Für *Sorten*  $e \in S$  wird die Existenz von  $\perp_e \in A_e$  vorausgesetzt und  $A_e$  als flacher CPO angenommen, d.h.

$$a \leq b \iff_{def} a = \perp_e \vee a = b$$

für alle  $a, b \in A_e$ .

Die Halbordnungen auf  $A_s$ ,  $s \in S$ , werden wie folgt auf Produkte und Funktionenräume fortgesetzt: Für alle  $e_1, \dots, e_n, e, e' \in \text{types}(S)$ ,  $a_1, b_1 \in A_{s_1}, \dots, a_n, b_n \in A_{s_n}$  und  $f, g : A_e \rightarrow A_{e'}$ ,

$$(a_1, \dots, a_n) \leq (b_1, \dots, b_n) \iff_{\text{def}} \forall 1 \leq i \leq n : a_i \leq b_i,$$

$$f \leq g \iff_{\text{def}} \forall a \in A : f(a) \leq g(a).$$

Sind alle in  $t_1, \dots, t_n$  auftretenden Funktionen zu monotonen Funktionen erweitert worden, dann ist auch

$$\Phi : A_1 \times \dots \times A_n \rightarrow A_1 \times \dots \times A_n$$

$$\Phi(a_1, \dots, a_n) =_{\text{def}} t[a_i/x_i \mid 1 \leq i \leq n]^{A_1 \times \dots \times A_n}$$

stetig und wir können den **Fixpunktsatz von Kleene** anwenden, nach dem

$$\sqcup_{i \in \mathbb{N}} \Phi^i(\perp) \text{ die kleinste Lösung von } (x_1, \dots, x_n) = t$$

in  $A_1 \times \dots \times A_n$  ist. Daraus ergibt sich die Interpretation einer  $\mu$ -Abstraktion:

$$(\mu x_1 \dots x_n. t)^{A_1 \times \dots \times A_n} =_{\text{def}} \sqcup_{i \in \mathbb{N}} \Phi^i(\perp).$$

Betrachten wir nun das Schema der nicht-rekursiven Definition einer Funktion  $f$ , deren lokale Definitionen in einer Gleichung der Form (1) zusammengefasst sind.

Seien  $x_1, \dots, x_m, z_1, \dots, z_n$  paarweise verschiedene Variablen und  $e, t$  beliebige Terme, in denen  $f$  nicht vorkommt und deren freie Variablen zur Menge  $\{x_1, \dots, x_m, z_1, \dots, z_n\}$  gehören.

$$f(x_1, \dots, x_m) = e \quad \text{where } (z_1, \dots, z_n) = t$$

Aus (7) und (8) ergibt sich die folgende Reduktionsregel zur Auswertung von  $f$ :

$$f(x_1, \dots, x_m) \rightarrow e[\pi_i \$ \mu z_1 \dots z_n . t / z_i \mid 1 \leq i \leq n] \quad \delta\text{-Regel}$$

## Beispiele

```

replace :: (a -> a -> a) -> Btree a -> Btree a
replace f t = u where (x,u) = foldRepl 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))

pal, palI :: Eq a => [a] -> Bool
pal s = b where (r,b) = revEq s r

palI s = b where (r,b) = revEqI s r []

```

```

sort, sortI :: Ord a => Btree a -> Btree a
sort t = u where (ls,u,_) = tipsReplRest 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)))

sortI t = u where (ls,u,_) = tipsReplRestI t (sort ls) []

```

## Funktionen mit beliebigen lokalen Definitionen

Bevor wir auf vollständige Strategien eingehen, definieren wir induktiv das allgemeine Schema der rekursiven Definition  $DS(F, globals)$  einer Menge  $F$  funktionaler oder nichtfunktionaler Objekte mit lokalen Definitionen, die selbst diesem Schema genügen.

$globals$  bezeichnet die Menge der globalen (funktionalen oder nichtfunktionalen) Objekte die in  $DS(F, globals)$  vorkommen.

- Sei  $eqs$  eine Definition von  $F$ , die aus Gleichungen der Form

$$f(p) = e$$

mit  $f \in F$  besteht, wobei  $p$  ein Pattern ist und alle in  $e$  verwendeten Funktionen Standardfunktionen sind oder zur Menge  $globals \cup F$  gehören.

Dann gilt  $eqs \in DS(F, globals)$ .

(a)

- Sei  $\delta$  eine Definition von  $F$ , die aus Gleichungen der Form

$$f(p) = e \text{ where } eqs \quad (eq)$$

mit  $f \in F$  besteht, wobei  $p$  ein Pattern ist und es Mengen  $G_{eq}$  und  $globals_{eq}$  von Funktionen gibt mit  $eqs \in DS(G_{eq}, globals_{eq} \cup F)$ .

Dann gilt  $eqs \in DS(F, \cup_{eq \in \delta} globals_{eq})$ . (b)

Sei  $F = \{f_1, \dots, f_k\}$  und  $eqs \in DS(F, \emptyset)$ .

Im Fall (a) kann jedes  $f \in F$  durch eine einzige  $\mu$ -Abstraktion dargestellt werden: Für alle  $1 \leq i \leq k$  seien

$$f_i(p_{i1}) = e_{i1}, \dots, f_i(p_{in_i}) = e_{in_i}$$

die Gleichungen für  $f_i$  innerhalb von  $eqs$ . Mit

$$\begin{aligned} \mu(eqs) =_{def} \mu f_1 \dots f_k. ( & \lambda p_{11}. e_{11} \parallel \dots \parallel p_{1n_1}. e_{1n_1}, \\ & \vdots \\ & \lambda p_{k1}. e_{k1} \parallel \dots \parallel p_{kn_k}. e_{kn_k} ) \end{aligned}$$

liefert die Gleichung  $(f_1, \dots, f_k) = \mu(eqs)$  eine zu  $eqs$  äquivalente Definition von  $F$ .

Im Fall (b) seien für alle  $1 \leq i \leq k$

$$\begin{aligned} f_i(p_{i1}) &= e_{i1} \text{ where } eqs_{i1}, \\ &\vdots \\ f_i(p_{in_i}) &= e_{in_i} \text{ where } eqs_{in_i} \end{aligned}$$

die Gleichungen für  $f_i$  innerhalb von  $eqs$ . Für alle  $1 \leq i \leq k$  und  $1 \leq j \leq n_i$  gibt es Mengen  $G_{ij}$  und  $globals_{ij}$  von Funktionen mit  $eqs_{ij} \in DS(G_{ij}, globals_{ij} \cup F)$ . Die Substitution  $\sigma_{ij}$  ersetze jede Funktion  $g \in G_{ij}$  in  $e_{ij}$  durch ihre äquivalente  $\mu$ -Abstraktion  $\pi_g(\mu(eqs_{ij}))$ . Mit

$$\begin{aligned} \mu(eqs) &=_{def} \mu f_1 \dots f_k. ( \lambda p_{11}. \sigma_{11}(e_{11}) \parallel \dots \parallel p_{1n_1}. \sigma_{1n_1}(e_{1n_1}), \\ &\quad \vdots \\ &\quad \lambda p_{k1}. \sigma_{k1}(e_{k1}) \parallel \dots \parallel p_{kn_k}. \sigma_{kn_k}(e_{kn_k}) ) \end{aligned}$$

liefert die Gleichung  $(f_1, \dots, f_k) = \mu(eqs)$  eine zu  $eqs$  äquivalente Definition von  $F$ . Aus ihr ergibt sich die folgende Reduktionsregel zur Auswertung von  $f_i$ ,  $1 \leq i \leq k$ :

$$f_i \rightarrow \pi_i \$ \mu(eqs) \quad \delta\text{-Regel}$$

## Die lazy-evaluation-Strategie

Nach einem auf getypte  $\lambda$ - und  $\mu$ -Abstraktionen übertragenen Resultat von Jean Vuillemin ist die folgende **parallel-outermost**, **call-by-need** oder **lazy evaluation** (verzögerte Auswertung) genannte Reduktionsstrategie vollständig:

- $\beta$ - und  $\delta$ -Regeln werden stets vor der Expansionsregel angewendet. (A)
- Die Expansionsregel wird immer parallel auf alle bzgl. der Teiltermordnung maximalen  $\mu$ -Abstraktionen angewendet. (B)

Der Beweis basiert auf der Beobachtung, dass die Konstruktion der kleinsten Lösung von  $(x_1, \dots, x_n) = t$  nach dem **Fixpunktsatz von Kleene** selbst eine Reduktionsstrategie widerspiegelt, die **full-substitution** genannt wird. Diese wendet die Expansionsregel im Unterschied zu (B) parallel auf *alle*  $\mu$ -Abstraktionen an. Da schon die parallele Expansion aller maximalen  $\mu$ -Abstraktionen viel Platz verbraucht, wird sie in der Regel nicht durchgeführt. Stattdessen wird nur die erste auf einem **strikten Pfad** gelegene  $\mu$ -Abstraktion expandiert. Enthält dieser eine kommutative Operation, dann gibt es möglicherweise mehrere solche Pfade, so dass die Strategie unvollständig wird.



Ein Pfad (der Baumdarstellung von)  $t$  ist strikt, wenn jeder Pfadknoten die Wurzel eines Teilterms von  $t$  ist, der zur Herleitung einer Normalform von  $t$  reduziert werden muss, m.a.W.: jeder Pfadknoten ist ein striktes Argument der Funktion im jeweiligen Vorgängerknoten (s.o.).

Sei  $RS$  eine Reduktionsstrategie mit (A). Da  $\beta$ - und  $\delta$ -Regeln niemals unendlich oft hintereinander angewendet werden können, lässt sich jede gemäß  $RS$  durchgeführte Termreduktion eindeutig als Folge

$$t_0 \rightarrow_{RS} t_1 \rightarrow_{RS} t_2 \rightarrow_{RS} \dots$$

von Termen repräsentieren derart, dass für alle  $i \in \mathbb{N}$   $t_{i+1}$  durch parallele Anwendungen der Expansionsregel aus  $t_i$  hervorgeht. Wertet man alle Terme in einem CPO aus, der die Funktionssymbole in den Termen durch monotone Funktionen interpretiert, dann wird aus der obigen Termreduktion eine Kette von Werten in  $A$ :

$$t_0^A \leq t_1^A \leq t_2^A \leq \dots$$

Der **von  $RS$  berechnete Wert von  $t_0$  in  $A$**  wird dann definiert durch:

$$t_{0,RS}^A =_{def} \sqcup_{i \in \mathbb{N}} t_i^A.$$

Diese Definition kann auf Funktionen höherer Ordnung erweitert werden: Sei  $A$  ein  $\mathbf{und}$   $t_0$  ein Term eines Typs  $FT = A_1 \setminus \{\perp\} \rightarrow \dots \rightarrow A_k \setminus \{\perp\} \rightarrow A$ . Dann nennen wir die für alle  $1 \leq i \leq k$  und  $a_i \in A_i$  durch

$$t_{RS}^A(a_1) \dots (a_k) =_{def} (t(a_1) \dots (a_k))_{RS}^A$$

definierte Funktion  $t_{RS}^A : FT$  den **von  $RS$  berechneten Wert von  $t$  in  $A$** .

Offenbar stimmt der von der full-substitution-Strategie ( $FS$ ) berechnete Wert von  $x_i$ ,  $1 \leq i \leq n$ , mit der ( $i$ -ten Projektion der) kleinsten Lösung von (1) in  $A$  überein:

$$x_{i,FS}^A = \pi_i(\sqcup_{j \in \mathbb{N}} \Phi^j(\perp)) = \pi_i(\mu x_1 \dots x_n. t)^A.$$

Das impliziert u.a., dass die kleinste Lösung von (1) niemals kleiner als der von  $RS$  berechnete Wert von  $(x_1, \dots, x_n)$  ist:

$$(x_{1,RS}^A, \dots, x_{n,RS}^A) \leq (x_{1,FS}^A, \dots, x_{n,FS}^A) = (\mu x_1 \dots x_n. t)^A.$$

$RS$  ist also genau dann vollständig, wenn der von  $RS$  berechnete Wert von  $(x_1, \dots, x_n)$  mit der kleinsten Lösung von  $(x_1, \dots, x_n) = t$  übereinstimmt.

Aus der o.g. Voraussetzung, dass die Terme einer Reduktion in einem CPO mit flacher Halbordnung interpretiert werden, folgt:

$$\text{Eine Reduktion } t_0 \rightarrow_{RS} t_1 \rightarrow_{RS} t_2 \rightarrow_{RS} \dots \text{ terminiert } \iff t_{0,RS}^A \neq \perp.$$

Zunächst einmal terminiert die Reduktion genau dann, wenn es  $k \in \mathbb{N}$  gibt, so dass  $t_k$  keine der Variablen von  $x_1, \dots, x_n$  enthält. Wie  $t$ , so ist dann auch  $t_k$  bottomfrei. Also gilt  $t_k^A \neq \perp$  und damit

$$\perp \neq t_k^A = t_k^{A(\perp)} \leq \sqcup_{i \in \mathbb{N}} t_i^{A(\perp)} = t_{0,RS}^A.$$

Enthält für alle  $i \in \mathbb{N}$   $t_i$  Variablen von  $\{x_1, \dots, x_n\}$ , dann gilt für alle  $i \in \mathbb{N}$   $t_i^A(\perp) = \perp$  und damit

$$t_{0,RS}^A = \sqcup_{i \in \mathbb{N}} t_i^{A(\perp)} = \perp.$$

Ein  $i \in \mathbb{N}$  mit  $t_i^{A(\perp)} \neq \perp$  würde nämlich zu einem Widerspruch führen: Sei  $j$  das kleinste  $i$  mit  $t_i^{A(\perp)} \neq \perp$ . Es gäbe eine aus Funktionen von  $t_i$  gebildete monotone Funktion  $f$  sowie  $a_1, \dots, a_m \in A$  mit

$$f(a_1, \dots, a_m, \perp, \dots, \perp) = t_j^{A(\perp)} \neq \perp.$$

Aus der Monotonie von  $f$  und der Flachheit der Halbordnung des CPOs, in dem  $t_j$  interpretiert wird, würde folgen, dass es  $k < j$  und  $b_1, \dots, b_r \in A$  gibt mit

$$t_k^{A(\perp)} = f(a_1, \dots, a_m, b_1, \dots, b_r) = f(a_1, \dots, a_m, \perp, \dots, \perp) \neq \perp$$

im Widerspruch dazu, dass  $j$  das kleinste  $i$  mit  $t_i^{A(\perp)} \neq \perp$  ist. (Ein ähnliches Argument wird verwendet, um zu zeigen, dass parallel-outermost vollständig ist; siehe Zohar Manna, *Mathematical Theory of Computation*, Theorem 5-4.)

Eine Reduktionsstrategie bevorzugt Anwendungen von  $\delta$ -Regeln, um danach  $\mu$ -Abstraktionen eliminieren zu können. Dazu müssen vorher Expansionsschritte die Redexe dieser Regeln erzeugen. Tun sie das nicht, dann kann die Reduktion nicht terminieren, da jedes Expansionsredukt einen neuen Expansionsredex enthält. Neben diesem sollte es also auch einen neuen  $\delta$ - (oder  $\beta$ -) Redex enthalten. Diese Bedingung ist z.B. in der obigen Definition von **pal** verletzt, sofern dort die obige Definition von **revEq** verwendet wird:

```
pal :: Eq a => [a] -> Bool
pal s = b where (r,b) = revEq s r

revEq :: Eq a => [a] -> [a] -> ([a],Bool)
revEq (x:s1) (y:s2) = (r++[x],x==y && b) where (r,b) = revEq s1 s2
revEq _ _          = ([],True)
```

Die Definition von **pal** liefert die  $\delta$ -Regel

$$pal(s) \rightarrow \pi_2 \$\mu r b.revEq(s,r). \quad (1)$$

Parallel-outermost-Reduktionen von *pal* terminieren nicht, weil die Expansionsschritte keine Redexe für die obige Definition von **revEq** liefern:

$$\begin{aligned}
pal[1, 2, 1] &\xrightarrow{(1)} \pi_2 \$ \mu \ r \ b.revEq([1, 2, 1], \textcolor{red}{r}) \\
&\xrightarrow{Expansion} \pi_2 \$ revEq([1, 2, 1], \pi_1 \$ \mu \ r \ b.revEq([1, 2, 1], \textcolor{red}{r})) \\
&\xrightarrow{Expansion} \pi_2 \$ revEq([1, 2, 1], \pi_1 \$ revEq([1, 2, 1], \pi_1 \$ \mu \ r \ b.revEq([1, 2, 1], \textcolor{red}{r})))
\end{aligned}$$

## Auswertung durch Graphreduktion

Manche Compiler funktionaler Sprachen implementieren  $\mu$ -Abstraktionen durch Graphen:  $\mu x_1, \dots, x_n. t$  wird zunächst als Baum dargestellt. Dann werden alle identischen Teilbäume von  $t$  zu jeweils einem verschmolzen (**collapsing**). Schließlich wird für alle  $1 \leq i \leq n$  die Markierung  $x_i$  in  $\pi_i$  umbenannt und von dem mit  $\pi_i$  markierten Knoten eine Kante zur Wurzel von  $t$  gezogen.

Expansionsschritte verändern den Graphen nicht, sondern die Position eines Zeigers  $\bullet$  auf die Wurzel des nächsten Redex. Jedes Fortschreiten des Zeigers auf einer Rückwärtskante implementiert einen Expansionsschritt. Die obige Reduktion von *pal*[1, 2, 1] entspricht folgender Graphtransformation:

$$\begin{aligned}
& \bullet pal[1, 2, 1] \xrightarrow{(1)} \pi_2 \$ \bullet \downarrow revEq([1, 2, 1], \pi_1 \uparrow) \bullet \xrightarrow{moves \ down} \pi_2 \$ \downarrow revEq([1, 2, 1], \bullet \pi_1 \uparrow) \\
& \bullet \xrightarrow{moves \ up} \pi_2 \$ \bullet \downarrow revEq([1, 2, 1], \pi_1 \uparrow) \bullet \xrightarrow{moves \ down} \pi_2 \$ \downarrow revEq([1, 2, 1], \bullet \pi_1 \uparrow) \\
& \bullet \xrightarrow{moves \ up} \pi_2 \$ \bullet \downarrow revEq([1, 2, 1], \pi_1 \uparrow) \bullet \xrightarrow{moves \ down} \dots
\end{aligned}$$

Die Pfeile  $\uparrow$  und  $\downarrow$  zeigen auf die Quelle bzw. das Ziel der einen Rückkante in diesem Beispiel.

Wie muss die Definition von **revEq** repariert werden, damit die Auswertung von  $pal[1, 2, 1]$  terminiert? Trifft der Zeiger  $\bullet$  auf den Ausdruck  $revEq([1, 2, 1], \pi_1 \uparrow)$ , dann muss auf diesen wenigstens ein Reduktionsschritt anwendbar sein, damit er modifiziert und damit der Zyklus, den der Zeiger durchläuft, durchbrochen wird. Man erreicht das mit der folgenden Definition von **revEq**, deren Anwendbarkeit im Gegensatz zur obigen Definition kein pattern matching des zweiten Arguments verlangt:

```

revEq :: Eq a => [a] -> [a] -> ([a], Bool)
revEq (x:s1) s = (r++[x], x==y && b) where y:s2 = s
                                           (r,b) = revEq s1 s2

revEq _ _ = ([], True)

```

Ein **lazy pattern** ist ein Muster, dem das Symbol  $\sim$  vorangestellt ist. Eine  $\lambda$ -Applikation mit einem lazy pattern kann auch dann zum Rumpf der angewendeten Abstraktion reduziert werden, wenn deren Argument keine Instanz des Musters ist:

$$\begin{aligned} (\lambda \sim p.t)(u) &\rightarrow t\sigma && \text{falls } p\sigma = u \\ (\lambda \sim p.t)(u) &\rightarrow t[(\lambda p.x)(u)/x \mid x \in \text{var}(p)] && \text{falls } p \not\leq u \end{aligned}$$

Zum Beispiel gilt  $(\lambda(x:s).5)\square \rightarrow \text{fail}$ , aber  $(\lambda \sim (x:s).5)\square \rightarrow 5$ .

Die obige Definition von **revEq** folgt Schema (1), so dass bei ihrer Überführung in Reduktionsregeln die lokalen Definitionen wie folgt entfernt werden können:

$$\begin{aligned} \text{revEq}(x:s_1, s) &\rightarrow \lambda \sim y:s_2.(\lambda \sim (r, b).(r ++ [x], x = y \&\& b)\$ \text{revEq}(s_1, s_2))\$s & (2) \\ \text{revEq}(\square, s) &\rightarrow (\square, \text{True}) & (3) \end{aligned}$$

Hiermit erhalten wir eine terminierende Reduktion von  $\text{pal}[1, 1]$ , die als Graphtransformation so aussieht: Die Pfeile  $\uparrow$ ,  $\downarrow$ ,  $\nwarrow$ ,  $\searrow$ ,  $\nearrow$  und  $\swarrow$  zeigen auf die Quelle bzw. das Ziel von drei verschiedenen Kanten. Redexe sind rot, die zugehörigen Redukte grün gefärbt.

$$\begin{aligned} &\bullet \text{pal}[1, 1] \\ &\xrightarrow{(1)} \pi_2 \$ \bullet \downarrow \text{revEq}([1, 1], \pi_1 \uparrow) \end{aligned}$$

$$\begin{array}{l}
\begin{array}{l}
(2) \\
\beta\text{-Regel} \\
\text{split term}
\end{array}
\rightarrow \\
\pi_2 \$ \bullet \downarrow (\lambda \sim y : s_2. (\lambda \sim (r, b). (r ++ [1], 1 = y \&\& b) \$ revEq([1], s_2)) \$ \pi_1 \uparrow \\
\pi_2 \$ \bullet \downarrow \lambda \sim (r, b). (r ++ [1], 1 = head \$ \pi_1 \uparrow \&\& b) \$ revEq([1], tail \$ \pi_1 \uparrow) \\
\pi_2 \$ \bullet \downarrow \lambda \sim (r, b). (r ++ [1], 1 = head \$ \pi_1 \uparrow \&\& b) \$ \nwarrow \\
\nwarrow revEq([1], tail \$ \pi_1 \uparrow) \\
\beta\text{-Regel} \\
\rightarrow \\
\pi_2 \$ \bullet \downarrow (\pi_1 \nwarrow ++ [1], 1 = head \$ \pi_1 \uparrow \&\& \pi_2 \nwarrow) \\
\nwarrow revEq([1], tail \$ \pi_1 \uparrow) \\
\bullet \xrightarrow{\text{moves down}} \\
\pi_2 \$ \downarrow (\pi_1 \nwarrow ++ [1], 1 = head \$ \pi_1 \uparrow \&\& \pi_2 \nwarrow) \\
\bullet \nwarrow revEq([1], tail \$ \pi_1 \uparrow) \\
(2) \\
\beta\text{-Regel} \\
\rightarrow \\
\pi_2 \$ \downarrow (\pi_1 \nwarrow ++ [1], 1 = head \$ \pi_1 \uparrow \&\& \pi_2 \nwarrow) \\
\bullet \nwarrow (\lambda \sim y : s_2. (\lambda \sim (r, b). (r ++ [1], 1 = y \&\& b) \$ revEq([], s_2)) \$ tail \$ \pi_1 \uparrow) \\
\pi_2 \$ \downarrow (\pi_1 \nwarrow ++ [1], 1 = head \$ \pi_1 \uparrow \&\& \pi_2 \nwarrow) \\
\bullet \nwarrow \lambda \sim (r, b). (r ++ [1], 1 = head \$ tail \$ \pi_1 \uparrow \&\& b) \$ revEq([], tail \$ tail \$ \pi_1 \uparrow) \\
\text{split term} \\
\rightarrow \\
\pi_2 \$ \downarrow (\pi_1 \nwarrow ++ [1], 1 = head \$ \pi_1 \uparrow \&\& \pi_2 \nwarrow) \\
\bullet \nwarrow \lambda \sim (r, b). (r ++ [1], 1 = head \$ tail \$ \pi_1 \uparrow \&\& b) \$ \nearrow \\
\swarrow revEq([], tail \$ tail \$ \pi_1 \uparrow)
\end{array}$$



$\beta\text{-Regel}$   
 $\rightarrow$

$$\begin{aligned} & \pi_2\$ \downarrow (\pi_1 \nwarrow ++ [1], 1 = \text{head}\$ \pi_1 \uparrow \&\& \pi_2 \nwarrow) \\ & \bullet \searrow (\pi_1 \nearrow ++ [1], 1 = \text{head}\$ \text{tail}\$ \pi_1 \uparrow \&\& \pi_2 \nearrow) \\ & \swarrow \text{revEq}([], \text{tail}\$ \text{tail}\$ \pi_1 \uparrow) \end{aligned}$$

$\bullet \text{ moves down}$   
 $\rightarrow$

$$\begin{aligned} & \pi_2\$ \downarrow (\pi_1 \nwarrow ++ [1], 1 = \text{head}\$ \pi_1 \uparrow \&\& \pi_2 \nwarrow) \\ & \searrow (\pi_1 \nearrow ++ [1], 1 = \text{head}\$ \text{tail}\$ \pi_1 \uparrow \&\& \pi_2 \nearrow) \\ & \bullet \swarrow \text{revEq}([], \text{tail}\$ \text{tail}\$ \pi_1 \uparrow) \end{aligned}$$

(3)  
 $\rightarrow$

$$\begin{aligned} & \pi_2\$ \downarrow (\pi_1 \nwarrow ++ [1], 1 = \text{head}\$ \pi_1 \uparrow \&\& \pi_2 \nwarrow) \\ & \searrow (\bullet \pi_1 \nearrow ++ [1], 1 = \text{head}\$ \text{tail}\$ \pi_1 \uparrow \&\& \bullet \pi_2 \nearrow) \\ & \swarrow ([], \text{True}) \end{aligned}$$

$\delta\text{-Regeln}$   
 $\rightarrow$

$$\begin{aligned} & \pi_2\$ \downarrow (\pi_1 \nwarrow ++ [1], 1 = \text{head}\$ \pi_1 \uparrow \&\& \pi_2 \nwarrow) \\ & \searrow (\bullet [] ++ [1], \bullet 1 = \text{head}\$ \text{tail}\$ \pi_1 \uparrow \&\& \text{True}) \end{aligned}$$

$\delta\text{-Regeln}$   
 $\rightarrow$

$$\begin{aligned} & \pi_2\$ \downarrow (\bullet \pi_1 \nwarrow ++ [1], 1 = \text{head}\$ \pi_1 \uparrow \&\& \bullet \pi_2 \nwarrow) \\ & \searrow ([1], 1 = \text{head}\$ \text{tail}\$ \pi_1 \uparrow) \end{aligned}$$

$\delta\text{-Regeln}$   
 $\rightarrow$

$$\pi_2\$ \downarrow (\bullet [1] ++ [1], 1 = \text{head}\$ \bullet \pi_1 \uparrow \&\& 1 = \text{head}\$ \text{tail}\$ \bullet \pi_1 \uparrow)$$

$\delta\text{-Regeln}$   
 $\rightarrow$

$$\pi_2\$ \downarrow ([1, 1], 1 = \text{head}\$ \bullet \pi_1 \uparrow \&\& 1 = \text{head}\$ \text{tail}\$ \bullet \pi_1 \uparrow)$$

$\delta\text{-Regel}$   
 $\rightarrow$

$$\bullet \pi_2\$ ([1, 1], 1 = \text{head}\$ [1, 1] \&\& 1 = \text{head}\$ \text{tail}\$ [1, 1])$$

$\delta\text{-Regel}$   
 $\rightarrow$

$$1 = \bullet \text{head}\$ [1, 1] \&\& 1 = \text{head}\$ \bullet \text{tail}\$ [1, 1]$$

$$\begin{array}{lcl}
\beta\text{-Regel} & \xrightarrow{\quad} & \bullet 1 = 1 \&\& 1 = \text{head}\$[1] \\
\delta\text{-Regel} & \xrightarrow{\quad} & \bullet \text{True} \&\& 1 = \text{head}\$[1] \\
\delta\text{-Regel} & \xrightarrow{\quad} & 1 = \bullet \text{head}\$[1] \\
\beta\text{-Regel} & \xrightarrow{\quad} & \bullet 1 = 1 \\
\delta\text{-Regel} & \xrightarrow{\quad} & \text{True}
\end{array}$$

In **Expander2** sieht die aus den obigen Regeln (1)-(3) bestehende Definition von **pal** und **revEq** folgendermaßen aus:

```
pal2(s) == get1(mu r b.revEq2(s)(r)) &
```

```
revEq2[]      == fun(~[], ([], bool(True))) &
revEq2(x:s1) == fun(~(y:s2), fun((r,b), (r++[x], bool(x=y & Bool(b))))
               (revEq2(s1)(s2))) &
```

Die darauf basierende Reduktion von **pal2[1,1]** enthält zwar z.T. größere Terme als die obige Graphreduktion von **pal[1,1]**. Dafür entfällt aber die dort erforderliche Zeigerverwaltung:

```
pal2[1,1]
```

```
get1(mu r b.(revEq2[1,1](r)))
```

```

get1(mu r b.(fun(~(y:s2),
    fun((r,b),(r++[1],bool(1 = y & Bool(b))))
    (revEq2[1](s2)))
(r)))

```

```

get1(mu r b.(fun((r0,b),(r0++[1],bool(1 = head(r) & Bool(b))))
    (revEq2[1](tail(r)))))

```

```

get1(mu r b.(fun((r0,b),(r0++[1],bool(1 = head(r) & Bool(b))))
    (fun(~(y:s2),
        fun((r,b),(r++[1],bool(1 = y & Bool(b))))
        (revEq2[] (s2)))
    (tail(r)))))

```

```

get1(mu r b.(fun((r0,b),(r0++[1],bool(1 = head(r) & Bool(b))))
    (fun((r0,b),(r0++[1],bool(1 = head(tail(r)) & Bool(b))))
    (revEq2[] (tail(tail(r)))))

```

```

get1(mu r b.(fun((r0,b),(r0++[1],bool(1 = head(r) & Bool(b))))
    (fun((r0,b),(r0++[1],bool(1 = head(tail(r)) & Bool(b))))
    (fun(~[],([],bool(True)))
    (tail(tail(r)))))

```

```

get1(mu r b.(fun((r0,b),(r0++[1],bool(1 = head(r) & Bool(b))))

```

```
(fun((r0,b),(r0++[1],bool(1 = head(tail(r)) & Bool(b))))  
  ([],bool(True))))
```

```
get1(mu r b.(fun((r0,b),(r0++[1],bool(1 = head(r) & Bool(b))))  
  ([]++[1],bool(1 = head(tail(r)) & Bool(bool(True)))))
```

```
get1(mu r b.(fun((r0,b),(r0++[1],bool(1 = head(r) & Bool(b))))  
  ([1],bool(1 = head(tail(r)))))
```

```
get1(mu r b.([1]++[1],bool(1 = head(r) & Bool(bool(1 = head(tail(r)))))
```

```
get1(mu r b.([1,1],bool(1 = head(r) & 1 = head(tail(r)))))
```

```
bool(1 = head(get0(mu r b.([1,1],bool(1 = head(r) & 1 = head(tail(r))))) &  
  1 = head(tail(get0(mu r b.([1,1],bool(1 = head(r) & 1 = head(tail(r)))))
```

```
bool(1 = head[1,1] &  
  1 = head(tail(get0(mu r b.([1,1],bool(1 = head(r) & 1 = head(tail(r)))))
```

```
bool(1 = 1 &  
  1 = head(tail(get0(mu r b.([1,1],bool(1 = head(r) & 1 = head(tail(r)))))
```

```
bool(1 = head(tail(get0(mu r b.([1,1],bool(1 = head(r) & 1 = head(tail(r)))))
```

```
bool(1 = head(tail[1,1]))
```

```
bool(1 = head[1])
```

```
bool(1 = 1)
```

```
bool(True)
```

```
Number of steps: 19
```

## Unendliche Objekte

Auch im Fall, dass einige Datenbereiche aus unendlichen Objekten bestehen (wie im Client/-Server-Beispiel (siehe [Das relationale Berechnungsmodell](#)), können die obigen Ergebnisse verwendet werden. Allerdings macht es i.d.R. keinen Sinn, solche Datenbereiche in der oben beschriebenen Weise zu einem CPO zu vervollständigen. Stattdessen wird z.B. eine unendliche Liste als Supremum ihrer endlichen Präfixe modelliert, die selbst als Ausdrücke der Form  $a_1 : \dots : a_n : \perp$  dargestellt werden. Die zugrundeliegende Halbordnung ist nicht flach, sondern wird von der Ungleichung  $\perp \leq s$  erzeugt (siehe z.B. [\[Bird1\]](#), Kapitel 9; [\[Pad1\]](#), Kapitel 17; [\[Pad2\]](#), Kapitel 20).

Wie rekursive Funktionen, so lassen sich auch unendliche Objekte als Lösungen iterativer Gleichungen beschreiben. So repräsentiert z.B. die Gleichung `ones = 1:ones` die unendliche Liste von Einsen.

Bevor wir eine allgemeine Struktur zur Modellierung von Mengen unendlicher Objekte behandeln, verweisen wir auf die [Expander2](#)-Version des Client/Server-Beispiels (siehe [Das relationale Berechnungsmodell](#)), mit deren Hilfe die oben angekündigte Termreduktion

`take 3 requests --> ... --> [0,2,6]`

durchgeführt werden kann:

```
CSR =  $\mu$  client server requests. ( $\lambda a. \lambda \sim(b:s). (a:client(mkRequest \$ b)(s)),$   

 $\lambda \sim(a:s). (mkResponse(a):server(s)),$   

  client(0) $ server $ requests) &
```

```
mkRequest = (*2) & mkResponse = (+1)
```

CSR fasst die Definitionen von `client`, `server` und `requests` zu einer  $\mu$ -Abstraktion zusammen. Der Term `take(3) requests = take(3) $ get2 CSR` wird in 66 Reduktionsschritten zu `[0,2,6]` reduziert (siehe [From Modal Logic to \(Co\)Algebraic Reasoning](#), §24).

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 [\[Pad1\]](#), Kapitel 2, oder [\[Pad2\]](#), Kapitel 11).

Die  $(BS \cup S)$ -sortige Menge  $CT_\Sigma$  der  $\Sigma$ -**Bäume** besteht aus allen partiellen Funktionen

$$t : \mathbb{N}^* \rightarrow F \cup (\cup BS)$$

derart, dass gilt:

- für alle  $B \in BS$ ,  $CT_{\Sigma,B} = B$ ,
- für alle  $s \in S$ ,  $t \in CT_{\Sigma,s}$  gdw  $\text{ran}(t(\epsilon)) = s$  und für alle  $w \in \mathbb{N}^*$ ,

$$\text{dom}(t(w)) = e_1 \times \cdots \times e_n \rightarrow s \Rightarrow \forall 0 \leq i < n : (t(wi) \in e_{i+1} \vee \text{ran}(t(wi)) = e_{i+1}).$$

Wir setzen voraus, dass es für alle  $s \in S$  eine Konstante  $\perp_s : \epsilon \rightarrow s$  in  $F$  gibt und definieren damit eine  $S$ -sortige Halbordnung auf  $CT_\Sigma$ : Für alle  $s \in S$  und  $t, u \in CT_{\Sigma,s}$ ,

$$t \leq u \iff_{\text{def}} \forall w \in \mathbb{N}^* : t(w) \neq \perp \Rightarrow t(w) = u(w).$$

Bezüglich dieser Halbordnung ist der  $\Sigma$ -Baum  $\Omega_s$  mit

$$\Omega_s(w) =_{\text{def}} \begin{cases} \perp_s & \text{falls } w = \epsilon \\ \text{undefiniert} & \text{sonst} \end{cases}$$

das kleinste Element von  $CT_{\Sigma,s}$ . Außerdem hat jede Kette  $t_1 \leq t_2 \leq t_3 \leq \dots$  von  $\Sigma$ -Bäumen ein Supremum: Für alle  $w \in \mathbb{N}^*$ ,

$$(\sqcup_{i \in \mathbb{N}} t_i)(w) =_{\text{def}} \begin{cases} t_i(w) & \text{falls } t_i(w) \neq \perp \text{ für ein } i \in \mathbb{N}, \\ \perp & \text{sonst.} \end{cases}$$

Zum Spezialfall unendlicher *Listen*, siehe Bird, Introduction to Functional Programming using Haskell, Kapitel 9.

## Verifikation

Die folgenden drei Methoden dienen dem Beweis von Eigenschaften der kleinsten bzw. größten Lösung einer Gleichung der Form

$$(x_1, \dots, x_n) = t. \tag{1}$$

## Fixpunktinduktion

ist anwendbar, wenn es einen CPO gibt, in dem sich (1) interpretieren lässt und die Funktionen von  $t$  monoton bzw.  $\omega$ -stetig sind. Die Korrektheit der Fixpunktinduktion folgt im ersten Fall aus dem **Fixpunktsatz von Knaster und Tarski** ([Pad2], Kapitel 3), im zweiten aus dem **Fixpunktsatz von Kleene**.

Fixpunktinduktion ist durch folgende Beweisregel gegeben:

$$\frac{\mu x_1 \dots x_n. t \leq u}{t[\pi_i(u)/x_i \mid 1 \leq i \leq n] \leq u} \Uparrow \tag{2}$$

Der Pfeil deutet die Schlußrichtung in einem Beweis an, in dem die Regel angewendet wird. Hier impliziert demnach als der *Sukzedent* der Regel ihren *Antezedenten*.



Der Fixpunktsatz von Knaster und Tarski besagt, dass die kleinste Lösung von (1) dem kleinsten  $t$ -abgeschlossenen Objekt entspricht. Ein Objekt heißt  **$t$ -abgeschlossen**, wenn es die Konklusion von (2) erfüllt.

Zur Anwendung der Fixpunktinduktion muss das Beweisziel die Form der Prämisse von (2) haben.

## Berechnungsinduktion

ist anwendbar, wenn es einen CPO gibt, in dem sich (1) interpretieren lässt und die Funktionen von  $t$   $\omega$ -stetig sind. Die Korrektheit der Berechnungsinduktion folgt aus dem **Fixpunktsatz von Kleene** und erfordert die **Zulässigkeit** des Beweisziels  $\varphi$ , d.h. für alle aufsteigenden Ketten  $a_0 \leq a_1 \leq a_2 \leq \dots$  muss aus der Gültigkeit von  $\varphi(a_i)$  für alle  $i \in \mathbb{N}$  die Gültigkeit von  $\varphi(\sqcup_{i \in \mathbb{N}} a_i)$  folgen. Beispielsweise sind Konjunktionen von Gleichungen oder Ungleichungen zulässig.

Berechnungsinduktion ist durch folgende Beweisregel gegeben:

$$\frac{\varphi(\mu x_1 \dots x_n. t)}{\varphi(\perp) \wedge \forall x_1, \dots, x_n : (\varphi(x_1, \dots, x_n) \Rightarrow \varphi(t))} \Uparrow \quad (3)$$

## Coinduktion

ist anwendbar, wenn sich Gleichung (1) in einer **finalen Coalgebra** lösen lässt (siehe [Pad1], Kapitel 2, oder [Pad2], Kapitel 12). Die Trägermengen dieser Coalgebra stimmen mit denen von  $CT_\Sigma$  überein (siehe **Unendliche Objekte**). Ihre Destruktoren sind

- für alle  $s \in RS$  eine Funktion

$$d_s : s \rightarrow \coprod_{c: s_1 \times \dots \times s_n \rightarrow s \in C} s_1 \times \dots \times s_n,$$

deren Interpretation in  $CT_\Sigma$  einen  $\Sigma$ -Baum  $t$  in seine Wurzel und seine Unterbäume zerlegt:

$$d_s^{CT_\Sigma}(t) =_{\text{def}} (t(\epsilon) : s_1 \times \dots \times s_n \rightarrow s, (\lambda w. t(0w), \dots, \lambda w. t((n-1)w))),$$

- für alle  $n > 1$ ,  $s_1, \dots, s_n \in S$  und  $1 \leq i \leq n$ , eine Funktion  $\pi_i : s_1 \times \dots \times s_n \rightarrow s_i$ , deren Interpretation in  $CT_\Sigma$  ein Baumpel auf seine  $i$ -te Komponente projiziert:

$$\pi_i^{CT_\Sigma}(t_1, \dots, t_n) = t_i.$$

Z.B. ist  $CT_\Sigma$  im Fall der Listensignatur  $\Sigma = (\{entry\}, \{list\}, E \cup \{\[], (:)\})$  isomorph zur Menge der endlichen und unendlichen Wörter über  $E$ .

Aus der Finalität von  $CT_\Sigma$  folgt u.a., dass für alle  $s \in S$  zwei  $\Sigma$ -Bäume  $t$  und  $u$  der Sorte  $s$  genau dann gleich sind, wenn sie bzgl. der oben definierten Destruktoren **verhaltensäquivalent** sind.

D.h.  $(t, u)$  liegt in der größten binären Relation  $\sim$  von  $CT_\Sigma$ , welche die Implikation

$$x \sim y \Rightarrow d_s(x) \sim d_s(y) \quad (4)$$

erfüllt.

Ein **coinduktiver Beweis** von  $t \sim u$  besteht darin, eine binäre Relation  $\approx$  zu finden, die das Paar  $(t, u)$  enthält und (4) erfüllt. Man geht aus von  $\approx = \{(t, u)\}$ , wendet (4) von links nach rechts auf die Paare von  $\approx$  an und erhält damit Instanzen der rechten Seite von (4), die zu  $\approx$  hinzugenommen werden. Auf die neuen Paare von  $\approx$  wird wieder (4) angewendet, usw. Das Verfahren terminiert, sobald alle durch Anwendungen von (4) auf  $\approx$  erzeugten Paare bereits im Äquivalenzabschluss von  $\approx$  liegen. Dann gilt (4) für  $\approx$  und wir schließen  $t \sim u$  daraus, dass  $\sim$  die größte Relation ist, die (4) erfüllt.

Dieses Verfahren basiert auf der zur Fixpunktinduktion dualen Regel:

$$\frac{u \leq \nu x_1 \dots x_n. t}{u \leq t[\pi_i(u)/x_i \mid 1 \leq i \leq n]} \Uparrow \quad (5)$$

(5) ist anwendbar, wenn es einen  $\omega$ -**covollständigen poset**, kurz: **coCPO**, gibt, in dem sich (1) interpretieren lässt und die Funktionen von  $t$  monoton bzw.  $\omega$ -**costetig** sind. Die Korrektheit der Coinduktion folgt im ersten Fall aus dem **Fixpunktsatz von Knaster und Tarski**, im zweiten aus dem **Fixpunktsatz von Kleene** für coCPOs.

Die im oben skizzierten coinduktiven Beweis verwendete Variante von (5) basiert auf dem **Potenzmengenverband** der durch prädikatenlogische Formeln gegebenen Relationen auf einer – ggf. mehrsortigen – Menge  $A$ . Die Halbordnung  $\leq$  entspricht dort der Mengeninklusion, das kleinste Element ist die leere Menge, das größte die Menge  $A$ . Damit wird (5) zur Beweisregel für Implikationen:

## Relationale Coinduktion

$$\frac{\psi \Rightarrow (\nu x_1 \dots x_n. \varphi)(\vec{x})}{\forall \vec{x} (\psi \Rightarrow \varphi[\pi_i(\lambda \vec{x}. \psi)/x_i \mid 1 \leq i \leq n](\vec{x}))} \Uparrow \quad (6)$$

$\varphi$  und  $\psi$  sind hier  $n$ -Tupel prädikatenlogischer Formeln,  $x_1, \dots, x_n$  Prädikatvariablen und  $\vec{x}$  ein Tupel von Individuenvariablen.  $\nu x_1 \dots x_n. \varphi$  wird interpretiert als das  $n$ -Tupel der größten Relationen, das die logische Äquivalenz

$$\langle x_1, \dots, x_n \rangle(\vec{x}) \iff \varphi(\vec{x}) \quad (7)$$

erfüllt, die der Gleichung (1) entspricht.

Substitution, Implikation und andere aussagenlogische Operatoren werden komponentenweise auf Formeltupel fortgesetzt:

$$\begin{aligned}
\langle \varphi_1, \dots, \varphi_n \rangle(\vec{x}) &=_{def} (\varphi_1(\vec{x}), \dots, \varphi_n(\vec{x})), \\
(\varphi_1, \dots, \varphi_n) \Rightarrow (\psi_1, \dots, \psi_n) &=_{def} (\varphi_1 \Rightarrow \psi_1) \wedge \dots \wedge (\varphi_n \Rightarrow \psi_n) \\
\dots
\end{aligned}$$

Die oben definierte  $s$ -Verhaltensäquivalenz  $\sim_s$  auf  $CT_{\Sigma,s}$  ist durch die Formel

$$\nu \approx_s .\lambda(x, y).d_s(x) \approx_{ran(d_s)} d_s(y)$$

als größte Lösung der Instanz

$$x \approx_s y \iff d_s(x) \approx_{ran(d_s)} d_s(y) \quad (8)$$

von (7) definiert. Die entsprechende Instanz der Coinduktionsregel (6) lautet demnach wie folgt:

$$\frac{x \approx_s y \Rightarrow x \sim_s y}{\forall x, y : (x \approx_s y \Rightarrow d_s(x) \approx_{ran(d_s)} d_s(y))} \Uparrow \quad (9)$$

M.a.W.: Alle Paare von  $\approx_s$  sind  $s$ -äquivalent, wenn  $\approx_s$  den Sukzedenten von (9) erfüllt, welcher der Bedingung entspricht.

Da die größte Lösung von (8) eine Äquivalenzrelation ist, also mit ihrem Äquivalenzabschluss übereinstimmt, bleibt Regel (9) korrekt, wenn ihr Sukzedent zu

$$\forall(x, y) (x \approx_s y \Rightarrow d_s(x) \approx_{ran(d_s)}^{eq} d_s(y)) \quad (10)$$

abgeschwächt wird. Deshalb können wir die oben beschriebene schrittweise Konstruktion von  $\approx_s$  bereits dann beenden, wenn sich der *Äquivalenzabschluss* von  $\approx_s$  nicht mehr verändert.

Alle wichtigen Induktions- und Coinduktionsregeln sowie zahlreiche Beispiele ihrer Anwendung finden sich in [From Modal Logic to \(Co\)Algebraic Reasoning](#) sowie [Expander2: Program Verification between Interaction and Automation](#).

Zum Schluss noch die beiden zur relationalen Coinduktion bzw. Berechnungsinduktion dualen Regeln:

## Relationale Fixpunktinduktion

$$\frac{(\mu x_1 \dots x_n. \varphi)(\vec{x}) \Rightarrow \psi}{\forall \vec{x} (\varphi[\pi_i(\lambda \vec{x}. \psi)/x_i \mid 1 \leq i \leq n](\vec{x}) \Rightarrow \psi)} \Uparrow \quad (11)$$

Mit dieser Regel beweist man u.a. Eigenschaften einer Funktion  $f$ , die durch ein rekursives, ggf. bedingtes, Gleichungssystem, also z.B. ein Haskell-Programm, definiert ist.  $\varphi$  bezeichnet dann die Ein/Ausgabe-Relation von  $f$ , hat also die Form  $f(x) = y$ , während  $\psi$  den erwarteten – nicht notwendig funktionalen – Zusammenhang zwischen den Argumenten und Werten von  $f$  beschreibt.

## Berechnungscoiduktion

ist anwendbar, wenn es einen coCPO gibt, in dem sich (1) interpretieren lässt und die Funktionen von  $t$  costetig sind. Die Korrektheit der Berechnungscoiduktion folgt aus dem [Fixpunktsatz von Kleene](#) und erfordert die **Zulässigkeit** des Beweisziels  $\varphi$ , d.h. für alle absteigenden Ketten  $a_0 \geq a_1 \geq a_2 \geq \dots$  muss aus der Gültigkeit von  $\varphi(a_i)$  für alle  $i \in \mathbb{N}$  die Gültigkeit von  $\varphi(\sqcap_{i \in \mathbb{N}} a_i)$  folgen. Beispielsweise sind Konjunktionen von Gleichungen oder Ungleichungen zulässig.

Berechnungscoiduktion ist durch folgende Beweisregel gegeben:

$$\frac{\varphi(\nu x_1 \dots x_n. t)}{\varphi(\top) \wedge \forall x_1, \dots, x_n : (\varphi(x_1, \dots, x_n) \Rightarrow \varphi(t))} \Uparrow \quad (12)$$

Anwendungen dieser Regel sind mir nicht bekannt.

## Bücher und Skripte

[Bird1] Richard Bird, [Introduction to Functional Programming using Haskell](#), Prentice Hall 1998 (in der Lehrbuchsammlung unter L Sr 449/2)

Richard Bird, [Pearls of Functional Algorithm Design](#), Cambridge University Press 2010

Richard Bird, [Thinking Functionally with Haskell](#), Cambridge University Press 2014

Marco Block, Adrian Neumann, Haskell-Intensivkurs, Springer 2011

Manuel M. T. Chakravarty, Gabriele C. Keller, Einführung in die Programmierung mit Haskell, Pearson Studium 2004

Ernst-Erich Doberkat, Haskell: Eine Einführung für Objektorientierte, Oldenbourg 2012

Kees Doets, Jan van Eijck, The Haskell Road to Logic, Maths and Programming, Texts in Computing Vol. 4, King's College 2004

Paul Hudak, The Haskell School of Expression: Learning Functional Programming through Multimedia, Cambridge University Press 2000

Paul Hudak, John Peterson, Joseph Fasel, [A Gentle Introduction to Haskell](#), Yale and Los Alamos 2000



Graham Hutton, Programming in Haskell, Cambridge University Press 2007

[Pad1] P. Padawitz Übersetzerbau, TU Dortmund 2017[Pad1]

[Pad2] P. Padawitz, Fixpoints, Categories, and (Co)Algebraic Modeling, TU Dortmund 2017

Peter Pepper, Petra Hofstedt, Funktionale Programmierung: Sprachdesign und Programmiertechnik, Springer 2006

Fethi Rabhi, Guy Lapalme, Algorithms: A Functional Programming Approach, Addison-Wesley 1999

Simon Thompson, Haskell: The Craft of Functional Programming, 3. Auflage, Addison-Wesley 2011

Raymond Turner, Constructive Foundations for Functional Languages, McGraw-Hill 1991

## Index

*Red*-Normalform, 303

*S*-sortige Funktion, 290

*S*-sortige Menge, 288

$BT_{\Sigma}(X)$ , 290

$T_{\Sigma}(X, C)$ , 289

$NF_{Red}(X, C)$ , 303

$P(X, C)$ , 289

$\Sigma$ -Baum, 327

$\Sigma$ -Term, 129, 289

$\Sigma$ -primitiv, 290

$\alpha$ -Konversion, 291

$\beta$ -Regel, 293

$\delta$ -Regel, 292

$\eta$ -Regel, 293

$free(t)$ , 289

$\lambda$ -Abstraktion, 17

$\lambda$ -Applikation, 18

$\mu$ -Abstraktion, 287

$nf(t)$ , 304

$\omega$ -covollständig, 331

$\sigma^*$ , 290

$\rightarrow_{Red}$ , 292

$n$ -Tupel, 12

$var(t)$ , 289

$(++)$ , 42

$(//)$ , 227

$(ij=)$ , 269

Abschlussoperator, 149

Algebra, 129

all, 66

any, 66

Applikationsoperator, 32

Array, 226

array, 226

Attribut, 83

Auswertungsfunktion, 294

bind, 177  
Bintree, 107  
BintreeL, 109  
  
co-CPO, 142  
co-Kette, 142  
co-stetig, 144  
co-vollständig, 142  
cobind, 269  
coCPO, 331  
Comonad, 269  
Compiler, 259  
const, 33  
costetig, 331  
Cotree, 127  
CPO, 142  
creturn, 181  
curry, 36  
  
Datentyp, 13  
denotationelle Semantik, 281  
Destruktor, 82  
  
disjunkte Vereinigung, 15  
do-Notation, 178  
drop, 43  
duplicate, 270  
dynamisches Objekt, 219  
  
elem, 66  
Eq, 102  
Exp, 90, 191  
exp2code, 132  
extract, 269  
  
fail, 177, 293  
field label, 82  
filter, 66  
Fixpunkt, 144  
Fixpunktsatz von Kleene, 144  
flacher CPO, 145  
flip, 35  
fold2, 59  
foldl, 53  
foldr, 60

freie Variable, 289  
Functor, 173  
Funktion höherer Ordnung, 24  
funktionale Abhängigkeit, 172  
Funktionsapplikation, 18  
Funktionsiteration, 39, 78  
gebundene Variable, 289  
guard, 180  
Halbordnung, 142  
head, 42  
Hue-Farbe, 25  
id, 33  
indices, 42  
Individuenvariable, 17  
init, 43  
Instanz, 18, 291  
Instanz eines Terms, 209  
Instanz eines Typs, 11  
iterate, 77  
Ix, 226  
join, 182  
Kellermaschine, 132  
Kette, 142  
Kind, 171  
Kleisli-Komposition, 181  
Komponente eines Tupels, 12  
Kompositionsoperator, 32  
konfluent, 302  
Konstruktor, 13  
Kopf einer Abstraktion, 289  
last, 43  
lazy pattern, 319  
lazy-Strategie, 19, 35  
leftmost-outermost-Strategie, 19, 35  
liftM, 185  
liftM2, 185  
lines, 49  
Liste, 40  
Listenkomprehension, 67  
logische Programmierung, 281

logische Reduktion, [282](#)  
lokale Definition, [27](#)  
lookup, [52](#)  
lookupM, [185](#)  
  
many, [182](#)  
map, [48](#)  
mapM, [184](#)  
Matching, [18](#)  
mehrwertige Funktion, [188](#)  
Methode, [83](#)  
mkArray, [226](#)  
Monad, [177](#)  
MonadPlus, [177](#)  
monomorpher Typ, [11](#)  
monoton, [144](#)  
mplus, [177](#)  
msum, [183](#)  
mzero, [177](#)  
  
nichtdeterministische Funktion, [188](#)  
notElem, [66](#)  
  
null, [42](#)  
Objektklasse, [83](#)  
operationelle Semantik, [281](#)  
  
partielle Funktion, [187](#)  
polymorpher Typ, [11](#)  
Poset, [142](#)  
Produktbildung, [12](#)  
  
range, [226](#)  
Record, [83](#)  
Redex, [19](#), [292](#)  
reduceE, [137](#)  
Redukt, [19](#), [292](#)  
Reduktionsregel, [292](#)  
Reduktionsrelation, [292](#)  
Reduktionsstrategie, [306](#)  
rekursiv definierte Funktion, [37](#)  
rekursiver Datentyp, [16](#)  
relationale Programmierung, [281](#)  
repeat, [77](#)  
replicate, [77](#)

return, 177  
reverse, 44  
Ring, 148  
root, 122  
Rumpf einer Abstraktion, 289  
Schrittfunktion, 145  
Seiteneffektmonade, 211  
Sektion, 31  
Selektor, 83  
Semiring, 148  
sequence, 183  
Show, 114  
Signatur, 129  
some, 182  
splitAt, 45  
StackCom(x), 132  
statisches Objekt, 218  
stetig, 143  
Substitution, 18  
Subsumptionsordnung, 291

subtrees, 123  
Summenbildung, 12  
Summentyp, 15  
Syntaxbäume, 263  
tail, 42  
take, 43  
Teiltermrelation, 301  
Termreduktion, 287  
transitiver Abschluss, 154  
Tree, 122  
Typ über  $S$ , 288  
Typfamilie, 171  
Typinferenzregeln, 20  
typisierbar, 302  
Typkonstruktor, 11  
Typvariable, 11  
uncurry, 36  
Unifikator, 209  
unifizierbar, 209  
unit-Typ, 11

unlines, [49](#)

unwords, [49](#)

update, [34](#)

updList, [44](#)

Variablenbelegung, [294](#)

Variablenumbenennung, [291](#)

vollständig, [142](#)

vollständige Reduktionsstrategie, [306](#)

vollständiger Semiring, [148](#)

vollständiger Verband, [142](#)

when, [181](#)

Wildcard, [34](#)

wohlfundiert, [301](#)

words, [49](#)

Wort, [40](#)

zip, [48](#)

zipWith, [48](#)

zipWithM, [184](#)

Zustandsäquivalenz, [164](#)