Diplomarbeit

Implementierung eines parametrisch polymorphen Typherleitungssystems für eine funktionale Kernsprache mit rekursivem let und Nichtdeterminismus

Igor Geier

eingereicht bei

Prof. Dr. Manfred Schmidt-Schauß,
Professur für künstliche Intelligenz und Softwaretechnologie,

Institut für Informatik,
Fachbereich Informatik und Mathematik,
Goethe-Universität Frankfurt am Main

Inhaltsverzeichnis

Danksagung

Den Mitarbeitern der Professur KIST verdanke ich ein großes Maß an gewonnener Erkenntnis und nicht zuletzt auch das Glück, ein interessantes und aktuelles Thema bearbeiten zu dürfen. Mein besonderer Dank gilt Prof. Dr. Manfred Schmidt-Schauß und Dr. David Sabel für ihre hervorragende und vor allem kompetente Betreuung.

Erklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Igor Geier, Frankfurt, 14.07.2009

Kapitel 1

Einleitung

1.1 Zusammenfassung

Die meisten gängigen Programmiersprachen besitzen ein Typsystem. Ein Typ kann als eine Menge von Werten gesehen werden. Ein (strenges) Typsystem sorgt dafür, dass eine Operation nur auf Werte angewendet wird, die einen dafür geeigneten Typ haben (einen Typ, mit dem die Operation umgehen kann). In manchen Programmiersprachen ist es so, dass der Typ einer Variablen oder einer Prozedur/Funktion/Methode explizit angegeben werden muss, in anderen wiederum muss der Programmierer in den meisten Fällen keine *Typannotationen* verwenden, so dass der Compiler/Interpreter alle (oder die meisten) Typen herleitet. Man unterscheidet zwischen *statischer* (Typcheck vor der Ausführung) und *dynamischer* (Typcheck während der Ausführung) Typisierung und zwischen *starker* (zur Laufzeit sind keine Typfehler erlaubt) und *schwacher* (Programme können dynamische Typfehler erzeugen) Typisierung.

Die so genannte *kontextuelle Äquivalenz* ist eine Theorie, bei der es darum geht, funktionale Programmausdrücke auf Gleichheit zu untersuchen (so dass Ausdrücke durch andere ersetzt werden können, ohne dass sich das Gesamtverhalten von Programmen ändert). In [1] wird die kontextuelle Äquivalenz von getypten Programmausdrücken behandelt.

In dieser Arbeit wird ein iteratives Typherleitungs-Verfahren vorgestellt und implementiert, das Ausdrücke in der Form liefert, wie sie in [1] verwendet werden. Das Verfahren transformiert einen Quelltext einer einfachen funktionalen Programmiersprache (Kernsprache) in einen Quelltext, in dem die meisten Unterausdrücke mit gültigen Typen markiert sind (das soll heißen, dass manche Teilausdrücke nicht markiert werden, nicht dass manche Typmarkierungen falsch sind) – das Besondere ist, dass die Ausgabe nicht nur der Typ des Gesamtausdrucks ist. Falls ein Typfehler entdeckt werden konnte, wird die Eingabe als nicht typisierbar zurückgewiesen. Manchmal kann weder ein Typ hergeleitet werden, noch ein Fehler gefunden werden. In solchen Fällen terminiert das Verfahren trotzdem und der Algorithmus liefert „?“ zurück. In Tabelle 1.1.1 sind einige

Tabelle 1.1.1: Beispiele

Eingabe	Ausgabe
$\lambda x.x$	$(\lambda x :: a.x :: a) :: a \to a$
$(\lambda x.x)\ []$	$((\lambda x :: [a].x :: [a]) :: [a] \to [a]\ [] :: [a]) :: [a]$
$\lambda f.f\ f$	Fehler
`letrec` $a = b : [],\ b = a : []$ `in` a	?
`letrec` $id = \lambda x.x$ `in` id	(`letrec` $id :: \forall b.b \to b =$ $(\lambda x :: a.x :: a) :: \forall b.b \to b$ `in` $id :: c \to c$ $) :: c \to c$

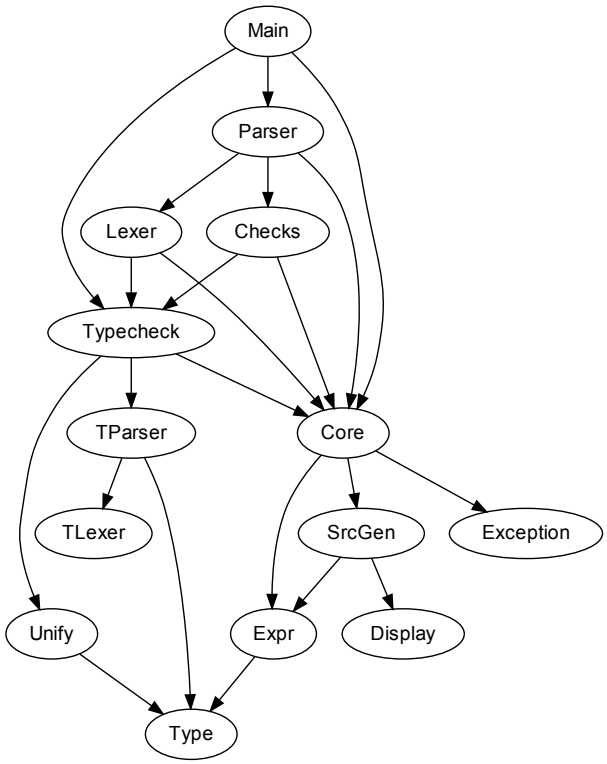
Beispiel-Ausdrücke und die zugehörigen Ausgaben dargestellt (die Syntax wird in späteren Abschnitten besprochen). Ein Quelltext kann natürlich auch Fehler enthalten, die keine Typfehler sind (Syntaxfehler zum Beispiel), außerdem muss ein Quelltext auf geeignete Weise eingelesen werden, bevor er weiterverarbeitet wird. Diese Probleme werden in Kapitel 3 näher betrachtet. Der Schwerpunkt liegt bei Kapitel 4. Dort wird fast ausschließlich die Typherleitung behandelt. Für die Implementierung wurde die funktionale Programmiersprache Haskell verwendet. Eine kurze Haskell-Einführung findet sich in Kapitel 2. Die Implementierung besteht aus mehreren Modulen, die sich in verschiedenen Verzeichnissen befinden.

```
+- bin
+- src
|  +- core
|  +- parser
|  +- types
+- tests
```

Das Unterverzeichnis `src` enthält den Hauptteil der Implementierung. Wie die Module in `src` voneinander abhängen ist in Abbildung 1.1.1 dargestellt. Es gibt eine make-Datei. Mit einem make-Programm kann ein Kommandozeilen-Programm namens `lplctc` erzeugt werden, das Typen berechnet und Ausdrücke mit eingefügten Typmarkierungen erzeugt. Zusätzlich wird der Glasgow Haskell Compiler[1] benötigt.

[1] http://www.haskell.org/ghc

Abbildung 1.1.1: Modul-Abhängigkeiten



1.2 Funktional und imperativ

Programmiersprachen lassen sich als imperative oder deklarative klassifizieren. Zu den letzteren gehören auch die funktionalen Programmiersprachen. In der puren funktionalen Programmierung ist das Prinzip der *referentiellen Transparenz* dafür verantwortlich, dass eine Funktion, aufgerufen mit den gleichen Argumenten, immer das gleiche Ergebnis zurück liefert. Ausdrücke bezeichnen Werte. Variablen bezeichnen Werte (Variablen bezeichnen Speicherplätze in imperativen Sprachen). Im Programmkontext können Ausdrücke durch andere Ausdrücke mit gleichem Wert ausgetauscht werden, wobei der Wert des Gesamtausdrucks sich dadurch nicht ändert (*Substitutionsprinzip*). Das alles ist bei imperativen Programmiersprachen i. A. nicht gegeben.

Die Ausführung eines funktionalen Programms entspricht einem Auswertungsprozess, bei dem Argumente an Funktionen übergeben werden. Ausdrücke können ausgewertet werden, indem Regeln der *Ersetzung* und der *Vereinfachung* angewendet werden. Die

einfachste äquivalente Form eines Ausdrucks ist ein Wert (eigentlich die Repräsentation eines Werts, nicht der Wert selbst, weil man nur mit Repräsentationen umgehen kann – im Rechner sind dies die binären Zahlen). Ein Ausdruck ist in kanonischer Form (oder *Normalform*), wenn er nicht weiter reduziert werden kann. Es gibt auch Ausdrücke, die keine kanonische Repräsentation besitzen (oder keine endliche), wie etwa die Zahl π. Einige Ausdrücke können überhaupt nicht ausgewertet werden, so zum Beispiel $1/0$. Man sagt auch, dass der Ausdruck nicht wohldefiniert ist. In funktionalen Programmiersprachen wird zwischen *strikter* und *nicht strikter* Auswertung unterschieden. Im ersten Fall werden alle Parameter vor einem Funktionsaufruf ausgewertet, in nicht strikten FPS werden Ausdrücke erst bei Bedarf ausgewertet (eine solche Sprache ist Haskell).

Die Menge aller Werte kann partitioniert werden, indem Typen eingeführt werden. Es gibt elementare Typen, wie Zahlen oder Zeichen, und es gibt zusammengesetzte Typen, wie (`Int`,`Char`) – ein Tupel, in dem das erste Element eine Zahl und das zweite ein Zeichen ist. Genauso wie der Wert eines Ausdrucks allein von den Werten der Teilausdrücke abhängt, hängt auch der Typ eines Ausdrucks allein von den Typen seiner Teilausdrücke ab. Das nennt man auch *strenge Typisierung*.

Die Basis für alle funktionalen Programmiersprachen ist der Lambda-Kalkül[2]. Diese formale Sprache wurde 1932 von Alonzo Church eingeführt und war ursprünglich als Hilfsmittel dafür gedacht, eine Grundlage der Mathematik zu finden. Der Versuch scheiterte, genau wie Gottlob Freges Versuch [12], die Mathematik auf die Logik zurückzuführen.

Der (typfreie) Lambda-Kalkül besteht aus Variablen, *Abstraktionen* und *Applikationen*. Eine Abstraktion ist eine „anonyme“ Funktion. Zum Beispiel ist $(\lambda x.x)$ eine Funktion, die ein Argument x entgegen nimmt und unverändert zurück gibt. Diese Funktion wird auch *Identitätsfunktion* genannt. Funktionen mit größerer Stelligkeit können so definiert werden: $(\lambda x.\lambda y.\lambda z.\ \ldots)$. Die Funktion $(\lambda x.\lambda y.x)$ nimmt zwei Argumente entgegen und liefert das erste zurück. Eine Applikation ist eine Anwendung einer Funktion auf ein Argument. Zum Beispiel ist $(\lambda x.x)\ (\lambda x.x)$ die Anwendung der Identitätsfunktion auf die Identitätsfunktion. Diese einfache Syntax reicht bereits aus, um alle berechenbaren Funktionen zu beschreiben. Der Lambda-Kalkül ist Turing-äquivalent.

1.3 Die Sprache L_{PLC}

In [1] wird eine funktionale Kernsprache[3] namens L_{PLC} vorgestellt. Die Ausgabe unseres Typcheckers sind (im positiven Fall) L_{PLC}-Ausdrücke.

Konvention 1.3.1. Wir bezeichnen Variablen mit x, y oder z, Datenkonstruktoren mit D, Typausdrücke mit Großbuchstaben wie R, S, T, Q und Typkonstruktoren mit K, möglicherweise mit Indizierung oder mit diakritischen Zeichen versehen, alle anderen Namen stehen für allgemeine Terme.

[2] Er wird in [4] sehr detailliert behandelt.

[3] eine funktionale Sprache mit einer simplen Syntax

Abbildung 1.3.1: Syntax von L_{PLC} ohne Typmarkierungen

$$
\begin{array}{rcl}
E & ::= & V \mid (E\ E) \mid \lambda V.E \mid (\texttt{amb}\ E\ E) \mid (\texttt{seq}\ E\ E) \\
& \mid & (\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E) \mid (D_i\ E_1 \ldots E_{\mathrm{ar}(D_i)}) \\
& \mid & (\texttt{case}_K\ E\ \texttt{of}\ \{Alt_1; \ldots; Alt_{|\mathbb{D}_K|}\}) \\
Alt_i & ::= & ((D_i\ V_1 \ldots V_{\mathrm{ar}(D_i)}) \to E)
\end{array}
$$

Die Sprache enthält einen Operator `seq` und einen nichtdeterministischen Operator `amb` (der Name steht für „ambigious“), der von J. McCarthy [17] eingeführt wurde. Der Operator `seq` nimmt zwei Ausdrücke entgegen, wertet den ersten aus und gibt den zweiten zurück. `amb` nimmt zwei Ausdrücke s und t entgegen und

- falls s nicht terminiert, wird der Wert von t zurückgeliefert,
- falls t nicht terminiert, wird der Wert von s zurückgeliefert,
- falls beide Ausdrücke terminieren, wird zufällig der Wert von s oder t zurückgeliefert,
- falls beide nicht terminieren, terminiert der gesamte Ausdruck nicht.

Lässt man den `amb`-Operator weg, kann die Sprache als eine Kernsprache von Haskell gesehen werden. Es gibt Variablen, Lambda-Ausdrücke, ein rekursives Let, Datenkonstruktoren und Case-Ausdrücke. Das rekursive Let lässt im Gegensatz zum nicht-rekursiven Let zu, dass in einem Ausdruck

$$\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$$

eine Letrec-gebundene Variable x_i in den rechten Seiten $e_1, \ldots, e_n$ der Bindungen vorkommt (und nicht nur in e). Die Sprache enthält vordefinierte Datenkonstruktoren `True`, `False`, [] und den Listenkonstruktor (:).[4] Die Stelligkeit eines Datenkonstruktors D bezeichnen wir mit $\mathrm{ar}(D)$, außerdem verlangen wir, dass alle Datenkonstruktoren (und die Operatoren `seq` und `amb`) gesättigt vorkommen, d. h. dass partielle Anwendungen wie (:) [] verboten sind. Es gibt weiterhin Typkonstruktoren, wobei ein Typkonstruktor K die Stelligkeit $\mathrm{ar}(K)$ hat. Wir nehmen an, dass wir bei der Typherleitung in der Lage sind, die Datenkonstruktoren eines Typs aufzuzählen. Die Menge der Datenkonstruktoren, die zu einem Typ gehören, bezeichnen wir mit $\mathbb{D}_K$. Case-Ausdrücke müssen mit einem Typkonstruktor K markiert sein, und genau die Datenkonstruktoren in den Pattern enthalten, die zu K gehören. Die Variablen in den Pattern müssen paarweise

[4] In der Implementierung gibt es noch die Konstruktoren `Left` und `Right`, die zum Typ *Either* gehören.

verschieden sein. Alle Teilausdrücke (bis auf wenige Ausnahmen) und alle Pattern müssen mit Typen markiert sein. In Abbildung 1.3.1 ist eine Kontextfreie Grammatik für L_{PLC} ohne Typmarkierungen dargestellt. Die Syntax von Typausdrücken ist wie folgt (a bezeichnet eine Typvariable):

$$T ::= a \mid (T \to T) \mid (K\ T_1 \ldots T_{\mathrm{ar}(K)})$$

Typausdrücke dürfen außerdem einen Quantor enthalten, der aber nur ganz außen vorkommen darf. Quantifizierte Typausdrücke haben die Form $\forall a_1, \ldots, a_n.T$, $a_1, \ldots, a_n$ sind Typvariablen. Die Reihenfolge der Typvariablen spielt keine Rolle, so dass der Typ auch so geschrieben werden darf: $\forall \mathcal{X}.T$, wobei $\mathcal{X}$ eine Menge von Typvariablen ist. Ein Quantor bindet Variablen. Die Menge der *freien* Typvariablen in einem Typausdruck T bezeichnen wir mit $\mathcal{FTV}(T)$. Wir nehmen an, dass uns eine Funktion *typeOf* zur Verfügung steht, die den Typ eines Datenkonstruktors liefert. Sei K ein Typkonstruktor. Jeder Datenkonstruktor $D_{K,i} \in \mathbb{D}_K$ muss einen Typ der Form

$$\forall a_1, \ldots, a_{\mathrm{ar}(K)}.T_{K,i,1} \to \ldots \to T_{K,i,m_i} \to K\ a_1 \ldots a_{ar(K)}$$

haben, wobei $m_i = \mathrm{ar}(D_{K,i})$, $a_1, \ldots, a_{\mathrm{ar}(K)}$ paarweise verschiedene Typvariablen sind und nur diese Variablen in $T_{K,i,1}, \ldots, T_{K,i,m_i}$ vorkommen dürfen.

Für tiefer gehende Betrachtungen benötigen wir die Begriffe „Variable kommt gebunden vor“ und „Variable kommt frei vor“.

Definition 1.3.1. Die Menge $\mathcal{FV}(e)$ der Variablen, die in einem Ausdrucks e *frei* vorkommen, sei definiert durch die Rechenvorschriften

$$\begin{aligned}
\mathcal{FV}(x) &:= \{x\} \\
\mathcal{FV}(y) &:= \emptyset \quad \text{(falls } y \text{ eine Konstante ist, } \texttt{True}, \texttt{False}) \\
\mathcal{FV}(\lambda x.t) &:= \mathcal{FV}(t) \setminus \{x\} \\
\mathcal{FV}(s\ t) &:= \mathcal{FV}(s) \cup \mathcal{FV}(t) \\
\mathcal{FV}(\texttt{seq}\ s\ t) &:= \mathcal{FV}(s) \cup \mathcal{FV}(t) \\
\mathcal{FV}(\texttt{amb}\ s\ t) &:= \mathcal{FV}(s) \cup \mathcal{FV}(t) \\
\mathcal{FV}(D\ s_1\ \ldots\ s_n) &:= \mathcal{FV}(s_1) \cup \ldots \cup \mathcal{FV}(s_n) \\
\mathcal{FV}(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) &:= \big(\mathcal{FV}(t) \cup \mathcal{FV}(s_1) \cup \ldots \cup \mathcal{FV}(s_n)\big) \setminus \{x_1, \ldots, x_n\} \\
\mathcal{FV}\left(\begin{array}{l}\texttt{case}_K\ s\ \texttt{of \{} \\ \quad D_1\ x_{1,1}\ \ldots\ x_{1,k} \to t_1, \\ \quad \ldots, \\ \quad D_n\ x_{n,1}\ \ldots\ x_{n,m} \to t_n \\ \texttt{\}}\end{array}\right) &:= \begin{array}{l}\mathcal{FV}(s) \\ \cup \big(\mathcal{FV}(t_1) \setminus \{x_{1,1}, \ldots, x_{1,k}\}\big) \\ \cup\ \ldots \\ \cup \big(\mathcal{FV}(t_n) \setminus \{x_{n,1}, \ldots, x_{n,m}\}\big)\end{array}
\end{aligned}$$

Definition 1.3.2. Die Menge $\mathcal{GV}(e)$ der Variablen, die in einem Ausdrucks e *gebunden* vorkommen, sei definiert durch die Rechenvorschriften

$$
\begin{array}{rcl}
\mathcal{GV}(x) & := & \emptyset \\
\mathcal{GV}(y) & := & \emptyset \quad \text{(Falls } y \text{ eine Konstante ist)} \\
\mathcal{GV}(\lambda x.t) & := & \mathcal{GV}(t) \cup \{x\} \\
\mathcal{GV}(s\ t) & := & \mathcal{GV}(s) \cup \mathcal{GV}(t) \\
\mathcal{GV}(\texttt{seq}\ s\ t) & := & \mathcal{GV}(s) \cup \mathcal{GV}(t) \\
\mathcal{GV}(\texttt{amb}\ s\ t) & := & \mathcal{GV}(s) \cup \mathcal{GV}(t) \\
\mathcal{GV}(D\ s_1\ \ldots\ s_n) & := & \mathcal{GV}(s_1) \cup \ldots \cup \mathcal{GV}(s_n) \\
\mathcal{GV}(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t) & := & \begin{array}{l} \mathcal{GV}(t) \\ \cup\, \mathcal{GV}(s_1) \cup \ldots \cup \mathcal{GV}(s_n) \\ \cup\, \{x_1, \ldots, x_n\} \end{array} \\
\mathcal{GV}\left(\begin{array}{l} \texttt{case}_K\ s\ \texttt{of}\ \texttt{\{} \\ \quad D_1\ x_{1,1}\ \ldots\ x_{1,k} \to t_1, \\ \quad \ldots, \\ \quad D_n\ x_{n,1}\ \ldots\ x_{n,m} \to t_n \\ \texttt{\}} \end{array}\right) & := & \begin{array}{l} \mathcal{GV}(s) \\ \cup\, \mathcal{GV}(t_1) \cup \ldots \cup \mathcal{GV}(t_n) \\ \cup\, \{x_{1,1}, \ldots, x_{1,k}, \ldots, x_{n,1}, \ldots, x_{n,m}\} \end{array}
\end{array}
$$

Definition 1.3.3. Ein Vorkommen einer Variablen x ist *gebunden*, wenn es einen einschließenden Term gibt, der x bindet, sonst ist das Vorkommen *frei*.

1.4 Reduktionen

Dieser Abschnitt beschreibt, wie Ausdrücke ausgewertet werden können und was die sog. *Normalordnungsreduktion* ist. Die betrachtete Sprache ist eine funktionale Kernsprache mit Lambda-, Case-Ausdrücken und Datenkonstruktoren.

Ein Auswertungsprozess terminiert im Idealfall und es entsteht ein Wert, bzw. ein Ausdruck, der nicht weiter ausgewertet werden muss.

Definition 1.4.1. Eine WHNF (weakest head normal form) ist ein Ausdruck der Form

- $\lambda x.e$ (Abstraktion) oder
- $(D\ t_1\ \ldots\ t_n)$ (Konstruktoranwendung) mit $n = \mathrm{ar}(D)$, t_i sind Ausdrücke.

Eine genauere Unterteilung ist: eine Abstraktion ist eine FWHNF, eine Konstruktoranwendung ist eine CWHNF.

Das Ziel ist, einen Ausdruck so auszuwerten, dass ein Ausdruck in WHNF entsteht. Die *operationale Semantik* beschreibt, wie Ausdrücke auszuwerten sind. Der Prozess der

Auswertung basiert auf sog. *Reduktionen*. Dabei wird ein Ausdruck vereinfacht (simplifiziert). In $(\lambda x.x)\ z$ kann das Argument z in den Funktionsrumpf eingesetzt werden und so der Gesamtausdruck zu z reduziert werden. Dieses Einsetzen heißt *β-Reduktion*, die Regel

$$\frac{((\lambda x.t)\ s)}{t[s/x]} \qquad (\beta\text{-}Reduktion)$$

besagt, dass eine Anwendung[5] transformiert wird, indem der Ausdruck durch t ersetzt wird und alle freien Vorkommen von x in t durch s ersetzt werden. Die Regel

$$\frac{(\texttt{case}_T\ (C\ t_1 \ldots t_n)\ \text{of}\ \{\ldots; C\ x_1 \ldots x_n\ \texttt{->}\ s; \ldots\})}{s[t_1/x_1, \ldots, t_n/x_n]}$$

reduziert einen Case-Ausdruck. Ein (Unter)Ausdruck, der nach diesen Regeln reduziert werden kann, wird *Redex* genannt.

Damit liegt aber noch keine eindeutige Vorschrift vor, mit der Ausdrücke deterministisch ausgewertet werden können. Für zum Beispiel

$$\underbrace{(\lambda x.x\ \overbrace{(\lambda y.y\ z)}^{Redex})}_{Redex}$$

gibt es zwei Möglichkeiten, den Ausdruck zu reduzieren. Durch *Reduktionskontexte* kann eine Reihenfolge festgelegt werden. Ein Reduktionskontext ist ein Ausdruck mit einem Loch:

$$R ::= [\,] \mid (R\ e) \mid (\texttt{case}\ R\ \texttt{of}\ \{p_1\ \texttt{->}\ t_1, \ldots, p_n\ \texttt{->}\ t_n\})$$

Die Schreibweise $R[s]$ bedeutet, dass der Ausdruck s in das Loch von R eingesetzt wird. Sei R ein Reduktionskontext, dann ist

$$R[s] \to R[t]$$

eine *Ein-Schritt-Normalordnungsreduktion*. Dabei ist s ein Redex (also keine WHNF) der unmittelbar zu t reduziert. Nach dieser Definition besitzt der Ausdruck $(\lambda x.x)\ ((\lambda y.y)\ z)$ die Normalornungsreduktion

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

mit dem Reduktionskontext $R = [\,]$. Die Reduktionsrelation wird mit n markiert. Weiteres Beispiel:

$$((\lambda x.\lambda y.x)\ s\ t) \xrightarrow{n} ((\lambda y.s)\ t) \quad \text{mit } R = ([\,]\ t).$$

Ein geschlossener Term t terminiert, wenn er zu einer WHNF reduziert werden kann. Die Terminierung (Konvergenz) wird mit $t \Downarrow$, die Divergenz mit $t \Uparrow$ bezeichnet.

Der Normalordnungsredex kann mit Hilfe des *Unwind-Algorithmus* gefunden werden. Dabei wird ein Label R unter Verwendung der Regeln

[5] Parameterübergabe

$$\begin{array}{lcl} C[(s\ t)^R] & \rightarrow & C[(s^R\ t)] \\ C[(\texttt{case}_T\ s\ \texttt{of}\ alts)^R] & \rightarrow & C[(\texttt{case}_T\ s^R\ \texttt{of}\ alts)] \end{array}$$

so lange verschoben, bis die Regeln nicht mehr angewendet werden können. Danach kann die Normalordnungsreduktion nach den Regeln

$$C[((\lambda x.t)^R\ s)] \ \rightarrow\ C[t[s/x]] \qquad (Beta)$$

$$\begin{array}{ll} C[\texttt{case}_T\ (c\ t_1\ \ldots\ t_n)^R\ \texttt{of}\ \{\ldots; c\ x_1\ \ldots\ x_n \texttt{ -> } s; \ldots\}] & (Case) \\ \rightarrow\ C[s[t_1/x_1, \ldots, t_n/x_n]] & \end{array}$$

durchgeführt werden.

Kapitel 2

Einige Grundlagen

Nachfolgend betrachten wir einige wichtige Sprachkonstrukte der Programmiersprache Haskell, wie sie in dieser Arbeit verwendet werden.[1] Haskell ist eine funktionale Programmiersprache mit einem starken Typsystem und verzögerter Auswertung (Ausdrücke werden bei Bedarf ausgewertet).

2.1 Aggregierte Datentypen

Typen wie `Int` oder `Char` sind elementare Typen. Oft ist es sinnvoll neue Datentypen aus anderen zusammenzusetzen, wodurch ein *aggregierter* Datentyp entsteht. So ist zum Beispiel `String` eine Liste von Zeichen. Listen können wiederum Bestandteile eines anderen Typs sein. In Haskell kann beispielsweise mit

```
data Address = Addr String String
```

eine (monomorphe) Datenstruktur definiert werden, die eine Adresse speichert. Hier ist `Address` ein *Typ-* und `Addr` ein *Datenkonstruktor* (die Bezeichner können auch gleich sein, da sie in verschiedenen Namensräumen liegen). Diese Definition ist vergleichbar mit der eines Records in anderen Sprachen. Man kann anschließend mit

```
showAddress :: Address -> String
showAddress (Addr name street) = name ++ "\n" ++ street
```

eine Funktion definieren, die auf solchen „Records“ operiert.

Datenstrukturen können polymorph sein. Der Begriff *Polymorphie* bedeutet Vielgestaltigkeit. Es gibt verschiedene Arten von Polymorphie. Als *Ad-hoc-Polymorphie* wird das Überladen von Funktionen bezeichnet. Bei Haskell-Funktionen, die *gleichartig* auf ver-

[1] Für eine vollständige Haskell-Einführung siehe [6] oder [8].

schiedenen Typen arbeiten (und zum Beispiel bei Templates in C++ oder JAVA), handelt es sich um *parametrische Polymorphie* – ein Name kann theoretisch unendlich viele Typen haben. Beispiel eines polymorphen Typs:

```
data Point a = Pt a a
```

Hier ist a eine *Typvariable* und `Point` ein Typkonstruktor. Wenn in einem Ausdruck, der einen Typ beschriebt, Typvariablen vorkommen, beschreibt der Ausdruck einen polymorphen Typ. Der oben definierte Typ enthält alle Punkte, deren Koordinaten den Typ a haben, wobei für `a` ein konkreter Typ eingesetzt werden kann. Der Typ von `Pt` ist $a \to a \to$ `Point` a. Datenkonstruktoren sind spezielle Funktionen. Im Unterschied zu den normalen Funktionen können sie als *Pattern* verwendet werden.

Ein Datenkonstruktor kann benutzt werden um einen Wert zu erzeugen und ein Typkonstruktor wird benutzt um einen Typ zu erzeugen. Durch

```
psum :: (Point Integer) -> Integer
psum (Pt x y) = x+y
```

wird eine Funktion definiert, die einen Parameter vom Typ `Point Integer` erwartet. (`Pt` x y) ist ein Pattern. Die selbe Funktion, definiert ohne die Typsignatur, hat den Typ

```
psum :: (Num a) => Point a -> a
```

und ist damit eine polymorphe Funktion, da sie für alle Typen aus der *Typklasse* `Num` definiert ist. Typdefinitionen können rekursiv sein, so ist zum Beispiel

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

eine Definition eines Typs, der einen binären Baum repräsentiert – ein Baum kann ein Blatt mit einem Wert sein, oder eine Verzweigung mit zwei Teilbäumen.

Bemerkung 2.1.1. `->` ist ein Typkonstruktor, der für zwei Typen T_1 und T_2 Funktionen von T_1 nach T_2 beschreibt.

2.2 Typklassen

Eine Typklasse ist eine Zusammenfassung von Typen, auf denen bestimmte Operationen ausgeführt werden können. Die Instanzen (Elemente) sind also Typen. Zum Beispiel ist die eingebaute Typklasse `Eq` eine Zusammenfassung von Typen, deren Elemente auf Gleichheit geprüft werden können.

```
class Eq a where
        (==), (/=) :: a -> a -> Bool
```

Der obigen Definition kann man entnehmen, dass für einen Typ, der zur Klasse `Eq` gehört (a ist eine Typvariable), zwei Funktionen `(==)` und `(/=)` implementiert sein müssen, die zwei Elemente dieses Typs auf `Bool` abbilden. Es folgt ein weiteres Beispiel. Eine Klasse von Typen, für deren Elemente ein Hashwert berechnet werden kann:

```
class Hash a where
        hash :: a -> Int
        hash _ = 0

data Pair a = Pr a a

instance (Enum a) => Hash (Pair a) where
        hash (Pr x y) =
            ((fromEnum x) + (fromEnum y)) `mod` 7

instance Hash Char
```

Die Typklasse `Hash` definiert eine Schnittstelle für eine Funktion (`hash`) und eine Standardimplementierung dafür. Der Typ `Pair` wird zu einer Instanz von `Hash` erklärt (man beachte, dass `Pair` ein polymorpher Typ ist). Dabei muss noch die Vorbedingung erfüllt sein, nämlich dass die Elemente des Paares zur Typklasse `Enum` gehören. Diese Implementierung der Funktion `hash` ist also speziell für Typen (`Pair` a), wobei a zu `Enum` gehört. Der Typ `Char` wird auch zu einer Instanz von `Hash` erklärt. Die Funktion `hash` wird nicht explizit definiert; es wird dann die Standardimplementierung benutzt, d. h. der Hashwert eines Zeichens ist immer 0.

Hashwerte können abhängig vom Typ jeweils anders berechnet werden, es ist aber so nicht nötig jeweils verschiedene Funktionsnamen zu verwenden. Mit Hilfe von Typklassen ist es möglich, Funktionen zu *überladen*. Je nach Typ wird dann die passende Implementierung verwendet (*Ad-Hoc-Polymorphismus*).

2.3 List-Comprehensions

List-Comprehensions (*ZF-Ausdrücke*) sind an die Zermolo-Fraenkel-Mengenschreibweise angelehnte[2] Ausdrücke, die Listen beschreiben. *Generatoren* und *Guards* bestimmen, welche Kriterien die enthaltenen Elemente erfüllen müssen. Zum Beispiel ist

$$[\overbrace{(\mathrm{x},\mathrm{y})}^{Kopf} \mid \overbrace{\mathrm{x} <- [1..6]}^{Generator},\ \mathrm{y} <- [1..6],\ \overbrace{\mathrm{x}+\mathrm{y}<5}^{Guard}]$$

[2] In der Mathematik ist die Schreibweise etwas weniger „normiert“. Man schreibt zum Beispiel $\{x \in \mathbb{N} \mid x \ldots\}$ *und auch* $\{(a,b) \mid a \in A,\ b \in B\}$ (Element-Zeichen vor oder hinter dem Strich), siehe [15]. Das ist nicht der einzige Unterschied; betrachte `f x = [x | x <- ys]` und $f(x) = \{x \mid x \in M\}$, die Haskell-Funktion ignoriert ihren Parameter.

eine Liste von Tupeln, deren Elemente addiert eine Zahl kleiner 5 ergeben. Der erste Generator iteriert am langsamsten. Lässt man die Generatoren über einen größeren Bereich laufen, ergibt das natürlich die selbe Liste. Im linken Teil eines Generators steht im einfachsten Fall eine Variable; allgemein ist es ein Pattern. Der Ausdruck

```
[a | (a,1) <- [(1,1),(1,2),(1,3)]]
```

ergibt die Liste `[1]`. Neue Variablen werden auf der linken Seite der Generatoren eröffnet. Der Geltungsbereich ist rechts vom Generator und der Kopfausdruck. Guards sind Ausdrücke vom Typ `Bool`.

List-Comprehensions können in ZF-freie Ausdrücke transformiert werden. Im einfachen Fall entspricht das Ergebnis einer Anwendung einer Funktion (die aus dem Kopfausdruck erzeugt wird) auf die Elemente der Liste, die von einem Generator-Ausdruck erzeugt wird. Zum Beispiel kann [f x | x <− xs] nach (**map** (\\x −> f x) xs) transformiert werden. In der allgemeineren Form

$$[e \mid x <- xs] \quad \rightarrow \quad \textbf{concat}\ (\textbf{map}\ (\backslash x -> [e])\ xs)$$

entsteht eine Liste von Listen, die durch die Anwendung von `concat` zu einer flachen Liste wird. Genauere Regeln für Transformationen sind nachfolgend gegeben.

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

Zum Beispiel kann damit der ZF-Ausdruck

$$[(x,y) \mid x <- xs,\ y <- ys]$$

nach **concat** (**map** (\x −> (**map** (y−> (x,y)) ys)) xs)

transformiert werden. Es ist nur ein `concat` im fertigen Ausdruck, da für einfache Ausdrücke die Regel

$$[e \mid x <- xs] = (\textbf{map}\ (\backslash x -> e)\ xs)$$

angewendet werden kann. Es lassen sich mit den obigen Regeln nicht alle Comprehensions richtig transformieren. Die Transformation des Ausdrucks

$$[x \mid \textbf{Just}\ x <- xs] \quad \rightarrow \quad \textbf{concat}\ (\textbf{map}\ (\backslash(\textbf{Just}\ x) -> [x])\ xs)$$

kann zur Laufzeit Fehler erzeugen. Bessere Übersetzung:

```
concat (map (\x -> case x of {(Just x) -> [x]; _ -> []}) xs)
```

Die Übersetzung ist besonders dann nützlich, wenn eine List-Comprehension auf Typfehler untersucht werden soll. Man muss so keine speziellen Regeln für die Typherleitung definieren. Folgende Funktion soll für eine Liste eine Liste von Listen liefern, die sich selbst nicht enthalten:

```
Prelude> \xs -> [x | x <- xs, x `notElem` x]

<interactive>:1:33:
    Occurs check: cannot construct the infinite type: t = [t]

    ...
```

Kapitel 3

Struktur eines Programms

Bevor wir mit einem Programmquelltext etwas anfangen können, bevor wir Typen berechnen können, müssen wir daraus erst eine geeignete Datenstruktur erstellen (einen Syntaxbaum). In diesem Kapitel beschreiben wir, wie aus einem Quelltext eine Datenstruktur erstellt wird, die das Programm im Speicher repräsentiert und mit der wir leichter umgehen können als mit einer Folge von Zeichen. Der erste Schritt dieser Transformation ist die *lexikalische Analyse*, danach folgt die *syntaktische* und anschließend die *semantische Analyse*. Quelltexte können Fehler enthalten, die in jeder der drei Phasen auftreten können (lexikalische, syntaktische oder semantische Fehler). Wir beschreiben, wie die Fehler aussehen und wie sie abgefangen werden. Die erkannte Sprache nennen wir $L_{PLC^{-T}}$, weil sie im Unterschied zu L_{PLC} keine Typmarkierungen enthält.

3.1 Zerlegung in Token

Ein *Lexer* (oder *Tokenizer*) transformiert den Quelltext eines Programms in eine Folge von *Token* (syntaktische Einheiten), wobei es verschiedene Tokentypen gibt. Im Wesentlichen entspricht jeder Variablenname, jeder Datenkonstruktorname, jedes Symbol (manchmal mehrere Zeichen lang), das eine Bedeutung hat, einem Token. Eine Zeichenfolge, die einem Token entspricht, nennen wir *Lexem*.

Tabelle 3.1.1: Beispiele für Tokentypen

Lexem	Tokentyp
`x`	`TokVarId`
`True`	`TokConId`
`(`	`TokOParen`

Der Lexer kann bereits einige Fehler im Quelltext erkennen – zum Beispiel unerlaubte Zeichen. Hingegen wird auf Fehler wie falsche Klammerung nicht getestet. Die lexikalische Analyse basiert auf der Erkennung einer regulären Sprache und kann in Linearzeit ausgeführt werden. Es geht jedoch, wie schon angedeutet, nicht ausschließlich um das Wortproblem, also das Entscheidungsproblem, ob der Quelltext aus einer bestimmten Grammatik abgeleitet werden kann oder nicht. Die Ausgabe des Lexers ist die Eingabe eines weiteren Verarbeitungsschritts, der syntaktischen Analyse, für den der Tokenstrom eine einfacher zu verarbeitende Form darstellt. Außerdem werden nicht alle Informationen bei der Weiterverarbeitung benötigt, so filtert der Lexer alle Kommentare und Leerzeichen heraus.

Wir modellieren die Tokentypen durch Datenkonstruktoren des Typs `Token`. Beim Lexen wird jedes Token zusätzlich mit einer Positionsmarkierung (Zeile und Spalte im Quelltext) versehen, welche, falls der Quelltext Fehler enthält, zusammen mit Fehlermeldungen ausgegeben wird.

```
data Token
       = TokVarId String
       | TokConId String
       | TokOParen
       | ...
```

Ein Bezeichner kann ein Variablen- oder ein Konstruktorname sein. Mit der Konvention, dass Konstruktornamen mit einem Großbuchstaben beginnen, können wir bereits beim Lexen eine entsprechende Trennung vornehmen. Token für Datenkonstruktoren und Variablen speichern zusätzlich den Konstruktor- bzw. Variablennamen.

Die Funktion `tokenize :: String -> [(Token, Pos)]` in `Lexer.hs` im Unterverzeichnis `parser` transformiert einen Zeichenstrom (Quelltext eines Programms) in eine Liste von Paaren bestehend aus einem Token und einer Positionsangabe. Das eigentliche Lexen übernimmt u. a. die Funktion

```
lexScanTokens :: Pos -> String -> [(Token, Pos)]
```

die den Quelltext liest, Token mit Positionsmarkierungen erzeugt und zu einer Liste verkettet. Nach jedem angehängten Token wird die aktuelle Position im Quelltext berechnet und die Funktion ruft sich selbst mit der neuen Position und dem Rest des Quelltextes auf.

Einzelne Zeichen, die Token entsprechen (Klammern, \, etc.), können am einfachsten erkannt werden. Dazu muss nur ein Zeichen der Eingabe gelesen und anschließend geprüft werden, ob es ein Terminal der Grammatik ist (`->` und `[]` werden erkannt, indem zwei Zeichen der Eingabe gelesen werden, für die leere Liste dürfen keine Leerzeichen zwischen den Klammern stehen). Diese Terminale können als Trennzeichen gesehen werden. Zum Beispiel kann `"a=True"` eindeutig in drei Token zerlegt werden.

```
*Lexer> tokenize "a=True"
[(TokVarId "a",(1,1)),(TokEq,(1,2)),(TokConId "True",(1,3))]
```

Leerzeichen sind auch Trennzeichen. Weil sie jedoch keine weitere Bedeutung haben, werden keine Token dafür erzeugt; wir müssen nur die Position im Quelltext aktualisieren. Für einen Tabulator oder für `'\r'` wird die Spaltennummer, genau wie für ein Leerzeichen, um 1 erhöht.

3.2 Schlüsselworterkennung

Jedem Schlüsselwort entspricht ein eigener Tokentyp. Um festzustellen, ob eine Folge von Zeichen ein Schlüsselwort ist, müssen wir einen Teil des so genannten Wörterbuchproblems lösen. Ein Wörterbuch ist ein abstrakter Datentyp, in dem Objekte (Daten) gespeichert, gesucht und gelöscht werden können. Jedem Objekt ist ein eindeutiger Schlüssel zugeordnet, so dass der Zugriff über diesen Schlüssel erfolgt. Manche Autoren bezeichnen auch eine Menge zusammen mit einer Einfüge-, Lösch- und Suchoperation als Wörterbuch (siehe [6], Abschnitt 8.3). Wir benötigen eine Zuordnung

$$\textit{Schlüsselwort} \mapsto \textit{Tokentyp}.$$

Die Schlüsselwörter der Sprache sind eindeutig, d. h. wir können sie als Schlüssel zusammen mit dem dazugehörigen Tokentyp in einem Wörterbuch speichern. Außerdem sind die Schlüsselwörter fest – während des Lexens kommen keine neuen Schlüsselwörter hinzu und es werden auch keine aus der Sprache entfernt, so dass wir nur die Suche benötigen.

Die Implementierung eines Wörterbuchs kann mit Hilfe verschiedener Datenstrukturen erfolgen: Hashtabelle, binärer Suchbaum, B-Baum, um einige zu nennen. Außerdem gibt es das Haskell-Modul `Data.Map`, welches bereits eine fertige Implementierung eines Wörterbuchs bereitstellt. Wir verwenden einen so genannten Trie (siehe auch [16]).

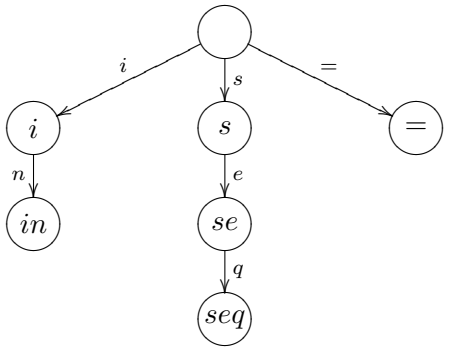
Definition 3.2.1. Ein *Trie*[1] ist eine Baumstruktur, in der jeder Knoten einen Präfix eines Wortes speichert. Die Wurzel repräsentiert das leere Wort. Die Kanten sind mit Zeichen markiert.

In Abbildung 3.2.1 ist ein Trie dargestellt, mit dessen Hilfe die Schlüsselwörter `in` und `seq` und das Gleichheitszeichen erkannt werden können. Obwohl das Gleichheitszeichen nicht unbedingt als Schlüsselwort zu bezeichneten ist, spricht nichts dagegen, es auch in einem Trie zu speichern. Weil kein Schlüsselwort Präfix eines anderen ist, werden Schlüssel nur in den Blättern gespeichert.

Sei ℓ ein Wort, das einem Lexem entspricht. Wir können durch partielles Traversieren des Tries feststellen, ob es sich um ein Schlüsselwort handelt oder nicht. Dazu setzen wir am

[1] Der Name leitet sich von re*trie*val ab.

Abbildung 3.2.1: Ein Trie



Anfang $w = \ell$ und (1) vergleichen das erste Zeichen w_1 mit den Kantenmarkierungen der Kanten, die von der Wurzel zu den Nachfolgerknoten führen. Existiert keine Markierung w_1, ist das Wort nicht im Wörterbuch enthalten. Falls ein Teilbaum T_{w_1} existiert, zu dem eine Kante mit einer Kantenmarkierung w_1 führt und

i) T_{w_1} ist kein Blatt, setzen wir $w = w_{2..n}$ (w ohne das erste Zeichen) und wenden das Verfahren ab (1) rekursiv auf T_{w_1} an, oder

ii) T_{w_1} ist ein Blatt, k ist der im Blatt gespeicherte Schlüssel und $|w| = 1$ (w_1 ist das letzte Zeichen im Wort) $\Rightarrow$ der Quelltext enthält das Schlüsselwort $k = \ell$. Falls $|w| > 1$, enthält der Quelltext einen Bezeichner mit Präfix k.

Die Knoten des Tries in `Lexer.hs` sind Funktionen. Die Wurzel ist die Funktion `lexTrie`. Jede Funktion konsumiert ein Zeichen der Eingabe und entscheidet dann ob eine weitere Funktion auf dem Rest der Eingabe aufgerufen werden muss oder ob die Suche beendet werden kann. Die skizzenhaften Definitionen

```
lexTrie xs = case xs of
    ('i':cs) -> lexTrie_i cs
    ('s':cs) -> lexTrie_s cs
    ('=':cs) -> Just (TokEq,1,cs)
    _        -> Nothing

lexTrie_i ('n':cs) = ...
lexTrie_i _        = Nothing

...
```

sind eine mögliche Teilimplementierung des Beispieltries in Abbildung 3.2.1. Im Falle einer erfolgreichen Suche wird der Tokentyp, die Länge des Lexems und der Rest des

Quelltextes als Tupel gekapselt in einem `Just` zurück gegeben. Falls die Suche in einem Blatt endete, muss für ein Schlüsselwort überprüft werden, ob es sich wirklich um ein Schlüsselwort handelt. Dazu lesen wir ein zusätzliches Zeichen c der Eingabe. Falls c ein alphanumerisches Zeichen ist (oder `_`, `'`, `~`) handelt es um einen Variablennamen.

Zusätzlich zu den Schlüsselwörtern speichern wir auch Klammern, den Pfeil `->`, etc. (eigentlich alles, was kein Bezeichner ist) im Trie. Hier muss nach dem Erkennen kein weiteres Zeichen des Quelltextes gelesen werden – der Tokentyp steht direkt fest.

Es existieren weitere Trie-Varianten. Zum Beispiel können die Kanten nicht nur mit einzelnen Zeichen markiert werden, sondern auch mit Strings. Dadurch kann unser Wörterbuch auch so implementiert werden, dass alle Nachfolgerknoten der Wurzel Blätter sind (insbesondere weil die reservierten Wörter sich nicht überlappen, präfixfrei sind).

Die Funktion `lexScanTokens` wendet zuerst die Trie-Suche auf den Eingabestrom an. Bei einer erfolglosen Suche müssen wir zurück setzen und die Eingabe ab der Position, wo die Trie-Suche startete, bis zum nächsten Trennzeichen lesen, um Bezeichner zu erkennen (bzw. Kommentare und Leerzeichen überspringen).

```
lexScanTokens :: Pos -> String -> [(Token, Pos)]
lexScanTokens p@(r, s) xs = case lexTrie xs of
    Just (tkn, n, ys) -> (tkn, p)
                         : lexScanTokens (r, s+n) ys

    Nothing -> case xs of
        ('\n':cs) -> lexScanTokens (r+1, 1) cs

        ...
```

Mit unserer Implementierung des Wörterbuchs müssen wir nur dann zurücksetzen und einen Teil des Quelltextes neu lesen, wenn wir ein Nicht-Schlüsselwort vorliegen haben. Wir müssen ein Lexem nicht komplett gelesen haben, bevor wir im Wörterbuch danach suchen können. Das Zurücksetzen und das Lesen des Quelltextes mit einem „neuen Ziel“ ist unter anderem dafür verantwortlich, dass die Fehlermeldung beim Aufruf

```
*Lexer> tokenize "[a]"
*** Exception: (1,1): lexical error at character ‘[’
```

auf den ersten Blick vielleicht nicht ganz zutreffend erscheint. Das Zeichen `[` ist zwar zulässig, aber nur dann wenn es ein Teil von `[]` ist.[2] Analog verhält sich der Lexer, wenn im Quelltext zum Beispiel `-␣>` statt `->` vorkommt. Wir erzeugen Lexer-Fehlermeldungen an einer zentralen Stelle (innerhalb von `lexScanTokens`) und verzichten auf Fallunterscheidungen, um bei der Fehlerausgabe etwa auf das falsche `a` in `[a]` hinzuweisen. Da wir nach einer fehlgeschlagenen Trie-Suche davon ausgehen, dass es sich nur noch um

[2] Es gibt keinen „syntaktischen Zucker“, der `[a]` in `a:[]` überführen würde.

einen Bezeichner handeln kann (oder ein Kommentar, etc.), weist der Fehler vielmehr darauf hin, dass eine eckige Klammer nicht Teil eines Bezeichners sein kann.

Der Lexer kann auch so implementiert werden, dass der Trie „selbst entscheidet“, was als Nächstes passieren soll, wenn ein Token erkannt bzw. (noch) nicht erkannt wurde. Statt (`Just` ...) oder `Nothing` zurückzuliefern, kann auch direkt eine für die Situation passende Funktion aufgerufen werden. Wir belassen es bei der gegenwärtigen Lösung, um den Quelltext nicht unklarer zu machen.

3.3 Syntax von $L_{PLC^{-T}}$

Bei der so genannten *syntaktischen Analyse* konstruieren wir aus der durch den Lexer erstellten Liste von Token einen Syntaxbaum. Man nennt diesen Vorgang auch *Parsing*. Ein *Parser* arbeitet ausgehend von einer kontextfreien Grammatik und kann feststellen, ob der Quelltext syntaktisch korrekt ist; falls ja, wollen wir einen Syntaxbaum erhalten, falls nein, soll eine aussagekräftige Fehlermeldung ausgegeben werden. Ähnlich wie beim Lexen, besteht also die Aufgabe des Parsers nicht nur darin, das kontextfreie Wortproblem zu lösen. Im Allgemeinen können nicht nur Listen von Token verarbeitet werden und es wird auch nicht unbedingt ein Syntaxbaum erzeugt. Zum Beispiel kann das Ergebnis eine Zahl sein, wenn arithmetische Ausdrücke geparst werden. In Abbildung 3.3.1 ist eine kontextfreie Grammatik in EBNF-Schreibweise für die Sprache dargestellt, die der Parser als syntaktisch korrekt ansehen soll. Wir werden später einige Einschränkungen bzw. Erweiterungen einführen (die wir nicht mit der Grammatik ausdrücken können), so dass die eigentliche erkannte Sprache eine Teilmenge der hier definierten ist.

Die erweiterte Backus-Naur-Form (kurz EBNF) ist eine Metasyntax, die benutzt werden kann, um die Syntax einer Programmiersprache (oder allgemein einer formalen Sprache) exakt zu beschreiben. Wir wollen ein paar wichtige Syntax-Elemente der erweiterten BNF selbst aufzählen (siehe auch [13]).

Produktionsregel Eine Produktionsregel besteht aus einem Metabezeichner (Name des Nichtterminals, das definiert wird), gefolgt von einem Gleichheitszeichen (=), gefolgt von einer Definitionsliste, und wird mit einem Semikolon (;) terminiert.

Definitionsliste Eine Definitionsliste ist eine geordnete Liste mit einer oder mehreren Definitionen, die durch senkrechte Striche (|) voneinander getrennt sind.

Definition Eine Definition ist eine geordnete Liste bestehend aus einem oder mehreren syntaktischen Termen, die durch Kommas (,) voneinander getrennt sind. Ein Komma bedeutet, dass zwei Terme konkateniert werden.

Syntaktischer Term Verkürzt: ein syntaktischer Term ist ein Terminal-String oder ein Metabezeichner (kann außerdem auch ein Term sein, der zum Beispiel Wiederholungen von Zeichen beschreibt).

Abbildung 3.3.1: Syntax

$$
\begin{array}{rcl}
Exp & = & \text{"}\lambda\text{"},\ \text{"var"},\ \text{"}\rightarrow\text{"},\ Exp \\
 & | & \text{"letrec"},\ Binds,\ \text{"in"},\ Exp \\
 & | & A \\
 & | & A,\ \text{":"},\ Exp; \\
 & & \\
A & = & P \\
 & | & A,\ P\ ; \\
 & & \\
P & = & P2 \\
 & | & \text{"seq"},\ P2,\ P2 \\
 & | & \text{"amb"},\ P2,\ P2 \\
 & | & \text{"constructor1"},\ P2; \\
 & & \\
P2 & = & \text{"var"} \\
 & | & \text{"constructor"} \\
 & | & \text{"case"},\ Exp,\ \text{"of"},\ \text{"\{"},\ Alts,\ \text{"\}"} \\
 & | & \text{"("},\ Exp,\ \text{")"}\ ; \\
 & & \\
Binds & = & Bind \\
 & | & Binds,\ \text{","},\ Bind\ ; \\
 & & \\
Bind & = & \text{"var"},\ \text{"="},\ Exp\ ; \\
 & & \\
Alts & = & Alt \\
 & | & Alts,\ \text{","},\ Alt\ ; \\
 & & \\
Alt & = & Pat,\ \text{"}\rightarrow\text{"},\ Exp\ ; \\
 & & \\
Pat & = & \text{"constructor"} \\
 & | & \text{"constructor1"},\ \text{"var"} \\
 & | & \text{"var"},\ \text{":"},\ \text{"var"} \\
 & | & \text{"("},\ Pat,\ \text{")"}\ ;
\end{array}
$$

Terminal-String Ein Terminal-String ist eine Zeichenfolge, wobei das erste und das letzte Zeichen ein Anführungszeichen (") ist.

Weiterhin bestimmen wir, dass der Metabezeichner *Exp* das Startsymbol ist, denn es ist nicht egal, welche Produktionsregel der Ausgangspunkt für eine Herleitung ist.

Eine wichtige Eigenart der Grammatik ist: wir definieren hier nicht, wie ein syntaktisch korrekter Quelltext auszusehen hat. Die Terminal-Strings repräsentieren keine Zeichenketten im Quelltext, sondern Token in der Darstellung des Programms als Tokenstrom. Wie wir bereits wissen, besteht das Alphabet unseres Parsers aus Token (die zusätzlich noch mit Positionen markiert sind). Die Terminal-Strings sollen selbsterklärend sein. Wir hätten auch "`TokConId`" anstelle von "`constructor`" oder "`TokLambda`" statt "λ" schreiben können, um eine Parallele zu den Token-Konstruktornamen zu ziehen.

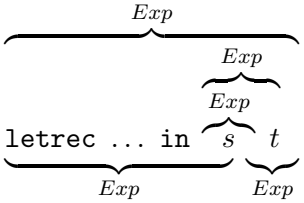
Alle null- bzw. einstelligen Datenkonstruktoren werden vom Lexer als `TokConId` bzw. als `TokConId1` gespeichert, so dass innerhalb der Grammatik für jede Stelligkeit eine eigene Regel existiert (der zweistellige Listenkonstruktor ist ein Sonderfall). Wegen der herausgenommenen Leerzeichen müssen wir nicht definieren, dass zum Beispiel zwischen zwei Argumenten beliebig viele davon stehen dürfen. Die Argumente sind nicht mehr auf diese Weise voneinander getrennt, sondern dadurch, dass es zwei Listenelemente sind, die sich an unterschiedlichen Positionen innerhalb der Liste befinden.

Die Zerlegung des Quelltextes in Token hat, abgesehen von den fehlenden Leerzeichen, insgesamt den Vorteil, dass wir uns jetzt nur noch mit den verschiedenen Tokentypen befassen müssen – es gibt nicht mehr unendlich viele Variablennamen, sondern nur Variablen-Token mit den darin gekapselten Namen. Deshalb gibt es keine Produktionsregeln für die Erzeugung von Bezeichnern, denn diese wurden schon durch die lexikalische Analyse erkannt.

Die Grammatik ist also etwas kompakter als die Definition der ursprünglichen Sprache. Auf den ersten Blick scheint unsere Beschreibung vielleicht nicht kompakt genug zu sein. Einige Produktionsregeln lassen sich möglicherweise eliminieren. Nehmen wir an, wir ersetzen die Regeln *Exp*, *A*, *P* und *P*2 durch eine einzige Regel

$$\begin{array}{rcl} Exp & = & \ldots \\ & | & \text{"letrec"},\ Binds,\ \text{"in"},\ Exp \\ & | & Exp,\ Exp\ ; \end{array}$$

und definieren somit Letrec-Ausdrücke und Applikationen (und auch sonstige zentrale Sprachkonstrukte) mit der selben Regel. Diese Grammatik ist nicht eindeutig. Zum Beispiel hat dann



mehrere Ableitungen und kann als

$$(\texttt{letrec ... in } (s\ t)) \tag{3.1}$$

$$\text{und als } ((\texttt{letrec ... in } s)\ t) \tag{3.2}$$

geparst werden. Ausdrücke der Form (3.1) können nicht durch solche wie (3.2) ersetzt werden. Zwar funktioniert das in bestimmten Ausnahmefällen, allgemein aber nicht. Im ersten Fall gelten die durch das Letrec definierten Variablen im gesamten Ausdruck, im zweiten Fall sind diese Variablen im Teilausdruck t unbekannt. Beispiel:

$$\texttt{letrec } id = \lambda x.x \texttt{ in } id\ id \quad \neq \quad (\texttt{letrec } id = \lambda x.x \texttt{ in } id)\ id$$

Eine Anwendung wie $(r\ s\ t)$ ist mit der geschrumpften Grammatik ebenfalls mehrdeutig, denn man kann daraus ebenso gut $((r\ s)\ t)$ wie $(r\ (s\ t))$ ableiten.

Wir beseitigen solche Mehrdeutigkeiten, indem wir eindeutige Regeln angeben. Die Applikation soll links-assoziativ sein: $(r\ s\ t)$ ist voll geklammert $((r\ s)\ t)$. Für

$$\texttt{letrec ... in } e$$

wollen wir definieren: ein `in` bindet am schwächsten; der Geltungsbereich der durch das Letrec gebundenen Variablen ist der Gesamtausdruck[3], wobei e ein Teilausdruck ist. Mit anderen Worten: es soll

$$x_1, \ldots, x_n \notin \mathcal{FV}(\texttt{letrec } x_1 = s_1, \ldots, x_n = s_n \texttt{ in } e)$$

gelten, wobei $x_1, \ldots, x_n$ Variablen und andere Namen allgemeine Ausdrücke sind, und $\mathcal{FV}(t)$ die Menge der freien Variablen in t ist (siehe Definition 1.3.1). Natürlich hält uns nichts davon ab, die implizite Klammerung durch eine explizite zu umgehen.

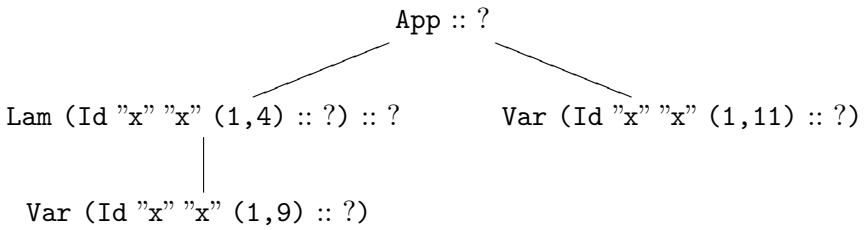
Die Grammatik stellt außerdem sicher, dass die vordefinierten Datenkonstruktoren und die Funktionen `seq` und `amb` nicht ungesättigt vorkommen. Aus diesem Grund gibt es für einstellige Konstruktoren, den Listenkonstruktor und für die beiden Funktionen eigene spezielle Tokentypen (`TokConId1`, `TokColon`, `TokSeq` und `TokAmb`). Verbietet man die partielle Anwendung nicht, können diese Bezeichner auch als `TokConId` und `TokVarId` gespeichert werden, wobei die Stelligkeiten dann kein „Syntaxproblem“ mehr sind.

3.4 Darstellung von Ausdrücken

Um das Ergebnis des Parsens (den Syntaxbaum) im Speicher zu repräsentieren, benötigen wir eine Datenstruktur. Diese Datenstruktur, der Datentyp `Expr`, ist in `Expr.hs` definiert. Für jedes Sprachkonstrukt (Variable, Funktionsanwendung, Abstraktion, usw.) gibt es einen Datenkonstruktor (`Var`, `App`, `Lam`, ...), wobei der Datentyp rekursiv ist. Zum Beispiel haben zwei Teile einer Anwendung und das, worauf eine durch ein Lambda gebundene Variable abgebildet wird, wiederum den Typ `Expr`.

[3] Das ist dann der Fall, wenn der Gesamtausdruck keine Applikation ist, sondern ein Letrec-Ausdruck.

Abbildung 3.4.1: Ein Syntaxbaum




```
data Expr
       = Var Id
       | App Expr Expr Type
       | Lam Id Expr Type
       | Amb Type Expr Expr Type
       | Seq Type Expr Expr Type
       | Let [Bind] Expr Type Int
       | ConApp Id [Expr] Type
       | Case TName Expr [Alt] Label

data Id = Id String String Pos Type

data Bind = Id :=: Expr

data Alt = Alt Id [Id] Type Expr
```

Zusätzlich speichern wir für jeden Teilausdruck den Typ, welcher aber erst nach dem Parsen ermittelt wird, so dass der Parser, während der Syntaxbaum erzeugt wird, keine gültigen Typen einsetzen kann, weil sie noch nicht bekannt sind; stattdessen setzen wir die Typen auf „unbekannt“.

In Abbildung 3.4.1 ist der Syntaxbaum für den Ausdruck $((\lambda x.x)\ x)$ dargestellt. Man kann leicht sehen, dass der Ausdruck nicht geschlossen ist, weil die Variable x einmal gebunden und einmal ungebunden vorkommt. Wir können den Baum dennoch angeben, weil der Ausdruck *syntaktisch* korrekt ist. Außerdem fällt auf, dass wir den Variablennamen `x` immer doppelt speichern. Der Grund dafür ist: im nächsten Schritt, während der semantischen Analyse[4], werden alle Variablen umbenannt (um eine gewisse Eindeutigkeit herzustellen), wir wollen aber trotzdem manchmal wissen, wie die Namen im Quelltext lauten, um den ursprünglichen Namen in Fehlermeldungen auszugeben, bzw. wollen wir den Syntaxbaum (oder Teile davon) wieder zurück in eine Qulltextdarstellung transfor-

[4]Bei der semantischen Analyse wird auf Fehler wie zum Beispiel undefinierte Variablen geprüft (die Typisierung zählen wir nicht dazu).

mieren können, in der die Variablen so heißen, wie sie ursprünglich hießen.

Die gespeicherte Position soll dazu dienen, Fehlermeldungen in weiteren Analyseschritten aussagekräftiger zu machen. Für Variablennamen, die direkt bei einem Lambda stehen (und in einigen anderen Fällen), benötigen wir eigentlich nicht alle Informationen. Die Positionsangabe ist nicht wichtig, weil an solchen Stellen nur Syntaxfehler auftreten können, und Falls dort ein Syntaxfehler auftritt, extrahieren wir die Informationen für die Fehlermeldung aus dem Token, das den Fehler verursachte.

3.5 Der Parser

Wir implementieren den Parser nicht „per Hand“. Stattdessen verwenden wir den Parsergenerator Happy[5]. Der Parsergenerator bekommt als Eingabe eine BNF-ähnliche Beschreibung der Sprache und erzeugt daraus einen Haskell-Quelltext – einen *Shift-Reduce-Parser*[6]. Die Datei `Parser.y` enthält diese Spezifikation zusammen mit Funktionsdefinitionen und einigen anderen Angaben; zum Beispiel teilen wir dem Generator mit, wie die zu generierende Parser-Funktion heißen soll.

```
%name parseTokens
```

Die Produktionsregeln in `Parser.y` entsprechen denen, die wir bereits aus Abbildung 3.3.1 (auf Seite 29) kennen.

```
Exp    : '\\' 'var' '->' Exp   { let (TokVarId x, p) = $2 in
                                   Lam (Id x x p NoType) $4 NoType }

       | 'let' Binds 'in' Exp  { Let (checkBinds $2) $4 NoType }
       | A                     { $1 }
       | A ':' Exp             { ConApp (Id ":" ":" (snd $2) NoType)
                                          [$1,$3] NoType }

Alt    : Pat '->' Exp          { checkPat ($1 $3) }

...
```

Jede Produktionsregel besteht auch hier aus einer oder mehreren Definitionen. Die Strings in Hochkommata bezeichnen Token. Welche Token das genau sind, definieren wir ebenfalls in der Eingabedatei des Parsergenerators.

```
%token
```

[5] http://www.haskell.org/happy/

[6] *Shift-Reduce-Parsing* ist eine spezielle Parsing-Strategie.

```
'\\'    { (TokLam, _)   }
'->'    { (TokArrow, _) }
'let'   { (TokLet, _)   }
'in'    { (TokIn, _)    }
':'     { (TokColon, _) }

...
```

Zu jeder Definition innerhalb einer Produktionsregel gehört ein Haskell-Ausdruck, welcher zwischen geschweiften Klammern steht und ein Teilergebnis des Parsens definiert. Dabei kann mit `$`i auf den Wert des i-ten syntaktischen Terms innerhalb der Definition zugegriffen werden (dadurch ist es kein reines Haskell). In einigen Fällen ist dieser Wert ein Token oder ein Teil eines Tokens, oder wir bekommen bereits einen Teilbaum des Syntaxbaums (wenn wir auf den Wert eines Metabezeichners zugreifen).

Wie schon erwähnt, speichert eine Bezeichner-Datenstruktur (Datentyp `Id`) den alten und den aktuellen Namen. Beim Parsen speichern wir Variablennamen doppelt, und ändern später einen der beiden Strings. Alternativ kann man für den Namen, der später geändert werden soll, erstmal einen leeren String speichern, oder statt `Expr` zwei Typen definieren: einen, der nur einen Namen pro Bezeichner speichert, und einen, der entsprechend erweitert ist. Die Umbenennung kann dann so implementiert werden, dass ein Syntaxbaum des ersten Typs in einen des zweiten Typs transformiert wird.

Die implizite Klammerung der Listen-Ausdrücke ist eine Rechtsklammerung. Wir wollen, dass zum Beispiel `True:True:[]` als `True:(True:[])` geparst wird, also definieren dass, der Listenkonstruktor rechtsassoziativ ist. Die Produktionsregel für den Listenkonstruktor ist etwas Besonderes, weil der Doppelpunkt ein Infix-Operator ist. Die Präfix-Schreibweise für `(x:xs)` ist `((:) x xs)` oder `(Cons x xs)`. Wir beschränken uns auf der Grammatik-Ebene auf die Infix-Notation, speichern Listen aber genauso wie sonstige Konstruktoren/Konstruktor-Anwendungen.[7]

```
True  →  ConApp (Id "True" ...
x:xs  →  ConApp (Id ":" ...
```

Um zu überprüfen, ob alle Datenkonstruktoren gesättigt vorkommen, müssen die Konstruktoren und die Stelligkeiten beim Parsen bekannt sein. Wenn man die Sprache erweitert und das Definieren von neuen Datentypen und Datenkonstruktoren im Programm erlaubt, kann keine kontextfreie Grammatik angegeben werden, die sicherstellt, dass jeder Konstruktor die richtige Anzahl von Argumenten übergeben bekommt. Zumindest dann nicht wenn für jede Stelligkeit eine eigene Produktionsregel benötigt wird, so dass für den Allgemeinfall die Beschreibung der Grammatik unendlich ist.

Eine weitere einschränkende Eigenschaft unserer Sprache ist: nicht alle Anwendungen werden links geklammert. Die Ausnahme sind Konstruktoranwendungen und Applika-

[7] `True` kann auch als eine nulläre Konstruktoranwendung gesehen werden.

tionen, die die vordefinierten Funktionen `seq` und `amb` enthalten, für die Verstöße gegen das Verbot der partiellen Anwendung ebenfalls anhand der Grammatik festgestellt werden. Zum Beispiel wird `(f amb a b)` als `(f (amb a b))` und nicht als `(((f amb) a) b)` geparst.

Es fällt vielleicht auf, dass die meisten rekursiven Definitionen, also solche, die Wiederholungen beschreiben, links-rekursiv sind. Im Fall von

```
A   : P       { $1 }
    | A P     { App $1 $2 NoType }
```

ist dies wegen der Links-Klammerung bei Applikationen gewollt. Die Regeln für Letrec-Bindungen und für die Liste der Case-Alternativen erzeugen hierdurch Listen, die die Elemente in einer anderen Reihenfolge enthalten als sie ursprünglich im Quelltext waren, nämlich genau umgekehrt. Laut Happy-Dokumentation ist diese Form der Rekursion effizienter.

Bemerkung 3.5.1. Aus der Grammatik allein folgt nicht, dass die Reihenfolge der Listenelemente im Syntaxbaum vertauscht wird – durch die Grammatik wird nur die Sprache definiert. Die Umkehrung resultiert aus der Verwendung des Listenkonstruktors, ist also implementierungs-spezifisch.

Die Reihenfolge spielt für die Menge der Bindungen keine Rolle. Man könnte sich höchstens um die Case-Alternativen Sorgen machen, denn zum Beispiel in Haskell haben die Case-Ausdrücke

$$\begin{aligned} & \textbf{case}\ \text{e}\ \textbf{of}\ \{\text{x:xs} \mathrel{-\!\!>} \text{a};\ \text{xs} \mathrel{-\!\!>} \text{b}\} \\ \text{und}\quad & \textbf{case}\ \text{e}\ \textbf{of}\ \{\text{xs} \mathrel{-\!\!>} \text{b};\ \text{x:xs} \mathrel{-\!\!>} \text{a}\} \end{aligned}$$

unterschiedliche Bedeutungen. In unserer Kernsprache interessieren wir uns auf der semantischen Ebene auch hier nicht für die Reihenfolge. Pattern wie `xs` oder geschachtelte Pattern sind verboten, bzw. müssen genau die Datenkonstruktoren eines Typs abgedeckt sein, so dass es ausgeschlossen ist, dass mehrere verschiedene Pattern zu einem Wert passen.

Die Funktion `parse :: String -> Expr` in `Parser.y` nimmt einen Quelltext entgegen und liefert eine `Expr`-Struktur zurück (`parse` bedient sich der Funktion `tokenize`, um die Eingabe in eine Liste von Token umzuwandeln, und wendet dann die generierte Funktion `parseTokens` darauf an), falls die lexikalische, syntaktische und die im Rest des Kapitels beschriebene *semantische Analyse* erfolgreich verlaufen, wobei alle Variablen wie angekündigt umbenannt werden und die Typen noch unbekannt sind.

```
*Parser> parse "\\x -> x"
```

```
Lam (Id "x1" "x" (1,2) NoType)
    (Var (Id "x1" "x" (1,7) NoType)) NoType
```

Der Parser kann mit dem Befehl `happy Parser.y` erzeugt werden, wobei der erzeugte Code dann in `Parser.hs` gespeichert wird.

3.6 Fehlerfreiheit

Wie wir gesehen haben, ist der Parser einigen Einschränkungen unterworfen. Andererseits ist die Happy-Grammatik-Beschreibung mit „Aktionen“ angereichert, was uns etwas mehr Handlungsfreiraum gewährt als eine bloße EBNF-Darstellung. Wenn eine Definition zu einer Token-Teilsequenz passt, ist das Ergebnis des Parsens der entsprechende Ausdruck in den geschweiften Klammern, oder genauer: dessen Wert. Da dieser Code Haskell-Ausdrücke im erzeugten Parser definiert und wir uns dadurch in der Hierarchie über den kontextfreien Sprachen befinden, können wir beim Parsen über die syntaktische Analyse hinaus gehen und den Quelltext nicht nur auf Syntaxfehler untersuchen.

Die Abwesenheit von Syntaxfehlern garantiert nicht, dass das Programm frei von sonstigen Fehlern ist. Mit anderen Worten: es gibt noch andere Fehlertypen. Es kann nicht maschinell festgestellt werden, ob ein Programm *korrekt* ist, denn dieses Problem ist unentscheidbar, zumindest dann, wenn der Korrektheitsbegriff über das bekannte *Halteproblem* definiert wird (siehe auch [21], Kapitel 2). Trotzdem können immer noch Fehler wie nicht oder nicht eindeutig definierte Variablen oder ungültige Konstruktornamen erkannt werden. Unser Ziel ist schließlich, so viele Fehler wie möglich auszumachen, und zwar bevor das Programm ausgeführt wird. Zum einen soll dadurch das Programm möglichst fehlerfrei werden, zum anderen soll zur Laufzeit so wenig Fehlerüberprüfung stattfinden wie möglich, weil das Rechenzeit kostet. Ein Teil dieser zusätzlichen Fehler wird direkt während des Parsevorgangs entdeckt – wie wir gerade festgestellt haben, kann unser Parser mehr tun als nur die Syntax zu verifizieren. Wir prüfen dennoch nicht auf alle Fehler bei der Konstruktion des Syntaxbaums und suchen die restlichen Fehler erst, nachdem der Baum komplett[8] aufgebaut wurde.

3.7 Gültige Konstruktoren

Nachdem die syntaktische Korrektheit abgehandelt ist, wollen wir uns als nächstes versichern, dass im Programm keine undefinierten Datenkonstruktoren vorkommen. Die Orte, wo sie vorkommen können, sind entweder innerhalb eines Patterns oder in einer Konstruktoranwendung. In beiden Fällen verwenden wir die Funktion `checkCon` aus

[8] Sicherlich können wir wegen der Laziness von Haskell nicht behaupten, dass der Baum an dieser Stelle tatsächlich komplett aufgebaut wurde, wir verwenden aber hin und wieder eine strikte Denkweise.

`Checks.hs`, die einen Bezeichner entgegen nimmt und ihn unverändert zurück gibt, falls es ein gültiger Konstruktorname ist. In der Produktionsregel

```
Pat    : 'Con' {(
           let (TokConId c, p) = $1
               d               = Id c c p NoType
           in  Alt (checkCon d) []
         )}

       | ...
```

für Pattern führen wir die Überprüfung beim Erzeugen der Case-Alternative durch. Analog verfahren wir beim Erzeugen einer Konstruktoranwendung. Innerhalb der Funktion `checkCon` bedienen wir uns der Funktion `typeOf` aus `Typecheck.hs`, die für unbekannte Konstruktornamen `NoType` zurück liefert. In solchen Fällen erzeugen wir eine Exception und geben den falschen Namen zusammen mit seiner Position aus.

Für den Listenkonstruktor (`:`) können wir die Alternative bzw. die Anwendung direkt erzeugen, ohne nachzusehen, ob es diesen Konstruktor „gibt“, weil wir dafür einen eigenen Tokentyp verwenden (`TokColon`) und nicht den Sammelbegriff `TokConId` wie für die anderen Konstruktoren. Hierdurch wird (`:`) schon durch die Definition in der Grammatik identifiziert.

Zusätzlich überprüfen wir, ob die Konstruktoren in den Case-Pattern eines Ausdrucks $\texttt{case}_K\ e\ \texttt{of}\ \{\ldots\}$ zum Typ K gehören, bzw. ob der Typkonstruktor K bekannt ist, und ob die Anzahl der Alternativen mit der Anzahl Datenkonstruktoren von K übereinstimmt. Die Funktionen

```
checkTC :: TName -> Pos -> TName

checkAlts2 :: TName -> [Alt] -> [Alt]

checkAlts3 :: TName -> Pos -> [Alt] -> [Alt]
```

erledigen diese Aufgaben und erzeugen Exceptions, falls etwas nicht stimmt. Die Positionsangaben werden in Fehlermeldungen mit ausgegeben.

3.8 Eindeutige Bezeichner

Im Pattern einer Case-Alternative müssen die Variablennamen, falls welche vorhanden sind, paarweise verschieden sein. Manchmal wünschen wir uns, dass es keine solche Beschränkung gibt, und können uns vorstellen, dass zum Beispiel das Pattern $(x : x : xs)$ zu Listen passen soll, in denen die ersten beiden Elemente gleich sind, wobei der Gleichheitsbegriff noch zu definieren wäre. Gewiss können wir dieses Feature nicht ohne weiteres auf

Listen von Funktionen anwenden, weil wir nicht „wissen“, wann zwei Funktionen gleich sind.

Bemerkung 3.8.1. Wir unterscheiden zwischen *gleich* und *identisch*, so dass ein anderes Element (*anders* im Sinne von *nicht identisch*, nicht *ungleich*) dennoch gleich sein kann, weil ein gleiches Element nicht zwangsweise identisch ist. In die andere Richtung gilt die Implikation natürlich, denn ein Element ist immer mindestens mit sich selbst gleich.

Solche Probleme sind nicht neu. Eine Definition der (allgemeinen) Gleichheit geht auf Leibniz zurück. Mit

> „Eadem sunt, quorum unum potest substitui alteri salva veritate“

hatte er eigentlich vielmehr die Identität im Sinn [2], dennoch lässt sich auf ähnliche Weise sagen, dass *zwei Dinge gleich sind, wenn das eine ohne Verlust der Wahrheit durch das andere ersetzt werden kann.*[9] Dieses so genannte leibnizsche Prinzip ist hier, wenn es darum geht die (Un)Gleichheit automatisch nachzuweisen, leider keine große Hilfe. Man kann auch einerseits definieren, dass zwei Funktionen gleich sind, genau dann wenn sie für gleiche Argumente immer gleiche Werte liefern[10], andererseits ist es unmöglich einen Gleichheitstest dafür anzugeben (nicht nur weil Definitionsbereiche unendlich sein können, sondern auch wegen des Halteproblems).

Je nach dem wie die Gleichheit aussehen würde, müssten zudem eventuell Ausdrücke ausgewertet werden, um auf sie testen zu können, wohingegen die Werte von x und xs in $(x : xs)$ nicht bekannt sein müssen, um festzustellen, ob das Pattern zu einer Liste passt. In unserer Sprache wollen wir uns nicht mit solchen Fragen auseinander setzen, jedenfalls in diesem Zusammenhang nicht. Pattern sollen auf Listen beliebigen Typs angewendet werden können, und geschachtelte Pattern sind ohnehin nicht zugelassen.

Wir wollen also überprüfen, ob eine Case-Alternative gegen diese Regel verstößt. Im Wesentlichen müssen wir in einer Liste zwei Elemente finden, die gleich sind, oder nachweisen, dass jedes Element der Liste ungleich jedem anderen ist. Ein naives Verfahren könnte also jedes Element mit den jeweils anderen vergleichen, womit es eine quadratische Worst-Case-Laufzeit hätte.

Um nachzuweisen, dass die Elemente paarweise verschieden sind, müssen wir nicht unbedingt alle Paare betrachten. Bei unserem Verfahren gehen wir davon aus, dass die Liste sortiert ist. Eine sortierte Liste hat die Eigenschaft, dass gleiche Elemente stets direkte Nachfolger bzw. Vorgänger voneinander sind (sonst wäre die Liste nicht sortiert), so dass wir die Liste nur einmal durchlaufen müssen, wobei wir jedes Element mit seinem Nachfolger vergleichen. Wir finden doppelte Elemente also in Linearzeit, und weil wir die Eingabe vorher sortieren müssen und die Laufzeit dadurch dominiert wird, bewegen

[9] „Ob man wie Leibniz 'dasselbe' sagt oder 'gleich', ist unerheblich“ [11].

[10] In [10] wird die Gleichheit zweier *Abbildungen* so definiert: „Zwei Abbildungen $f : X \to Y$ und $g : X \to Y$ heißen *gleich* [...], wenn $f(x) = g(x)$ für alle $x \in X$“.

wir uns insgesamt in der Sortierkomplexität. Ironischerweise kann man gerade hier ein um $(x : x : xs)$-Pattern erweitertes Haskell gut gebrauchen.

```
distinct (x:x:xs) = error ...
distinct (x:xs)   = distinct xs
...
```

Andererseits ist eine Lösung nach dem gleichen Schema auch ohne eine solche Erweiterung mühelos vorstellbar – etwa mit einem `if`-Ausdruck, der die beiden ersten Elemente der Teilliste auf Gleichheit überprüft.

Wenn es nur darum geht, festzustellen ob alle Elemente einer sortierten Liste xs voneinander verschieden sind oder nicht, kann die Lösung

```
or $ zipWith (==) xs (tail xs)
```

sein. Zuerst wird jedes Element in xs und dessen Nachfolger mit dem Gleichheitsoperator verknüpft und man bekommt eine Liste von Wahrheitswerten – falls alle Elemente in xs paarweise verschieden sind, enthält sie nur `False`'s, sonst ist mindestens ein `True` enthalten. Anschließend liefert die Funktion `or` genau dann `True`, wenn mindestens einer der Wahrheitswerte `True` ist. In der ähnlichen Lösung

```
any (uncurry (==)) $ zip xs (tail xs)
```

werden zuerst Tupel (a, b) erzeugt, wobei b jeweils der Nachfolger von a in xs ist. Danach wird festgestellt, ob ein Tupel mit zwei gleichen Elementen existiert. Die Funktion `uncurry` macht aus einer Funktion, die zwei Parameter erwartet, eine die ein Zweiertupel entgegen nimmt.

Beide Lösungen sind für unser eigentliches Problem nicht ganz geeignet. Wir wollen nicht nur wissen, ob alle Elemente der Liste eine bestimmte Eigenschaft erfüllen, sondern im Nein-Fall auch ein Element herauspicken, das das Prädikat nicht erfüllt. Eine elegante Methode ist in [6] beschrieben. Die List-Comprehension in

```
    [Just a | (a,b) <- zip xs (tail xs), a == b]
++ [Nothing]
```

liefert für eine sortierte Liste xs eine Liste, die Elemente enthält (jeweils gekapselt in einem `Just`), die in xs mehrfach vorkommen. Unser Ziel ist es, eine Funktion

$$\texttt{distinct}\ xs \begin{cases} \texttt{Just}\ a, & \text{falls es ein } a \in xs \text{ gibt, das mehrfach vorkommt} \\ \texttt{Nothing}, & \text{sonst} \end{cases}$$

zu haben, die auf einer unsortierten Liste operiert, die Elemente beliebigen Typs enthält; die einzige Einschränkung soll sein, dass die Elemente mit `compare` miteinander verglichen werden können, also dass deren Typ der Typklasse `Ord` angehört, sonst können wir die Liste nicht sortieren und unser Verfahren nicht anwenden.

Zur Vervollständigung muss nicht viel getan werden. Zuerst sortieren wir die Eingabeliste (das Sortieren übernimmt die Bibliotheksfunktion `sort`), wenden die Listcomprehension darauf an und hängen ein `Nothing` an. Der Kopf der Ergebnisliste ist dann gerade (`Just` a), wobei a mehrfach vorkommt, falls es ein solches a gibt, oder die List-Comprehension erzeugt eine leere Liste, so dass der Kopf `Nothing` ist. Wir müssen also einfach den Kopf der Ergebnisliste zurück liefern. Der Name `distinct` ist jetzt vielleicht etwas weniger treffend.

Es gibt etwas bessere Verfahren. Man kann das Problem auch mit Hilfe eines Tries oder einer Hashtabelle lösen. Dabei werden die Listenelemente eins nach dem anderen in ein Wörterbuch eingefügt, so dass gleiche Schlüssel zwangsweise auf dem selben Speicherplatz landen und das Verfahren im Worst-Case genauso viele Einfügeoperationen benötigt, wie es Listenelemente gibt.

Dafür dass in einem Pattern höchstens zwei Variablennamen vorkommen können, mag unsere Herangehensweise etwas übertrieben erscheinen. Um festzustellen, ob zwei Werte ungleich sind, muss nur geprüft, ob sie nicht gleich sind. Es gibt ein weiteres Teilproblem, bei dem wir in einer Liste nach doppelten Elementen suchen müssen. Die durch ein Letrec eingeführten Bezeichner müssen nämlich auch verschieden sein. Die Anzahl dieser Namen ist nicht begrenzt.

Sowohl bei den Pattern als auch bei den Letrec-Bindungen speichern wir die Variablennamen als Listen von `Id`, die wir an `distinct` übergeben wollen. Zwar ist unsere Implementierung *poly*morph und kommt daher mit *vielen* Listentypen zurecht, nämlich mit genau denen, auf deren Elementen eine Ordnungsrelation definiert ist, kann aber gerade deswegen (noch) nicht auf [`Id`] angewendet werden. Man kann natürlich die Namen erst mit zum Beispiel

```
map (\(Id _ v _ _) -> v) xs
```

extrahieren und die Ergebnisliste übergeben, aber dann gibt es im Fehlerfall keine Information über die Position des Bezeichners im Quelltext mehr. Man kann auch die Implementierung von `distinct` spezieller machen: eine monomorphe Funktion definieren, die *nur* auf [`Id`] angewendet werden kann, und genau „weiß“, wie solche Listen sortiert werden müssen (etwa durch Verwendung von `sortBy` statt `sort`) und wie Elemente zu vergleichen sind. Wir entscheiden uns für einen anderen Weg. Um `distinct` mitzuteilen, wann ein Element ungleich einem anderen ist und wann das eine ein lexikographischer Nachfolger des anderen ist, erklären wir `Id` zur Instanz der Typklassen `Eq` und `Ord` und geben Implementierungen für die Funktionen (`==`) und `compare` an.

```
instance Eq Id where
    v1 == v2  =  getName v1 == getName v2

instance Ord Id where
    compare v1 v2 = compare (getName v1) (getName v2)
```

Aus diesem Grund haben wir die relativ kleine Datenstruktur `Id` nicht etwa als Typsynonym für ein Vierertupel definiert, sondern als eigenständigen Typ, denn Typsynonyme können keine Instanzen von Typklassen sein (nicht wenn es dem Haskell-98-Standard entsprechen soll). Zwei `Id`-Elemente sollen genau dann gleich sein, wenn die gekapselten Namen gleich sind, genauer: die Namen, die die Funktion `getName` liefert (eine überladene Funktion in `Expr.hs`, die Namen extrahiert). In der Implementierung von `compare` vergleichen wir ebenfalls diese Namen. In unserer Happy-Grammatik sind die Definitionen für Case-Alternativen und für Letrecs mit Aufrufen von `checkPat` und `checkBinds` verbunden (siehe Produktionsregeln auf Seite 33), die letztendlich die Verschiedenheit der Variablennamen mit Hilfe von `distinct` testen, und im Fehlerfall eine passende Exception generieren.

In Abschnitt 3.5 wurde verlangt, dass die Case-Alternativen in einem Case-Ausdruck eindeutig sind, so dass keine zwei Pattern den gleichen Konstruktor enthalten dürfen. Auch hier verwenden wir das Sortierte-Liste-Verfahren. Beim Parsen eines Case-Ausdrucks rufen wir die Funktion `checkAlts` auf. Es werden die Konstruktornamen aus der Alternativen-Liste extrahiert und die so entstandene Liste von Bezeichnern (`Id`) wird an `distinct` übergeben. Im Fehlerfall wird wie gewohnt eine Exception generiert. Hier kann man statt einer Fehlermeldung nur eine Warnung ausgeben, wie das zum Beispiel in Haskell der Fall ist. In Haskell spielt die Reihenfolge der Alternativen eine Rolle und die Konvention ist, dass die erste Alternative ausgewählt wird, die zutrifft. Für uns ist die Reihenfolge unwichtig, folglich können wir keine deterministische Regel für die Alternativen-Auswahl angeben und es ist sinnvoll, den Parsevorgang mit einer Fehlermeldung abzubrechen, falls das gleiche Pattern mehrmals benutzt wird.

3.9 Eingefangene Variablen

Im nächsten Schritt wollen wir alle Variablen in einem Programm umbenennen, so dass kein Name mehrfach verwendet wird (im Sinne von: für verschiedene Werte nicht). Gleichzeitig wollen wir feststellen, ob das Programm ein geschlossener Ausdruck ist. Das ist gleichbedeutend damit, dass es keine undefinierten Variablen gibt.

Durch Umbenennungen kann unter anderem verhindert werden, dass (in einem unmittelbaren Auswertungsschritt) Variablen „eingefangen“ werden. Das Problem wird *variable capture* oder auch *name capture* genannt. Beispielsweise wird beim Auswertungsschritt (β-Reduktion)

$$\frac{(\lambda x.\lambda y.(x\ y))\ \underline{y}}{\lambda y.(\underline{y}\ y)}$$

das unterstrichene y eingefangen bzw. gebunden, und es entsteht eine Funktion, die ein Argument erwartet, welches sie auf dasselbe anwendet. Stattdessen hätten wir eine Funktion erhalten müssen, die das Argument an eine Funktion übergibt, die nicht zwangsweise das Argument selbst ist.

Bemerkung 3.9.1. Wir vertreten die Sichtweise, dass es innerhalb eines Ausdrucks nur eine Variable mit dem gleichen Namen geben kann (nicht zwei verschiedene Variablen, die gleich heißen), es aber vom Kontext abhängt, für welchen Wert der Name steht. Sei $t = (\lambda y.y)\ y$, dann ist $\mathcal{GV}(t) = \{y\}$ die Menge der gebundenen Variablen (die Menge der Variablen, die gebunden vorkommen) und die der freien ist $\mathcal{FV}(t) = \{y\}$. Nehmen wir an, es gibt zwei verschiedene y's (ein freies und ein gebundenes). Die Annahme steht im Widerspruch zu $\mathcal{GV}(t) \cup \mathcal{FV}(t) = \{y\}$ (in der Vereinigung fallen die y's zusammen).

Je nachdem in welchem Geltungsbereich (*Skopus*) y auftritt, bezeichnet es im allgemeinen nicht denselben Wert. Der Ausdruck vor der Reduktion hat einen anderen Typ als danach (bzw. hat der Ausdruck danach keinen Typ, zumindest nicht in unserem Typsystem, siehe Kapitel 4). Die Geltungsbereiche können beliebig verschachtelt sein. Das ist nicht in allen Programmiersprachen so, hier aber schon. Durch die Reduktion gerät „das freie“ y in einen anderen Geltungsbereich hinein, wo es nicht mehr seinen ursprünglichen Wert bezeichnet (sondern den Parameter der Funktion). Dieses Problem tritt im Reduktionsschritt

$$\frac{(\lambda x.\lambda y.(x\ y))\ z}{\lambda y.(z\ y)}$$

nicht auf, wenn wir die Namen also skopus-übergreifend eindeutig wählen. Es wäre unzweckmäßig, zu fordern, dass bereits der Quelltext diese Eigenschaft erfüllen soll. Außerdem müsste dann auf diese Eigenschaft geprüft werden, was der Umbenennung von Variablen fast gleich käme. Es soll reichen, wenn das Programm an sich nicht mehrdeutig ist – zum Beispiel, dass durch ein Letrec die gleiche Variable nicht mehrmals definiert wird.

Interessanterweise genügt es nicht, die Variablen ein mal umzubenennen, damit bei mehreren Auswertungsschritten hintereinander die Gefahr des Einfangens sich nicht wiederholt. Das Problem lässt sich generell vermeiden, wenn bei *jedem* β-Reduktionsschritt sichergestellt wird, dass die freien Variablen des Arguments verschieden sind von den gebundenen Variablen der Abstraktion, die im Rumpf der Abstraktion vorkommen, so dass in jedem Reduktionsschritt eventuell aufs Neue umbenannt werden muss (α-Konversion).

Das Einfangen kann auch durch eine bestimmte Auswertungsreihenfolge/-vorschrift verhindert werden (Normalordnungs-Reduktion) – wenn man nicht zulässt, dass überall beliebig reduziert werden darf, wo reduziert werden kann (siehe auch [19], Abschnitt 11.3.2; in [7] wird ein Lambda-Kalkül vorgestellt, in dem kein Einfangen von Variablen stattfinden kann).

3.10 Umbenennungen und Geschlossenheit

So drängt sich die Frage auf, warum wir dann eine Umbenennenung auf Variablen brauchen. Dafür gibt es verschiedene Gründe. Es gibt beispielsweise ein weiteres Detail, auf

das beim Auswerten geachtet werden muss: *name clash*. Dabei werden Bindungen „verdeckt“. Aus diesem Grund ist die β-Reduktion so definiert, dass durch einen Auswertungsschritt $(\lambda x.s)\ t \xrightarrow{\beta} s[t/x]$ nur *freie* Vorkommen von x in s ersetzt werden dürfen. Zum Beispiel ist die Reduktion

$$\frac{(\lambda x.(\lambda x.x))\ \texttt{True}}{\lambda x.\texttt{True}}$$

per Definition falsch. Benennt man die Variablen um, so muss beim Ersetzen nicht darauf geachtet werden, ob ein Vorkommen von x im Rumpf frei ist oder nicht, denn nach der Umbenennung gibt es dort nur freie Vorkommen von x.

Bemerkung 3.10.1. Ein Ausdruck muss nicht geschlossen sein, damit er reduziert werden kann. Im Beispiel oben können wir annehmen, dass der Ausdruck ein Teilausdruck eines geschlossenen Ausdrucks ist, oder wir können annehmen, dass das der Gesamtausdruck ist. Im Allgemeinen ist es nicht so, dass für ein Programm, das freie Variablen enthält, eine Folge von Reduktionen hin zu einem „sinnvollen“ Wert existiert, obwohl es manchmal so sein kann. Beispiel:

$$\frac{(\lambda x.\texttt{True})\ y}{\texttt{True}}$$

Die Variable y wird nicht gebraucht, und wenn man das Argument nicht vor seiner Übergabe auswertet, stört es nicht, dass der Wert unbekannt ist. Für die Auswertung von $((\lambda x.x)\ y)$ wollen wir aber schon wissen, was y ist. Abgesehen davon bereitet es uns Probleme, bei der Typberechnung den Typ einer unbekannten Variablen zu bestimmen.

Es gibt weitere Situationen, in denen umbenannt werden muss. Eine Umbenennungs-Operation benötigt man auch, wenn Ausdrücke (etwa zur Optimierung) in andere transformiert werden, wobei die Bedeutung erhalten bleiben muss. Siehe zum Beispiel [18], Abschnitt 6.2. Dort wird

$$\lambda y.\texttt{let}\ y = x * x\ \texttt{in}\ y \quad \text{nach} \quad \texttt{let}\ y = x * x\ \texttt{in}\ \lambda y.y$$

transformiert, um zu zeigen, dass man (bei Verwendung der dort eingeführten Umformungsregeln) keinen äquivalenten Ausdruck erhält, solange man nicht umbenennt.

Schließlich kann die so genannte α-Äquivalenz nachgewiesen werden, indem Variablen umbenannt werden, was aus der Definition folgt: zwei Terme sind α-äquivalent, wenn sie bis auf Umbenennung der gebundenen Variablen gleich sind.

Nachdem der fertige Syntaxbaum vorliegt, können wir die Variablen umbenennen, indem wir den Baum traversieren, wobei einige Bindungsregeln beachtet werden müssen. Die Definition 1.3.2 auf Seite 13 erklärt, wie die Menge der gebundenen Variablen für einen Ausdruck bestimmt werden kann. Auf diese Weise erfahren wir aber noch nicht, wo genau die Variablen gebunden werden, bzw. wo die Geltungsbereiche sind. Ohne diese Information können wir keine korrekte Umbenennung vornehmen. Es hängt vom

Geltungsbereich ab, ob zwei Vorkommen des gleichen Namens den gleichen Wert repräsentieren und ob die Namen nach der Umbenennung weiterhin gleich sein werden oder verschieden. Abgesehen davon, kann mit Hilfe der Menge $\mathcal{GV}(e)$ nicht argumentiert werden, dass ein Ausdruck e geschlossen ist, höchstens dass er es nicht ist. Deshalb ergänzen wir:

Definition 3.10.1. Der *Geltungsbereich* von Variablen (der Unterausdruck, in dem sie gültig sind) ist für die einzelnen Binder (λ, `letrec`, Pattern) definiert durch

Term	Geltungsbereiche
$\lambda x.t$	x gilt in t
`letrec` $x_1 = s_1, \ldots,$ $x_n = s_n$ `in` t	$x_1, \ldots, x_n$ gelten in $s_1, \ldots, s_n$ und in t
`case_`$\ldots$ s `of {`$\ldots,$ $D\ x_1\ \ldots\ x_n \to t,\ \ldots$`}`	$x_1, \ldots, x_n$ gelten in t

Bemerkung 3.10.2. Man beachte den Unterschied zwischen `letrec` (*let recursively*) und (dem in unserer Sprache nicht vorhandenen) `let`: in `(let` $x_1 = s_1, \ldots, x_n = s_n$ `in` t`)` gelten die Variablen $x_1, \ldots, x_n$ *nur* in t, so dass ein x_i in den s_j frei sein kann.

Nachdem die Geltungsbereiche nun festgelegt wurden, wissen wir, wie die Umbenennung vorzunehmen ist.

Definition 3.10.2. Sei R eine *Namensabbildung*, die für einen Variablennamen einen anderen liefert und $\mathrm{Dom}(R)$ die Menge der Variablen, die R auf andere abbildet. Sei $x \mapsto y$ ein Argument-Wert-Paar[11], dann ist

$$R \cup \{x \mapsto y\}$$

ist die *Erweiterung* von R um ein neues Argument-Wert-Paar, oder es ist eine *Aktualisierung* eines alten Paares, falls $x \in \mathrm{Dom}(R)$. Der $\cup$-Operator ist nicht kommutativ. Statt $R \cup \{x_1 \mapsto y_1\} \cup \ldots \cup \{x_n \mapsto y_n\}$ schreiben wir $R \cup \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$. Die rekursive Operation

$$\cdot\langle\cdot\rangle$$

ist eine Ersetzung auf Ausdrücken. Für jeden neu eingeführten Variablennamen y, y_i gilt, dass er nicht schon vorher verwendet wurde.

[11] Eine Abbildung ist für uns an dieser Stelle eine Menge von Argument-Wert-Paaren, keine Rechenvorschrift.

$$
\begin{array}{rcl}
x\langle R\rangle & := & \begin{cases} R(x)\ , & \text{falls } x \in \mathrm{Dom}(R) \\ \text{Fehler}\ , & \text{sonst} \end{cases} \\
(\lambda x.t)\langle R\rangle & := & \lambda y.t\langle R \cup \{x \mapsto y\}\rangle \\
(s\ t)\langle R\rangle & := & s\langle R\rangle\ t\langle R\rangle \\
(\texttt{seq}\ s\ t)\langle R\rangle & := & \texttt{seq}\ s\langle R\rangle\ t\langle R\rangle \\
(\texttt{amb}\ s\ t)\langle R\rangle & := & \texttt{amb}\ s\langle R\rangle\ t\langle R\rangle \\
(D\ s_1\ \ldots\ s_n)\langle R\rangle & := & D\ s_1\langle R\rangle\ \ldots\ s_n\langle R\rangle \\
\left(\begin{array}{ll}\texttt{letrec} & x_1 = s_1, \ldots, \\ & x_n = s_n\ \texttt{in}\ t\end{array}\right)\langle R\rangle & := & \left(\begin{array}{ll}\texttt{letrec} & y_1 = s_1\langle R'\rangle, \ldots, \\ & y_n = s_n\langle R'\rangle\ \texttt{in}\ t\langle R'\rangle\end{array}\right) \\
 & & \text{mit } R' = R \cup \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}
\end{array}
$$

$$
\left(\begin{array}{l}\texttt{case}_K\ s\ \texttt{of}\ \texttt{\{} \\ \quad \ldots, \\ \quad D\ x_1\ \ldots\ x_n \to t \\ \quad \ldots\ \texttt{\}}\end{array}\right)\langle R\rangle := \left(\begin{array}{l}\texttt{case}_K\ s\langle R\rangle\ \texttt{of}\ \texttt{\{} \\ \quad \ldots, \\ \quad D\ y_1\ \ldots\ y_n \to \\ \qquad t\langle R \cup \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}\rangle \\ \quad \ldots\ \texttt{\}}\end{array}\right)
$$

Wenn t ein geschlossener Programmausdruck ist, dann ist $t\langle\emptyset\rangle$ ein Ausdruck, in dem alle Variablen umbenannt sind. Die Namensabbildung ist am Anfang also leer.

Die Einführung der Notationen $R \cup \{\ldots\}$ und $\cdot\langle\cdot\rangle$ soll helfen, den Umbenennungsprozess kompakter darzustellen. $t\langle R\rangle$ bedeutet, dass R genau die Variablen beinhaltet, die in t gelten und weiter außen gebunden wurden, und dass in t Umbenennungen stattfinden, oder genauer, dass die freien Variablen in t (irgendwann, wenn der Basisfall der Rekursion eintritt) gemäß R umbenannt werden, und dass in jedem Rekursionsschritt für jede äußerste Bindung ein neues Element in R eingefügt wird.

Wir können anhand von R in $t\langle R\rangle$ nicht sofort schließen, wie t nach der Umbenennung aussehen wird. Zum Beispiel bedeutet $(\lambda z.\lambda x.x)\langle\{x \mapsto y\}\rangle$ nicht etwa, dass x überall durch y ersetzt werden soll. Ein möglicher Umbenennungsvorgang

$$
\begin{array}{rcl}
(\lambda z.\lambda x.x)\langle\{x \mapsto y\}\rangle & \to & \ldots \\
\to\ (\ \lambda z'.\lambda x'.x\langle\{x \mapsto x', z \mapsto z'\}\rangle\ ) & \to & (\lambda z'.\lambda x'.x')
\end{array}
$$

zeigt, dass der Name y im Ergebnisausdruck überhaupt nicht vorkommt. Der Grund ist: sobald in einem Geltungsbereich ein Variablenname gebunden wird, der schon weiter außen gebunden wurde, muss dafür ein frischer Name eingesetzt werden. Wir können daher nur die neuen Namen der *freien* Variablen in einem Ausdruck aus R ableiten. Für eine freie Variable wird kein anderer Name mehr eingeführt, weil das nur durch eine Bindung passieren kann.

Wenn wir uns durch den Syntaxbaum bewegen, hat eine gegebene Abbildung R eine nützliche Eigenschaft, nämlich dass falls darin kein Element $(x \mapsto \ldots)$ existiert, die

Variable x im gerade besuchten Geltungsbereich nicht gilt. Von diesem Argument wird in der ersten Regel in der Definition der Umbenennungs-Operation Gebrauch gemacht. Stoßen wir auf eine Variable, können wir, indem wir R betrachten, herausfinden, ob sie weiter außen gebunden wurde oder nicht. Falls nein, kann die Umbenennung mit einem Fehler abgebrochen werden, da wir ein solches Programm nicht als ausführbar betrachten. Prinzipiell sind wir in der Lage, alle undefinierten Variablen zu finden, also die Menge der freien Variablen zu bestimmen. Manche Compiler (oder Interpreter) geben gleich mehrere Fehlermeldungen aus, was die Fehlersuche/-beseitigung benutzerfreundlicher machen soll, weil man nicht für jede undefinierte Variable den Kompiliervorgang neu starten muss.

Es bleibt zu klären, wie wir in jedem Umbenennungsschritt an die frischen Variablennamen herankommen. Als erstes benötigen wir eine Liste mit Namen. Die Namen in der Liste müssen paarweise verschieden sein, und die Liste muss unendlich sein, da wir a priori nicht wissen, wie viele Variablennamen benötigt werden. Die simple List-Comprehension

```
['#' : show i | i <- [1..]]
```

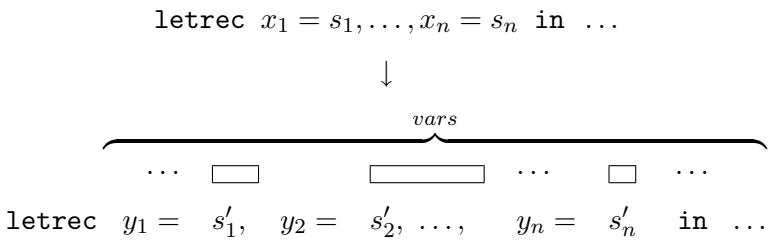
liefert die Liste `["#1", "#2", "#3", ...]` (wir nennen sie *newvars*), welche die gewünschte Eigenschaft besitzt. Da die natürlichen Zahlen paarweise verschieden sind, können wir davon ausgehen, dass auch eine Konkatenation von `"#"` und der String-Darstellung einer Zahl verschieden ist von einer Konkatenation mit einer anderen Zahl als String. Das Zeichen `#` darf im Quelltext nicht vorkommen; der Lexer bricht die Verarbeitung ab, falls ein nicht erlaubtes Zeichen gelesen wird. Somit sind alle neuen Variablennamen verschieden von den Namen, die in einem syntaktisch korrekten Quelltext vorkommen können.

In jedem Umbenennungsschritt bekommen wir eine Liste – nennen wir sie *vars* – mit paarweise verschiedenen Variablennamen als Parameter übergeben (ganz am Anfang ist es *newvars*), entfernen Variablennamen daraus, falls wir welche benötigen, und merken uns die Ergebnisliste, die dann im nächsten Schritt verwendet wird; arbeiten also mit dem veränderten *vars* weiter. Dabei kann jeder weitere Schritt seinerseits Variablennamen verbrauchen. Wir bekommen den Rest im Rückgabewert, den wir entweder weiter verwenden, falls wir weitere Umbenennungen auf dieser Rekursionsebene durchführen müssen, oder geben ihn zurück.

Besonders einfach ist die Namensbeschaffung, wenn wir genau einen neuen Namen benötigen, wie das bei einer Abstraktion ($\lambda x.t$) der Fall ist. Wir müssen die Bindung umbenennen und das Verfahren auf den Geltungsbereich t rekursiv anwenden. Dabei entfernen wir den ersten Namen aus der aktuellen Namensliste, und übergeben den Rest an den nächsten Rekursionsschritt.

$$\begin{array}{c} [\, y_1,\ y_2,\ y_3,\ \ldots\,] \\ \updownarrow \\ \lambda\, x\, .\, t \end{array}$$

Abbildung 3.10.1: Letrec-Umbenennungen



Der Kopf von *vars* überlagert sozusagen die Lambda-Bindung und ein Teil vom Rest überlagert t. Es ist klar, wie die Liste aufgeteilt werden muss. Anders sieht es aus, wenn wir es zum Beispiel mit einer Menge von Letrec-Bindungen zu tun haben. Zuerst müssen genauso viele Namen aus der Namensliste entfernt werden wie es Bindungen gibt. Nachdem die definierten Variablen an den Bindungen umbenannt wurden, müssen wir in die entsprechenden Geltungsbereiche absteigen. Wir wissen nicht, wie viele Namen der Umbenennungsvorgang pro Geltungsbereich verbrauchen wird. Die unterschiedlich langen Balken (die Teillisten von *vars* darstellen) in Abbildung 3.10.1 sollen diese Ungewissheit veranschaulichen. Ein Balken über einem s'_i bedeutet, dass die Umbenennungen in s_i eine entsprechende unbestimmte Menge von Variablennamen verbrauchen. Wir iterieren über die Geltungsbereiche und übergeben in jedem Iterationsschritt die jeweils übrig gebliebenen Namen an den nächsten Schritt. Insgesamt lässt sich die Umbenennung eines Letrec-Ausdrucks grob in die Phasen

1. für jede Bindung in (`letrec` $x_1 = s_1, \ldots, x_n = s_n$ `in` t) trenne die linke von rechten Seite,
2. benenne $x_1, \ldots, x_n$ um, wende die Umbenennung auf $s_1, \ldots, s_n$ und t an, und
3. füge die Teile wieder zusammen

unterteilen. In der zweiten Phase liegt uns eine Liste von Ausdrücken vor, die wir sequentiell abarbeiten. Nachdem wir die Variablen in s_i umbenannt haben, fahren wir mit s_{i+1} und einer neuen Namensliste fort. Das gleiche Teilproblem haben wir, wenn die Variablen in einer Konstruktoranwendung umbenannt werden sollen; die Liste der aktuellen Parameter ist eine Liste[12] von Ausdrücken.

Es gibt ausgeklügeltere Methoden der Namensbeschaffung. Die aktuelle Namensliste muss nicht immer hin und her gereicht werden. Zum Beispiel kann die Eindeutigkeit

[12]Die Argumentliste ist direkt gegeben. Wegen der Forderung, dass Konstruktoren nicht ungesättigt vorkommen dürfen, bietet es sich an, die Parameter als Liste zu speichern und nicht als eine geschachtelte Anwendung. Diese Umsetzung zieht leider zusätzliche Fallunterscheidungen nach sich, denn allgemeine Funktionsanwendungen (`App` ...) gibt es natürlich auch noch.

der Namen jedes mal, wenn der Rekursionsbaum sich aufspaltet, dadurch erreicht werden, dass ein eindeutiger Präfix den Namen vorangestellt wird (ein weiteres Verfahren ist in [19], Abschnitt 9.6 beschrieben). Das hat den Vorteil, dass man nicht warten muss, bis ein Teilbaum im Syntaxbaum fertig umbenannt wurde, bzw. bis die restlichen Namen zurück geliefert wurden.

Abschließend wollen wir beides zusammen betrachten: die Manipulation einer Namensabbildung und die Beschaffung von frischen Variablennamen. Im Wesentlichen ist es eine Erweiterung der Operation $\cdot\langle\cdot\rangle$ um einen Parameter, nämlich um eine Liste von Namen. Wir nennen diese neue Operation `rename` und wollen sie anhand des Beispiels der Verarbeitung von Case-Ausdrücken umreißen. Für die Speicherung von Paaren

$$(\mathit{alter\ Variablenname},\ \mathit{neuer\ Variablenname})$$

verwenden wir das Bibliotheksmodul `Data.Map` mit der darin enthaltenen effizienten Implementierung eines Wörterbuchs. Eine Namensabbildung ist also ein solches. Unsere Eingabe ist

- ein Wörterbuch *bmap*, das die weiter außen gebundenen Variablennamen zusammen mit den dafür neu eingesetzten Namen enthält,
- ein Ausdruck `case_...` s `of {`$D_1\ x_{1,1}\ \ldots\ x_{1,k} \to t_1,\ \ldots,\ D_n\ x_{n,1}\ \ldots\ x_{n,m} \to t_n$`}` und
- eine Liste $vars$ mit paarweise verschiedenen Variablennamen.

Als erstes rufen wir `rename` auf *bmap*, s und $vars$ rekursiv auf und bekommen einen neuen Ausdruck s' mit umbenannten Variablen und eine neue Liste $vars'$ mit übrig gebliebenen Variablennamen zurück. Die Bindungen in s haben keine Auswirkung auf den Rest des Case-Ausdrucks, so dass wir uns für eventuelle Veränderungen in *bmap* nicht interessieren und weiterhin mit dem *bmap* arbeiten, das wir als Argument erhalten haben. Für jede Case-Alternative $(D\ x_1\ \ldots\ x_n \to t)$ gilt, dass die neu eingeführten $x_1, \ldots, x_n$ nur in t gelten und nicht etwa in allen anderen Alternativen auch. Für jede Alternative erweitern wir:

$$bmap' = bmap \cup \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$$

und rufen die Umbenennung mit $bmap'$ und der aktuellen Variablennamensliste rekursiv auf t auf. Als Ergebnis erhalten wir einen neuen Ausdruck und eine neue Namensliste, die wir für die nächste Alternative verwenden. Wir starten also jedes mal mit dem alten *bmap*, müssen aber die übrig gebliebenen Variablennamen immer weiter reichen. Es ist auch sonst immer so, dass wir *bmap* (verändert oder nicht) zwar als Parameter an weitere Aufrufe von `rename` übergeben, die dadurch stattfindenden Veränderungen des Wörterbuchs auf der aktuellen Rekursionsebene für uns aber nicht relevant sind; es sind genau die Namen wichtig, die sich *jetzt* in *bmap* befinden.

In der Definition 3.10.2 wurde darauf hingewiesen, dass es nicht unwichtig ist, in welcher Reihenfolge Elemente zu einer Namensabbildung hinzugefügt werden; dasselbe gilt für *bmap*. Falls der gleiche Schlüssel mehrmals eingefügt wurde, enthält das Wörterbuch das jüngste Schlüssel-Wert-Paar. Das kann höchstens dann Probleme bereiten, wenn durch ein Pattern (oder ein Letrec) mehrere Variablen gebunden werden, die gleich heißen, was aber an sich schon ein Problem ist und daher durch vorhergehende Überprüfungen abgefangen wird.

Die im vorigen Abschnitt beschriebenen Name-Clash-Konflikte können nur dann auftreten, wenn in einem Geltungsbereich eine Variable gebunden wird, die schon weiter außen gebunden wurde. Als Konsequenz muss eine Variable nicht umbenannt werden, wenn sie eindeutig ist – benennt man Variablen um, obwohl die Namen eindeutig sind, hat man nichts davon, außer dass alle Variablen anders heißen. Nehmen wir an, wir erweitern das Verfahren, wobei wir die Umbenennungen in $(\lambda x.t)$ so definieren:

$$\begin{aligned}(\lambda x.t)\langle R\rangle \quad &:= \quad \lambda y.t\langle R \cup \{x \mapsto y\}\rangle \\ &\qquad \text{mit } y = \begin{cases} \text{neuer Name}\,, & \text{falls } x \in \mathrm{Dom}(R) \\ x\,, & \text{sonst} \end{cases}\end{aligned}$$

Dieses Verfahren benötigt eine zusätzliche[13] Lookup-Operation im Wörterbuch, um festzustellen, ob der Name bereits gebunden wurde. Man beachte, dass falls keine Umbenennung stattfindet, trotzdem ein neuer Eintrag $(x \mapsto x)$ in R eingefügt wird, weil R genau die gebundenen Variablen als Definitionsbereich haben muss. Da wir die alten Namen sowieso mit speichern, bringt eine solche Fallunterscheidung keine besonderen Vorteile mit sich. Wir benennen daher alle Variablen um. Die Funktion

```
renameVars :: Expr -> [Name] -> (Expr, [Name])
```

im Modul `Checks.hs` nimmt einen Syntaxbaum und eine unendliche Liste mit Variablennamen entgegen und liefert einen neuen Syntaxbaum, in dem alle Variablen umbenannt sind, und die Liste mit unverbrauchten Variablennamen in einem Tupel zurück.

Ein Programm, das soweit fehlerfrei ist, dass es u. a. keine freien Variablen enthält (ausgenommen die vordefinierten Schlüsselwörter), wird in [19] als „not 'trivially' malformed“ bezeichnet. Wir schließen die semantische Analyse mit der Überprüfung auf Geschlossenheit des Programmausdrucks und der gleichzeitigen Umbenennung der Variablen ab.

[13] Wir bekommen hier keinen Zeiger auf eine (freie) Speicherposition oder dergleichen, so dass der Aufwand nicht ohne weiteres weg optimiert werden kann.

Kapitel 4

Typisierung

Nachdem uns ein Programm in Form einer „bequemen“ Datenstruktur vorliegt, wir uns vergewissert haben, dass es keine Trivialfehler enthält, und sonstige Bedingungen erfüllt sind, wollen wir die Typen der Unterausdrücke berechnen. Am Ende dieses Kapitels erhalten wir ein Verfahren, das in der Lage ist sämtliche Sprachkonstrukte der Sprache L_{PLC-T} zu typisieren.

In einigen Abschnitten verwenden wir Baumdarstellungen von Ausdrücken. Diese Bäume wurden mit Hilfe der Funktion `showTree` (Modul `SrcGen.hs`) erzeugt.

4.1 Darstellung von Typausdrücken

Genau wie es bei den Programmausdrücken war, benötigen wir jetzt eine Datenstruktur, die Typen speichert. Typausdrücke sind nicht Teil des Quelltextes, den wir als Eingabe bekommen, sind aber Teil dessen, was wir daraus erzeugen – falls für die Eingabe ein Typ hergeleitet werden konnte, sind im Ausgabeausdruck die (meisten) Teilausdrücke mit Typen markiert. Wir verwenden die Datenstruktur `Expr` um die Eingabe zu speichern und gleichzeitig auch für die Ausgabe. Aus diesem Grund sind dort Plätze für Typen vorgesehen. Der Datentyp `Type` im Modul `Type.hs` im Unterverzeichnis `types` speichert Typen.

```
type TName = String

data Type
        = TVar TName
        | TCon TName [Type]
        | ForAll [TName] Type
        | NoType
    deriving (Eq)
```

Der Konstruktor `NoType` drückt aus, dass der Typ unbekannt ist. Wir erinnern uns: beim Parsen werden die Typen erstmal auf „unbekannt“ gesetzt. Ein Typ ist eine Typvariable, ein Typkonstruktor mit Parametern (die wiederum Typen sind) oder ein Quantifizierter Typ, wobei wir die Menge der quantifizierten Variablen als Liste speichern. Positionsmarkierungen gibt es diesmal nicht (da ja die Typen nicht aus dem Eingabequelltext kommen). Zum Beispiel ist der Typ $\forall a, b, c.a \to b \to c$

```
ForAll ["a","b","c"] (
    TCon "->" [
        TVar "a",TCon "->" [
            TVar "b",TVar "c"
        ]
    ]
)
```

als `Type`. Weil die Reihenfolge der Variablen beim Quantor unwesentlich ist, gibt es noch weitere $3! - 1$ Permutationen. Wir erklären `Type` zu einer Instanz von `Eq`, um Typen auf Gleichheit überprüfen zu können.

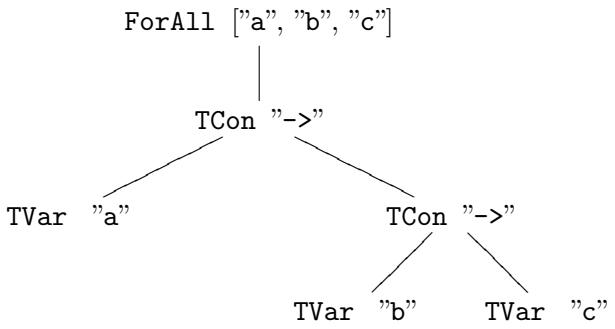
- Zwei Typausdrücke a und b sind gleich, gdw. $a = b$, wobei a und b Typvariablen sind.
- Zwei Typausdrücke $K_1\ S_1 \ldots S_n$ und $K_2\ T_1 \ldots T_m$ sind gleich, gdw. $K_1 = K_2$ und $S_1 = T_1, \ldots, S_n = T_m$.
- Zwei Typausdrücke $\forall \mathcal{X}.T_1$ und $\forall \mathcal{Y}.T_2$ sind gleich, gdw. $\mathcal{X} = \mathcal{Y}$ und $T_1 = T_2$.

Listen sind geordnet, womit die Implementierung des Gleichheitstests von der Gleichheitsdefinition abweicht, denn zum Beispiel die Listen `["a","b"]` und `["b","a"]` sind nicht gleich, obwohl sie die die gleiche Menge von Variablen speichern. Wie man leicht sieht, sind Typausdrücke auch baumartig. Die Abbildung 4.1.1 zeigt den Typ $\forall a, b, c.a \to b \to c$ als Baum.

Dadurch dass der Datenkonstruktor `ForAll` zu `Type` gehört, sind verbotene Typen wie $(\forall \ldots . \forall \ldots)$ möglich; ein Quantor darf nur ganz außen stehen. Der Compiler wird uns auf solche Fehler nicht aufmerksam machen. Das heißt, wir müssen selbst darauf achten, dass derartige Anomalien nicht vorkommen.

Die Schreibweise in Haskell-Syntax ist bereits für mittelgroße Typausdrücke etwas unübersichtlich und deshalb fehleranfällig. Wir haben daher einen Parser für Typen, den wir gelegentlich verwenden, um bekannte Typen, wie etwa den von `seq`, in der Implementierung einfacher aufschreiben zu können. Die Funktion `parseType` in `TParser.hs` im Unterverzeichnis `parser` parst Typen in gewohnter Schreibweise. Der Parser ist ähnlich implementiert wie der für Programmausdrücke. Es gibt einen Lexer und eine Happy-Datei.

Abbildung 4.1.1: Ein Typ-Syntaxbaum




```
*TParser> parseType "forall a b . a -> b -> b"
ForAll ["b","a"] (TCon "->" [TVar "a",TCon "->" [TVar "b",TVar "b"]])
```

4.2 Substitutionen und Unifikation

Während der Typberechnung müssen wir in einigen Fällen Typen „gleich machen“. Der Typ eines Ausdrucks, der eine Funktionsanwendung ist (wobei der Typ der Funktion und der des Arguments bekannt sind) lässt sich bestimmen, indem der Typ des formalen Parameters mit dem Typ des aktuellen Parameters verglichen wird,

- falls die beiden gleich sind, ist der Typ der Anwendung, der Typ, auf den die Funktion abbildet,
- falls nein, liegt ein Typfehler vor.

Zum Beispiel können wir, wenn wir annehmen, dass $f :: \mathit{Bool} \to \mathit{Bool}$ und `True` $:: \mathit{Bool}$, schließen, dass (f `True`) $:: \mathit{Bool}$. Diese Regel funktioniert zunächst nur für monomorphe Typen – nicht für Typausdrücke, die Typvariablen enthalten. Ist der Typ der linken/rechten Seite der Applikation polymorph, müssen wir überprüfen ob zwei Typen gleich gemacht werden können, um festzustellen, ob das Argument an die Funktion übergeben werden darf, bzw. ob die linke Seite überhaupt eine Funktion ist oder sein kann.[1] Konkreter interessieren wir uns für eine Ersetzung auf Typausdrücken, die Typvariablen derart ersetzt, dass die Typausdrücke danach syntaktisch gleich sind. Dazu verwenden wir ein Verfahren, dass *Unifikation* heißt und auf J. A. Robinson [20] zurück geht. Robinson verwendete die Unifikation in einem anderen Kontext. Ihm ging es um einen automatischen Beweiser basierend auf dem Resolutionskalkül. Die Fragestellung war eine ähnliche: gibt

[1] Betrachte `letrec` $u = u$ `in` u, auf den ersten Blick ist nicht unbedingt klar, dass der Ausdruck als Funktion verwendet werden kann.

es eine Ersetzung, die zwei prädikatenlogische Formeln gleich macht, und falls ja, wie sieht sie aus? Siehe auch [3].

Definition 4.2.1. Eine *Typsubstitution* ist eine Abbildung, die Typen für Typvariablen liefert. Diese können wiederum Typvariablen sein oder Typvariablen enthalten. ε ist die *leere* Typsubstitution, so dass $\forall a : \varepsilon(a) = a$. Eine Variable a, die von einer Typsubstitution φ nicht *versetzt* wird, ist eine Variable für die $\varphi(a) = a$ gilt. $\mathrm{Dom}(\varphi)$ ist die Menge der Typvariablen, die von φ versetzt werden. Die Menge

$$\{a_1 \mapsto T_1, \ldots, a_n \mapsto T_n\}$$

ist eine Typsubstitution mit *Substitutionskomponenten* $a_i \mapsto T_i$. Eine Substitution der Form

$$\{a \mapsto T\}$$

ist eine *Delta-Substitution*. Sie enthält genau eine Substitutionskomponente, bildet also nur eine Variable auf einen Typ ab. Zwei Typsubstitutionen φ und ψ sind *gleich*, genau dann wenn $\forall a : \varphi(a) = \psi(a)$ gilt. Wenn φ eine Typsubstitution und T ein Typ ist, dann ist

$$T^{\varphi}$$

eine *Instanz* von T gemäß φ, die entsteht, indem alle freien Vorkommen von allen $a \in \mathrm{Dom}(\varphi)$ in T durch $\varphi(a)$ ersetzt werden. Manchmal sagen wir „wir wenden eine Substitution an“ statt „wir erzeugen eine Instanz“ (als dürfte man eine Typsubstitution auf Typvariablen *und* auf Typausdrücke anwenden). Statt $S^{\varphi} = T^{\varphi}$ schreiben wir $S =_{\varphi} T$, wobei S und T Typen sind. Ein *Unifikator* von S und T ist eine Typsubstitution φ, die die beiden Typen syntaktisch gleich macht, so dass

$$S =_{\varphi} T$$

gilt. Ein *allgemeinster Unifikator* ist ein Unifikator φ mit der Eigenschaft, dass jeder andere Unifikator erzeugt werden kann, indem neue Substitutionskomponenten zu φ hinzugefügt werden. Eine *Substitutionskomposition*

$$\psi \circ \varphi$$

ist eine Typsubstitution, die alle Substitutionskomponenten $a_i \mapsto T_i$ aus ψ enthält, für die $\nexists S : a_i \mapsto S \in \varphi$ gilt, und Substitutionskomponenten $b_i \mapsto S_i^{\psi}$, wobei $b_i \mapsto S_i \in \varphi$. Eine Typsubstitution φ ist *idempotent*, genau dann wenn eine nochmalige Instanziierung einen Term nicht weiter verändert, also $(T^{\varphi})^{\varphi} = T^{\varphi}$, bzw. $\varphi \circ \varphi = \varphi$ gilt. Ein Typausdruck T ist ein *Fixpunkt* einer Typsubstitution φ, genau dann wenn $T^{\varphi} = T$ gilt.

Damit ein Ausdruck $(s\ t)$ einen Typ hat, wobei $s :: S \to Q$ und $t :: T$, muss es eine Typsubstitution φ geben, so dass $S =_{\varphi} T$ gilt. Wir fordern zusätzlich, dass φ ein allgemeinster Unifikator ist. Wenn φ zum Beispiel $\{a \mapsto b\}$ ist, dann ist $\varphi \circ \{b \mapsto (c \to c)\}$ eine Substitution, die für a einen spezielleren Typ liefert. Die Regeln in Definition 4.2.2 können verwendet werden, um einen allgemeinsten Unifikator φ zu berechnen, so dass $S =_{\varphi} T$ gilt.

Bemerkung 4.2.1. Zwar kann eine Typsubstitution φ prinzipiell auf alle Variablennamen angewendet werden, so dass der Definitionsbereich eigentlich der kleenesche Abschluss einer Symbolmenge ist, die Definition von $\mathrm{Dom}(\varphi)$ ist jedoch „brauchbarer“ für uns, da es oft darauf ankommt, wann eine Variable durch etwas anderes ersetzt wird.

Definition 4.2.2. (Unifikationsregeln) Sei G eine Multimenge (Elemente müssen nicht paarweise verschieden sein) von Gleichungen der Form $Q_1 \doteq Q_2$, wobei Q_1 und Q_2 Typen sind. Falls G die Form hat, wie sie über einem Strich steht, darf G so verändert werden, wie es unter dem Strich steht.

$$\frac{G = \{K\ S_1\ \ldots\ S_n \doteq K\ T_1\ \ldots\ T_n\} \cup H}{G = \{S_1 \doteq T_1, \ldots, S_n \doteq T_n\} \cup H} \qquad \text{(Dekomposition)}$$

$$\frac{G = \{a \doteq T\} \cup H}{G = \{a \doteq T\} \cup H[T/a]} \qquad \text{(Ersetzung)}$$

$$\frac{G = \{a \doteq a\} \cup H}{G = H} \qquad \text{(Vereinfachung)}$$

$$\frac{G = \{S \doteq T\} \cup H}{G = \{T \doteq S\} \cup H} \qquad \text{(Vertauschung)}$$

Bei einer Dekomposition müssen die Typkonstruktoren gleich sein, andernfalls kann die Berechnung abgebrochen werden, weil das ein Typfehler ist. Die Berechnung kann beendet werden, wenn ein Gleichungssystem der Form

$$a_1 \doteq Q_1, \ldots, a_m \doteq Q_m$$

vorliegt, wobei a_i Typvariablen und Q_i Typen sind und kein a_i in Q_i vorkommen darf. Falls während der Berechnung irgendwann eine Gleichung $a = \ldots\ a\ \ldots$ entsteht, liegt ebenfalls ein Typfehler vor (*Occurs-Check*-Einschränkung).

In Abhängigkeit von den Gleichungen, die sich in G befinden, wird G durch die Anwendung der Regeln solange verändert, bis die gewünschte Form erreicht ist. Nachfolgend wollen wir einige Beispiele betrachten und benutzen darin einige bekannte Funktionen, die es auch in der Standardbibliothek von Haskell gibt.

Beispiel 4.2.1. Um zum Beispiel den Typ von (*id const*) zu bestimmen, nehmen wir als Erstes der Einfachheit halber an, dass die Typen von *id* und *const* bekannt sind. Die Namen von Typvariablen dürfen sich nicht überschneiden, weshalb sie ggf. umbenannt werden müssen. Bei der im Folgenden verwendeten Schreibweise stehen über einem Strich bekannte Fakten und darunter eine Folgerung.

$$\frac{id :: a \to a,\ const :: a' \to b' \to a'}{id\ const :: a^{\varphi}}$$

Nachdem *const* als Parameter an *id* übergeben wurde, „verschwindet“ das erste a und das was „übrig bleibt“ ist der Typ der Anwendung, wobei die Typsubstitution φ noch

berechnet werden muss (weil a allein zu allgemein sein könnte, und es hier auch ist). Die Berechnung von φ ist für dieses Beispiel besonders einfach. Im ersten und einzigen Schritt

$$a \doteq a' \to b' \to a'$$

setzen wir a, den Typ des Parameters, mit dem Typ des Arguments gleich. Das Gleichungssystem hat sofort die Form, bei der keine Schritte mehr durchgeführt werden müssen. Damit ist die Substitution $\varphi = \{a \mapsto (a' \to b' \to a')\}$ und somit $a' \to b' \to a'$ der Typ von (*id const*).

Beispiel 4.2.2. Die Berechnung des Typs von (*comp concat concat*) benötigt mehr Schritte, wobei wir die Typen von *comp* und *concat* als gegeben ansehen.

$$\frac{\mathit{comp} :: (b \to c) \to (d \to b) \to d \to c,\ \mathit{concat} :: [[a]] \to [a]}{\mathit{comp\ concat} :: ((d \to b) \to d \to c)^{\varphi}}$$

Jetzt müssen wir die Typsubstitution φ berechnen. Der Typ des ersten Parameters von *comp* ist $(b \to c)$.

$$\frac{b \to c \doteq [[a]] \to [a]}{b \doteq [[a]],\ c \doteq [a]} \quad \text{(Dek.)}$$

Der Pfeil $\to$ ist ein Typkonstruktor, also wenden wir die Dekompositionsregel an. Damit liegt die Typsubstitution $\varphi = \{b \mapsto [[a]],\ c \mapsto [a]\}$ vor, und wir können weiter rechnen.

$$\frac{(\mathit{comp\ concat}) :: (d \to [[a]]) \to d \to [a], \mathit{concat} :: [[a']] \to [a']}{\mathit{comp\ concat\ concat} :: (d \to [a])^{\varphi}}$$

Wir haben die Typvariablen im Typ von *concat* umbenannt und setzen den Typ des ersten Parameters mit dem Typ von *concat* gleich.

$$\frac{d \to [[a]] \doteq [[a']] \to [a']}{d \doteq [[a']],\ [[a]] \doteq [a']} \quad \text{(Dek.)}$$

Da das Gleichungssystem noch nicht die gewünschte Form hat (so dass links nur Typvariablen stehen), machen wir weiter.

$$\frac{d \doteq [[a']],\ [[a]] \doteq [a']}{d \doteq [[a']],\ [a] \doteq a'} \quad \text{(Dek.)}$$

Die eckigen Klammern $[\cdot]$ sind ein Typkonstruktor für Listentypen, wir konnten also wieder die Dekompositionsregel anwenden. Im nächsten Schritt

$$\frac{d \doteq [[a']],\ [a] \doteq a'}{d \doteq [[a']],\ a' \doteq [a]} \quad \text{(Vert.)}$$

bringen wir a' auf die linke Seite (weil in den rechten Seiten keine Typvariablen vorkommen dürfen, die schon irgendwo links stehen) und erreichen durch die Ersetzung

$$\frac{d \doteq [[a']],\ a' \doteq [a]}{d \doteq [[[a]]],\ a' \doteq [a]} \quad \text{(Ers.)}$$

die Form, an der die Typsubstitution abgelesen werden kann. Also ist $\varphi = \{d \mapsto [[[a]]],\ a' \mapsto [a]\}$ und $(\mathit{comp\ concat\ concat}) :: [[[a]]] \to [a]$.

Beispiel 4.2.3. Bei der Berechnung des Typs von (*comp comp comp*) benennen wir zuerst die Variablen in den Typen für jedes *comp* um.

$$\frac{comp :: (b \to c) \to (d \to b) \to d \to c,\ comp :: (b' \to c') \to (d' \to b') \to d' \to c'}{(comp\ comp) :: ((d \to b) \to d \to c)^{\varphi}}$$

$$\frac{(b \to c) \doteq (b' \to c') \to ((d' \to b') \to (d' \to c'))}{b \doteq (b' \to c'),\ c \doteq (d' \to b') \to d' \to c'} \quad \text{(Dek.)}$$

Durch volle Klammerung[2] der Funktionstypen, kann leichter erkannt werden, was womit gleichgesetzt werden muss. Die Typsubstitution kann abgelesen werden und wir setzen die Berechnung fort.

$$\frac{\begin{array}{c} comp\ comp :: (d \to b' \to c') \to d \to (d' \to b') \to d' \to c', \\ comp :: (\bar{b} \to \bar{c}) \to (\bar{d} \to \bar{b}) \to \bar{d} \to \bar{c} \end{array}}{(comp\ comp\ comp) :: (d \to (d' \to b') \to d' \to c')^{\varphi}}$$

$$\frac{\dfrac{d \to (b' \to c') \doteq (\bar{b} \to \bar{c}) \to ((\bar{d} \to \bar{b}) \to (\bar{d} \to \bar{c}))}{d \doteq (\bar{b} \to \bar{c}),\ b' \to c' \doteq (\bar{d} \to \bar{b}) \to (\bar{d} \to \bar{c})}}{d \doteq (\bar{b} \to \bar{c}),\ b' \doteq (\bar{d} \to \bar{b}),\ c' \doteq (\bar{d} \to \bar{c})}$$

$$\Rightarrow \quad (comp\ comp\ comp) :: (\bar{b} \to \bar{c}) \to (d' \to \bar{d} \to \bar{b}) \to d' \to \bar{d} \to \bar{c}$$

Die Beispiele geben uns ein Gefühl dafür, wie eine Unifizierung aussehen kann, aber noch haben wir kein deterministisches Verfahren angegeben. Wir benötigen einen Algorithmus, der als Eingabe zwei Typausdrücke S, T bekommt und einen allgemeinsten Unifikator für S und T zurück liefert.

Definition 4.2.3. Sei A eine Menge von Typausdrücken. Eine Menge B ist eine *Disagreement*-Menge von A, wenn B genau die Teilausdrücke aus A enthält, die nicht in allen Oberausdrücken gleich sind. Beispiel:

$$A = \{a \to (b \to c),\ a \to d,\ a \to e\}$$
$$B = \{b \to c,\ d,\ e\}$$

Robinsons Unifikationsalgorithmus (die Eingabe ist eine Menge A von Typausdrücken, wenn $A = \{T_1, \ldots, T_n\}$ schreiben wir A^{φ} für $\{T_1^{\varphi}, \ldots, T_n^{\varphi}\}$):

1. setze $\varphi_0 = \varepsilon$ und $k = 0$,
2. falls A^{φ_k} nicht genau ein Element enthält → gehe zu Schritt 3, sonst → terminiere und liefere φ_k zurück,

[2] Der Typkonstruktor → ist rechtsassoziativ und $a \to b \to c$ ist voll geklammert $a \to (b \to c)$.

3. sei V_k das lexikographisch letzte und U_k das vorletzte Element in der Disagreement-Menge B_k von A^{φ_k}, falls V_k eine Typvariable ist und nicht in U_k vorkommt $\rightarrow$ setze $\varphi_{k+1} = \{V_k \mapsto U_k\} \circ \varphi_k$ und $k = k + 1$, und gehe zu 2, sonst $\rightarrow$ terminiere.

Bevor wir die Unifikation implementieren können, benötigen wir eine Implementierung für Typsubstitutionen. Der Definition entnehmen wir, dass eine Typsubstitution eine Funktion ist, die für eine Typvariable einen Typausdruck liefert. Einerseits sind Typvariablen gleichzeitig auch Typen, andererseits wollen wir nicht zur Laufzeit überprüfen, ob eine Substitution nur Variablen als Eingabe bekommt, bzw. Funktionen nur für den `TVar`-Konstruktor definieren. Diese Einschränkung realisieren wir, indem wir den Typ einer Typsubstitution als eine Funktion definieren, die einen String als Parameter erwartet.

```
type Subst = TName -> Type
```

Die einfachste Form einer Substitution ist ε (die leere Typsubstitution). Die Funktion `id_subst` in `Type.hs` ist ε. Sie liefert für einen beliebigen Variablennamen den Namen als Typ.

```
id_subst :: Subst
id_subst tvn = TVar tvn
```

Meistens wollen wir eine Typsubstitution nicht auf Typvariablen anwenden, sondern auf Typen, also eine Instanz erzeugen. Die Funktion `sub_type` kann mit Typausdrücken umgehen und ersetzt alle Variablen in einem Typausdruck, wobei eine Variable durch den Typ ersetzt wird, den die als Parameter übergebene Typsubstitution angewendet auf die Variable liefert. Die Funktion ist nur für Typausdrücke definiert, die keinen Quantor enthalten.

```
sub_type :: Subst -> Type -> Type
sub_type phi (TVar tvn)     = phi tvn
sub_type phi (TCon tcn ts) =
    TCon tcn (map (sub_type phi) ts)
```

Die Komposition ist ein wichtiges Konzept. Beim Unifizieren erzeugen wir die Ausgabe-Substitution durch Komposition, wobei die erste Substitution immer ε ist. Zudem wird immer nur eine Substitutionskomponente auf einmal hinzugefügt, die einzelnen Substitutionen sind also Delta-Substitutionen.

$$\{a \mapsto T, b \mapsto T\} = \{b \mapsto T\} \circ \{a \mapsto b\}$$

In der Implementierung ist eine Typsubstitution daher weniger als eine Menge von Substitutionskomponenten zu sehen. Der Unterschied wird deutlich, wenn man versucht, die Substitutionskomponenten aus einer Typsubstitution zu extrahieren. Es ist dann vorteilhafter, beispielsweise eine `Data.Map`-Struktur zu haben. Die Funktion `scomp` erwartet zwei Substitutionen als Eingabe und liefert deren Komposition.

```
scomp :: Subst -> Subst -> TName -> Type
scomp sub2 sub1 tvn = sub_type sub2 (sub1 tvn)
```

Weil eine Typsubstitution angewendet auf eine Typvariable einen Typ ergibt, können wir die zweite Substitution nicht einfach auf das Ergebnis anwenden, also die Komposition nicht ohne weiteres als Funktionskomposition realisieren. Die Komposition gewährleistet, dass für zwei Substitutionen φ und ψ

$$\forall S : S^{\rho} = (S^{\varphi})^{\psi} \qquad \text{mit } \rho = \psi \circ \varphi$$

gilt. Die Substitutionskomposition ist nicht kommutativ. Seien zum Beispiel $\varphi = \{a \mapsto b\}$, $\psi = \{b \mapsto c\}$ zwei Typsubstitutionen, $\rho_1 = \psi \circ \varphi$ und $\rho_2 = \varphi \circ \psi$, dann ist $\rho_1(a) = c$ und $\rho_2(a) = b$. Die Funktion `delta` erzeugt eine Delta-Substitution.

```
delta :: TName -> Type -> TName -> Type
delta tvn t tvn'
  | (tvn == tvn') = t
  | otherwise = TVar tvn'
```

Wir erhalten die Substitution durch partielles Anwenden (mit Hilfe von Currying), wenn wir nur zwei Argumente übergeben. Das Ergebnis ist dann eine Funktion, die einen Variablennamen erwartet. Falls der Name die Variable der Substitutionskomponente ist, liefert die Funktion den entsprechenden Typ zurück, ansonsten verhält sich die Funktion wie die leere Substitution ε.

Das Unifikationsverfahren (angelehnt an die MIRANDA-Implementierung des Algorithmus von Robinson in [19]) im Modul `Unify.hs` startet mit einer Typsubstitution φ und einer Gleichung $T_1 \doteq T_2$. Gleichungen lassen sich in drei Arten unterteilen, so dass wir diesbezüglich eine Fallunterscheidung machen.

1. Falls $T_1 = (K\ S_1 \ldots S_n)$, $T_2 = a$ und a eine Variable ist $\rightarrow$ rufe das Verfahren mit φ und $a \doteq K\ S_1 \ldots S_n$ rekursiv auf (Vertauschung).
2. Falls $T_1 = a$, $T_2 = T$ und a eine Variable ist $\rightarrow$
   i) falls a durch φ nicht versetzt wird und a nicht in T vorkommt (Occurs-Check) $\rightarrow$ terminiere und liefere $\{a \mapsto T^{\varphi}\} \circ \varphi$ zurück, falls a in T vorkommt $\rightarrow$ terminiere mit einem Fehler,
   ii) falls a durch φ versetzt wird $\rightarrow$ rufe das Verfahren auf φ und $\varphi(a) \doteq T^{\varphi}$ rekursiv auf (wende die Substitution auf beide Seiten an).
3. Falls $T_1 = K_1\ S_1 \ldots S_n$ und $T_2 = K_2\ Q_1 \ldots Q_m \rightarrow$
   i) falls $K_1 \neq K_2 \rightarrow$ terminiere mit einem Fehler,
   ii) falls $K_1 = K_2 \rightarrow$ wende das Verfahren auf φ und $[S_1 \doteq Q1, \ldots, S_n \doteq Q_m]$ an, wobei $n = m$ gilt, weil die Konstruktoren gleich sind.

Im Fall 3, falls wir es mit zwei Typkonstruktoren zu tun haben, müssen wir eine Liste von Gleichungen verarbeiten können. Die Funktion

```
unifyl :: Subst -> [(Type, Type)] -> UResult
unifyl phi eqns = foldr unify' (Right phi) eqns
    where unify' eqn (Right phi) = unify phi eqn
          unify' _   err         = err
```

nimmt eine Typsubstitution und eine Liste von Paaren entgegen und liefert einen allgemeinsten Unifikator zurück – falls es einen gibt – der die Typausdrücke in den Tupeln gleich macht. Bei der Verarbeitung der Liste müssen die einzelnen Teil-Typsubstitutionen nicht mittels $\circ$ verknüpft werden. In jedem Schritt wird eine Substitution durch die Unifikation um neue Komponenten erweitert und an den nächsten Schritt weiter gereicht. Das Ergebnis ist eine Struktur, die ein negatives oder ein positives Ergebnis speichern kann.

```
type UResult = Either (Type, Type) Subst
```

Im Fehlerfall liefern wir `Left` (T_1, T_2) zurück, wobei T_1 und T_2 (Teil-)Typen sind, die nicht gleich gemacht werden können. Konnte eine Typsubstitution φ berechnet werden, ist der Rückgabewert `Right` φ. Die Funktion

```
unify :: Subst -> (Type, Type) -> UResult
```

nimmt eine Typsubstitution und ein Tupel mit zwei Typausdrücken entgegen und liefert einen allgemeinsten Unifikator zurück, falls es einen gibt.

4.3 Regeln für die Typherleitung

Die nachfolgende Definition 4.3.1 definiert Begriffe und Regeln, so wie sie in [1] angegeben sind (wobei leichte Änderungen vorgenommen wurden) und mit deren Hilfe Typen hergeleitet werden können. Das ist *die* zentrale Definition für die Implementierung unseres polymorphen Typherleitungssystems.

Definition 4.3.1. Eine *Typumgebung* ist eine Abbildung, die Typen für Variablennamen liefert (eine Menge von Argument-Wert-Paaren, wobei einem Argument genau ein Wert zugewiesen wird). Typumgebungen bezeichnen wir mit Γ. Der *Definitionsbereich* einer Umgebung Γ, $\mathrm{Dom}(\Gamma)$, ist die Menge der Variablen, die Γ auf Typen abbildet und $\mathrm{Cod}(\Gamma)$ die Menge $\{\Gamma(x) \mid x \in \mathrm{Dom}(\Gamma)\}$. $\Gamma, x :: T$ ist eine *Erweiterung* von Γ um die *Typannahme* $x :: T$ (x hat den Typ T), wobei $x \notin \mathrm{Dom}(\Gamma)$. Eine *Typinferenzrelation* $\Gamma \vdash s :: T$ besteht dann, wenn unter den Annahmen in Γ gilt, dass der Ausdruck s den Typ T hat. In Abbildung 4.3.1 sind Ableitungsregeln für die Herleitung von Typen von $L_{PLC^{-T}}$-Ausdrücken in Form von Schlüssen dargestellt. Die Wahrheit der Prämissen über einem Strich zieht die Wahrheit der Konklusion darunter nach sich.

Abbildung 4.3.1: Typinferenzregeln

$$\frac{(x :: S) \in \Gamma}{\Gamma \vdash x :: S} \qquad \text{(Var)}$$

$$\frac{\Gamma \vdash s :: S_1 \to S_2,\ \Gamma \vdash t :: S_1}{\Gamma \vdash (s\ t) :: S_2} \qquad \text{(App)}$$

$$\frac{\Gamma, x :: S_1 \vdash s :: S_2}{\Gamma \vdash (\lambda x.s) :: S_1 \to S_2} \qquad \text{(Abs)}$$

$$\frac{\Gamma, y :: \text{typeOf}(D) \vdash y\ s_1\ \ldots\ s_n :: S}{\Gamma \vdash (D\ s_1\ \ldots\ s_n) :: S} \quad \text{mit } n = \text{ar}(c) \qquad \text{(Cons)}$$

$$\frac{\begin{array}{ll} \Gamma & \vdash s :: K\ S_1\ \ldots\ S_m \\ \Gamma, x_{1,1} :: T_{1,1}, \ldots, x_{1,n_1} :: T_{1,n_1} & \vdash t_1 :: T \\ \Gamma, x_{1,1} :: T_{1,1}, \ldots, x_{1,n_1} :: T_{1,n_1} & \vdash (D_1\ x_{1,1}\ \ldots\ x_{1,n_1}) :: K\ S_1\ \ldots\ S_m \\ & \vdots \\ \Gamma, x_{k,1} :: T_{k,1}, \ldots, x_{k,n_k} :: T_{k,n_k} & \vdash t_k :: T \\ \Gamma, x_{k,1} :: T_{k,1}, \ldots, x_{k,n_k} :: T_{k,n_k} & \vdash (D_k\ x_{k,1}\ \ldots\ x_{k,n_k}) :: K\ S_1\ \ldots\ S_m \end{array}}{\Gamma \vdash \texttt{case}_K\ s\ \texttt{of}\ \{D_1\ x_{1,1}\ \ldots\ x_{1,n_1} \to t_1,\ \ldots\} :: T} \qquad \text{(Case)}$$

$$\frac{\begin{array}{l} \Gamma, x_1 :: \forall \mathcal{X}_1.T_1, \ldots, x_n : \forall \mathcal{X}_n.T_n \vdash t_1 :: \forall \mathcal{X}_1.T_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \Gamma, x_1 :: \forall \mathcal{X}_1.T_1, \ldots, x_n : \forall \mathcal{X}_n.T_n \vdash t_n :: \forall \mathcal{X}_n.T_n \\ \Gamma, x_1 :: \forall \mathcal{X}_1.T_1, \ldots, x_n : \forall \mathcal{X}_n.T_n \vdash t :: R \end{array}}{\Gamma \vdash (\texttt{letrec}\ x_1 = t_1, \ldots, x_n = t_n\ \texttt{in}\ t) :: R} \qquad \text{(Letrec)}$$

$$\frac{\Gamma \vdash t :: T}{\Gamma \vdash t :: \forall \mathcal{X}.T} \quad \begin{array}{lll} \text{mit} & \mathcal{X} & = \mathcal{FTV}(T) \setminus \mathcal{Y} \\ & \mathcal{Y} & = \bigcup_{S \in \text{Cod}(\Gamma)} \mathcal{FTV}(S) \end{array} \qquad \text{(Generalize)}$$

$$\frac{\Gamma \vdash t :: \forall \mathcal{X}.S_1}{\Gamma \vdash t :: S_2} \quad \text{mit } S_2 = S_1^{\rho},\ \text{Dom}(\rho) = \mathcal{X} \qquad \text{(Instance)}$$

Definition 4.3.2. Sei $\Gamma = \{x_1 :: T_1, \ldots, x_n :: T_n\}$ eine Typumgebung und φ eine Typsubstitution. $\Gamma^\varphi = \{x_1 :: T_1^\varphi, \ldots, x_n :: T_n^\varphi\}$ ist eine Typumgebung, die entsteht indem φ auf alle freien Variablen in Γ angewendet wird.

Ein zentraler Begriff ist die Typumgebung. Während der Typberechnung speichert sie Typannahmen, die wir benötigen, um weitere Typannahmen abzuleiten. Wir verwenden das Haskell-Modul `Data.Map`. Die Datenstruktur für Typumgebungen ist im Modul `Typecheck.hs` definiert:

```
type TypeEnv = Map.Map Name Type
```

`TypeEnv` ist ein Typsynonym. Der Typ (`Map.Map Name Type`) ist ein Wörterbuch, das Typen für Variablennamen liefert.

In den folgenden Abschnitten erläutern wir detailliert, wie der Algorithmus aussieht, der aus der mathematischen Definition der Typherleitung entwickelt wurde. Vieles lässt sich auch in [19] nachlesen (dort wird ein Typsystem implementiert, das einiges mit unserem Typsystem gemeinsam hat).

4.4 Variablen

Am einfachsten ist es, wenn wir den Typ einer Variablen x bestimmen müssen (wobei wir hier nicht solche Variablen betrachten, die unmittelbar hinter einem λ oder vor einem Gleichheitszeichen innerhalb eines Letrec-Ausdrucks stehen). In diesem Fall enthält die Typumgebung Γ notwendigerweise einen Eintrag für x. Da das Programm keine freien Variablen enthält, können wir schließen, dass x weiter außen gebunden wurde, so dass durch einen anderen Typisierungsschritt eine Typannahme garantiert eingefügt wurde. Wir überprüfen daher nicht ob $x \in \mathrm{Dom}(\Gamma)$ gilt. Es kann sein, dass der Typ gebundene Typvariablen enthält. Wir benennen alle gebundenen Variablen um und entfernen den Quantor, was einer Anwendung der Regel (Instance) entspricht. Bei der Instanziierung erhalten wir einen Typausdruck T und eine unendliche Liste $[b_1, \ldots]$ mit frischen Variablennamen.

1. Falls T keinen Quantor enthält $\longrightarrow$ liefere T zurück,
2. sonst hat T die Form $\forall[a_1, \ldots, a_n].S \longrightarrow$ erzeuge eine Typsubstitution $\varphi = \{a_1 \mapsto b_1\} \circ \ldots \circ \{a_n \mapsto b_n\}$ und liefere S^φ und die Liste $[b_{n+1}, \ldots]$ zurück.

Wir wollen nun die Herleitung des Typs beschreiben. Das Verfahren erhält eine Typumgebung Γ, eine Liste mit frischen Variablennamen v und einen Variablennamen x als Eingabe.

1. Suche den Eintrag $\Gamma(x)$ und setze $T = \Gamma(x)$.

2. Erzeuge eine neue Instanz T' von T
3. Liefere ε, den Ausdruck $x :: T'$ und v' zurück, wobei v' die Liste der übrig gebliebenen Namen ist.

Die leere Typsubstitution im Rückgabewert drückt aus, dass wir keine neuen Ersetzungen eingeführt haben, die Typen verändern.

Bei der Umbenennung von Programmvariablen verwendeten wir eine unendliche Liste mit frischen Variablennamen. Bei der Typherleitung benötigen wir ebenfalls eine solche Liste. Die Liste

```
new_tvars = ["a", "b", "c", "d", "e",
             "a'", "b'", "c'", "d'", "e']
            ++ ['t' : show i | i <- [1..] ]
```

enthält unendlich viele paarweise verschiedene Variablennamen. Im Unterschied zur Liste `newvars` wird diese Liste nicht nur für Umbenennungen gebraucht. Wir verwenden sie um neue Typausdrücke zu erstellen und um Typvariablen umzubenennen. Es geht jedoch nicht darum, Variablen umzubenennen, die gleich heißen. Typvariablen sind von Anfang an eindeutig. Wenn irgendwo eine Typvariable a vorkommt, können wir schließen, dass jedes andere Vorkommen von a den gleichen Typ bezeichnet. Das Verfahren achtet darauf, dass wenn eine neue Typvariable eingeführt wird, der Name nicht schon vorher verwendet wurde. Die Funktion

```
newinstance :: [TName] -> Type -> (Type, [TName])
newinstance tvars (ForAll vs t) = (sub_type phi t, tvars')
    where al     = zip vs tvars
          phi    = al_to_subst al
          tvars' = drop (length vs) tvars
newinstance tvars t = (t, tvars)
```

in `Type.hs` nimmt einen Typausdruck entgegen und erzeugt eine neue Instanz, indem der Quantor entfernt und alle gebundenen Variablen umbenannt werden. Die verwendete Funktion `al_to_subst` nimmt eine Liste von Paaren (Zuordnungen $Variablenname \mapsto Variablenname$) entgegen und erzeugt daraus eine Typsubstitution. Die Anwendung der Substitution auf alle Typvariablen eines quantor-freien Typausdrucks liefert einen neuen Typausdruck, in dem alle Typvariablen umbenannt wurden, wobei alle Variablennamen frisch sind.

Bemerkung 4.4.1. Wenn wir in diesem Kapitel von Instanziierung sprechen, ist die Anwendung von `newinstance` gemeint. Der Typ $Bool \to Bool$ ist eine Instanz von $\forall a, b.a \to b$. So etwas macht `newinstance` aber nicht.

Eine Variable wird in der Implementierung durch (`TVar` x) repräsentiert. Um den Typ zu bestimmen, übergeben wir x (x ist eine `Id`-Struktur) an die Funktion

```
tcVar :: TypeEnv -> [TName] -> Id -> TCResult Expr
```

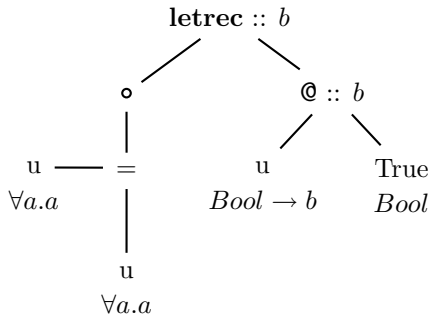
in `Typecheck.hs`, die eine Typumgebung, eine Liste mit Variablennamen und einen Bezeichner als Parameter entgegen nimmt. Das Ergebnis der Typberechnung ist (nicht nur bei Variablen) eine `TCResult`-Struktur.

```
type TCResult a = Either String (Subst, a, [TName])
```

Im Erfolgsfall wird eine Typsubstitution, ein typisierter Ausdruck (oder eine Liste von Ausdrücken), und die Liste der übrig gebliebenen Variablennamen zurückgeliefert, oder eine Fehlermeldung, falls ein Fehler aufgetreten ist.

4.5 Applikationen

Wie wir bereits aus Abschnitt 4.2 wissen, können wir den Typ einer Funktionsanwendung nicht direkt angeben, falls die beteiligten Typen Typvariablen enthalten – wir müssen unifizieren. In den Beispielen zur Unifikation, haben wir den Typ des formalen Parameters mit dem Typ des aktuellen Parameters gleich gesetzt. Dieses Vorgehen ist unpraktisch, wenn wir uns in der Implementierung befinden. Erstens muss man dann den Typ des formalen Parameters extrahieren, und außerdem kann es sein, dass die Funktion einen Typ der Form $T \to S$ gar nicht hat, wie es zum Beispiel bei u in



(`letrec` $u = u$ `in` u `True`)

der Fall ist: unmittelbar vor der Unifikation und nach der Instanziierung hat u im Unterausdruck (u `True`) einen Typ, der äquivalent ist zu a'. Aus diesem Grund gehen wir ein bisschen anders vor. Wir erhalten eine Typumgebung Γ, eine Liste mit frischen Variablennamen und zwei Ausdrücke e_1 und e_2 als Eingabe.

1. Berechne die Typen von e_1 und e_2:

   i) Berechne den Typ von e_1, falls ein Fehler auftritt $\longrightarrow$ terminiere mit einem Fehler, sonst liegt eine Typsubstitution γ_1 und ein Ausdruck $e_1' :: T_1$ vor $\longrightarrow$ setze $\Gamma' = \Gamma^{\gamma_1}$ (wende γ_1 auf alle Typen in Γ an) und gehe zu Schritt 1.ii).

   ii) Berechne den Typ von e_2, verwende dabei Γ', falls ein Fehler auftritt $\longrightarrow$ terminiere mit einem Fehler, sonst liegt eine Typsubstitution γ_2 und ein Ausdruck $e_2' :: T_2$ vor $\longrightarrow$ gehe zu Schritt 2.

2. Setze $\psi = \gamma_2 \circ \gamma_1$ und rufe die Unifizierung mit ψ, T_1 und $T_2 \to a$ auf, wobei a eine frische Typvariable ist,

   i) falls ein Fehler auftritt $\longrightarrow$ terminiere mit einem Fehler,

   ii) sonst ist φ die zurückgegebene Typsubstitution, die T_1 und $(T_2 \to a)$ gleich macht $\longrightarrow$ liefere φ, den Ausdruck $(e_1' :: T_1\ e_2' :: T_2) :: a$ und die Liste mit übrig gebliebenen Variablennamen zurück.

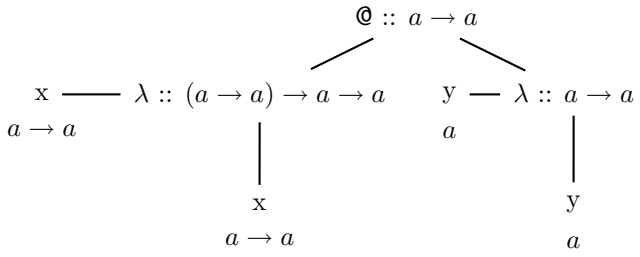
Nachdem der Typ der linken Seite einer Applikation berechnet wurde, wenden wir die berechnete Typsubstitution auf alle Typen in der Typumgebung an.

```
sub_te :: Subst -> TypeEnv -> TypeEnv
sub_te phi gamma = Map.map (sub_qtype phi) gamma

sub_qtype :: Subst -> Type -> Type
sub_qtype phi (ForAll vs t) =
    ForAll vs (sub_type (exclude phi vs) t)

    where exclude phi vs tvn
            | tvn `elem` vs = TVar tvn
            | otherwise     = phi tvn
```

Die Funktion `sub_te` ist die Implementierung dieser Operation. Sie nimmt eine Typsubstitution φ und eine Typumgebung Γ als Parameter entgegen und liefert eine neue Typumgebung, die aus Γ entsteht, indem φ auf die rechten Seiten der gespeicherten Typannahmen $x :: T$ angewendet wird, wobei nur freie Typvariablen ersetzt werden (siehe auch Definition 4.3.2). Wir aktualisieren die Typumgebung bevor wir den Typ der rechten Seite einer Applikation berechnen, damit eventuelle Änderungen der Typannahmen nicht „vergessen“ werden. Das ist besonders dann wichtig, wenn eine weiter außen gebundene Variable in beiden Seiten der Applikation vorkommt.

Es gibt noch ein anderes Detail. Wir erhalten den Typ der Applikation, indem wir φ auf a anwenden. Das liefert aber nicht endgültigen Typ, weil der Typ durch weitere Schritte „von außen“ verändert werden kann. Aus diesem Grund wenden wir die Substitution noch nicht an (obwohl es kein Fehler wäre). Die Informationen, die wir brauchen sind in φ gespeichert, und gehen nicht verloren, weil wir φ nach außen weiter geben. Seien e_1, e_2 und e_3 Ausdrücke, $(e_1\ e_2) :: a$ und φ die Typsubstitution, die bei der Berechnung des Typs von $(e_1\ e_2)$ entstanden ist. Bei der Berechnung des Typs von $((e_1\ e_2)\ e_3)$ ersetzt die Unifikation a auf jeden Fall durch $\varphi(a)$, falls $\varphi(a) \neq a$ (Schritt 2.ii auf Seite 59), was garantiert, dass der Typ nicht zu allgemein ist.

Abbildung 4.5.1:



Beispiel 4.5.1. Betrachten wir die Ausdrücke $e_1 = (\lambda x.x)$ und $e_2 = (\lambda y.y)$. Zunächst hat jeder Ausdruck für sich den gleichen Typ $a \to a$, bzw. hat der zweite Ausdruck einen bis auf Umbenennungen gleichen Typ $b \to b$ (siehe nächstes Unterkapitel). Um den Typ der Anwendung $(e_1\ e_2)$ zu bestimmen, unifizieren wir $a \to a$ und $(b \to b) \to c$.

$$\frac{a \to a \doteq (b \to b) \to c}{a \doteq b \to b,\ a \doteq c} \tag{1}$$

$$\frac{\varphi = \{a \mapsto (b \to b)\},\ a \doteq c}{b \to b \doteq c} \tag{2}$$

$$\frac{\varphi = \{a \mapsto (b \to b)\},\ b \to b \doteq c}{c \doteq b \to b} \tag{3}$$

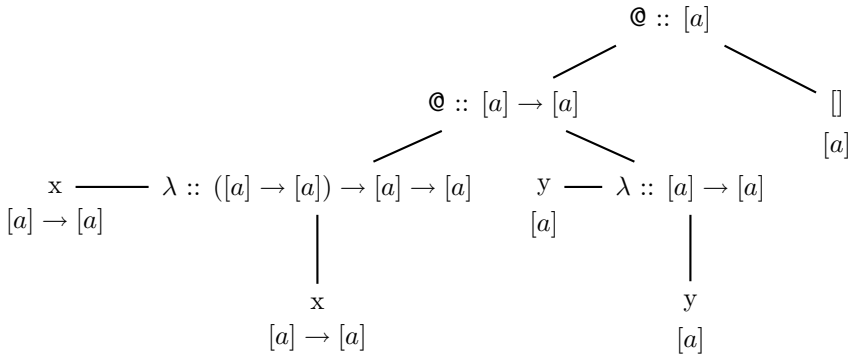
Als letzten Unifikationsschritt erweitern wir die Typsubstitution und setzen $\varphi' = \{c \mapsto (b \to b)\} \circ \varphi$, so dass der Typ von e_1 nun $(b \to b) \to b \to b$ ist. Während der Unifikation, dürfen wir eine Typsubstitution ψ nur dann um eine Substitutionskomponente $a \mapsto T$ erweitern, wenn $a \notin \mathrm{Dom}(\psi)$ gilt. Im Schritt (2) wenden wir zuerst $\varphi = \{a \mapsto (b \to b)\}$ auf beide Seiten von $a \doteq c$ an und erhalten zuerst $b \to b \doteq c$ und durch Vertauschen $c \doteq b \to b$, so dass wir letztendlich φ um $c \mapsto (b \to b)$ erweitern. Wenden wir φ' auf alle Typvariablen an und benennen danach alle Typvariablen um, erhalten wir insgesamt den Ausdruck in Abbildung 4.5.1.

Beispiel 4.5.2. Wir betrachten einen Fall, bei dem wir die Applikations-Regel mehrmals hintereinander anwenden müssen. Seien

$$\begin{aligned} e_1 &= \lambda x.x, \\ e_2 &= \lambda y.y, \\ e_3 &= [] \end{aligned}$$

Ausdrücke. Die Anwendung der Abstraktions-Regel liefert: $e_1 :: a \to a$ und $e_2 :: b \to b$, wobei wir die verbrauchten Variablen nicht nochmal verwenden. Sei weiterhin $[d]$ der Typ von e_3 (eigentlich erhalten wir den Typ von Nil, indem wir den von `typeOf("[]")` zurückgegebenen Typ instanziieren, die Funktion `typeOf` liefert Typen von Datenkonstruktoren). Wir wollen den Typ der Anwendung $((e_1\ e_2)\ e_3)$ bestimmen.

Abbildung 4.5.2:



1. Die Berechnung des Typs von $(e_1\ e_2)$ verläuft genau so, wie im vorigen Beispiel: wir unifizieren $a \to a$ und $(b \to b) \to c$, wobei wir mit der leeren Typsubstitution anfangen, und erhalten die Substitution $\varphi = \{c \to (b \to b)\} \circ \{a \to (b \to b)\}$. Der Typ der inneren Anwendung $(e_1\ e_2)$ ist c.

2. Als nächstes unifizieren wir c und $[d] \to e$, wobei wir φ in die Berechnung mit einbeziehen (und so tun, als könnten wir den Wert einer Variablen ändern, um nicht jedes mal einen neuen Namen einführen zu müssen). Die erste Gleichung ist $c \doteq [d] \to e$. Da c von φ versetzt wird, dürfen wir nicht einfach φ um

   $$c \mapsto ([d] \to e)$$

   erweitern, sondern müssen φ auf beide Seiten der Gleichung anwenden. Wir erhalten $b \to b \doteq [d] \to e$ und rechnen weiter.

   $$\frac{\varphi = \{c \mapsto (b \to b)\} \circ \{a \mapsto (b \to b)\},\ b \to b \doteq [d] \to e}{b \doteq [d],\ b \doteq e} \tag{1}$$

   $$\frac{\varphi = \{b \mapsto [d]\} \circ \{c \mapsto (b \to b)\} \circ \{a \mapsto (b \to b)\},\ b \doteq e}{[d] \doteq e} \tag{2}$$

   $$\frac{\varphi = \{b \mapsto [d]\} \circ \{c \mapsto (b \to b)\} \circ \{a \mapsto (b \to b)\},\ [d] \doteq e}{e \doteq [d]} \tag{3}$$

   Weil $e \notin \mathrm{Dom}(\varphi)$, erweitern wir φ um $\{e \mapsto [d]\}$. Der Typ der gesamten Anwendung $(\lambda x.x)\ (\lambda y.y)\ []$ ist e.

3. Wir wenden φ auf alle Typmarkierungen an und benennen alle Typvariablen um. In Abbildung 4.5.2 ist der Ausdruck mit eingefügten Typmarkierungen dargestellt.

Jetzt ist klar, warum wir die Typsubstitution erst ganz am Schluss anwenden und nicht direkt nachdem der Typ einer Anwendung berechnet wurde. Wendet man im vorigen Beispiel die „noch nicht fertige“ Substitution auf c in $(\lambda x.x)\ (\lambda y.y) :: c$ an, erhält man $(\lambda x.x)\ (\lambda y.y) :: b \to b$ statt $(\lambda x.x)\ (\lambda y.y) :: [d] \to [d]$ ($[d] \to [d]$ ist der Typ vor dem Umbenennen), womit der Typ zu allgemein ist.

Bemerkung 4.5.1. Da die Funktion `unifyl` die Eingabeliste mit Gleichungen in umgekehrter Reihenfolge verarbeitet, entspricht unsere Vorgehensweise hier nicht ganz dem, was die Funktion macht – wir haben die Gleichungen beim Unifizieren von links nach rechts verarbeitet.

Die Funktion

```
tcApp :: TypeEnv -> [TName] -> Expr -> Expr
         -> TCResult Expr
```

erwartet eine Typumgebung, eine Liste mit frischen Variablen und zwei Ausdrücke e_1, e_2 als Parameter und liefert im Erfolgsfall eine Typsubstitution, $(e_1'\ e_2') :: T$ und die Liste der übrig gebliebenen Variablen zurück, wobei e_1', e_2' die Ausdrücke e_1, e_2 mit eingefügten Typmarkierungen sind.

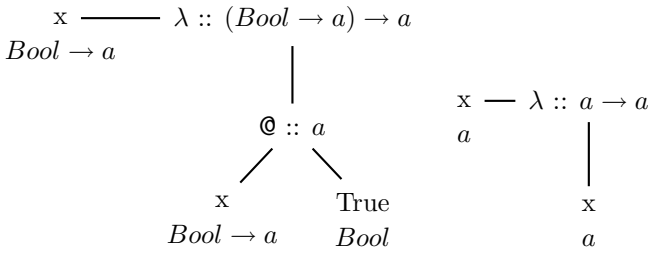
4.6 Abstraktionen

Um den Typ eines Lambda-Ausdrucks $\lambda x.t$ zu bestimmen, müssen wir erst den Typ von t kennen. Nehmen wir an, wir konnten $t :: T$ berechnen. Dann muss der Typ des Gesamtausdrucks die Form $S \to T$ haben. Wir können T nicht berechnen, ohne dass es für x einen Typ in der Typumgebung gibt (außer wenn x in t nicht verwendet wird). Da wir, bevor wir anfangen T herzuleiten, nichts über x wissen, müssen wir das Allgemeinste annehmen und speichern $x :: a$ in Γ. Die Verwendung von x in t bestimmt andererseits den Typ von x. Ist der Ausdruck zum Beispiel $(\lambda x.x$ `True`$)$, dann muss x eine Funktion sein, die auf `Bool` angewendet werden kann. Bei der Berechnung des Typs der Identitätsfunktion hingegen wird der Typ des Parameters nicht spezieller gemacht, da der Ausdruck, auf den abgebildet wird, x selbst ist.

Bei der Berechnung des Typs einer Abstraktion erhalten wir eine Typumgebung Γ, eine Liste v mit frischen Variablennamen und einen Ausdruck $\lambda x.t$ als Eingabe.

1. Speichere eine neue Typannahme $x :: a$ in Γ, wobei a ein frischer Variablenname ist.
2. Wende die Typberechnung auf t an,
   i) falls diese fehlschlägt $\longrightarrow$ terminiere mit einem Fehler,

Abbildung 4.6.1: $(\lambda x.x$ `True`$)$ und $(\lambda x.x)$



ii) sonst ist das Ergebnis (φ, t', v'), wobei φ eine Typsubstitution, t' der Ausdruck t mit eingefügten Typmarkierungen und v' die Liste mit übrig gebliebenen Namen ist $\longrightarrow$ liefere φ, $(\lambda x :: a.t' :: T) :: a \to T$ und v' zurück.

Im Ergebnis hat x unabhängig von seiner Verwendung in t immer noch den allgemeinsten Typ a. In der Typsubstitution φ ist gespeichert, welchen Typ x wirklich hat. Um diesen Typ zu erhalten müssen wir φ auf a anwenden. Das machen wir aber ganz am Schluss, nachdem die Typberechnung für den Gesamtausdruck abgeschlossen wurde.

Beispiel 4.6.1. In diesem Beispiel betrachten wir den Ausdruck $(\lambda x.x$ `True`$)$ und wollen die Typen der Unterausdrücke in berechnen.

1. Als erstes fügen wir eine neue Typannahme $x :: a$ in Γ ein.
2. Durch Unifikation erhalten wir $(x :: Bool \to b$ `True` $:: Bool) :: b$.
3. Der Gesamtausdruck hat somit den Typ $(Bool \to a) \to a$, wenn wir die Typvariable umbenennen.

Es gibt mehr als eine Möglichkeit dafür, wie die Annahme für den Typ des formalen Parameters aussehen kann. An dieser Stelle erlaubt unser Typsystem keine Quantoren. Damit erzwingen wir, dass der Typ eines Parameters x innerhalb eines Lambda-Ausdrucks überall gleich ist.[3] Zum Beispiel hat f in

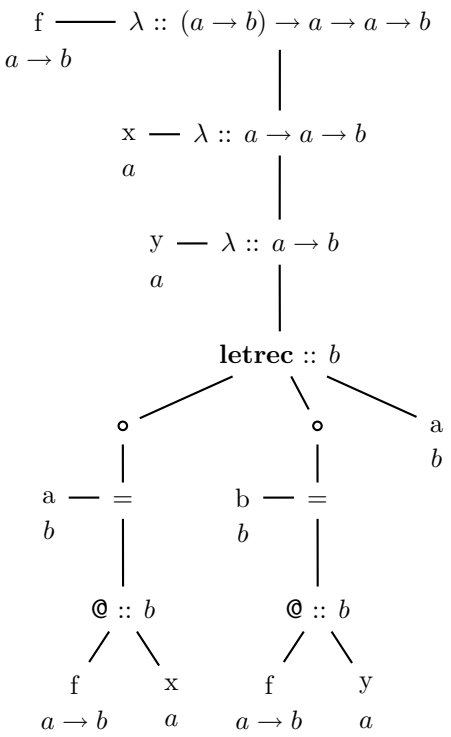
$$\lambda f.\lambda x.\lambda y.\texttt{letrec}\ a = f\ x,\ b = f\ y\ \texttt{in}\ a$$

überall den gleichen Typ (der typisierte Ausdruck ist in Abbildung 4.6.2 dargestellt). Die Konsequenz ist, dass auch die Typen der Parameter x und y gleich sind. Eine Anwendung wie

$$(\lambda f.\lambda x.\lambda y.\texttt{letrec}\ a = f\ x,\ b = f\ y\ \texttt{in}\ a)\ (\lambda x.x)\ True\ []$$

[3] Es kann sein, dass es im Funktionsrumpf weitere x-Bindungen gibt. Da wir alle Programmvariablen umbenannt haben und mit den umbenannten Variablen arbeiten, kommen solche Fälle auf dieser Ebene nicht vor.

Abbildung 4.6.2:



ist nicht erlaubt, obwohl die Identitätsfunktion eigentlich Argumente beliebigen Typs verarbeiten kann.

```
*Main> typecheck $ parse "(\\f -> \\x -> \\y -> letrec a = f x
                              , b = f y in a) (\\x.x) True []"
*** Exception:
Type error: Bool /= [d'], applying
   ((\f -> \x -> \y -> letrec a = f x, b = f y in a) (\x -> x)) True
to
   []
```

Nehmen wir an, der Typ darf quantifiziert werden: $(x :: \forall a.a) \in \Gamma$. Dann kann x im Rumpf an verschiedenen Stellen, verschiedene Typen haben, weil die Instanziierung alle gebundenen Variablen umbenennt. Erlaubt man Quantoren, werden Ausdrücke nicht abgelehnt, die Typfehler enthalten können – wenn im Rumpf einer Abstraktion der Parameter zum Beispiel ein mal auf einen `Bool`-Wert und einmal auf eine Liste angewendet wird und der aktuelle Parameter nur für `Bool` definiert ist.

Beispiel 4.6.2. Betrachten wir den Ausdruck $e = \lambda f.$`seq` $(f$ `True`$)$ $(f$ $[])$. Wir wol-

len feststellen ob der Ausdruck Typfehler enthält, wobei wir Quantoren zulassen. Sei weiterhin

$$\Gamma = \{\texttt{seq} :: \forall a, b.a \to b \to b,\ \texttt{True} :: \mathit{Bool}, [] :: \forall a.[a]\}$$

eine Typumgebung. Als erstes fügen wir $f :: \forall a.a$ in Γ ein (es macht nichts, wenn die Namen der gebundenen Typvariablen sich überschneiden).

1. Bei der Berechnung des Typs von (f `True`) erzeugen wir eine Instanz von $\Gamma(f) = \forall a.a$, so dass der Typ von f zunächst c ist. Durch Unifikation erhalten wir $f :: \mathit{Bool} \to d$.

2. Um den Typ von (f []) zu bestimmen, gehen wir genauso vor, wobei $f :: \forall a.a$ sich immer noch in Γ befindet (die Typsubstitution aus der Berechnung des Typs von (f `True`) wird nicht auf gebundene Variablen in Γ angewendet), und erhalten $f :: [e] \to a'$ für das zweite Auftreten von f.

Aus der Sicht *dieses* Typsystems enthält der Unterausdruck (f []) keine Typfehler. Nehmen wir an, der Gesamtausdruck enthält auch sonst keine Typfehler. Sei g nun eine Funktion, die den Typ $\mathit{Bool} \to \mathit{Bool}$ hat (etwa eine Funktion, die ihren Parameter negiert zurück gibt). Nehmen wir an, das Typsystem lässt den Ausdruck (e g) zu. Übergeben wir nun g an e, ist unklar, was beim Aufruf (g []) passiert.

Es gibt Kontexte, in denen e keine (dynamischen) Typfehler verursacht (beispielsweise e ($\lambda x.x$)). Mit dem Beispiel haben wir gezeigt, dass nicht immer alles gut geht. Für weitere Beispiele siehe [9] oder [19]. Abschließend kann man danach fragen, warum eine durch ein Letrec gebundene Variable einen Quantifizierten Typ haben darf. Der Unterschied ist, dass eine solche Variable nicht „von außen“ kommt, so dass man mehr Gewissheit über ihren Typ hat.

Die Funktion

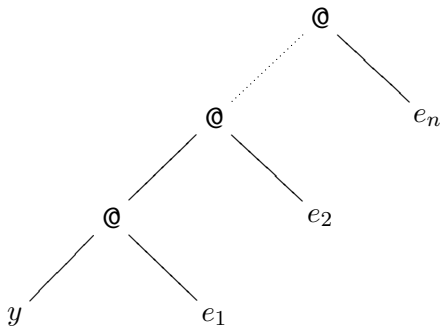
```
tcLambda :: TypeEnv -> [TName] -> Id -> Expr
            -> TCResult Expr
```

berechnet den Typ einer Abstraktion, und liefert im Erfolgsfall eine Typsubstitution, einen Lambda-Ausdruck, in dem alle Typmarkierungen gesetzt wurden, und die Liste der übrig gebliebenen Typvariablen zurück.

4.7 Konstruktoranwendungen, `seq`, `amb`

Datenkonstruktoren sind reservierte Wörter in unserer Sprache. Der Typ eines Datenkonstruktors steht fest, bevor ein Quelltext eingelesen wird. Diese Typen werden bereits durch die Implementierung festgelegt, und müssen nicht zur Laufzeit ermittelt werden.

Abbildung 4.7.1: Funktionsanwendung



Was wir dennoch während des Typchecks ableiten müssen, ist der Typ einer Anwendung eines Datenkonstruktors auf Argumente. Insbesondere weil Datenkonstruktoren nicht ungesättigt vorkommen dürfen (und wir uns während der semantischen Analyse davon überzeugt haben, dass der Quelltext diese Regel nicht verletzt) gibt es nur den Anwendungsfall. Manchmal werden Konstanten bzw. nullstellige Konstruktoren ebenfalls als Anwendungen gesehen. Wir behandeln diesen Fall gesondert und unterscheiden zwischen den Fällen, wo ein Konstruktor auf Argumente angewendet wird, und den Fällen, wo ein Konstruktor auf „nichts angewendet“ wird.

Wenn wir die Typherleitungsregeln auf Seite 61 genauer betrachten, erkennen wir, dass die Regel (Cons) für die Herleitung des Typs einer Konstruktoranwendung etwas ungewöhnlich ist. Der Typ wird so hergeleitet, als handle es sich um eine Funktionsanwendung. In der Implementierung gehen wir dementsprechend vor: wir lassen die Typherleitung glauben, dass der Typ einer Funktionsanwendung berechnet werden muss. Dazu führen wir eine Konvertierung

$$D\ e_1 \ldots e_n \quad \rightsquigarrow \quad y\ e_1 \ldots e_n$$

durch, wobei D ein Datenkonstruktor, y eine Variable und $e_1, \ldots, e_n$ Parameter sind. Die Variable y hat den gleichen Namen wie D. Die Parameter einer Konstruktoranwendung liegen als Liste vor, und die Applikation lässt sich zunächst als

$$[y, e_1, \ldots, e_n]$$

darstellen. Um eine Funktionsanwendung wie in Abbildung 4.7.1 zu erhalten, müssen wir aus einer Liste einen Baum konstruieren. Anwendungen werden links geklammert, so dass die Funktion zuerst auf den ersten Parameter angewendet wird. Die voll geklammerte Version

$$(\ldots\ ((y\ e_1)\ e_2)\ \ldots\ e_n)$$

suggeriert, dass uns die Bibliotheksfunktion `foldl` bei der Konstruktion des Baums behilflich sein kann. `foldl` erwartet eine binäre Operation $\otimes$, einen Startwert a und

eine Liste xs als Eingabe, und verknüpft a und alle Elemente von xs mittels $\otimes$, wobei Linksklammerung benutzt wird. Der Startwert ist für den Fall, dass die Eingabeliste leer ist – dann ist das Ergebnis a (es gibt eine Variante von `foldl`, die Funktion `foldl1`, die keinen Startwert benötigt und nur auf nicht leere Listen angewendet werden kann). Wird (`foldl` $\otimes$ a) zum Beispiel auf $[e_1, e_2, e_3]$ angewendet, ist das Ergebnis eine linksassoziative Verknüpfung

$$(((a \otimes e_1) \otimes e_2) \otimes e_3)$$

von a, e_1, e_2, e_3. Die Liste $[y, e_1, \ldots, e_n]$ lässt sich zu einem Baum falten, wenn wir einen geeigneter Operator definieren. Die Funktion

```
to_app :: [Expr] -> Expr
to_app (e:es) = foldl app e es
    where app r s = App r s NoType
```

in `Typecheck.hs` nimmt eine Liste von Ausdrücken entgegen und liefert einen Anwendungsbaum zurück, wobei wir davon ausgehen, dass die Liste nicht leer sein kann. Die binäre Operation ist die Unterfunktion `app`, die eigentlich nur dazu dient, einen Teilausdruck mit `NoType` zu markieren. Angewendet auf eine aus

$$(\texttt{ConApp} \quad \ldots \quad [e_1, \ldots, e_n])$$

erzeugte Liste, liefert Funktion `to_app` eine Struktur

$$\texttt{App} \ (\ldots \ (\texttt{App} \ (\texttt{App} \ (\texttt{Var} \ \ldots) \ e_1) \ e_2) \ \ldots) \ e_n$$

zurück, deren Teile von der Funktion `tcApp` verarbeitet werden können (die Typen von Funktionsanwendungen berechnet). Falls keine Typfehler auftreten, erhalten wir nach der Typberechnung einen Applikations-Ausdruck mit eingefügten Typmarkierungen.

Da wir die Datenstruktur, die das Programm repräsentiert nicht verändern wollen (außer dass wir Typmarkierungen einfügen), muss das ganze auch rückgängig gemacht werden können. Wir wollen den Anwendungsbaum also wieder in eine Konstruktoranwendung umwandeln. Bäume können auf verschiedene Weisen traversiert werden. Wir beschreiben drei Möglichkeiten. Sei T ein geordneter Baum mit der Wurzel v_r und den Teilbäumen $T_1, \ldots, T_m$.

- *Postorder*: durchlaufe $T_1, \ldots, T_m$ gemäß ihrer Links-Rechts-Ordnung rekursiv in Postorder, besuche danach v_r.
- *Preorder*: besuche v_r, durchlaufe dann $T_1, \ldots, T_m$ rekursiv gemäß ihrer Links-Rechts-Ornung in Preorder.
- *Inorder*: durchlaufe T_1 rekursiv in Inorder, besuche dann v_r, durchlaufe dann die Teilbäume $T_2, \ldots, T_m$ rekursiv in Inorder. Bei einem binären Baum wird zuerst der linke Teilbaum, dann die Wurzel, dann der rechte Teilbaum besucht. Falls der linke Teilbaum leer ist, wird zuerst die Wurzel besucht und dann der rechte Teilbaum (für unsere Bäume gilt: ein Anwendungsknoten hat immer zwei Nachfolger).

Wir erklären Parameter-Teilbäume in einem Anwendungsbaum zu Blättern. Dann erhält man die Blätter eines Anwendugsbaums in geeigneter Reihenfolge, indem man eine *beliebige* Traversierung durchführt (insbesondere weil die inneren Knoten im Ergebnis nicht vorkommen). Wir verwenden dennoch keines der drei Verfahren und besuchen die Teilbäume von rechts nach links. Dabei konstruieren wir eine Liste der Blätter, indem wir eine bereits erzeugte Liste an ein neues Element anhängen. Die Funktion

```
unfold :: Expr -> [Expr]
unfold = unfold' []
    where unfold' acc (App r s _)
              | isVar r    = r:s:acc
              | otherwise = unfold' (s:acc) r
```

erwartet einen Applikationsbaum und liefert eine Liste $[y, e_1, \ldots, e_n]$ zurück, wobei y eine Funktion und e_i die Parameter sind. Aus der Liste lässt sich eine Konstruktoranwendung (`ConApp` ... $[e_1, \ldots, e_n]$ ...) erzeugen.

Wir fassen zusammen und ergänzen: bei der Berechnung des Typs einer Konstruktoranwendung erhalten wir eine Typumgebung Γ, eine Liste v von frischen Variablennamen, den Konstruktornamen D und eine Liste es von Parametern.

1. Falls die Liste es leer ist $\longrightarrow$
   i) erzeuge eine Instanz T von typeOf(D),
   ii) liefere die leere Typsubstitution ε, $(D :: T\ []) ::$ `NoType` und die Liste der übrig gebliebenen Namen zurück.
2. Falls es nicht leer ist $\longrightarrow$
   i) erzeuge eine Variable y, die den gleichen Namen hat wie D, und eine Liste $(y : es)$, wandle $(y : es)$ in eine `Expr`-Struktur r um,
   ii) rufe die Typberechnung auf Γ, v und r auf,
   iii) falls ein Fehler auftritt $\longrightarrow$ terminiere mit einem Fehler, sonst liegt eine Typsubstitution φ, ein Ausdruck r' mit eingefügten Typmarkierungen und eine Liste v' mit übrig gebliebenen Variablennamen vor $\longrightarrow$ erzeuge eine Konstruktoranwendung $c = (D :: T_1\ es') :: T_2$ aus r' und liefere φ, c und v' zurück, wobei es' die Parameterliste mit eingefügten Typmarkierungen ist.

Nullstellige Konstruktoren werden genauso gespeichert wie Konstruktoranwendungen mit Parametern, so dass es auch hier einen Speicherplatz für den Typ des Konstruktors selbst und für den Typ der Anwendung gibt. Der Typ der Anwendung ist `NoType`, falls es keine Parameter gibt. Alternativ kann man definieren, dass die Typmarkierung die selbe ist wie die am Konstruktor. Bei der Berechnung weichen wir etwas von der Inferenzregel (Cons) ab. Die Typumgebung Γ wird in der Implementierung, wie wir schon wissen,

Abbildung 4.7.2: Eine Konstruktoranwendung

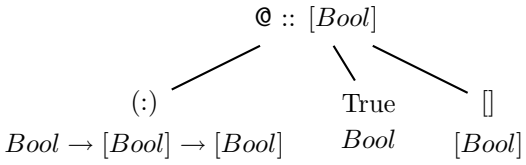
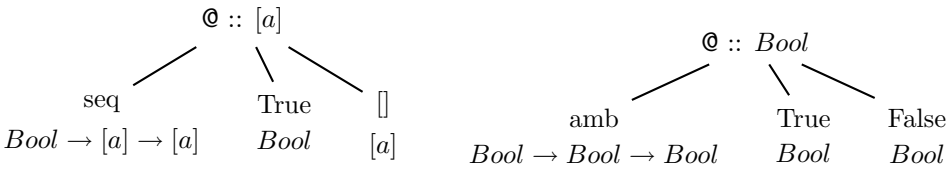


Abbildung 4.7.3: `seq` und `amb`



bereits am Anfang mit Einträgen für alle reservierten Wörter gefüllt, so dass wir nicht jedes mal eine Typannahme einfügen, wenn wir auf ein reserviertes Wort stoßen. Die Funktion

```
tcConApp :: TypeEnv -> [TName] -> Id -> [Expr]
         -> TCResult Expr
```

nimmt eine Typumgebung, eine Liste mit Variablennamen, eine `Id`-Struktur mit dem darin gespeicherten Konstruktornamen und eine Liste mit Parametern entgegen und berechnet die Typen für die Konstruktoranwendung. Die Funktionen

```
tcSeq :: TypeEnv -> [TName] -> Expr -> Expr
      -> TCResult Expr

tcAmb :: TypeEnv -> [TName] -> Expr -> Expr
      -> TCResult Expr
```

berechnen die Typen für `seq`- und `amb`-Anwendungen. Die Berechnung verläuft genauso wie die für Konstruktoranwendungen, außer dass es den nullstelligen Fall nicht gibt. Die Funktion

```
typeOf :: TName -> Type
```

nimmt den Namen eines Datenkonstruktors entgegen und liefert den Typ des Konstruktors zurück.

Bemerkung 4.7.1. Wie wir wissen, werden die Parameter einer Konstruktor-, `seq`- oder `amb`-Anwendung als Liste gespeichert. Die Typen der Teilanwendungen werden dabei

nicht mit gespeichert. So gibt es zum Beispiel in

$$(\texttt{seq} :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}\ \ \texttt{True} :: \mathit{Bool}\ \ \texttt{True} :: \mathit{Bool}) :: \mathit{Bool}$$

keine Typmarkierung am Teilausdruck ($\texttt{seq} :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}\ \ \texttt{True} :: \mathit{Bool}$). Wir verwerfen also einige berechnete Typmarkierungen.

4.8 Case-Ausdrücke

Während der semantischen Analyse haben wir einen Teil der Überprüfung auf Typfehler schon durchgeführt. Case-Ausdrücke sind immer mit einem Typkonstruktor K markiert, und wir haben überprüft, dass alle Datenkonstruktoren in den Pattern der Case-Alternativen zu K passen. Außerdem wissen wir, dass alle Datenkonstruktoren von K durch Pattern abgedeckt werden, so dass der Case-Ausdruck auf alle Werte reagieren kann.[4] Bei der Typberechnung beachten wir K nicht mehr.

Wir erhalten eine Typumgebung Γ, eine Liste v mit Variablennamen, einen Typkonstruktornamen K einen Ausdruck e und eine Liste $[a_1, \ldots, a_n]$ mit Case-Alternativen als Eingabe, wobei n die Anzahl der Datenkonstruktoren von K ist.

1. Berechne den Typ von e,

   i) falls ein Fehler auftritt $\longrightarrow$ terminiere mit einem Fehler,

   ii) sonst liegt eine Typsubstitution φ, ein Ausdruck $e' :: R$ mit eingefügten Typmarkierungen und eine Liste v' mit übrig gebliebenen Variablennamen vor $\longrightarrow$ gehe zu Schritt 2.

2. Berechne die Typen der Case-Alternativen, für alle $a_i = D_i\ x_{i,1} \ldots x_{i,n_i} \to s_i$, wobei $n_i = \mathrm{ar}(D_i)$:

   i) berechne den Typ des Patterns $(D_i\ x_{i,1} \ldots x_{i,n_i})$, hierbei kann kein Typfehler auftreten, danach liegt ein Pattern

   $$p_i = (D_i :: Q_i\ x_{i,1} :: S_{i,1}\ \ \ldots\ \ x_{i,n_i} :: S_{i,n_i}) :: T_i$$

   und eine Liste v_i mit übrig gebliebenen Variablennamen vor,

   ii) berechne den Typ von s_i, falls ein Fehler auftritt $\longrightarrow$ terminiere mit einem Fehler, sonst liegt eine Typsubstitution γ_i, ein Ausdruck $s_i' :: S_i$ mit eingefügten Typmarkierungen und eine Liste v_i' mit übrig gebliebenen Variablennamen vor $\longrightarrow$ setze $\Gamma = \Gamma^{\gamma_i}$ (wende γ_i auf alle Typen in Γ an) und $a_i' = p_i \to s_i' :: S_i$.

[4] Für die Typherleitung ist das weniger relevant.

3. Füge die Substitutionen γ_i zusammen, setze $\psi = \gamma_k \circ \ldots \circ \gamma_1$, setze

$$\begin{aligned} \varphi' &= \psi \circ \varphi, \\ ts &= R : [T_1, \ldots, T_n] \quad \text{(Typ von } e \text{ und Typen der Pattern/Alternativen)}, \\ ts' &= [S_1, \ldots, S_n], \end{aligned}$$

   wobei n die Anzahl der Alternativen ist (der Typ einer Case-Alternative ist der Typ des Patterns).

4. Unifiziere $[ts_1 \doteq ts_2, ts_2 \doteq ts_3, \ldots, ts_{n-1} \doteq ts_n]$ unter Verwendung von φ', wobei ts_i der Typ an Stelle i in ts und n die Länge der Liste ist,

   i) falls ein Fehler auftritt $\longrightarrow$ terminiere mit einem Fehler,

   ii) sonst liegt eine Typsubstitution σ vor $\longrightarrow$ gehe zu Schritt 5.

5. Unifiziere $[ts'_1 \doteq ts'_2, ts'_2 \doteq ts'_3, \ldots, ts'_{n-1} \doteq ts'_n]$ unter Verwendung von σ, wobei ts'_i der Typ an Stelle i in ts' und n die Länge der Liste ist,

   i) falls ein Fehler auftritt $\longrightarrow$ terminiere mit einem Fehler,

   ii) sonst liegt eine Typsubstitution τ vor $\longrightarrow$ setze den Typ des Case-Ausdrucks auf S_1 und liefere τ, den typisierten und wieder zusammengesetzten Case-Ausdruck, und die Liste v'_k der übrig gebliebenen Variablennamen zurück.

Der Einfachheit halber haben wir oben bei der Berechnung der Typen von Pattern nicht die Fälle betrachtet, bei denen ein Pattern aus einem nullstelligen Konstruktor besteht (zum Beispiel []) und nicht beschrieben, wie der Typ eines Patterns genau berechnet wird. Das holen wir jetzt nach. Wir erhalten eine Liste v mit Variablennamen, einen Konstruktor D und eine Liste $xs = [x_1, \ldots, x_n]$ als Eingabe, wobei $n = \mathrm{ar}(D)$.

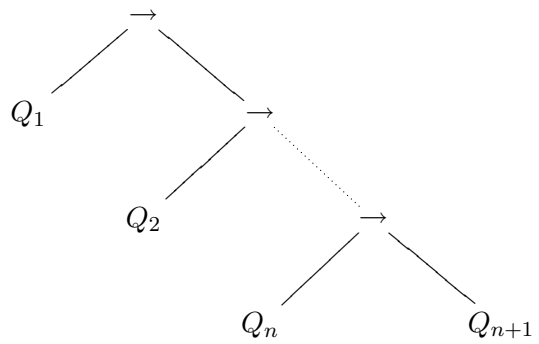
1. Erzeuge eine Instanz T von typeOf(D).

2. Falls xs leer ist $\longrightarrow$ liefere $(D :: T\ []) :: \texttt{NoType}$ und die Liste mit übrig gebliebenen Variablennamen zurück.

3. Der Typ des Konstruktors hat die Form $Q_1 \to \ldots \to Q_{n+1}$, falls xs nicht leer ist, $\longrightarrow$ liefere $(D :: T\ [x_1 :: Q_1, \ldots, x_n :: Q_n]) :: Q_{n+1}$ und die Liste der übrig gebliebenen Variablennamen zurück.

Ein Konstruktor mit Stelligkeit $n > 0$ hat einen Typ der Form $Q_1 \to \ldots \to Q_{n+1}$. Bei der Berechnung des Pattern-Typs weisen wir die Typen $Q_1, \ldots, Q_n$ den Patternvariablen $x_1, \ldots, x_n$ zu; Q_{n+1} ist der Typ des Patterns. Um die Typen $Q_1, \ldots, Q_{n+1}$ zu extrahieren, müssen wir einen Baum verarbeiten. Ein solcher Baum ist in in Abbildung 4.8.1 dargestellt. Die Funktion

Abbildung 4.8.1: Typ eines Konstruktors




```
unfoldt :: Type -> ([Type], Type)
unfoldt (TCon "->" [q1, q2]) =
    let (qs, q) = unfoldt q2 in (q1:qs, q)
unfoldt q = ([], q)
```

zerlegt einen Konstruktortyp; `unfoldt` nimmt einen Typ T entgegen, und liefert, falls T die Form $Q_1 \to \ldots \to Q_{n+1}$ hat, ein Tupel $([Q_1, \ldots, Q_n], Q_{n+1})$, und sonst ein Tupel $([], T)$ zurück.

Wir berechnen die Patterntypen etwas anders, als es in [1] beschrieben wird bzw. als man es aus den Inferenzregeln herauslesen kann (man kann die Patterntypen genauso berechnen wie die Typen von Konstruktoranwendungen). Die Parameter eines Konstruktors in einem Pattern sind keine allgemeinen Ausdrücke sondern Variablen, so dass an dieser Stelle keine Typfehler auftreten können, bzw. nichts unifiziert werden muss. Es gibt dennoch Gemeinsamkeiten. Die Datenstruktur `Alt` speichert einen Konstruktor mit einer Typmarkierung und zusätzlich den Typ der „Konstruktoranwendung“.

```
data Alt = Alt Id [Id] Type Expr
```

Falls es keine Parameter gibt (der Konstruktor nullstellig ist), setzen wir den Typ der „Konstruktoranwendung“ auf `NoType`. Der Typ des Patterns darf beim Unifizieren der Patterntypen nicht `NoType` sein, weshalb wir eine Fallunterscheidung machen und in solchen Fällen den Typ des Patterns mit dem Typ des Konstruktors gleich setzen, wobei, wie schon erwähnt, der Typ des Patterns gleichzeitig der Typ der Case-Alternative ist (entgegen der Intuition, dass der Typ etwas anderes sein müsste).

```
getType (Alt c [] _ _) = getType c
getType (Alt _ _ t _)  = t
```

Der Typ von e und die Patterntypen in $\texttt{case}_K\ e\ \texttt{of}\ \{\ldots\}$ müssen zu K passen. Ob diese Bedingung erfüllt ist, überprüfen wir teilweise während der semantischen Analyse und

teilweise durch Unifikation. Bei der Typberechnung müssen wir nicht überprüfen ob der Typ von e zu K passt. Ist dies nicht der Fall, schlägt die Unifikation im Schritt 4 fehl. Die Konstruktoren in den Pattern gehören zu diesem Zeitpunkt garantiert zu K.

Der Gesamttyp des Case-Ausdrucks wird im letzten Schritt der Typberechnung mit dem Typ der rechten Seite der ersten Alternative gleichgesetzt. Alle Typen der rechten Seiten sind nach der Unifikation gleich (falls kein Fehler aufgetreten ist), so dass wir jede beliebige Alternative nehmen können.

Beispiel 4.8.1. Wir berechnen den Typ den Typ einer Funktion, die überprüft, ob eine Liste leer ist (eine solche Funktion gibt es in der Haskell-Standardbibliothek – sie heißt `null`).

$$\lambda xs.\texttt{case}_{List}\ xs\ \texttt{of}\ \{[] \to \texttt{True};\ y : ys \to \texttt{False}\}$$

1. Als erstes fügen wir eine Typannahme ($xs :: a$) in die Typumgebung Γ ein.

2. Innerhalb des Case-Ausdrucks hat xs dann ebenfalls den Typ a (es wird zunächst nichts weiter mit xs gemacht, xs ist der Ausdruck, mit dem die Pattern verglichen werden).

3. Die Typen der Pattern sind $[] :: [b]$ und $((:) :: c \to [c] \to [c]\ \ y :: c\ \ ys :: [c]) :: [c]$, durch Unifikation erhalten wir $[] :: [c]$ und $xs :: [c]$.

4. Die Typen der rechten Seiten der Case-Alternativen sind jeweils *Bool* und verändern sich durch die Unifikation nicht.

5. Der Typ des Case-Ausdrucks ist somit *Bool*. Wir wenden die berechnete Typsubstitution auf alle Typvariablen an und benennen anschließend alle Typvariablen um. Der Typ der Abstraktion ist dann $[a] \to Bool$.

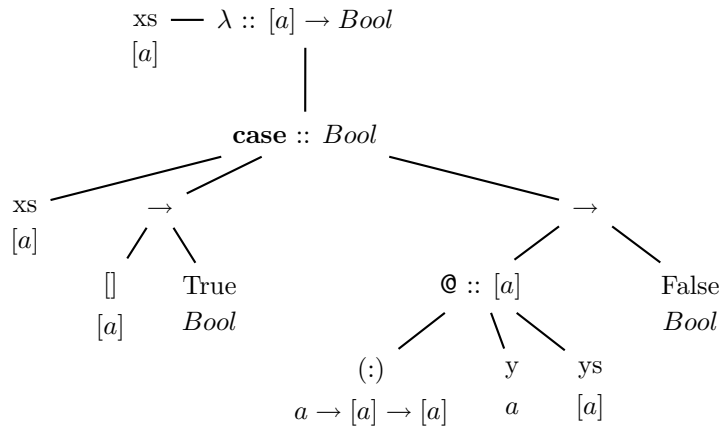
Der Typ von xs wird durch die Unifikation mit den Patterntypen spezieller gemacht, so dass klar wird, dass die Funktion auf Listen operiert. Der Ausdruck mit eingefügten Typmarkierungen ist in Abbildung 4.8.2 dargestellt.

Die Funktion

```
tcCase :: TypeEnv -> [TName] -> Pos -> TName -> Expr
       -> [Alt] -> TCResult Expr
```

nimmt eine Typumgebung, eine Liste mit Variablennamen, eine Positionsmarkierung (die Stelle im Quelltext, an der der Case-Ausdruck beginnt, um Fehlermeldungen präziser zu machen), einen Typnamen (das K in $\texttt{case}_K$), einen Ausdruck und eine Liste mit Case-Alternativen entgegen und liefert einen Case-Ausdruck mit eingefügten Typmarkierungen zurück, falls keine Typfehler auftreten.

Abbildung 4.8.2:



4.9 Letrec-Ausdrücke

Wir verwenden ein iteratives Typherleitungsverfahren. Dabei werden solange neue Typannahmen für Letrec-gebundene Variablen hergeleitet, bis die Typen sich nicht mehr ändern. Im Unterschied zur Milner-Typisierung (die Typausdrücke als ungetypt ablehnen kann, obwohl sie einen Typ haben, bzw. eingeschränktere Typen liefern kann) kann es sein, dass das die Berechnung nicht terminiert. Wir erzwingen die Terminierung, indem wir die Anzahl der Iterationen begrenzen. Falls diese Anzahl überschritten wird, liefern wir eine Fehlermeldung zurück, die ausdrückt, dass kein Typ hergeleitet werden konnte.

Bei der Berechnung der Typmarkierungen für einen Letrec-Ausdruck erhalten wir eine Typumgebung Γ, eine Liste v' mit Variablennamen, eine Liste $[x_1, \ldots, x_n]$ der linken Seiten der Letrec-Bindungen, eine Liste $[e_1, \ldots, e_n]$ der rechten Seiten, einen Ausdruck r und eine natürliche Zahl $maxIter$ als Eingabe, wobei n die Anzahl der Bindungen ist. $maxIter$ ist die Anzahl der Iterationen, die wir durchführen, bevor wir aufgeben, falls die Typannahmen für Letrec-gebundene Variablen sich nicht stabilisieren. Wir skizzieren das Verfahren erst und klären danach die wichtigen Details.

1. Erstelle eine Typannahme $\forall a.a$ für alle $x \in [x_1, \ldots, x_n]$, (die Namen dürfen alle gleich sein, die Instanziierung benennt alle gebundenen Typvariablen um), setze $\hat{\Gamma} = \{x_1 :: \forall a.a, \ldots, x_n :: \forall a.a\} \cup \Gamma$.

2. Berechne die Typen der Ausdrücke $e_1, \ldots, e_n$ und verwende dabei $\hat{\Gamma}$,

   i) falls ein Fehler auftritt $\longrightarrow$ terminiere mit einem Fehler,

   ii) sonst liegt eine Liste $[e'_1 :: T_1, \ldots, e'_n :: T_n]$, eine Typsubstitution φ (während der Berechnung der Typen von $e_1, \ldots, e_n$ entstehen Teilsubstitutionen,

die mittels $\circ$ zusammengefügt werden) und eine Liste mit übrig gebliebenen Variablennamen vor $\longrightarrow$ gehe zu Schritt 3.

3. Setze $ts = [S^{\varphi} \mid S \in [T_1, \ldots, T_n]]$, generalisiere alle Typen in ts, $ts^{\forall}$ sei die Liste der generalisierten Typen,

   i) falls die Typannahmen konsistent sind $\longrightarrow$ gehe zu Schritt 4, sonst $\longrightarrow$ gehe zu Schritt 3.ii),

   ii) falls die Anzahl der durchlaufenen Iterationen = $maxIter$ $\longrightarrow$ terminiere mit einem Fehler, sonst $\longrightarrow$ gehe zu Schritt 3.iii).

   iii) Füge neue Typannahmen ein: überschreibe die Typen in den alten Typannahmen in $\hat{\Gamma}$ mit den Typen aus $ts^{\forall}$ und gehe zu Schritt 2.

4. Setze $ts' = \{x_1 :: ts_1^{\forall}, \ldots, x_n :: ts_n^{\forall}\}$ (wobei $ts_i^{\forall}$ das i-te Element von $ts^{\forall}$ ist), setze $\Gamma' = ts' \cup \Gamma$ (füge alle Elemente aus ts' in Γ ein).

5. Berechne den Typ von r, verwende dabei Γ',

   i) falls ein Fehler auftritt $\longrightarrow$ terminiere mit einem Fehler,

   ii) sonst liegt ein Ausdruck r', eine Typsubstitution ψ und eine Liste w mit übrig gebliebenen Variablennamen vor $\longrightarrow$ markiere alle Bindungen mit (generalisierten) Typen, und liefere die Typsubstitution $\psi \circ \varphi$, den zusammengesetzten Letrec-Ausdruck und die Liste w zurück.

Im Schritt 2 des oben beschriebenen Verfahrens muss eine Liste $[e_1, \ldots, e_n]$ von Ausdrücken verarbeitet werden, so dass eine neue Liste mit typisierten Ausdrücken erzeugt wird. Es reicht nicht, wenn wir die Elemente der Liste getrennt voneinander betrachten. Vielmehr müssen die gewonnenen Informationen nach der Berechnung der Typmarkierungen für einen Ausdruck in weiteren Schritten weiterverwendet werden. Der Ausdruck

$$\lambda f.\texttt{letrec}\ a = f\ \texttt{True},\ b = f\ []\ \texttt{in}\ a$$

enthält, wie wir wissen, (mindestens) einen Typfehler. Die λ-gebundene Variable f darf im Funktionsrumpf nicht zwei verschiedene Typen an zwei verschiedenen Stellen haben. Damit die Typberechnung den Fehler entdecken kann, genügt es nicht $f :: a$ anzunehmen und die Unterausdrücke (f `True`) und (f []) jeweils für sich zu betrachten, denn dann wird für jeden Ausdruck ein scheinbar gültiger Typ hergeleitet. Aus diesem Grund gehen wir anders vor: wir wenden die in einem Schritt berechnete Typsubstitution auf alle Typen in der Typumgebung an und rechnen mit der veränderten Typumgebung weiter (genau wie bei der Berechnung des Typs einer Funktionsanwendung).

Wir erhalten eine Typsubstitution Γ, Liste $[e_1, \ldots, e_n]$ von Ausdrücken und eine Liste v mit Variablennamen als Eingabe.

1. Für i = $1, \ldots, n$:

   i) berechne den Typ von e_i, falls ein Typfehler auftritt $\longrightarrow$ terminiere mit einem Fehler, sonst liegt eine Typsubstitution φ_i, ein Ausdruck e_i' mit eingefügten Typmarkierungen und eine Liste v_i mit übrig gebliebenen Variablennamen vor $\longrightarrow$ gehe zu Schritt 1.ii),

   ii) setze $\Gamma = \Gamma^{\varphi_i}$ (wende die Typsubstitution φ_i auf alle Typen in Γ an).

2. Verknüpfe alle berechneten Substitutionen mittels $\circ$, setze $\psi = \varphi_n \circ \ldots \circ \varphi_1$.

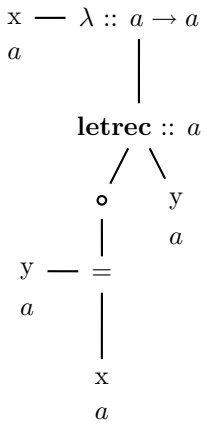
3. Liefere die Typsubstitution ψ, die Liste $[e_1', \ldots, e_n']$ und die Liste v_n mit übrig gebliebenen Variablennamen zurück.

Bei der Berechnung der Typen in einem Letrec-Ausdruck müssen die Typen der rechten Seiten der Bindungen in jeder Iteration generalisiert werden. Es werden Quantoren eingeführt, und Typvariablen werden gebunden. Dabei dürfen keine Typvariablen „eingefangen“ werden. Der Ausdruck

$$\lambda x.\texttt{letrec}\ y = x\ \texttt{in}\ y$$

enthält zum Beispiel keine Teilausdrücke, deren Typen einen Quantor enthalten dürfen. Das liegt zunächst daran, dass die Variable x durch ein λ gebunden wird und daher keinen quantifizierten Typ hat. Bei der Berechnung des Typs von x hinter dem Gleichheitszeichen darf die Typvariable a daher nicht gebunden werden. Die Abbildung 4.9.1 zeigt den Ausdruck mit eingefügten Typmarkierungen.

Abbildung 4.9.1:



Um keine Typvariablen „versehentlich einzufangen“ müssen wir beim Generalisieren wissen, welche Typvariablen nicht gebunden werden dürfen. Im Wesentlichen sind dies die Typvariablen, die in den Typen der freien Programmvariablen des betrachteten Unterausdrucks frei vorkommen [1]. Im Ausdruck ($\lambda x.\texttt{letrec}\ y = x\ \texttt{in}\ y$) kommt x rechts

vom Gleichheitszeichen frei vor und die Typvariable a in der Typannahme $(x :: a) \in \Gamma$ ist ebenfalls frei, wobei Γ die gegebene Typumgebung ist. Anders als es in den Typinferenzregeln in [1] definiert wird, betrachten wir beim Generalisieren nicht nur die Typen der freien Programmvariablen sondern einfach alle Typvariablen, die in Γ vorkommen. Wir stellen beide Regeln nochmal gegenüber.

$$\frac{\Gamma \vdash t :: T}{\Gamma \vdash t :: \forall \mathcal{X}.T} \quad \text{mit} \quad \begin{array}{rcl} \mathcal{X} & = & \mathcal{FTV}(T) \setminus \mathcal{Y} \\ \mathcal{Y} & = & \bigcup_{x \in \mathcal{FV}(t)} \mathcal{FTV}(\Gamma(x)) \end{array} \qquad \text{(Generalize)}$$

$$\frac{\Gamma \vdash t :: T}{\Gamma \vdash t :: \forall \mathcal{X}.T} \quad \text{mit} \quad \begin{array}{rcl} \mathcal{X} & = & \mathcal{FTV}(T) \setminus \mathcal{Y} \\ \mathcal{Y} & = & \bigcup_{S \in \mathrm{Cod}(\Gamma)} \mathcal{FTV}(S) \end{array} \qquad \text{(Generalize)}$$

Wenn wir uns innerhalb eines Letrec-Ausdrucks befinden, enthält die Typumgebung Typannahmen für genau die Programmvariablen, die weiter außen gebunden wurden und innerhalb des Letrec-Ausdrucks frei sind (nicht alle Variablen müssen im Letrec-Ausdruck vorkommen, doch wenn eine solche Variable vorkommt, kommt sie frei vor). Die Funktion

```
free :: Type -> Set.Set TName
free t = case t of
    (ForAll as q) ->
        vs_in q `Set.difference` Set.fromList as
    _ -> vs_in t
    where vs_in = Set.fromList . tvars_in
```

nimmt einen Typausdruck als Parameter entgegen und liefert die Menge der darin enthaltenen freien Typvariablen (wir verwenden das Haskell-Modul `Data.Set`). Die Funktion

```
free_te :: TypeEnv -> Set.Set TName
free_te gamma = Set.unions (map free ts)
    where ts = Map.elems gamma
```

erwartet eine Typumgebung als Parameter und liefert die Menge der Typvariablen, die frei in den Typen der gespeicherten Typannahmen vorkommen. Die Funktion

```
add_decls :: TypeEnv -> [TName] -> [Type]
             -> ([Type], Subst, [TName])
```

nimmt eine Typumgebung, eine Liste mit frischen Variablennamen und eine Liste mit Typen entgegen, und liefert eine Liste mit generalisierten Typen, eine Typsubstitution φ und die Liste der übrig gebliebenen Variablen zurück. Die neu gebundenen Variablen werden umbenannt, so dass wir beim Generalisieren Variablennamen verbrauchen. Die Typsubstitution φ speichert diese Umbenennungen.

Während der Typherleitung müssen wir in jedem Iterationsschritt alte Typannahmen mit neu hergeleiteten Typen vergleichen. Damit die Typannahmen als konsistent angesehen werden können, müssen die Typen nicht exakt gleich sein, es reicht wenn sie bis auf Umbenennungen gleich sind.

Definition 4.9.1. Seien T_1 und T_2 Typausdrücke. $T_1 \sim T_2$ gilt genau dann wenn beide Ausdrücke nach der Umbenennung der Typvariablen gleich sind, wobei zwei mal die gleiche Liste mit frischen Variablennamen verwendet wird.

Wir wollen genauer beschreiben, wie die Relation $\sim$ im Programm realisiert ist. Das Verfahren bekommt zwei Typausdrücke T_1 und T_2 als Eingabe und liefert `True` oder `False` zurück.

1. Falls einer der Beiden Typen die Form $(\forall \ldots)$ hat und der andere nicht (also keine gebundenen Variablen enthält) $\longrightarrow$ liefere `False` zurück.
2. Falls beide Typen nicht quantifiziert sind $\longrightarrow$ setze $T_1' = \forall\, [a_1, \ldots, a_n].T_1$ und $T_2' = \forall\, [b_1, \ldots, b_n].T_2$, wobei $a_1, \ldots, a_n$ die Typvariablen sind, die in T_1 vorkommen, und $b_1, \ldots, b_n$ die Typvariablen sind, die in T_2 vorkommen. Erzeuge neue Instanzen von T_1' und T_2' und verwende dabei jeweils die gleiche Namensliste, falls die Instanzen gleich sind $\longrightarrow$ liefere `True` zurück, sonst $\longrightarrow$ liefere `False` zurück.
3. Falls T_1 die Form $\forall[a_1, \ldots, a_n].S_1$ und T_2 die Form $\forall[b_1, \ldots, b_m].S_2$ hat $\longrightarrow$
   i) setze $S_1' = \forall\, [a_1', \ldots, a_n'].S_1$ und $S_2' = \forall\, [b_1', \ldots, b_n'].S_2$, wobei $a_1', \ldots, a_n'$ die Typvariablen sind, die in T_1 vorkommen, und $b_1', \ldots, b_n'$ die Typvariablen sind, die in S_2 vorkommen (verwende die Funktion `tvars_in`).
   ii) Erzeugen neue Instanzen S_1'', S_2'' von S_1' und S_2', verwende dabei jeweils die gleiche Namensliste $[c_1, \ldots]$.
   iii) Setze $as = [(a_1', c_1), \ldots, (a_n', c_n)]$ und $bs = [(b_1', c_1), \ldots, (b_n', c_n)]$, sei xs eine Liste mit Paaren und R^{xs} eine Operation, die angewendet auf einen Variablennamen x einen neuen Namen y liefert, falls $(x, y) \in xs$ (und einen Fehler sonst, wir nehmen an, dass für einen Typ $\forall a_1, \ldots, a_n.T$ gilt, dass alle Variablen $a_1, \ldots, a_n$ in T vorkommen).
   iv) Falls $\forall[R^{as}(a_1), \ldots, R^{as}(a_n)].S_1'' = \forall[R^{bs}(b_1), \ldots, R^{bs}(b_n)].S_1''$ (sortiere die Listen $[R^{as}(a_1), \ldots, R^{as}(a_n)]$, $[R^{bs}(b_1), \ldots, R^{bs}(b_n)]$ vor dem Vergleich) $\longrightarrow$ liefere `True` zurück, sonst $\longrightarrow$ liefere `False` zurück.

Die Funktion `(=~) :: Type -> Type -> Bool` im Modul `Typecheck.hs` nimmt zwei Typausdrücke T_1 und T_2 als Parameter entgegen und liefert `True`, falls $T_1 \sim T_2$, und `False` sonst.

Jetzt können wir präzisieren, wann das Verfahren (ohne Fehler) anhält: wir stoppen die Iteration, wenn

$$T_{i-1,1} \sim T_{i,1}, \ldots, T_{i-1,n} \sim T_{i,n}$$

Tabelle 4.9.1: Anwendungsbeispiele

T_1	T_2	$T_1 \sim T_2$
$\forall a.a$	a	`False`
$\forall b, t_1.t_1 \to b$	$\forall e, d.d \to e$	`True`
$\forall a, b.a \to b$	$\forall b, a.a \to b$	`True`
$\forall a, b.a \to b$	$\forall b, a.b \to a$	`True`
$\forall a, b, c.a \to [b] \to [[c]]$	$\forall a, b, c.c \to [b] \to [[a]]$	`True`
$\forall b, c.a \to [b] \to [[c]]$	$\forall a, b, c.c \to [b] \to [[a]]$	`False`
$\forall a.a \to [a] \to \mathit{Bool}$	$a \to [a] \to \mathit{Bool}$	`False`
$a \to b \to a$	$b \to a \to b$	`True`
$a \to b \to b$	$a \to a \to b$	`False`

gilt, wobei $x_1 :: T_{i-1,1}, \ldots, x_n :: T_{i-1,n}$ die Typannahmen aus dem vorhergehenden Iterationsschritt, $x_1 :: T_{i,1}, \ldots, x_n :: T_{i,n}$ die neuen Annahmen und $x_1, \ldots, x_n$ die Letrec-gebundenen Variablen sind.

Beispiel 4.9.1. Wir wollen nun ein Beispiel betrachten. Es sollen die Typen der Unterausdrücke in

$$\texttt{letrec}\ g = \lambda x.\overbrace{[] : (g\ (g\ []))}^{r}\ \texttt{in}\ g$$

mit dem iterativen Typherleitungsverfahren berechnet werden. Einfachheitshalber lassen wir u. a. Schlussstriche weg.

1. Als erstes nehmen wir an, dass eine Typumgebung Γ bereits einige Einträge enthält.

$$\Gamma = \{[] :: \forall a.[a],\ (:) :: \forall a.a \to [a] \to [a]\}$$

2. Dann fügen wir Typannahmen für g und für den Parameter x der Abstraktion in Γ ein (die veränderte Typumgebung heißt weiterhin Γ).

$$\Gamma = \{[] :: \forall a.[a],\ (:) :: \forall a.a \to [a] \to [a],\ g :: \forall a.a,\ x :: b\}$$

   Die Namen der gebundenen Typvariablen dürfen sich überschneiden. Beim Instanziieren werden diese Variablen umbenannt.

3. Wir berechnen den Typ von r (Rumpf der Abstraktion).

$$[] :: [c] \quad (\mathit{Inst.})$$
$$(:) :: d \to [d] \to [d] \quad (\mathit{Inst.})$$

Wir wenden Unifikation an, um den Typ von ((:) []) zu erhalten, dabei berechnen wir eine Typsubstitution, die angewendet auf e den Typ der Applikation liefert (und die Parametertypen gleich macht).

$$d \to [d] \to [d] \doteq [c] \to e$$
$$d \doteq [c],\ [d] \to [d] \doteq e$$
$$\varphi_1 = \{d \mapsto [c]\},\ e \doteq [d] \to [d]$$
$$\varphi_1 = \{d \mapsto [c]\},\ e \doteq [[c]] \to [[c]]$$
$$\varphi_1 = \{e \mapsto ([[c]] \to [[c]])\} \circ \{d \mapsto [c]\}$$
$$\varphi_1(e) = [[c]] \to [[c]]$$

Der Typ von ((:) []) ist somit $[[c]] \to [[c]]$. Um den Typ des zweiten Parameters zu bestimmen, müssen wir erst den Typ von $(g\ [])$ berechnen.

$$g :: f \quad (Inst.)$$
$$[] :: [h] \quad (Inst.)$$
$$f \doteq [h] \to i$$
$$\varphi_2 = \{f \mapsto ([h] \to i)\}$$

Die Funktion g hat an dieser Stelle also den Typ $[h] \to i$ und der Typ von $(g\ [])$ ist i. Die Typsubstitution φ_2 liefert nichts Neues für i. Jetzt können wir den Typ von $(g\ (g\ []))$ berechnen.

$$g :: j \quad (Inst.)$$
$$j \doteq i \to k$$
$$\varphi_3 = \{j \mapsto (i \to k)\} \circ \varphi_2$$

An dieser Stelle hat g den Typ $i \to k$ und der Typ von $(g\ (g\ []))$ ist k. Um den Typ der gesamten Konstruktoranwendung zu erhalten, müssen wir nochmal unifizieren.

$$[[c]] \to [[c]] \doteq k \to l$$
$$[[c]] \doteq k,\ [[c]] \doteq l$$
$$k \doteq [[c]],\ [[c]] \doteq l$$
$$\varphi_4 = \{k \mapsto [[c]]\} \circ \varphi_3 \circ \varphi_1,\ [[c]] \doteq l$$
$$\varphi_4 = \{k \mapsto [[c]]\} \circ \varphi_3 \circ \varphi_1,\ l \doteq [[c]]$$
$$\varphi_4 = \{l \mapsto [[c]]\} \circ \{k \mapsto [[c]]\} \circ \varphi_3 \circ \varphi_1$$

Der Typ des Unterausdrucks r ist somit $\varphi_4(l) = [[c]]$.

4. Die Variable x kommt im Rumpf der Abstraktion nicht vor; dadurch haben wir keinen spezielleren Typ dafür erhalten. Der Typ der Abstraktion ist somit $b \to [[c]]$.

5. Wir generalisieren den Typ $b \to [[c]]$ (führen einen Quantor ein und benennen die Typvariablen um) und erhalten eine neue Typannahme

$$g :: \forall a', b'.a' \to [[b']]$$

für g. Das ist der Typ an der Letrec-Position. Die Funktion g darf in r keinen quantifizierten Typ haben.

Die alte und die neue Typannahme für g sind nicht gleich. Um zu überprüfen, ob der hergeleitete Typ von g endgültig ist und sich nicht weiter verändert, müssen wir den Typ mit der neuen Typannahme nochmal herleiten.

1. Wir starten mit der gleichen Typumgebung Γ wie im vorigen Iterationsschritt und

2. fügen die neue Typannahme für g und eine Typannahme für x ein.

$$\Gamma = \{[] :: \forall a.[a],\ (:) :: \forall a.a \to [a] \to [a],\ g :: \forall a', b'.a' \to [[b']],\ x :: c'\}$$

3. Der Typ von ((:) []) verändert sich nicht, wir benennen nur die Typvariable um und erhalten $[[d']] \to [[d']]$. Als Nächstes ist der Unterausdruck $(g\ [])$ an der Reihe.

$$\begin{gathered}
g :: a'' \to [[b'']] \quad (Inst.) \\
[] :: [c''] \quad (Inst.) \\
a'' \to [[b'']] \doteq [c''] \to d'' \\
a'' \doteq [[c'']], [[b'']] \doteq d'' \\
\varphi_1 = \{a'' \mapsto [c'']\},\ [[b'']] \doteq d'' \\
\varphi_1 = \{a'' \mapsto [c'']\},\ d'' \doteq [[b'']] \\
\varphi_1 = \{d'' \mapsto [[b'']]\} \circ \{a'' \mapsto [c'']\}
\end{gathered}$$

Wenden wir die Typsubstitution φ_1 auf d'' an, erfahren wir, dass $[[b'']]$ der Typ von $(g\ [])$ ist. Wir instanziieren und unifizieren weiter, um den den Typ von $(g\ (g\ []))$ zu bestimmen.

$$\begin{gathered}
g :: \bar{a} \to [[\bar{b}]] \quad (Inst.) \\
\bar{a} \to [[\bar{b}]] \doteq [[b'']] \to \bar{c} \\
\bar{a} \doteq [[b'']],\ [[\bar{b}]] \doteq \bar{c} \\
\varphi_2 = \{\bar{a} \mapsto [[b'']]\} \circ \varphi_1,\ [[\bar{b}]] \doteq \bar{c} \\
\varphi_2 = \{\bar{a} \mapsto [[b'']]\} \circ \varphi_1,\ \bar{c} \doteq [[\bar{b}]] \\
\varphi_2 = \{\bar{c} \mapsto [[\bar{b}]]\} \circ \{\bar{a} \mapsto [[b'']]\} \circ \varphi_1
\end{gathered}$$

Die Anwendung von φ_2 auf $\bar{c}$ liefert $[[\bar{b}]]$, womit das der Typ von $(g\ (g\ []))$ ist. Schließlich können wir den Typ des Abstraktionsrumpfes berechnen.

$$\begin{gathered}
[[d']] \to [[d']] \doteq [[\bar{b}]] \to \bar{d} \\
[[d']] \doteq [[\bar{b}]],\ [[d']] \doteq \bar{d} \\
[d'] \doteq [\bar{b}],\ [[d']] \doteq \bar{d} \\
d' \doteq \bar{b},\ [[d']] \doteq \bar{d} \\
\varphi_3 = \{d' \mapsto \bar{b}\} \circ \varphi_2,\ [[d']] \doteq \bar{d} \\
\varphi_3 = \{d' \mapsto \bar{b}\} \circ \varphi_2,\ \bar{d} \doteq [[d']] \\
\varphi_3 = \{\bar{d} \mapsto [[d']]\} \circ \{d' \mapsto \bar{b}\} \circ \varphi_2
\end{gathered}$$

Wir wenden die berechnete Typsubstitution φ_3 auf $\bar{d}$ an (die Typsubstitution hat einen kleineren Index als bei der ersten Iteration, weil wir jetzt am Anfang den Typ von $((:)\ [])$ nicht neu berechnet und eine Substitution weggelassen haben) und erhalten $([] : (g\ (g\ []))) :: [[d']]$.

4. Der Typ der Abstraktion ist dann $c' \to [[d']]$, und wir sehen schon, dass wir bei diesem Typ nichts Neues hergeleitet haben (nur die Variablennamen sind jetzt anders).

5. Die Alleinführung[5] liefert $\forall t_1, t_2.t_1 \to [[t_2]]$ als neue Typannahme für g. Nach dem Umbenennen der Typvariablen, ist die alte und die neue Typannahme gleich, so dass die Iteration hier zu Ende ist.

Um den Typ von g zu berechnen sind also zwei Iterationen nötig. Die Abbildung 4.9.2 zeigt den Ausdruck ($\texttt{letrec}\ g = \lambda x.[] : (g\ (g\ []))\ \texttt{in}\ g$) mit eingefügten Typmarkierungen nachdem alle Typvariablen umbenannt wurden.

Beispiel 4.9.2. Eine abgeänderte Version ($\texttt{letrec}\ g = \lambda x \to [] : (g\ (g\ \texttt{True}))\ \texttt{in}\ g$) des Ausdrucks aus dem vorigen Beispiel ist mit dem Typherleitungssystem von Haskell nicht typisierbar (iterativ aber schon). In Abbildung 4.9.3 ist dieser Ausdruck mit eingefügten Typmarkierungen dargestellt.

Beispiel 4.9.3. Dieses Beispiel zeigt einen simpleren Fall. Es werden trotzdem zwei Iterationen benötigt. Um den Typ von g in

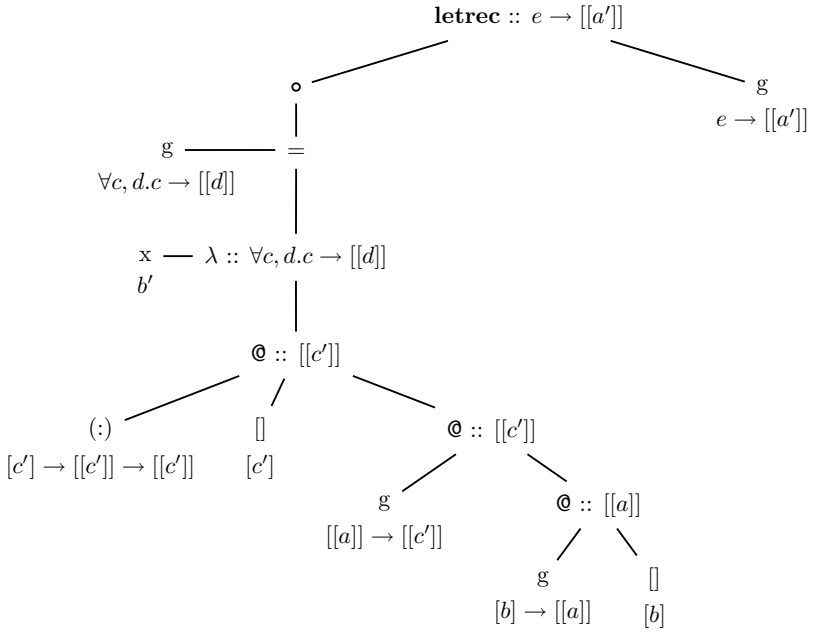
$$\lambda x.\texttt{letrec}\ g = (\lambda y.y)\ x\ \texttt{in}\ \texttt{True}$$

zu berechnen, fügen wir als erstes die Typannahmen $x :: a$, $g :: \forall a.a$ in eine Typumgebung Γ ein.

1. Der Typ von $(\lambda y.y)$ ist $b \to b$.

[5] Generalisierung

Abbildung 4.9.2:



2. Wir unifizieren, um den Typ Typ von $((\lambda y.y)\ x)$ zu bestimmen (diesmal werden die Substitutionskomponenten in einer anderen Reihenfolge erstellt, was aber nichts an der gesamten Vorgehensweise ändert).

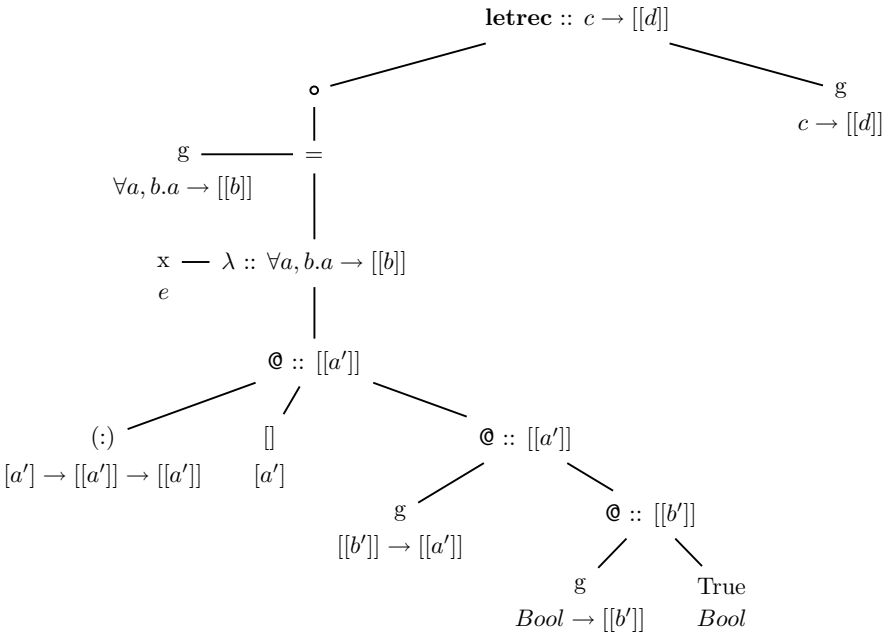
$$
\begin{gathered}
b \to b \doteq a \to c \\
b \doteq a,\ b \doteq c \\
\varphi = \{b \mapsto c\},\ b \doteq a \\
\varphi = \{b \mapsto c\},\ c \doteq a \\
\varphi = \{c \mapsto a\} \circ \{b \mapsto c\}
\end{gathered}
$$

   Der Typ von $(\lambda y.y)$ ist $(a \to a)$, nachdem die Typsubstitution angewendet wurde. Der Typ der Applikation ist $\varphi(c) = a$.

3. Wir wenden die Typsubstitution φ auf alle Typen in Γ an und erhalten $x :: a$ (der Typ ändert sich nicht), so dass die Typvariable a frei ist und wir keinen Quantor im Typ von g einführen dürfen. Die Typen $\forall a.a$ und a sind zwar ähnlich aber dennoch ungleich, so dass eine weitere Iteration durchgeführt werden muss. Danach ist die Annahme und der neu hergeleitete Typ von g gleich.

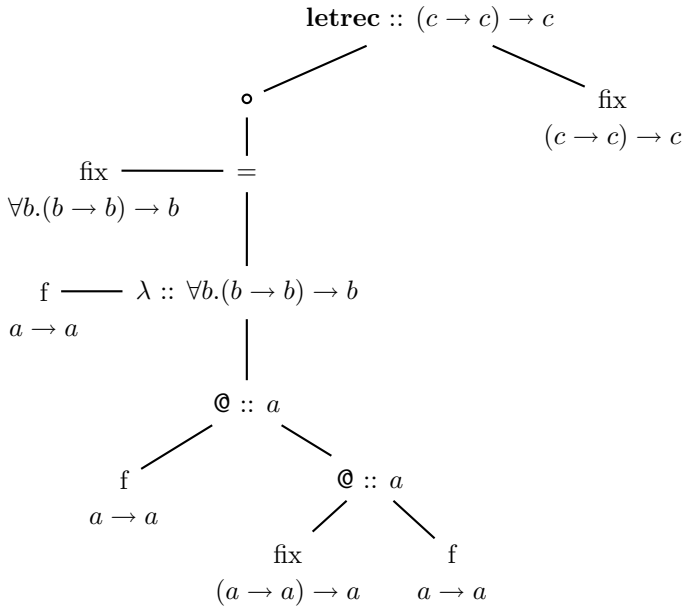
Beispiel 4.9.4. Gibt es Ausdrücke, die mehr als zwei Iterationen benötigen? In Abbildung 4.9.4 ist der Ausdruck (`letrec` $fix = \lambda f.f\ (fix\ f)$ `in` fix) mit eingefügten

Abbildung 4.9.3:



Typmarkierungen dargestellt. Um den Typ von fix herzuleiten werden drei Iterationen benötigt.

Abbildung 4.9.4:



Beispiel 4.9.5. Wir wollen ein letztes Beispiel betrachten. Die Funktion *concat* nimmt eine Liste von Listen entgegen und konkateniert die enthaltenen Listen.

Listing 4.1: *concat.plc*

```
-- Haskell:
--
-- (++) :: [a] -> [a] -> [a]
-- []     ++ ys = ys
-- (x:xs) ++ ys = x : (xs ++ ys)
--
-- foldr :: (a -> b -> b) -> b -> [a] -> b
-- foldr f z []     = z
-- foldr f z (x:xs) = f x (foldr f z xs)
--
-- concat :: [[a]] -> [a]
-- concat xss = foldr (++) [] xss

letrec
    concat' = \xs -> \ys ->
        case_List xs of {[] -> ys; z:zs -> z:concat' zs ys},

    foldr = \f -> \z -> \xs ->
```

```
        case_List xs of {[] -> z; y:ys -> f y (foldr f z ys)},
    concat = \xss -> foldr concat' [] xss
    in concat
```

Listing 4.2: *concat.lplc*

```
(letrec
    concat'::forall c' . [c'] -> [c'] -> [c'] =
        (\xs::[d] . (\ys::[d] . case_List xs::[d] of {
            ([]::[d]) -> ys::[d]
          , (((:)::d -> [d] -> [d] z::d zs::[d]) :: [d]) ->
            ((:)::d -> [d] -> [d] z::d
                ((concat'::[d] -> [d] -> [d] zs::[d]
                    ) :: [d] -> [d] ys::[d]) :: [d]) :: [d]
        } :: [d]) :: [d] -> [d]) :: forall c' . [c'] -> [c'] -> [c']
  , foldr::forall a' b' . (b' -> a' -> a') -> a' -> [b'] -> a' =
        (\f::c -> b -> b . (\z::b . (\xs::[c] . case_List xs::[c] of {
            ([]::[c]) -> z::b
          , (((:)::c -> [c] -> [c] y::c ys::[c]) :: [c]) ->
                ((f::c -> b -> b y::c) :: b -> b ((
                    (foldr::(c -> b -> b) -> b -> [c] -> b f::c -> b -> b
                        ) :: b -> [c] -> b z::b
                ) :: [c] -> b ys::[c]) :: b) :: b
        } :: b) :: [c] -> b) :: b -> [c] -> b
            ) :: forall a' b' . (b' -> a' -> a') -> a' -> [b'] -> a'
  , concat::forall e . [[e]] -> [e] =
        (\xss::[[a]] . (((foldr::([a] -> [a] -> [a]) -> [a] -> [[a]] -> [a]
            concat'::[a] -> [a] -> [a]) :: [a] -> [[a]] -> [a] []::[a]
                ) :: [[a]] -> [a] xss::[[a]]) :: [a]
            ) :: forall e . [[e]] -> [e]
    in concat::[[d']] -> [d']
) :: [[d']] -> [d']
```

Literaturverzeichnis

[1] M. Schmidt-Schauß, D. Sabel, F. Harwath (2009) *Contextual Equivalence in Lambda-Calculi extended with letrec and with a Parametric Polymorphic Type System*, Technical Report Frank-36

[2] I. Angelelli (1967) *On Identity and Interchangeability in Leibniz and Frege*, Notre Dame Journal of Formal Logic, Vol. VIII

[3] F. Bader, W. Snyder (2001) *Unification Theory*, Handbook of Automated Reasoning, Elsevier

[4] H. P. Barendregt (1984) *The Lambda Calculus: Its Syntax and Semantics, revised edition*, North-Holland

[5] R. Bird, P. Wadler (1988) *Introduction to Functional Programming*, Prentice Hall

[6] R. Bird (1998) *Introduction to Functional Programming using Haskell, second edition*

[7] W. Blum, C.-H. L. Ong (2007) *The Safe Lambda Calculus*, TLCA '07

[8] A. Davie (1992) *An Introduction to Functional Programming Systems Using Haskell*, Cambridge University Press

[9] A. Diller (1989) *Compiling Functional Languages*, John Wiley & Sons

[10] G. Fischer (2002) *Lineare Algebra, 13. Auflage*, Viehweg

[11] G. Frege (1884) *Die Grundlagen der Arithmetik – Eine logisch mathematische Untersuchung über den Begriff der Zahl*, Meiner (1988)

[12] G. Frege [1893/1903] *Grundgesetze der Arithmetik, begrifflich abgeleitet*, Olms (1998)

[13] ISO/IEC (1996) *Information technology – Syntactic metalanguage – Extended BNF*, ISO-Standard 14977

[14] F. Henglein (1988) *Type inference and semi-unification*

[15] K. Jänich (2002) *Lineare Algebra, 9. Auflage*, Springer

[16] D. E. Knuth (1973) *The Art of Computer Programming, Vol. 3 / Sorting and Searching*, Addison-Wesley

[17] J. McCarthy (1963) *A Basis for a Mathematical Theory of Computation*, Computer Programming and Formal Systems, p. 33-70. North-Holland

[18] S. L. Peyton Jones, D. Lester (1991) *A modular fully-lazy lambda lifter in Haskell*, Software Practice and Experience 21(5)

[19] S. L. Peyton Jones (1987) *The Implementation of Functional Programming Languages*, Prentice Hall

[20] J. A. Robinson (1965) *A machine-oriented logic based on the resolution principle*, Journal of the ACM, Vol. 12, No. 1, 23-41

[21] I. Wegener (1999) *Theoretische Informatik – eine algorithmenorientierte Einführung, 2. Auflage*, Teubner