Funktionale Programmierung I, Sommersemester 2000

Prof. Dr. Manfred Schmidt-Schauß
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
E-mail:schauss@ki.informatik.uni-frankfurt.de

No Institute Given

1 Die Kernsprache KFP

In diesem Kapitel wollen wir zunächst eine funktionale Kernsprache einführen, auf die man die höheren Konstrukte einer funktionalen Programmiersprache zurückführen kann. Diese Kernsprache ist für Grundlagenbetrachtungen für nicht-strikte funktionale Programmiersprachen geeignet. Wenn man sie leicht verändert, dann ist sie auch für strikte FPS als Basis verwendbar.
Alle Definitionen in diesem Kapitel beziehen sich auf die Kernsprache KFP.

1.1 Die funktionale 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, aber wie sich herausstellen wird, nicht die minimal mögliche Syntax. KFP hat ein schwaches Typsystem (Programme mit Typfehlern sind möglich). Beweise von Eigenschaften sind möglich und überschaubar, und können auch auf eine darauf aufgebaute funktionale Programmiersprache übertragen werden. Ebenso sind Analysen von Programmen einfacher zu entwerfen und zu durchschauen.
Diese Kernsprache folgt der Lambda-Schreibweise im Lambda-Kalkül, der von Alonzo Church entwickelt wurde. Der Lambda-Kalkül ist Teil der betrachteten Kernsprache.

Syntax:
Es wird unterschieden zwischen der Angabe von Typen und der Definition von Ausdrücken. Wir nehmen an, daß es irgendeine Möglichkeit gibt, Typen und zugehörige Konstruktoren mit Stelligkeit anzugeben, ohne dies näher zu spezifizieren. Zum Beispiel als eine Auflistung der Typen zusammen mit den zugehörigen Konstruktoren und deren Stelligkeit. Z.B.

```
Bool
     True  0
     False 0
List
     Nil 0
     Cons 2
......
```

Danach kann man Ausdrücke definieren. In denen kommen vor:

- Variablen,
- Konstruktoren (gehören jeweils zu einem Typ und haben eine feste Stelligkeit $ar(c)$),
- `case`-Ausdrücke, wobei es für jeden Typ ein `case`-Schema gibt.
- Funktionsabstraktionen (Abstraktionen),
- Terme (Ausdrücke, Expressions),

Definition 1.1. *Eine einfache kontextfreie Grammatik für KFP-Ausdrücke (Terme, Expressions) ist:*

EXP ::= *Konstruktor* | *Variable* |
$\lambda V\ .\ EXP$ | *wobei* V *eine Variable ist.*
$(EXP\ EXP)$ |
$(\texttt{case}_{TYP}\ EXP\ (Pat_1 \to Exp_1)\dots(Pat_n \to Exp_n))$
Hierbei muß im case-Ausdruck n die Anzahl der Konstruktoren von TYP sein und für jeden Konstruktor von TYP muß es genau ein Pattern geben.
Das Konstrukt $Pat_i \to Exp_i$ *heißt auch* `case`*-Alternative.*

Pat ::= *Konstruktor* $V_1 \dots V_n$
Hier muß n die Stelligkeit des Konstruktors sein.
Die Variablen V_i *müssen alle verschieden sein.*
Diese Konstrukte nennt man Pattern *(*Muster*)*

Beispiel 1.2. Die Funktion, die erkennt, ob eine Liste leer ist, könnte man schreiben als:

```
\ xs . (case_List xs (Nil -> True) ((Cons y ys) -> False))
```

Ausdrücke werden folgendermaßen unterschieden:

- $\lambda x.s$ sind *Abstraktionen* oder Funktionsausdrücke.
- $(s\ t)$ sind *Anwendungen (Applikationen).*
- $(\texttt{case}_T\ e\ ALTS)$ sind `case`*-Ausdrücke (Fallunterscheidungen).*

Der Umgang mit Klammern ist folgendermaßen geregelt: $((x\ y)\ z)$ ist dasselbe wie $(x\ y\ z)$. D.h. man kann wie im Beispiel oben statt $((Paar\ x)\ y)$ auch $(Paar\ x\ y)$ schreiben. Der erste Ausdruck entspricht der Syntax. Dasselbe gilt für Patternausdrücke. $(Paar\ x\ y)$ ist ein Pattern, das auch als $((Paar\ x)\ y)$ geschrieben werden kann. Dies wird auch `Currying` genannt (Haskell B. Curry).

Beachte, dass $(x\ (y\ z))$ **nicht** dasselbe wie $(x\ y\ z)$ ist.
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).
Zur Kernsprache von Haskell gehört auch noch ein sogenanntes `letrec`. Läßt man `letrec` zu, so gewinnt man Effizienz beim Übersetzen und kann auch eine abstrakte Maschine besser und effizienter formulieren. Die formale Behandlung eines `letrec` in der Kernsprache ist sehr aufwendig und unübersichtlich. Zudem kann man mit einer kleineren Kernsprache KFP für formale Untersuchungen der wesentlichen Aspekte der operationalen Semantik auskommen. Wir lassen deshalb `letrec` in der Kernsprache weg.

Beispiel 1.3. Der Ausdruck $\lambda x.x$ ist die Identitätsfunktion.
Der Datentyp `Bool` besteht aus den Konstruktoren `True`, `False`. Den gewohnten Ausdruck

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

kann man darstellen durch den Ausdruck

$$(\texttt{case}_{\texttt{Bool}}\ A\ (\texttt{True} \rightarrow B)\ (\texttt{False} \rightarrow C))$$

Der Typ `Paar` kann durch einen zweistelligen Konstruktor `Paar` modelliert werden. D.h. in KFP hat er genau diesen zweistelligen Konstruktor `Paar`. Ein Paar von zwei Objekten `True`, `False` ist dann darstellbar als (`Paar True False`). Die Abfrage, ob ein Paar als erstes Element ein `False` hat, kann man schreiben als:

$$(\texttt{case}_{\texttt{Paar}}\ p\ ((\texttt{Paar}\ x\ y) \rightarrow (\texttt{case}_{\texttt{Bool}}\ x\ (\texttt{True} \rightarrow \texttt{False})(\texttt{False} \rightarrow \texttt{True}))))$$

Der Typ `Liste` kann durch zwei Konstruktoren `Cons`, `Nil` definiert werden, wobei der Konstruktor `Cons` zweistellig ist und `Nil` eine Konstante ist. Ein Liste mit den beiden Elementen `True`, `False` ist dann darstellbar als

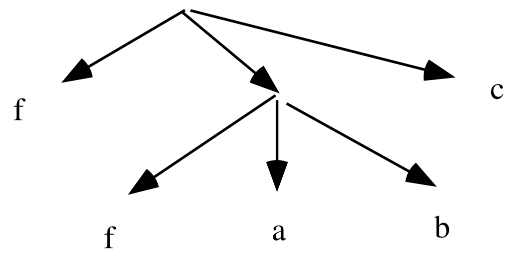
$$\texttt{Cons True (Cons False Nil)}$$

in üblicher Notation auch als [`True`, `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.

Beispiel 1.4. Für Ausdrücke und Funktionen.

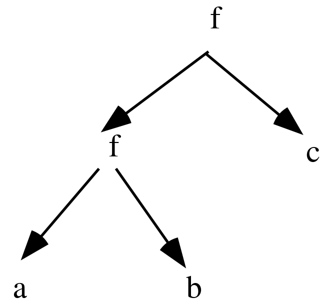
$$\begin{aligned} Id\ \ &::= \lambda x.x \\ K\ x\ y &::= \lambda x.\lambda y.x \end{aligned}$$

Die Darstellung von Ausdrücken als markierte geordnete Bäume wird oft verwendet, wenn auch nur implizit: Als Beispiel betrachte $f\ (f\ a\ b)\ c$

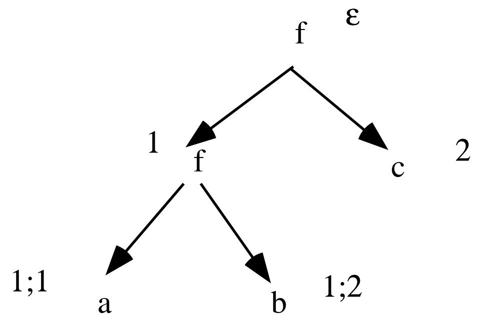


Der erste Term in einer Liste von Termen ist die Funktion, der Rest sind die Argumente:

Man kann eine kompaktere Darstellung wählen:

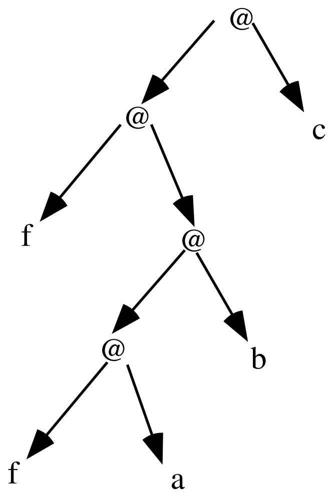


Bei Termen als Bäumen werden wir auch die Adressierungsmethode in Bäumen verwenden:



Die Wurzel (mit Markierung f) hat Adresse ε (leeres Wort) Der Knoten mit Markierung c hat Adresse 2. Die Blätter mit den Markierungen a, und b haben die Adressen 1;1 und 1;2.

Die Darstellung ohne Currying ist folgendermaßen. Die Knoten die mit @ markiert sind, sind Anwendungsknoten.



1.2 Freie und gebundene Variablen

Im Ausdruck $(\lambda x\ .\ (y\ x))$ ist x eine gebundene Variable (durch λ gebunden), und y eine freie Variable.

Definition 1.5. *Die Bindungsregeln in KFP sind:*

- *Im Ausdruck $\lambda x\ .\ E$ ist x eine gebundene Variable, der Gültigkeitsbereich (Skopus) ist E.*
- *In der* `case`*-Alternative $((c\ x_1 \ldots x_n) \to E)$ werden die Variablen x_i 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 1.6.
$$\lambda x\ .\ (x\ (\lambda x\ .\ x))$$
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 vereinfachen, 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 Bindungsbereiche sichtbar, aber sie löst den Konflikt selbst nicht auf. Ausdrücke die bis auf Umbenennung gleich sind, nennt man auch α-äquivalent.

Definition 1.7. *Ein Ausdruck ohne freie Variablen heißt* geschlossener Ausdruck*, anderenfalls* offener Ausdruck*. Ein* Programm *ist ein geschlossener Ausdruck.*

Beispiel 1.8. Der Ausdruck

$$\lambda x\ .\ (\texttt{case}_{\texttt{List}}\ \texttt{x}\ (\texttt{cons y ys} \rightarrow \texttt{ys})\ (\texttt{Nil} \rightarrow \texttt{Nil}))$$

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

$$(\texttt{case}_{\texttt{List}}\ \texttt{x}\ (\texttt{cons y ys} \rightarrow \texttt{ys})\ (\texttt{Nil} \rightarrow \texttt{Nil}))$$

Bei Lambda-Ausdrücken, die von der Form $\lambda x_1\ .\ (\lambda x_2\ .\ \ldots (\lambda x_n\ .\ e))$ sind, werden wir der Einfachheit halber manchmal $(\lambda x_1, \ldots, x_n\ .\ e)$ schreiben.

1.3 Auswertungsregeln

Was uns jetzt noch fehlt, ist die operationale Semantik, bzw. die Erklärung: wie wertet man Ausdrücke aus? Diese operationale Semantik ist als Transformationssystem auf den Ausdrücken selbst definiert. Die Werte, die man am Ende 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, was die Auswertung dann nacheinander macht, sind daraus abgeleitet.
Die Vorgehensweise ist im Sinne von Plotkin, der eine Programmiersprache als aus vier Komponenten bestehend definiert: Ausdrücke, Kontexte, Werte, und eine Auswertungsrelation auf Ausdrücken.

Definition 1.9. *Ein* `Wert` *bzw. eine WHNF (weak head normal form, schwache Kopfnormalform)* [1] *in KFP ist ein Ausdruck entweder von der Form $c\ t_1 \ldots t_n$, wobei $n \leq arity(c)$ und c ein Konstruktor ist, oder eine Abstraktion: $\lambda x\ .\ e$.*
Wir unterscheiden die Werte nach ihrer Struktur als

- FWHNF*, wenn die WHNF eine Abstraktion ist, oder ein Ausdruck der Form $(c\ s_1 \ldots s_n)$, c ein Konstruktor und $ar(c) > n$, oder*
- SCWHNF *(gesättigte* **c***onstructor WHNF), wenn die WHNF ein Ausdruck der Form $(c\ s_1 \ldots s_n)$, c ein Konstruktor und $ar(c) = n$*

[1] Eine HNF oder Kopfnormalform ist ein Ausdruck der einen Lambda-Präfix hat und als erste Nichtabstraktion muß $(x\ t_1 \ldots t_n)$ oder $(c\ t_1 \ldots t_n)$ 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 (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 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 richtige 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.
- Im Pattern $((c\ x_1 \dots x_n) \to e)$: ersetze x_i durch bisher nicht verwendete und verschiedene Variablen z_i. In e ersetze alle freien Vorkommen von x_i durch z_i: Das neue Pattern ist: $((c\ z_1 \dots z_n) \to e[z_1/x_1, \dots z_n/x_n])$

Beispiel 1.10. 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.

Übungsaufgabe 1.11. Zeige: Geschlossene Terme werden durch die Reduktion in geschlossene Terme überführt. Gilt das auch für offene Terme?

Diese Auswertungsregeln sollen in allen Programmkontexten gelten. Dazu definieren wir zunächst, was *Programmkontexte* sind:

Definition 1.12. *Ein* Programmkontext (ein Kontext) *ist ein Ausdruck, der an einer Stelle ein "Loch" hat, an dem ein Ausdruck eingesetzt werden kann. Formal definieren wir: Ein Kontext kann sein:*

$$\begin{aligned} C ::= &\ [] \mid (e\ C) \mid (C\ e) \mid (\lambda x\ .\ C) \mid \\ &\ \texttt{case}\ C\ \ (p_1 \to t_1) \dots (p_n \to t_n) \mid \\ &\ \texttt{case}\ t\ \ (p_1 \to t_1) \dots (p_i \to C) \dots (p_n \to t_n) \end{aligned}$$

Beispiel 1.13. Damit sind als Kontexte z.B. definiert:

- $[]$: der leere Kontext. Setzt man einen Ausdruck e in den leeren Kontext ein, so ergibt sich e selbst.
- $(x\ (\cdot\ y))$. Einsetzen von e in den Kontext ergibt $(x\ (e\ y))$

– $\texttt{case}\ (x\ \cdot)\ \ (p_1 \to t_1) \ldots (p_n \to t_n)$. Einsetzen von e in den Kontext ergibt $\texttt{case}\ (x\ e)\ \ (p_1 \to t_1) \ldots (p_n \to t_n)$

Beachte, dass innerhalb eines Pattern kein Loch sein kann, ebenso ist der Kontext $\lambda \cdot .e$ nicht möglich.
Innerhalb von Kontexten erlauben wir das Einfangen von freien Variablen. D.h. es tritt der Effekt auf, dass die Namen von gebundenen Variablen eine Rolle spielen, sofern das Loch innerhalb des Rumpfs einer Abstraktion ist. Z.B. ist der Kontext $\lambda x\ .\ []$ vom Kontext $\lambda u\ .\ []$ verschieden, da beim Einsetzen von x jeweils andere Ausdrücke entstehen. Wir werden den Begriff der Gleichheit und Ungleichheit von Ausdrücken noch formaler spezifizieren.

Definition 1.14. *Auswertungsregeln (Reduktionsregeln):*

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

$$\mathit{Case}\quad \frac{(\texttt{case}_T\ (c\ t_1 \ldots t_n) \ldots (c\ x_1 \ldots x_n \to s) \ldots)}{s[t_1/x_1, \ldots, t_n/x_n]}$$

Wir nennen das die unmittelbare Reduktion *von Ausdrücken.*
Die Auswertungsregeln sollen in allen Programmkontexten verwendet werden dürfen. D.h. Wenn $s \to t$ in einem Schritt auswertet, dann gilt auch $C[s] \to C[t]$ für jeden Programmkontext C.
Die Reduktionsrelation schreiben wir als $s\ \to\ t$, wenn die Reduktion in einem Schritt erfolgt. Die transitive bzw. reflexiv-transitive Hülle schreiben wir als $s\ \xrightarrow{+}\ t$ bzw. $s\ \xrightarrow{*}\ t$.
Der Begriff des Redex (reducible expression) kann jetzt definiert werden: Wenn in $C[s]$ das s unmittelbar reduziert werden kann, dann ist s (zusammen mit seiner Position) ein Redex in $C[s]$.

Bei den Auswertungsregeln fällt auf, daß nicht alle Ausdrücke sinnvoll auswertbar sind. Z.B. kann man $\texttt{case}_{\texttt{Bool}}\ \mathit{Nil}\ (\mathit{True} \to t_1)(\mathit{False} \to t_2)$ nicht sinnvoll auswerten, da `Nil` nicht zum Typ `Bool` gehört. Das kann man als Typfehler zur Laufzeit ansehen.
Die so definierten Auswertungsregeln haben noch zuviele Möglicheiten der Anwendung innerhalb eines Ausdrucks. Z.B. kann man $(\lambda x\ .x)((\lambda y\ .y)z)$ an zwei Stellen reduzieren. Deshalb definiert man eine Normalordnungs-Reduktion (*normal-order-Reduktion, Standard-Reduktion*), wozu man Reduktionskontexte braucht.

Definition 1.15. *Reduktionskontexte sind:*

$$R ::= []\ |\ (R\ e)\ |\ \texttt{case}_T\ R\ (p_1 \to t_1) \ldots (p_n \to t_n)$$

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 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 $R[s] \to R[t]$ die Normalordnungsreduktion (Standardreduktion, normal order reduction) *von $R[s]$.*
Der Unterterm s zusammen mit seiner Position wird auch Normalordnungsredex *genannt. Die Reduktionsrelation wird mit dem Index* no *versehen. Analog wie oben bezeichnen wir die transitive und reflexiv-transitive Hülle mit $\xrightarrow{+}_{no}$ bzw. $\xrightarrow{*}_{no}$.*
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 $t\Uparrow$.*

Manchmal sagen wir auch, *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. $t \equiv$ (`cons` $((\lambda x.x)$ `True`$)$) `Nil`) ist selbst in WHNF, aber (`cons True Nil`) ist ebenfalls eine WHNF zu t.
Damit haben wir alle zur Auswertung von Ausdrücken notwendigen Begriffe definiert und können jetzt Ausdrücke auswerten.

Beispiel 1.17. $((\lambda x.x)\ \texttt{Nil}) \to_{no}\ \texttt{Nil}$
$((\lambda x.\lambda y.x)\ s\ t) \xrightarrow{no} ((\lambda y.s)\ t) \xrightarrow{no} s$.

Lemma 1.18. *Es gilt:*

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

Beweis. Das erkennt man an der Definition der Normalordnungsreduktion. □

Definition 1.19. *Dynamische Typregeln:*

- *Für* `case`*:*
  $(\texttt{case}_T\ e\ p_1 \dots p_m)$ *ist* direkt dynamisch ungetypt
  wenn e eine der folgenden Formen hat:
  - $(c\ a_1 \dots a_n)$ *und c ist ein Konstruktor der nicht zum Typ T gehört,*
  - $(c\ a_1 \dots a_n)$ *und c hat nicht die Stelligkeit n,*
  - $\lambda x\ .\ t$.
- *für Konstruktoren:*
  $(c\ e_1 \dots e_m)$ *ist* direkt dynamisch ungetypt *wenn c ein Konstruktor ist, und die Stelligkeit von c echt kleiner als m ist.*

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 Normalordnungsredex.

Beachte, dass Haskell eine Typisierung einführt, die viel restriktiver ist, als nur die Bedingung "es tritt kein Typfehler auf".

Wir führen den komplementären Begriff *wohlgetypt* später (auch für Haskell) ein, da es Wahlmöglichkeiten in der Definition gibt, und führen dann auch Mittel und Werkzeuge ein, um Wohlgetyptheit von Ausdrücken zu erkennen.

Der Algorithmus, der in einem Ausdruck nach dem Normalordnungsredex sucht (auch "Unwind" genannt), geht folgendermaßen vor:
Er startet mit der Wurzel eines Ausdrucks und geht in Anwendungen in den linken Term, in `case`-Ausdrücken in den zu analysierenden Term und sucht solange, bis einer der folgenden Fälle eintritt:

- Der Unterterm ist eine Variable (das kann bei Auswertung von geschlossenen Termen nicht passieren)
- Der Unterterm ist eine Abstraktion $(\lambda x\ .\ s)$.
  - Wenn der nächste Oberterm die Form $((\lambda x\ .\ s)\ s')$ hat, dann ist ein Normalordnungsredex gefunden.
  - Wenn der nächste Oberterm die Form $(\texttt{case}\ (\lambda x\ .\ s)\ (\ldots))$ hat, dann ist ein dynamischer Typfehler aufgetreten.
  - Wenn kein Oberterm existiert und kein direkt dynamischer Typfehler aufgetreten ist, dann ist der Term in WHNF.
- Der Unterterm ist von der Form $(c\ t_1 \ldots t_n)$.
  - Wenn $ar(c) < n$, dann ist das ein dynamischer Typfehler.
  - Wenn der nächste Oberterm die Form $((c\ t_1 \ldots t_n)\ s')$ hat und $ar(c) \geq n+1$, dann gehe zum nächsten Oberterm.
  - Wenn der nächste Oberterm die Form $((c\ t_1 \ldots t_n)\ s')$ hat und $ar(c) \leq n$, dann ist ein dynamischer Typfehler aufgetreten.
  - Wenn der nächste Oberterm die Form $\texttt{case}_T\ (c\ t_1 \ldots t_n)(\ldots)$ hat und $ar(c) > n$, dann ist das ein dynamischer Typfehler.
  - Wenn der nächste Oberterm die Form $\texttt{case}_T\ (c\ t_1 \ldots t_n)(\ldots)$ hat und $ar(c) = n$, dann ist ein Normalordnungsredex gefunden.
  - Wenn der nächste Oberterm die Form $\texttt{case}_T\ (c\ t_1 \ldots t_n)(\ldots)$ hat und c kein Konstruktor des Typs T ist, dann ist das ein dynamischer Typfehler.
  - Wenn kein Oberterm existiert und kein direkt dynamischer Typfehler aufgetreten ist, dann ist der Term in WHNF.

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 1.20.

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

ist ein Ausdruck, für den die Normalordnungsreduktion nicht terminiert.

Beispiel 1.21. Einige Funktionen, die auf dem Datentyp `Bool` definierbar sind, sind `and`, `or`, `not`:

$$\begin{aligned}
\texttt{and} &= \lambda x\ .\ \lambda y\ .\ \texttt{case}_{\texttt{Bool}}\ x\ (\texttt{True} \to y)\ (\texttt{False} \to \texttt{False})\\
\texttt{or} &= \lambda x\ .\ \lambda y\ .\ \texttt{case}_{\texttt{Bool}}\ x\ (\texttt{True} \to \texttt{True})\ (\texttt{False} \to y)\\
\texttt{not} &= \lambda x\ .\ \texttt{case}_{\texttt{Bool}}\ x\ (\texttt{True} \to \texttt{False})\ (\texttt{False} \to \texttt{True})
\end{aligned}$$

2 Rekursive Superkombinatoren: KFP+

Um den Übergang zwischen KFP und Haskell in kleinen und überprüfbaren Schritten zu machen, erweitern wir als erstes KFP zu KFP+, das wie KFP ein Fragment von Haskell (bzw. eine rudimentäre Haskell-syntax) ist.

Die Gründe sind:
Der Schritt von KFP nach KFP+ soll die Programmierung (in KFP) erleichtern durch Definitionsmöglichkeit von Superkombinatoren, die Lambdas implizit verwenden und bei der Reduktion das Einfangen von Variablen weitgehend vermeidet. Es ist dann auch erlaubt, rekursive Algorithmen wie in Haskell zu schreiben. In KFP+ sind Abstraktionen (d.h. lambdas) erlaubt, aber wie sich herausstellen wird, kann man in KFP+ leicht ohne explizite Abstraktionen auskommen. Ein weiterer Grund ist, dass der Zusammenhang zwischen Haskell und KFP und entsprechende Nachweise durch Konzentration auf das Wesentliche übersichtlicher werden.
In KFP+ fehlt von Haskell noch das Drumherum wie Typen, Typklassen, Typkonversion und Ein-Ausgabe.

KFP+ erlaubt zusätzlich folgendes: Es gibt Funktionsnamen (Superkombinatornamen) als neue Konstanten, und auf oberster Ebene die Möglichkeit, Funktionen zu definieren:

Definition 2.1. *Syntax von KFP+*
Die Syntax der KFP+-Ausdrücke ist wie KFP, nur um Superkombinatornamen erweitert.
Es gibt zusätzlich eine Menge von Definitionen:

$$DEF ::= \textit{Funktionsname}\ \ V_1 \ldots\ V_n = EXP$$

Folgende Bedingungen müssen eingehalten werden:

- *Die Variablen in $V_1 \ldots V_n$ sind alle verschieden; außerdem kommen im definierenden Ausdruck EXP nur diese Variablen frei vor.*

– *Die Namensräume für Variablen, Konstruktoren, Funktionsnamen sind disjunkt.*
– *Jede Funktion wird höchstens einmal definiert*

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. *Ein KFP+-*Programm *besteht aus:*

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

$$\mathtt{main} = \ EXP$$

definiert werden.

Kontexte und Reduktionskontexte in KFP+ sind analog wie in KFP definiert, nur erweitert um Superkombinatornamen in Ausdrücken.
Der Wert des Programms ist der Wert des Ausdrucks (bzw. von `main` nach Auswertung). Die Auswertung eines Superkombinatorprogramms erfolgt durch eine abgewandelte β-Reduktion und Case-Reduktion:

Definition 2.3. *Die Auswertung von Ausdrücken in KFP+ geschieht nach den Regeln:*

$$\textit{SK-Beta} \quad \frac{(\mathtt{f}\ a_1 \ldots\ a_n)}{r[a_1/x_1, \ldots, a_n/x_n]}$$
wenn `f` *definiert ist als:* $\mathtt{f}\ x_1\ \ldots\ x_n = r$

$$\textit{Case} \quad \frac{(\mathtt{case}\ (c\ t_1 \ldots t_n) \ldots (c\ x_1 \ldots x_n \to s) \ldots)}{s[t_1/x_1, \ldots, t_n/x_n]}$$

Diese Auswertung soll in allen KFP+-Kontexten erlaubt sein.
Analog zur KFP-Auswertung definieren wir die KFP+-Normalordnungsreduktion als diejenige Reduktion, die innerhalb eines KFP+-Reduktionskontexts unmittelbar mit Case oder SK-Beta reduziert.
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 2.4. *Eine KFP+-WHNF ist entweder*

– *ein Ausdruck $(c\ t_1 \ldots t_n)$ wobei $ar(c) \geq n$; oder*
– *ein Ausdruck $(f\ t_1 \ldots t_n)$, wobei $ar(f) > n$; oder*
– *ein Ausdruck $\lambda x.s$)*

Die formalen Eigenschaften und die Beziehungen zwischen KFP- und KFP+-Reduktion behandeln wir in einem extra Kapitel.

2.1 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. Dies soll dem Verständnis der Programmierung in Haskell dienen.
Die Typklassen und polymorphe Typprüfung wird erst später besprochen. In KFP oder KFP+ gibt es eigentlich keine Typen. Als passendes Fragment von Haskell könnte man etwa KFP+Typen definieren. Der Unterschied zwischen KFP und KFP+Typen ist in etwa, dass jeder Ausdruck in "KFP+Typen" einen Typ hat, aber nicht alle KFP+-Ausdrücke.
Die Syntax von polymorphen Typen ist:

$$T ::= V \mid (TC\ T_1 \ldots T_n) \mid (T_1 \to T_2)$$

wobei V eine Typvariable, T ein Typ, und TC ein Typname (Typkonstruktor) ist. Wir schreiben mit weniger Klammern $T_1 \to T_2 \to T_3$ statt $T_1 \to (T_2 \to T_3)$. Typkonstruktoren sind Namen wir `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.
Wir erläutern das Typkonzept an Beispielen. Der Typ einiger Ausdrücke, wie sie in Haskell gefunden bzw. definiert werden.

$$\begin{array}{ll} \texttt{True} & :: \texttt{Bool} \\ \texttt{False} & :: \texttt{Bool} \\ \texttt{and} & :: \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool} \\ \texttt{or} & :: \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool} \\ & \\ \texttt{Cons} & :: a \to [a] \to [a] \\ \texttt{Nil} & :: [a] \end{array}$$

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

- $\dfrac{f :: a \to b, s :: a}{(f\ s) :: b}$
- $\dfrac{s :: T}{s :: T'}$, wobei $T' = \sigma(T)$ und σ eine Einsetzung von Typen für Typvariablen ist.
- $\dfrac{s :: T, t_1 : a, \ldots, t_n :: a}{(\texttt{case}_T\ s\ (p_1 \to t_1) \ldots) :: a}$

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

Die Klammerregeln für Typen und Ausdrücke sind konsistent: Sei $f :: T_1 \to (T_2 \to T_3)$ und $a :: T_1, b :: T_2$. Dann gilt mit den Typregeln: $(f\ a) :: (T_2 \to T_3)$ und $((f\ a)\ b) :: T_3$.
Läßt man alle Klammern weg, dann ist das konsistent: $f :: T_1 \to T_2 \to T_3$ und $f\ a\ b :: T_3$.

2.2 Arithmetische Funktionen

Wir zeigen, wie arithmetische Funktionen $*, +, -$ definiert werden können, sowie nicht-negative ganze Zahlen: $0, 1, 2, 3, ...$

Beispiel 2.5. Peano-Kodierung von Zahlen mit Typ `Int`. Der Typ `Int` hat eine nullstellige Konstante 0 und einen einstelligen Konstruktor `S`. Dann kodieren $0, (\texttt{S}\ 0), \texttt{S}(\texttt{S}\ 0), \texttt{S}(\texttt{S}(\texttt{S}\ 0)), \ldots$ die nicht-negativen ganzen Zahlen $0, 1, 2, 3, \ldots$. Die Typen sind: $0 :: \texttt{Int}$, $\texttt{S} :: \texttt{Int} \to \texttt{Int}$.
Damit sind arithmetische Funktionen auf allen nicht-negativen ganzen Zahlen in KFP+ definierbar:

$$\begin{array}{ll}
bot & = bot \\
\texttt{istZahl}\ x & = \texttt{case}_{Int}\ x\ (0 \to \texttt{True})\ ((\texttt{S}\ y) \to (\texttt{istZahl}\ y)) \\
+\ x\ y & = \texttt{if}\ (\texttt{and}(\texttt{istZahl}\ x)(\texttt{istZahl}\ y)) \\
 & \quad \texttt{then}\ (\texttt{plus}\ x\ y)\ \texttt{else}\ bot \\
\texttt{plus}\ x\ y & = \texttt{case}_{Int}\ x\ (0 \to y)\ ((\texttt{S}\ z) \to (\texttt{S}\ (\texttt{plus}\ z\ y)))
\end{array}$$

Der Typ der Funktionen ist:

- $\texttt{istZahl} :: \texttt{Int} \to \texttt{Bool}$
- $+ :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}$
- $\texttt{plus} :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}$

Die Funktion `istZahl` dient dem Zweck, die Eingabe vorher zu prüfen und sicherzustellen, dass eine Zahl eingegeben wurde. Ein anderer Zweck ist es, zu erreichen, dass sich die hier definierten Funktionen wie arithmetische Funktionen verhalten. Z.B. soll sowohl $(+\ bot\ 1)$ als auch $(+\ 1\ bot)$ 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 z.B. `(S bot)` oder `unendlich = (S unendlich)`.

Übungsaufgabe 2.6. Übungsaufgabe: Definiere Multiplikation $*$ auf dem Datentyp `Int`.

Beispiel 2.7. Die Funktion (das Prädikat) $\leq$ zum Vergleichen von Zahlen kann man definieren als:

```
leq x y =
   and (istZahl x)
       (and (istZahl y)
             (case_I x
               (0 -> True)
               (S xx -> case_I y (0 -> False) ((S yy) -> leq xx yy))))
```

Im folgenden benutzen wir statt der etwas umständlichen Methode mit `S` und 0 die übliche Schreibweise mit Zahlen als Konstanten und arithmetischen Funktionen. Beachte, dass Implementierungen, die einen beschränkten Zahlbereich verwenden, nicht 100% kompatibel zu den Peanozahlen sind.

Wir erlauben uns auch ab jetzt, der besseren Lesbarkeit halber und wegen der in Haskell erlaubten Schreibweise, die arithmetischen Funktionen in Infix-Schreibweise zu benutzen.

Beispiel 2.8. Die Fakultätsfunktion mit `fak 0 = 1, fak 1 = 1, fak n = 1*2* ... n` auf nichtnegativen Zahlen, mit Superkombinatoren, rekursiv definiert:

$$\texttt{fak}\ x = \texttt{case}_{\texttt{Int}}\ x\ (0 \to 1)\ ((\texttt{S}\ y) \to (x * (\texttt{fak}\ y)))$$

Der Typ ist: `fak` :: `Int` $\to$ `Int`. Wir zeigen eine Auswertung der Funktion `fak` (Fakultät), wobei die Auswertung auch einige Schritte abkürzt und auch Schritte macht, die nicht in Normalordnung sind.

$$\begin{aligned}
\texttt{fak}\ (1+1) &\to (\texttt{case}_{\texttt{Int}}\ (1+1)\ (0 \to 1)((\texttt{S}\ y) \to ((1+1) * (\texttt{fak}\ y)))) \\
&\to (\texttt{case}_{\texttt{Int}}\ 2\ (0 \to 1)((\texttt{S}\ y) \to ((1+1) * (\texttt{fak}\ y)))) \\
&\to ((1+1) * (\texttt{fak}\ 1)) \\
&\to (2 * (\texttt{fak}\ 1)) \\
&\to (2 * (\texttt{case}_{\texttt{Int}} 1 (0 \to 1)\ ((\texttt{S}\ y) \to (1 * (\texttt{fak}\ y))))) \\
&\to (2 * (1 * (\texttt{fak}\ 0))) \\
&\to (2 * (1 * (\texttt{case}_{\texttt{Int}}\ 0\ (0 \to 1)\ ((\texttt{S}\ y) \to (0 * (\texttt{fak}\ y)))))) \\
&\to (2 * (1 * 1)) \\
&\to (2 * 1) \\
&\to 2
\end{aligned}$$

Übungsaufgabe 2.9. Welche Schritte werden in einer Normalordnungsreduktion nicht gemacht, wenn man die Implementierung der arithmetischen Funktionen wie oben mit Peano-Zahlen gemacht hat?

Beispiel 2.10. Ein Prädikat `schaltjahr` kann man unter Verwendung des Infix-operator `mod`, der die Reste bei Division als Resultat hat, ganz einfach definieren als:

```
schaltjahr x = and (x mod 4 == 0)
                   (or (x mod 100 \= 0)  (x mod 400 == 0))
```

Haskell-code:

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

Beispiel 2.11. Die Funktion, die Zahlen quadriert sei definiert als:

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

Beispiel 2.12. Die *Komposition* von Funktionen kann man definieren als:

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

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

Beispiel 2.13. Zu getypten und ungetypt arithmetischen Ausdrücken:

1. (2 5) ist ungetypt, da die Stelligkeitsregel für `S` verletzt ist.
2. (+ 3 *quadrat*) ist nicht direkt ungetypt, aber der Test (*istZahl quadrat*), der sich nach einigen Reduktionen ergibt, ist dynamisch ungetypt.

Aber: $(*\ 6)$ ist kein dynamischer Typfehler, denn $((*\ 6)\ 2)$ ist äquivalent zu $(6*2)$, was zu 12 reduziert. Die Funktion $(*\ 6)$ ist die Funktion die ihr Argument mit 6 multipliziert.

2.3 Tupel, Paare

Die Syntax von Haskell erlaubt Tupel einfach als eingeklammerte Objekte zu schreiben: Ein 5-Tupel kann man schreiben als $(1, 2, 3, 4, 5)$.
Die Kernsprachendeutung ist: es gibt einen Typ `Tupel5` mit einem Konstruktor `Tupel5`, der Stelligkeit 5 hat. Das heißt, pro Länge eines Tupels gibt es einen eigenen Typ.
Der polymorphe Typ eines Tupelkonstruktors ist z.B.

$$\texttt{Tupel5} :: a_1 \to a_2 \to a_3 \to a_4 \to a_5 \to (a_1, a_2, a_3, a_4, a_5)$$

In Haskell gibt es ein null-stelliges Tupel, geschrieben (). In KFP ist das einfach ein Typ `NullTupel` mit einem einzigen Konstruktor `NullTupel`.

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

Beispiel 2.14.

```
fst x = case_Paar x ((Tupel2 u v) -> u)
snd x = case_Paar x ((Tupel2 u v) -> v)
```

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

```
paar  fg x   = case_Paar fg ((Tupel2 f g) -> (Tupel2 (f x) (g x)))
cross fg xy  = case_Paar fg
                 ((Tupel2 f g) -> (Tupel2 (f (fst xy)) (g (snd xy))))
```

Damit kann man z.B. (in Haskell-Schreibweise) berechnen:

- `paar (quadrat,id) 2` $\to (4, 2)$.
- `cross (quadrat,id) (2,3)` $\to (4, 3)$

Beispiel 2.15. Eine Variante von `cross`, die scheinbar das gleiche wie `cross` tut ist:

```
cross'  fg xy = case_Paar fg ((Tupel2 f g) ->
                 case_Paar xy ((Paar x y) -> (Tupel2 (f x) (g y))))
```

Vergleicht man aber `cross (K 1,K 2) bot` und `cross' (K 1,K 2) bot`, wobei `K x y = x`, dann sieht man dass das erste zu $(1, 2)$ reduziert, während das zweite undefiniert ist, bzw. $= bot$ ist. Man sagt auch, die Definition von `cross` ist verzögerter als die von `cross'`. Im allgemeinen ist die Variante `cross` vorzuziehen, da diese Funktion auf mehr Werten definiert ist. Der kleine Vorteil, der hier etwas weit hergeholt scheint, kann sich bei grösseren Programmen (z.B. beim Speicherbedarf) vorteilhaft auswirken, wenn sehr große Datenstrukturen zwar spezifiziert sind, aber nur die gerade benötigten Teile berechnet werden.

3 Listen: Datenstruktur und Operationen

In diesem Kapitel werden Listenoperationen definiert und erklärt, wobei wir zunächst KFP+ als Programmiersprache verwenden. 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 versuchen einen fließenden Übergang von KFP+ nach Haskell mit eingeschobenen Begründungen. Die exakten Eigenschaften der operationale Semantik bzw. die Transformation nach KFP+ und andere formale Begründungen werden später in der Vorlesung behandelt.

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`. Im folgenden schreiben wir $\texttt{case}_{\texttt{List}}$ ohne den Index `List`.

Wir geben auch die Haskell-Definition der Funktionen an. Der Unterschied ist, dass es in Haskell möglich ist, 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 Definitionen 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 KFP+ wird durch Beispiele erläutert.

3.1 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. die nicht terminiert. Wir nennen solche Objekte auch "unendliche Listen".
Die Definitionen sind:

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

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

Die etwas bessere Definition von `from` ist:
`from m  =  m : (strict from (m+1))`.

3.2 map, ++, 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 (Nil -> []) (((f h):t) ->  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` $:: (a \to b) \to [a] \to [b]$.

Als Nebenbemerkung geben wir an, wie ein Haskell-Pattern am sinnvollsten nach KFP zu übersetzen ist. 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 allgemeinen sind die Pattern sequentiell gemeint. D.h. die Pattern werden von oben nach unten abgearbeitet. Auch dies ist leicht nach KFP zu übersetzen, wenn die Typen bekannt sind.

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

```
++ xs ys = case xs (Nil -> ys) ((h:t) -> h:(t ++ ys ))
```

Sie hängt zwei eingegebene Listen aneinander: $[1,2,3]$ $++$ $[4,5,6]$ $\rightarrow$ $[1,2,3,4,5,6]$. 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 (Nil -> 0) ((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` $:: [a] \rightarrow Int$
Diese Definition der Länge ist eine Abbildung in die Peanozahlen. Man sieht hier einen Unterschied zwischen Peanozahlen und lazy Peanozahlen. Die normale Definition funktioniert nur für endliche Listen. Z.B. `length xs > 0` Eine Definition für lazy Peanozahlen könnte auch solche Ausdrücke noch sinnvoll auswertbar machen.

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 (Nil -> Nil) ((h:t) -> ((reverse t) ++ [h]))
```

Z.B. `reverse` $[1,2,3] \to [3,2,1]$
Die Haskell-Definition ist:

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

Der Typ ist: `reverse` $:: [a] \to [a]$.
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 ([] -> 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:

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

```
hd xs          =  case xs ((h:t) -> h) ([] -> bot)
tl xs          =  case xs ((h:t) -> t) ([] -> bot)

transpose xss  =  case xss
                    ([] -> bot)
                    ((h:t) ->
                       (case h ([] -> [])
                               ((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` :: $[a] \to a$
`tl` :: $[a] \to [a]$
`transpose` :: $[[a]] \to [[a]]$

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

$$\begin{array}{ll} & [1,3,5] : \texttt{transpose}\ [[2],[4],[6]] \\ \to & [[1,3,5],[2,4,6]] \end{array}$$

Von jetzt an geben wir die Funktionen nur noch in Haskell-Notation an. Es sollte für den Leser jetzt kein Problem mehr sein, zwischen KFP+ 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` :: $[Int] \to [Int] \to [Int]$.
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)
vectoradd_2             =  map2 (+)
```

Der Typ ist: `map2` :: $(a \to b) \to [a] \to [a] \to [b]$.
! Argumente kann man weglassen, da die Funktionen curried angewendet werden.

Übungsaufgabe 3.1. Was macht die Funktion `map2` mit verschieden langen Listen?

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

$$\texttt{let}\ x_1 \ = \ t_1; \ldots; x_n \ = \ t_n\ \texttt{in}\ EXP$$

bzw.

$$EXP\ \texttt{where}\ x_1 \ = \ t_1; \ldots; x_n \ = \ t_n$$

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 ist das let rekursiv. D.h. der Bindungsbereich der x_i ist $t_1, \ldots, t_n, EXP$.
Ein nicht-rekursives `let (where)` könnte man als Lambda-Ausdruck darstellen:

```
let x_1 = t_1;....  ; x_n = t_n~ in EXP
```

entspricht dann (in etwa)

```
(\ x_1 ....   x_n . EXP) t_1 ...  t_n
```

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

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` :: $(a \to$ `Bool`$) \to [a] \to [a]$
Damit kann man eine Funktion definieren, die alle Elemente entfernt, die das Prädikat *pred* erfüllen:

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

In der Definition von `remove` ist das Listenargument nicht notwendig.

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) xs) ++
                     (h :  (qs (filter (> h) xs)))
```

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 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. 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`:

```
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]` $\to ((((0+1)+2)+3)+4)$. D.h. die Summation fängt vorne an.

Übungsaufgabe 3.2. Eine Aufgabe zum Unterschied in der Wirkungsweise:

```
concatr =   foldr (++) []
concatl =   foldl (++) []
```

Wie hoch ist Platz und Speicherbedarf (geschätzt) von

`hd` (`concatr` *lst*)

und

`hd` (`concatl` *lst*)

für verschiedene Listen von Listen?

Die Funktion `foldl` kann ihre Stärke in bestimmten Anwendungen wie z.B. für die Summation von Listen von Zahlen erst voll ausspielen, wenn sie noch leicht abgeändert wird:

> `foldl` sollte strikt im zweiten Argument sein. !!!

D.h. `foldl` sollte zuerst sein zweites Argument zur WHNF auswerten, und dann mit der Normalordnung weitermachen. Das ist eine Veränderung der Normalordnung.
Wenn `foldl` strikt im zweiten Argument ist, dann 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.
Die Funktion `strict`, angewendet auf eine Funktion f, ergibt als (`strict` f) eine Funktion, die strikt ist in ihrem ersten Argument. D.h. sie wertet zuallererst ihr erstes Argument zu WHNF aus.
Als Hinweis zur operationalen Semantik von `strict`: Der Operator `strict` läßt sich nicht in allgemeiner Form in KFP darstellen bzw. simulieren. Ist der Typ von $(f\ x)$ ein algebraischer Datentyp, dann läßt sich dies auch in KFP implementieren durch eine Transformation. I.a. ist ein Änderung der Normalordnungsreduktion erforderlich. Diese Version hat in vielen Fällen diesselbe Semantik wie `foldl`, wie z.B. bei der Summation über Listen von Zahlen, aber nicht immer. 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:

$$\texttt{foldr} :: (a \to b \to b) \to b \to [a] \to b$$
$$\texttt{foldl} :: (a \to b \to a) \to a \to [b] \to a$$

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

$$(a \to a \to a) \to a \to [a] \to a$$

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 Funktioen.

```
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]
=    [1,2,3,6]
```

Beispiel 3.3. Will man die Teilsummen zur Berechnung von 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
```

4 Haskell und Gofer

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-Ausdrücke (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 compiliert nach C.
- Haskell-B-Compiler (HBC) von Chalmers:
- Es gibt eine abgespeckte Variante: Gofer, vom Mark P. Jones, das nur im Typklassensystem ausgefeilter als Haskell ist.
- Die angepasste Variante ist HUGS: (Haskell Users Gofer System)

In diesem Kapitel werden einige syntaktischen Konstrukte und das Typklassensystem von Gofer/Haskell beschrieben. Die Semantik wird durch Transformation 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 höhersprachlichen Konstrukte in eine Kernsprache übersetzt und sie erst danach in eine weitere Zwischensprache für eine abstrakte Maschine transformiert.

4.1 Syntax

Gofer/Haskell und (teilweise Miranda) wird in einem kurzen, nicht vollständigen Überblick dargestellt. Für genauere Informationen ist das entsprechende Handbuch bzw. Haskellreport zu konsultieren.

Zahlen, Character sind vorhanden, wobei es verschiedene Zahlentypen gibt: beschränkte ganez Zahlen (4 byte), unbeschränkte ganze Zahlen, Gleitkommazahlen, und auch rationale Zahlen.
Listen werden abgekürzt als $[1, 2, 3]$. Dies steht für $1 : 2 : 3 : \texttt{Nil}$, dies wiederum für $1 : (2 : (3 : \texttt{Nil})))$. “:“ 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 $1 : 2 : 3 : nil \equiv 1 : (2 : (3 :$ `Nil`)))
z.B. `FAC` $2 +$ `FAC` $3 \equiv$ (`FAC` 2) + (`FAC` 3), wenn `FAC` höhere Priorität als + hat.
Beispiele der Deklarationen der Operatoren:

```
infixl  7  *
infix   7  /,'div',
infixl  6  +,-
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.
Die Anwendung ist wie üblich
$F\ G$ bedeutet: $(F\ G)$: F angewendet auf G.
$F\ G\ H$ klammert sich $(F\ G)\ H$

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 Benutzer angegeben werden.

Überladung: Dasselbe Symbol wird für verschiedene Operationen benutzt, z.B. + für reelle bzw integer-Addition. Bei Gofer 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 Markierungen offensichtlich.

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 Gofer und 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. Es gibt ein if-then-else: (`if` *cond* `then` *expr* `else` *expr*) und ein `case` Konstrukt. `let` wird in der Form (`let` $x = a$ `in` *expr*) verwendet. `where` ist eine zu let

äquivalente Syntax: ($expr$ **where** $x = a$) ist äquivalent zu obigem (`let` $x = a$ `in` $expr$).
Es gibt in manchen FPS Unterschiede in der Interpretation des `let, where`: Es wird je nach FPS teilweise als rekursives oder auch als nicht-rekursives let interpretiert.
Funktionsdefinitionen werden i.a. als eine oder mehrere Gleichungen hingeschrieben, wobei an den Argumentpositionen Patterns benutzt werden. Die Unterscheidung 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 `Text`) auf Strings abbildet. d.h. `show` : $a \to$ `String`.

Tupel Sind algebraische Datentypen mit genau einem Konstruktor. Für jede Stelligkeit wird ein eigener Konstruktor benötigt der zu einem eigenen Datentyp gehört. $tupel_n : a_1 \to \ldots \to a_n \to tupel_n(a_1, \ldots, a_n)$ (Hier wird der Name $tupel_n$ als Daten- und Typ-konstruktorname verwendet, was aber üblich ist)
Tupel sind syntaktisch eingebaut, der Konstruktor ist nur als Klammerung sichtbar: $(1,'a')$ ist ein Tupel vom Typ (`Int`, `Char`). Es gibt kein 1-Tupel, aber manchmal ein Null-tupel, oder einen Unit-type
Tupel sind im Gegensatz zu Listen heterogen, müssen dafür aber feste Länge haben. Wegen des Typsystems kann man keine Funktion schreiben, die für alle Tupel funktioniert, und sich die Elemente explizit anschaut, z.B. eine Funktion, die aus jedem Tupel das erste Element extrahiert, unabhängig von der Länge des Tupels, ist nicht typisierbar.
Die Funktion, die dies mit einem Paar macht, ist einfach:

```
fst :: (a,b)  ->  a              |     snd: (a,b)  ->  b
fst (x,y) = x                    |     snd (x,y) = y
```

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

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 Konstruktorkonstanten, auch, $1, 2, 3, \ldots$) 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)`.

Zum Beispiel ist `wurzel` (`quadrat` x) $= x$ nicht möglich, da `quadrat` kein Konstruktor ist.
In Haskell / Gofer gibt es auch ein Lambda-Konstrukt: $\lambda x.exp$ wird dort geschrieben als $\backslash x \rightarrow exp$

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`).

4.2 Semantik

Die Zurückführung auf die Basissprache KFP ist bei fast 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 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 und Gofer verarbeiten 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 Gofer 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.
– Eine erzwungene Auswertung mittels der Funktion `strict`: (`strict` f) x wertet zunächst x bis WHNF aus, dann erst $(f\ x)$. Dies kann nur dann nach KFP übersetzt werden, wenn das nur eine Umstellung der Normalordnung ist, oder der Typ des Ausdrucks festgelegt ist und Instanz eines algebraischer Datentyp ist (dann als `case`). Siehe dazu die Bemerkungen zu KFP+strict.

Durch diese Transformationsregeln werden folgende Eigenschaften von KFP übertragen:

1. Die Semantik der Funktionen und Konstrukte in Haskell wird festgelegt.
2. Die Gültigkeit der kontextuellen Gleichheit und der Korrektheit von Programmtransformationen gilt auch für Haskell

List Comprehensions List Comprehensions werden auch ZF-Expressions genannt (Zermelo & Fränkel). Dies sind einfachere syntaktischere Schreibweisen für Listenkonstruktionen, die an die Mengenschreibweise wie $\{x * x \mid x > 0\}$ angelehnt sind.
Z.B. kann man `map` und `filter` damit folgendermaßen schreiben:

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

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

Die Schreibweise $\{exp \mid \ldots\}$ 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.

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

Neue Variablen werden durch Generatoren eröffnet, der Geltungsbereich liegt in allen Ausdrücken rechts des Generatorausdrucks und in dem Kopfausdruck. D.h. x aus $x \leftarrow [1..6]$ hat einen Geltungsbereich: $(x,y), x+y < 10$. Die Scopusregeln gelten wie üblich. Wenn man Variablen doppelt verwendet, erzeugt man einen neuen Geltungsbereich.
Allgemeiner kann in den Generatoren links auch ein Pattern mit neuen Variablen stehen.

Beispiel 4.1.

```
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 []`

Im Falle dass Q leer ist, schreibt man $[e]$ für $[e \mid Q]$.
Damit kann man einfache Fälle optimieren:
`[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)`.

Beispiel 4.2. 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 Hugs oder Gofer 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 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 / Gofer 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]`:

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

In Gofer ist es auch möglich, lokale Definitionen in List-Comprehensions zu verwenden. Diese sind von der Form $pat = exp$ und werden in die Generatoren und Prädikate eingereiht. In pat können neue Variablen definiert werden, die Skopusregeln gelten für diese genauso wie für Generatoren.

4.3 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. Diese 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* $(3+2i)$. Ein Nachteil des Überladungsverbots ist die Vermehrung von Namen allgemeiner Funktionen, z.B. Summe einer Liste müßte für 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 Tyklassen.

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

4.4 Typklasse

Eine Typklasse kann man sich vorstellen als eine Menge von Typen zusammen mit zugehörigen Funktionen bzw. Methoden. Die Typklassen werden als eigene syntaktische Einheiten (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. Im allgemeinen gibt es nur eine Variable a.
Die Bedingung in Haskell ist nur eine Unterklassenbeziehung von Typklassen:

```
class (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`. Leider ist dies von der logischen Schreibweise her falsch geraten.
Notwendig ist eine extra Syntax zur Deklaration von Typklassen und zur Deklaration von Typen von Funktionen.

Beispiel 4.3. Der Typ von `==` ist

```
Eq(a) =>  a -> a -> Bool
```

Das bedeutet, daß für alle `a` die in `Eq` sind, die Funktion den Typ `a -> a -> Bool` hat.

Bei der aktuellen Verwendung von `==` in einem Ausdruck wird der Typechecker dann prüfen, ob die getesteten Objekte tatsächlich in der Klasse `Eq` sind.
In `Eq` sind alle Typen, bei denen ein Gleichheitstest implementiert werden kann. Nicht sinnvoll ist der Vergleich von Funktionen mit `==`, obwohl man das auch definieren kann.

Beispiel 4.4. Definition von `Eq`:

```
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.
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 (sogenannte **`deriving`**), der die Strukturgleichheit implementiert, wobei rekursiv wieder == verwendet wird.

Beispiel 4.5.
```
instance Eq Char      where
          x == y    =   ord x == ord 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 4.6.

```
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:

```
 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 Konstruktor-möglichkeiten definiert
– In Haskell ist dies auch möglich, allerdings wird dort eine default Definition benutzt, bei der direkt an der Typdeklaration steht: `deriving (Eq,Ord)`. Diese Möglichkeit gibt es auch in Hugs, aber nicht in Gofer.

4.5 Die Klasse Ord

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

Beispiel 4.7.

```
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 -> Bbool
    max, min              :: a -> a -> a

    compare x y | x==y      = EQ
                | x<=y      = LT
                | otherwise = GT

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

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

Die primitive Operation ist offensichtlich $<=$, die anderen sind darauf aufgebaut.

4.6 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. $Read(Show(x)) = x$.
Funktion (Methode), die auf solchen Typen (α) immer vorhanden ist:

`showS`: $\alpha \rightarrow$ string
`readS`: string $\rightarrow \alpha$

Diese ist für einige eingebaute Typen bereits vordefiniert und muß für andere entsprechend definiert werden.

4.7 Mehrfache Vererbung

Es gibt auch etwas analoges zur multiplen Vererbung:

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

    x - y           = x + negate y
    fromInt         = fromIntegral

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

4.8 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 sichergestellt, daß eine Anwendung von Methoden auf primitive Objekte mit der richtigen primitiven Methode erfolgt.
!!! Im Haskell/Gofer -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 manchmal merkwürdige Konsequenzen: z.B. läßt sich [] == [] nicht immer typen, da ein [] von Typ $list(a)$, das andere von Typ $list(b)$ ist, und für a und b der (Typklassen-)Kontext nicht bestimmt werden kann.
Im Vergleich dazu existieren im Java Laufzeitsystem noch die Typinformationen und können auch in Abfragen verwendet werden.
Die interne Überladungsauflösung funktioniert so, daß überladene Funktionen intern einen zusätzlichen Parameter, den sogenannten `Dictionary`-Parameter bekommen. Zum Aufrufzeitpunkt einer Funktion muss dieser Parameter dann so bekannt sein, daß die zugehörige Implementierung im dictionary gefunden werden kann. Außerdem wird intern 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. Diese Methode entspricht in etwa dem *dynamischen Binden* in objektorientierten PS. Wenn zur Compilezeit der Typ eines Objektes ein Grundtyp ist, also vollständig bekannt, dann kann die exakte Methode eingesetzt werden. Diese Optimierung ist bei objektorientierten PS als *statisches Binden* bekannt.
Ein Dictionary kann auch rekursiv aufgebaut sein, wobei jedoch die Bedingung der Endlichkeit eingehalten werden muß.

4.9 Verwendung der Definition der Gleichheit

Beispiel 4.8. **Mengen**

```
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
```

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.

4.10 Aufzählungsklasse `Enum`

Damit kann man die Datentypen und erfassen, deren Elemente sich (vorwärts und rückwärts) aufzählen lassen, wie nichtnegative ganze Zahlen, die Zeichen eines Alphabets, oder Farben, falls man diese in eine Ordnung bringt. Für Elemente dieser Typen kann man dann ein Aufzählungsverfahren analog zu Zahlen verwenden: $[2..], [a..z], \ldots$.

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

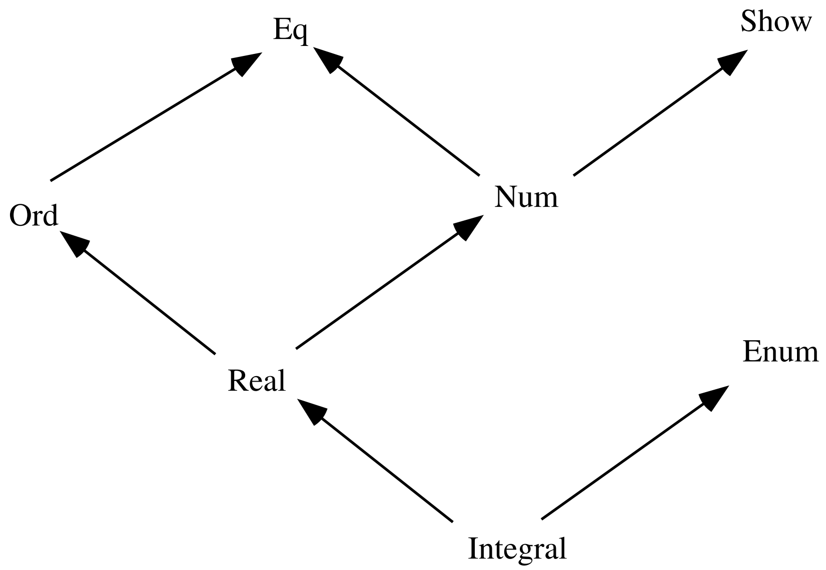
```
class Enum a where
    toEnum              :: Int -> a
    fromEnum            :: a -> Int
    enumFrom            :: a -> [a]              -- [n..]
    enumFromThen        :: a -> a -> [a]         -- [n,m..]
    enumFromTo          :: a -> a -> [a]         -- [n..m]
    enumFromThenTo      :: a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



Beispiel 4.9. Definition einer überladenen Konstanten.

```
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 Typkontext 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 4.10. Allgemeine Summe über alle Listen von Typen, die zur Typklasse `Num` gehören.
Der folgende Versuch scheitert:

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

Der Grund ist, daß die Zahl 0 den festen Typ *Int* hat.
Die allgemeine Summe über eine Liste erhält man folgendermaßen (noch effizienter mit Akkumulator):

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

Dies löst das "Problem" der überladenen Konstanten.

Ein Problem mit Verallgemeinerungen ist: Läßt man mehr Variablen in der Bedingung zu, dann ist unklar, ob es einen entscheidbaren Typcheckalgorithmus für Typen und Typklassen gibt.
Möglicherweise ist ein Typsystem einfacher zu handhaben, das abhängige Typen (dependent types) verwendet. Dies bedeutet: ein sehr ausdrucksstarkes allerdings unentscheidbares Typsystem.

4.11 Klasse Functor

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

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

instance Functor Maybe where
    map f Nothing  = Nothing
    map f (Just x) = Just (f x)
```

```
instance Functor [] where
    map f []     = []
    map f (x:xs) = f x : map f xs
```

Sinnvoll ist es, auch die 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
    map = baummap


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