

Kapitel 2

Einführung in Haskell

Dieses Kapitel enthält eine Einführung in Haskell, eine nicht-strikte funktionale Programmiersprache mit polymorphem Typsystem.

Ziel in diesem Kapitel ist:

- Darstellung der syntaktischen Grundlagen,
- Unterscheidung zwischen Kernbestandteilen der Programmiersprache und zusätzlichen Features,
- Programmierverständnis in Haskell,
- operationale Semantik

In einem späteren Kapitel wird die operationale Semantik seziert und analysiert. Die Ergebnisse werden für weitreichende Aussagen über die Kernsprache verwendet. Auch dies soll hier mit vorbereitet werden.

2.1 Die Kernsprache KFPT

Zunächst werden wir eine schwach getypte Kernsprache KFPT einführen, die später auf einer Kernsprache KFP aufgesetzt wird.

Die Kernsprache folgt der Lambda-Schreibweise im Lambda-Kalkül, der von Alonzo Church entwickelt wurde. Der Lambda-Kalkül ist Teil der betrachteten Kernsprache und auch von Haskell.

Syntax

Es gibt *Konstantensymbole*, die jeweils eine feste Stelligkeit haben. Diese nennen wir auch *Konstruktoren*.

Wir nehmen an, dass es eine Möglichkeit gibt, alle Konstruktoren mit Stelligkeit anzugeben, ohne dass wir diese Methode näher spezifizieren. Zum Beispiel als eine Auflistung aller Konstruktoren mit Stelligkeit.

Z.B.

```

True    0
False   0
Nil     0
Cons    2

```

.....

Es gibt eine Menge von Typen, syntaktisch sind das nur Typnamen. Jeder Konstruktor gehört zu genau einem Typ. D.h. wir nehmen eine disjunkte Zerlegung der Constructoren an: z.B. seien die Typen `Bool` und `List`; zu `Bool` gehören `True` und `False`, zu `List` gehören `Cons` und `Nil`.

Es gibt pro Typ ein `case`-Konstrukt: `caseTypname`.

Jetzt kann man syntaktisch die Ausdrücke definieren.

Definition 2.1.1 Eine einfache kontextfreie Grammatik für KFPT-Ausdrücke (Terme, Expressions *Exp*) ist:

$$\begin{aligned}
 \text{Exp} ::= & V && \text{Hierbei steht } V \text{ für Variablen} \\
 & | (\backslash V \rightarrow \text{Exp}) && \text{wobei } V \text{ eine Variable ist.} \\
 & | (\text{Exp}_1 \text{ Exp}_2) \\
 & | (c \text{ Exp}_1 \dots \text{Exp}_n) && \text{wobei } n = ar(c) \\
 & | (\text{case}_{\text{Typname}} \text{Exp of } \{Pat_1 \rightarrow \text{Exp}_1; \dots; Pat_n \rightarrow \text{Exp}_n\}) \\
 & && \text{Hierbei ist } Pat_i \text{ Pattern zum Konstruktor } i, \\
 & && \text{Es kommen genau die Constructoren zum Typname vor} \\
 & && Pat_i \rightarrow \text{Exp}_i \text{ heißt auch } \text{case-Alternative.}
 \end{aligned}$$

$$\begin{aligned}
 Pat ::= & (c V_1 \dots V_{ar(c)}) \\
 & \text{Die Variablen } V_i \text{ müssen alle verschieden sein.}
 \end{aligned}$$

Das Konstrukt $(c V_1 \dots V_{ar(c)})$ nennt man *Pattern* bzw. *Muster*.

Die Notation der Lambda-Ausdrücke im Lambda-Kalkül ist $\lambda x.s$, was $\backslash x \rightarrow s$ in KFPT entspricht. Die Syntax in Haskell ist genauso.

Beispiel 2.1.2 Die Lambda-Notation ist notwendig und sinnvoll, wenn man die Konstante 1 von der Funktion unterscheiden will, die ein Argument hat, und konstant 1 liefert. Letzere stellt man als $\lambda x.1$ dar.

Beispiel 2.1.3 Die Funktion, die erkennt, ob eine Liste, die mit den Constructoren `Nil`, `Cons` aufgebaut wurde, leer ist, könnte man schreiben als:

```
\xs -> (case_List xs of {Nil -> True; Cons y ys -> False})
```

Die konkrete Syntax der `case`-Ausdrücke in Haskell ist nur insofern anders als in KFPT, als man den Index `List` weglassen muss.

Ausdrücke werden folgendermaßen unterschieden:

- $\backslash x \rightarrow s$ sind *Abstraktionen* oder Funktionsausdrücke.

- $(s\ t)$ sind *Anwendungen (Applikationen)*.
- $(c\ t_1 \dots t_{ar(c)})$ sind *Konstruktor-Anwendungen*.
- $(\text{case}_T\ e\ \text{of}\ \text{Alts})$ sind *case-Ausdrücke (Fallunterscheidungen)*.

Der Umgang mit Klammern ist folgendermaßen geregelt: Bei Anwendungen gilt: $((x\ y)\ z)$ ist dasselbe wie $(x\ y\ z)$. D.h. man kann statt $((f\ x)\ y)$ auch $(f\ x\ y)$ schreiben. Der erste Ausdruck entspricht der Syntax. Beachte aber, dass $(x\ (y\ z))$ **nicht** dasselbe wie $(x\ y\ z)$ ist.

Allerdings sind bei Konstruktoranwendungen und Pattern feste Stelligkeitsregeln vorgesehen, da es sich um syntaktische Strukturen handelt.

Die Schreibweise mit λ zur abstrakten Notation einer Funktion wurde von Alonso Church entwickelt, zusammen mit einer operationalen Methode, den Wert einer Funktionsanwendung zu berechnen (Lambda-Kalkül).

Beispiel 2.1.4 *Der Ausdruck $\backslash x \rightarrow x$ (bzw. $\lambda x.x$) ist die Identitätsfunktion. Den Ausdruck*

$$\text{if } A \text{ then } B \text{ else } C$$

kann man darstellen durch den Ausdruck

$$(\text{case}_{\text{Bool}}\ A\ \text{of}\ \{\text{True} \rightarrow B; \text{False} \rightarrow C\})$$

Wenn man einen zweistelligen Konstruktor Paar hat, kann man ein Paar von zwei Objekten True, Nil darstellen als (Paar True Nil). Die Destruktoren bzw. Selektoren kann man schreiben als

$$\begin{aligned} \backslash x \rightarrow & (\text{case_Paar } x \text{ of } (\text{Paar } y1\ y2) \rightarrow y1) \\ \backslash x \rightarrow & (\text{case_Paar } x \text{ of } (\text{Paar } y1\ y2) \rightarrow y2) \end{aligned}$$

In Haskell sind Tupel, insbesondere Paare, bereits vorhanden (s.u.) Schreibweise ist (x, y) für ein Paar, z.B. $(1, 2, \text{Nil})$ für ein Tripel usw.

Ein Liste mit den beiden Elementen True, False ist darstellbar als

$$\text{Cons True (Cons False Nil)}$$

in üblicher Notation auch als $[\text{True}, \text{False}]$ geschrieben. Die Konvention für Listen ist, dass die Listenelemente im ersten Argument stehen, die Restliste im zweiten und die Liste mit Nil abgeschlossen wird.

In Haskell sind Listen bereits vorhanden (s.u.); Schreibweise ist $[x, y]$.

Beispiel 2.1.5 *Einige Standardabkürzungen sind: Id für die identische Abbildung und K für den konstanten Kombinator:*

$$\begin{aligned} \text{Id} & ::= \backslash x \rightarrow x && (\text{Lambda-Notation: } \lambda x.x) \\ \text{K } x\ y & ::= \backslash x \rightarrow (\backslash y \rightarrow x) && (\text{Lambda-Notation: } \lambda x.(\lambda y.x)) \end{aligned}$$

Beispiel 2.1.6 *Die Abfrage, ob ein Paar als erstes Element ein False hat, kann man schreiben als:*

```
(case_Paar p of {Paar x y ->
                 (case_Bool x of {True -> False; False -> True})})
```

Beispiel 2.1.7

$$bot := (\lambda x.(x x)) (\lambda y.(y y))$$

ist ein Ausdruck, für den wir zeigen werden, dass dessen Auswertung nicht terminiert.

2.1.1 Freie und gebundene Variablen

Die Bindungsregeln in einer Abstraktion $\lambda x \rightarrow t$ sind so geregelt: Alle Vorkommen der Variablen x in t gehören zu diesem Präfix λx (sind an dieses x gebunden).

Zum Beispiel ist im Ausdruck $(\lambda x . (y x))$ die Variable x eine gebundene Variable (durch λ gebunden), und y eine freie Variable.

Definition 2.1.8 Die Bindungsregeln in KFPT sind:

- Im Ausdruck $\lambda x . e$ ist x eine gebundene Variable, der Gültigkeitsbereich (Skopus) ist e .
- In der **case**-Alternative $c x_1 \dots x_n \rightarrow e$ wirkt das Pattern wie ein Lambda-Präfix: die Variablen x_i in e werden durch das Pattern gebunden, der Gültigkeitsbereich (Skopus) ist e .

Variablen (bzw. Vorkommen von Variablen), die in keinem Gültigkeitsbereich einer Bindung stehen, sind frei. Da es auch Konflikte geben kann durch Verwendung desselben Variablennamens, gilt in diesen Fällen die Regel: der innerste Bindungsbereich zählt.

Diese Methode nennt man *lexikalische Bindung*.

Wenn eine Variable mehrfach innerhalb eines Ausdrucks vorkommt, können verschiedene Vorkommen zu verschiedenen Bindungen gehören:

Beispiel 2.1.9

$$\lambda x . (x (\lambda x . x))$$

Die genaue Zuordnung kann man durch Indizes sichtbar machen:

$$\lambda x_1 . (x_1 (\lambda x_2 . x_2))$$

.

Um Konflikte durch gleiche Variablennamen zu umgehen, kann man die Variablennamen umbenennen. Dahinter steckt das Prinzip, dass es nur auf die Verbindung zwischen dem gebundenen Namen und dem Vorkommen ankommt. Die Umbenennung macht somit die eigentlichen Bindungsbeziehungen sichtbar, aber sie löst den Konflikt selbst nicht auf.

Ausdrücke die bis auf Umbenennung gleich sind, nennt man auch α -äquivalent.

Definition 2.1.10 Ein Ausdruck ohne freie Variablen heißt geschlossener Ausdruck, anderenfalls offener Ausdruck.

Einen geschlossenen Ausdruck nennen wir manchmal auch KFPT-Programm .

Beispiel 2.1.11 Der Ausdruck

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

ist ein geschlossener Ausdruck. Die Variable x ist frei im Unterausdruck

```
case_List x of {Cons y ys -> ys; Nil -> Nil}
```

Bei Lambda-Ausdrücken, die von der Form $\lambda x_1 . (\lambda x_2 . \dots (\lambda x_n . e))$ sind, werden wir der Einfachheit halber manchmal $(\lambda x_1, \dots, x_n . e)$ schreiben. Bzw in der KFPT-Syntax: $\backslash x_1 x_2 \dots x_n -> e$.

2.1.2 Auswertungsregeln

Was uns jetzt noch fehlt, ist die operationale Semantik, bzw. die Erklärung: wie wertet man Ausdrücke aus? Diese operationale Semantik kann man definieren z.B. durch Angabe einer abstrakten Maschine. Das werden wir später nachholen. Wir gehen hier zunächst einen anderen Weg:

Die operationale Semantik ist als Transformationssystem auf den Ausdrücken definiert.

Die Werte, die man am Ende einer Auswertung eines geschlossenen Ausdrucks erhält, sind spezielle geschlossene Ausdrücke, die als Wert syntaktisch erkennbar sind.

Man muss sich klarmachen, dass man diese Auswertung auf einem Blatt Papier nachvollziehen kann, indem man jeweils die Ausdrücke die nach einer Umformung entstehen, hinschreibt. Andere Erklärungen, wie die Auswertung vorgeht und was passiert usw., sind daraus abgeleitet.

Definition 2.1.12 Ein Wert bzw. eine WHNF (weak head normal form, schwache Kopfnormalform)¹ in KFPT ist ein Ausdruck entweder von der Form

1. $(c t_1 \dots t_n)$, wobei $n = \text{arity}(c)$ und c ein Konstruktor ist, oder
2. eine Abstraktion: $\lambda x . e$.

Wir unterscheiden die Werte nach ihrer Struktur als

- FWHNF, wenn die WHNF eine Abstraktion ist, oder
- CWHNF (constructor WHNF), wenn die WHNF eine Konstruktoranwendung der Form $(c s_1 \dots s_n)$ ist.

¹Eine HNF oder Kopfnormalform ist ein Ausdruck der einen Lambda-Präfix hat und als erste Nichtabstraktion muss $(x t_1 \dots t_n)$ oder $(c t_1 \dots t_n)$ vorkommen, wobei c ein Konstruktor ist. Wir verwenden Kopfnormalformen nicht in der Vorlesung

Im folgenden soll der Ausdruck $s[t/x]$ bedeuten, dass im Ausdruck s die freien Vorkommen der Variablen x durch t ersetzt werden. Hierbei ist als Konvention zu beachten, dass keine freien Variablen "eingefangen" werden (no variable capturing). Dazu muss nur beachtet werden, dass alle Variablen, die in s gebunden werden, von allen freien Variablen in t verschieden sind. Das ist leicht durch Umbenennung von gebundenen Variablen in s zu erreichen.

Hinter der Umbenennung von gebundenen Variablen steckt die Idee, dass es auf den Variablennamen x in $\lambda x.e$ nicht ankommt, nur auf den richtigen Verweis. Eine Umbenennung kann man durchführen, indem man mehrere Umbenennungen der folgenden Form macht:

- In $\lambda x.e$: ersetze x durch eine bisher nicht verwendete Variable (z). In e ersetze alle freien Vorkommen von x durch z : Geschrieben als: $\lambda z.e[z/x]$. Da z bisher nirgendwo verwendet wird, braucht man dabei nicht auf das Einfangen zu achten.
- In der **case**-Alternative $c\ x_1 \dots x_n \rightarrow e$: ersetze x_i durch bisher nicht verwendete und verschiedene Variablen z_i . In e ersetze alle freien Vorkommen von x_i durch z_i : Die neue **case**-Alternative ist: $c\ z_1 \dots z_n \rightarrow e[z_1/x_1, \dots, z_n/x_n]$

Beispiel 2.1.13 *Zum Einfangen von Variablen: Was ist die richtige Ersetzung in $(\lambda x.\lambda y.(x\ z))[y/z]$?*

- *Die falsche Variante ist: $(\lambda x.\lambda y.(x\ y))$. Dies würde einer Funktion entsprechen, die Ihre Argumente aufeinander anwendet.*
- *Die richtige Variante ist: $(\lambda x.\lambda u.(x\ y))$. D.h. hier wurde zunächst umbenannt und y durch u ersetzt. Diese Funktion nimmt das erste Argument, ignoriert ihr zweites Argument, und wendet das erste Argument auf (die freie Variable) y an.*

Die folgenden Auswertungsregeln sollen in allen Programmkontexten gelten, d.h. an jeder Stelle im Programm, außer in Pattern.

Definition 2.1.14 *Auswertungsregeln (Reduktionsregeln):*

$$\text{Beta} \quad \frac{((\lambda x.t)\ s)}{t[s/x]}$$

$$\text{Case} \quad \frac{(\text{case}_T (c\ t_1 \dots t_n) \text{ of } \{ \dots ; c\ x_1 \dots x_n \rightarrow s ; \dots \})}{s[t_1/x_1, \dots, t_n/x_n]}$$

Der Begriff des Redex (reducible expression) kann jetzt definiert werden: Der Unterausdruck s in einem Oberausdruck r , der unmittelbar reduziert werden kann nach obigen Regeln, nennt man, zusammen mit seiner Position, einen Redex in r .

Die so definierten Auswertungsregeln haben noch zuviele Möglichkeiten der Anwendung innerhalb eines Ausdrucks. Z.B. kann man $(\lambda x.x)((\lambda y.y)z)$ an zwei Stellen reduzieren, so dass man noch nicht den Begriff einer deterministischen Auswertung definieren kann.

Deshalb definiert man zunächst eine Normalordnungs-Reduktion (*normal-order-Reduktion*, *Standard-Reduktion*), wozu man Reduktionskontexte braucht.

Definition 2.1.15 *Reduktionskontexte R sind:*

$$R ::= [] \mid (R \ e) \mid (\text{case}_T \ R \ \text{of} \ \{p_1 \rightarrow t_1; \dots; p_n \rightarrow t_n\})$$

Es fällt auf, dass das Loch eines Reduktionskontextes nicht unter einem Konstruktor oder im Rumpf einer Abstraktion ist, ebenso nicht im Argument einer Anwendung.

Jetzt ist die Normalordnungs-Reduktion diejenige, die immer in einem Reduktionskontext den Ausdruck unmittelbar reduziert:

Definition 2.1.16 *Sei $R[s]$ ein Ausdruck, so dass R ein Reduktionskontext ist, wobei s keine WHNF ist, und s reduziert unmittelbar zu t :*

Dann ist $R[s] \rightarrow R[t]$ die Ein-Schritt-Normalordnungsreduktion (Standardreduktion, normal order reduction) von $R[s]$.

Der Unterterm s zusammen mit seiner Position wird auch Normalordnungsredex genannt. Die Reduktionsrelation wird mit dem Marker n versehen. Die transitive und reflexiv-transitive Hülle bezeichnen wir mit $\xrightarrow{n,+}$ bzw. $\xrightarrow{n,}$.*

Die Relation $\xrightarrow{n,}$ bezeichnet man auch als Normalordnungsrelation oder als Auswertung eines Terms.*

Wenn ein geschlossener Term t unter Normalordnung zu einer WHNF reduziert, dann sagen wir, t terminiert und bezeichnen dies mit $t\downarrow$. Anderenfalls divergiert t , bezeichnet mit $t\uparrow$.

Wir sagen, t hat eine WHNF, wenn t zu einer WHNF reduziert. Die WHNF, zu der t unter Normalordnung reduziert, ist eindeutig, aber es gibt i.a. viele WHNFs zu denen ein Term t reduzieren kann. Z.B. $t \equiv (\text{cons } ((\lambda x.x) \ \text{True})) \ \text{Nil}$ ist selbst in WHNF, aber $(\text{cons } \text{True } \text{Nil})$ ist ebenfalls eine WHNF zu t .

Damit haben wir alle zur Auswertung von Ausdrücken notwendigen Begriffe definiert und können jetzt Ausdrücke auswerten.

Beispiel 2.1.17 *Wir betrachten nochmal das Beispiel von oben:*

$(\lambda x \ .x)((\lambda y \ .y)z)$ hat zwei Reduktionsmöglichkeiten, aber nur eine Normalordnungsreduktion:

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

Der zugehörige, eindeutige, Reduktionskontext R ist: $([] \ ((\lambda y \ .y)z))$

Beispiel 2.1.18 $((\lambda x.x) \ \text{Nil}) \xrightarrow{n} \text{Nil}$

$$((\lambda x.\lambda y.x) \ s \ t) \xrightarrow{n} ((\lambda y.s) \ t) \xrightarrow{n} s.$$

Lemma 2.1.19 *Es gilt:*

- Jede unmittelbare Reduktion in einem Reduktionskontext ist eine Normalordnungsreduktion.
- Der Normalordnungsredex und die Normalordnungsreduktion sind eindeutig (falls eine Normalordnungsreduktion existiert).
- Eine WHNF hat keinen Normalordnungsredex und erlaubt keine Normalordnungsreduktion.

Beweis. Das erkennt man an der Definition der Reduktionskontexts und der Normalordnungsreduktion. \square

Unsere Vorgehensweise ist im Sinne von Gordon Plotkin, der eine Programmiersprache als aus vier wichtigen Komponenten bestehend definiert:

- Ausdrücke
- Kontexte
- Werte
- eine Auswertungsrelation auf Ausdrücken.

Das definiert dann u.a. $t \Downarrow$.

Die Theorie der korrekten Transformationen kann man darauf aufbauen:

s, t sind *gleich*, wenn für alle Kontexte C : $C[s] \Downarrow \Leftrightarrow C[t] \Downarrow$

(siehe zweite Semesterhälfte)

2.1.3 Dynamische Typregeln

Definition 2.1.20 *Dynamische Typregeln für KFPT:*

- Für *case*:
 $(\text{case}_T e \text{ of } \{alt_1; \dots; alt_m\})$ ist direkt dynamisch ungetypt
wenn e eine der folgenden Formen hat:
 - $(c a_1 \dots a_{arc})$ und c ist ein Konstruktor, der nicht zum Typ T gehört,
 - $\lambda x . t$.
- für *Konstruktoren*:
 $((c a_1 \dots a_{arc}) e)$ ist direkt dynamisch ungetypt.

Ein Ausdruck e ist dynamisch ungetypt, wenn er sich mittels Normalordnung auf einen Ausdruck reduzieren läßt, der direkt dynamisch ungetypt ist.

Diese Begriffe verwenden wir auch für den KFPT-Normalordnungsredex.

Beachte, dass Haskell eine Typisierung einführt, die restriktiver ist, als nur die Bedingung, dass kein dynamischer Typfehler auftritt.

Wir definieren *wohlgetypt* als den komplementären Begriff.

Das wird später auch für Haskell eingeführt,

2.1.4 Unwind: Suche nach dem Normalordnungsredex

Wir definieren einen Algorithmus zur Bestimmung des Normalordnungsredex mittels Regeln zum Verschieben eines Labels. Sei R das Label:

$$\begin{aligned} C[(s\ t)^R] &\rightarrow C[(s^R\ t)] \\ C[(\text{case } s\ \text{alts})^R] &\rightarrow C[(\text{case } s^R\ \text{alts})] \end{aligned}$$

Starte mit t^R und wende die Regeln solange an, bis keine Anwendung mehr möglich ist.

Danach kann man, falls es möglich ist, die Normalordnungs-Reduktion durchführen:

$$\begin{aligned} \text{Beta } C[(\lambda x.t)^R\ s] &\rightarrow C[t[s/x]] \\ \text{Case } C[(\text{case}_T(c\ t_1 \dots t_n)^R\ \text{of } \{\dots; c\ x_1 \dots x_n \rightarrow s; \dots\})] &\rightarrow C[s[t_1/x_1, \dots, t_n/x_n]] \end{aligned}$$

Wenn keine Normalordnungsreduktion möglich ist, dann gibt es zwei Möglichkeiten:

t ist eine WHNF oder es ist ein dynamischer Typfehler.

Der Fall WHNF kann nur vorkommen, wenn das R -Label des Top-Terms sich durch die Regeln nicht verschieben lässt.

Aus der Sicht eines Normalordnungsredex reduziert die Normalordnung diesen Unterterm solange, bis er in WHNF ist und setzt dann die Reduktion weiter oben fort. Dies ergibt einen Auswertungsalgorithmus auf Ausdrücken, der leicht rekursiv formuliert werden kann.

Beispiel 2.1.21

$$\text{bot} := (\lambda x.(x\ x))\ (\lambda y.(y\ y))$$

ist ein Ausdruck, für den die Normalordnungsreduktion nicht terminiert. Denn die (beta)-Reduktion ergibt:

$$(\lambda x.(x\ x))\ (\lambda y.(y\ y)) \xrightarrow{n} ((\lambda y.(y\ y))\ (\lambda y'.(y'\ y')) \xrightarrow{n} \dots$$

Man sieht, dass die Terme bis auf Umbenennung gleich sind, d.h. sie sind α -äquivalent.

Beispiel 2.1.22 Einige Funktionen, die auf dem Datentyp `Bool` definierbar sind, sind die aussagenlogischen Verknüpfungen `und`, `oder`, `nicht`:

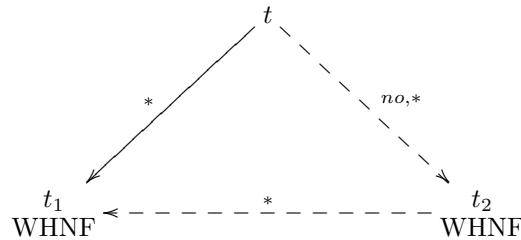
$$\begin{aligned} \text{und} &= \lambda x . \lambda y . \text{case}_{\text{Bool}}\ x\ \text{of } \{\text{True} \rightarrow y; \text{False} \rightarrow \text{False}\} \\ \text{oder} &= \lambda x . \lambda y . \text{case}_{\text{Bool}}\ x\ \text{of } \{\text{True} \rightarrow \text{True}; \text{False} \rightarrow y\} \\ \text{nicht} &= \lambda x . \text{case}_{\text{Bool}}\ x\ \text{of } \{\text{True} \rightarrow \text{False}; \text{False} \rightarrow \text{True}\} \end{aligned}$$

In Haskell sind die logischen Verknüpfungen bereits im Prelude definiert als `&&`, `||`, `not`.

Folgende Aussage, deren Beweis wir verschieben, garantiert, dass jede Auswertung zum richtigen Wert führen kann, wobei die erreichte WHNF etwas weiter ausgewertet sein kann als die mit Normalordnung erreichte.

D.h. um einen Ausdruck auszuwerten, können wir (fast) beliebig reduzieren, bis eine WHNF erreicht ist. Wenn die erreichte WHNF eine Konstruktorkonstante ist, dann ist sie sogar eindeutig.

Satz 2.1.23 (Standardisierung). *Sei t ein (evtl. offener) Ausdruck. Wenn $t \xrightarrow{*} t_1$ mit (Beta) und (case)-Reduktionen, wobei t_1 eine WHNF ist, dann existiert eine WHNF t_2 , so dass $t \xrightarrow{no,*} t_2$, und $t_2 \xrightarrow{*} t_1$.*



Folgerung: es ist korrekt, an beliebigen Positionen zu reduzieren, bis eine WHNF erreicht ist. Wenn die erreichte WHNF eine Konstruktorkonstante ist, dann ist sie sogar eindeutig.

2.1.5 KFP

Wir geben die Syntax von KFP hier zur Information schon mal an. Sie wird genauer besprochen in einem späteren Kapitel

2.1.6 Syntax der funktionalen Kernsprache KFP

Die Kernsprache KFP ist angelehnt an Kernsprachen der Compiler von funktionalen Programmiersprachen und hat möglichst einfache Syntax, und möglichst wenig vordefinierte Funktionen, Namen und Konstrukte. KFP hat ein schwaches Typsystem (Programme mit Typfehlern sind möglich).

Syntax: Es gibt Konstantensymbole, die jeweils eine feste Stelligkeit haben. Diese nennen wir *Konstruktoren*. Die Anzahl der Konstruktoren sei N , die Konstruktoren seien mit $c_i, i = 1, \dots, N$ bezeichnet.

Wir nehmen an, dass es eine Möglichkeit gibt, alle Konstruktoren mit Stelligkeit anzugeben, ohne dass wir diese Methode näher spezifizieren. Zum Beispiel als eine Auflistung aller Konstruktoren mit Stelligkeit.

Definition 2.1.24 *Eine einfache kontextfreie Grammatik für KFP-Ausdrücke (Terme, Expressions EXP) ist:*

$EXP ::= V \quad V \text{ sind Variablen}$
 $| \lambda V . EXP \quad \text{wobei } V \text{ eine Variable ist.}$
 $| (EXP EXP)$
 $| (c EXP_1 \dots EXP_n) \quad \text{wobei } c \text{ Konstruktor ist und } n = ar(c)$
 $| (\text{case } EXP \{Pat_1 \rightarrow Exp_1; \dots; Pat_{N+1} \rightarrow Exp_{N+1}\})$
Hierbei ist Pat_i Pattern zum Konstruktor i , und
 *Pat_{N+1} das Pattern **lambda**.*
 *$(Pat_i \rightarrow Exp_i)$ heißt auch **case-Alternative**.*

$Pat ::= (c V_1 \dots V_{ar(c)}) \mid \text{lambda}$
Die Variablen V_i müssen alle verschieden sein.

Wesentlich ist die andere Struktur des jetzt ungetypten **case**-Konstruktes: Es gibt nur ein **case**, und es sind stets alle Konstruktoren als Alternativen vorhanden, ebenso eine weitere Alternative, die mit dem Pattern **lambda** abgedeckt wird, und die zum Zuge kommt, wenn der zu untersuchende Ausdruck eine Abstraktion ist.

2.2 Rekursive Superkombinatoren: KFPTS

In Haskell kommen noch (rekursive) Superkombinatoren dazu. Die Erweiterung nennen wir KFPTS.

Es gibt Funktionsnamen (Superkombinatornamen) als neue Konstanten, und auf oberster Ebene eine Definitionsmöglichkeit für Funktionen:

Definition 2.2.1 Syntax von KFPTS

Die Syntax der KFPTS-Ausdrücke ist wie KFPT, nur um Superkombinatornamen erweitert.

Es gibt zusätzlich eine Menge von Definitionen der Form

$$\text{Superkombinatorname } V_1 \dots V_n = Exp$$

Folgende Bedingungen müssen eingehalten werden:

- Die Variablen $V_1 \dots V_n$ sind alle verschieden; außerdem kommen im definierenden Ausdruck Exp nur diese Variablen frei vor.
- Die Namensräume für Variablen, Konstruktoren, Kombinatornamen sind disjunkt.
- Jeder Superkombinator wird höchstens einmal definiert

Statt Kombinator sprechen wir auch von Funktionen. Die Stelligkeit von Funktionsnamen f , geschrieben als $ar(f)$, ist die Anzahl der Variablen in der Definition von f .

Als Zusatz eröffnet dies die Möglichkeit, rekursive Funktionen zu definieren, indem die Namen von Funktionen im Rumpf verwendet werden.

Definition 2.2.2 Ein KFPTS-Programm besteht aus:

1. Einer Menge von Typen und Konstruktoren wie in KFPT,
2. einer Menge von Kombinator-Definitionen
3. und aus einem Ausdruck. Dieser kann auch mittels der `main`-Konvention als

$$\text{main} = \text{Exp}$$

definiert werden.

Reduktionskontexte in KFPTS sind analog wie in KFPT definiert, nur erweitert um Superkombinatornamen in Ausdrücken.

Der Wert des Programms ist der Wert des `main`-Ausdrucks nach Auswertung. Die Auswertung eines Superkombinatorprogramms erfolgt durch eine abgewandelte Beta-Reduktion, die erst aktiv wird, wenn ausreichend viele Argumente da sind. Die Case-Reduktion schreiben wir nochmal hin:

Definition 2.2.3 Die Auswertung von Ausdrücken in KFPTS geschieht nach den Regeln:

$$\text{Beta} \quad \frac{((\lambda x.t) s)}{t[s/x]}$$

$$\text{SK-Beta} \quad \frac{(\mathbf{f} a_1 \dots a_n)}{r[a_1/x_1, \dots, a_n/x_n]} \quad \begin{array}{l} \text{wenn } \mathbf{f} \text{ definiert ist als: } \mathbf{f} x_1 \dots x_n = r \\ \text{d.h. wenn } ar(\mathbf{f}) = n \end{array}$$

$$\text{Case} \quad \frac{(\text{case}_T (c t_1 \dots t_n) \text{ of } \{ \dots ; c x_1 \dots x_n \rightarrow s ; \dots \})}{s[t_1/x_1, \dots, t_n/x_n]}$$

wobei $n = ar(c)$ und c ist T-Konstruktor

Diese Auswertung soll in allen KFPTS-Kontexten erlaubt sein.

Die KFPTS-Normalordnungsreduktion ist diejenige Reduktion, die innerhalb eines KFPTS-Reduktionskontexts unmittelbar mit Case, Beta oder SK-Beta reduziert.

Verwendet man R-labeling wie in KFPT, dann braucht man nur die extra Regel: $C[(SK^R s_1 \dots s_{ar(SK)})] \rightarrow C[r_{SK}[s_1/V_1, \dots, s_n/V_n]]$

Da die Superkombinatorreduktion SK-Beta etwas anders definiert ist (eine Funktion benötigt alle Argumente, bevor diese reduziert), sind auch WHNFs (weak head normal forms) anders definiert:

Definition 2.2.4 Eine KFPTS-WHNF ist entweder

- ein Ausdruck $(c t_1 \dots t_n)$; oder
- ein Ausdruck $(f t_1 \dots t_n)$, wobei $n < ar(f)$; oder
- ein Ausdruck $\lambda x.s$

Die formalen Eigenschaften der operationalen Semantik und die Beziehungen zwischen KFP- KFPT- und KFPTS-Reduktion behandeln wir in einem extra Kapitel.

2.2.1 Kernsprachen und strict und seq

(strict f t) ist ein Operator, der t f strikt im ersten Argument macht.

$$\frac{\text{Wert}(f t) = a; t \Downarrow}{\text{Wert}(\text{strict } f t) = a} \quad \text{und} \quad \frac{t \Uparrow}{(\text{strict } f t) \Uparrow}$$

(seq s t) wertet s und t aus, und gibt den Wert von t zurück:

$$\frac{\text{Wert}(t) = a; s \Downarrow}{\text{Wert}(\text{seq } s t) = a} \quad \text{und} \quad \frac{s \Uparrow}{(\text{seq } s t) \Uparrow}$$

strict und seq sind gleichwertig:

$$\text{strict } f t = \text{seq } t (f t) \qquad \text{seq } s t = \text{strict } (\lambda x \rightarrow t) s$$

In KFP sind beide definierbar:

```
strict f = \x . case x of {p_1 -> f x; ... ;p_N -> f x;p_{N+1} -> f x}
```

Betrachtete Kernsprachen

KFP	Abstraktionen, Anwendungen, Konstruktoranwendungen jedes case hat alle Pattern und ein extra Pattern lambda
KFPT	Abstraktionen, Anwendungen, Konstruktoranwendungen jedes case hat nur Pattern eines Typs
KFPTS	Wir KFPT; Superkombinatoren sind erlaubt.
KFPT+ seq	KFPT und seq wird als definiert angenommen
KFPTS+ seq	KFPTS und seq wird als definiert angenommen

Beachte: In KFPT, KFPTS sind strict und seq nicht definierbar.

2.3 Haskell: nächste Stufe der Einführung

Wir führen ab jetzt die Haskell-Konzepte ein, und deuten teilweise an, wie man diese auf KFPTS aufsetzen kann. Wenn wir uns vergewissert haben, dass dies geht und verstanden haben, wie es geht, dann nehmen wir die Möglichkeit der

Implementierung als Fakt an und benutzen nur noch die Haskell-Sprech- und Schreibweisen.

Die Syntax von Haskell kann aus Büchern, oder dem Online-Handbuch entnommen werden, so dass wir diese nur teilweise in diesem Skript wiederholen.

2.3.1 Polymorphe Typen

In diesem Paragraph führen wir informell polymorphe Typen ein, deren Schreibweise und Verwendung, so dass man Typen von Funktionen verstehen und deuten kann. Das zugrundeliegende Typsystem ist der parametrische Polymorphismus (Milners Typsystem). Dies soll dem Verständnis der Programmierung in Haskell dienen.

Wir machen das etwas allgemeiner als Haskell, da man nur dann die Haskell-Beschränkungen verstehen kann. Haskell benutzt nämlich nicht die allgemeinste Form eines parametrischen polymorphen Typsystems, sondern eine pragmatisch beschränkte Variante.

Die weitergehenden Typklassen und die polymorphe Typprüfung werden erst später besprochen. In KFPT oder KFPTS gibt es keine Typen von Ausdrücken.

KFPTSP: polymorphe Typen

Als passendes Fragment von Haskell könnte man etwa KFPTSP definieren. Der Unterschied zwischen KFPTS und KFPTSP soll sein, dass jeder KFPTSP-Ausdruck auch ein KFPTS-Ausdruck ist, und in "KFPTSP" jeder Ausdruck einen polymorphen Typ hat, aber nicht alle KFPTS-Ausdrücke.

Die Syntax von polymorphen Typen ist:

$$T ::= V \mid (TC \ T_1 \ \dots \ T_n) \mid (T_1 \rightarrow T_2)$$

wobei V eine Typvariable, T ein Typ, und TC ein parametrisierter Typname (Typkonstruktor) ist. Typkonstruktoren sind Namen wie `Bool`, `List`, wobei `List` noch einen Parameter hat: den Typ der Elemente der Liste. Statt `List a` schreibt man in Haskell `[a]`. Wir werden dies hier auch verwenden.

Die (Daten-) Konstruktoren zu einem Typ müssen dann im Vorspann vom Programmierer mit dem gewünschten Typ versehen werden. Dieser Typ kann auch rekursiv definiert sein. Z.B. bei der Definition des Typs `Liste` muss man die Konstruktoren `Nil`, `Cons` bzw. `[]`, `:` mit den richtigen Typen versehen. Man kann parametrisierte Datentypen selbst definieren, z.B. binäre Bäume mit Typ a von Markierungen der Blätter.

Wir schreiben mit weniger Klammern $T_1 \rightarrow T_2 \rightarrow T_3$ statt $T_1 \rightarrow (T_2 \rightarrow T_3)$.

Wir erläutern das Typkonzept an Beispielen. Der Typ einiger Ausdrücke, wie sie in Haskell gefunden bzw. definiert werden.

```

True   :: Bool
False  :: Bool
&&     :: Bool → Bool → Bool
||     :: Bool → Bool → Bool

Cons   :: a → [a] → [a]  a ist Typvariable
Nil    :: [a]             a ist Typvariable
(:)    :: a → [a] → [a]  a ist Typvariable
[]     :: [a]             a ist Typvariable

```

Als (vereinfachte) Typregeln braucht man sich nur zu merken:

- $\frac{f :: a \rightarrow b, s :: a}{(f\ s) :: b}$
- $\frac{s :: T}{s :: T'}$ wobei $T' = \sigma(T)$ und σ eine Einsetzung von Typen für Typvariablen ist.
- $\frac{s :: T, t_1 : a, \dots, t_n :: a}{(\text{case}_T\ s\ \text{of}\ \{alt_1 \rightarrow t_1; \dots\}) :: a}$.
Beachte, dass man in der vollständigen Typregel die Variablen des Pattern berücksichtigen muss.

In Zukunft geben wir Typen an, wenn die Ausdrücke in Haskell definierbar sind und dort einen Typ haben.

Beispiel 2.3.1 Die Funktion `&&` hat den Typ `Bool → Bool → Bool`. Den Typ des Ausdrucks `True && False` erhält man folgendermaßen unter Verwendung der Regeln:

$$\frac{\frac{\&\& :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}; \quad \text{True} : \text{Bool}}{(\&\& \text{True}) :: \text{Bool} \rightarrow \text{Bool}}; \quad \text{False} : \text{Bool}}{(\&\& \text{True False}) :: \text{Bool}}$$

Auf ähnlich Weise kann man aus den Typen $(:) :: \alpha \rightarrow [\alpha] \rightarrow [\alpha]$, und $[] :: [\alpha']$ nach mehrfacher Verwendung der Regeln den Typ `[Int]` für `1 : 2 : 3 : []` erhalten. Die Schlussweise ist etwas allgemeiner, da man noch Typvariablen instanziiieren muss:

$$\frac{\frac{(\:) :: \alpha \rightarrow [\alpha] \rightarrow [\alpha], \quad 1 : \text{Int}}{(1\ :) :: [\text{Int}] \rightarrow [\text{Int}]}; \quad [] :: [\alpha']}{(1\ : []) :: [\text{Int}]}}$$

Die Klammerregeln für Typen und Ausdrücke sind konsistent:

Sei $f :: T_1 \rightarrow (T_2 \rightarrow T_3)$ und $a :: T_1, b :: T_2$. Dann gilt mit den Typregeln: $(f\ a) :: (T_2 \rightarrow T_3)$ und $((f\ a)\ b) :: T_3$.

Lässt man alle Klammern weg, dann ist das konsistent: $f :: T_1 \rightarrow T_2 \rightarrow T_3$ und

$f a b :: T_3$, wobei $a :: T_1, b :: T_2$.

Bei Konstruktoren und Konstruktoranwendung und Pattern kann man den Typ so schreiben, als sei eine Konstruktoranwendung (bzw. ein Pattern) eine iterierte Anwendung. z.B. `Cons: a -> [a] -> [a]`

2.3.2 Arithmetische Operatoren in Haskell

Die arithmetischen Datenstrukturen und Operatoren sind in Haskell verfügbar. Der korrekte und sichere Umgang mit dem vollen Umfang der Funktionalität erfordert ein Verständnis der Typisierung in Haskell.

Es gibt beschränkte ganze Zahlen (`Int`), unbeschränkte ganze Zahlen (`Integer`), Gleitkommazahlen (`Float`), doppelt genaue Gleitkommazahlen (`Double`), rationale Zahlen (`Ratio α`).

Beispiel 2.3.2

```

2 * 2                = 4
2.0/3.0             = 0.666666667
(1%2)/(3%2)         = (1%3) :: Ratio Integer
(123456789 :: Int) * (987654321 :: Int) = -67153019 :: Int
123456789 * 987654321 = 121932631112635269 :: Integer
    
```

Die numerischen Typen sind in der Typklasse `Num` zusammengefasst. Man kann die arithmetischen Operatoren `+`, `-`, `*` dann auf allen Objekten mit Typ aus der Typklasse `Num` verwenden.

Z.B. ist der Typ von `(+)`: `Num a => a -> a -> a`

Das soll bedeuten: `(+)` hat alle Typen `a -> a -> a`, wenn `a` ein numerischer Typ ist. D.h. z.B. den Typ `Int -> Int -> Int`, `Integer -> Integer -> Integer`, `Double -> Double -> Double`.

Es gibt weitere Typklassen, z.B.: `Eq` für alle Typen, deren Objekte einen Gleichheitstest `==` erlauben, und `Show` für alle Typen, deren Objekte eine Druckmethode haben.

2.3.3 Arithmetische Funktionen: Implementierung in KFPT

Die Implementierung der Haskell-Arithmetik in KFPTS kann man sich entweder als Datentyp und Funktionen vorstellen, die endlich viele Zahl-Konstanten haben, und für jede Kombination einen Fall. Damit lässt sich im Prinzip alles implementieren, nur der Typ `Integer` macht ein Problem, da er unendlich viele Zahlkonstanten bräuchte.

Die Implementierungsmethode dazu sind die sogenannten Peanozahlen, die man mit zwei Konstruktoren aufbauen kann.

Beispiel 2.3.3 *Peano-Kodierung von Zahlen vom Haskell-Typ `Integer` mit Typ `Pint`. Der Typ `Pint` hat eine nullstellige Konstante `0` und einen einstelligen Konstruktor `S`. Dann kodieren `0, (S 0), S(S 0), S(S(S 0)), ...` die nicht-*

negativen ganzen Zahlen $0, 1, 2, 3, \dots$. Die passenden Typen sind: $0 :: \text{Pint}$, $S :: \text{Pint} \rightarrow \text{Pint}$.

Damit sind arithmetische Funktionen auf allen nicht-negativen ganzen Zahlen in KFPTS leicht zu definieren:

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

istZahl x = case x of {Zero -> True; Succ y -> istZahl y}

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

Der Typ der Funktionen ist:

- $\text{istZahl} :: \text{Pint} \rightarrow \text{Bool}$
- $\text{peanoPlus} :: \text{Pint} \rightarrow \text{Pint} \rightarrow \text{Pint}$
- $\text{pplus} :: \text{Pint} \rightarrow \text{Pint} \rightarrow \text{Pint}$

Die Funktion `istZahl` dient dem Zweck, die Eingabe vorher vollständig auszuwerten und somit auch zu prüfen und sicherzustellen, dass eine Zahl eingegeben wurde. Genauer: dass das Ziel einer korrekten Implementierung eingehalten wird.

Wenn zwei Funktionen gleich sein sollen, dann müssen bei gleicher Eingabe die gleichen Ergebnisse berechnet werden, ebenso muss das Terminierungsverhalten gleich sein.

Mit `istZahl` kann man erreichen, dass auch das Terminierungsverhalten korrekt ist. Insbesondere kann man erreichen, dass sich die hier definierten Funktionen wie arithmetische Funktionen verhalten. Z.B. soll sowohl $(+ \text{ bot } 1)$ als auch $(+ 1 \text{ bot})$ nicht terminieren.

Die Trennung in `plus` und Erzwingung der Auswertung dient dazu, zu demonstrieren, dass man auch ein nicht-striktes `plus` definieren kann, welches mit sogenannten partiellen Zahlen umgehen kann. Partielle Zahlen sind hierbei z.B. $(S \text{ bot})$ oder $\text{unendlich} = (S \text{ unendlich})$.

Beispiel 2.3.4 Die Funktion (das Prädikat) \leq zum Vergleichen von Zahlen kann man in KFPTS auf dem Typ `Pint` definieren als:

```
peanoleq x y =
  (istZahl x) &&
  (istZahl y) &&
  (case x of
    {Zero -> True;
     Succ xx -> case y of {Zero -> False; Succ yy -> peanoleq xx yy}})
```

Beispiel 2.3.5 *Die Fakultätsfunktion mit Definition*

```
fak 0 := 1
fak 1 := 1
fak n := 1*2* ... *n
```

Die Fakultätsfunktion auf Peanozahlen, wird mit Superkombinatoren in KFPTS rekursiv so definiert (hier die Kodierung bereits in Haskell):

```
pmal x y = case x of Zero -> Zero
              Succ z -> pplus (pmal z y) y
```

```
pfak x = case x of Zero -> Succ Zero
              Succ z -> (pmal x (pfak z))
```

Die Haskell-Definition auf den in Haskell implementierten Zahlen ist:

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

Beispiel 2.3.6 *Ein Prädikat schaltjahr kann man unter Verwendung des Infixoperator mod, der die Reste bei Division als Resultat hat, ganz einfach definieren als:*

```
schaltjahr x = ((x 'mod' 4) == 0) &&
                ((x 'mod' 100 /= 0) || ((x 'mod' 400) == 0))
```

Beispiel 2.3.7 *Die Funktion, die Zahlen quadriert kann man definieren als:*

$$\text{quadrat } x = (x * x)$$

Die Komposition von Funktionen (mit einem Argument) kann man definieren als:

$$\text{comp } f \ g \ x = (f (g \ x))$$

In Haskell wird Komposition durch einen Punkt als Infixfunktion geschrieben, d.h. statt (comp f g) schreibt man f . g

2.3.4 Tupel, Paare

Die Syntax von Haskell erlaubt Tupel einfach als eingeklammerte und durch Komma getrennte Objekte zu schreiben: Ein 5-Tupel kann man schreiben als (1,2,3,4,5).

Die Übersetzung nach KFPT ist: es gibt einen Typ `Tuple15` mit einem Konstruktor `Tuple15`, der Stelligkeit 5 hat. Das heißt, pro Länge eines Tupels gibt es einen eigenen Typ und einen extra Konstruktor.

Der polymorphe Typ eines Tupelkonstruktors ist z.B.

$$\text{Tupel5} :: a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow (a_1, a_2, a_3, a_4, a_5)$$

Hierbei sind a_i Typvariablen.

In Haskell gibt es auch ein null-stelliges Tupel, geschrieben `()`. In KFPT ist das einfach ein Typ `NullTupel` mit einem einzigen Konstruktor `NullTupel`. Es gibt in Haskell kein einstelliges Tupel, was eher pragmatische Gründe hat, denn es wäre sehr verwirrend, wenn man Klammern mit mehreren Bedeutungen versieht: Z.B.: Ist `(1 + 1)` der Ausdruck `1 + 1`, oder das 1-Tupel aus dem Ausdruck `1 + 1`?

Man kann für Tupel *Selektoren* definieren, die die einzelnen Komponenten aus dem Tupel liefern. Als Beispiel zeigen wir das für Paare:

Beispiel 2.3.8 *In Haskell sieht das so aus:*

```
fst x = case x of {(u, v) -> u}
snd x = case x of {(u, v) -> v}
```

oder einfacher, aber äquivalent:

```
fst (u, v) = u
snd (u, v) = v
```

Die Kodierung in KFPTS erfordert die Angabe des Typs:

```
fst x = case_Tupel2 x of {Tupel2 u v -> u}
snd x = case_Tupel2 x of {Tupel2 u v -> v}
```

Ein Beispiel für die sinnvolle Verwendung von Paaren ist die Kombination von Werten als Rückgabewerte von Funktionen.

Es gilt z.B. dass Tripel verschieden sind von einem verschachtelten Tupel: $(1, 2, 3) \neq ((1, 2), 3) \neq (1, (2, 3)) \neq (1, 2, 3)$

2.4 Listen: Datenstruktur und Operationen

In diesem Kapitel werden Listenoperationen definiert und erklärt. Das Ziel dieses Abschnitts ist, listenverarbeitende funktionale Programme (insbesondere Haskell-Programme) zu verstehen und selbst schreiben zu können.

Die entsprechenden Haskell-Definitionen kann man im Buch von Bird bzw. von Bird & Wadler oder im Prelude von Haskell bzw. Hugs finden. Wir geben teilweise die KFPTS-Kodierung an.

Der Datentyp Liste hat die Konstruktoren `Cons`, `Nil`, wobei im folgenden, analog zur Haskell-Syntax statt `Cons` auch `“:` verwendet werden kann, wobei dieser Konstruktor in Infixschreibweise verwendet wird. Eine Liste schreibt man z.B. als `[1, 2, 3]`. Damit konsistent ist die Schreibweise `[]` für `Nil`.

Funktionsdefinition in Haskell und case

Es gibt ein `case` Konstrukt in Haskell, das allerdings keine explizite Typmarkierung hat. Die Typmarkierung kann vom Typchecker errechnet werden, wenn der Typchecker die Typen der beteiligten Ausdrücke ermittelt.

Funktionsdefinitionen in Haskell werden meist mit einer Fallunterscheidung kombiniert. D.h. ein oberstes `case` wird in mehrere Definitionsgleichungen mit Pattern übersetzt.

Z.B.

```
letztes_element []      = error "zu spaet"
letztes_element (x:xs) = if xs == [] then x
                        else letztes_element xs
```

In Haskell ist es möglich, die Alternativen eines obersten `case` in einer Definition direkt in den Argumenten anzugeben, und die verschiedenen Alternativen als alternative Definitionen anzugeben. Die Pattern dürfen auch zusammengesetzt, d.h. geschachtelt sein, es dürfen Joker (`.`) vorkommen, aber die Variablen im Pattern müssen verschieden sein. Dies ist normalerweise direkt als ein `case` übersetzbar, evtl. geschachtelt. D.h. für jede Funktion können mehrere Definitionsalternativen hingeschrieben werden, wobei jede Definition einen anderen Fall, unterschieden durch Pattern, behandelt. Eine saubere Programmierung ist die Fallunterscheidung mit disjunkten Pattern. Haskell erlaubt auch nicht-disjunkte Pattern, wobei die Entscheidung, welcher Fall eintritt, von oben nach unten getestet wird.

Die Unterscheidung nach den Argumenten können durch sogenannte Wächter (guards) an den Fällen noch in der Handhabung erleichtert werden, d.h. man kann noch Bedingungen formulieren. Die Verwendung und Übersetzung nach KFPTS wird ggfs. durch Beispiele erläutert.

Ein Haskell-Pattern ist folgendermaßen am sinnvollsten nach KFPT zu übersetzen. Eigentlich wird durch diese Übersetzung die exakte operationale Semantik erst festgelegt.

Eine Definition

```
f~p_1 = e_1
f~p_2 = e_2
.....
f~p_n = e_n
```

wobei die Pattern `p_i` disjunkt sind, kann man einfach mit einem geschachtelten `case` übersetzen. Wenn die Pattern flach sind, dann ergibt sich ein einziges `case`, wobei die fehlenden Fälle als Ausdruck einfach `bot` haben. Damit diese Übersetzung machbar ist, muss eine Typprüfung ergeben, welcher Typ gemeint ist.

Im Haskell sind die Pattern sequentiell gemeint. D.h. die Pattern werden von oben nach unten abgearbeitet. Auch dies ist leicht nach KFPT zu übersetzen, indem man hinreichend viele `case`-Ausdrücke schachtelt.

Sind die Pattern geschachtelt, dann wird aus jedem Pattern ein geschachteltes **case**. Hierbei muss man sich festlegen, wie die Auswertung innerhalb eines Patterns gesteuert wird: Normalerweise ist das ein depth-first Durchlauf.

2.4.1 Listen von Zahlen

Einige einfache Funktionen, die Listen von Zahlen bereitstellen, sind in Haskell definiert. Z.B.

```
[1..10] ----> [1,2,3,4,5,6,7,8,9,10]
[1..] ----> [1,2,3,4,5,6,7,8,9,10,11,.....
```

Das ist eine Kombination von syntaktischen Extras mit einer Funktionsdefinition. Die Funktionen sind als **upto** und **from** definiert. Zu beachten ist, dass die zweite Funktion eine Liste darstellt, die kein **Nil** am rechten Ende hat, d.h. deren volle Auswertung als Liste nicht terminiert. Wir nennen solche Objekte auch "unendliche Listen".

Die Definitionen (nach KFPTS übersetzbar) sind:

```
upto m n = if m > n then []
           else m : (upto (m+1) n)
```

```
from m = m : (from (m+1))
```

2.4.2 map, append, filter, reverse, fold, ...

map: Die Funktion **map** wendet eine Funktion auf jedes Element einer Liste an und konstruiert als Ergebnis die Liste der Resultate.

```
map f xs = case xs of {Nil -> Nil; Cons h t -> Cons (f h) (map f t)}
```

Z.B. `map quadrat [1..4]` ergibt `[1,4,9,16]` bei Auswertung zur Normalform (oder Verwendung von `show`).

Die Haskell-Definition ist:

```
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

Der Typ ist: `map :: (a -> b) -> [a] -> [b]`.

Die Funktion **append**, Infix geschrieben als `++` ist definiert als:

```
++ xs ys = case xs of {Nil -> ys; Cons h t -> Cons h (t ++ ys)}
```

Sie hängt zwei eingegebene Listen aneinander: `[1,2,3] ++ [4,5,6] -> [1,2,3,4,5,6]`. Die Haskell-Definition ist:

```
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Der Typ ist: $++ :: [a] \rightarrow [a] \rightarrow [a]$.

length Die Länge einer Liste definiert man so:

```
length xs = case xs of {Nil -> 0; Cons h t -> (1 + (length t))}
```

Z.B ergibt `length [1,2,3,4]` $\rightarrow 4$.

Die Haskell-Definition ist:

```
length []      = 0
length (_:t)   = 1 + (length t)
```

Der Typ ist `length :: [a] -> Int`

Diese Definition der Länge ist eine Abbildung in die Peanozahlen.

zip / unzip Die Funktionen `zip`, `unzip` behandeln Listen von Paaren, `zip` macht aus zwei Listen eine Liste von Paaren, während `unzip` aus einer Liste von Paaren ein Paar von zwei Listen macht.

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

unzip []      = ([], [])
unzip ((x,y):xs) = (x:xl, y:yl)
                  where (xl,yl) = unzip xs
```

Die Funktion `unzip` liefert

```
unzip [(1,2), (3,4), (5,6)] ---> ([1,3,5], [2,4,6])
```

reverse Die Funktion `reverse` vertauscht in einer Liste die Reihenfolge der Elemente.

```
reverse xs = case xs of {Nil -> Nil; Cons h t -> ((reverse t) ++ [h])}
```

Z.B. `reverse [1, 2, 3]` $\rightarrow [3, 2, 1]$

Die Haskell-Definition ist:

```
reverse []      = []
reverse (h:t)   = (reverse t) ++ [h]
```

Der Typ ist: `reverse :: [a] -> [a]`.

Diese Version der Funktion `reverse` ist quadratisch in der Länge der Liste, falls die ganze Resultatliste ausgewertet wird, da das `append` jede Teilliste durchgehen muss. Eine effizientere Methode zum Umdrehen ist die Verwendung eines Stacks:

```
reverse x          = rev_accu x []
rev_accu xs stack = case xs of {[] -> stack;
                               h:t -> (rev_accu t (h:stack))}
```

Die Funktion `rev_accu` ist endrekursiv (tail-recursive), d.h. die rekursiven Aufrufe sind alle wieder von der Art `rev_accu s t`, d.h. man kann hier eine Compileroptimierung machen, die bewirkt, dass der Stack der Funktionsaufrufe am Ende nicht wieder abgerollt werden muss.

Transpose: Transponiert eine Matrix, wenn diese als Liste von Listen dargestellt ist. Die Darstellung ist:

$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ wird als `[[1,2],[3,4]]` dargestellt.

```
hd xs          = case xs of {h:t -> h; [] -> bot}
tl xs          = case xs of {h:t -> t; [] -> bot}

transpose xss = case xss of
  {[] -> bot;
   h:t ->
     case h of {[] -> [];
                h:t -> (map hd xss)
                       : (transpose (map tl xss))}}
```

Haskell-Definition:

```
transpose ([] : rest) = []
transpose x           = (map hd x) : (transpose (map tl x))
```

Die Typen sind:

```
hd :: [a] -> a
tl :: [a] -> [a]
transpose :: [[a]] -> [[a]]
```

`transpose [[1,2],[3,4],[5,6]]` ergibt bei der Auswertung nacheinander:

```
[1,3,5] : transpose [[2],[4],[6]]
→ [[1,3,5],[2,4,6]]
```

Beachte: es gibt in Haskell auch einen mitgelieferten Datentyp `Array`, der sich analog zu anderen Programmiersprachen über Indizes ansprechen lässt, dessen interne Implementierung bestimmte Optimierungen erlaubt.

Von jetzt an geben wir die Funktionen nur noch in Haskell-Notation an. Es sollte für den Leser jetzt kein Problem mehr sein, zwischen KFPTS und den Funktionen in Haskell-Definition hin und her zu übersetzen.

Bei `transpose` ist auch zu sehen, dass nicht alle Fälle im `case` angegeben sind. Die Konvention in Haskell ist, das syntaktisch zuzulassen, aber einen Laufzeitfehler zu melden, wenn ein unpassender Konstruktor im `case` auftaucht.

vectoradd Damit könnte man eine Vektoraddition programmieren. Die erste Version ist

```
vectoradd_1 xs ys = map vadd (transpose [xs, ys])
vadd [x,y]      = x + y
```

Der Typ ist: `vectoradd_1 :: [Int] -> [Int] -> [Int]`. Das ist ungünstig programmiert, da z.B. ständig Listen konstruiert und wieder dekonstruiert werden müssen. Besser ist die Benutzung einer Funktion, die zwei Listen gleichzeitig bearbeitet. Eine andere Möglichkeit ist es auf eine bessere Optimierung wie "deforestation" zu hoffen.

```
map2 f [] []          = []
map2 f (hx:tx) (hy:ty) = (f hx hy) : (map2 f tx ty)
map2 f _ _           = error "Listen verschieden lang"
vectoradd_2          = map2 (+)
```

Der Typ ist: `map2 :: (a -> b -> c) -> [a] -> [b] -> [c]`.

let, where

Wir führen ein neues Konstrukt von Haskell ein: **where** hat die gleiche Bedeutung wie ein **let**, ist aber nachgestellt.

```
let x1 = t1; ...; xn = tn in Exp
```

bzw.

```
Exp where x1 = t1; ...; xn = tn
```

Diese Konstrukte binden den Wert von t_i an x_i , der dann unter dem Namen x_i in `Exp` verwendet werden kann. Die Auswertung erfolgt erst, wenn der Wert benötigt wird. In Haskell sind `let` und `where` rekursiv. D.h. der Bindungsbereich der x_i ist t_1, \dots, t_n, Exp .²

Ein nicht-rekursives `let` (**where**) könnte man als Lambda-Ausdruck darstellen:

```
let x1 = t1; ... ; xn = tn in Exp
```

entspricht dann (in etwa)

```
(\x1 ... xn . Exp) t1 ... tn
```

Für rekursive `let` benötigt man die gleichen Techniken wie für die Auflösung rekursiver Definitionen.

Die Komposition von zwei einstellig Funktionen wird als Punkt (`.`) geschrieben:

²Beachte, dass im Haskell-Report die Verwendung des **where** eingeschränkt ist.

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

filter filtert Elemente aus Listen, wobei diese Elemente ein bestimmtes Prädikat erfüllen müssen. D.h., eine Testfunktion mit Resultattyp Bool.

```
filter pred [] = []
filter pred (h:t) = if pred h then h:rest
                   else rest
                   where rest = filter pred t
```

Der Typ ist: `filter :: (a -> Bool) -> [a] -> [a]`
 Damit kann man eine Funktion definieren, die alle Elemente entfernt, die das Prädikat *pred* erfüllen:

```
remove p xs = filter (\x -> not (p(x))) xs
```

Oder kürzer geschrieben:

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

Quicksort Wir können jetzt ganz einfach quicksort definieren (wenn wir von Typklassendeklarationen mal absehen)

```
partition p [] = ([],[])
partition p (h:t)
  | (p h)      = (h:oks, noks)
  | otherwise = (oks, h:noks)
               where (oks, noks) = partition p t
```

```
quicksort [] = []
quicksort (h:t) = quicksort kleine ++ (h : (quicksort grosse))
                 where (kleine, grosse) = partition (< h) t
```

Alternativ, unter Verwendung von filter:

```
qs [] = []
qs (h:t) = qs (filter (<= h) t) ++
           (h : (qs (filter (> h) t)))
```

FOLDR und FOLDL

Diese Funktionen benutzt man, um alle Elemente einer Liste zu verknüpfen, z.B.

- Summe aller Elemente einer Liste
- Produkt aller Elemente einer Liste

- Vereinigung aller Mengen (in einer Liste)
- Alle Listen in einer Liste von Listen zusammenhängen.

Dies kann man auf zwei Arten machen:

1. Summe von (1, 2, 3, 4) in der Form $((1 + 2) + 3) + 4$
2. Summe von (1, 2, 3, 4) in der Form $1 + (2 + (3 + 4))$

Um die Funktionalität allgemein zu haben, geht man von einer binären, assoziativen Operation `op` und einem zugehörigen Einheitselement `e` aus:

```
foldr op e [] = e
foldr op e (h:t) = h 'op' (foldr op e t)
```

Man sieht: das entspricht der zweiten Variante, allerdings in der Version, dass die Summe von [1, 2, 3, 4] in der Form $(1 + (2 + (3 + (4 + 0))))$ berechnet wird. Diese Berechnung kann man sehen als Operation auf einer Liste, die `Cons` durch eine zweistellige Funktion ersetzt und `Nil` durch ein neutrales Element:

```
1 : (2 : (3 : (4 : []))) → (1 + (2 + (3 + (4 + 0))))
```

Weitere abgeleitete Funktionen sind:

```
summe = foldr (+) 0
produkt = foldr (*) 1
concat = foldr (++) []
```

`concat` macht eine Liste von Listen zu einer flachen Liste mittels `append`.

Die andere Variante ist `foldl`, die aber nur für endliche Listen wie gewünscht funktioniert.

```
foldl op e [] = e
foldl op e (h:t) = (foldl op (e 'op' h) t)
```

Dies ergibt bei Summe mit der Definition:

```
summe_l = foldl (+) 0
summe_l [1,2,3,4] → (((0+1)+2)+3)+4). D.h. die Summation fängt vorne an.
```

Die Funktion `foldl` kann ihre Stärke in bestimmten Anwendungen wie z.B. für die Summation von Listen von Zahlen erst voll ausspielen, wenn sie so abgeändert wird, dass sie strikt im zweiten Argument ist. D.h. diese Variante von `foldl` sollte zuerst das zweite Argument zur WHNF auswerten, und dann mit der Normalordnung weitermachen. In diesem Fall kann die Auswertung sofort einfache Additionen ausführen lassen, statt sich zu merken, dass diese Addition auszuführen ist. Dadurch wird der Speicherbedarf bei der Summation von endlichen Listen von Zahlen konstant.

Generell sollte man bei sauberer Programmierung auf solche Veränderungen verzichten, da die Optimierung durch einen Compiler zumindest prinzipiell

diese Abänderung selbst erzeugen kann.

Die Funktion `strict`, angewendet auf eine Funktion f , ergibt als `(strict f)` eine Funktion, die wie f ist, aber strikt in ihrem ersten Argument. D.h. sie wertet zuallererst ihr erstes Argument zu WHNF aus. Eine ähnliche Funktion ist `seq`, die Infix verwendet wird, und zuerst das linke und dann das rechte Argument auswertet. Beide sind durch die jeweils andere Funktion definierbar. Als Hinweis zur operationalen Semantik von `strict`: Der Operator `strict` lässt sich in allgemeiner Form in KFP (aber nicht in KFPT) simulieren: Wenn f eine einstellige Funktion ist, dann gilt folgendes, wobei unter den Pattern alle Konstruktoren vertreten sind, ebenfalls `lambda` als Pattern für eine Abstraktion:

```
strict f = \x . case x of {p_1 -> f x; ... ;p_N -> f x;p_{N+1} -> f x}
```

Die neue Version von `foldl` sieht dann so aus.

```
foldls op e [] = e
foldls op e (h:t) = strict (foldls op) (e 'op' h) t
```

Typen von `foldl` und `foldr` sind:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Diese Typen sind zunächst etwas überraschend, da man z.B. eher den Typ

$$(a \rightarrow a \rightarrow a) \rightarrow a \rightarrow [a] \rightarrow a$$

erwartet hätte.

Dies bedeutet, dass diese Funktionen eine allgemeinere Verwendung haben können, als wir vorgesehen haben. Z.B. kann man damit auch die Länge einer Liste definieren:

```
lengthfl = foldl ladd 0
          where ladd x _ = 1+x
```

In diesem Fall ist allerdings `foldr` und `foldl` nicht so schnell austauschbar, da die Funktion `ladd` nicht symmetrisch in den Typen ist.

```
lengthfr = foldr radd 0
          where radd _ x = 1+x
```

ist analog.

scanr, scanl

Wenn man die sukzessiven Ergebnisse eines `foldl` bzw. eines `foldr` über einer Liste ebenfalls wieder in einer Liste benötigt, dann sind `scanl`, `scanr` die richtigen Funktionen.

```
scanl (+) 0 [1,2,3,4] --> [0,1,3,6,10]
```

Die naive Definition ist:

```
inits []      = [[]]
inits (h:t) = [] : (map (h:) (inits t))
```

```
scanl f e     = map (foldl f e) . inits
```

Diese Definition ist nicht effizient (quadratisch), da für jede Anfangsliste der Wert jedesmal neu berechnet wird.

Eine lineare Version ist:

```
scanl f e []      = [e]
scanl f e (x:xs) = e : (scanl f (f e x) xs)
```

Eine Beispielauswertung (nicht in Normalordnung, sondern zur Normalform) ergibt:

```
      scanl (*) 1 [1,2,3]
--> 1: (scanl (*) (1 * 1) [2,3]
--> 1: 1 : scanl (*) (1 * 2) [3]
--> 1: 1 : 2 : scanl (*) (2*3) []
--> 1: 1 : 2 : [6]
=      [1,1,2,6]
```

Beispiel 2.4.1 Will man die Teilsummen zur Berechnung der Eulerschen Zahl e in einer Liste haben, dann kann man das programmieren als:

```
scanl (+) 0.0 (map (1.0 /) (scanl (*) 1.0 [1.0 ..]))
```

Analog kann man die sukzessiven Ergebnisse eines foldr über die Tails einer Liste ausrechnen:

```
tails []      = [[]]
tails (x:xs) = (x:xs) : tails xs
```

```
scanr f e     = (map (foldr f e)) . tails
```

Dies ist wieder eine quadratische Version, die man verbessern kann zu:

```
scanr f e []      = [e]
scanr f e (x:xs) = (f x (hd ys)) : ys
                  where ys = scanr f e xs
```

Beispielauswertung (nicht NO)

```

    scanr (*) 1 [1,2,3]
--> (1 * (hd ys)):ys   where ys = scanr * 1 [2,3]
--> (1 * (hd ys)):ys   where ys = 2 * (hd ys'):ys'
                        where ys' = scanr * 1 [3]
--> (1 * (hd ys)):ys   where ys = 2 * (hd ys'):ys'
                        where ys' = 3 * (hd ys''):y''
                        where ys'' = scanr * 1 []
--> (1 * (hd ys)):ys   where ys = 2 * (hd ys'):ys'
                        where ys' = 3 * (hd ys'') :y''
                        where ys'' = [1]
--> (1 * (hd ys)):ys   where ys = 2 * (hd ys'):ys'
                        where ys' = [3,1]
--> (1 * (hd ys)):ys   where ys = [6,3,1]
--> [6,6,3,1]

```

2.5 Ströme als unendliche Listen in Haskell

Ein Strom ist eine Folge oder Liste von Daten, die man in Haskell als Liste bzw. auch als potentiell unendliche Liste darstellen kann. Die Modellvorstellung zur Verarbeitung von Strömen ist, dass man einen Eingabe-Strom nur einmal einlesen kann, und nur einen kleinen Teil davon zur Verarbeitung bzw. zur Erzeugung eines Ausgabestroms verwenden bzw. speichern kann. Diese Verarbeitung passt zu Kanälen, die z.B. einen Strom von Zeichen verarbeiten. Das Lesen eines sequentiellen Files kann man ebenso als Verarbeitung eines Stroms sehen, denn Strings sind Listen von Zeichen. Auch die lexikalische und syntaktische Analysephase eines Compilers ist eine Strom-Verarbeitung.

Die meisten Haskell-Listenfunktionen sind für Ströme geeignet, und können auch leicht zu weiteren strom-verarbeitenden Funktionen zusammgebaut werden, da Haskell verzögert auswertet. Geeignet sind z.B.: `map`, `filter`, `nub`, `zip`, `scanr`, `scanl`, `concat`, `foldr`, `words`, `unwords`, `lines`, `unlines`, aber nicht `length`, `reverse`, `foldl`.

Zur Verarbeitung von mehreren Strömen sind zum Beispiel geeignet: `zip`, `zipWith`, `merge` aber z.B. nicht die Sortierfunktionen, da diese den ganzen Strom kennen müssen.

2.5.1 Ströme, unendliche Listen

In Haskell ist es aufgrund der verzögerten Auswertung leicht möglich, unendliche Listen zu verarbeiten. Diese kann man auch als Strom von Werten ansehen.

Man kann in Haskell leicht mittels unendlichen Listen ausreichend Beispiele und Beispielverarbeitungen für Ströme veranschaulichen. Zum Beispiel ergibt [1..] die Liste aller natürlichen Zahlen ab 1, die man als Strom aller natürlichen Zahlen interpretieren kann.

Verarbeitet man immer das aktuelle Objekt und hat keine Referenz auf den Anfang des Stroms, dann wird die Endrekursions-Optimierung zusammen mit dem Garbage-Collector dafür sorgen, dass man bei vielen interessanten Verarbeitungen mit konstantem Speicher den Strom bzw. die Eingabeströme verarbeiten kann.

```
Beispiel 2.5.1 Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24, ...]
```

```
Prelude> map quadrat [1..]
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,...]
Hier wird jeweils ein Element aus dem Strom verarbeitet und gedruckt, im zweiten Beispiel wird es vorher quadriert.
```

Beim Programmieren ist Vorsicht geboten, da man leicht nichtterminierende Anfragen stellen kann:

```
Prelude> length [1..]
^CInterrupted.
```

Beispiel 2.5.2 Will man wissen, ob ein Strom `xs` `n` oder mehr Elemente enthält, dann sollte man das mittels `drop n xs /= []` programmieren

```
length_ge str n = drop n xs /= []
```

```
*Main> length_ge [1..] 2
True
*Main> length_ge [1..10] 20
False
```

Bei der rekursiven Programmierung der Verarbeitung von Strömen ist der Entwurf bzw. Nachweis der Terminierung mittels vollständiger Induktion so nicht mehr möglich, da es sich um unendliche Listen handelt. Man kann annäherungsweise annehmen, dass die Listen sehr groß aber nicht unendlich sind, und dann die übliche Methode verwenden. Im Kapitel zur kontextuellen Semantik wird eine allgemeine Methode besprochen.

Weitere vordefinierte Funktionen, die für Ströme verwendbar sind: `filter`, `nub`, `scanr`, `scanl`, `concat`, `foldr`, `zip`, `zipWith`. Eine Beispielverarbeitung ist:

```
*Main> filter (\x -> x > 100 && x < 1000) [1..]
[101,102,103,104,105,106,107,108,109, .....
992,993,994,995,996,997,998,999^CInterrupted.
```

Wenn `filter` einen endlichen aus einem unendlichen Strom herausfiltert, dann wird am Ende das Programm steckenbleiben. Besser ist dann z.B. `takeWhile` oder `dropWhile`, wenn man die Bedingung nur am Anfang testen will.

```
*Main> takeWhile (\x -> x < 1000) [1..]
[1,2,3,4,5,6,7, ...
996,997,998,999]
```

„Zufällige“ Ströme bzw. Listen (alter Haskell-Standard) erhält man mit

```
randomInts 1 2
[2147442192,436925867,1434534200,990707786,1573909306, ...
```

Beispiel 2.5.3 Man kann leicht Funktionen schreiben, die zB nur jedes k -te Element eines Stroms auswählen:

```
strom_ktes xs k = let (y:ys) = drop k xs    in y : ( strom_ktes ys k)

strom_ktes [1..] 10
[11,22,33,44,55,66,77,88,99,110,121,132,143 ..
```

Beispiel 2.5.4 Die Funktion `zipWith` wendet eine Funktion auf die jeweiligen ersten Elemente der beiden Ströme an:

```
zipWith (+) [1..] [2,4..]
[3,6,9,12,15,18,21,24,27,30,33,36,39,...
```

Man kann auch Verarbeitungen definieren, die nicht synchron sind:

```
zipWith (\ x y -> take x (repeat y)) [1..] ['a'..]
["a","bb","ccc","dddd","eeee","ffffff","ggggggg","hhhhhhh"],
```

Eine zufällige Vermehrung der Elemente einer Liste ergibt ebenfalls eine nicht-synchrone Verarbeitung: Hier ist die Liste der Strings wieder mit `concat` zu einer Liste verbunden:

```
str_kopiere str1  :: [Int] -> [b]  -> [b]
str_kopiere str1 str2 =
    concat (zipWith (\ x y -> take x (repeat y)) str1 str2)

str_kopiere (map (\x-> x `rem` 3) (randomInts 1 2)) ['a'..]
"bbccddeffgghijjkkppqstuu ....
```

Beispiel 2.5.5 Das Kreuzprodukt von zwei Strömen kann man folgendermaßen erhalten, wobei man bei der Programmierung eine faire Abzählung benutzen muss.

```
strKreuz [] ys = []
strKreuz xs [] = []
-- strKreuz (x:xs) (y:ys) = concat (zip (map head )
-- tails (x:xs) = (x:xs) : tails xs
stKreuz xs (y:ys) = concat (stKreuzr (tails xs) [y:ys] ys)
strKreuzr xstails ystailsr [] = [(zip (map head xstails) (map head ystailsr))]
```

```
stKreuzr xstails ystailsr (y:ysend) =
    (zip (map head xstails) (map head ystailsr))
    : (stKreuzr xstails ((y:ysend) : ystailsr) ysend)
```

```
stKreuz [1..] [1..]
[(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3),(3,2),(4,1),(1,5),(2,4),
(3,3),(4,2),(5,1),(1,6),(2,5),(3,4),(4,3),(5,2),(6,1),
(1,7),(2,6),(3,5),(4,4),(5,3),(6,2),(7,1), . . . . .]
```

Beispiel 2.5.6 *Folgende Funktion scanl berechnet das foldl jeweils der ersten n Elemente:*

```
scanl (+) 0 [1..]
[0,1,3,6,10,15,21,28,36,45,55,66,78,91, . . .]
```

Das Produkt der jeweils ersten n Zahlen ergibt Liste der Fakultäten:

```
scanl (*) 1 [1..]
[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001
600,6227020800,87178291200,1307674368000, . . . . .]
```

```
scanl (\x y-> y:x) [] (repeat 'a')
["","a","aa","aaa","aaaa","aaaaa","aaaaaa","aaaaaaa",
"aaaaaaaa","aaaaaaaaa","aaaaaaaaaaa","aaaaaaaaaaaa", . . . . .]
```

Beispiel 2.5.7 *Die Funktion scanr berechnet das foldr jeweils der Tails des Stroms. Die kann man nur verwenden, wenn der foldr-Anteil entweder nur den Anfang des jeweiligen tails benötigt, oder wieder einen Strom erzeugt. Die Summe der tails terminiert nicht:*

```
scanr (+) 0 [1..]
[^\CInterrupted.
```

Man kann sinnvolle Verarbeitungen damit programmieren

```
tails xs = (scanr (:) [] xs) --- ineffizient
map head (scanr (:) [] [1..])
```

Sortierte Ströme

Eine interessante Verarbeitung von sortierten Strömen ist das Mischen von 2 aufsteigend sortierten Strömen, das wieder einen aufsteigend sortierten Strom ergibt. Das Mischen entspricht der Vereinigung.

Beispiel 2.5.8 *Mischen von geraden und ungeraden Zahlen:*

```

mische [] ys = ys
mische xs [] = xs
mische (x:xs) (y:ys) =
    if x <= y then x: (mische xs (y:ys))
    else y: (mische (x:xs) ys)

mische [1,3..] [2,4..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,
53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,
71,72,73,74,75,76,77,78,79,...

> mische [2,4..] [3,6..]
[2,3,4,6,6,8,9,10,12,12

```

Beispiel 2.5.9 *Mischen der Vielfachen von 2,3,5:*

```

Main> mische (mische (map (2*) [1..]) (map (3*) [1..])) (map (5*) [1..])
[2,3,4,5,6,6,8,9,10,10,12,12,14,15,15,16,18,18,20,20,21,22,24,24,25,26,27,
28,30,30,30,32,33,34,35,36,36,38,39,40,40,42,42,44,45,45,46,48,48,50,50,...

```

Die Ausgabe enthält auch doppelte Elemente. Will man diese entfernen, kann man entweder mit nub diese nachträglich entfernen, oder gleich beim Mischvorgang diese überlesen. Das muss man dann selbst programmieren.

```

*Main> nub (mische (mische (map (2*) [1..]) (map (3*) [1..]))
             (map (5*) [1..]) )
[2,3,4,5,6,8,9,10,12,14,15,16,18,20,21,22,24,25,26,27,28,30,32,33,34,...

```

```

mischeNub [] ys = ys
mischeNub xs [] = xs
mischeNub (x:xs) (y:ys) =
    if x == y then (mischeNub xs (y:ys))
    else
    if x <= y then x: (mischeNub xs (y:ys))
    else y: (mischeNub (x:xs) ys)

```

```

*Main> mischeNub (mischeNub (map (2*) [1..]) (map (3*) [1..]))
             (map (5*) [1..])
[2,3,4,5,6,8,9,10,12,14,15,16,18,20,21,22,24,25,26,27,28,30,32,33,34,35,

```

Beispiel 2.5.10 *Das Programm zur Berechnung der Differenz von zwei Strömen kann man leicht definieren, wenn die Eingabeströme aufsteigend sortierte Zahlen enthalten:*

```

strom_minus xs [] = xs
strom_minus [] ys = []
strom_minus (x:xs) (y:ys) = if x == y then strom_minus xs (y:ys)
                             else if x > y then strom_minus (x:xs) ys
                             else x: (strom_minus xs (y:ys))

> strom_minus [1..]
  (nub (mische (map (2*) [1..])
            (mische (map (3*) [1..]) (map (5*) [1..]) )))
[1,7,11,13,17,19,23,29,31,37,41,43,

```

Beispiel 2.5.11 *Es fehlt noch die Funktion zur Berechnung des Schnitts von zwei Strömen, die aufsteigend sortierte Zahlen sind:*

```

-- strom_schnitt nur fuer sortierte Eingaben:
strom_schnitt xs [] = []
strom_schnitt [] ys = []
strom_schnitt (x:xs) (y:ys) = if x == y then x: strom_schnitt xs ys
                               else if x > y then strom_schnitt (x:xs) ys
                               else strom_schnitt xs (y:ys)

*Main> strom_schnitt [2,4..] [3,6..]
[6,12,18,24,30,36,42,48,54,60,66,72,78,84,...

```

```

Beispiel 2.5.12 strom_minus xs [] = xs
strom_minus [] ys = []
strom_minus (x:xs) (y:ys) = if x == y then strom_minus xs (y:ys)
                             else if x > y then strom_minus (x:xs) ys
                             else x: (strom_minus xs (y:ys))

> strom_minus [1..]
  (nub (mische (map (2*) [1..])
            (mische (map (3*) [1..]) (map (5*) [1..]) )))
[1,7,11,13,17,19,23,29,31,37,41,43,

```

Man kann diese Funktionen auch allgemeiner schreiben, wenn die Ordnungsrelation anders definiert ist, so dass diese Funktionen auch für absteigende Ströme funktionieren.

Eine Fingerübung ist die rekursive Definition aller Primzahlen als Strom.

```

primes = 2: [x | x <- [3,5..],
             and (map (\t-> x `mod` t /= 0)
                  (takeWhile (\y -> y^2 <= x) primes))]

```

Der Vergleich mit einer durch den Fermatschen Primzahltest erzeugten Liste ergibt:

```
primesDifference = strom_minus
                  (2:[x | x <- [3,5..], fermat_test_naiv x])
                  primes
```

```
*Main> primesDifference
[561,1105,1729,2465,2821,6601^C
```

Beispiel 2.5.13 *Man kann auch die Funktion foldr für Ströme verwenden, während die Funktion foldl ungeeignet ist, da sie für unendliche Ströme nicht terminiert.*

```
*Main> foldl (\y x -> x:x:y) [] [1..10]
[10,10,9,9,8,8,7,7,6,6,5,5,4,4,3,3,2,2,1,1]
*Main> foldl (\y x -> x:x:y) [] [1..]
^CInterrupted.
```

```
*Main> foldr (\x y -> x:x:y) [] [1..10]
[1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10]
foldr (\x y -> x:x:y) [] [1..]
[1,1,2,2,3,3,4,4,5,5,6,6.....
*Main>
```

Man kann sequentielle Text-Files bearbeiten mit folgenden in Haskell bereitgestellten Funktionen:

```
words :: String -> [String]
unwords :: [String] -> String
lines :: String -> [String]
unlines :: [String] -> String
```

Die verzögerte Auswertung erlaubt es, auch kombinierte Anfragen zu erstellen, die den Strom nur einmal lesen.

Leider passiert es hin und wieder, dass man durch ungeeignetes Festhalten von Elementen des Stroms zuviel Speicher blockiert. Das kann z.B. durch die feste Definition eines Stromes als Konstante im Programm passieren, wie der Strom `primes`, der im Heap erhalten bleibt bis zum Ende des Programms, wobei die Länge davon abhängt, wie weit man die Liste irgendwann angefordert hat.

Verwendung der Stromverarbeitung für endliche Mengen

Wenn man endliche Mengen von gleichartigen Objekten (Records, Datensätze, o.ä.) als aufsteigend sortierte Listen darstellen kann, dann sind die Operationen auf endlichen Mengen wie Schnitt, Vereinigung, und Differenz recht einfach und

sehr effizient durchzuführen, wobei man das Ergebnis ebenfalls als aufsteigend sortierte Liste erhält. Die Implementierung der Funktionen ist genau wie bei Strömen.

Dadurch kann man diese Operationen auch mehrfach und geschachtelt auf den entsprechenden Listen durchführen, wobei die Laufzeit linear in der Anzahl der angefassten Elemente ist.

Nimmt man an, dass die Listen auch als Array vorliegen, dann ist auch die Suche effizient, da man bei binärer Suche nur logarithmische Laufzeit hat.

Weitere Funktionalitäten:

Potenzmenge ist nicht sinnvoll bei Strömen, aber bei Mengen, und ist in jedem Fall exponentiell, da die Ausgabe exponentiell ist. Will man die Mengen geordnet ausgeben, kann man verschiedene Ordnungen wählen:

Nach Kardinalität, dann analog zur lexikographischen Ordnung, oder nur nach lexikographischer Ordnung auf Folgen.

---Potenzmenge, lexikographisch

```
potenzmenge xs = []:potenzmengeR xs
potenzmengeR [x] = [[x]]
potenzmengeR (x:xs) = let pm = (potenzmengeR xs)
                      in [x]: (map (x:) pm) ++ pm
```

---Potenzmenge, Kardinalitaet, dann lexiko

```
potenzmengeKL xs = []: (concatMap (\n -> potenzmengeKLR xs n)
                               [1.. length xs])

potenzmengeKLR [] _ = []
potenzmengeKLR xs 0 = [[]]
potenzmengeKLR xs 1 = map (\u-> [u]) xs
potenzmengeKLR (x:xs) (n+1) =
    let pmax = potenzmengeKLR xs (n+1)
        pmx = potenzmengeKLR xs n
    in (map (x:) pmx) ++ pmax
```

Kreuzprodukt bzw. die Menge aller Paare bzw. n-Tupel für gegebenes n . Die Funktion für Ströme kann hier nicht die lexikographische Ordnung einhalten, wenn die Ausgabe fair sein soll. Auf endlichen Mengen ist die Ausgabe jedoch leicht in richtiger Reihenfolge zu implementieren, wenn die Eingabe sortiert ist:

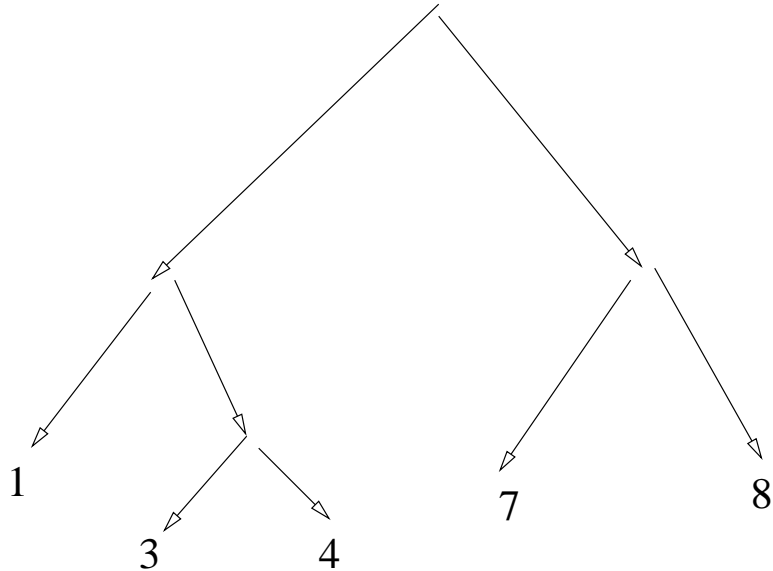
```
kreuzprodukt xs ys = [(x,y) | x <- xs, y <- ys]
```

2.5.2 Datenstruktur Baum

Definition von binären geordneten Bäumen in Haskell:

```
data Binbaum a = Bblatt a | Bknoten (Binbaum a) (Binbaum a)
```

Beispiel 2.5.14 *Der folgende binäre Baum*



hat eine Darstellung als

```
Bknoten (Bknoten (Bblatt 1)
                (Bknoten (Bblatt 3) (Bblatt 4)))
        (Bknoten (Bblatt 7) (Bblatt 8))
```

Im Folgenden geben wir einige Verarbeitungsfunktionen für diese Baumdatenstruktur an.

Berechnet den Rand des Baumes: eine Liste der Markierungen der Blätter:

```
b_rand (Bblatt x)      = [x]
b_rand (Bknoten bl br) = (b_rand bl) ++ (b_rand br)
```

Die folgende Funktion testet, ob ein gegebenes Element im Baum ist:

```
b_in x (Bblatt y)      = (x == y)
b_in x (Bknoten bl br) = b_in x bl || b_in x br
```

Die folgende Funktion entspricht einem `map` über Bäumen. Sie wendet eine Funktion auf alle Markierungen der Blätter eines Baumes an, Das Resultat ist der Baum der Resultate.

```
b_map f (Bblatt x)      = Bblatt (f x)
b_map f (Bknoten bl br) =
    Bknoten (b_map f bl) (b_map f br)
```

Berechnet die Größe des Baumes:

```
b_size (Bblatt x)      = 1
b_size (Bknoten bl br) = 1+ (b_size bl) + (b_size br)
```

Berechnet die Summe aller Blätter, falls die Markierungen der Blätter Zahlen sind:

```
b_sum (Bblatt x)      = x
b_sum (Bknoten bl br) = (b_sum bl) + (b_sum br)
```

Eine Funktion zum Berechnen der Tiefe des Baumes:

```
b_depth (Bblatt x) = 0
b_depth (Bknoten bl br) =
    1 + (max (b_depth bl) (b_depth br))
```

Ein schnelles fold über binäre Bäume kann man folgendermaßen definieren:

```
foldbt :: (a -> b -> b) -> b -> Binbaum a -> b
foldbt op a (Bblatt x)      = op x a
foldbt op a (Bknoten x y) = (foldbt op (foldbt op a y) x)
```

Eine Variante von foldbt mit optimiertem Stackverbrauch:

```
foldbt' :: (a -> b -> b) -> b -> Binbaum a -> b
foldbt' op a (Bblatt x)      = op x a
foldbt' op a (Bknoten x y) =
    (((foldbt' op) $! (foldbt' op a y)) x)      -- $! = strict
```

Damit kann man effizientere Versionen von b_rand und b_sum definieren:

```
b_rand_eff = foldbt (:) []
b_baum_sum' = foldbt' (+) 0
```

Beispiel 2.5.15 Eine Beispiel-Auswertung mit foldbt

Werte foldbt (+) 0 "(((1,2),3),(4 ,5))" aus:

```
foldbt (+) 0 "(((1,2),3),(4 ,5))"
--> foldbt (+) (foldbt (+) 0 (4 ,5)) ((1,2),3)
--> foldbt (+) (foldbt (+) (foldbt (+) 0 (5)) (4) ((1,2),3))
--> foldbt (+) (foldbt (+) (5+0) (4) ((1,2),3))
--> foldbt (+) (4+ (5+0)) ((1,2),3)
--> foldbt (+) (foldbt (+) (4+ (5+0)) (3)) (1,2)
--> foldbt (+) (3+ (4+ (5+0))) (1,2)
--> foldbt (+) (foldbt (+) (3+ (4+ (5+0))) (2) (1))
--> foldbt (+) (2+ (3+ (4+ (5+0)))) (1)
--> 1+ (2+ (3+ (4+ (5+0))))
```

Allgemeinere Bäume mit beliebiger Anzahl Töchter kann man ebenfalls definieren: Die Programmierung der Verarbeitungsfunktionen ist analog zu binären Bäumen

```

data Nbaum a = Nblatt a | Nknoten [Nbaum a]

nbaumrand :: Nbaum a -> [a]
nbaumrand  (Nblatt x) = [x]
nbaumrand  (Nknoten xs) = concatMap nbaumrand xs

```

2.6 Haskell und Hugs

Haskell (Haskell. B. Curry) ist eine lazy auswertende funktionale Sprache, die einen Standardisierungsversuch darstellt, an dem u.a. beteiligt waren und z.T. noch sind: Paul Hudak, Philip Wadler, Simon Peyton Jones, John Hughes, ... Zusätzlich zu lazy evaluation, statischem strengem Typsystem, polymorphen Funktionen höherer Ordnung, algebraischen Datentypen, ZF-Ausdrücken (List-Comprehensions) sind folgende Konzepte verwirklicht:

- Typklassensystem
- Behandlung von Überladung (overloading)
- monadisches Programmieren, monadisches I-O

Es gibt verschiedene Implementierungen (Compiler) von Haskell:

- GHC: Glasgow Haskell-Compiler: weitgehend in Haskell selbst geschrieben. GHC kompiliert nach C.
- Haskell-B-Compiler (hbc) von Chalmers (wird nicht mehr weiterentwickelt).
- nhc bzw. nhc98 ; siehe auch <http://www.cs.york.ac.uk/fp/nhc98/>.
- Eine interaktive Variante ist ghci.
- Eine andere interaktive Variante ist HUGS (Haskell Users Gofer System)
- Ein Vorläufer von Hugs ist Gofer, implementiert von Mark P. Jones.

In diesem Kapitel werden einige syntaktischen Konstrukte und das Typklassensystem von Haskell beschrieben. Die Semantik wird durch Transformation nach KFPT bzw. im Endeffekt nach KFP angegeben, wobei die Semantik komplizierter Konstrukte durch Angabe einer Transformation in einfacheres Haskell angegeben wird. Die existierenden Objekte, Daten wie Funktionen kann man sich als KFP-Objekte vorstellen. Der vorgegebene Weg der Transformation in eine Kernsprache wird auch vom Compiler genommen, der sukzessive die höchsprachlichen Konstrukte in eine Kernsprache übersetzt und sie erst danach in eine weitere Zwischensprache für eine abstrakte Maschine transformiert.

2.6.1 Syntax

Haskell (mit Hinweisen auf die Vorläufersprache Miranda) wird in einem kurzen, nicht vollständigen Überblick dargestellt. Für genauere Informationen ist das entsprechende (Online-) Handbuch bzw. Haskellreport zu konsultieren.

Zahlen, Character sind vorhanden, wobei es verschiedene Zahlentypen gibt: beschränkte ganze Zahlen (4 byte), unbeschränkte ganze Zahlen, Gleitkommazahlen, und auch rationale Zahlen.

Listen werden abgekürzt als [1, 2, 3]. Dies steht für $1 : 2 : 3 : \text{Nil}$, dies wiederum für $1 : (2 : (3 : \text{Nil}))$. ":" ist der Infix-Konstruktor **Cons**.

Es gibt als vordefinierte Typen: Character 'a', 'A', Strings sind Listen von Charactern; **String** = [Char]

Prioritäten Die Operatoren haben eine Priorität, damit man Klammern weglassen kann. Man kann Operatoren als Präfix-, Infix-, Postfix-Operatoren benutzen, je nach Ihrer Definition.

Man kann die Infix-Eigenschaft einer Funktion aufheben, z.B. wenn man diese als Argument übergeben will, indem man diese einklammert: ((+) 1 2) ist äquivalent zu (1 + 2). Analog kann man zweistellige Funktionen infix benutzen, wenn man diese in Anführungszeichen setzt: 'primPlusInt'.

Gruppierung von 2-stelligen Funktionen kann linksassoziativ oder rechtsassoziativ oder weder/noch sein: $1 - 2 - 3 \equiv (1 - 2) - 3$ aber $1 : 2 : 3 : [] \equiv 1 : (2 : (3 : \text{Nil}))$

z.B. $\text{fac } 2 + \text{fac } 3 \equiv (\text{fac } 2) + (\text{fac } 3)$, wenn **fac** höhere Priorität als + hat. Beispiele der Deklarationen der Operatoren:

```
infixl 7 *
infixl 6 +, -, /, 'div'
infixr 5 ++
infix 4 ==, /= < <=, >=, >
infixr 3 && (logisches und)
infixr 2 || (logisches oder)
```

Dies würde bei $1 < 2 < 3$ einen Syntaxfehler ergeben, da < nicht als gruppierbar definiert ist.

Miranda hat einen Typ *Num*, der reelle und ganze Zahlen umfasst mit automatischer Umwandlung (coerce) von ganzen in reelle und umgekehrt. In Haskell muss die Umwandlung vom Programmierer angegeben werden, wobei ein Typklassensystem zur Unterstützung dient.

Überladung: Dasselbe Symbol wird für verschiedene Operationen benutzt, z.B. + für reelle bzw integer-Addition. Bei Haskell wird die Überladung vor der Ausführung weitgehend aufgelöst, es gibt keine Markierungen an Objekten zur Laufzeit, die integer und reelle Zahlen unterscheiden. Bei Miranda gibt es diese internen Markierungen.

Man kann $(2 +)$ als Funktion benutzen, die 2 addiert, ebenso wie $(+ 2)$.
 $(/ 2)$ ist die Funktion, die halbiert
 $(2 /)$ die Funktion, die $2 / x$ berechnet.
 Eine Ausnahme ist $(-x)$, die die Zahl negiert statt die Funktion zu erzeugen, die x subtrahiert.

Layout-Regeln werden vom Haskell-Parser beachtet, insbesondere bei Funktionsdefinitionen. Bei Einrückung werden Klammern “{“ eingefügt, bei Ausrückung entsprechend viele Klammern geschlossen. Es gibt Sonderregelungen für **where** und **let**. Diese können manchmal zu verwirrenden Fehlermeldungen führen, die man durch richtige optische Anordnung des Quellcodes beheben kann.

Die Unterscheidung in einer Funktionsdefinition nach den Argumenten können durch sogenannte Wächter (guards) an den Fällen noch in der Handhabung erleichtert werden:

product $[1..n]$ berechnet somit $(n!)$ Sie berechnet das Produkt aller Zahlen in einer Liste.

Fakultät mit guards:

```
fac 0          = 1
fac n | n > 0 = n * fac (n - 1)
```

$n > 0$ ist hier ein Wächter (guard).

show

Es gibt eine eingebaute Funktion **show**, die alle Ausdrücke (der Typklasse **Show**) auf Strings abbildet. d.h. $show : a \rightarrow String$.

Geschachtelte Pattern

Haskell erlaubt kompliziertere Patterns, z.B. $f [x,y] = x+y$ ist möglich. Eine Beispielfunktion ist das Löschen von gleichen, direkt aufeinanderfolgenden Elementen in einer Liste:

```
remdup []          = []
remdup [x]         = [x]
remdup (x:(y:z)) | x == y = remdup (y:z)
                  | otherwise = x : (remdup (y:z))
```

Ein Pattern ist ein Ausdruck, der nur aus Konstruktoren, (auch Konstruktor-konstanten, auch, 1,2,3,...) und Variablen besteht. Jede Variable darf dabei nur einmal vorkommen. Syntaktisch darf für eine anonyme Variable auch **_** stehen. Dies zählt jedesmal als neue Variable, die sonst nirgendwo vorkommt. (x, x) oder $x : x : z$ sind als Pattern verboten.

Patternsyntaxerweiterung:

Listenpattern: $[x, y, z]$ ist eine Liste mit drei Elementen, die für $x : y : z : []$ steht .

markierte Pattern: $p@(x, y)$. Der Name p referenziert das ganze Paar, x, y jeweils die erste und zweite Komponente

$m + 1$ -Pattern ($n + 1$) matcht positive Werte, n ist bereits eins weniger. Dies entspricht einem Pattern `succ(n)`.

Undefinierte Werte

Es gibt keinen eigenen undefinierten Wert, der z.B. bei `1/0` als Wert zurückgegeben wird. Implementierungsabhängig gibt dies i.a. eine Meldung an den Benutzer zur Laufzeit. Eine andere Situation, in der ein Wert undefiniert ist, wenn in einer Definition kein Pattern zutrifft. z.B. `hd []`, wenn `hd (x : _) = x`
 In der Semantik der Sprache werden solche Situationen als äquivalent zu Nichtterminierung gedeutet. In der Semantik wird das undefinierte Objekt mit \perp bezeichnet. Alle Objekte, die keine WHNF haben, sind dann semantisch $= \perp$, ebenso, Ausdrücke, wie `1/0` oder `(hd Nil)`.

2.6.2 Semantik

Die Zurückführung auf die Basissprache KFPT bzw KFP ist bei allen bisher eingeführten Funktionen und Operatoren unproblematisch.

- Die Benutzung von Schlüsselwörtern, spezielle Gruppierung, Infix o.ä sind als Eigenschaften der Syntax zu betrachten, die ein Präprozessor die Eingabe in eine eindeutige Sprache der Ausdrücke überführt.
- Datentypen wie Gleitkommazahlen, Integer, Character usw. kann man implementieren als algebraischen Datentyp mit Konstanten, bzw. als Peanozahlen.
- Die Transformation von Funktionen wie `show` verschieben wir auf den Abschnitt in dem Typklassen behandelt werden.
- Tupel, Listen, if-then-else sind leicht als algebraische Datentypen definierbar.
- Etwas genauer muss man sich die Übersetzung der Funktionsdefinitionen anschauen, die als mehrfache Gleichungen aufgeschrieben werden. Haskell verarbeitet Patterns von oben nach unten. Deshalb wird die Übersetzung ein sequentielles Matchen sein. Innerhalb einer Patternliste von links nach rechts. Bei nichtpassenden Werten wird allerdings das Objekt nicht für das ganze Pattern ausgewertet. In Spezialfällen kann man dies als ein einzelnes `case` übersetzen: Wenn es keine überlappenden Pattern gibt, wenn

die Argumentpattern bestimmte Struktur haben,... Solange nur einfache Pattern, und keine guards verwendet werden, gibt es eine einfache Übersetzung. Diese Definitionen sind leicht als geschachteltes `case` zu übersetzen, allerdings ist nicht immer offensichtlich, welches Argument zuerst `case` zugeführt werden soll. Diese Reihenfolge beeinflusst die Terminierung (damit die Semantik) der Funktionen. z.B. eine Funktion `f`, die mit

```
f x [] = ...
f x (y:ys) = ...
```

definiert wird, sollte nur ein `case` über das 2.te Argument machen, aber nicht über das erste.

- Guards sind ebenfalls leicht zu transformieren. Zu beachten ist, dass bei einem Fehlschlagen eines guards das nächste pattern genommen wird.
- Kompliziertere Pattern sind leicht als verschachtelte `case` zu übersetzen, allerdings muss bei Definitionen mit mehreren Gleichungen, wobei mehrere Argumente als komplexe Pattern auftauchen, festgelegt sein, welche Strategie beim Kompilieren verwendet wird. In Hugs ist dies sequentiell, in anderen Programmiersprachen wurde auch schon mit anderen Strategien gearbeitet. Wenn wir davon ausgehen, dass bereits vom Compiler die Reihenfolge der Patternverarbeitung festgelegt wird, dann haben wir keine Probleme beim Übersetzen. Es gibt verschiedene Möglichkeiten: man könnte von links nach rechts abarbeiten, dann jeweils in den Pattern von oben nach unten, oder die Gleichungen in der Reihenfolge des Hinschreibens; oder sogar eine kompliziertere Strategie verfolgen. Wesentlich ist auf jeden Fall, dass der Programmierer weiß, welche Strategie verwendet wird, da sonst die Effekte des Compilierens nicht durchschaubar sind bzw. die Semantik nicht exakt definiert ist.
- Rekursive Definitionen mittels `let` oder durch rekursive Superkombinatoren sind nicht durch einfache syntaktische Umformungen nach KFPT bzw KFP transformierbar. Hierzu kann man sogenannte Fixpunktkombinatoren verwenden, z.B.

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

(Siehe ein späteres Kapitel)

Ein nichtrekursives `let` ist durch eine Anwendung und eine Abstraktion übersetzbar:

```
let x = s in t ≡ ((\x.t) s).
```

- Eine erzwungene Auswertung mittels der Funktion `strict`: (`strict f`) `x` wertet zunächst `x` bis WHNF aus, dann erst `(f x)`. Dies ist leicht nach KFP zu übersetzen.

Durch diese Transformationsregeln werden folgende Eigenschaften von KFPT auf Haskell übertragen:

1. Die Semantik der Funktionen und Konstrukte in Haskell wird festgelegt.
2. Die Gültigkeit der kontextuellen Gleichheit (siehe späteres Kapitel) und die Korrektheit von Programmtransformationen.

List Comprehensions

List Comprehensions werden auch ZF-Expressions genannt (Zermelo & Fränkel). Dies sind einfachere syntaktischere Schreibweisen für Listenkonstruktionen, die an die Mengenschreibweise wie $\{x * x \mid x < -[1..100], x > 5\}$ angelehnt sind. Z.B. kann man `map` und `filter` damit folgendermaßen schreiben:

$$[f\ x \mid x \leftarrow l] \equiv \text{map } f\ l$$

$$[x \mid x \leftarrow l, p\ x] \equiv \text{filter } p\ l$$

Die Schreibweise $\{exp \mid \dots\}$ soll eine Liste andeuten, wobei exp ein Ausdruck für die Elemente der Liste ist und rechts Generatoren (\leftarrow Ausdrücke) und Guards wie $p\ x$ stehen können.

Bei komplizierteren Ausdrücken ist der Geltungsbereich der Variablen zu beachten: Die Reihenfolge in der die Elemente in der neuen Liste eingereiht werden, ist durch die Semantik des ZF-Ausdrucks festgelegt.

$$[(x, y) \mid x \leftarrow [1..6], y \leftarrow [1..6], x+y < 5] \rightarrow [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (3, 1)]$$

Neue Variablen werden in der linken Seite von Generatoren eröffnet, der Geltungsbereich liegt in allen Ausdrücken rechts des Generatorausdrucks und in dem Kopfausdruck. D.h. x aus $x \leftarrow [1..6]$ hat den Geltungsbereich: $(x, y), x+y < 10$. Die Scopusregeln gelten wie üblich. Wenn man Variablen mehrfach einführt, erzeugt man jeweils neue Geltungsbereichs.

Allgemeiner kann in den Generatoren links auch ein Pattern mit neuen Variablen stehen.

Beispiel 2.6.1

```
nblanks n      = [' ' | x <- [1..n]]
kartprod xs ys = [(x,y) | x <- xs, y <-ys]

kartprod [1,2] ['a', 'b'] ergibt [(1,'a'), (1,'b'), (2,'a'), (2,'b')].
```

Wir geben eine Übersetzung von ZF-Ausdrücken in ZF-freies Haskell an, wobei wir annehmen, dass alle Pattern in Generatoren nur Variablen sind. Die Erweiterung auf Pattern, wie z.B. $(h : t)$, kann dann als Übungsaufgabe gemacht werden:

Wir benötigen 2 Regeln, die rekursiv angewendet werden können.

```
ZFgen    [e|x <- xs,Q] = concat (map (\x -> [e|Q]) xs)
ZFguard  [e|p, Q]     = if p then [e|Q] else []
ZFnil    [e|]         = [e]
```

Einfache Fälle werden optimiert:

`[e|x <- xs]` wird normalerweise in `concat (map (\x -> [e]) xs)` übersetzt. Da das `concat` aber nichts weiter leistet, als die Listenklammern um alle `[e]` wieder zu entfernen, ist das äquivalent zu `(map (\x -> e) xs)`. D.h. bei einfachen List Comprehensions wendet man die Regeln für `map` und `filter` an.

```
[e|x <- xs]    = map (\x -> e) xs
[e|x <- xs,p]  = filter p xs
```

Beispiel 2.6.2 Wir transformieren die Definition von `kartprod` in eine ZF-freie Darstellung: `[(x,y) | x <- xs, y <- ys]`.

```
[(x,y) | x <- xs, y <- ys]
≡ concat (map (\x -> [(x,y)|y <- ys]) xs)
≡ (concat (map (\x -> (map (\y -> (x,y)) ys)) xs))
```

Wie man sich durch Eingeben in `ghci` überzeugen kann, ergeben

```
(concat (map (\x -> (map (\y -> (x,y)) [4,5,6])) [1,2,3]))
```

und

```
[(x,y) | x<-[1,2,3], y<-[4,5,6]]
```

dieselbe Liste

```
[(1,4), (1,5), (1,6), (2,4), (2,5), (2,6), (3,4), (3,5), (3,6)]
```

Die Übersetzung terminiert und kann auch als Grundlage für den Typcheck verwendet werden. In Haskell gibt es vermutlich eigene Typregeln für list comprehensions.

In vielen Fällen ist diese Übersetzung nicht optimal. Z.B. wenn in dem Ausdruck `[(x,y) | x <- xs, y <- ys]` die Variable `x` nicht in `ys` vorkommt, dann kann man dies effizienter übersetzen als:

```
kp xs ys = kpr xs ys ys
```

```
kpr []      ys      zs = []
kpr (x:xs)  []      zs = kpr xs zs zs
kpr xs@(x:_)(y:ys) zs = (x,y):(kpr xs ys zs)
```

In Haskell ist es in ZF-Ausdrücken möglich mit Pattern zu arbeiten, die wie ein Filter wirken, z.B. `[x | (x:h) <- [], [1]]` ergibt `[1]`.

Die Übersetzung für den einfachsten Fall ist:

```
ZF1* [exp | pat <- lst]:
```

```

filterPat lst
  where filterPat []      = [ ]
        filterPat (pat:xs) = exp:filterPat xs
        filterPat (_:xs)  = filterPat xs

```

Eine allgemeine Übersetzung kann man so formulieren

```

[e|pat <- xs,Q] =
concat (filterPat xs)
  where filterPat []      = [ ]
        filterPat (pat:xs) = [e|Q]:filterPat xs
        filterPat (_:xs)  = filterPat xs

```

Beispiel 2.6.3 `zfex1 = [x*x | (x:_) <- [[], [1], [2], [3], []], x >= 2]`

```

zfex2 = concat (filterPat [ [], [1], [2], [3], [] ])
  where filterPat []      = [ ]
        filterPat ((x:_) : xs) = [x*x|x >= 2]:filterPat xs
        filterPat (_:xs)  = filterPat xs

```

```

*Main> zfex1
[4,9]
*Main> zfex2
[4,9]

```

2.6.3 Typregeln für polymorphe Typen

Die wichtigste Typregel in Haskell, die auch in anderen Programmiersprachen mit monomorphem Typsystem gilt, betrifft die Anwendung von Funktionen auf Argumente: Wenn σ, τ Typen sind, dann gilt die Regel:

$$\frac{s :: \sigma \rightarrow \tau, \quad t :: \sigma}{(s \ t) :: \tau}$$

Zum Beispiel ist die Anwendung der Funktion `quadrat :: Int → Int` auf eine Zahl `2 :: Int`. D.h. `quadrat 2 :: Int`.

Wenn man z.B. die Funktion `+` anwendet, mit dem Typ `+ :: Int → Int → Int`, dann erhält man für `(1 + 2)` die voll geklammerte Version `((+ 1) 2)`. `(+ 1)` hat dann den Typ `(Int → Int)`, und `((+ 1) 2)` den Typ `Int`. Man kann auch die Regel im Fall mehrerer Argumente etwas einfacher handhaben, indem man sofort alle Argumententypen einsetzt. Die zugehörige Regel ist dann

$$\frac{s :: \sigma_1 \rightarrow \sigma_2 \dots \rightarrow \sigma_n \rightarrow \tau, \quad t_1 :: \sigma_1, \dots, t_n :: \sigma_n}{(s \ t_1 \ \dots \ t_n) :: \tau}$$

Zum Beispiel ergibt das für `(+ 1 2)`:

$$\frac{+ :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \quad 1 :: \text{Int}, \quad 2 :: \text{Int}}{(+ \ 1 \ 2) :: \text{Int}}$$

Dies lässt sich noch nicht so richtig auf die Funktion `length` oder `map` anwenden. Da die Haskelltypen aber Typvariablen enthalten können, formulieren wir die erste Regel etwas allgemeiner, benötigen dazu aber den Begriff der Instanz eines Typs.

Definition 2.6.4 Wenn γ eine Funktion auf Typen ist, die Typen für Typvariablen einsetzt, dann ist γ eine Typsubstitution. Wenn τ ein Typ ist, dann nennt man $\gamma(\tau)$ eine Instanz von τ .

Beispiel 2.6.5 Man kann mit $\gamma = \{a \mapsto \text{Char}, b \mapsto \text{Float}\}$ die Instanz $\gamma([a] \rightarrow \text{Int}) = [\text{Char}] \rightarrow \text{Int}$ bilden.

Die Regel für die Anwendung lautet dann:

$$\frac{s : \sigma \rightarrow \tau, \quad t : \rho \quad \gamma(\sigma) = \gamma(\rho)}{(s \ t) :: \gamma(\tau)} \quad \text{wobei die Typvariablen } \rho \text{ umbenannt sind}$$

Die Umbenennung soll so sein, dass die Mengen der Typvariablen von σ → τ und ρ disjunkt sind, damit keine ungewollten Konflikte entstehen.

Beispiel 2.6.6 Die Anwendung für den Ausdruck `map quadrat` ergibt folgendes. Zunächst ist

$$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

Instanziiert man das mit der Typsubstitution $\{a \mapsto \text{Int}, b \mapsto \text{Int}\}$, dann erhält man, dass `map` u.a. den Typ $(\text{Int} \rightarrow \text{Int}) \rightarrow [\text{Int}] \rightarrow [\text{Int}]$ hat. Damit kann man dann die Regel verwenden:

$$\frac{\text{map} : (\text{Int} \rightarrow \text{Int}) \rightarrow [\text{Int}] \rightarrow [\text{Int}], \quad \text{quadrat} :: (\text{Int} \rightarrow \text{Int})}{(\text{map quadrat}) :: [\text{Int}] \rightarrow [\text{Int}]}$$

Die Erweiterung auf eine Funktion mit n Argumenten, wenn γ eine Typsubstitution ist, sieht so aus:

$$\frac{s :: \sigma_1 \rightarrow \sigma_2 \dots \rightarrow \sigma_n \rightarrow \tau, \quad t_1 : \rho_1, \dots, t_n : \rho_n \quad \text{und } \forall i : \gamma(\sigma_i) = \gamma(\rho_i)}{(s \ t_1 \dots t_n) :: \gamma(\tau)}$$

Auch hierbei müssen die Mengen der Typvariablen von $\sigma_1 \rightarrow \sigma_2 \dots \rightarrow \sigma_n \rightarrow \tau$ einerseits und ρ_1, \dots, ρ_n andererseits, disjunkt sein.

Bei diesen Regeln ist zu beachten, dass man das allgemeinste γ nehmen muss, um den Typ des Ergebnisses zu ermitteln. Nimmt man irgendeine andere, passende Typsubstitution, dann erhält man i.a. einen zu speziellen Typ.

Beispiel 2.6.7 Betrachte die Funktion `id` mit der Definition `id x = x` und dem Typ $a \rightarrow a$. Um Konflikte zu vermeiden, nehmen wir hier $a' \rightarrow a'$. Der Typ von `(map id)` kann berechnet werden, wenn man obige Regel mit der richtigen Typsubstitution benutzt. Der Typ des ersten Arguments von `map` muss $a \rightarrow b$

sein, und `id` erzwingt, dass $a = b$. Die passende Typsubstitution ist $\gamma = \{b \mapsto a, a' \mapsto a\}$. Das ergibt unter Anwendung der Regel

$$\frac{\text{map} : (a \rightarrow b) \rightarrow ([a] \rightarrow [b]), \quad \text{id} :: a' \rightarrow a'}{(\text{map id}) :: \gamma([a] \rightarrow [b])}$$

d.h. $(\text{map id}) :: ([a] \rightarrow [a])$.

Berechnen der Typsubstitution

Im folgenden zeigen wir, wie man die Typsubstitution γ nicht nur geschickt rät, sondern ausrechnet, wenn man Typen $\delta_i, \rho_i, i = 1, \dots, n$ gegeben hat, die nach der Einsetzung gleich sein müssen. Diese Berechnung nennt man auch *Unifikation*.

γ muss so gewählt werden, dass folgendes gilt $\forall i : \gamma(\delta_i) = \gamma(\rho_i)$. Man kann dieses Gleichungssystem umformen bzw. zerlegen. Zur besseren Lesbarkeit notieren wir $\gamma(\sigma) = \gamma(\tau)$ als $\sigma =_\gamma \tau$.

Sei G eine Multimenge von Gleichungen, TC ein Typkonstruktor, a eine Typvariable, σ, τ (polymorphe) Typen.

Wir dürfen folgende Regeln zur Umformung des Gleichungssystems $\delta_i =_\gamma \rho_i, i = 1, \dots, n$ benutzen:

$$(Dekomposition) \quad \frac{(TC \sigma_1 \dots \sigma_m) =_\gamma (TC \tau_1 \dots \tau_m), G}{\sigma_1 =_\gamma \tau_1, \dots, \sigma_m =_\gamma \tau_m, G}$$

Wenn die Typkonstruktoren rechts und links verschieden sind, dann kann man die Berechnung abbrechen; es kann keine Lösung geben.

$$(Ersetzung) \quad \frac{a =_\gamma \sigma, G}{a =_\gamma \sigma, G[\sigma/a]}$$

wobei $G[\sigma/a]$ bedeutet: Ersetze alle Vorkommen der Typvariablen a durch den Typ σ . Hierbei ist wegen der Terminierung der Berechnung darauf zu achten, dass σ die Typvariable a nicht enthält, da man sonst bei der Berechnung keinen Fortschritt erzielt, bzw. die Berechnung nicht terminiert. Man kann auch zeigen, dass es in diesem Fall keine Lösung gibt.

$$(Vereinfachung) \quad \frac{a =_\gamma a, G}{G}$$

$$(Vertauschung) \quad \frac{\sigma =_\gamma \tau, G}{\tau =_\gamma \sigma, G}$$

Wir sind fertig, wenn das Gleichungssystem die Form $a_1 =_\gamma \tau_1, \dots, a_k =_\gamma \tau_k$ hat, wobei a_i Typvariablen sind, und die a_i in keiner rechten Seite τ_j einer Gleichung auftreten.

Die Typsubstitution ist dann ablesbar als

$$\gamma = \{a_1 \mapsto \tau_1, \dots, a_k \mapsto \tau_k\}$$

Die Korrektheit des Verfahrens wollen wir hier nicht zeigen, aber es ist offensichtlich, dass die so berechnete Substitution das letzte Gleichungssystem erfüllt. Es lautet nach der Substitution: $\tau_1 = \tau_1, \dots, \tau_k = \tau_k$. Jetzt muss man für jede der Regeln zeigen, dass sie diese Korrektheit erhält.

Beispiel 2.6.8 Bei der Typisierung für `(map id)` zeigt sich, dass man je nach Regelanwendung unterschiedliche Typsubstitutionen finden kann:
Die eigentliche Typisierung ist:

$$\frac{\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b], \text{id} :: a' \rightarrow a'}{(\text{map id}) :: \gamma([a] \rightarrow [b])}$$

Berechnen von γ :

$$\frac{a \rightarrow b =_{\gamma} a' \rightarrow a'}{a =_{\gamma} a', b =_{\gamma} a'}$$

Das ergibt die Typsubstitution $\gamma = \{a \mapsto a', b \mapsto a'\}$

Eine andere Berechnung von γ mit vertauschten Regelanwendungen ergibt:

$$\frac{\frac{a \rightarrow b =_{\gamma} a' \rightarrow a'}{a =_{\gamma} a', b =_{\gamma} a'}}{a' =_{\gamma} a, b =_{\gamma} a'}}$$

Das ergibt die Typsubstitution

$$\gamma = \{a' \mapsto a, b \mapsto a\}$$

Wir erhalten dann entweder `(map id):: [a'] → [a']` oder `(map id):: [a] → [a]`, was jedoch aufgrund der All-Quantifizierung bis auf Umbenennung identische Typen sind.

Definition 2.6.9 Sei V eine Menge von Typvariablen und γ, γ' zwei Typsubstitutionen. Dann ist γ allgemeiner als γ' (bzgl. V), wenn es eine weitere Typsubstitution δ gibt, so dass für alle Variablen $x \in V$: $\delta(\gamma(x)) = \gamma'(x)$.

Um das zum Vergleich von Typsubstitutionen anzuwenden, nimmt man normalerweise V als Menge der Typvariablen die in der Typgleichung vor der Unifikation enthalten sind.

Beispiel 2.6.10 Nimmt man $V = \{a_1\}$, dann ist $\gamma = \{a_1 \mapsto (a_2, b_2)\}$ allgemeiner als $\gamma' = \{a_1 \mapsto (\text{Int}, \text{Int})\}$, denn man kann γ' durch weitere Instanziierung von γ erhalten: Nehme $\delta = \{a_2 \mapsto \text{Int}, b_2 \mapsto \text{Int}\}$. Dann ist $\delta(\gamma(a_1)) = (\text{Int}, \text{Int}) = \gamma'(a_1)$.

Beispiel 2.6.11 Der Typ der Liste `[1]` kann folgendermaßen ermittelt werden:

- $[1] = 1 : []$
- $1 :: \text{Int}$ und $[] :: [b]$ folgt aus den Typen der Konstanten.
- $(:) :: a \rightarrow [a] \rightarrow [a]$
- Anwendung der Regel mit $\gamma = \{a \mapsto \text{Int}\}$ ergibt:
 $(1 :) :: [\text{Int}] \rightarrow [\text{Int}]$
- Nochmalige Anwendung der Regel mit $\gamma = \{b \mapsto \text{Int}\}$ ergibt:
 $(1 : []) :: [\text{Int}]$

Beispiel 2.6.12 Wir weisen nach, dass es keinen Typ von `[1,'a']` gibt: Der voll geklammerte Ausdruck ist $1 : ('a' : [])$.

$1 :: \text{Int}$, $[] :: [b]$ und $'a' :: \text{Char}$ folgen aus den Typen der Konstanten.

Wie oben ermittelt man: $(1 :) :: [\text{Int}] \rightarrow [\text{Int}]$ und

$'a' : [] :: [\text{Char}]$.

Wenn wir jetzt die Typregel anwenden wollen, dann stellen wir fest: es gibt kein γ , das $[\text{Int}]$ und $[\text{Char}]$ gleichmacht. D.h. die Regel ist nicht anwendbar.

Da das auch der allgemeinste Versuch war, einen Typ für diese Liste zu finden, haben wir nachgewiesen, dass die Liste `[1,'a']` keinen Typ hat; d.h. der Typchecker wird sie zurückweisen.

```
> [1,'a']
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : [1,'a']
*** Type       : Num Char => [Char]
```

Beispiel 2.6.13 Wir zeigen, wie der Typ von `(map quadrat [1,2,3,4])` ermittelt wird.

- `map` hat den Typ $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$
`quadrat` den Typ $\text{Integer} \rightarrow \text{Integer}$, und $[1, 2, 3, 4] :: [\text{Integer}]$.
- Wir nehmen $\gamma = \{a \mapsto \text{Integer}, b \mapsto \text{Integer}\}$.
- Das ergibt $\gamma(a) = \text{Integer}$, $\gamma([a]) = [\text{Integer}]$, $\gamma([b]) = [\text{Integer}]$.
- Damit ergibt sich mit obiger Regel, dass das Resultat vom Typ $\gamma([b]) = [\text{Integer}]$ ist.

Etwas komplexer ist die Typisierung von definierten Funktionen, insbesondere von rekursiv definierten. Man benötigt auch weitere Regeln für Lambda-Ausdrücke, `let`-Ausdrücke und List-Komprehensionen.

In Haskell und ML-Varianten wird im wesentlichen der sogenannte Typcheckalgorithmus von Robin Milner verwendet. Dieser ist i.a. schnell, aber hat eine sehr schlechte worst-case Komplexität: in seltenen Fällen hat er exponentiellen Platzbedarf. Dieser Algorithmus liefert allgemeinste Typen im Sinne des Milnerschen Typsystems. Allerdings nicht immer die allgemeinsten möglichen polymorphen Typen, wie wir im späteren Kapitel sehen werden. Folgender Satz gilt im Milner-Typsystem. Wir formulieren ihn für Haskell.

Satz 2.6.14 *Sei t ein getypter Haskell-Ausdruck, der keine freien Variablen enthält (d.h. der geschlossen ist). Dann wird die Auswertung des Ausdrucks t nicht mit einem Typfehler abbrechen.*

Dieser Satz sollte auch in allen anderen stark typisierten Programmiersprachen gelten, sonst ist die Typisierung nicht viel wert.

Wir untermauern den obigen Satz, indem wir uns die Wirkung der Beta-Reduktion auf die Typen der beteiligten Ausdrücke anschauen.

Erinnerung Beta-Reduktion: $\frac{((\lambda x.t) s)}{t[s/x]}$.

Der Typ von $\lambda x.t$ sei $\tau_1 \rightarrow \tau_2$. Der Typ von s sei σ . Damit der Ausdruck $((\lambda x.t) s)$ einen Typ hat, muss es eine (allgemeinste) Typsubstitution γ geben, so dass $\gamma(\tau_1) = \gamma(\sigma)$ ist. Der Ausdruck hat dann den Typ $\gamma(\tau_2)$.

Jetzt verwenden wir γ um etwas über den Typ des Resultatterms $t[s/x]$ herauszufinden. Dazu muss man wissen, dass der Milner-Typcheck nur den Typ der Unterterme beachtet, nicht aber deren genaue Form. Verwendet man γ , dann hat unter γ der Ausdruck s und die Variable x den gleichen Typ; jeweils als Unterterme von t betrachtet. D.h. Der Typ von $t[s/x]$ ist unter der Typsubstitution γ gerade $\gamma(\tau_2)$.

Da man das für jede Typsubstitution machen kann, kann man daraus schließen, dass die Beta-Reduktion den Typ erhält.

Leider ist die Argumentation etwas komplizierter, wenn die Beta-Reduktion innerhalb eines Terms gemacht wird, d.h. wenn t Unterausdruck eines anderen Ausdrucks ist, aber auch in diesem Fall gilt die Behauptung, dass die Beta-Reduktion den Typ nicht ändert.

Analog kann man die obige Argumentation für andere Auswertungsregeln verwenden.

2.7 Typklassen und Überladung in Haskell

Typklassen wurden entwickelt, um Überladung von Funktionen im polymorphen Typsystem systematisch zu benutzen und auch benutzerdefinierbar zu machen. (Überladung: Es gibt mehrere Funktionen mit gleichem Namen). Dazu muss spätestens zum Zeitpunkt der Ausführung bekannt sein, welche Funktion denn tatsächlich ausgeführt werden soll. Dieses Problem ist praxisrelevant, denn es ist nicht einzusehen, dass man z.B. $2+5$ schreiben darf, aber bei komplexen Zahlen statt $(1+i) + (3+2i)$ in etwa schreiben müsste: $(1+i)$ `complexadd` $(3+2i)$. Als weiteren Nachteil eines Überladungsverbots würden sich die Namen der

allgemeinen Funktionen vermehren, z.B. Summe einer Liste müsste für reelle Zahlen und komplexe Zahlen extra definiert werden.

Ein spezifisches Problem der Typklassen in Milner-getypten Sprachen wie Haskell ist die korrekte Kombination der normalen Typisierung mit Typklassen.

- $+$, $*$, $-$, $/$ will man für verschiedene Arten von Zahlen verwenden: `Integer`, `Float`, `Rational`, ...
- Es gibt Funktionalitäten, die nicht auf allen Typen definierbar sind, sondern nur auf einer Menge von vorgegebenen Typen: z.B. Gleichheit (`==`) oder die Funktion `show`, die Objekte in eine druckbare Form überführt.

2.7.1 Typklassen

Eine Typklasse kann man sich vorstellen als eine Menge von Typen zusammen mit zugehörigen Funktionen bzw. Methoden. Die Typklassen werden als eigene syntaktische Einheiten (d.h. sie haben einen Namen) eingeführt.

Typklassendefinition

Eine neue Typklasse in Haskell kann definiert werden durch:

```
class <Vorbedingung> => NEWCLASS(a)
```

<Vorbedingung>: Klassenbedingungen an die Variable a

NEWCLASS: neuer Klassenname. Muss mit Großbuchstabe beginnen

Im allgemeinen gibt es nur eine Variable a . Als <Vorbedingung> ist in Haskell nur eine Unterklassenbeziehung von Typklassen erlaubt.

Als Beispiel zeigen wir den Anfang der Definition der Typklasse `Num` in Haskell:

```
class (Eq a, Show a) => Num a
```

Hier bedeutet `Eq a`, dass der Typ a zur Typklasse `Eq` gehört. Die Vorbedingung `(Eq a, Show a) => Num a` bedeutet: Nur für Typen a , die bereits in den Klassen `Eq`, `Show` sind, wird a auch in `Num` definiert: d.h. `Num` ist Unterklasse von `Eq`, `Show`. Von der logischen Schreibweise her kann das missverständlich sein, da bei der Betrachtung von Mengen und Aussagenlogik normalerweise $A \Rightarrow B$ zu $A \subseteq B$ passt, während `Eq a => Num a` in Haskell die Bedeutung `Num` \subseteq `Eq` hat.

Die Deklaration von Typklassen und die Deklaration von Typen von Funktionen erfordert eine erweiterte Syntax:

Die Syntax für Typen kann eine Vorbedingung an die Zugehörigkeit von Typvariablen zu Typklassen enthalten.

```
class <Vorbedingung> => <Typ>
```

Beispiel 2.7.1 *Der Typ von `==` ist*

```
Eq a => a -> a -> Bool
```

Das bedeutet, dass für alle Typen `a`, die in `Eq` sind, die Gleichheit `==` den Typ `a -> a -> Bool` hat.

Diese Schreibweise kann man auch folgendermaßen interpretieren:

Ein Ausdruck hat alle Grundtypen, die Instanz des Typs sind, wobei man bei der Einsetzung die Typklassenbedingung beachten muss.

Bei der aktuellen Verwendung von `==` in einem Ausdruck wird der Typchecker dann prüfen, ob die auf Gleichheit getesteten Objekte tatsächlich in der Klasse `Eq` sind; und natürlich, ob sie den gleichen Typ haben. Wenn nein, wird der Ausdruck als ungetypt zurückgewiesen. Wenn ja, wird der berechnete Typ verwendet, um evtl. den zu benutzenden Gleichheitstest direkt einzusetzen. In `Eq` sind alle Typen, für die ein Gleichheitstest implementiert worden ist. Nicht sinnvoll ist der Vergleich von Funktionen mit `==`, obwohl man das auch definieren kann, beispielsweise für Funktionen mit endlichen Definitionsbereich.

Beispiel 2.7.2 *Definition der Typklasse Eq (im Haskell-Prelude):*

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y    = (not (x == y))
```

Vergleicht man das mit der üblichen objektorientierten Programmierung (z.B. Java), so entspricht `Eq` einer Klasse, die Methoden sind `==` und `/=`. Für die Methode `==` wird nur die Schnittstelle zur Verfügung gestellt, während `/=` standardmäßig bereits aufbauend auf `==` definiert wird. Die Typen der Methoden werden ebenfalls festgelegt und können nicht überschrieben werden.

Umgekehrt ist es nicht sinnvoll, alle Klassen einer objektorientierten Sprache auch als Typklassen in Haskell zu implementieren. Oft ist die richtige Übersetzung ein Typ: z.B. Listen.

Danach kann man noch selbst-definierte Typen zu Instanzen der Typklasse `Eq` erklären. Oder auch rekursive Regeln zur Erzeugung von Instanztypen angeben. Dies bedeutet: `==` (ist gleich) muss bei jeder solchen Instanzdefinition mit definiert werden, `/=` (ungleich) ist dann automatisch ebenfalls definiert. Allerdings gibt es auch einen Default (`.. deriving Eq`), der die Strukturgleichheit implementiert, wobei rekursiv wieder `==` verwendet wird.

Beispiel 2.7.3

```
instance Eq Char where
    x == y = fromEnum x == fromEnum y
```

Dies erklärt `Char` zu einem Typ der Typklasse `Eq`, auf den `==` angewendet werden kann. Gleichheit auf Zahlen wird im Haskell-Prelude mittels eingebauter Funktionen implementiert.

Beispiel 2.7.4

```
instance Eq Int     where (==)    = primEqInt
instance Eq Integer where (==)    = primEqInteger
instance Ord Int   where compare = primCmpInt
instance Ord Integer where compare = primCmpInteger
```

Bei (selbstdefinierten) algebraischen Datentypen, die auch noch rekursiv definiert sind (hier Beispiel Listen) ist das etwas komplizierter. Hier wird die komplette Gleichheitsdefinition für Listen implementiert.

```
instance Eq a => Eq [a]     where
  [] == []           = True
  [] == (x:xs)       = False
  (x:xs) == []       = False
  (x:xs) == (y:ys)  = x == y && xs == ys
```

- Dies ist eine rekursive Definition einer unendlichen Menge von Eq-Typen
- Die Gleichheit ist explizit für alle Konstruktormöglichkeiten (`[]` und `:`) definiert.
- In Haskell kann statt der expliziten Implementierung eine Default-Definition benutzt werden, bei der direkt an der Typdeklaration steht: `deriving (Eq,Ord)`. Hierbei werden offenbar die Daten auf Strukturgleichheit getestet.

2.7.2 Verwendung der Definition der Gleichheit

Will man nicht die Standarddefinition der Gleichheit, sondern hat eine andere Gleichheit intendiert, dann ist die entsprechende Implementierung möglich. Hier am Beispiel von Mengen, die als Listen implementiert sind.

Beispiel: Mengen

Die Implementierung als Instanz von `Ord` ist etwas unschön, da man beachten muss, dass `Ord` eigentlich nur für totale (d.h. lineare) Ordnungen konzipiert ist.

```
data Menge a = Set [a]

Instance Eq a => Eq (Menge a)  where
  Set xs == Set ys  = subset xs ys && subset ys xs

subset: Eq a => a -> a -> a
subset xs ys = all ('elem' ys) xs

instance (Ord a) => Ord (Menge a)  where
  Set xs <= Set ys = subset xs ys
  Set xs >= Set ys = subset ys xs
```

```

Set xs < Set ys = subset xs ys && not (subset ys xs)
Set xs > Set ys = subset ys xs && not (subset xs ys)
min (Set xs) (Set ys) =
  if subset xs ys then Set xs else
  if subset ys xs then Set ys
  else error "subset nicht total"
max (Set xs) (Set ys) =
  if subset ys xs then Set xs else
  if subset xs ys then Set ys
  else error "subset nicht total"
compare (Set xs) (Set ys) =
  if xs == ys then EQ else if subset xs ys then LT
  else if subset ys xs then GT
  else error "subset nicht total"

```

Damit kann man erreichen, dass verschiedene algebraische Datentypen auf ihre eigene und korrekte Weise auf Gleichheit verglichen werden, auch wenn diese verschachtelt sind, z.B. Mengen von Mengen, Mengen von Listen, oder Listen von Mengen.

Man kann das Drucken von Mengen analog zum Drucken von Listen implementieren:

```

instance Show a => Show (Menge a) where
  showsPrec p = showMenge

showMenge (Set [])      = showString "{}"
showMenge (Set (x:xs)) = showChar '{' . shows x . showm xs
  where showm []        = showChar '}'
        showm (x:xs)   = showChar ',' . shows x . showm xs

```

Testen der Mengendarstellung ergibt:

```

Main>> Set [[1],[2,3]]
{[1],[2,3]} :: Menge [Integer]

```

```

Main> Set [Set [1], Set [1,2]]
{{1},{1,2}} :: Menge (Menge Integer)

```

Übungsaufgabe 2.7.5 *Implementiere Multimengen analog zu Mengen.*

2.7.3 Die Typklasse Ord

Das ist die Klasse der Typen mit einer totalen Ordnungsrelation $<$.

Beispiel 2.7.6 *Es folgt die Definition der Typklasse Ord aus dem Prelude von Haskell.*

```

data Ordering = LT | EQ | GT
    deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)

class (Eq a) => Ord a where
    compare      :: a -> a -> Ordering
    (<), (<=), (>=), (>)  :: a -> a -> Bool
    max, min     :: a -> a -> a

    -- Minimal complete definition: (<=) or compare
    -- using compare can be more efficient for complex types
    compare x y
        | x==y      = EQ
        | x<=y      = LT
        | otherwise = GT

    x <= y          = compare x y /= GT
    x < y           = compare x y == LT
    x >= y          = compare x y /= LT
    x > y           = compare x y == GT

    max x y
        | x >= y    = x
        | otherwise = y
    min x y
        | x <= y    = x
        | otherwise = y

```

Die primitive Operation ist offensichtlich \leq , die anderen sind darauf aufgebaut. Damit kann man nicht nur Zahlen und Character vergleichen, sondern auch Tupel und Listen. Zum Beispiel gilt $[1,2] < [1,2,3]$, $[1,2] < [1,3]$, und $(1, 'c') < (2, 'b')$. D.h. , es wird die lexikographische Ordnung auf Tupeln verwendet. Unglücklicherweise sind in Haskell98 Tupel nur bis zu 5 -Tupeln vordefiniert. Will man größere Tupel verwenden, so muss man die Definitionen selbst machen.

Verwendet man in einer Datentypdefinition `deriving (Eq,Ord)`, werden offenbar die Datenkonstruktoren ihrem Index nach geordnet, und Komponenten lexikographisch.

2.7.4 Typklassen Read und Show

Die Klasse `Show` enthält alle Typen, deren Objekte eine Repräsentation als Text haben. Die Klasse `Read` enthält die Typen, deren Objekte aus einer Textrepräsentation zu rekonstruieren sind. Wenn ein Typ sowohl zu `Show` als auch `Read` gehört, dann sollte die Textrepräsentation wieder eindeutig zu parsen sein und das Objekt ergeben, d.h. für alle sinnvollen x muss gelten: `read(show(x)) = x`.

Funktionen (Methoden) für die Klassen `Read`, `Show`:

```
shows  :: Show a => a -> ShowS
reads  :: Read a => ReadS a
show   :: Show a => a -> String
read   :: Read a => String -> a
type   ReadS a = String -> [(a,String)]
type   ShowS  = String -> String
```

Diese sind für einige eingebaute Typen bereits vordefiniert und müssen für andere entsprechend definiert werden.

Der technische Grund, zunächst die Funktionen `shows` und `reads` zu definieren, hat ihren Grund in der besseren Ressourcennutzung. Analog zum `fold` auf Bäumen ist es besser, hier mittels Funktionskomposition zu programmieren, auch wenn dies etwas umständlicher und gewöhnungsbedürftig ist.

2.7.5 Mehrfache Vererbung

Es gibt bei der Definition von Typklassen Möglichkeiten analog zur multiplen Vererbung:

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

  -- Minimal complete definition: All, except negate or (-)
  x - y          = x + negate y
  negate x      = 0 - x

instance Num Int where
  (+)          = primPlusInt
  (-)          = primMinusInt
  negate      = primNegInt
  (*)         = primMulInt
  abs         = absReal
  signum     = signumReal
  fromInteger = primIntegerToInt
```

Die numerischen Typklassen müssen somit die Methoden von `Eq` und `Show` implementiert haben. Interessanterweise ist dies analog zum Java-Mechanismus der multiplen Vererbung, die nur Interface-Vererbung erlaubt.

2.7.6 Auflösung der Überladung

Wird an einer Stelle in einer Funktion eine überladene Funktion (Methode) verwendet, dann muss das Typsystem während der Compilezeit dafür sorgen, dass die Überladung aufgelöst wird, bzw. zur Laufzeit aufgelöst werden kann. Es wird durch den Typechecker in Kombination mit dem Compiler sichergestellt, dass eine Anwendung von Methoden auf primitive Objekte mit der richtigen primitiven Methode erfolgt.

!!! Im Haskell-Laufzeitsystem gibt es keine Typinformationen mehr. D.h. man kann zur Laufzeit Listen-Objekte von Bäumen nicht unterscheiden (mittels Tags). Insbesondere ist eine Java-Methode wie `instanceof` in Haskell nicht möglich.

Zum Zeitpunkt des Typchecks hat das in früheren Haskell-Versionen manchmal merkwürdige Konsequenzen: z.B. ließ sich `[] == []` nicht typen, da ein `[]` von Typ `[a]`, das andere von Typ `[b]` ist, und über `a, b` nichts weiter bekannt ist. In aktuellen Haskellversionen, z.B. `ghci`-Interpreter, kann der Typchecker bei Eingabe von `[] == []` erkennen, dass der Typ `a` in `[] :: [a]` redundant ist, und somit ergibt sich kein Fehler.

Im Vergleich dazu existieren im Java Laufzeitsystem noch die Typinformationen und können auch in Abfragen verwendet werden.

Die interne Überladungsauflösung in Haskell funktioniert so, dass überladene Funktionen, die zum Zeitpunkt des Typchecks keinen eindeutigen Typ haben, intern einen zusätzlichen Parameter, den sogenannten `Dictionary`-Parameter bekommen. Zum Aufrufzeitpunkt einer Funktion muss dieser Parameter dann soweit bekannt sein, dass die zugehörige Implementierung im Dictionary gefunden werden kann. Außerdem wird intern vom Compiler ein "Dictionary" aufgebaut, das zur Laufzeit als normale Haskell-Datenstruktur vorhanden ist, und die Funktionen mit den richtigen "Methoden" versorgt. Wenn der Dictionary-Parameter einem primitiven Typ wie `Int` entspricht, dann wird die zugehörige Gleichheit benutzt. D.h. die rekursive Aufrufhierarchie bzgl. der Typen als Aufrufhierarchie des Dictionary von `==` abgebildet.

Eine Konsequenz dieses Verfahrens ist, dass in Haskell zur Compilezeit genügend viel Typinformation (auch für Unterausdrücke) zur Verfügung stehen muss, um überladene Funktionen mit den richtigen Dictionary-Parametern zu versehen.

Da z.B. die Funktion `==` überladen ist, und nur für Argumente der Typklasse `Eq` zur Verfügung steht, muss für eine Gleichung `s == t` für die Ausdrücke `s` und `t` entweder ein Grundtyp berechnet werden, oder, wenn Typvariablen vorkommen, muss aus dem übergeordneten Ausdruck hervorgehen, welches Dictionary zu benutzen ist. Dieser Dictionary-Parameter ist i.a. ein (implizites) Argument einer übergeordneten Funktion.

Ein Dictionary kann auch rekursiv aufgebaut sein, wobei jedoch das Dictionary endlich sein muss.

2.8 Aufzählungsklasse Enum

Mit dieser Typklasse kann man die Funktionalität von Datentypen erfassen, deren Elemente sich vorwärts und rückwärts aufzählen lassen, wie z.B. nichtnegative ganze Zahlen; die Zeichen eines Alphabets, oder Farben, falls man diese in eine Ordnung bringt. Für Elemente dieser Typen kann man das Aufzählungsverfahren mit der Syntax analog zu Zahlen verwenden: `[2..]`, `['a'..'z']`, `['1','0'..]`, ...

Als Beispiel als ein Auszug aus dem Hugs-Prelude die Definition der Klasse.

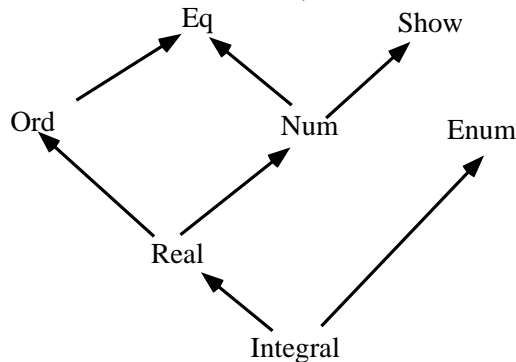
```
class Enum a where
  toEnum      :: Int -> a
  fromEnum    :: a -> Int
  enumFrom    :: a -> [a]           -- [n..]
  enumFromThen :: a -> a -> [a]    -- [n,m..]
  enumFromTo   :: a -> a -> [a]    -- [n..m]
  enumFromThenTo :: a -> a -> a -> [a] -- [n,n'..m]

  enumFromTo x y      = map toEnum [fromEnum x .. fromEnum y ]
  enumFromThenTo x y z = map toEnum [fromEnum x,
                                     fromEnum y .. fromEnum z ]

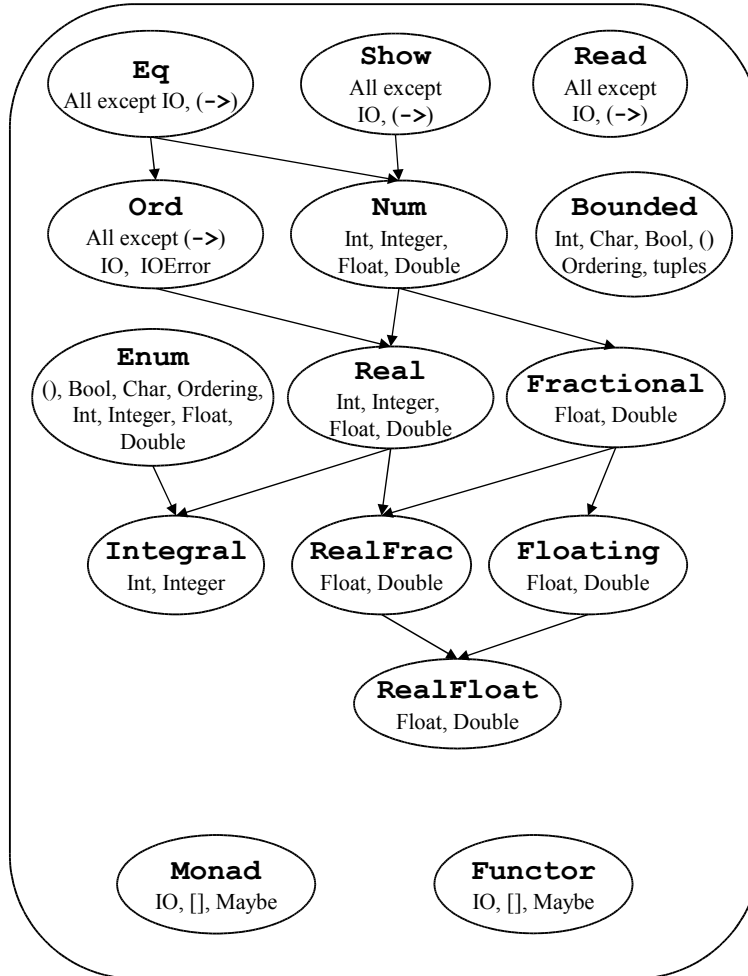
succ, pred :: Enum a => a -> a
succ      = toEnum . (1+)      . fromEnum
pred      = toEnum . subtract 1 . fromEnum

instance Enum Char where
  toEnum      = primIntToChar
  fromEnum    = primCharToInt
  enumFrom c  = map toEnum
               [fromEnum c .. fromEnum (maxBound::Char)]
  enumFromThen c d =
    map toEnum [fromEnum c, fromEnum d .. fromEnum (lastChar::Char)]
    where lastChar = if d < c then minBound else maxBound
```

Typklassenhierarchie (Ausschnitt aus den vordefinierten Typklassen)



Hier das Bild der vordefinierten Klassen aus dem Haskell-Report:



Beispiel 2.8.1 *Definition einer überladenen Konstanten.*

Die Klasse `Finite` hat als einzige Methode die Funktion `members`, die alle Elemente des Typs als Liste enthält.

```

class Finite a where members :: [a]

instance Finite Bool where members = [True, False]

instance (Finite a, Finite b) => Finite (a, b) where
    members = [(x,y) | x <- members, y <- members]
    
```

Dies definiert eine Konstante in einer Typklasse, die abhängig von ihrem Typ-kontext ihren Wert ändert. Weitere Typen können mittels `Bool` und Paarbildung zusammengesetzt werden.

Damit erhält man z.B.

```
members::Bool -> [ True, False]
length (members :: [(Bool, Bool), (Bool, Bool)]) ---> 16
```

Beispiel 2.8.2 *Programmieren die allgemeine Summe über alle Listen von Typen, die zur Typklasse Num gehören.*

Der folgende Versuch scheiterte in einer vorherigen Version von Haskell:

```
summe [] = 0
summe (x:xs) = x+summe xs
```

Der Grund war, dass die Zahl 0 beim Einlesen den festen Typ Int hatte. Dies wurde in der aktuellen Haskell-Version so abgeändert, dass ganzzahlige Konstanten als Integer erkannt werden und dann mit der Konversionsfunktion fromInteger eingepackt werden.

Die allgemeine Summe von Zahlen (Objekten von einem Typ der Typklasse Num) über eine Liste erhält man folgendermaßen (noch effizienter mit Akkumulator):

```
allgemeine_summe :: Num a => [a] -> a
allgemeine_summe [] = 0
allgemeine_summe (x:xs) = x + allgemeine_summe xs
```

2.8.1 Klasse Functor

Diese Klasse kann man verwenden, um die Funktion map für eine Menge von Typen zu verwenden, die eine vergleichbare map-Funktion wie Listen haben. In der aktuellen Haskell-Version wird der Namen fmap statt map verwendet.

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)

data Maybe a = Just a | Nothing
    deriving (Eq, Ord, Read, Show)

instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)

instance Functor [] where
    -- [] bedeutet hier List als Typkonstruktor, nicht nil
    fmap = map
```

Sinnvoll ist es, auch andere Datenstrukturen, für die man ein map definieren kann, zu Instanzen von Functor zu machen.

```

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

instance Functor Baum where
    fmap = baummap

baummap f (Blatt a) = Blatt (f a)
baummap f (Knoten x y) = Knoten (baummap f x) (baummap f y)

test = fmap (\x -> 2*x) (Knoten (Blatt 1) (Blatt 2))

```

Der Nutzen ist hier nicht so groß wie bei `Eq` und `Ord`, da diese Funktion nicht rekursiv über verschiedene Typen operiert.

2.8.2 Verallgemeinerungen von Typklassen

Eine Verallgemeinerung von Typklassen ist die Möglichkeit, mehr als eine Typvariable in der Definition einer Typklasse zu erlauben, bzw. komplexer strukturierte Typen.

Ein Problem dieser Verallgemeinerungen ist die Frage, ob es einen entscheidbaren Typcheckalgorithmus für Typen und Typklassen gibt. Pragmatischer muss man fragen, ob es einen brauchbaren Typcheck gibt, ob man den Compiler mit Überladungsauflösung noch sinnvoll implementieren kann, und welche Vorteile und Änderungen dies für die Programmierung in Haskell bedeutet.

Es ist bekannt, dass es der Typcheck in GHC unentscheidbar wird, wenn man Multi-Parameter Typklassen und noch einige hier nicht besprochene Erweiterung hat: *functional dependencies*, *undecidable instances*. Diese sind alle im GHC als Erweiterung verfügbar; es gibt sogar eine Implementierung des (Turing-vollständigen) SKI-Kombinator-Kalküls nur auf Basis des Typ-checks.

Möglicherweise ist ein Typsystem einfacher in der Praxis zu handhaben, das abhängige Typen (*dependent types*) verwendet. Dies würde bedeuten: man hat ein sehr ausdrucksstarkes, allerdings unentscheidbares Typsystem. Damit kann man den Typ von Objekten von Werten von Variablen abhängig machen. Ein einfaches Beispiel sind Matrizen, bei denen man in der Typbeschreibung die Anzahl der Zeilen und Spalten angeben muss. Diese sind i.a. nur dynamisch bekannt und sind Zahlenwerte im Programm. Der einfachste Fall in Haskell sind Tupel, die man in einem *dependent-type* System in vollem Umfang implementieren könnte. Zur Erinnerung: In Haskell sind manche Tupelfunktionalitäten nicht vordefiniert; man bräuchte unendlich viele Funktionen, um alles in Haskell vorzudefinieren, wie z.B. extrahiere *i*-tes Element eines *n*-Tupels. Es gibt experimentelle (auch funktionale) Programmiersprachen, die das Konzept der *dependent-types* implementiert haben.